从 0 开始移植 μC/OS-II 到野火 stm32 开发板

作者	fire 野火
E-Mail	firestm32@foxmail.com
QQ	313303034
博客	firestm32.blog.chinaunix.net
硬件平台	野火 STM32 开发板
库版本	ST3.0.0
系统版本	μc/OS-II V2.86

目录

前言

uC/OS 是一个微型的实时操作系统，包括了一个操作系统最基本的一些特性，如任务调度、任务通信、内存管理、中断管理、定时管理等。而且这是一个代码完全开放的实时操作系统，简单明了的结构和严谨的代码风格，非常适合初涉嵌入式操作系统的人士学习。

很多人在学习 STM32 中，都想亲自移植一下 uC/OS，而不是总是用别人已经移植好的。在我学习 uC/OS 的过程中，查找了很多资料，也看过很多关于如何移植 uC/OS 到 STM32 处理器上的教程，但都不尽人意，主要是写得太随意了，思路很乱，读者看到最后还是不确定该怎样移植。为此，我决定写这个教程，让广大读者真正了解怎样移植。

学前建议：C 语言 + 数据结构

野火嵌入式开发工作室
2011 年 10 月 1 日

官方源代码介绍

首先我们下载源代码，官方下载地址：
http://micrium.com/page/downloads/ports/st/stm32 （下载资料需要注册帐号）

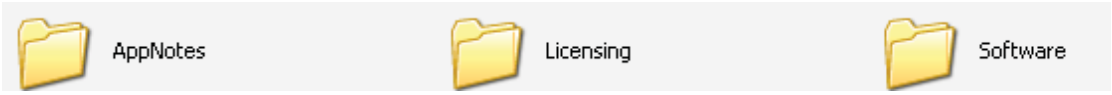
或者网盘下载：http://dl.dbank.com/c0jnhmfxcp

我们需要下载的就是下面这个，因为我用到的开发板芯片是 STM32F103VET6

注意：下载的源代码开发环境是 IAR 编译器的，我们用的是 MDK 的，这点不要搞混了！

我们使用的 uCOS 是 2.86 版本。

下载解压后可以看到 Micrium 含有三个文件夹：



文件名	说明			
AppNotes	包含 uCOS-II 的说明文件，其中文件 **Micrium\AppNotes\AN1xxx-RTOS\AN1018-uCOS-II-Cortex-M3\AN-1018.pdf 是很重要的**。这个文件对 uC/OS 在 M3 内核移植过程中需要修改的代码做了详细的说明。			
Licensing	包含了 uCOS-II 使用许可证			
Software	应用软件，我们这里用到的就是 uCOS-II 文件夹。在整个移植过程中我们只需用到 uCOS-II 下的两个文件，分别是 Ports 和 Source.			
	uCOS-II	**Doc**	uC/OS 官方自带说明文档和教程	
		Ports	官方移植到 M3 的移植文件（IAR 工程）	
			cpu.h	定义数据类型、处理器相关代码、声明函数原型
			cpu_c.c	定义用户钩子函数，提供扩充软件功能的入口点。（所谓钩子函数，就是指那些插入到某函数中拓展这些函数功能的函数）
			cpu_a.asm	与处理器相关汇编函数，主要是任务切换函数
			os_dbg.c	内核调试数据和函数
		Source	uC/OS 的源代码文件	
			ucos_ii.h	内部函数参数设置
			os_core.c	内核结构管理，uC/OS 的核心，包含了内核初始化，任务切换，事件块管理、事件标志组管理等功能。
			os_time.c	时间管理，主要是延时
			os_tmr.c	定时器管理，设置定时时间，时间到了就进行一次回调函数处理。
			os_task.c	任务管理
			os_mem.c	内存管理
			os_sem.c	信号量
			os_mutex.c	互斥信号量
			os_mbox.c	消息邮箱
			os_q.c	队列

			os_flag.c	事件标志组
	CPU	STM32 标准外设库		
	EvalBoards	micrium 官方评估板的代码		
		OS-Probe-LCD	**os_cfg.h**	内核配置
	uC-CPU	基于 micrium 官方评估板的 CPU 移植代码		
	uC-LIB	micrium 官方的一个库代码		
	uC-Probe	uC-Probe 有关的代码，是一个通用工具，能让嵌入式开发人员在实时环境中监测嵌入式系统。		

以上这些都是下载下来的官方资源。有没有发现，uC/OS 的代码文件都被分开放到不同的文件夹里了？呵呵，这个是官方移植好到 STM32 的 uC/OS 系统，他已经帮我们对 uC/OS 的文件进行分类存放。如果你不想要移植好的，也可以下载没有移植的，那样就所以文件都放在一个文件夹里。

下载地址：http://micrium.com/download/Micrium-uCOS-II-V290.ZIP

在自己亲自移植之前，总是看到移植好的例程包含有 CPU、uC-CPU、uC-LIB、uCOS-II 四个文件夹下的代码。uCOS-II 文件夹下的是源代码，这个好理解；但是前面三个有什么用啊？

通常看其他移植教程时，一般都说只需改 os_cpu.h，os_cpu_a.asm 和 os_cpu_c.c 就可以了，就没听说过有 CPU、uC-CPU、uC-LIB 这些的。心中一直很纳闷，难道后三个都要自己编写的吗？后来在上面网址把源代码下载后，才知道 CPU、uC-CPU、uC-LIB 这三个文件是官方自己写的移植文件，而我们使用了标准外设库 CMSIS 中提供的启动文件及固件库了，因此可以不用这三个文件，哈哈，心中的疑团解决了！

先看一下开发板与 uC/OS-II 的框架图（注意 APP.C 就是 main 文件，我们下面移植的文件并没有 APP_VECT.C 这个文件，应用文件可以灵活处理的）

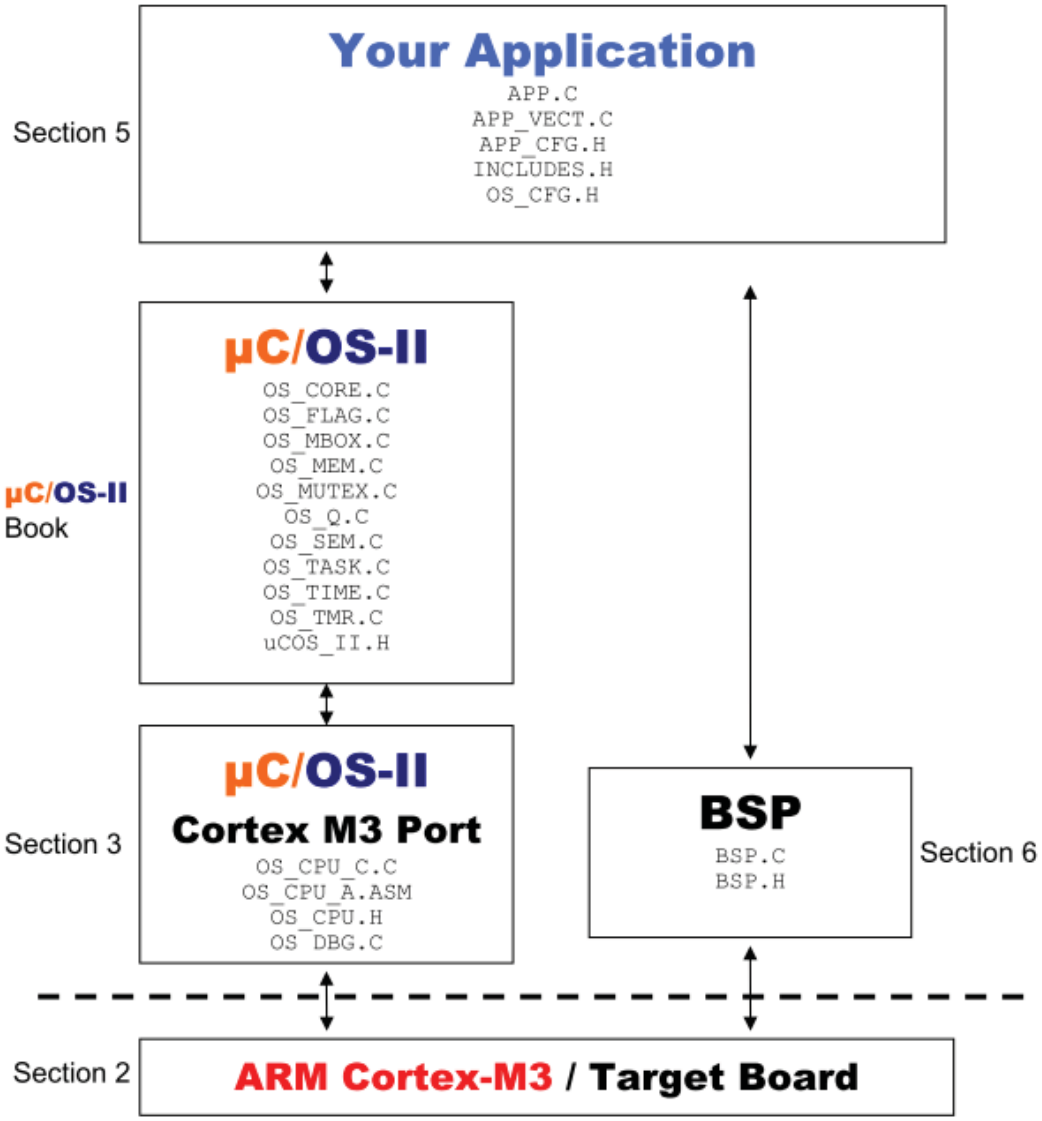


Figure 1-1, Relationship between modules.

重要文件代码详解

移植前，我们需要先了解一下 uC/OS 的重要文件代码。

对于从没接触过 uC/OS 或者其他嵌入式系统的朋友们，你们需要先了解 uC/OS 的工作原理和各模块功能，不然就不知道为啥这样移植。

推荐教程

作者	书名	推荐理由
野火团队	初探 uCOS-II（是一个文档教程）	清晰简单地讲解了 uC/OS 的运行流程，方便初学者学习。
任哲	嵌入式实时操作系统	通俗易懂的一本 uC/OS 教程，非

	uC/OS-II 原理及应用 （北京航空航天出版社）	常适合初学者学习。 不过教程没得到更新，不能适应 uC/OS 的发展，但还是非常值得推荐。
Joseph Yiu 著 宋岩 译	Cortex-M3 权威指南	呵呵，不用说吧？移植 uC/OS 到 M3 内核中，怎么能不了解内核呢？

下面的内容主要来自于刚才下载的文件里面的 Micrium\AppNotes\AN1xxx-RTOS\AN1018-uCOS-II-Cortex-M3\AN-1018.pdf 文件来讲的，因为这文件是 uC/OS 作者移植 uC/OS 到 STM32 的移植手册，里面谈到很多移植说需要注意的事项和相关知识。我在这里添加也按照作者的思路来讲解，并加入个人理解，如果有误，欢迎斧正，新手交流，能者指教。

os_cpu.h

定义数据类型、处理器相关代码、声明函数原型

全局变量

OS_CPU_GLOBALS 和 OS_CPU_EXT 允许我们是否使用全局变量。

```
1.  #ifdef  OS_CPU_GLOBALS
2.  #define OS_CPU_EXT
3.  #else                           //如果没有定义 OS_CPU_GLOBALS
4.  #define OS_CPU_EXT  extern      //则用 OS_CPU_EXT 声明变量已经外部定义了。
5.  #endif
```

数据类型

```
6.  typedef unsigned char  BOOLEAN;
7.  typedef unsigned char  INT8U;
8.  typedef signed   char  INT8S;
9.  typedef unsigned short INT16U;      //大多数 Cortex-M3 编译器，short 是 16 位,int 是 32 位
10. typedef signed   short INT16S;
11. typedef unsigned int   INT32U;
12. typedef signed   int   INT32S;
13. typedef float          FP32;        //尽管包含了浮点数，但 uC/OS-II 中并没用到
14. typedef double         FP64;
15. typedef unsigned int   OS_STK;      //M3 是 32 位，所以堆栈的数据类型 OS_STK 设置 32 位
16. typedef unsigned int   OS_CPU_SR;   //M3 的状态寄存器（xPSR）是 32 位
```

临界段

临界段，就是不可被中断的代码段，例如常见的入栈出栈等操作就不可被中断。

uC/OS-II 是一个实时内核，需要关闭中断进入和开中断退出临界段。为此，uC/OS-II 定义了两个宏定义来关中断 OS_ENTER_CRITICAL()和开中断 OS_EXIT_CRITICAL()。

```
17. #define  OS_CRITICAL_METHOD    3  //进入临界段的三种模式，一般选择第 3 种，即这里设置为 3
18. #define  OS_ENTER_CRITICAL()   {cpu_sr = OS_CPU_SR_Save();} //进入临界段
19. #define  OS_EXIT_CRITICAL()    {OS_CPU_SR_Restore(cpu_sr);} //退出临界段
```

事实上，有 3 种开关中断的方法，根据不同的处理器选用不同的方法。大部分情况下，选用第 3 种方法。

另外，关于汇编函数 OS_CPU_SR_Save() 和 OS_CPU_SR_Restore() ，在后面谈到 os_cpu_a.asm 文件时会再说。

栈生长方向

M3 的栈生长方向是由高地址向低地址增长的，因此 OS_STK_GROWTH 定义为 1。

```
20. #define  OS_STK_GROWTH        1
```

任务切换宏

定义任务切换宏，关于汇编函数 OSCtxSw()，在后面谈到 os_cpu_a.asm 文件时会再说。

```
21. #define  OS_TASK_SW()         OSCtxSw()
```

函数原型

开中断和关中断

如果定义了进入临界段的模式为 3，就声明开中断和关中断函数

```
22. #if OS_CRITICAL_METHOD == 3
23. OS_CPU_SR  OS_CPU_SR_Save(void);
24. void       OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
25. #endif
```

任务管理函数

```
26. /*****************任务切换的函数*****************/
27. void       OSCtxSw(void);                    //用户任务切换
28. void       OSIntCtxSw(void);                 //中断任务切换函数
29. void       OSStartHighRdy(void);             //在操作系统第一次启动的时候调用的任务切换
30.
31. void       OS_CPU_PendSVHandler(void);       //用户中断处理函数，旧版本为 OSPendSV
32.
33. void       OS_CPU_SysTickHandler(void);      //系统定时中断处理函数，时钟节拍函数
34. void       OS_CPU_SysTickInit(void);         //系统 SysTick 定时器初始化
35.
36. INT32U     OS_CPU_SysTickClkFreq(void);      //返回 SysTick 定时器的时钟频率
```

这三个函数是为 SysTick 定时器服务的

关于任务切换，要用到异常处理知识，可以看《Cortex-M3 权威指南》（Joseph Yiu 著 宋岩译）中第 3.4 小节。

关于 PendSV，有不懂的朋友，可以看《Cortex-M3 权威指南》中第 7.6 小节 SVC 和 PendSV：

SVC（系统服务调用，亦简称系统调用）和 PendSV（可悬起系统调用），它们多用在上了操作系统的软件开发中。

SVC 用于产生系统函数的调用请求，SVC 异常是必须在执行 SVC 指令后立即得到响应的。PendSV（可悬起的系统调用）则不同，它是可以像普通的中断一样被悬起的（不像 SVC 那样会上访）。OS 可以利用它“缓期执行”一个异常——直到其它重要的任务完成后才执行动作。悬起 PendSV 的方法是：手工往 NVIC 的 PendSV 悬起寄存器中写 1。悬起后，如果优先级不够高，则将缓期等待执行。

PendSV 的典型使用场合是在上下文切换时（在不同任务之间切换）。例如，一个系统中有两个就绪的任务，上下文切换被触发的场合可以是：

- 执行一个系统调用
- 系统滴答定时器（SysTick）中断，（轮转调度中需要）

注：此部分内容出自《Cortex-M3 权威指南》

关于 SysTick 定时器的三个函数，为了便于理解，我们把它注释掉，不采用官方的，自己编写：

需要注释的函数

OS_CPU_SysTickHandler()	在 os_cpu_c.c 中定义，是 SysTick 中断的中断处理函数，而在 stm32f10x_it.c 中已经有该中断函数的定义 SysTick_Handler()，这里也就不需要了。
OS_CPU_SysTickClkFreq()	定义在 BSP.C 中，此函数我们自己会编写，把它注释掉。
OS_CPU_SysTickInit()	定义在 os_cpu_c.c 中，用于初始化 SysTick 定时器，它依赖于 OS_CPU_SysTickClkFreq()，也要注释掉。

os_cpu_c.c

移植 uC/OS 时，我们需要写 10 个相当简单的 C 函数：9 个钩子函数和 1 个任务堆栈结构初始化函数。

钩子函数

所谓钩子函数，指那些插入到某些函数中为扩展这些函数功能的函数。一般地，钩子函数为第三方软件开发人员提供扩充软件功能的入口点。为了拓展系统功能，uC/OS-II 中提供有大量的钩子函数，用户不需要修改 uC/OS-II 内核代码程序，而只需要向钩子函数添加代码就可以扩充 uC/OS-II 的功能。

注：此部分内容出自 张勇的《嵌入式操作系统原理与面向任务程序设计——基于 uC/OS-II v2.86 和 ARM920T》

尽管 uC/OS-II 中提供了大量的钩子函数，但实际上，移植时我们需要编写的也就 9 个钩子函数：

```
OSInitHookBegin()        //OSIinit()          系统初始化函数开头的钩子函数
OSInitHookEnd()          //OSIinit()          系统初始化函数结尾的钩子函数
OSTaskCreateHook()       //OSTaskCreate()或 OSTaskCreateExt()    创建任务钩子函数
OSTaskDelHook()          //OSTaskDel()        删除任务钩子函数
OSTaskIdleHook()         //OS_TaskIdle()      空闲任务钩子函数
OSTaskStatHook()         //OSTaskStat()       统计任务钩子函数
OSTaskSwHook()           //OSTaskSW()         任务切换钩子函数
OSTCBInitHook()          //OS_TCBInit()       任务控制块初始化钩子函数
OSTimeTickHook()         //OSTaskTick()       时钟节拍钩子函数
```

这些函数都是一些钩子函数，一般由用户拓展。如果要用到这些钩子函数，需要在 OS_CFG.H 中定义 OS_CPU_HOOKS_EN 为 1，即：

```
#define  OS_CPU_HOOKS_EN   1  //在 OS_CFG.H 中定义
```

钩子函数的编写，例如：

```
/*** 系统初始化函数 OSInit()  开头调用   ***/
void OSInitHookBegin (void)
{
#if OS_TMR_EN > 0        //当使用 OS_TMR.C 定时器管理模块
    OSTmrCtr = 0;        //初始化系统节拍计数变量 OSTmrCtr 为 0
                         //每个时钟节拍 OSTmrCtr（全局变量，初始值为 0）增 1
#endif
}


/***  创建任务 OSTaskCreate()  或  OSTaskCreateExt()中调用   ***/
void OSTaskCreateHook (OS_TCB *ptcb)
{
#if OS_APP_HOOKS_EN > 0             //如果有定义应用任务
    App_TaskCreateHook(ptcb);       //调用应用任务创建钩子函数
#else                               //否则
    (void)ptcb;                     //告诉编译器 ptcb 没用到
#endif
}


/***  切换任务时被调用   ***/
void  OSTaskSwHook (void)
```

```
{
#if OS_APP_HOOKS_EN > 0
    App_TaskSwHook();                   //应用任务切换时调用的钩子函数
#endif
}


/***  每个系统节拍到了   ***/
void  OSTimeTickHook (void)
{
#if OS_APP_HOOKS_EN > 0
    App_TimeTickHook();                 //应用软件的时钟节拍钩子
#endif

#if OS_TMR_EN > 0                       //如果有启动定时器管理
    OSTmrCtr++;                         //计时变量 OSTmrCtr 加 1
    if (OSTmrCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC)) {//如果时间到了
        OSTmrCtr = 0;   //计时清 0
        OSTmrSignal(); //发送信号量 OSTmrSemSignal（初始值为 0）
                       //以便软件定时器扫描任务 OSTmr_Task 能请求到信号量而继续运行下去
    }
#endif
}
```

这些钩子函数是必须声明的，但不是必须定义的，只是为了拓展你的系统功能而已。

任务堆栈结构初始化函数

```
OSTaskStkInit()          //任务堆栈结构初始化函数
```

通常，我们的任务定义都是这样的：

```
void  MyTask (void *p_arg)
{
    /* 可选，例如处理 'p_arg'变量 */
    while (1) {
        /* 任务主体 */
    }
}
```

典型的 ARM 编译器（Cortex-M3 也是这样）都会把这个函数的第一个参量传递到 R0 寄存器中。

对于像 ARM 内核一般都比较多寄存器的单片机，我们可以把函数中断的局部变量保存在寄存器中，以加快速度。

```
101.OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg,
102.                                                    OS_STK *ptos, INT16U opt)
103.{
104.    OS_STK *stk;
105.
106.
107.    (void)opt;                          // 'opt' 并没有用到，防止编译器提示警告
108.    stk       = ptos;                   // 加载栈指针
109.
110.    /* 中断后 xPSR，PC，LR，R12，R3-R0 被自动保存到栈中*/
111.    *(stk)    = (INT32U)0x01000000L; // xPSR
112.    *(--stk)  = (INT32U)task;         // 任务入口
113.    *(--stk)  = (INT32U)0xFFFFFFFEL; // R14 (LR)
114.    *(--stk)  = (INT32U)0x12121212L; // R12
115.    *(--stk)  = (INT32U)0x03030303L; // R3
116.    *(--stk)  = (INT32U)0x02020202L; // R2
117.    *(--stk)  = (INT32U)0x01010101L; // R1
118.    *(--stk)  = (INT32U)p_arg;        // R0 : 变量
119.
120.    /* 剩下的寄存器需要手动保存在堆栈     */
121.    *(--stk)  = (INT32U)0x11111111L; // R11
122.    *(--stk)  = (INT32U)0x10101010L; // R10
123.    *(--stk)  = (INT32U)0x09090909L; // R9
124.    *(--stk)  = (INT32U)0x08080808L; // R8
125.    *(--stk)  = (INT32U)0x07070707L; // R7
126.    *(--stk)  = (INT32U)0x06060606L; // R6
127.    *(--stk)  = (INT32U)0x05050505L; // R5
128.    *(--stk)  = (INT32U)0x04040404L; // R4
129.
130.    return (stk);
131.}
```

这是初始化任务堆栈函数。OSTaskStkInit()被任务创建函数调用，所以要在开始时，在栈中作出该任务好像刚被中断一样的假象。

在 ARM 内核中，函数中断后，xPSR，PC，LR，R12，R3-R0 被自动保存到栈中的，R11-R4 如果需要保存，只能手工保存。为了模拟被中断后的假象，OSTaskStkInit()的工作就是在任务自己的栈中保存 cpu 的所有寄存器。这些值里 R1-R12 都没什么意义，这里用相应的数字代号（如 R1 0x01010101）主要是方便调试。

野火在这里问大家两个问题，看看大家是否掌握了这个知识点：

为什么程序是 *(--stk) = (INT32U)******;而不是保存寄存器的值：*(--stk) = *(INT32U)*******呢？

答案很简单，就是上面说的，任务还没开始运行，栈里保存的 R1-R12 值都没什么意义的，这里仅仅是模拟中断那样的假象，R1-R12 可以是其他任意义的值。

为什么程序是 *(--stk) = (INT32U)******;而不是*(++stk) = (INT32U)******

前面已经讲过，M3 的栈生长方向是由高地址向低地址增长的。

栈初始化后，各寄存器的初始值如下：

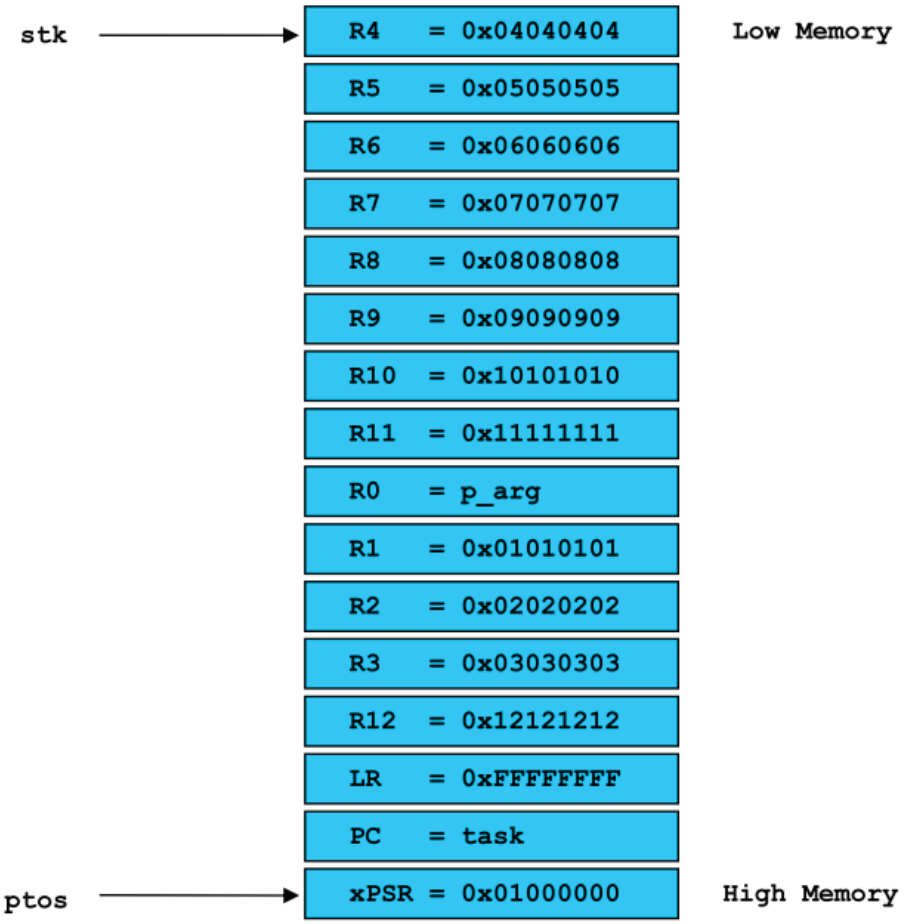


Figure 3-2, The Stack Frame for each Task for ARM Cortex-M3 port.

xPSR = 0x01000000L，**xPSR T 位（第 24 位）置 1**，否则第一次执行任务时 Fault，**PC 必须指向任务入口， R14 = 0xFFFFFFFEL，最低 4 位为 E**，是一个非法值，主要目的是不让使用 R14，即任务是不能返回的。 **R0 用于传递任务函数的参数，因此等于 p_arg**。。

SysTick 时钟初始化

OS_CPU_SysTickInit()会被第一个任务调用，以便初始化 SysTick 定时器。

OS_CPU_SysTickInit() 将会调用 OS_CPU_SysTickClkFreq() 获取系统时钟频率，用户需要为自己的开发板编写此函数获取时钟频率。

```
132.void  OS_CPU_SysTickInit (void)
133.{
134.    INT32U  cnts;
135.
136.
137.    cnts = OS_CPU_SysTickClkFreq() / OS_TICKS_PER_SEC;
138.      //OS_CPU_SysTickClkFreq()获取时钟频率
139.      //OS_TICKS_PER_SEC 定义每秒时钟节拍中断的次数，即时钟节拍时间为 1/OS_TICKS_PER_SEC
140.
141.    /* 使能 SysTick 定时器            */
142.    OS_CPU_CM3_NVIC_ST_RELOAD = (cnts - 1);
143.
144.    /* 使能 SysTick 定时器中断 */
145.    OS_CPU_CM3_NVIC_ST_CTRL  |= OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC
146.                             |  OS_CPU_CM3_NVIC_ST_CTRL_ENABLE;
147.
148.    OS_CPU_CM3_NVIC_ST_CTRL  |= OS_CPU_CM3_NVIC_ST_CTRL_INTEN;
149.}
```

但在这里，为了便于理解，我们需要手动修改成自己的，不用这些函数（看上面任务管理函数中需要注释掉的函数）。

除了注释刚才上面说的三个函数外，我们还要注释掉这些宏定义：

```
150./*
151.*********************************************************************************
152.*                                          SYS TICK DEFINES
153.*********************************************************************************
154.*/
155.#define  OS_CPU_CM3_NVIC_ST_CTRL    (*((volatile INT32U *)0xE000E010))
156.                                             /* SysTick Ctrl & Status Reg. */
157.#define  OS_CPU_CM3_NVIC_ST_RELOAD  (*((volatile INT32U *)0xE000E014))
158.                                             /* SysTick Reload  Value Reg. */
159.#define  OS_CPU_CM3_NVIC_ST_CURRENT (*((volatile INT32U *)0xE000E018))
160.                                             /* SysTick Current Value Reg. */
161.#define  OS_CPU_CM3_NVIC_ST_CAL     (*((volatile INT32U *)0xE000E01C))
162.                                             /* SysTick Cal     Value Reg. */
163.#define  OS_CPU_CM3_NVIC_PRIO_ST    (*((volatile INT8U  *)0xE000ED23))
164.                                             /* SysTick Handler Prio  Reg. */
165.
166.#define  OS_CPU_CM3_NVIC_ST_CTRL_COUNT                    0x00010000
167.                                             /* Count flag.                */
168.#define  OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC                  0x00000004
```

```
169.                                                    /* Clock Source.                */
170. #define  OS_CPU_CM3_NVIC_ST_CTRL_INTEN                      0x00000002
171.                                                    /* Interrupt enable.            */
172. #define  OS_CPU_CM3_NVIC_ST_CTRL_ENABLE                     0x00000001
173.                                                    /* Counter mode.                */
174. #define  OS_CPU_CM3_NVIC_PRIO_MIN                                0xFF
175.                                                    /* Min handler prio.            */
```

因为它们为 SysTick 定时器服务的，即需要把所有与 SysTick 有关的代码都要去掉。

os_cpu_a.asm

这个文件包含了需要用汇编编写的代码。

声明外部定义

```
176. EXTERN  OSRunning                          ; 声明外部定义，相当于 C 语言的 extern
177. EXTERN  OSPrioCur
178. EXTERN  OSPrioHighRdy
179. EXTERN  OSTCBCur
180. EXTERN  OSTCBHighRdy
181. EXTERN  OSIntNesting
182. EXTERN  OSIntExit
183. EXTERN  OSTaskSwHook
```

申明这些变量是在其他文件定义的。

声明全局变量

由于编译器的原因，我们需要将下面的 PUBIC 改为 EXPORT。（如果下载的源代码是用 RealView 编译的，则此处就不用改了，因为代码本来就是用 EXPORT）

```
184. PUBLIC  OS_CPU_SR_Save                          ; 声明函数在此文件定义
185. PUBLIC  OS_CPU_SR_Restore
186. PUBLIC  OSStartHighRdy
187. PUBLIC  OSCtxSw
188. PUBLIC  OSIntCtxSw
189. PUBLIC  OS_CPU_PendSVHandler
```

修改后

```
190. EXPORT  OS_CPU_SR_Save                          ; 声明函数在此文件定义
191. EXPORT  OS_CPU_SR_Restore
192. EXPORT  OSStartHighRdy
193. EXPORT  OSCtxSw
194. EXPORT  OSIntCtxSw
195. EXPORT  OS_CPU_PendSVHandler
```

关于 EXPORT 的用法和意义，可以参考 **RealView 编译工具 4.0 版《汇编器指南》**第 7.8.7 小节 EXPORT 或 GLOBAL：

EXPORT 指令声明一个符号，链接器可以使用该符号解析不同对象和库文件中的符号引用。 GLOBAL 是 EXPORT 的同义词。

使用 EXPORT 可使其他文件中的代码能够访问当前文件中的符号。

与 EXPORT 相对应的是 IMPORT，可以参考 RealView 编译工具 4.0 版《汇编器指南》第 7.8.10 小节 IMPORT 和 EXTERN ：

这些指令为汇编器提供一个未在当前汇编中定义的名称。在链接时，名称被解析为在其他对象文件中定义的符号。 该符号被当作程序地址。 如果未指定 [WEAK] 且在链接时没有找到相应的符号，则链接器会产生错误。

段

由于编译器的原因，也要将下面的内容替换一下：

```
196.RSEG CODE:CODE:NOROOT(2)     ; RSEG CODE：选择段 code。第二个 CODE 表示代码段的意思，只读。
197.                             ;  NOROOT 表示：如果这段中的代码没调用，则允许连接器丢弃这段
198.                             ; (2)表示：4 字节对齐。假如是(n)，则表示 2^n 对齐
```

替换为：

```
199.AREA |.text|, CODE, READONLY, ALIGN=2 ;AREA |.text|  表示：选择段 |.text|。
200.                                       ;CODE 表示代码段，READONLY 表示只读（缺省）
201.                                       ;ALIGN=2 表示 4 字节对齐。若 ALIGN=n，这 2^n 对齐
202.THUMB                                  ;Thumb 代码
203.REQUIRE8                               ;指定当前文件要求堆栈八字节对齐
204.PRESERVE8                              ;令指定当前文件保持堆栈八字节对齐
```

对于汇编命令，想了解更多，请看 **RealView 编译工具 4.0 版《汇编器指南》**

关于段的补充：段可以分为代码段和数据段，其中代码段的内容就是可执行代码。

用 keil 编译时，经常会出现这样的提示：

```
linking...
Program Size: Code=3732 RO-data=336 RW-data=24 ZI-data=512
FromELF: creating hex file...
```

Code 是代码占用的空间，RO-data 是 Read Only 只读常量的大小，如 const 型，RW-data 是（Read Write） 初始化了的可读写变量的大小，ZI-data 是（Zero Initialize）没有初始化的可读写变量的大小。ZI-data 不会被算做代码里因为不会被初始化。

简单的说就是在烧写的时候是 FLASH 中的被占用的空间为：Code+RO Data+RW Data

程序运行的时候，芯片内部 RAM 使用的空间为： RW Data + ZI Data

向量中断控制器 NVIC

前面讲过，关于 PendSV，可以看《Cortex-M3 权威指南》中第 7.6 小节 SVC 和 PendSV。不知道有多少位朋友看过呢？呵呵，如果看过，那下面的内容，就容易理解很多，不然，像看天书那样。

```
205. NVIC_INT_CTRL   EQU      0xE000ED04 ;中断控制及状态寄存器 ICSR 的地址
206.                                      ;见《Cortex-M3 权威指南》第 8.4.5 小节 表 8.5)
207. NVIC_SYSPRI14   EQU      0xE000ED22 ;系统异常优先级寄存器 PRI_14
208.                                      ;即设置 PendSV 的优先级
209.                                      ;见《Cortex-M3 权威指南》第 8.4.2 小节 表 8.3B
210. NVIC_PENDSV_PRI EQU           0xFF ;定义 PendSV 的可编程优先级为 255，即最低
211.                                      ;为啥是最低呢？大家自个思考下
212. NVIC_PENDSVSET  EQU      0x10000000 ;中断控制及状态寄存器 ICSR 的位 28
213.                                  ; 写 1 以悬起 PendSV 中断。读取它则返回 PendSV 的状态
```

关于向量中断控制器 NVIC，推荐大家看《Cortex-M3 权威指南》的第 7、第 8 章，里面有很详细的说明，我这里就不做太多的解释。

回答一下刚才提出的问题：**为啥要把 PendSV 的可编程优先级设为最低？**

与 SVC 异常必须在执行 SVC 指令后立即得到响应的不同，它是可以像普通的中断一样被悬起的（不像 SVC 那样会上访）。OS 可以利用它“缓期执行”一个异常——直到其它重要的任务完成后才执行动作。

悬起 PendSV 的方法是：手工往 NVIC 的 PendSV 悬起寄存器中写 1。悬起后，如果优先级不够高，则将缓期等待执行（这里是为什么需要定义 NVIC_PENDSVSET 的原因）。

PendSV 的典型使用是用在任务切换上。

假如系统使用 SysTick 异常进行任务切换，则正常情况下：

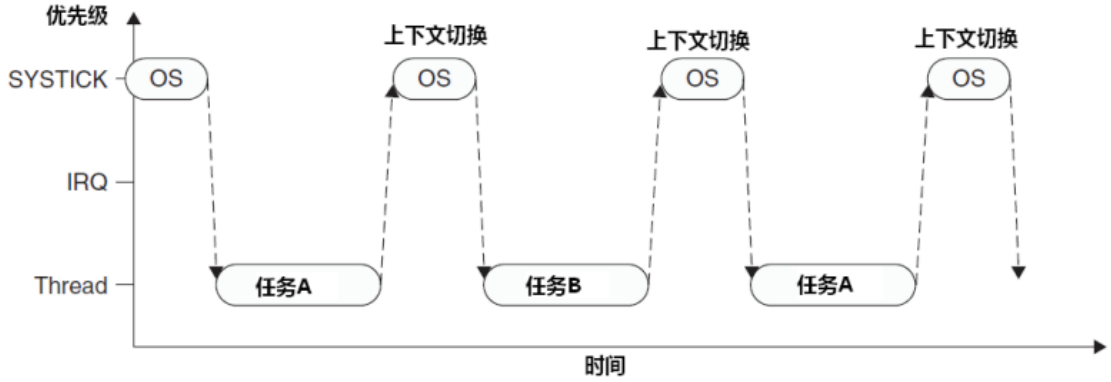
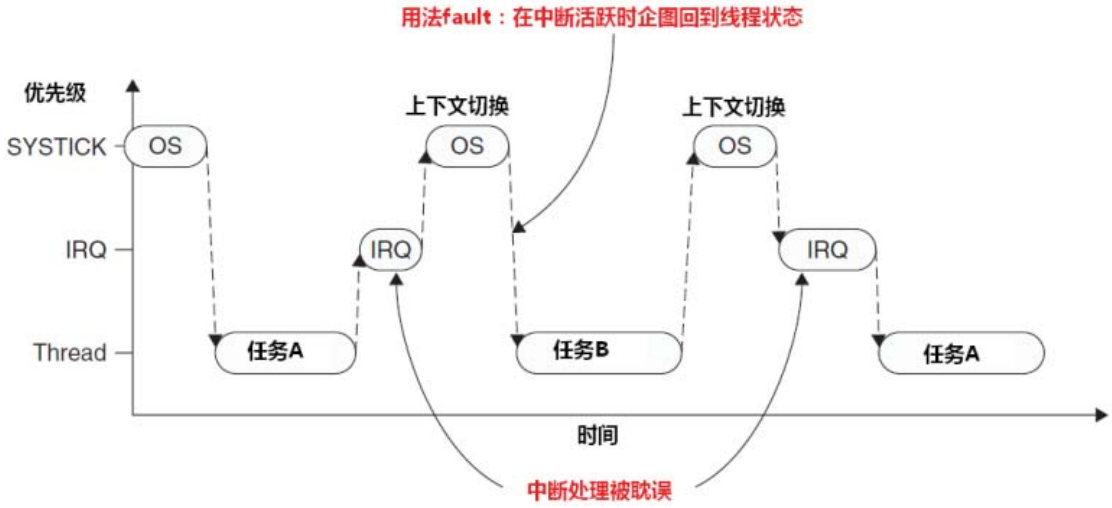
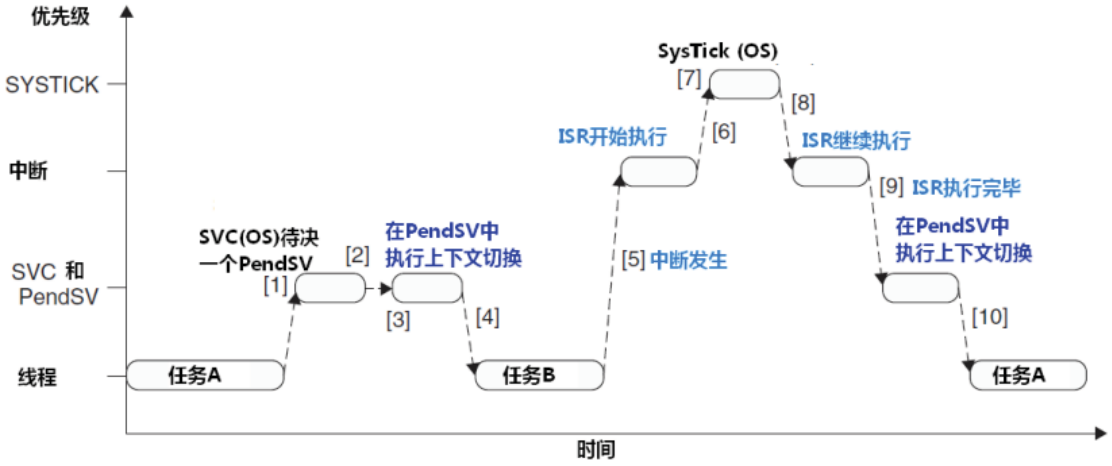


图 7.15 两个任务间通过 SysTick 进行轮转调度的简单模式

但实际上，有时候单片机会进入中断状态响应其他中断，这时如果再产生滴答定时器中断，进行任务切换，打断了原来的中断服务，则运行流程为：



显然，中断服务被打断了，间距的时间比较长，这是实时系统所无法忍受的。为此，引入了 PendSV 来完美解决这个问题了：



PendSV 异常会自动延迟上下文切换的请求，直到其它的 ISR 都完成了处理后才放行。为实现这个机制，需要把 PendSV 编程为最低优先级的异常。

注：此部分内容出自《Cortex-M3 权威指南》

中断

与中断方式 3 相关的有两个汇编函数：

```
214.;  OS_ENTER_CRITICAL() 里进入临界段调用，保存现场环境
215.OS_CPU_SR_Save
216.        MRS     R0, PRIMASK     ; 读取 PRIMASK 到 R0(保存全局中断标记，除了故障中断)
217.        CPSID   I               ; PRIMASK=1，关中断
218.        BX      LR              ; 返回，返回值保存在 R0
219.
220.
221.;  OS_EXIT_CRITICAL()  里退出临界段调用，恢复现场环境
222.OS_CPU_SR_Restore
```

```
223.        MSR     PRIMASK,R0      ; 读取 R0 到 PRIMASK 中(恢复全局中断标记)，通过 R0 传递参数
224.        BX      LR
```

功能：关全局中断前，保存全局中断标志，进入临界段。退出临界段后恢复中断标记。

汇编命令讲解：

功能	作用
CPS （更改处理器状态）	会更改 CPSR 中的一个或多个模式以及 A、I 和 F 位，但不更改其他 CPSR 位。 CPSID 就是中断禁止，CPSIE 中断允许。 A 表示 启用或禁用不精确的中止。 I 表示 启用或禁用 IRQ 中断。 F 表示 启用或禁用 FIQ 中断。 此处 CPSID I 就表示禁止 IRQ 中断
MRS	将 CPSR 或 SPSR 的内容移到一个通用寄存器中。
MSR	将立即数或通用寄存器的内容加载到 CPSR 或 SPSR 的指定字段中。
BL	跳转指令，可将下一个指令的地址复制到 LR（R14，链接寄存器）中。

注：此部分内容出自 RealView 编译工具 4.0 版《汇编器指南》

启动最高优先级任务

OSStartHighRdy()启动最高优先级任务，由 OSStart()里调用，调用前必须先调用 OSTaskCreate 创建至少一个用户任务，否则系统会发生崩毁。

```
225.OSStartHighRdy
226.    LDR     R0, =NVIC_SYSPRI14      ; 装载 系统异常优先级寄存器 PRI_14
227.                                    ; 即设置 PendSV 中断优先级的寄存器
228.    LDR     R1, =NVIC_PENDSV_PRI    ; 装载 PendSV 的可编程优先级(255)
229.    STRB    R1, [R0]                ; 无符号字节寄存器存储。R1 是要存储的寄存器
230.                                    ; 存储到内存地址所基于的寄存器
231.                                    ; 即设置 PendSV 中断优先级为 255
232.
233.    MOV     R0, #0                  ; 把数值 0 复制到 R0 寄存器
234.    MSR     PSP, R0                 ; 将 R0 的内容加载到程序状态寄存器 PSR 的指定字段中。
235.
236.    LDR     R0, __OS_Running        ; OSRunning = TRUE
237.    MOV     R1, #1
238.    STRB    R1, [R0]
```

```
239.
240.     LDR      R0, =NVIC_INT_CTRL      ; 装载 中断控制及状态寄存器 ICSR 的地址
241.     LDR      R1, =NVIC_PENDSVSET     ; 中断控制及状态寄存器 ICSR 的位 28
242.     STR      R1, [R0]                ; 设置 中断控制及状态寄存器 ICSR 位 28 为 1
243.                                      ; 以悬起(允许)PendSV 中断
244.
245.     CPSIE    I                       ; 开中断(前面已经讲解过)
```

任务切换

当任务放弃 CPU 的使用权时，就会调用 OS_TASK_SW()

一般情况下，OS_TASK_SW()是做任务切换。但在 M3 中，任务切换的工作都被放到 PendSV 的中断处理服务中去做以加快处理速度，因此 OS_TASK_SW()只需简单的悬起(允许)PendSV 中断即可。当然，这样就只有当再次开中断的时候，PendSV 中断处理函数才能执行。

OS_TASK_SW()是由 OS_Sched() （此函数在 OS_CORE.C）调用。

```
246./*****************任务级调度器*****************/
247.void  OS_Sched (void)
248.{
249.#if OS_CRITICAL_METHOD == 3
250.    OS_CPU_SR  cpu_sr = 0;
251.#endif
252.
253.    OS_ENTER_CRITICAL();
254.    if (OSIntNesting == 0) {                                  //如果没中断服务运行
255.        if (OSLockNesting == 0) {                             //调度器没上锁
256.            OS_SchedNew();                                    //查找最高优先级就绪任务
257.                                              //见 os_core.c ，会修改 OSPrioHighRdy
258.            if (OSPrioHighRdy != OSPrioCur) {  //如果得到的最高优先级就绪任务不等于当前
259.                                               //注：当前运行的任务也在就绪表里
260.                OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy]; //得到任务控制块指针
261.#if OS_TASK_PROFILE_EN > 0
262.                OSTCBHighRdy->OSTCBCtxSwCtr++; //统计任务切换到次任务的计数器加 1
263.#endif
264.                OSCtxSwCtr++;                  //统计任务切换次数的计数器加 1
265.                OS_TASK_SW();                  //进行任务切换
266.            }
267.        }
268.    }
269.    OS_EXIT_CRITICAL();                        //退出临界段，开中断
270.}
```

悬起(允许)PendSV 中

开中断，执行 PendSV 中断服

OS_TASK_SW()就是用宏定义包装的 OSCtxSw()(见 OS_CPU.H)：

```
271.#define  OS_TASK_SW()          OSCtxSw()
```

前面已经说了，OS_TASK_SW()只需简单的悬起(允许)PendSV 中断即可。

```
272.OSCtxSw
273.        ; 悬起(允许)PendSV 中断 (看不懂这段代码的，可参考前面见过的 OSStartHighRdy )
274.        LDR      R0, =NVIC_INT_CTRL          ; 装载 中断控制及状态寄存器 ICSR 的地址
275.        LDR      R1, =NVIC_PENDSVSET         ; 中断控制及状态寄存器 ICSR 的位 28
276.        STR      R1, [R0]                    ; 设置 中断控制及状态寄存器 ICSR 位 28 为 1
277.                                             ; 以悬起(允许)PendSV 中断
278.        BX       LR                          ; 返回
```

中断退出处理

当中断处理函数退出时，就会调用 OSIntExit()来决定是否有优先级更高的任务需要执行。如果有， OSIntExit()会调用 OSIntCtxSw() 做任务切换。

在 M3 里，与 OSCtxSw 一样，任务切换时，OSIntCtxSw 都只需简单的悬起(允许)PendSV 中断即可，真正的任务切换工作放在 PendSV 中断服务程序里，等待开中断时才正在执行任务切换。

在这里，OSCtxSw 的代码是与 OSCtxSw 完全相同的：

```
279.OSIntCtxSw
280.        LDR      R0, =NVIC_INT_CTRL      ; trigger the PendSV exception
281.        LDR      R1, =NVIC_PENDSVSET
282.        STR      R1, [R0]
283.        BX       LR
```

尽管这里的 SCtxSw()和 OSIntCtxSw()代码是完全一样的，但事实上，这两个函数的意义是不一样的。

OSCtxSw() 做的是任务之间的切换。例如任务因为等待某个资源或做延时，就会调用这个函数来进行任务调度，有任务调度进行任务切换。

OSIntCtxSw()则是中断退出时，如果最高优先级就绪任务并不是被中断的任务就会被调用，由中断状态切换到最高优先级就绪任务中，所以 OSIntCtxSw()又称中断级的中断任务。

由于调用 OSIntCtxSw()之前肯定发生了中断，所以无需保存 CPU 寄存器的值了。这里只不过由于 CM3 的特殊机制导致了在这两个函数中只要做触发 PendSV 中断即可，具体切换由 PendSV 中断服务来处理。

PendSV 中断服务

前面已经讲解过很多次 PendSV 的作用了，这里就不啰嗦了，先来 PendSV 中断服务的伪代码吧，方便理解：

```
284.//OS_CPU_PendSVHandler 伪代码思路
285.OS_CPU_PendSVHandler:
286.  if (PSP != NULL) {    //当调用 OS_CPU_PendSVHandler() 时，
287.                        //CPU 就会自动保存 xPSR、PC、LR、R12、R0-R3 寄存器到堆栈
```

```
288.                        //保存后，CUP 的栈 SP 指针会切换到使用主堆栈指针 MSP 上
289.                        //我们只需检测 进入栈指针 PSP 是否为 NULL 就知道是否进行任务切换
290.                        //因此当我们第一次启动任务是,OSStartHighRdy() 就把 PSP 设为 NULL,
291.                        //避免系统以为已经进行任务切换
292.    Save R4-R11 onto task stack;    //手动保存 R4-R11
293.    OSTCBCur->OSTCBStkPtr = SP;     //保存进入栈指针 PSP 到任务控制块
294.                                    //以便下次继续任务运行时继续使用原来的栈
295.  }
296.  OSTaskSwHook();                   //此处便于我们使用钩子函数来拓展功能
297.  OSPrioCur = OSPrioHighRdy;        //获取最高优先级就绪任务的优先级
298.  OSTCBCur  = OSTCBHighRdy;         //获取最高优先级就绪任务的任务控制块指针
299.  PSP       = OSTCBHighRdy->OSTCBStkPtr;    //保存进入栈指针
300.  Restore R4-R11 from new task stack;       //从新的栈恢复 R4-R11 寄存器
301.  Return from exception;                     //返回
```

具体的汇编代码：

```
302.OS_CPU_PendSVHandler                    ;CPU 会自动保存 xPSR, PC, LR, R12, R0-R3
303.      CPSID   I                         ;关中断
304.      MRS     R0, PSP                   ;PSP 就是栈指针，R0=PSP
305.      CBZ     R0, OSPendSV_nosave       ;当 PSP==0，执行 OSPendSV_nosave 函数
306.
307.      SUB     R0, R0, #0x20             ;装载 r4-11 到栈 ，共 8 个寄存器，32 位，4 个字节
308.                                        ;即 8*4=32=0x20
309.      STM     R0, {R4-R11}              ;
310.
311.      LDR     R1, __OS_TCBCur           ;R1=&OSTCBCur
312.      LDR     R1, [R1]                  ;R1=*R1 (R1=OSTCBCur)
313.      STR     R0, [R1]                  ;*R1=R0 (*OSTCBCur=SP)
314.
315.OSPendSV_nosave
316.      PUSH    {R14}                     ;保存 R14
317.      LDR     R0, __OS_TaskSwHook       ;调用钩子函数 OSTaskSwHook()
318.      BLX     R0
319.      POP     {R14}                     ;恢复 R14
320.
321.      LDR     R0, __OS_PrioCur          ;设置当前优先级为最高优先级就绪任务的优先级
322.                                        ;OSPrioCur = OSPrioHighRdy
323.      LDR     R1, __OS_PrioHighRdy
324.      LDRB    R2, [R1]
325.      STRB    R2, [R0]
326.
327.      LDR     R0, __OS_TCBCur           ;设置当前任务控制块指针
328.      LDR     R1, __OS_TCBHighRdy       ;OSTCBCur  = OSTCBHighRdy
```

```
329.    LDR     R2, [R1]
330.    STR     R2, [R0]
331.
332.    LDR     R0, [R2]                ;R0 是新的 SP
333.                                    ;SP = OSTCBHighRdy->OSTCBStkPtr;
334.
335.    LDM     R0, {R4-R11}            ;从新的栈恢复 R4-R11
336.    ADD     R0, R0, #0x20
337.    MSR     PSP, R0                 ;PSP=R0,用新的栈 SP 加载 PSP
338.    ORR     LR, LR, #0x04           ;确保 LR 位 2 为 1，返回到使用进程堆栈
339.    CPSIE   I                       ;开中断
340.    BX      LR                      ;返回
```

当第一次开始任务切换时时，而任务刚创建时 R4-R11 已经保存在堆栈中，此时不用再保存，就会跳到 OS_CPU_PendSVHandler_nosave 执行。

前面已经说过真正的任务切换是在 PendSV 中断处理函数里做的，由于 M3 在中断时会有一些的寄存器自动保存到任务堆栈里，所以在 PendSV 中断处理函数中只需保存 R4-R11 并调节堆栈指针即可。其中 xPSR, PC, LR, R12, R0-R3 已自动保存，不用我们管了。

下面是一个任务切换时寄存器的情况：

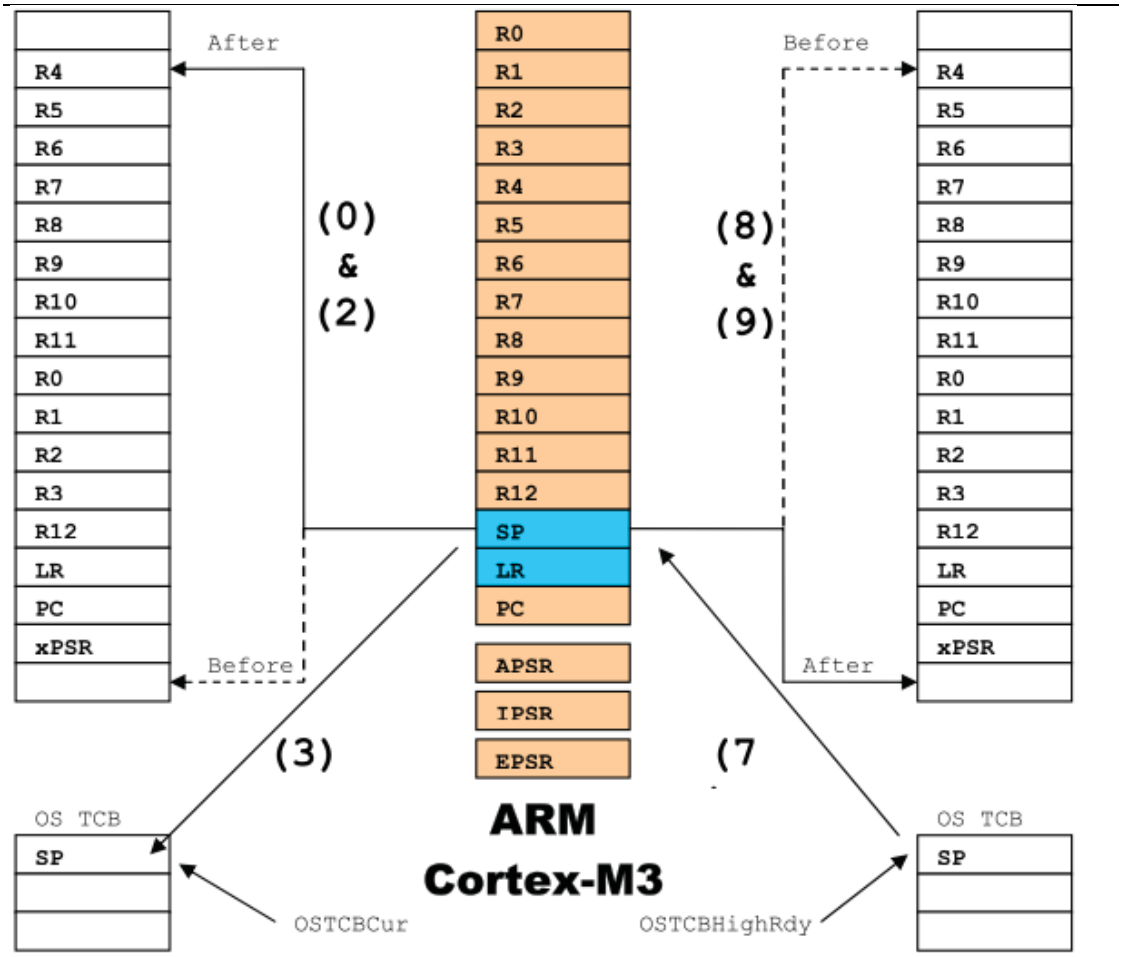
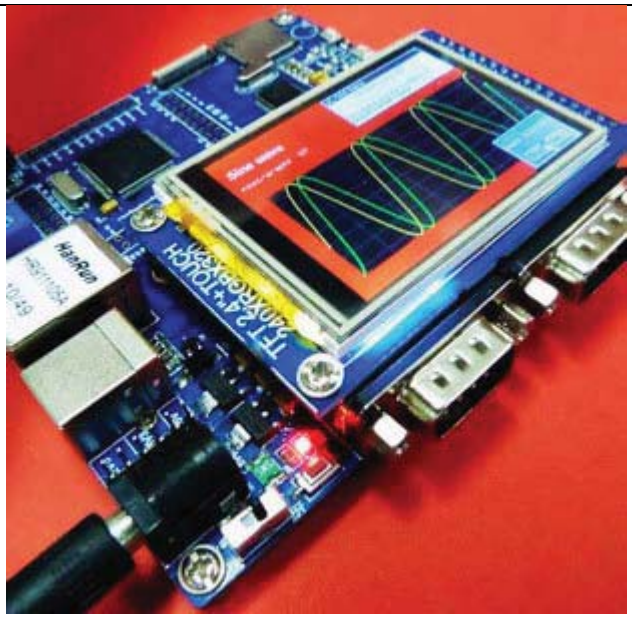


Figure 3-3, ARM Cortex-M3 Context Switch.

到此，重要的代码知识点就讲解完毕了，呵呵，初学者看起来会有点困难，不过要加油哦！多看几次就可以弄懂的！限于个人能力，欢迎各位高手指出错误，在此先表达谢意！

uC/OS-II 移植到 STM32 处理器的步骤

下面，我们将讲解移植 uC/OS-II 到野火开发板的示范实验，先来一张野火开发板的图片：



我们的 uC/OS-II 移植实验是在野火 STM32 开发板附带的 LED 实验基础上来讲的，所用的工程文件也是野火 STM32 开发板所带的 LED 例程。

对于没接触过野火 STM32 开发板实验教程的朋友，建议你们还是看下野火 的 LED 教程。

好了，转到正题上。看完前面的内容，不知道各位是否对 uC/OS-II 的移植有了整体的把握了？对于 uC/OS-II 的工程文件结构，又是否了解呢？

我们先来回顾一下一个 uC/OS-II 的开发板工程的文件结构吧：

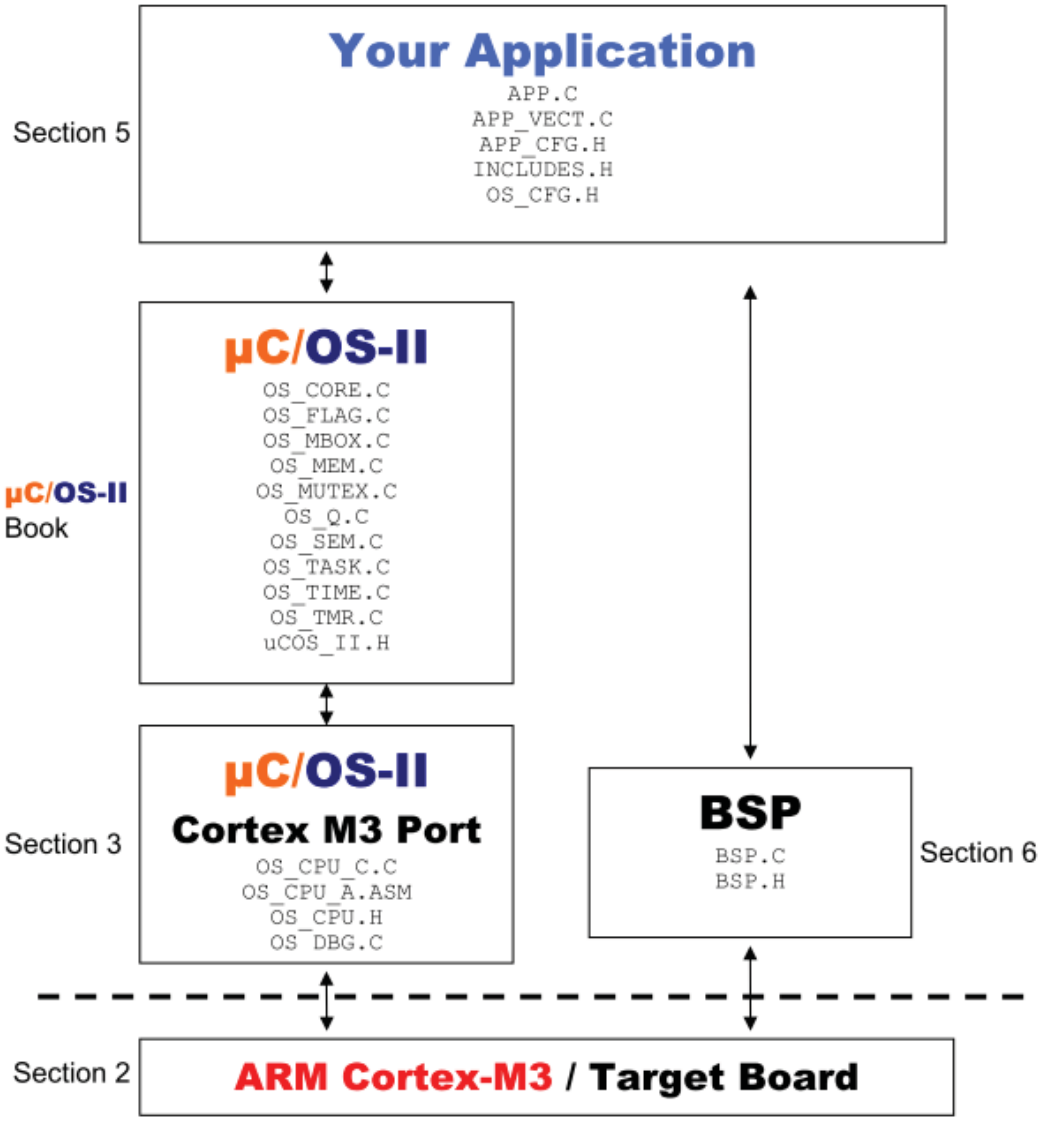


Figure 1-1, Relationship between modules.

很明显，为了让开发板硬件驱动程序与 uC/OS-II 系统的文件系统分开，好让我们开发工程时不必太乱，我们需要按照一定的规则建立分类文件夹。

好了，下面开始正式移植 uC/OS-II 了：

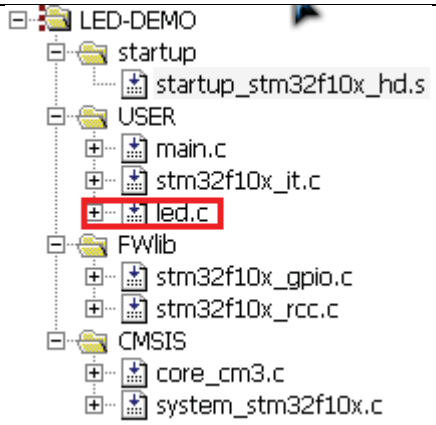
在这里，我们直接采用野火 STM32 的 LED 工程来作为基础，进行 uC/OS-II 移植的讲解(如果不知道 LED 工程如何建立，请看野火 STM32 的 LED 教程，这里就不再重复)：

打开 LED 工程模版

首先，我们从野火 STM32 光盘资料那里提取 LED 实验:

LED 工程文件在光盘目录下:\实验代码+PDF 教程\野火 Stm32-实验代码\ 2-LED.rar

解压打开工程后，就会看到 LED 工程的文件结构为：



这个是我们过往开发裸机单片机程序时写的工程文件结构，但对于 uC/OS-II，或者其他大型点的软件工程，这样的文件结构就会很乱的。

搭建 uC/OS-II 工程文件结构

我们需要建立的文件结构为(其他没显示出来的文件，按照原来位置那样不改变)：

```
STM32+UCOS+LED
 |
 ├──USER
 |       main.c
 |       includes.h          //新建
 ├──uCOS-II
 |    ├──Source              //这文件夹来自于下载附件的 Micrium\Software\uCOS-II
 |    |       os_core.c
 |    |       os_flag.c
 |    |       os_mbox.c
 |    |       os_mem.c
 |    |       os_mutex.c
 |    |       os_q.c
 |    |       os_sem.c
 |    |       os_task.c
 |    |       os_time.c
 |    |       os_tmr.c
 |    |       ucos_ii.h
 |    └──Ports      //里面文件来自于
\Micrium\Software\uCOS-II\Ports\arm-cortex-m3\Generic\IAR
 |            os_cpu.h
 |            os_cpu_a.asm
 |            os_cpu_c.c
 |            os_dbg.c
 ├──BSP                      //这文件夹新建，里面文件来自 USER 文件夹
 |       led.c
 |       led.h
 |       BSP.c
 |       BSP.h
 └──APP                      //这文件及里面的文件（除 os_cfg.h）都是新建
```

app.c
app.h
app_cfg.h　//是用来配置应用软件，主要是任务的优先级和堆栈大小，中断优先级等
os_cfg.h　//拷贝自 Micrium\Software\EvalBoards\ST\S.\I.\OS.\os_cfg.h

为了方便初学者，下面的为具体的详细步骤，如果会自行搭建文件结构，可跳过这一小节：

① 把 LED 工程所在的文件夹先改名为：STM32+UCOS+LED (建议这样做，避免与原来 LED 工程混乱)

② 在 USER 文件夹下新建 includes.h 头文件。

③ 按照之前给的 uC/OS-II 文件结构图，我们在工程的根目录下建立 BSP 文件夹、APP 文件夹和 uCOS-II 文件夹。

BSP 文件夹　存放外设硬件驱动程序。
APP 文件夹　存放应用软件任务
uCOS-II 文件夹　uC/OS-II 的相关代码

④ 把 USER 文件夹下的 led.h 和 led.c 文件剪切到 BSP 文件夹里。

在 BSP 文件夹里新建 BSP.c 和 BSP.h 文件。

⑤ 在 APP 文件夹下建立 app.h、app.c 和 app_cfg.h 文件。

拷贝 uC/OS-II 源代码附件那里的 Micrium\Software\EvalBoards\ST\STM32F103ZE-SK\IAR\OS-Probe-LCD\os_cfg.h 到此目录。

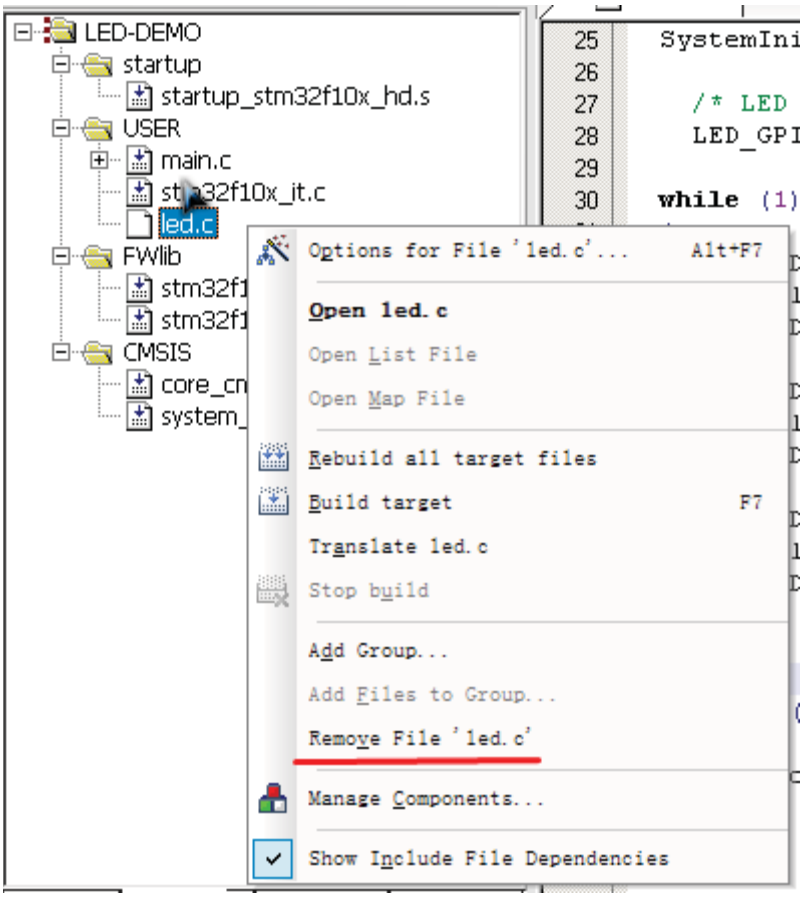
⑥ 把 uC/OS-II 源代码附件那里的\Micrium\Software\uCOS-II 下的 Source 文件夹复制到工程里刚才新建的 uCOS-II 文件夹里。

把 Micrium\Software\uCOS-II\Ports\arm-cortex-m3\Generic\IAR 下的文件复制到工程 uCOS-II 文件夹中新建的 Ports 文件夹里。复制后，选中全部文件，右键——属性——去掉只读属性——确定。

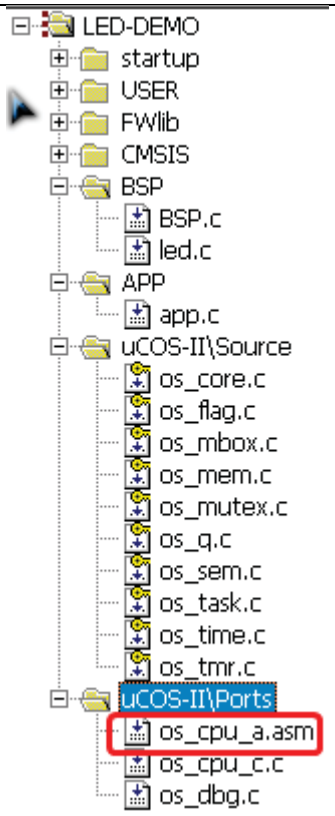
到此，工程的目录结构就建立好了，需要修改工程设置。

⑦ 打开工程文件，会发现提示出错，不需要管他，直接点击确定就可以了。

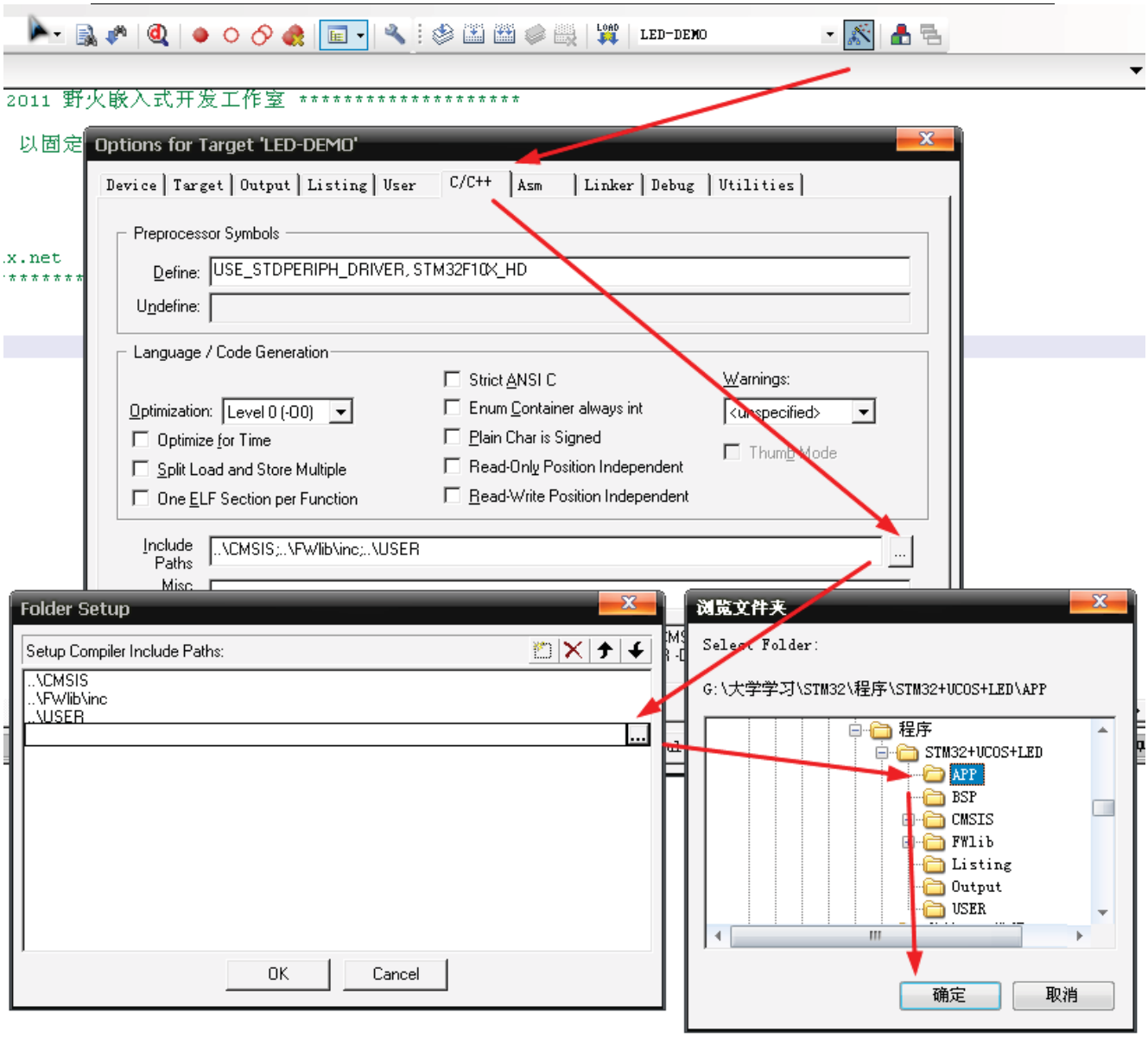
原因是我们修改了 led.h 和 led.c 的路径。所以我们需要在项目里手动删掉原来的 led.c ：



建立 BSP、APP 和 uCOS-II 下两个文件夹，即共四个文件夹的组，并添加进相应的文件：



注意：别漏了在 uCOS-II\Ports 中添加汇编文件 os_cpu_a.asm！！！
也要添加这四个文件进去编译路径：



即 include paths 设置为：

..\CMSIS;..\FWlib\inc;..\USER;..\APP;..\BSP;..\uCOS-II\Ports;..\uCOS-II\Source

至此，完成全部工程的设置，需要开始移植修改代码了！

配置 uC/OS-II

首先，修改代码，当然是从配置 uC/OS-II 开始啦。因为我们做的是简单的实验，为此，我们需要把多余的模块剪裁掉，等需要用到再启用，以减少内核体积。

os_cfg.h

os_cfg.h 是用来配置系统功能的，我们需要通过修改它来达到剪裁系统功能的目的。

在做实际项目时，我们通常也不会用完全部的 uC/OS-II 功能，我们需要通过裁剪内核以避免浪费系统的宝贵资源。

配置 os_cfg.h ，是每个入门移植 uC/OS-II 的初学者都应该需要学会的，尽管非常枯燥无味。

其实，os_cfg.h 的配置也是有规律的。

os_cfg.h 配置表格

文件名	分类		配置宏	注解
os_cfg.h	功能裁剪	任务	OS_TASK_CHANGE_PRIO_EN	改变任务优先级
			OS_TASK_CREATE_EN	
			OS_TASK_CREATE_EXT_EN	
			OS_TASK_DEL_EN	
			OS_TASK_NAME_SIZE	
			OS_TASK_PROFILE_EN	
			OS_TASK_QUERY_EN	获得有关任务的信息
			OS_TASK_STAT_EN	使用统计任务
			OS_TASK_STAT_STK_CHK_EN	检测任务堆栈
			OS_TASK_SUSPEND_EN	
			OS_TASK_SW_HOOK_EN	
		信号量集	OS_FLAG_EN	
			OS_FLAG_ACCEPT_EN	
			OS_FLAG_DEL_EN	
			OS_FLAG_QUERY_EN	
			OS_FLAG_WAIT_CLR_EN	
		消息邮箱	OS_MBOX_EN	
			OS_MBOX_ACCEPT_EN	
			OS_MBOX_DEL_EN	
			OS_MBOX_PEND_ABORT_EN	
			OS_MBOX_POST_EN	
			OS_MBOX_POST_OPT_EN	
			OS_MBOX_QUERY_EN	
		内存管理	OS_MEM_EN	
			OS_MEM_QUERY_EN	
		互斥信号量	OS_MUTEX_EN	
			OS_MUTEX_ACCEPT_EN	
			OS_MUTEX_DEL_EN	
			OS_MUTEX_QUERY_EN	
		队列	OS_Q_EN	
			OS_Q_ACCEPT_EN	
			OS_Q_DEL_EN	
			OS_Q_FLUSH_EN	
			OS_Q_PEND_ABORT_EN	
			OS_Q_POST_EN	
			OS_Q_POST_FRONT_EN	
			OS_Q_POST_OPT_EN	
			OS_Q_QUERY_EN	
		信号量	OS_SEM_EN	
			OS_SEM_ACCEPT_EN	
			OS_SEM_DEL_EN	
			OS_SEM_PEND_ABORT_EN	

			OS_SEM_QUERY_EN	
			OS_SEM_SET_EN	
		时间管理	OS_TIME_DLY_HMSM_EN	
			OS_TIME_DLY_RESUME_EN	
			OS_TIME_GET_SET_EN	
			OS_TIME_TICK_HOOK_EN	
		定时器管理	OS_TMR_EN	
		其他	OS_APP_HOOKS_EN	应用函数钩子函数
			OS_CPU_HOOKS_EN	CPU 钩子函数
			OS_ARG_CHK_EN	
			OS_DEBUG_EN	调试
			OS_EVENT_MULTI_EN	使能多重事件控制
			OS_TICK_STEP_EN	使能节拍定时
			OS_SCHED_LOCK_EN	使能调度锁
	数据结构	任务	OS_MAX_TASKS	
			OS_TASK_TMR_STK_SIZE	
			OS_TASK_STAT_STK_SIZE	统计任务堆栈容量
			OS_TASK_IDLE_STK_SIZE	
		信号量集	OS_MAX_FLAGS	
			OS_FLAG_NAME_SIZE	
			OS_FLAGS_NBITS	
		内存管理	OS_MAX_MEM_PART	内存块的最大数目
			OS_MEM_NAME_SIZE	
		队列	OS_MAX_QS	消息队列的最大数目
		定时器管理	OS_TMR_CFG_MAX	
			OS_TMR_CFG_NAME_SIZE	
			OS_TMR_CFG_WHEEL_SIZE	
			OS_TMR_CFG_TICKS_PER_SEC	
		其他	OS_EVENT_NAME_SIZE	
			OS_LOWEST_PRIO	最低优先级
			OS_MAX_EVENTS	事件控制块的最大数量
			OS_TICKS_PER_SEC	节拍定时器每 1s 定时次数

我们需要对 os_cfg.h 进行如下修改：

① 首先肯定是禁用信号量、互斥信号量、邮箱、队列、信号量集、定时器、内存管理，关闭调试模式：

```
341.#define OS_FLAG_EN                0    //禁用信号量集
342.#define OS_MBOX_EN                0    //禁用邮箱
343.#define OS_MEM_EN                 0    //禁用内存管理
344.#define OS_MUTEX_EN               0    //禁用互斥信号量
345.#define OS_Q_EN                   0    //禁用队列
346.#define OS_SEM_EN                 0    //禁用信号量
347.#define OS_TMR_EN                 0    //禁用定时器
348.#define OS_DEBUG_EN               0    //禁用调试
```

② 现在也用不着应用软件的钩子函数，也禁掉；多重事件控制也禁掉

```
349.#define OS_APP_HOOKS_EN           0
350.#define OS_EVENT_MULTI_EN         0
```

这些所做的修改主要是把一些功能给去掉，减少内核大小，也利于调试。等用到的时候，再开启相应的功能。

注意，有时候，配置时，会出现无法通过编译，例如提示某个变量没声明。一方面有可能是你自己配置问题，另外一方面，也有可能是作者代码不够完善。

做完这个移植实验后，你们可以来试验一下.

把 OS_Q_EN 和 OS_MBOX_EN 都设为 0，OS_EVENT_MULTI_EN 为 1，编译时会提示:

..\uCOS-II\Source\os_core.c(535): error: #136: struct "os_tcb" has no field "OSTCBMsg"

意思是在 os_core.c 第 535 行 结构体 os_tcb 没有 OSTCBMsg 这个成员。当然，解决方法也很简单。

修改 os_cpu.h

前面我们已经介绍了移植过程中要修改的三个文件，首先我们来看 os_cpu.h:

```
void        OS_CPU_SysTickHandler(void);
void        OS_CPU_SysTickInit(void);
INT32U      OS_CPU_SysTickClkFreq(void);
```

将以上三个文件注释掉即可。

修改 os_cpu_c.c

把 OS_CPU_SysTickHandler(), OS_CPU_SysTickInit()及

```
#define  OS_CPU_CM3_NVIC_ST_CTRL     (*((volatile INT32U *)0xE000E010))
#define  OS_CPU_CM3_NVIC_ST_RELOAD   (*((volatile INT32U *)0xE000E014))
#define  OS_CPU_CM3_NVIC_ST_CURRENT  (*((volatile INT32U *)0xE000E018))
#define  OS_CPU_CM3_NVIC_ST_CAL      (*((volatile INT32U *)0xE000E01C))
#define  OS_CPU_CM3_NVIC_PRIO_ST     (*((volatile INT8U  *)0xE000ED23))

#define  OS_CPU_CM3_NVIC_ST_CTRL_COUNT                    0x00010000
#define  OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC                  0x00000004
#define  OS_CPU_CM3_NVIC_ST_CTRL_INTEN                    0x00000002
#define  OS_CPU_CM3_NVIC_ST_CTRL_ENABLE                   0x00000001
#define  OS_CPU_CM3_NVIC_PRIO_MIN                           0xFF
```

注释掉（前面加#if 0 ，后面加#endif 就能注释掉）

修改 os_cpu_a.asm

由于编译器的原因：

要将下面的 PUBIC 改为 EXPORT

即：

```
351.    EXPORT  OS_CPU_SR_Save              ; Functions declared in this file
352.    EXPORT  OS_CPU_SR_Restore
353.    EXPORT  OSStartHighRdy
354.    EXPORT  OSCtxSw
355.    EXPORT  OSIntCtxSw
356.    EXPORT  OS_CPU_PendSVHandler
```

改为：

```
357.    PUBLIC  OS_CPU_SR_Save                  ; Functions declared in this file
358.    PUBLIC  OS_CPU_SR_Restore
359.    PUBLIC  OSStartHighRdy
360.    PUBLIC  OSCtxSw
361.    PUBLIC  OSIntCtxSw
362.    PUBLIC  OS_CPU_PendSVHandler
```

下面这个也要修改下

原来的：

```
363.    RSEG CODE:CODE:NOROOT(2)
```

修改后：

```
364.    AREA |.text|, CODE, READONLY, ALIGN=2    ;AREA |.text| 选择段 |.text|。
365.                                             ;CODE 表示代码段,READONLY 表示只读(缺省)
366.                                            ;ALIGN=2 表示 4 字节对齐。若 ALIGN=n，这 2^n 对齐
367.    THUMB                                    ;Thumb 代码
368.    REQUIRE8                                 ;指定当前文件要求堆栈八字节对齐
369.    PRESERVE8                                ;令指定当前文件保持堆栈八字节对齐
```

修改 os_dbg.c

将 os_dbg.c 中

```
370.#define  OS_COMPILER_OPT  __root
```

修改为：

```
371.#define  OS_COMPILER_OPT   // __root
```

这个问题也是由编译器不同产生的。

修改 startup_stm32f10x_hd.s

修改完了这几个必要的部分后，有一处我们也必须要注意的。因为我们的移植是使用标准外设库 CMSIS 中 startup_stm32f10x_hd.s 作为启动文件的，还没有设置 OS_CPU_SysTickHandler。而 startup_stm32f10x_hd.s 文件中，PendSV 中断向量名为 PendSV_Handler，因此只需把所有出现 PendSV_Handler 的地方替换成 OS_CPU_PendSVHandler 即可。

至此，修改 uC/OS-II 代码就差不多结束，剩下的，就是编写我们自己的代码。

编写 includes.h

includes.h 是保存全部头文件的头文件，方便我们理清工程函数思路。先给大家看我们用到的头文件，以便让大家知道我们的工程是怎样的一个架构。

```
372.#ifndef  __INCLUDES_H__
373.#define  __INCLUDES_H__
374.
375.#include "stm32f10x.h"
376.#include "stm32f10x_rcc.h"   //SysTick 定时器相关
377.
378.#include  "ucos_ii.h"       //uC/OS-II 系统函数头文件
379.
380.#include    "BSP.h"         //与开发板相关的函数
381.#include    "app.h"         //用户任务函数
382.#include    "led.h"         //LED 驱动函数
383.
384.#endif //__INCLUDES_H__
```

编写 BSP

在前面我们讲到 SysTick 定时器我们自己定义，因此在 BSP.c 中我们加入我们自己的定义并在 BSP.h 中声明这个函数。这个函数需要添一个头文件 stm32f10x_rcc.h

另外，我们也需要编写一个开发板初始化启动函数 BSP_Init()，包含设置系统时钟，初始化硬件。

BSP.C 文件代码

```
385.#include "includes.h"
386.
387./*
388. * 函数名: BSP_Init
389. * 描述  : 时钟初始化、硬件初始化
390. * 输入  : 无
391. * 输出  : 无
392. */
393.void BSP_Init(void)
394.{
395.    SystemInit();    /* 配置系统时钟为 72M */
396.    LED_GPIO_Config();  /* LED 端口初始化 */
397.}
398.
399./*
400. * 函数名: SysTick_init
401. * 描述  : 配置 SysTick 定时器
402. * 输入  : 无
403. * 输出  : 无
404. */
405.void SysTick_init(void)
406.{
407.    SysTick_Config(SystemFrequency/OS_TICKS_PER_SEC);//初始化并使能 SysTick 定时器
408.}
```

BSP.h 头文件

```
409.
410.#ifndef __BSP_H
411.#define __BSP_H
412.
413.void SysTick_init(void);
414.void BSP_Init(void);
415.
416.#endif // __BSP_H
```

编写 stm32f10x_it.c

需要在 stm32f10x_it.c 添加 SysTick 中断的处理代码：

```
417.void SysTick_Handler(void)
418.{
419.    OSIntEnter();
```

```
420.     OSTimeTick();
421.     OSIntExit();
422. }
```

因为调用 uC/OS-II 的函数，所以
这样之后，时钟也配置好了。下面我们可以创建任务了。

创建任务

编写 app_cfg.h

用来设置任务的优先级和栈大小

```
423. #ifndef  __APP_CFG_H__
424. #define  __APP_CFG_H__
425.
426. /*******************设置任务优先级******************/
427. #define STARTUP_TASK_PRIO       4
428.
429. /************设置栈大小（单位为 OS_STK ）************/
430. #define STARTUP_TASK_STK_SIZE   80
431.
432. #endif
```

编写 app.c

这个是创建 LED 显示任务

```
433. #include "includes.h"
434.
435. void Task_LED(void *p_arg)
436. {
437.     SysTick_init();
438.     while (1)
439.     {
440.         LED1( ON );
441.         OSTimeDlyHMSM(0, 0,0,500);
442.         LED1( OFF);
443.
444.         LED2( ON );
445.         OSTimeDlyHMSM(0, 0, 0,500);
446.         LED2( OFF);
447.
448.         LED3( ON );
449.         OSTimeDlyHMSM(0, 0,0,500);
450.         LED3( OFF);
451.
```

```
452.     }
453. }
```

编写 app.h 头文件

```
454. #ifndef _APP_H_
455. #define _APP_H_
456. 
457. /*************** 用户任务声明 ******************/
458. void Task_LED(void *p_arg);
459. 
460. #endif  //_APP_H_
```

main 函数

```
461. #include "includes.h"
462. 
463. static OS_STK startup_task_stk[STARTUP_TASK_STK_SIZE];        //定义栈
464. 
465. int main(void)
466. {
467.     BSP_Init();
468.     OSInit();
469.     OSTaskCreate(Task_LED,(void *)0,
470.         &startup_task_stk[STARTUP_TASK_STK_SIZE-1], STARTUP_TASK_PRIO);
471. 
472.     OSStart();
473.     return 0;
474. }
```

编译之后，发现没错误了，下载下去看下灯闪了，哈哈，成功了。

简单的 uC/OS 移植就这样完成了，难不？

运行多任务

移植 uC/OS-II 弄好了，那运行多任务更简单。

现在，我们要做的实验就是：主任务 Task_Start 先创建，再在主任务运行时创建两个任务 Task_LED2 和 Task_LED3。三个任务都分别控制 3 个 LED 灯。

这次，我们只需修改 4 个文件即可完成这次实验：main.c 、app.c 、app.h 、app_cfg.h，其他的都是跟原来工程一样的。

这次都是依葫芦画瓢，就不讲解了，直接上代码：

修改 app.c

```
475.#include "includes.h"
476.
477.OS_STK task_led2_stk[TASK_LED2_STK_SIZE];          //定义栈
478.OS_STK task_led3_stk[TASK_LED3_STK_SIZE];          //定义栈
479. //主任务
480.void Task_Start(void *p_arg)
481.{
482.    (void)p_arg;                                  // 'p_arg' 并没有用到，防止编译器提示警告
483.    SysTick_init();
484.
485.    OSTaskCreate(Task_LED2,(void *)0,             //创建任务 2
486.                 &task_led2_stk[TASK_LED2_STK_SIZE-1], TASK_LED2_PRIO);
487.
488.    OSTaskCreate(Task_LED3,(void *)0,             //创建任务 3
489.                 &task_led3_stk[TASK_LED3_STK_SIZE-1], TASK_LED3_PRIO);
490.
491.    while (1)
492.    {
493.        LED1( ON );
494.        OSTimeDlyHMSM(0, 0,0,100);
495.        LED1( OFF);
496.        OSTimeDlyHMSM(0, 0,0,100);
497.    }
498.}
499.
500.  //任务 2
501.void Task_LED2(void *p_arg)
502.{
503.    (void)p_arg;
504.    SysTick_init();
505.
506.    while (1)
507.    {
508.        LED2( ON );
509.        OSTimeDlyHMSM(0, 0,0,200);
510.        LED2( OFF);
511.        OSTimeDlyHMSM(0, 0,0,200);
512.    }
513.}
```

```
514.
515.//任务 3
516.void Task_LED3(void *p_arg)
517.{
518.    (void)p_arg;
519.    SysTick_init();
520.
521.    while (1)
522.    {
523.        LED3( ON );
524.        OSTimeDlyHMSM(0, 0,0,300);
525.        LED3( OFF);
526.        OSTimeDlyHMSM(0, 0,0,300);
527.    }
528.}
```

编写 app.h

```
529.#ifndef _APP_H_
530.#define _APP_H_
531.
532./*************** 用户任务声明 ******************/
533.void Task_Start(void *p_arg);
534.void Task_LED2(void *p_arg);
535.void Task_LED3(void *p_arg);
536.#endif  //_APP_H_
```

编写 app_cfg.h

```
537.#ifndef  __APP_CFG_H__
538.#define  __APP_CFG_H__
539.
540.
541./******************设置任务优先级******************/
542.#define STARTUP_TASK_PRIO       4
543.#define TASK_LED2_PRIO          5
544.#define TASK_LED3_PRIO          6
545.
546./************设置栈大小（单位为 OS_STK ）************/
547.#define STARTUP_TASK_STK_SIZE   80
548.#define TASK_LED2_STK_SIZE      80
549.#define TASK_LED3_STK_SIZE      80
550.
551.#endif
```

编写 main.c

```
552./******************** (C) COPYRIGHT 2011 野火嵌入式开发工作室 ********************
553. * 文件名  : main.c
554. * 描述    : 建立 3 个任务，每个任务控制一个 LED，以固定的频率轮流闪烁（频率可调）。
555. * 实验平台：野火 STM32 开发板
556. * 库版本  : ST3.0.0
557. *
558. * 作者    : fire  QQ: 313303034
559. * 博客    : firestm32.blog.chinaunix.net
560.*******************************************************************************/
561.#include "includes.h"
562.
563.OS_STK startup_task_stk[STARTUP_TASK_STK_SIZE];        //定义栈
564.
565.int main(void)
566.{
567.    BSP_Init();
568.    OSInit();
569.    OSTaskCreate(Task_Start,(void *)0,
570.       &startup_task_stk[STARTUP_TASK_STK_SIZE-1], STARTUP_TASK_PRIO);
571.
572.    OSStart();
573.    return 0;
574. }
575.
576./******************** (C) COPYRIGHT 2011 野火嵌入式开发工作室 *****END OF FILE****/
```

依葫芦画瓢，相信你可以很快就学会。

最后野火祝大家学习愉快^_^……

—— fire 野火　　2011 于广州