
从 0 开始移植 uCOS II 到野火 stm32 开发板

作者	Fire Team
E-Mail	firestm32@foxmail.com
QQ	313303034
淘宝	http://firestm32.taobao.com
件平台	野火 STM32 开发板
库版本	ST3.0.0

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 11 月 3 日

目录

从 0 开始移植 UCOS II 移植到野火 stm32 开发板	1
前言	1
第一章 官方源代码介绍	3
第二章 重要文件代码详解	6
2.1 os_cpu.h	7
2.2 os_cpu_c.c	9
2.3 os_cpu_a.asm	13
第三章 uC/OS-II 移植到 STM32 处理器的步骤	21
3.1 打开 LED 工程模版	22
3.2 搭建 uC/OS-II 工程文件结构	24
3.3 配置 uC/OS-II	27
3.4 修改 os_cpu.h	30
3.5 修改 os_cpu_c.c	30
3.6 修改 os_cpu_a.asm	30
3.7 修改 os_dbg.c	31
3.8 修改 startup_stm32f10x_hd.s	31
3.9 编写 includes.h	31
3.10 编写 BSP	32
3.11 编写 stm32f10x_it.c	32
3.12 创建任务	33
3.13 main 函数	33
第四章 运行多任务	34
4.1 修改 app.c	34
4.2 编写 app.h	35
4.3 编写 app_cfg.h	35
4.4 编写 main.c	35
第五章 升级到最新版本 V2.90	36
5.1 不得不说的这些话儿	36
5.2 从 V2.86 升级到 V2.90 说明	37
5.3 更新步骤	38
5.4 更新其他工程	39
第六章 移植计算器	40
6.1 处理外部中断	40
6.2 移植步骤	40
6.3 增加通信功能——队列	48




第一章 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>

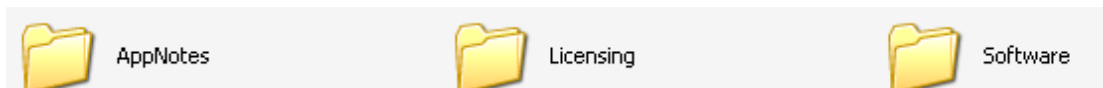
我们需要下载的就是下面这个，因为我用到的开发板芯片是 STM32F103VET6

Download	Processor	OS version	Compiler	Contributor
Download  see STM3210B-EVAL see STM3210E-EVAL see STM32-SK	STM32 (Cortex-M3)	V2.86	IAR & ARM/Keil	Micrium
Download  see STM32F103ZE-SK	<u>STM32 (Cortex-M3)</u>	V2.86	<u>IAR</u>	Micrium
Download  see uC-Eval-STM32F107	STM32F107	v2.92	IAR V6.10.5	Micrium

注意：下载的源代码开发环境是 IAR 编译器的，注意哦！

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
		官方移植到 M3 的移植文件（IAR 工程）
		Ports
		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>

提示一下，如果是没移植好的，是找不到 main 函数的哦！初学者，相信很多都下载没移植好的，然后直接看它的源代码，然后看到头晕也找不到工程的入口。其实，uC/OS 就是一个库而已，熟悉它的运行流程和函数接口，就可以基本跑起来。

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）

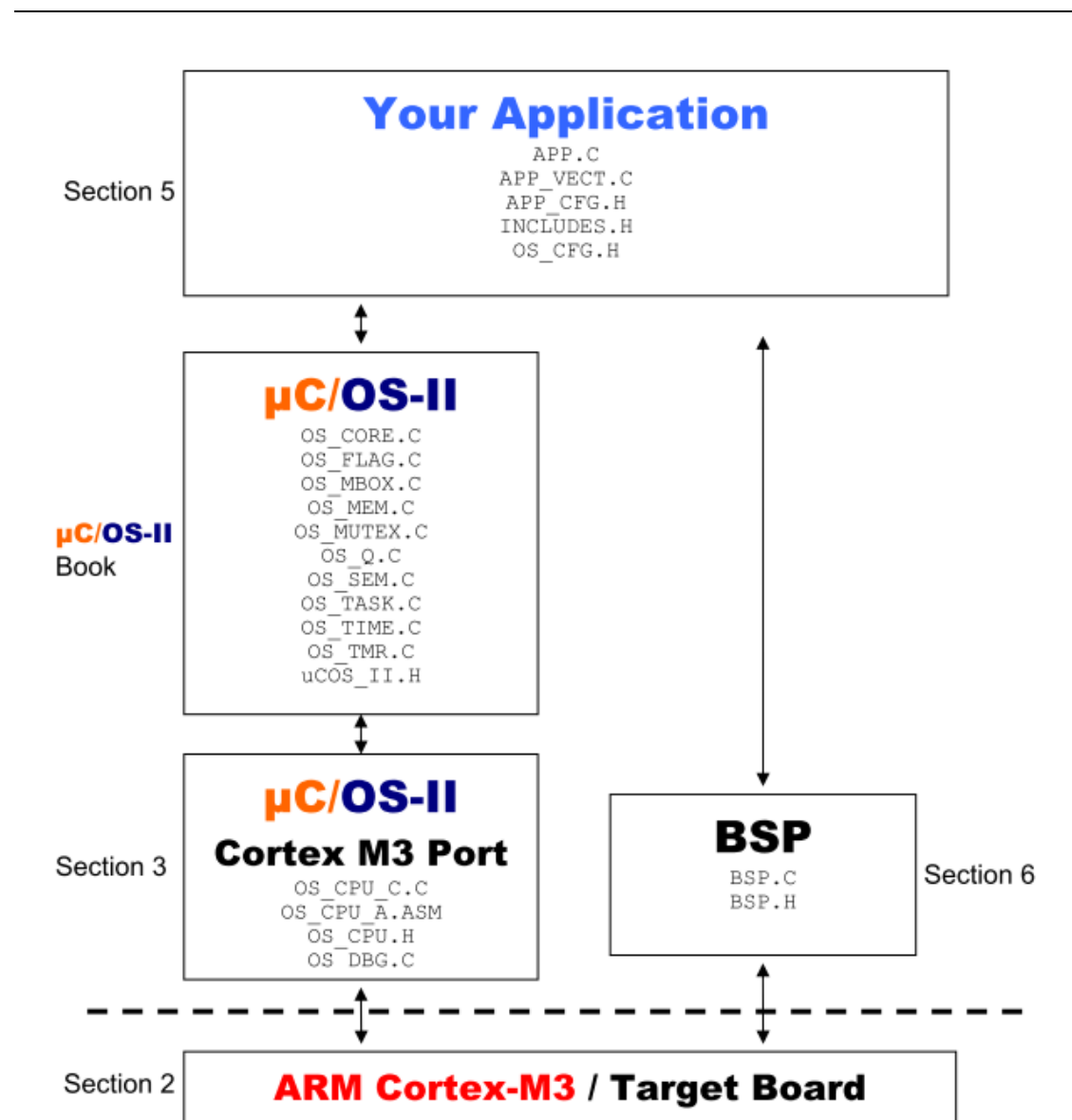


Figure 1-1, Relationship between modules.

第二章 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	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 uC/OS-II 原理及应用 (北京航空航天大学出版社)	通俗易懂的一本 uC/OS 教程，非常适合初学者学习。 不过教程没得到更新，不能适应

Joseph Yiu 著
宋岩 译

Cortex-M3 权威指南

uC/OS 的发展，但还是值得推荐。
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，欢迎指出错误。**

2.1 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.
16. typedef unsigned int OS_STK; //M3 是 32 位，所以堆栈的数据类型 OS_STK 设置 32 位
17. 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()。

```
18. #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
19.
20.
21. #define OS_ENTER_CRITICAL() {cpu_sr = OS_CPU_SR_Save();} //进入临界段
22. #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。

```
23. #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。

```
24. #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

```
25. #if OS_CRITICAL_METHOD == 3
26. OS_CPU_SR OS_CPU_SR_Save(void);
27. void OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
28. #endif
```

任务管理函数

```
29. /*****任务切换的函数*****/
30. void OSCtxSw(void); //用户任务切换
31. void OSIntCtxSw(void); //中断任务切换函数
32. void OSStartHighRdy(void); //在操作系统第一次启动的时候调用的任务切换
33.
34. void OS_CPU_PendSVHandler(void); //用户中断处理函数，旧版本为 OSPendSV
35.
36. void OS_CPU_SysTickHandler(void); //系统定时中断处理函数，时钟节拍函数
37. void OS_CPU_SysTickInit(void); //系统 SysTick 定时器初始化
38.
39. 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，利用到异常处理知识，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。

2.2 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：

40. OSInitHookBegin()	//OSInit()	系统初始化函数开头的钩子函数
41. OSInitHookEnd()	//OSInit()	系统初始化函数结尾的钩子函数
42. OSTaskCreateHook()	//OSTaskCreate() 或 OSTaskCreateExt()	创建任务钩子函数
43. OSTaskDelHook()	//OSTaskDel()	删除任务钩子函数
44. OSTaskIdleHook()	//OS_TaskIdle()	空闲任务钩子函数
45. OSTaskStatHook()	//OSTaskStat()	统计任务钩子函数
46. OSTaskSwHook()	//OSTaskSW()	任务切换钩子函数
47. OSTCBInitHook()	//OS_TCBInit()	任务控制块初始化钩子函数
48. 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：

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

钩子函数的编写，例如：

```
50. /*** 系统初始化函数 OSInit() 开头调用 ***/
51. void OSInitHookBegin(void)
52. {
53.     #if OS_TMR_EN > 0 //当使用 OS_TMR.C 定时器管理模块
54.         OSTmrCtr = 0; //初始化系统节拍计数变量 OSTmrCtr 为 0
55.         //每个时钟节拍 OSTmrCtr（全局变量，初始值为 0）增 1
```

```
56. #endif
57. }
```

```
58. /*** 创建任务 OSTaskCreate() 或 OSTaskCreateExt() 中调用 ***/
59. void OSTaskCreateHook (OS_TCB *ptcb)
60. {
61.     #if OS_APP_HOOKS_EN > 0           //如果有定义应用任务
62.         App_TaskCreateHook (ptcb);    //调用应用任务创建钩子函数
63.     #else                             //否则
64.         (void)ptcb;                  //告诉编译器 ptcb 没用到
65. #endif
66. }
```

```
67. /*** 切换任务时被调用 ***/
68. void OSTaskSwHook (void)
69. {
70.     #if OS_APP_HOOKS_EN > 0
71.         App_TaskSwHook();             //应用任务切换时调用的钩子函数
72.     #endif
73. }
```

```
74. /*** 每个系统节拍到了 ***/
75. void OSTimeTickHook (void)
76. {
77.     #if OS_APP_HOOKS_EN > 0
78.         App_TimeTickHook();           //应用软件的时钟节拍钩子
79.     #endif
80.
81.     #if OS_TMR_EN > 0                 //如果有启动定时器管理
82.         OSTmrCtr++;                  //计时变量 OSTmrCtr 加 1
83.         if (OSTmrCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC)) { //如果时间到了
84.             OSTmrCtr = 0;            //计时清 0
85.             OSTmrSignal();            //发送信号量 OSTmrSemSignal (初始值为 0)
86.             //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
87.         }
88.     #endif
89. }
```

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

```
90. 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：

```
91. void MyTask (void *p_arg)
92. {
93.     /* 可选，例如处理 'p_arg' 变量 */
94.     while (1) {
95.         /* 任务主体 */
96.     }
97. }
```

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。

```
98. OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg,  
99.                        OS_STK *ptos, INT16U opt)  
100. {  
101.     OS_STK *stk;  
102.  
103.  
104.     (void)opt; // 'opt' 并没有用到，防止编译器提示警告  
105.     stk = ptos; // 加载栈指针  
106.  
107.     /* 中断后 xPSR, PC, LR, R12, R3-R0 被自动保存到栈中*/  
108.     *(stk) = (INT32U)0x01000000L; // xPSR  
109.     *--stk = (INT32U)task; // 任务入口 (PC)  
110.     *--stk = (INT32U)0xFFFFFFFFL; // R14 (LR)  
111.     *--stk = (INT32U)0x12121212L; // R12  
112.     *--stk = (INT32U)0x03030303L; // R3  
113.     *--stk = (INT32U)0x02020202L; // R2  
114.     *--stk = (INT32U)0x01010101L; // R1  
115.     *--stk = (INT32U)p_arg; // R0 : 变量  
116.  
117.     /* 剩下的寄存器需要手动保存在堆栈 */  
118.     *--stk = (INT32U)0x11111111L; // R11  
119.     *--stk = (INT32U)0x10101010L; // R10  
120.     *--stk = (INT32U)0x09090909L; // R9  
121.     *--stk = (INT32U)0x08080808L; // R8  
122.     *--stk = (INT32U)0x07070707L; // R7  
123.     *--stk = (INT32U)0x06060606L; // R6  
124.     *--stk = (INT32U)0x05050505L; // R5  
125.     *--stk = (INT32U)0x04040404L; // R4  
126.  
127.     return (stk);  
128. }
```

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。

问大家两个问题，以便大家知道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：

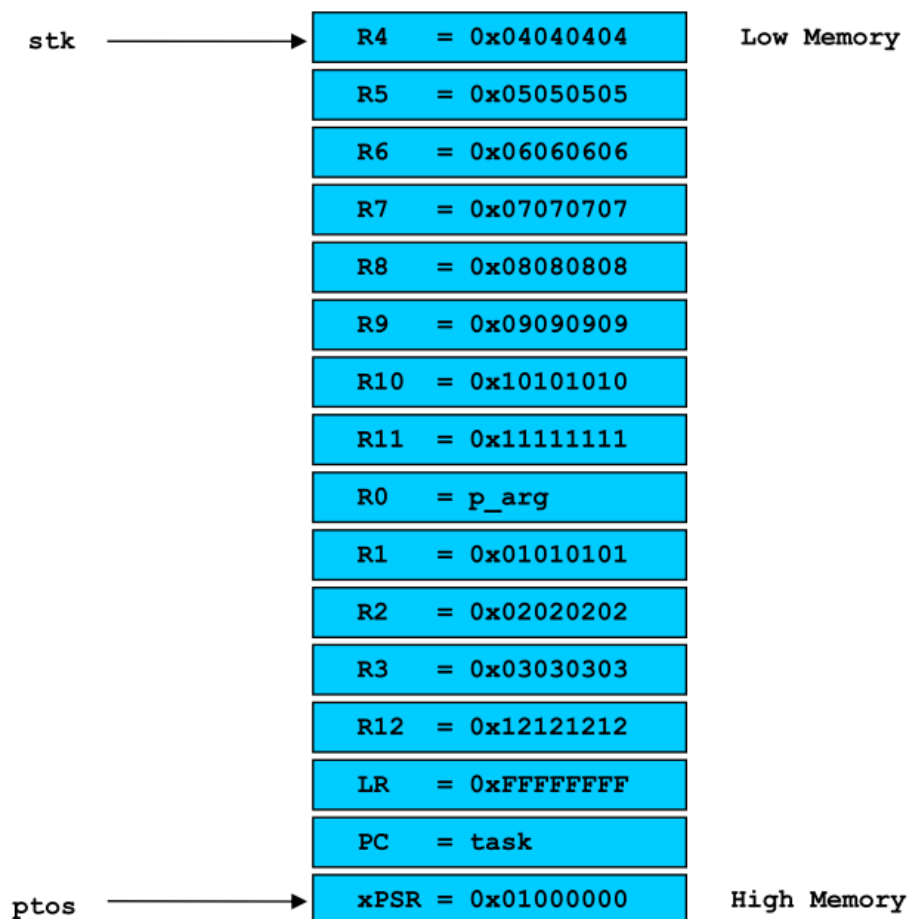


Figure 3-2, The Stack Frame for each Task for ARM Cortex-M3 port.

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 = 0xFFFFFFFFEL**, 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。

```

129. void OS_CPU_SysTickInit (void)
130. {
131.     INT32U cnts;
132.
133.
134.     cnts = OS_CPU_SysTickClkFreq() / OS_TICKS_PER_SEC;
135.     //OS_CPU_SysTickClkFreq() 获取时钟频率
136.     //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
137.
138.     /* 使能 SysTick 定时器 */
139.     OS_CPU_CM3_NVIC_ST_RELOAD = (cnts - 1);
140.
141.     /* 使能 SysTick 定时器中断 */
142.     OS_CPU_CM3_NVIC_ST_CTRL |= OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC

```

```

143.             | OS_CPU_CM3_NVIC_ST_CTRL_ENABLE;
144.
145. OS_CPU_CM3_NVIC_ST_CTRL |= OS_CPU_CM3_NVIC_ST_CTRL_INTEN;
146. }

```

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：

```

147. /*****
148. *                               SYS TICK DEFINES
149. *****/
150. #define OS_CPU_CM3_NVIC_ST_CTRL      (*((volatile INT32U *)0xE000E010))
151.                                     /* SysTick Ctrl & Status Reg. */
152. #define OS_CPU_CM3_NVIC_ST_RELOAD    (*((volatile INT32U *)0xE000E014))
153.                                     /* SysTick Reload Value Reg. */
154. #define OS_CPU_CM3_NVIC_ST_CURRENT  (*((volatile INT32U *)0xE000E018))
155.                                     /* SysTick Current Value Reg. */
156. #define OS_CPU_CM3_NVIC_ST_CAL       (*((volatile INT32U *)0xE000E01C))
157.                                     /* SysTick Cal Value Reg. */
158. #define OS_CPU_CM3_NVIC_PRIO_ST      (*((volatile INT8U *)0xE000ED23))
159.                                     /* SysTick Handler Prio Reg. */
160.
161. #define OS_CPU_CM3_NVIC_ST_CTRL_COUNT      0x00010000
162.                                     /* Count flag. */
163. #define OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC    0x00000004
164.                                     /* Clock Source. */
165. #define OS_CPU_CM3_NVIC_ST_CTRL_INTEN      0x00000002
166.                                     /* Interrupt enable. */
167. #define OS_CPU_CM3_NVIC_ST_CTRL_ENABLE     0x00000001
168.                                     /* Counter mode. */
169. #define OS_CPU_CM3_NVIC_PRIO_MIN           0xFF
170.                                     /* 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。

2.3 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

```

171. EXTERN  OSRunning                ; 声明外部定义，相当于 C 语言的 extern
172. EXTERN OSPrioCur
173. EXTERN OSPrioHighRdy
174. EXTERN OSTCBCur
175. EXTERN OSTCBHighRdy
176. EXTERN OSIntNesting
177. EXTERN OSIntExit
178. 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 PUBLIC 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）

```

179. PUBLIC OS_CPU_SR_Save                ; 声明函数在此文件定义
180. PUBLIC OS_CPU_SR_Restore
181. PUBLIC OSStartHighRdy
182. PUBLIC OSCtxSw
183. PUBLIC OSIntCtxSw
184. PUBLIC OS_CPU_PendSVHandler

```

修改后

```
185. EXPORT OS_CPU_SR_Save ; 声明函数在此文件定义
186. EXPORT OS_CPU_SR_Restore
187. EXPORT OSStartHighRdy
188. EXPORT OSCtxSw
189. EXPORT OSIntCtxSw
190. 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：

```
191. 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。
192. ; NOROOT 表示：如果这段中的代码没调用，则允许连接器丢弃这段
193. ; (2) 表示：4 字节对齐。假如是 (n)，则表示 2^n 对齐
```

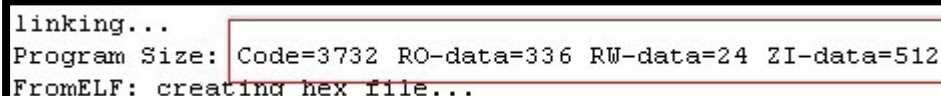
替换为：

```
194. AREA |.text|, CODE, READONLY, ALIGN=2 ;AREA |.text| 表示：选择段 |.text|。
195. ;CODE 表示代码段，READONLY 表示只读（缺省）
196. ;ALIGN=2 表示 4 字节对齐。若 ALIGN=n, 这 2^n 对齐
197. THUMB ;Thumb 代码
198. REQUIRE8 ;指定当前文件要求堆栈八字节对齐
199. 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**）初始化了的读写变量的大小，**ZI-data** 是（**Zero Initialize**）没有初始化的读写变量的大小。**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。

```
200. NVIC_INT_CTRL    EQU    0xE00ED04 ; 中断控制及状态寄存器 ICSR 的地址
201.                  ; 见《Cortex-M3 权威指南》第 8.4.5 小节 表 8.5)
202. NVIC_SYSPRI14    EQU    0xE00ED22 ; 系统异常优先级寄存器 PRI_14
203.                  ; 即设置 PendSV 的优先级
204.                  ; 见《Cortex-M3 权威指南》第 8.4.2 小节 表 8.3B
205. NVIC_PENDSV_PRI   EQU    0xFF ; 定义 PendSV 的可编程优先级为 255, 即最低
206.                  ; 为啥是最低呢? 大家思考一下
207. NVIC_PENDSVSET    EQU    0x10000000 ; 中断控制及状态寄存器 ICSR 的位 28
208.                  ; 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:

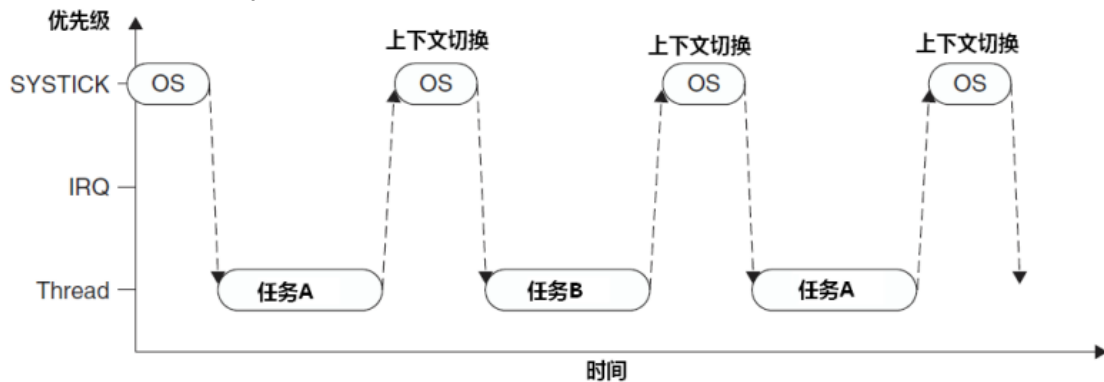
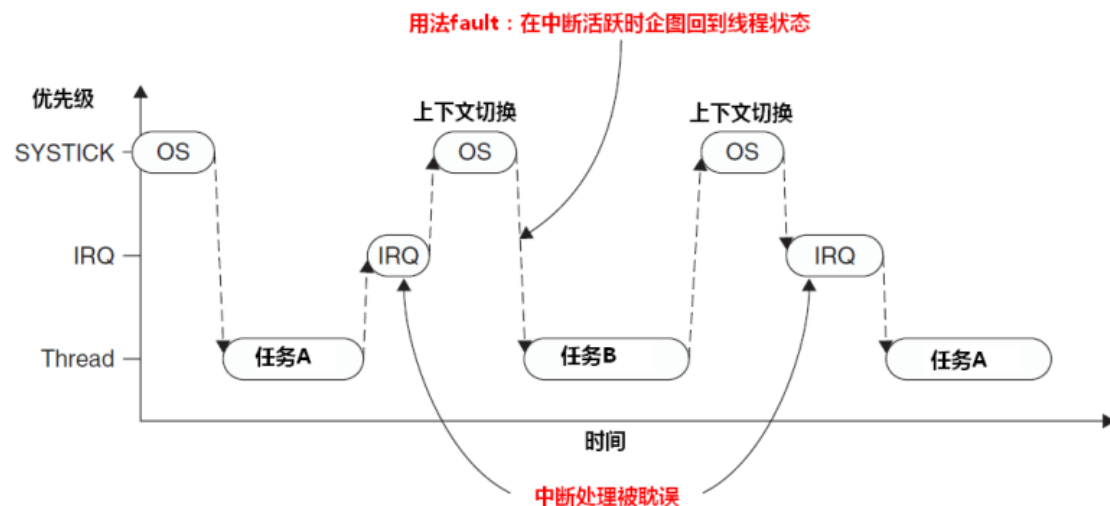
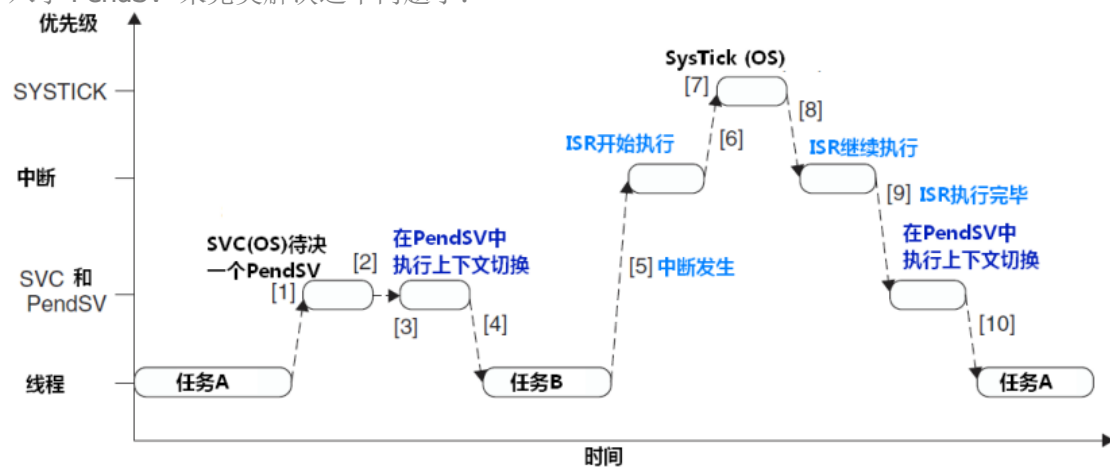


图 7.15 两个任务间通过 SysTick 进行轮转调度的简单模式

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：

```

209.; OS_ENTER_CRITICAL() 里进入临界段调用，保存现场环境
210.OS_CPU_SR_Save
211.    MRS    R0, PRIMASK    ; 读取 PRIMASK 到 R0 (保存全局中断标记，除了故障中断)
212.    CPSID   I              ; PRIMASK=1，关中断
213.    BX      LR              ; 返回，返回值保存在 R0
214.
215.
216.; OS_EXIT_CRITICAL() 里退出临界段调用，恢复现场环境
217.OS_CPU_SR_Restore
218.    MSR     PRIMASK, R0     ; 读取 R0 到 PRIMASK 中 (恢复全局中断标记)，通过 R0 传递参数
219.    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。

```
220. OSStartHighRdy
221.     LDR    R0, =NVIC_SYSPRI14      ; 装载 系统异常优先级寄存器 PRI_14
222.                                     ; 即设置 PendSV 中断优先级的寄存器
223.     LDR    R1, =NVIC_PENDSV_PRI    ; 装载 PendSV 的可编程优先级 (255)
224.     STRB   R1, [R0]                ; 无符号字节寄存器存储。R1 是要存储的寄存器
225.                                     ; 存储到内存地址所基于的寄存器
226.                                     ; 即设置 PendSV 中断优先级为 255
227.
228.     MOV     R0, #0                  ; 把数值 0 复制到 R0 寄存器
229.     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。
230.
231.     LDR     R0, _OS_Running         ; OSRunning = TRUE
232.     MOV     R1, #1
233.     STRB   R1, [R0]
234.
235.     LDR     R0, =NVIC_INT_CTRL      ; 装载 中断控制及状态寄存器 ICSR 的地址
236.     LDR     R1, =NVIC_PENDSVSET     ; 中断控制及状态寄存器 ICSR 的位 28
237.     STR     R1, [R0]                ; 设置 中断控制及状态寄存器 ICSR 位 28 为 1
238.                                     ; 以悬起 (允许) PendSV 中断
239.
240.     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。

```
241. /*****任务级调度器*****/
242. void OS_Sched (void)
243. {
244.     #if OS_CRITICAL_METHOD == 3
245.         OS_CPU_SR cpu_sr = 0;
246.     #endif
247.
248.     OS_ENTER_CRITICAL();
249.     if (OSIntNesting == 0) { //如果没中断服务运行
250.         if (OSLockNesting == 0) { //调度器没上锁
251.             OS_SchedNew(); //查找最高优先级就绪任务
252.             //见 os_core.c，会修改 OSPrioHighRdy
253.             if (OSPrioHighRdy != OSPrioCur) { //如果得到的最高优先级就绪任务不等于当前
254.                 //注：当前运行的任务也在就绪表里
255.                 OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy]; //得到任务控制块指针
256.                 #if OS_TASK_PROFILE_EN > 0
257.                     OSTCBHighRdy->OSTCBCtxSwCtr++; //统计任务切换到次任务的计数器加 1
258.                 #endif
259.                 OSTCtxtSwCtr++; //统计任务切换次数的计数器加 1
260.                 OS_TASK_SW(); //进行任务切换
261.             }
262.         }
263.     }
264.     OS_EXIT_CRITICAL(); //退出临界段，开中断
265. }
```

悬起(允许)PendSV 中断

开中断，执行 PendSV 中断服务

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

```
266. #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。

```
267. OSCtxSw
268.     ; 悬起(允许)PendSV 中断 (看不懂这段代码的，可参考前面见过的 OSStartHighRdy)
269.     LDR    R0, =NVIC_INT_CTRL           ; 装载 中断控制及状态寄存器 ICSR 的地址
270.     LDR    R1, =NVIC_PENDSVSET         ; 中断控制及状态寄存器 ICSR 的位 28
271.     STR    R1, [R0]                   ; 设置 中断控制及状态寄存器 ICSR 位 28 为 1
272.                                           ; 以悬起(允许)PendSV 中断
273.     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 OSIntCtxSw 完全相同的：

```
274. OSIntCtxSw
275.     LDR    R0, =NVIC_INT_CTRL           ; trigger the PendSV exception
276.     LDR    R1, =NVIC_PENDSVSET
277.     STR    R1, [R0]
278.     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：

```
279. //OS_CPU_PendSVHandler 伪代码思路
280. OS_CPU_PendSVHandler:
281.   if (PSP != NULL) {           //当调用 OS_CPU_PendSVHandler() 时，
282.                                   //CPU 就会自动保存 xPSR、PC、LR、R12、R0-R3 寄存器到堆栈
283.                                   //保存后，CUP 的栈 SP 指针会切换到使用主堆栈指针 MSP 上
284.                                   //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
285.                                   //因此当我们第一次启动任务是,OSStartHighRdy() 就把 PSP 设为 NULL，
286.                                   //避免系统以为已经进行任务切换
287.       Save R4-R11 onto task stack; //手动保存 R4-R11
288.       OSTCBCur->OSTCBStkPtr = SP;  //保存进入栈指针 PSP 到任务控制块
289.                                   //以便下次继续任务运行时继续使用原来的栈
290.   }
291.   OSTaskSwHook();                 //此处便于我们使用钩子函数来拓展功能
292.   OSPrioCur = OSPrioHighRdy;     //获取最高优先级就绪任务的优先级
293.   OSTCBCur = OSTCBHighRdy;        //获取最高优先级就绪任务的任务控制块指针
294.   PSP = OSTCBHighRdy->OSTCBStkPtr; //保存进入栈指针
295.   Restore R4-R11 from new task stack; //从新的栈恢复 R4-R11 寄存器
296.   Return from exception;           //返回
```

具体的汇编代码：

```
297. OS_CPU_PendSVHandler                ;CPU 会自动保存 xPSR, PC, LR, R12, R0-R3
298.     CPSID     I                      ;关中断
299.     MRS       R0, PSP                 ;PSP 就是栈指针, R0=PSP
300.     CBZ       R0, OSPendSV_nosave    ;当 PSP==0, 执行 OSPendSV_nosave 函数
301.
302.     SUB       R0, R0, #0x20           ;装载 r4-11 到栈, 共 8 个寄存器, 32 位, 4 个字节
303.                                           ;即 8*4=32=0x20
304.     STM       R0, {R4-R11}           ;
305.
306.     LDR       R1, __OS_TCBCur        ;R1=&OSTCBCur
307.     LDR       R1, [R1]               ;R1=*R1 (R1=OSTCBCur)
308.     STR       R0, [R1]               ;*R1=R0 (*OSTCBCur=SP)
309.
310. OSPendSV_nosave
311.     PUSH      {R14}                 ;保存 R14
312.     LDR       R0, __OS_TaskSwHook    ;调用钩子函数 OSTaskSwHook()
313.     BLX       R0
314.     POP       {R14}                 ;恢复 R14
315.
316.     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
317.                                           ;OSPrioCur = OSPrioHighRdy
318.     LDR       R1, __OS_PrioHighRdy
319.     LDRB      R2, [R1]
320.     STRB      R2, [R0]
321.
```

```

322.    LDR    R0, __OS_TCBCur      ;设置当前任务控制块指针
323.    LDR    R1, __OS_TCBHighRdy ;OSTCBCur = OSTCBHighRdy
324.    LDR    R2, [R1]
325.    STR    R2, [R0]
326.
327.    LDR    R0, [R2]              ;R0 是新的 SP
328.                                ;SP = OSTCBHighRdy->OSTCBStkPtr;
329.
330.    LDM     R0, {R4-R11}          ;从新的栈恢复 R4-R11
331.    ADD     R0, R0, #0x20
332.    MSR     PSP, R0               ;PSP=R0,用新的栈 SP 加载 PSP
333.    ORR     LR, LR, #0x04         ;确保 LR 位 2 为 1, 返回到使用进程堆栈
334.    CPSIE   I                    ;开中断
335.    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：

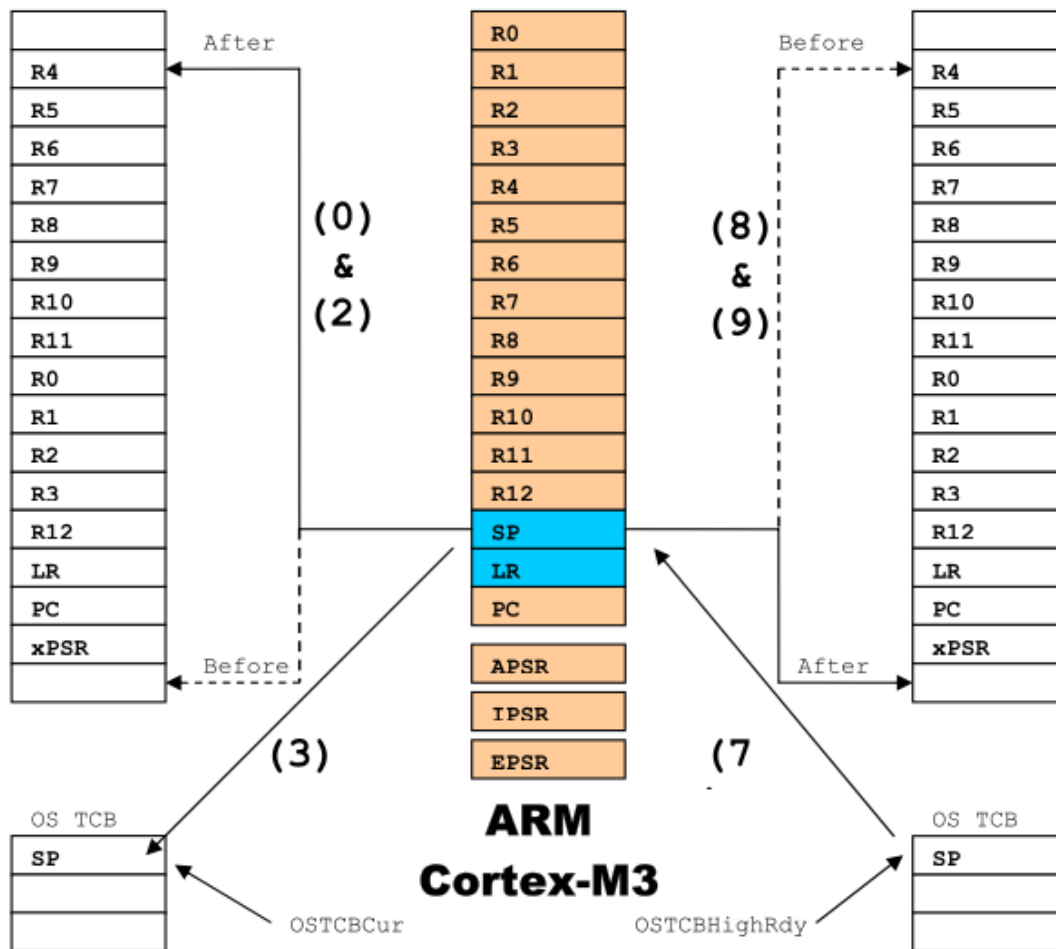
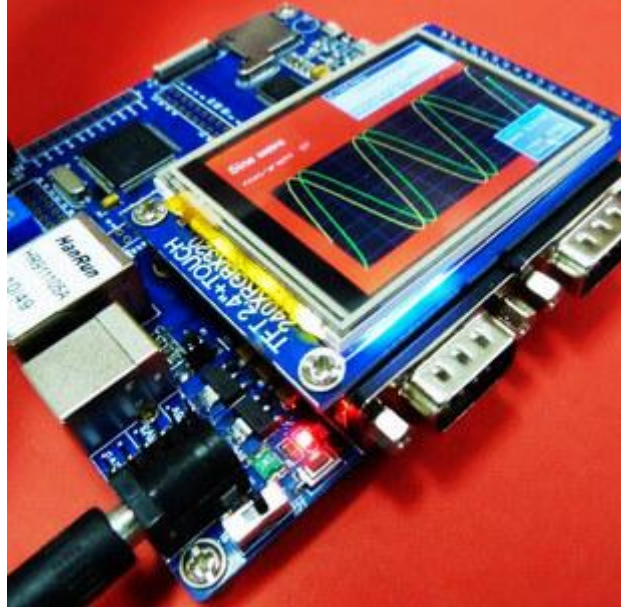


Figure 3-3, ARM Cortex-M3 Context Switch.

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！多看几次就弄懂的！**限于个人能力，欢迎各位高手指出错误，在此先表达感谢！**

第三章 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：

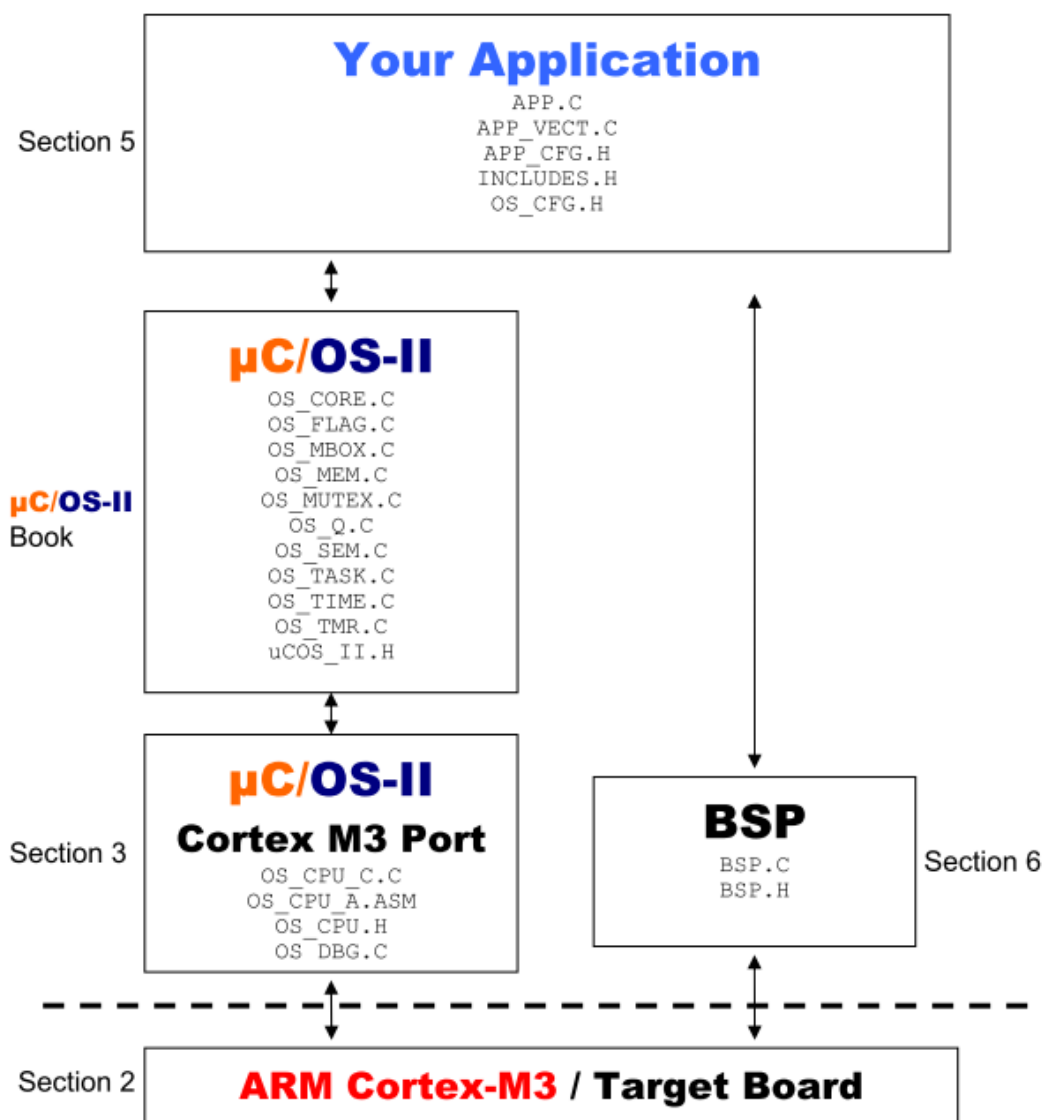


Figure 1-1, Relationship between modules.

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）：

3.1 打开 LED 工程模版

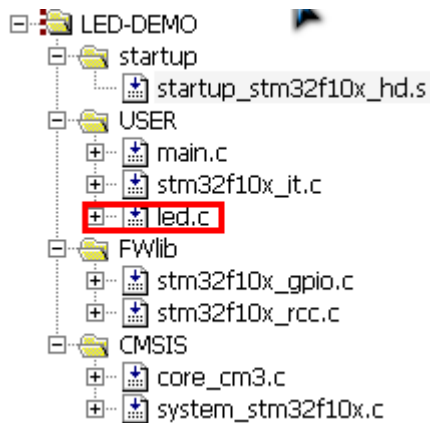
首先，我们从野火 STM32 光盘资料那里提取 LED 实验：



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：

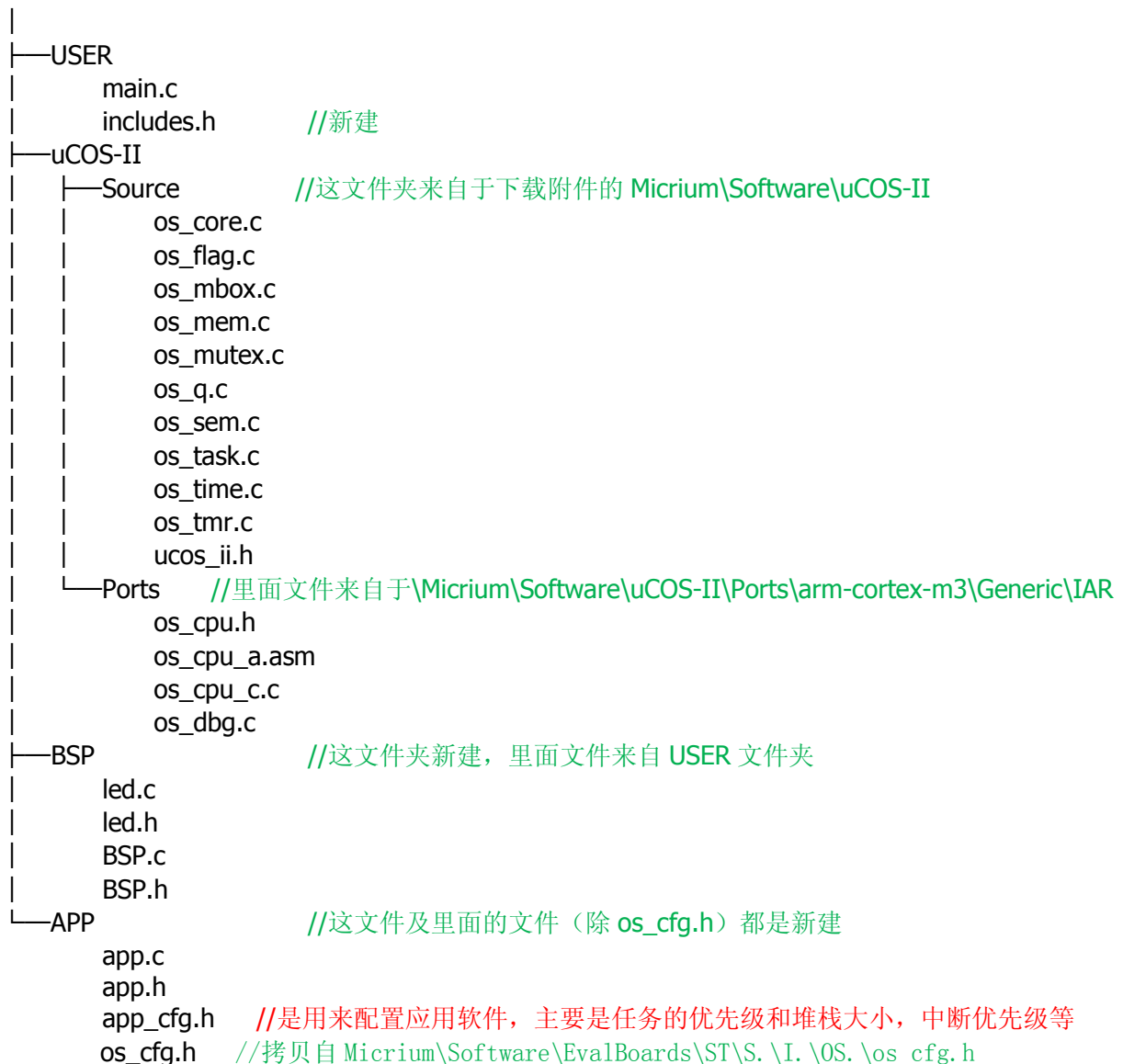


这个是我们过往开发裸机单片机程序时写的工程文件结构，但对于 uC/OS-II，或者其他大型点的软件工程，这样的文件结构就会很乱的。

3.2 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



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。

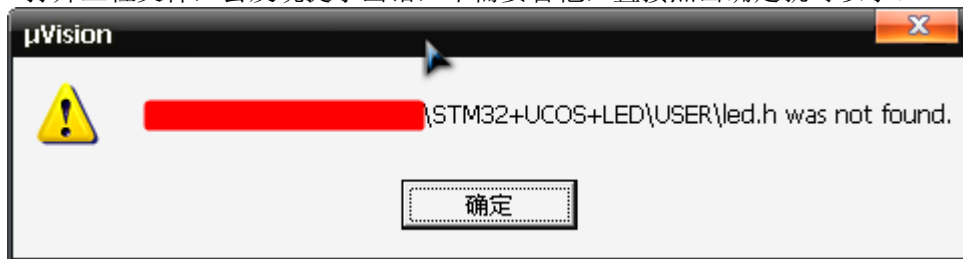
拷贝 uC/OS-II 源代码附件那里的 Micrium\Software\EvalBoards\ST\STM32F103ZE-SK\IAR\OS-Probe-LCD\os_cfg.h 到此目录。

- 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。

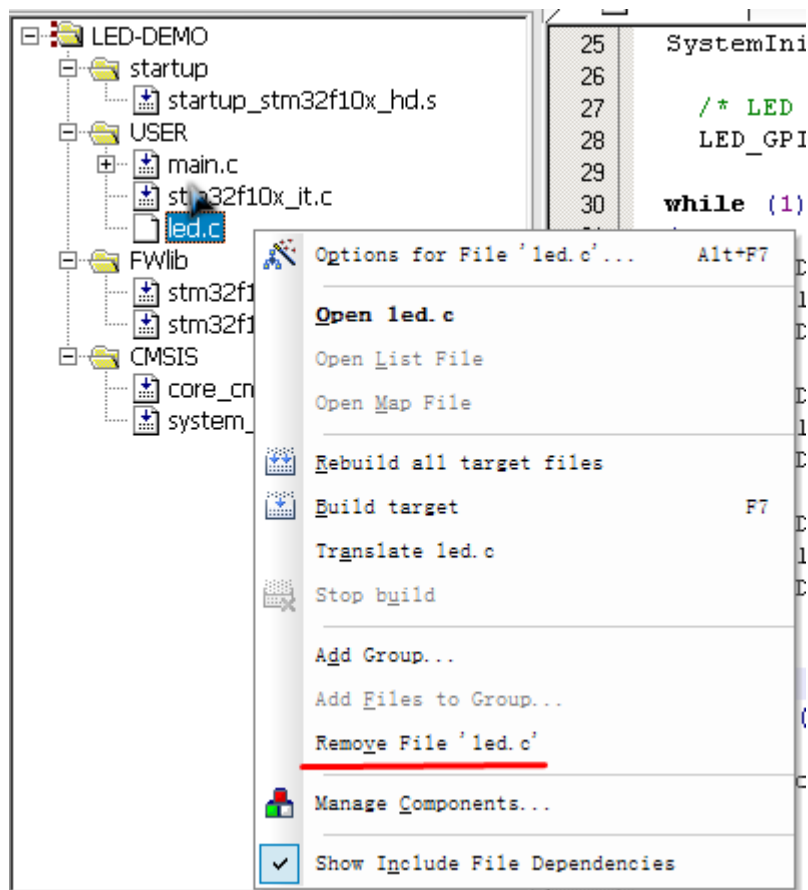
把 Micrium\Software\uCOS-II\Ports\arm-cortex-m3\Generic\IAR 下的文件复制到工程 uCOS-II 文件夹中新建的 Ports 文件夹里。复制后，选中全部文件，右键——属性——去掉只读属性——确定。

到此，工程的目录结构就建立好了，需要修改工程设置。

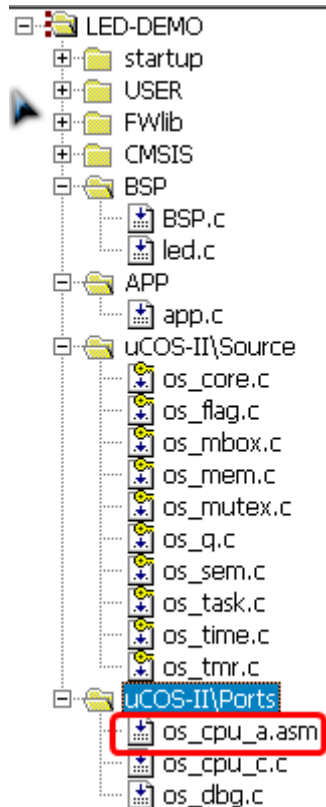
- ⑦ 打开工程文件，会发现提示出错，不需要管他，直接点击确定就可以了。



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：

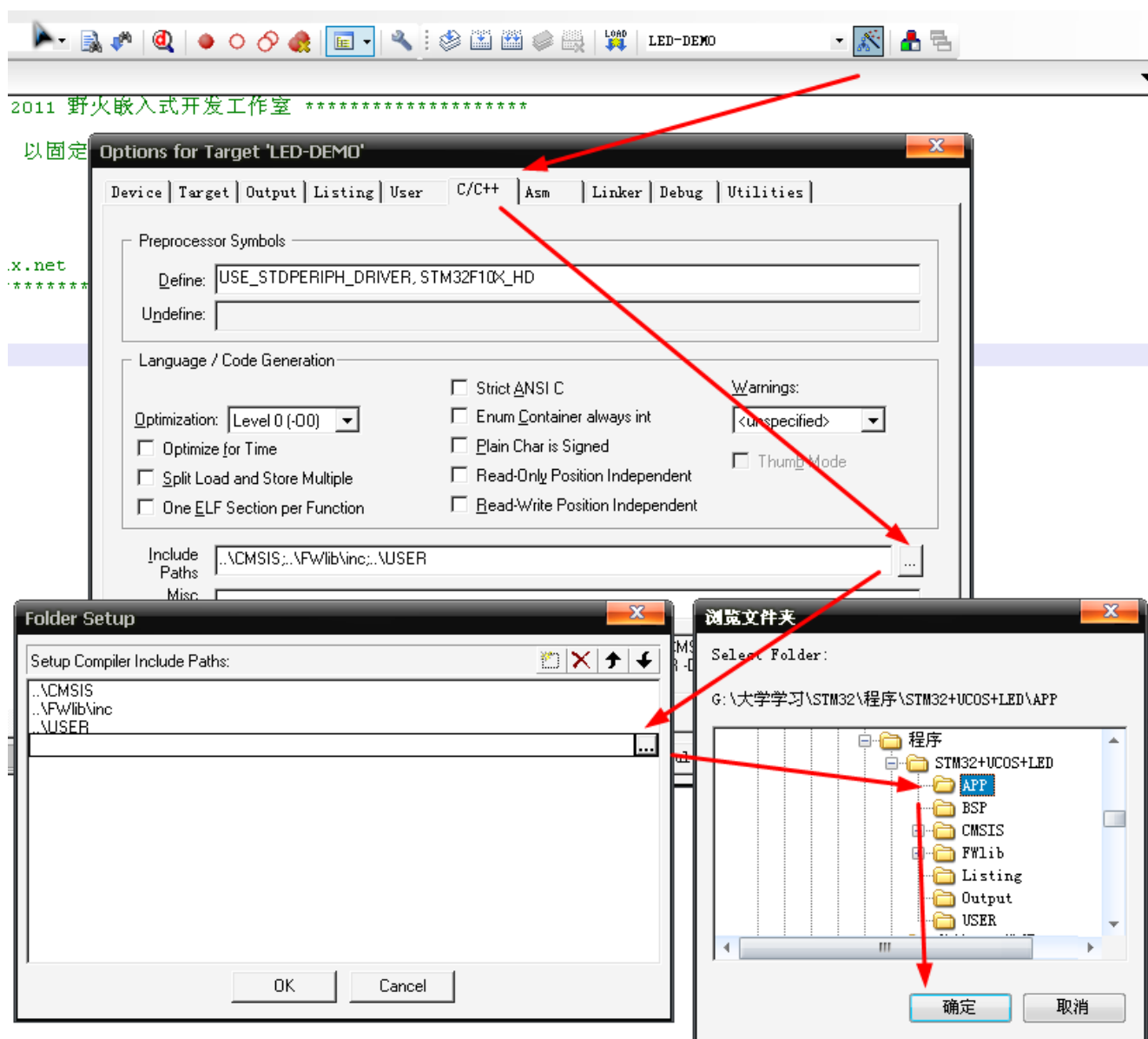


建立 BSP、APP 和 uCOS-II 下两个文件夹，即共四个文件夹的组，并添加进相应的文件：



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！

3.3 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	
		内存管理	OS_MBOX_QUERY_EN	
			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	
		时间管理	OS_SEM_SET_EN	
			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：

```

336. #define OS_FLAG_EN          0    //禁用信号量集
337. #define OS_MBOX_EN          0    //禁用邮箱
338. #define OS_MEM_EN           0    //禁用内存管理
339. #define OS_MUTEX_EN         0    //禁用互斥信号量
340. #define OS_Q_EN             0    //禁用队列
341. #define OS_SEM_EN           0    //禁用信号量
342. #define OS_TMR_EN           0    //禁用定时器
343. #define OS_DEBUG_EN         0    //禁用调试

```

- ② 现在也用不着应用程序的钩子函数，也禁掉；多重事件控制也禁掉

```

344. #define OS_APP_HOOKS_EN      0
345. #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。

3.4 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。

3.5 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）

3.6 修改 os_cpu_a.asm

由于编译器的原因：

要将下面的 PUBLIC 改为 EXPORT

即：

```
346. PUBLIC OS_CPU_SR_Save ; Functions declared in this file
347. PUBLIC OS_CPU_SR_Restore
348. PUBLIC OSStartHighRdy
349. PUBLIC OSCtxSw
350. PUBLIC OSIntCtxSw
351. PUBLIC OS_CPU_PendSVHandler
```

改为：

```

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

```

下面这个也要修改下
原来的:

```

358.     RSEG CODE:CODE:NOROOT (2)

```

修改后:

```

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

```

3.7 修改 os_dbg.c

将 os_dbg.c 中

```

365. #define OS_COMPILER_OPT __root

```

修改为:

```

366. #define OS_COMPILER_OPT // __root

```

这个问题也是由编译器不同而产生的。

3.8 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。

3.9 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头文件,以便让大家知道我们的工程是怎样的一个架构。

```

367. #ifndef __INCLUDES_H__
368. #define __INCLUDES_H__
369.
370. #include "stm32f10x.h"
371. #include "stm32f10x_rcc.h" //SysTick 定时器相关
372.
373. #include "ucos_ii.h" //uC/OS-II 系统函数头文件
374.
375. #include "BSP.h" //与开发板相关的函数

```

```
376. #include    "app.h"           //用户任务函数
377. #include    "led.h"           //LED 驱动函数
378.
379. #endif // __INCLUDES_H__
```

3.10 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

```
380. #include "includes.h"
381.
382. /*
383. * 函数名: BSP_Init
384. * 描述   : 时钟初始化、硬件初始化
385. * 输入   : 无
386. * 输出   : 无
387. */
388. void BSP_Init(void)
389. {
390.     SystemInit();           /* 配置系统时钟为 72M */
391.     SysTick_init();         /* 初始化并使能 SysTick 定时器 */
392.     LED_GPIO_Config();      /* LED 端口初始化 */
393. }
394.
395. /*
396. * 函数名: SysTick_init
397. * 描述   : 配置 SysTick 定时器
398. * 输入   : 无
399. * 输出   : 无
400. */
401. void SysTick_init(void)
402. {
403.     SysTick_Config(SystemFrequency/OS_TICKS_PER_SEC); //初始化并使能 SysTick 定时器
404. }
```

BSP.h 头文件

```
405.
406. #ifndef __BSP_H
407. #define __BSP_H
408.
409. void SysTick_init(void);
410. void BSP_Init(void);
411.
412. #endif // __BSP_H
```

3.11 编写 stm32f10x_it.c

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

```
413. void SysTick_Handler(void)
414. {
415.     OSIntEnter();
416.     OSTimeTick();
417.     OSIntExit();
418. }
```


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。

3.12 创建任务

编写 app_cfg.h

用来设置任务的优先级和栈大小

```
419. #ifndef APP_CFG_H
420. #define APP_CFG_H
421.
422. /*****设置任务优先级*****/
423. #define STARTUP_TASK_PRIO 4
424.
425. /*****设置栈大小（单位为 OS_STK）*****/
426. #define STARTUP_TASK_STK_SIZE 80
427.
428. #endif
```

编写 app.c

这个是创建 LED 显示任务

```
429. #include "includes.h"
430.
431. void Task_LED(void *p_arg)
432. {
433.     (void)p_arg; // 'p_arg' 并没有用到，防止编译器提示警告
434.     while (1)
435.     {
436.         LED1( ON );
437.         OSTimeDlyHMSM(0, 0, 0, 500);
438.         LED1( OFF );
439.
440.         LED2( ON );
441.         OSTimeDlyHMSM(0, 0, 0, 500);
442.         LED2( OFF );
443.
444.         LED3( ON );
445.         OSTimeDlyHMSM(0, 0, 0, 500);
446.         LED3( OFF );
447.     }
448. }
```

编写 app.h 头文件

```
449. #ifndef APP_H
450. #define APP_H
451.
452. /***** 用户任务声明 *****/
453. void Task_LED(void *p_arg);
454.
455. #endif // APP_H
```

3.13 main 函数

```
456. #include "includes.h"
457.
458. static OS_STK startup_task_stk[STARTUP_TASK_STK_SIZE]; //定义栈
459.
460. int main(void)
461. {
462.     BSP_Init();
```

```

463.     OSInit();
464.     OSTaskCreate(Task_LED, (void *)0,
465.         &startup_task_stk[STARTUP_TASK_STK_SIZE-1], STARTUP_TASK_PRIO);
466.
467.     OSStart();
468.     return 0;
469. }

```

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？

第四章 运行多任务

移植 uC/OS-II 弄好了，那运行多任务更简单。

注意哦，这里说的单任务、多任务是对于用户任务而言的。刚才那个实验，实际上也有两个任务，一个我们创建的 LED 任务，另外一个系统是自带的空闲任务。

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：

4.1 修改 app.c

```

470. #include "includes.h"
471.
472. OS_STK task_led2_stk[TASK_LED2_STK_SIZE];           //定义栈
473. OS_STK task_led3_stk[TASK_LED3_STK_SIZE];           //定义栈
474.
475. //主任务
476. void Task_Start(void *p_arg)
477. {
478.     (void)p_arg;                                       // 'p_arg' 并没有用到，防止编译器提示警告
479.
480.     OSTaskCreate(Task_LED2, (void *)0,               //创建任务 2
481.         &task_led2_stk[TASK_LED2_STK_SIZE-1], TASK_LED2_PRIO);
482.
483.     OSTaskCreate(Task_LED3, (void *)0,               //创建任务 3
484.         &task_led3_stk[TASK_LED3_STK_SIZE-1], TASK_LED3_PRIO);
485.
486.     while (1)
487.     {
488.         LED1( ON );
489.         OSTimeDlyHMSM(0, 0,0,100);
490.         LED1( OFF);
491.         OSTimeDlyHMSM(0, 0,0,100);
492.     }
493. }
494.
495. //任务 2
496. void Task_LED2(void *p_arg)
497. {
498.     (void)p_arg;

```

```

499.
500.     while (1)
501.     {
502.         LED2( ON );
503.         OSTimeDlyHMSM(0, 0,0,200);
504.         LED2( OFF);
505.         OSTimeDlyHMSM(0, 0,0,200);
506.     }
507. }
508.
509. //任务3
510. void Task_LED3(void *p_arg)
511. {
512.     (void)p_arg;
513.
514.     while (1)
515.     {
516.         LED3( ON );
517.         OSTimeDlyHMSM(0, 0,0,300);
518.         LED3( OFF);
519.         OSTimeDlyHMSM(0, 0,0,300);
520.     }
521. }

```

4.2 编写 app.h

```

522. #ifndef _APP_H_
523. #define _APP_H_
524.
525. /***** 用户任务声明 *****/
526. void Task_Start(void *p_arg);
527. void Task_LED2(void *p_arg);
528. void Task_LED3(void *p_arg);
529. #endif // _APP_H_

```

4.3 编写 app_cfg.h

```

530. #ifndef _APP_CFG_H_
531. #define _APP_CFG_H_
532.
533.
534. /***** 设置任务优先级 *****/
535. #define STARTUP_TASK_PRIO      4
536. #define TASK_LED2_PRIO        5
537. #define TASK_LED3_PRIO        6
538.
539. /***** 设置栈大小（单位为 OS_STK） *****/
540. #define STARTUP_TASK_STK_SIZE  80
541. #define TASK_LED2_STK_SIZE     80
542. #define TASK_LED3_STK_SIZE     80
543.
544. #endif

```

4.4 编写 main.c

```

545. /***** (C) COPYRIGHT 2011 野火嵌入式开发工作室 *****/
546. * 文件名   : main.c
547. * 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）。
548. * 实验平台：野火STM32开发板
549. * 库版本   : ST3.0.0
550. *
551. * 作者     : fire QQ: 313303034
552. * 博客     : firestm32.blog.chinaunix.net

```

```

553. *****/
554. #include "includes.h"
555.
556. OS_STK startup_task_stk[STARTUP_TASK_STK_SIZE];           //定义栈
557.
558. int main(void)
559. {
560.     BSP_Init();
561.     OSInit();
562.     OSTaskCreate(Task_Start, (void *)0,
563.         &startup_task_stk[STARTUP_TASK_STK_SIZE-1], STARTUP_TASK_PRIO);
564.
565.     OSStart();
566.     return 0;
567. }
568.
569. /***** (C) COPYRIGHT 2011 野火嵌入式开发工作室 *****END OF FILE*****/

```

编译一下，下载到开发板，就可以看到 3 个 LED 以不同频率闪烁

这个实验是按葫芦画瓢的，应该容易明白吧？呵呵，这次的教程就到这里结束。

第五章 升级到最新版本 v2.90

5.1 不得不说的这些话儿

写这小节之前，首先非常感谢网友：i55x 提出的批评。

详情请看：批判一下“野火开源《从 0 开始移植 uc/OS 到野火 stm32 开发板》pdf 教程”
http://www.ourdev.cn/bbs/bbs_content.jsp?bbs_sn=5252775&bbs_page_no=1&bbs_id=1008

呵呵，网友提出的批评确实非常有道理的。👍

- ① 我们的教程选用的 uC/OS-II 版本为 V2.86，这个版本确实是有 Debug 的，所以网友提出的批评是正确的。

不过，我也解释一下为啥要选用 V2.86 这个版本：因为我们这个是教程，官方已经提供了这个版本给我们参考，我们就可以轻松讲解给初学者，以便初学者容易入门。作为初学者，不必追求完美的版本，而是追求通俗易懂的版本入门。事实上，我们本来就有打算写更新到最新版本的教程，只是限于时间的关系，没来得及写。

- ② 我们使用的工作是 MDK，这个确实在调试 uC/OS-II 的时候非常痛苦，尤其是面对栈溢出问题。另外，我们在使用 MDK 优化的时候，也发现 MDK 的优化有问题，导致函数结果与不开优化时不一样。

我们也知道 IAR 有 uC/OS plug-in，正在考虑是否换成 IAR。因为我们之前的教程一直围绕着 MDK 来讲的，如果换成 IAR，相信不少的朋友会不习惯。

真的，非常感谢网友：i55x 提出的批评。也欢迎各位高手提出批评和建议，没有你们的批评，我们就不会有进步！！！！

好了，现在回归正题。

5.2 从 v2.86 升级到 v2.90 说明

前面已经讲过，移植的时候，我们只需修改几个文件即可。升级其实也很简单，对于原来不需要修改的文件，我们可以直接通过简单的替换就可以；对于需要修改的文件，依葫芦画瓢就行。

首先，我们需要从官网下载 uC/OS-II 的最新版本。

下载地址为：http://micrium.com/page/downloads/source_code

目前最新的版本为 V2.90，而之前我们提供的是 V2.86，前面已经说到 V2.86 是一个有 Debug 的版本，所以有必要进行升级。

下载最新版本 V2.90 后，首先当然先看下更新文档：

[Micrium-uCOS-II-V290\Micrium\Software\uCOS-II\Doc\ReleaseNotes.PDF](#)

我们的版本是 V2.86，所以从 V2.87 开始看起。事实上，我们只需看我们移植过程中需要更改的那两个文件就可以了：os_cfg.h 和 os_dbg_r.c。

os_dbg_r.c，我们暂时还没讲到，所以也可以不去看，**也就是说，我们只需看 os_cfg.h 的更新就好了**。当然，如果你要了解 uC/OS 的更新细节，其他的内容也最好看下。

v2.87 更新日志

在 V2.87 里，更新的文件有：

OS_CFG.H、OS_TIME.C、OS_CORE.C、OS_MBOX.C、OS_MUTEX.C、OS_Q.C、OS_SEM.C、OS_TASK.C、OS_TMR.C

在 OS_CFG.H 文件中，谈到了将几个 size 配置改为 EN 使能选项，可以减少内存浪费。呵呵，官方这个更新，可以为我们减少不少配置烦恼，使得配置更容易。既然更新了 OS_CFG.H，那我们就得重新配置内核。

v2.88 更新日志

在 V2.88 里，更新的文件有：

OS_CORE.C、uCOS_II.H

没修改 OS_CFG.H，跳过不看。

v2.89 更新日志

在 V2.89 里，更新的文件有：

All Files、OS_CORE.C、OS_MEM.C、OS_MUTEX.C、OS_TASK.C

All Files 中谈到，对于每个常量，后面都添加‘u’，为啥呢？

例如：

```
570. #define num 128
```

那么编译器编译的时候，默认会把 128 当成 signed 型来看待，这时，会用 int 型来保存 128。

而添加 ‘u’ 后，即

```
571. #define num 128u
```

告诉了编译器，这个是 unsigned 型，所以编译器就会用 unsigned char 来保存 128。

显然，这个更新无关重要，不影响我们更新 uC/OS-II。

v2.90 更新日志

在 V2.90 里，更新的文件有：

All Files 、 OS_CORE.C 、 uCOS_II.H 、 OS_FLAG.C 、 OS_MBOX.C 、 OS_MEM.C 、 OS_MUTEX.C 、 OS_Q.C 、 OS_SEM.C 、 OS_TASK.C 、 OS_TMR.C

All Files 中谈到，根据 MISRA-C 2004 的规则来修改所有的文件：

8.5 –不得在头文件中定义函数和变量

14.7–函数都应该有一个单点退出，即 return。

15.2–switch 中，每个非空的 case 都应该有个 break 来中止。

15.4–只有一个 case——switch 的表达不应该是一个布尔值。（只有一个，那应该用 if）

17.4–数组索引只适用于对象定义为一个数组类型

17.4 – 不应该使用指针运算

不影响 OS_CFG.H ， 原来的 OS_CFG.H 中并没有定义变量和函数。

所以，上面那么多更新，就 V2.87 更新的影响最大，但这个配置文件，我们可以直接替换后重新配置就行。

好了，该谈更新步骤了。

5.3 更新步骤

这里，我们利用前面已经做好的 STM32+UCOS+LED(单任务) 的程序来讲解如何更新到最新版本：

- ① 先把 STM32+UCOS+LED(单任务) 复制一份出来，命名为 STM32+UCOS+LED(V2.90 单任务) ， 这样避免我们以后的版本混乱。
- ② 复制下载的 uC/OS-II 新版本附件里 Micrium-uCOS-II-V290\Micrium\Software\uCOS-II\Source 文件夹下除 os_cfg_r.h 和 os_dbg_r.c 外的其他文件到我们的项目 uCOS-II\Source 文件夹下，直接替换原来已有的文件。
- ③ 复制刚才提到的 os_cfg_r.h 文件到我们的项目 APP 文件夹下
- ④ 打开 os_cfg_r.h ， 由到了配置、裁剪 uC/OS-II 的时候了。

这里，我们仅仅是简单的 LED 显示， 不需要使用通信等模块，可以这样配置：

```
572. #define OS_APP_HOOKS_EN           0u
573. #define OS_DEBUG_EN               0u
574. #define OS_EVENT_MULTI_EN        0u
575. #define OS_EVENT_NAME_EN         0u
576.
577. #define OS_TICKS_PER_SEC           1000u
578.                                     // 设置每秒中断次数，我们设置为每 1ms 中断一次比较合适
579.
580. /*裁剪其他模块*/
581. #define OS_FLAG_EN                 0u
582. #define OS_MBOX_EN                 0u
583. #define OS_MEM_EN                 0u
584. #define OS_Q_EN                   0u
585. #define OS_TMR_EN                 0u
```

保存为 os_cfg.h ， 本来就已经存在这样一个文件，所以可以直接覆盖保存。

- ⑤ 用 MDK 打开工程，编译一下，提示出错：

..\Output\STM32-DEMO.axf: Error: L6218E: Undefined symbol OSTaskReturnHook (referred from os_task.o).

这里说我们没有定义 OSTaskReturnHook，搜索一下：Edit——Find in Files——填入“OSTaskReturnHook”，找到两个地方有 OSTaskReturnHook。

一个在 uclos_ii.h 中声明（第 1313 行）：

```
586. void OSTaskReturnHook (OS_TCB *ptcb);
```

另外一个在 os_task.c 中调用（第 1206 行）

```
587. void OS_TaskReturn (void)
588. {
589.     OSTaskReturnHook (OSTCBCur);
590.
591. #if OS_TASK_DEL_EN > 0u
592.     (void)OSTaskDel (OS_PRIO_SELF);
593. #else
594.     for (;;) {
595.         OSTimeDly (OS_TICKS_PER_SEC);
596.     }
597. #endif
598. }
```

只有声明，没有定义。呵呵，又是一个钩子函数，显然，需要我们去定义。（这个前面讲移植的时候说过的哦，有些钩子函数是需要我们去定义！）

可以在 os_cpu_c.c 中添加这段代码：

```
599. #if OS_CPU_HOOKS_EN > 0u && OS_VERSION > 290u
600. void OSTaskReturnHook (OS_TCB *ptcb)
601. {
602.     (void)ptcb;
603. }
604. #endif
```

编译一下，没错了，下载到板子上验证，LED 闪烁了，呵呵，升级完毕。

说一下，关于 os_dbg_r.c，我们这里还没用到，不过也先复制到工程 uCOS-II\Ports 文件夹下，以便以后用得着的时候再去重新网上下附件就麻烦了。

5.4 更新其他工程

呵呵，前面都已经讲到，升级的步骤，其实就是复制替换到原来旧版本的文件，然后重新配置 os_cfg.h，可能需要在 os_cpu_c.c 里创建某些钩子函数。

所以，这里我们可以直接把我们升级好的 uC/OS-II 工程里的文件复制到还没升级好的工程里。需要注意的是 os_cfg.h 和 os_cpu_c.c，这部分的代码，还是需要根据工程的需要进行某些简单的修改。

更新 STM32+UCOS+LED 多任务的例子

- ① 先把 STM32+UCOS+LED(多任务) 复制一份出来，命名为 STM32+UCOS+LED(V2.90 多任务)，这样避免我们以后的版本混乱。
- ② 把 STM32+UCOS+LED(V2.90 单任务)里的 uCOS-II 文件夹整个复制到 STM32+UCOS+LED(V2.90 多任务)里，覆盖同名文件。

- ③ 打开 STM32+UCOS+LED(V2.90 多任务)\APP 下的 os_cfg.h 命名为 os_cfg_V2.86.h。这样子是为了以后方便了解原先如何配置的。当然，如果你熟悉 uC/OS-II 的配置，也可以把这步删掉。
- ④ 把 STM32+UCOS+LED(V2.90 单任务)\APP 下的 os_cfg.h 和 os_cfg_r.h 复制到 STM32+UCOS+LED(V2.90 多任务)\APP 下。
- ⑤ 然后参考 os_cfg_V2.86.h 文件，配置新的 os_cfg.h。因为这个多任务跟单任务都没用到通信这些模块，所以这里第四步直接复制过来的 os_cfg.h 就可以直接用。

打开工程，编译一下没问题，下载到板子上，呵呵，没问题了……更新就这样完成了，简单吧？

第六章 移植计算器

6.1 处理外部中断

上面都是简单的点亮 LED，相信不少朋友看到后，对 uC/OS-II 下编程还是会有所不解的，尤其是在 uC/OS-II 里如何处理外部中断。

其实，uC/OS-II 里的中断 跟 裸机的中断 没啥区别，唯一的区别就是 uC/OS-II 中断服务中需要调用中断服务函数 OSIntEnter 和 OSIntExit。

OSIntEnter 用来统计中断嵌套层数

OSIntExit 当中断嵌套层数为 0，OSLockNesting 为 0，有最高优先级进入就绪表时，就会进行任务切换。

M3 移植过程中，把 uC/OS-II 的任务切换编程为最低中断优先级，即在中断函数里，不会发生任务切换这类的事情，等中断结束后调用中断服务函数 OSIntExit()进行任务调度（中断嵌套层数为 0 且 OSLockNesting 为 0，有最高优先级进入就绪表）。而裸机的时候，中断返回到我们被中断的函数，两者的差别就在开头和结尾一步。

参考在 stm32f10x_it.c 的 SysTick 中断的处理代码：

```
605. void SysTick_Handler(void)
606. {
607.     OSIntEnter();           //开头添加 OSIntEnter
608.     OSTimeTick();           //中间就是中断函数需要处理的代码
609.     OSIntExit();            //结尾添加 OSIntExit
610. }
```

6.2 移植步骤

这次我们在《30-计算器-20111214》这个版本的基础上进行移植，这版本重新整理了 LCD 接口，使用起来会方便很多，也修正了一个计算 Debug，详情可以看里面的 USER\提示.txt 文件。

uC/OS-II 工程就用 STM32+UCOS+LED(V2.90 多任务) 为基础，进行移植。

思路就是把计算器工程的代码复制到 uC/OS-II 工程里，设置好中断服务函数，编译出错后，再修改代码。为了便于理解，这次的计算器，我们同样不会用通信功能。

当然，也可以往计算器里面移植 uC/OS-II，但那样就不利于了解 uC/OS-II 里如何处理外部中断。

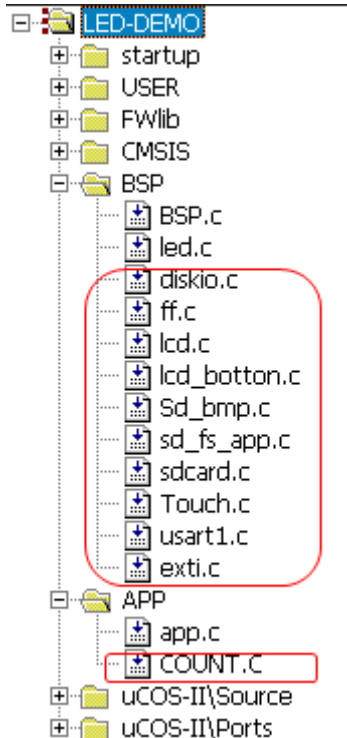
具体步骤

- ① 把 STM32+UCOS+LED(V2.90 多任务) 工程文件夹命名为 “STM32+UCOS+计算器(V290 不通信)”。
- ② 把 30-计算器-20111214\USER 文件夹下涉及硬件功能的文件都 copy 到 STM32+UCOS+计算器(V290 不通信)\BSP 文件夹下。

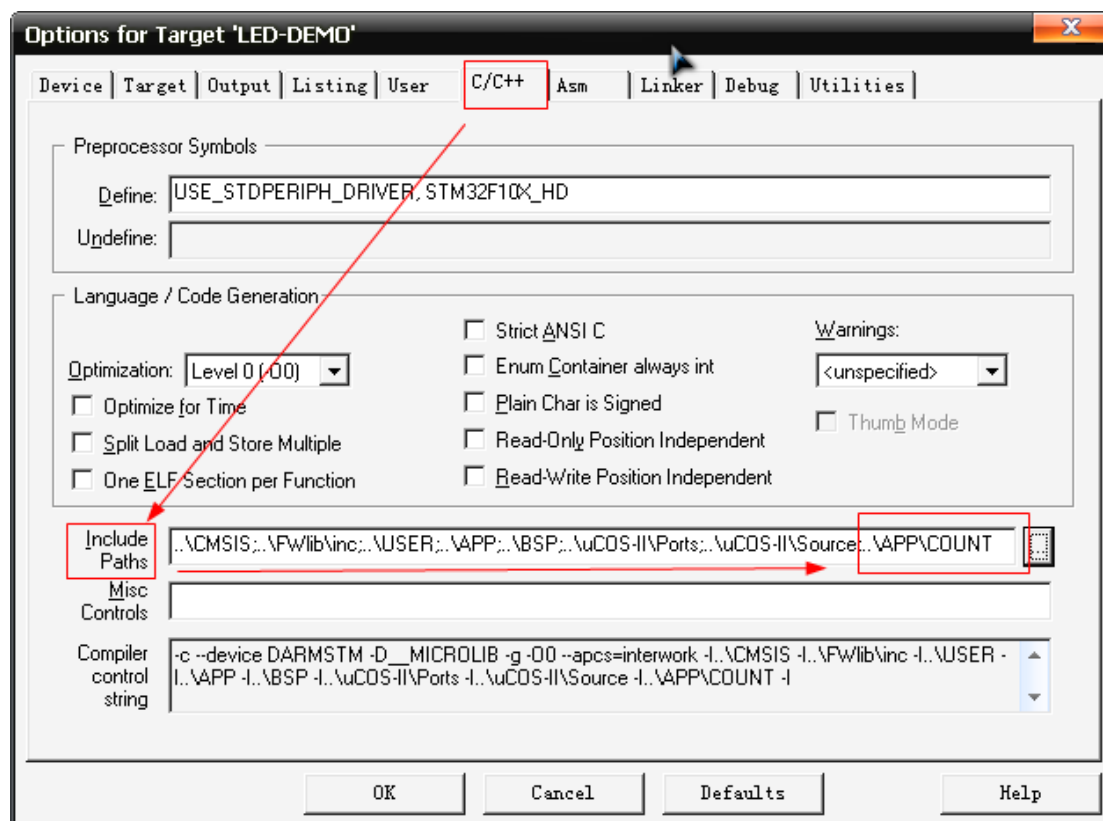
这些文件为：

ascii.h	
asc_font.h	
diskio.c	diskio.h
exti.c	exti.h
ff.c	ff.h
ff_config.h	
integer.h	
lcd.c	lcd.h
lcd_botton.c	lcd_botton.h
sdcard.c	sdcard.h
Sd_bmp.c	Sd_bmp.h
sd_fs_app.c	sd_fs_app.h
Touch.c	Touch.h
usart1.c	usart1.h

- ③ 把 30-计算器-20111214\USER 文件夹下 COUNT 文件夹复制到 STM32+UCOS+计算器(V290 不通信)\APP 文件夹下
- ④ 打开工程，也往工程里面添加这些对应的文件。



记得别忘了在 Include Path 里添加 ..\APP\COUNT



⑤ 编译一下，出现各类异常的错误：

- a) ..\BSP\Touch.c(11): error: #5: cannot open source input file "systick.h":
No such file or directory

原因： 找不到"systick.h"文件

解决方法： 这个是延时头文件，我们这里用 uC/OS-II 的延时函数，而不用之前的延时函数，所以，直接把这个注释掉就可以。

- b) ..\BSP\usart1.c(28): error: #20: identifier "USART_InitTypeDef" is
undefined
还有几个与之类似的……

原因： "USART_InitTypeDef"没定义。

解决方法： 在原来计算器工程那里跳到定义处，发现这个是在
"stm32f10x_usart.h"里定义的。

呵呵，还没配置好库文件，打开 STM32+UCOS+计算器(V290 不通
信)\USER\stm32f10x_conf.h。取消 #include "stm32f10x_usart.h" 的注释。

顺便也取消 #include "stm32f10x_exti.h" 的注释，因为触摸中断用到。

- c) ..\APP\COUNT\COUNT_CFG.H(81): error: #5: cannot open source input
file "systick.h": No such file or directory

这个，是计算器的移植配置文件，为了便于下次移植方便，我们把

```

612. #ifndef      DELAY
613.     #include   "systick.h"
614.     #define     DELAY      delay_ms(300)    //延时函数，避免触摸屏按的速度太快。
615.     #define     DELAY_MS(x)  delay_ms(x)
616. #endif

```

替换为：

```

617. #ifndef      DELAY
618.     #if 0
619.         #include   "systick.h"
620.         #define     DELAY      delay_ms(300)    //延时函数，避免触摸屏按的速度太快。
621.         #define     DELAY_MS(x)  delay_ms(x)
622.     #else
623.         #include   <ucos_ii.h>
624.         #define     DELAY      OSTimeDlyHMSM(0, 0, 0, 300);
625.         #define     DELAY_MS(x)  OSTimeDlyHMSM(0, 0, 0, x);
626.     #endif
627. #endif //DELAY

```

⑥ 继续重新编译一下，出现其他问题：

a) ..\BSP\Touch.c(729): warning: #223-D: function "delay_ms" declared implicitly

原因：没有定义"delay_ms"

解决方法：uC/OS-II 里用的是 uC/OS 自带的函数，我们在 Touch.c 文件上面添加：

```

628. #ifndef      DELAY
629.     #if 0
630.         #include   "systick.h"
631.         #define     DELAY      delay_ms(300)    //延时函数，避免触摸屏按的速度太快。
632.         #define     DELAY_MS(x)  delay_ms(x)
633.     #else
634.         #include   <ucos_ii.h>
635.         #define     DELAY      OSTimeDlyHMSM(0, 0, 0, 300);
636.         #define     DELAY_MS(x)  OSTimeDlyHMSM(0, 0, 0, x);
637.     #endif
638. #endif //DELAY

```

把所有的 delay_ms 函数替换为 DELAY_MS 宏定义。

b) ..\BSP\Touch.c(794): warning: #223-D: function "delay_init" declared implicitly

原因：没有定义"delay_init"延时初始化函数。

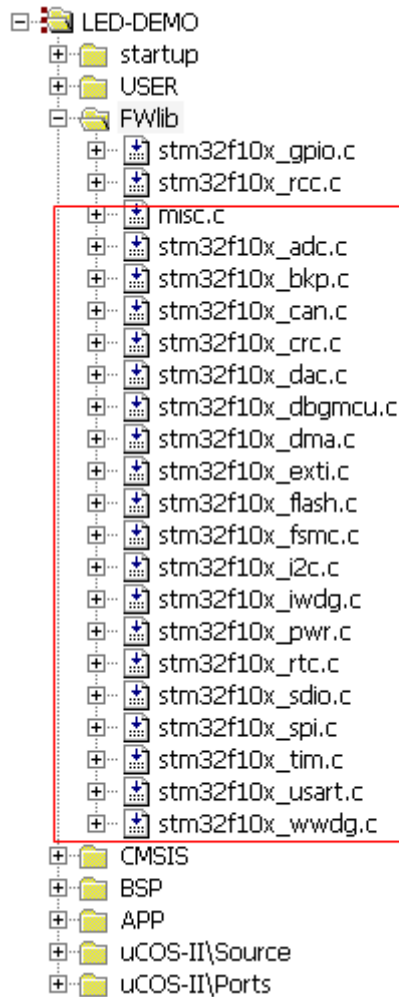
解决方法：晕，估计队友不小心在 TouchI_Calibrate 函数里添加了延时初始化函数，这个应该在 main 函数了实现的，所以直接删掉^_^

c) ..\Output\STM32-DEMO.axf: Error: L6218E: Undefined symbol EXTI_Init (referred from exti.o).

原因：没定义 EXTI_Init

解决方法：在原来工程那里跳到定义处，发现这个是在"stm32f10x_exti.c"里定义的。

呵呵，还没给工程添加好库文件。把 STM32+UCOS+计算器(V290 不通信)\FWlib\src 里的 C 文件都添加到工程里面。



其他类似的错误，也是添加库后就解决。

⑦ 再次编译一下，也出现了错误：

a) ..\APP\COUNT\COUNT.C(374): warning: #223-D: function "ScanTouch" declared implicitly

原因：没定义 ScanTouch

解决方法：呵呵，这个是读取触摸按键的，计算器工程那里是放在 main.c 里的，我们就在 app.c 里添加这个函数功能：

```
639. /*****
640. * 名 称 ScanTouch
641. * 功 能：扫描触摸屏，识别按下的键
642. * 入口参数：key 指针，用于保存识别到按下的键。
643. * 出口参数：无
644. * 说 明：函数，在 CountMain 函数里被调用。
645. * 调用方法：ScanTouch(&Key) ,注：此处 Key 定义为 char 型
646. * 完成程度：测试成功，by Song 2011.12.15
647. *****/
648. //触摸读取按键
649. //设置触摸扫描键盘设置
650. #define stx 94
```

```

651. #define sty      8
652. #define tx1     120
653. #define txc     36
654. #define ty1     40
655. #define tyc     65
656. char ScanTouch(char *key)                //测试成功
657. {
658.     u16 x,y;
659.     if(Get_touch_place(&x, &y)==0)        /*如果触笔有按下*/
660.     {
661.
662.         x = 240-x;
663.         if(x>stx      && x<=tx1 )          //x=0~40
664.         {
665.             if      (y>=sty      && y<=ty1      ) *key='7';
666.             else if (y>=sty+tyc   && y<=ty1+tyc   ) *key='8';
667.             else if (y>=sty+tyc*2 && y<=ty1+tyc*2 ) *key='9';
668.             else if (y>=sty+tyc*3 && y<=ty1+tyc*3 ) *key='+';
669.             else if (y>=sty+tyc*4 && y<=ty1+tyc*4 ) *key='(';
670.             else return 0;
671.         }
672.         else if(x>=stx+txc      && x<=tx1+txc )    //x=70~110
673.         {
674.             if      (y>=sty      && y<=ty1      ) *key='4';
675.             else if (y>=sty+tyc   && y<=ty1+tyc   ) *key='5';
676.             else if (y>=sty+tyc*2 && y<=ty1+tyc*2 ) *key='6';
677.             else if (y>=sty+tyc*3 && y<=ty1+tyc*3 ) *key='-';
678.             else if (y>=sty+tyc*4 && y<=ty1+tyc*4 ) *key=')';
679.             else return 0;
680.         }
681.         else if(x>=stx+txc*2     && x<=tx1+txc*2 )    //x=140~180
682.         {
683.             if      (y>=sty      && y<=ty1      ) *key='1';
684.             else if (y>=sty+tyc   && y<=ty1+tyc   ) *key='2';
685.             else if (y>=sty+tyc*2 && y<=ty1+tyc*2 ) *key='3';
686.             else if (y>=sty+tyc*3 && y<=ty1+tyc*3 ) *key='*';
687.             else if (y>=sty+tyc*4 && y<=ty1+tyc*4 ) *key='D';
688.             else return 0;
689.         }
690.         else if(x>=stx+txc*3     && x<=tx1+txc*3 )    //x=210~250
691.         {
692.             if      (y>=sty      && y<=ty1      ) *key='0';
693.             else if (y>=sty+tyc   && y<=ty1+tyc   ) *key='.';
694.             else if (y>=sty+tyc*2 && y<=ty1+tyc*2 ) *key='=';
695.             else if (y>=sty+tyc*3 && y<=ty1+tyc*3 ) *key='/';
696.             else if (y>=sty+tyc*4 && y<=ty1+tyc*4 ) *key='C';
697.             else return 0;
698.         }
699.         else return 0;
700.         return 1;
701.     }
702.     else return 0;
703. }

```

在 app.h 里声明:

```
704. extern char ScanTouch(char *key);
```

⑧ 编译一下，又出现问题:

a) ..\APP\app.c(74): warning: #223-D: function "Get_touch_place" declared implicitly

原因: 没定义 Get_touch_place , 其实是没有包含头文件

解决方法：在 `includes.h` 里添加对应的头文件。为了方便，把其他用到的头文件也一起加进来：

```
705. #include "lcd.h" //LCD 接口
706. #include "sd_fs_app.h" //SD 卡文件系统接口
707. #include "Sd_bmp.h" //SD 卡解码 BMP 接口
708. #include "usart1.h" //串口 1
709. #include "Touch.h" //触摸
710. #include "COUNT.H" //计算器头文件
```

⑨ 编译一下，没报错了。但我们还没有编写计算任务，目前仅仅还是 LED 的任务。

在 `app.c` 里，先删掉任务 `Task_LED2` 和 `Task_LED3`，再创建计算任务 `Task_Count`，修改启动任务 `Task_START`：

```
711. static OS_STK Count_task_stk[COUNT_TASK_STK_SIZE]; //定义 Count 任务的栈
712.
713. /*****
714. * 名 称 Task_Start
715. * 功 能：启动任务，创建另外一个任务，然后进入死循环闪烁 LED1
716. * 入口参数：无
717. * 出口参数：无
718. * 说 明：任务
719. * 调用方法：必须作为 OSTaskCreate 函数参数来调用
720. * 完成程度：测试成功，by Song 2011.12.15
721. *****/
722. void Task_Start(void *p_arg)
723. {
724.     (void)p_arg; // 'p_arg' 并没有用到，防止编译器提示警告
725.
726.     OSTaskCreate(Task_Count, (void *)0,
727.         &Count_task_stk[COUNT_TASK_STK_SIZE-1], COUNT_TASK_PRIO);
728.
729.     while (1)
730.     {
731.         LED1( ON );
732.         OSTimeDlyHMSM(0, 0,0,500);
733.         LED1( OFF);
734.         OSTimeDlyHMSM(0, 0,0,500);
735.     }
736. }
737. /*****
738. * 名 称 Task_Count
739. * 功 能：计算任务，包括扫描触摸屏，识别按键，对输入的按键进行处理。
740. * 入口参数：无
741. * 出口参数：无
742. * 说 明：任务
743. * 调用方法：必须作为 OSTaskCreate 函数参数来调用
744. * 完成程度：测试成功，by Song 2011.12.15
745. *****/
746. void Task_Count(void *p_arg)
747. {
748.     (void)p_arg; // 'p_arg' 并没有用到，防止编译器提示警告
749.     while (1)
750.     {
751.         CountMain(); //计算器函数
752.     }
753. }
```

在 `app.h` 里声明 `Task_Count`，并删掉任务 `Task_LED2` 和 `Task_LED3` 的声明：

```
754. void Task_Count(void *p_arg);
```

在 **app_cfg.h** 里设置优先级和栈:

```
755. #ifndef __APP_CFG_H__
756. #define __APP_CFG_H__
757.
758. /*****设置任务优先级*****/
759. #define STARTUP_TASK_PRIO 0 //开始启动的任务的优先级
760. #define COUNT_TASK_PRIO 5 //计算任务的优先级
761.
762. /*****设置栈大小(单位为 OS_STK) *****/
763. #define STARTUP_TASK_STK_SIZE 80
764. #define COUNT_TASK_STK_SIZE 100
765.
766. #endif
```

⑩ 添加硬件初始化

在 **BSP.c** 里, 修改 **BSP_Init** 函数:

```
767. void BSP_Init(void)
768. {
769.     SystemInit(); // 配置系统时钟为 72M */
770.     SysTick_init(); // 初始化并使能 SysTick 定时器 */
771.     LED_GPIO_Config(); // LED 端口初始化 */
772.     LCD_Init();
773.     USART1_Config();
774.     sd_fs_init(); //文件系统初始化*/
775.     Touch_init(); //触摸初始化*/
776.     while(Touch1_Calibrate() != 0); //触摸品校准*/
777.     Lcd_show_bmp(0, 0, "/CAL.bmp"); //在 LCD 上显示 SD 卡上存放的 test.bmp 图片*/
778.     LCD_Str_O(29, 2, "10", BLACK);
779. }
```

⑪ 添加中断函数

注意了, 这里就告诉你怎么处理外部中断。

在 **stm32f10x_it.c** 里添加下面两个函数:

```
780. /*
781. * 函数名: SDIO_IRQHandler
782. * 描述 : SDIO 全局中断请求服务程序
783. * 输入 : 无
784. * 输出 : 无
785. */
786. void SDIO_IRQHandler(void)
787. {
788.     OSIntEnter();
789.     SD_ProcessIRQSrc(); // 处理 SDIO 全部中断 */
790.     OSIntExit(); //调用中断服务函数
791. }
792.
793. /* XPT2046 中断处理函数 */
794. void EXTI9_5_IRQHandler(void)
795. {
796.     OSIntEnter();
797.     if(EXTI_GetITStatus(EXTI_Line6) != RESET)
798.     {
799.         touch_flag = 1; // 触摸按下 */
800.         EXTI_ClearITPendingBit(EXTI_Line6);
801.     }
802.     OSIntExit(); //调用中断服务函数
803. }
```

其实就是开头和结尾分别多了个：OSIntEnter()和 OSIntExit()。可以打开计算器工程里的 stm32f10x_it.c 看看就知道。

- ⑫ 编译一下，出现错误了：stm32f10x_it.c(65): error: #20: identifier "touch_flag" is undefined

原因：没声明 touch_flag

解决方法：在 **Touch.h** 里添加声明：

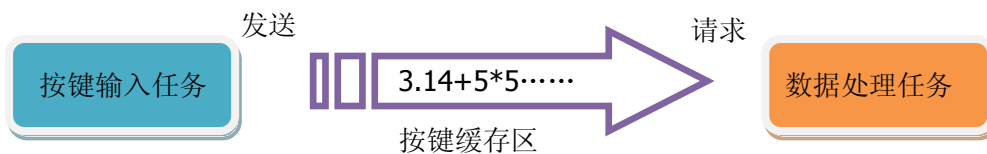
```
804. volatile extern unsigned char touch_flag;
```

呵呵，编译一下，就会发现没错的。烧到单片机上试试，记得要拷贝计算器模块用到的资源到 SD 卡哦。

辛苦了那么久，这部分终于讲完。其实修改的内容也不多，不过你们看着教程来修改，肯定也得花不少时间。

6.3 增加通信功能——队列

这一小节，我们给计算器实验添加通信功能。把原来的计算器任务分为按键输入和数据处理并显示出来两个部分：



那样对于用户而已，或许他的输入速度较快，单片机还没来得及处理完数据，则队列就提供一个缓冲区来缓存数据，等单片机处理完一个数据时就会请求另外一个数据，继续处理。

下面，我们来在原来的《STM32+UCOS+计算器(V290 不通信)》修改步骤：

- ① 复制《STM32+UCOS+计算器(V290 不通信)》出来，把文件夹命名为《STM32+UCOS+计算器(V290 队列)》

- ② 修改 os_cfg.h，把里面的 OS_Q_EN 宏定义定义为 1u：

```
805. #define OS_Q_EN 1u //允许使用消息队列
```

- ③ 修改 app.c 的各个任务：

```
806. #include "includes.h"
807.
808. /*-----设置任务堆栈-----*/
809. static OS_STK Count_task_stk[COUNT_TASK_STK_SIZE]; //定义 Count 任务的栈
810. static OS_STK Touch_task_stk[TOUCH_TASK_STK_SIZE]; //定义 Touch 任务的栈
811.
812. /*-----设置队列参量-----*/
813. #define N_MESSAGES 10u //定义消息队列长度
814. void *MsgGrp[N_MESSAGES]; //定义消息指针数组
815. OS_EVENT *Touch_Q; //定义事件控制块
816.
817. char touchkey[N_MESSAGES]; //定义按键缓冲区
818. unsigned char err; //队列返回错误类型
819.
820.
821. /*****
```



```

822.* 名    称 Task_START
823.* 功    能: 启动任务, 创建其他两个任务, 然后进入死循环闪烁 LED1
824.* 入口参数: 无
825.* 出口参数: 无
826.* 说    明: 任务
827.* 调用方法: 必须作为 OSTaskCreate 函数参数来调用
828.* 完成程度: 测试成功, by Song 2011.12.15
829.*/*****/
830.void Task_Start(void *p_arg)
831.{
832.    (void)p_arg;                // 'p_arg' 并没有用到, 防止编译器提示警告
833.    SysTick_init();
834.
835.    Touch Q=OSQCreate(&MsgGrp[0],N_MESSAGES);                //创建消息队列
836.
837.    OSTaskCreate(Task_Touch,(void *)0,
838.        &Touch_task_stk[TOUCH_TASK_STK_SIZE-1],TOUCH_TASK_PRIO); //创建触摸任务
839.
840.    OSTaskCreate(Task_Count,(void *)0,                //创建计算任务
841.        &Count_task_stk[COUNT_TASK_STK_SIZE-1],COUNT_TASK_PRIO);
842.
843.    while (1)
844.    {
845.        LED1( ON );
846.        OSTimeDlyHMSM(0, 0,0,500);
847.        LED1( OFF);
848.        OSTimeDlyHMSM(0, 0,0,500);
849.    }
850.}
851.
852.
853.*/*****/
854.* 名    称 Task_Touch
855.* 功    能: 触摸任务, 不断检测是否有发送触摸, 把按下的键发送到队列里。
856.* 入口参数: 无
857.* 出口参数: 无
858.* 说    明: 任务
859.* 调用方法: 必须作为 OSTaskCreate 函数参数来调用
860.* 完成程度: 测试成功, by Song 2011.12.15
861.*/*****/
862.void Task_Touch(void *p_arg)
863.{
864.    char    i=0;                //用于计数
865.    (void)p_arg;                // 'p_arg' 并没有用到, 防止编译器提示警告
866.    while (1)
867.    {
868.        if(ScanTouch(&touchkey[i]))                //按下键了
869.        {
870.            OSQPost(Touch Q,&touchkey[i]);                //发送消息
871.            i++;                //执行下个缓冲区, 避免覆盖原来的按键数据
872.            if(i==N_MESSAGES)i=0;
873.            OSTimeDlyHMSM(0, 0,0,200);                //延时 200ms, 避免太灵敏
874.        }
875.        OSTimeDlyHMSM(0, 0,0,10);                //延时, 避免低优先级无法执行
876.    }
877.}
878.
879.
880.*/*****/
881.* 名    称 Task_Count
882.* 功    能: 计算任务, 不断请求队列, 请求到数据后, 处理数据, 并在 LCD 显示结果
883.* 入口参数: 无
884.* 出口参数: 无

```

```

885.* 说明：任务
886.* 调用方法：必须作为 OSTaskCreate 函数参数来调用
887.* 完成程度：测试成功，by Song 2011.12.15
888.*****/
889.void Task_Count(void *p_arg)
890.{
891.    char    key;
892.    char    CountStatus=calnot;                //计算状态
893.    struct  num_point tmp;                     //计算结果
894.    struct  str_point sp;                     //计算式子
895.    u8      Result[RN];                       //计算结果字符串，把计算结果转换为字符串
896.
897.    (void)p_arg;                             // 'p_arg' 并没有用到，防止编译器提示警告
898.
899.    ClearSP(&sp);
900.    while (1)
901.    {
902.        key=((char *)OSQPend(Touch_Q,0,&err));
903.        //向 Touch_Q 队列请求消息。等待时限为 0，即无限等待下去，直到获取消息
904.
905.        /*****进入计算部分*****/
906.        switch(key)
907.        {
908.            case '=': //-----计算式子-----
909.                if(CountStatus==caled)break;
910.                CountStatus=caled;                //计算完毕
911.
912.                tmp=calculate(sp.str,10);         //计算字符串结果
913.                if(tmp.point==0)
914.                {
915.                    PERROR;                       //输入式子有误
916.                    break;
917.                }
918.                CLEAN_R;                          //清屏
919.                float2stre(tmp.result,Result);    //把计算结果转为字符串
920.                PRINT_R(Result);                 //显示计算结果
921.                break;
922.            case 'C': //-----清空式子-----
923.                CountStatus=caling;
924.                ClearSP(&sp);
925.                CLEAN_R;                          //清屏
926.                CLEAN_S;
927.                break;
928.            case 'D': //-----退格-----
929.                CountStatus=caling;
930.                DelSP(&sp);
931.                CLEAN_R;                          //清屏
932.                CLEAN_S;
933.                PRINT_S(sp.str);                 //更新屏幕字符串
934.                break;
935.            default: //-----输入式子-----（剩下的就是运算表达式）
936.                if(CountStatus==caled)           //如果计算完成后，就清空
937.                {
938.                    CountStatus=caling;
939.                    ClearSP(&sp);
940.                    CLEAN_R;                      //清屏
941.                    CLEAN_S;
942.                }
943.                if(AddCheckSP(&sp,key))          //输入 key 有效
944.                {
945.                    PRINT_S(sp.str);             //更新屏幕字符串
946.                }
947.            }//switch
948.        }

```

```
949. }
```

④ 修改 app.h (添加声明) :

```
950. #ifndef _APP_H_
951. #define _APP_H_
952.
953. /***** 用户任务声明 *****/
954. void Task_Start(void *p_arg);
955. void Task_Count(void *p_arg);
956. void Task_Touch(void *p_arg);
957.
958. extern char ScanTouch(char *key); //触摸扫描程序
959.
960. #endif // _APP_H_
```

⑤ 修改 app_cfg.h (设置优先级和栈大小) :

```
961. #ifndef _APP_CFG_H_
962. #define _APP_CFG_H_
963.
964.
965. /***** 设置任务优先级 *****/
966. #define STARTUP_TASK_PRIO 0 //开始启动的任务的优先级
967. #define TOUCH_TASK_PRIO 1 //触摸优先级
968. #define COUNT_TASK_PRIO 6 //计算优先级
969.
970. /***** 设置栈大小 (单位为 OS_STK) *****/
971. #define STARTUP_TASK_STK_SIZE 100
972. #define COUNT_TASK_STK_SIZE 100
973. #define TOUCH_TASK_STK_SIZE 100
974.
975. #endif
```

呵呵，其实就是把 CountMain()函数的功能拆分成两个部分，用两个任务来实现，然后通过队列来通信。修改后，编译一下，再烧写到单片机上，计算器的功能能够正常使用，与没通信的看不出有啥区别，就这样完成了。