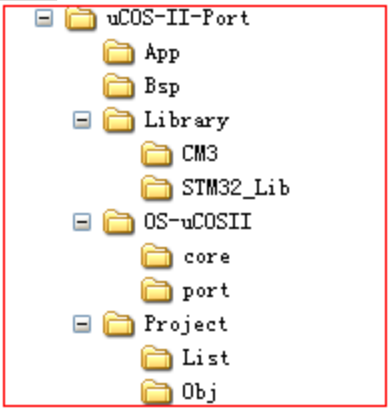
# STM32 平台移植 uCOS-II 详细说明 v1.0

**硬件平台：** 盘古 UE-STM32F103 开发板
**软件平台：** RVMDK_v4.20 + uCOS-II_v2.86 + StmLib_v3.5
**联系方式：** WWW.UE-TECH.NET
**淘宝店铺：** UETECH.TAOBAO.COM
**文档作者：** 合嵌电子科技有限公司

## 移植准备

1. 建立工程所需的文件夹
   - 建立文件夹 uCOS-II-Port ：工程根目录
   - 建立文件夹 uCOS-II-Port/App ：存放用户应用程序相关
   - 建立文件夹 uCOS-II-Port/Bsp ：存放开发板初始化驱动文件
   - 建立文件夹 uCOS-II-Port/Library
   - 建立文件夹 uCOS-II-Port/Library/CM3 ：存放启动文件及内核支撑文件
   - 建立文件夹 uCOS-II-Port/Library/CM3/startup
   - 建立文件夹 uCOS-II-Port/Library/STM32_Lib ：存放标准外设函数库文件
   - 建立文件夹 uCOS-II-Port/OS-uCOSII
   - 建立文件夹 uCOS-II-Port/OS-uCOSII/core ：存放 uCOS-II 源代码，无需修改
   - 建立文件夹 uCOS-II-Port/OS-uCOSII/port ：存放移植相关文件，需修改
   - 建立文件夹 uCOS-II-Port/Project ：存放工程相关文件
   - 建立文件夹 uCOS-II-Port/Project/List
   - 建立文件夹 uCOS-II-Port/Project/Obj

   此步骤完成以后，目录结构如下所示：




2. 移植源码包(光盘中附带)：
   - STM32 标准外设驱动库 v3.5
     此源代码的文件结构不再说明
   - uCOS-II 系统源代码 v2.86
     解压后文件结构如下：


具体文件结构说明如下图所示：

| 文件名 | 说明 | | | |
|---|---|---|---|---|
| AppNote | uCOS-II说明，其中AppNotes\AN1xxx-RTOS\AN1018-uCOS-II-Cortex-M3.pdf文件介绍了移植过程的详细说明。 | | | |
| Licensing | uCOS-II的应用许可。 | | | |
| Software | 此文件夹在移植过程中需要修改，具体为uCOS-II文件夹下的ports和source。 | | | |
| | uCOS-II | Doc | 官方自带说明文档和教程 | |
| | | Ports | 官方移植M3的文件(RealView) | |
| | | | Cpu.h | 数据类型、处理器相关代码、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 | uCOS-Ii的源代码文件 | |
| | | | ucos_ii.h | 内核函数参数设置 |
| | | | 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 | 延时处理 |
| | | | 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_mutex.c | 信号量管理 |
| | | | os_mbox.c | 邮箱消息 |
| | | | os_q.c | 队列 |
| | | | os_flag.c | 事件标志组 |
| | CPU | STM32标准外设固件库 | | |
| | EvalBoards | micrium 官方评估板的代码 | | |
| | 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 | 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。 | | |

## 3. 文件对号入座

通过之前的准备工作，我们需要把官方源码包中相应的文件，拷贝到我们建立的工程文件夹中，首先进行库函数源代码搬移工作：

- 打开 `STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\STM32F10x_StdPeriph_Driver`

将其下的 inc 和 src 拷贝至 uCOS-II-Port\Library\STM32_Lib

- 打开 STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3
  其下有 CoreSupport 和 DeviceSupport 两个文件夹
  - 分别将 CoreSupport 下的 core_cm3.c 和 core_cm3.h 和 DeviceSupport\ST\STM32F10x 下的 stm32f10x.h、system_stm32f10x.c 和 system_stm32f10x.h 拷贝至 uCOS-II-Port\Library\CM3，并去掉只读属性
  - 再将 DeviceSupport\ST\STM32F10x\startup\arm 下的 startup_stm32f10x_hd.s 拷贝至 uCOS-II-Port\Library\CM3\startup
    注：盘古 UE-STM32F103 的主芯片的内部 flash 为 512K
- 打开 STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Template
  将其下的 stm32f10x_conf.h、stm32f10x_it.c 和 stm32f10x_it.h 拷贝至 uCOS-II-Port\App

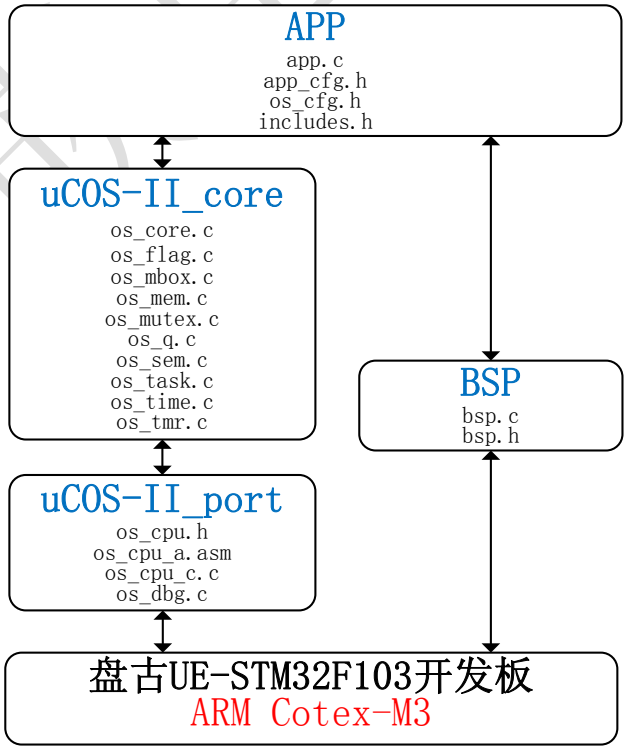
至此，库函数的源代码搬移工作已经完成，现在进行 uCOS-II 的源代码搬移工作：

- 打开 Micrium\Software\uCOS-II\Source
  将其下的所有文件拷贝至 uCOS-II-Port\OS-uCOSII\core
- 打开 Micrium\Software\uCOS-II\Ports\ARM-Cortex-M3\Generic\RealView
  将其下的所有文件拷贝至 ucos\uCOS-II-Port\OS-uCOSII\port
- 打开 Micrium\Software\EvalBoards\ST\STM3210B-EVAL\RVMDK\OS-Probe
  将其下的 os_cfg.h 拷贝至 ucos\uCOS-II-Port\App

至此，所有的可利用的文件已经搬移结束，不过仍然需要建立一些文件，这个工程的文件结构才算完整，具体如下：

- 打开 ucos\uCOS-II-Port\App
  新建 app.c、app_cfg.h 和 includes.h 三个空文件
- 打开 ucos\uCOS-II-Port\Bsp
  新建 bsp.c 和 bsp.h 两个空文件

到目前为止，我们所有的文件准备工作已经完成，我们可以了解一下 uCOS-II 的体系结构，如下所示：



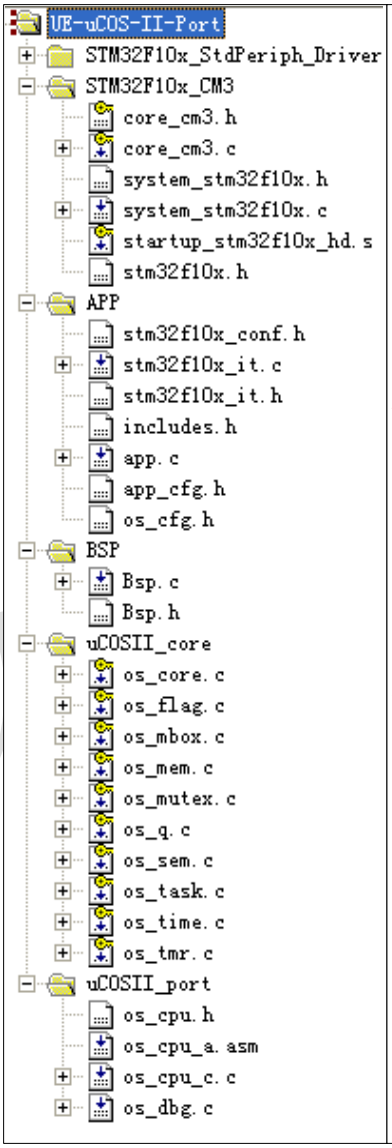
## 4. 建立 Keil 工程

- 打开 Keil_v4.20，新建工程 UE-uCOS-II-Port 工程，并将其保存至 uCOS-II-Port\Project

在随后跳出的窗口中，选择芯片型号，盘古 UE-STM32F103 开发板的芯片为：STM32F103VET6
点击 OK，跳出对话框，是否自动添加启动文件，注意此处选择否，因为我们会自己添加。

- 右击项目窗口中 Target1，选择 Manage Components，在窗口中创建文件组，并在相应的组添加文件，具体如下：
  - 将 Project Targets 中的 Target1 重命名为 UE-uCOS-II-Port
  - 新建组 STM32F10x_StdPeriph_Driver，并将 uCOS-II-Port\Library\STM32_Lib\src 下的所有文件添加到此组下
  - 新建组 STM32F10x_CM3，并将 uCOS-II-Port\Library\CM3 下所有文件添加到此组中（包括 C 文件、H 文件和 startup 下的文件）
  - 新建组 APP，并将 uCOS-II-Port\App 下所有文件添加到此组中
  - 新建组 BSP，并将 uCOS-II-Port\Bsp 下所有文件添加到此组中
  - 新建组 uCOSII_core，并将 uCOS-II-Port\OS-uCOSII\core 下所有 C 文件添加到此组中
  - 新建组 uCOSII_port，并将 uCOS-II-Port\OS-uCOSII\port 下所有文件添加到此组中

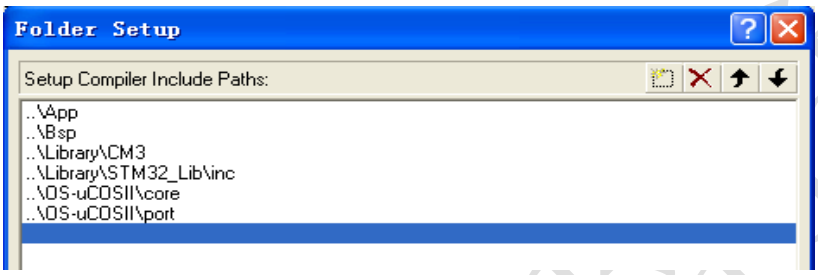
  具体操作结果，及各文件说明如下图所示：



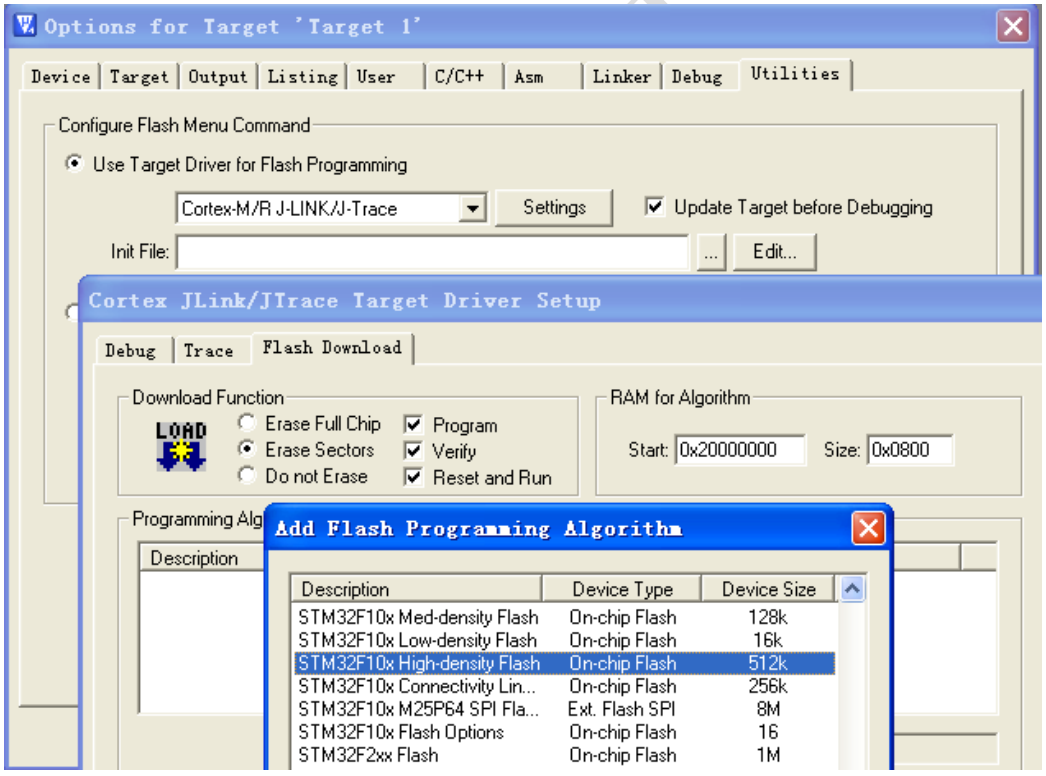

STM32F10x_StdPeriph_Driver:
STM32固件库函数v3.5，包含了各个外设驱动代码；
STM32F10x_CM3：
core_cm3.h和core_cm3.c文件为内核支撑文件，其他CM3核的芯片也能使用；
stm32f10x.h为标准函数库的入口文件，包含了一些寄存器的定义；
system_stm32f1ox.h、system_stm32f10x.c提供了初始化stm32芯片的库函数，以及配置PLL、系统时钟和内置flash的接口函数；
startup_stm32f10x_hd.s为stm32的启动文件，hd表示大容量的芯片。
APP:
stm32f10x_conf.h为外设配置文件，此文件可以使能/禁用外设驱动；
stm32f10x_it.c和stm32f10x_it.h为中断服务程序文件；
includes.h为全部头文件的头文件，对头文件进行统一管理；
app.c为应用程序文件，包含Main函数；
app_cfg.h用来配置应用软件，主要是任务的优先级和堆栈大小及中断优先级；
os_cfg.h为内核配置头文件，移植时需要修改；
BSP:
Bsp.c存放了开发板初始化启动函数，包含设置系统时钟，初始化硬件；
Bsp.h包含有与开发板初始化相关函数的头文件；
uCOSII_core:
此文件组包含了uCOS-II的源代码文件，在移植的过程中，不需要修改。
uCOSII_port:
此文件组包含了移植的相关文件
os_cpu.h进行数据类型定义，处理器相关代码和几个函数型；
os_cpu_c.c定义一些用户 hook 函数；
os_cpu_a.asm为移植需要用汇编代码完成的函数，主要就是任务切换函数；
os_dbg.c为内核调试相关数据和函数，可以不改。

5. 设置 Option 选项

- Device 选项卡
  此步骤前面已经操作，即选择主芯片：stm32f103vet6
- Output 选项卡
  设置工程输出文件至：uCOS-II-Port\Project\Obj
- Listing 选项卡
  设置工程 Listing 路径值 uCOS-II-Port\Project\List
- C/C++选项卡
  设置 H 文件的路径




- Debug 选项卡
  在此选项卡中选择你所连接的 JLINK，并作相应配置
- Utilities 选项卡
  作出如下选择操作




至此为止，工程已经建立完毕，接下来需要对相关文件进行修改移植。

6. 移植修改

以下移植步骤来自 Micrium\AppNotes\AN1xxx-RTOS\AN1018-uCOS-II-Cortex-M3\AN-1018.pdf

- os_cpu.h

此文件定义数据类型、处理器相关代码、声明函数原型，下面为部分代码的解释说明。

```
/*全局变量*/
#ifdef  OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT  extern
#endif

/*数据类型*/
typedef unsigned char BOOLEAN;
typedef unsigned char INT8U;
typedef signed char INT8S;
typedef unsigned short INT16U;
typedef signed short INT16S;
typedef unsigned int INT32U;
typedef signed int INT32S;
typedef float FP32;
typedef double FP64;
typedef unsigned int OS_STK;
typedef unsigned int OS_CPU_SR;

/*临界段*/
#define  OS_CRITICAL_METHOD 3      //进入临界段的三种模式，一般选择第 3 种
#define  OS_ENTER_CRITICAL()    {cpu_sr = OS_CPU_SR_Save();}
#define  OS_EXIT_CRITICAL() {OS_CPU_SR_Restore(cpu_sr);}
```

为了实现资源共享，一个操作系统必须提供临界段擦作的功能。

uCOS-II 为了处理临界段代码需要关中断，处理完毕后再开中断。这使得 uCOS-II 能够避免同时有其它任务或中断服务进入临界段代码。

微处理器一般都有关中断/开中断指令，用户使用的 C 语言编译器必须有某种机制能够在 C 中直接实现关中断/开中断地操作。某些 C 编译器允许在用户的 C 源代码中插入汇编语言的语句。这使得插入微处理器指令来关中断/开中断很容易实现。而有的编译器把从 C 语言中关中断/开中断放在语言的扩展部分。uCOS-II 定义两个宏(macros)来关中断和开中断，以便避开不同 C 编译器厂商选择不同的方法来处理关中断和开中断。uCOS-II 中的这两个宏调用分别是：OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()。

```
/*栈方向*/
#define  OS_STK_GROWTH        1
```

Cotex-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

```
/*任务切换宏*/
#define  OS_TASK_SW()         OSCtxSw()

/*开中断 关中断*/
#if OS_CRITICAL_METHOD == 3
```

```
OS_CPU_SR   OS_CPU_SR_Save(void);
void        OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
#endif
```

其 中 OS_CPU_SR_Save() 和 OS_CPU_SR_Restore()是用汇编代码写的，代码在 os_cpu_a.asm

```
/*任务切换的函数*/
void OSCtxSw(void);                    //用户任务切换
void OSIntCtxSw(void);                 //中断任务切换函数
void OSStartHighRdy(void);             //在操作系统第一次启动的时候调用的任务切换
void OS_CPU_PendSVHandler(void);       //用户中断处理函数
void OS_CPU_SysTickHandler(void);      //系统定时中断处理函数，时钟节拍函数
void OS_CPU_SysTickInit(void);         //系统 SysTick 定时器初始化
INT32U OS_CPU_SysTickClkFreq(void);//返回 SysTick 定时器的时钟频率
```

关于任务切换，会涉及到异常处理，具体为 SVC（系统服务调用，亦简称系统调用）和 PendSV（可悬起系统调用），它们常用于在操作系统之上的软件开发中。

SVC 用于产生系统函数的调用请求。例如，操作系统不让用户程序直接访问硬件，而是通过提供一些系统服务函数，用户程序使用 SVC 发出对系统服务函数的呼叫请求，以这种方法调用它们来间接访问硬件。因此，当用户程序想要控制特定的硬件时，它就会产生一个 SVC 异常，然后操作系统提供的 SVC 异常服务例程得到执行，它再调用相关的操作系统函数，后者完成用户程序请求的服务。SVC 异常通过执行”SVC”指令来产生，该指令需要一个立即数，充当系统调用代号。SVC 异常服务例程稍后会提取出此代号，从而解释本次调用的具体要求，再调用相应的服务函数。

另一个相关的异常是 PendSV（可悬起的系统调用），它和 SVC 协同使用。一方面，SVC 异常是必须立即得到响应的（若因优先级不比当前正处理的高，或是其它原因使之无法立即响应，将上访成硬 fault），应用程序执行 SVC 时都是希望所需的请求立即得到响应。另一方面，PendSV 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。

具体异常处理相关知识，若想知其原理，请详细阅读《Cotex-M3 权威指南》。

在此处，我们需要对此文件进行修改：

1） void OS_CPU_PendSVHandler(void) 需替换成 void PendSV_Handler(void)

一般我们自己开发基于 stm32 芯片的软件，都会使用标准外设库 CMSIS 中提供的启动文件，比如 startup_stm32f10x_hd.s，而 Micrium 官方没有用 ST 的标准启动文件，而且分开写成了两个.s 文件，即

init.s 和 vectors.s（Micrium\Software\EvalBoards\ST\STM3210B-EVAL\RVMDK）

init.s 负责进入 main()，vectors.s 设置中断向量

由于 OS_CPU_PendSVHandler 这个中断向量就是在 vectors.s 中被设置的，且我们使用的是 startup_stm32f10x_hd.s 作为启动文件的，而在 startup_stm32f10x_hd.s 文件中，PendSV 的中断向量名为 PendSV_Handler，所以只需用 PendSV_Handler 替换掉相应文件的 OS_CPU_PendSVHandler，其中函数声明在 OS_CPU_C.h 中，具体的中断服务函数原型在 OS_CPU_A.ASM 中，后面也将对其进行修改。

这样子，替换后的 PendSV_Handler 函数在 OS_CPU_C.h 中有声明，在 OS_CPU_A.ASM 中有具体的中断服务函数代码，与 startup_stm32f10x_hd.s 中的向量地址就对应上了。

2） 注释掉最后三个关于 SysTick 服务函数

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

其中，OS_CPU_SysTickHandler 函数在 ST 标准库 stm32f10x_it.c 中已定义，此处不需要；其中，OS_CPU_SysTickInit 定义在 os_cpu_c.c 中，依赖于 OS_CPU_SysTickClkFreq，用于初始化 SysTick 定时器，需注释掉；其中，OS_CPU_SysTickClkFreq 定义在官方 EvalBoards 的 BSP.c 中，需解除依赖，若需要，我们可以在 bsp.c 中实现。

修改后如下所示：

```
109 //void        OS_CPU_PendSVHandler(void);
110 void PendSV_Handler(void);                          /* See OS_CPU_C.C
111
112 //void        OS_CPU_SysTickHandler(void);
113 //void        OS_CPU_SysTickInit(void);             /* See BSP.C
114 //INT32U      OS_CPU_SysTickClkFreq(void);
115 #endif
```

SysTick 作为 OS 的“心跳”，可称为滴答时钟，本质上来说就是一个定时器，和 PendSV 中断一样，在 startup_stm32f10x_hd.s 中 SysTick 的中断向量名为 SysTick_Handler，且因为 ST 标准库已经有相关库函数，所以我们只需作如下修改：

打开 os_cpu_c.c 文件，找到 void OS_CPU_SysTickHandler(void)的内容代码

```
OS_CPU_SR  cpu_sr;
OS_ENTER_CRITICAL();
OSIntNesting++;
OS_EXIT_CRITICAL();
OSTimeTick();
OSIntExit();
```

复制到 stm32f10x_it.c 文件中的 SysTick_Handler (void)函数内；

```
void SysTick_Handler(void)
{
    OS_CPU_SR  cpu_sr;
    OS_ENTER_CRITICAL();
    OSIntNesting++;
    OS_EXIT_CRITICAL();
    OSTimeTick();
    OSIntExit();
}
```

并且在文件头部添加：#include<ucos_ii.h>

```
025 #include "stm32f10x_it.h"
026 #include <ucos_ii.h>
```

- os_cup_a.asm

根据前面的描述，OS_CPU_PendSVHandler 中断服务函数的原型在此文件中，我们需要用 PendSV_Handler 将其替换，以实现在 startup_stm32f10x_hd.s 中的中断向量的匹配。

1) 注释掉 EXPORT OS_CPU_PendSVHandler，并修改成 EXPORT  PendSV_Handler，如下所示：

```
041 ;   EXPORT  OS_CPU_PendSVHandler
042     EXPORT  PendSV_Handler
```

2) 找到 OS_CPU_PendSVHandler 程序原型，并重命名为 PendSV_Handler

```
204 ;OS_CPU_PendSVHandler
205 PendSV_Handler
206     CPSID   I
207     MRS     R0, PSP
208     CBZ     R0, OS_CPU_PendSVHandler_nosave
209
210     SUBS    R0, R0, #0x20
211     STM     R0, {R4-R11}
212
213     LDR     R1, =OSTCBCur
214     LDR     R1, [R1]
215     STR     R0, [R1]
```

这样 PendSV_Handler 中断服务函数就成功建立了，同时，我们需要注释掉 stm32f10x_it.h 和 stm32f10x_it.c 中的相关 PendSV_Handler 的声明和定义，以防止冲突，如下所示：

```
40 void MemManage_Handler(void);
41 void BusFault_Handler(void);
42 void UsageFault_Handler(void);
43 void SVC_Handler(void);
44 void DebugMon_Handler(void);
45 //void PendSV_Handler(void);
46 void SysTick_Handler(void);
```

```
127
128 /*void PendSV_Handler(void)
129 {
130
131
132 }
133 */
134
```

- os_cpu_c.c

此文件需要由我们来写 10 个相当简单的 C 函数

OSInitHookBegin()
OSInitHookEnd()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskIdleHook()
OSTaskStatHook()
OSTaskStkInit()
OSTaskSwHook()
OSTCBInitHook()
OSTimeTickHook()

主要包括 9 个钩子函数和 1 个负责建立任务堆栈的函数 OSTaskStkInit()。

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而存在的函数。一般来说，钩子函数是进行软件功能扩充的入口点。不仅如此，uCOS-II 中还提供有大量的钩子函数，用户不需要修改 uCOS-II 的内核代码程序，而只需要向钩子函数添加代码即可拓展 uCOS-II 的功能，如果要用到这些钩子函数，需要在

OS_CFG.H 中使能 OS_CPU_HOOKS_EN 为 1，即：#define OS_CPU_HOOKS_EN 1

同时关于 OSTaskStkInit，OSTaskCreate 和 OSTaskCreateExt 通过调用 OSTaskStkInt 来初始化任务的堆栈结构，因此，堆栈看起来就像刚发生过中断并将所有的寄存器保存到堆栈中的情形一样。一旦用户初始化了堆栈，OSTaskStkInit 就需要返回堆栈指针所指的地址，OSTaskCreate

和 OSTaskCreateExt 会获得该地址并将它保存到任务控制块(OS_TCB)中，处理器的文档会告诉用户堆栈指针会指向下一个堆栈空闲位置，还是会指向最后存入数据的堆栈单元位置。

下面进行文件的移植说明：

1) 把最后 OS_CPU_SysTickHandler()，OS_CPU_SysTickInit()这两个函数的内容代码注释掉；

2) 禁用以下宏定义，因为他们涉及到上一步注释的 Systick 服务函数

```
#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_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
```

- os_cfg.h

此文件为配置内核的头文件，在这个文件，我们可以禁用信号量、互斥信号量、邮箱、队列、信号量集、定时器、内存管理，调试模式：

```
#define OS_FLAG_EN       0    //禁用信号量集
#define OS_MBOX_EN       0    //禁用邮箱
#define OS_MEM_EN            0    //禁用内存管理
#define OS_MUTEX_EN      0    //禁用互斥信号量
#define OS_Q_EN          0    //禁用队列
#define OS_SEM_EN            0    //禁用信号量
#define OS_TMR_EN            0    //禁用定时器
#define OS_DEBUG_EN      0    //禁用调试
```

也可以禁用应用软件的钩子函数和多重事件控制

```
#define OS_APP_HOOKS_EN     0
#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，也利于编译调试。等用到的时候，再开启相应的功能。

7 应用实例

至此，所有的移植已经完成，如需更详细的说明整个移植过程，请参考 AN-1018.pdf，接下来我们将编写应用相关的代码，其中有一些入门知识需要说明，uCOS-II 可以管理多达 64 个任务，但保留了优先级为 0、1、2、3、OS_LOWEST_PRIO-3、OS_LOWEST_PRIO-2，OS_LOWEST_PRIO-1 以及 OS_LOWEST_PRIO 这 8 个任务以被将来使用，用户可以有多达 56 个应用任务，必须给每个任务赋以不同的优先级，优先级号越低，任务的优先级越高。

uCOS-II 的初始化流程为：在调用 uCOS-II 的任何其它任务之前，uCOS-II 要求用户首先调用系统初始化函数 OSInit()，且多任务的启动是用户通过调用 OSStart()实现的。然而，启动 uCOS-II 之前，用户至少要建立一个应用任务，用户可以通过传递任务地址和其它参数到以下两个函数之一来建立任务：OSTaskCreate() 或者 OSTaskCreateExt()，如以下所示：

```
main()
{
```

```
    ….
    OSInit();      /* 初始化 uC/OS-II*/
    ……
    OSTaskCreate()或 OSTaskCreateExt();
    ……
    OSStart();   /*开始多任务调度!永不返回 */
}
```

在 UE-STM32F103 开发板上有 3 个 LED，我们将创建两个任务，分别控制这 3 个 LED，具体代码如下所示：

- app.c

```
#include <includes.h>
static  OS_STK  led1_task_stk[LED1_TASK_STK_SIZE];  //开辟任务堆栈
static  OS_STK  led2_task_stk[LED1_TASK_STK_SIZE ]; //开辟任务堆栈
static void systick_init(void);                              //函数声明
static void systick_init(void)
{
    RCC_ClocksTypeDef  rcc_clocks;
    RCC_GetClocksFreq(&rcc_clocks);                  //调用标准库函数，获取系统时钟。
    SysTick_Config(rcc_clocks.HCLK_Frequency / OS_TICKS_PER_SEC);   //初始化并使能
SysTick
}
static void led1_task(void *para)
{
    para = para;
    while(1)
    {
          GPIO_SetBits(GPIOD, GPIO_Pin_7);
          OSTimeDlyHMSM(0,0,1,0);       //1s 延时，释放 CPU 控制权
          GPIO_ResetBits(GPIOD, GPIO_Pin_7);
          OSTimeDlyHMSM(0,0,1,0);       //1s 延时，释放 CPU 控制权
    }
}
static void led2_task(void *para)
{
    para = para;
    while(1)
    {
          GPIO_SetBits(GPIOD, GPIO_Pin_5);
          GPIO_SetBits(GPIOD, GPIO_Pin_6);
          OSTimeDlyHMSM(0,0,0,500);         //500ms 延时，释放 CPU 控制权
          GPIO_ResetBits(GPIOD, GPIO_Pin_5);
          GPIO_ResetBits(GPIOD, GPIO_Pin_6);
          OSTimeDlyHMSM(0,0,0,500);         //500ms 延时，释放 CPU 控制权
    }
```

合肥电子科技

```
}

int main(void)
{
    BSP_Init();
    OSInit();
    systick_init();
    OSTaskCreate(led1_task, 0, &led1_task_stk[LED1_TASK_STK_SIZE - 1],
LED1_TASK_PRIO);
    OSTaskCreate(led2_task, 0, &led2_task_stk[LED2_TASK_STK_SIZE - 1],
LED2_TASK_PRIO);
    OSStart();
    return 0;
}
```

- app_cfg.h

```
/* task priority */
#define LED1_TASK_PRIO      4
#define LED2_TASK_PRIO      6

/* task stack size */
#define LED1_TASK_STK_SIZE      80
#define LED2_TASK_STK_SIZE      80
```

- Bsp.c

```
#include <includes.h>
static  void  BSP_LED_Init(void);
void  BSP_Init (void)
{
     SystemInit();
     BSP_LED_Init();                    /* Initialize the LED  */
}

static  void BSP_LED_Init(void)
{
     GPIO_InitTypeDef GPIO_InitStructure;
     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);  //使能时钟
     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;    //LED_pin
     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
     GPIO_Init(GPIOD, &GPIO_InitStructure);
}
```

- Bsp.h

```
#ifndef  __BSP_H
```

合泰电子科技

```
#define  __BSP_H
void     BSP_Init(void);
#endif
```

# 说明

**本开发板可以以下几个模块配套使用，模块详情请登录官网查询，或直接进入店铺购买**

- 双路 RS485/422 转 RS232 模块
- GPS 模块
- GPRS 模块
- Zigbee 模块
- WIFI 模块
- 陀螺仪模块（支持 MPU6050 加速度+陀螺仪、HMC5883 磁力计、BMP085 气压计）
- 7 寸液晶屏模块

**合嵌电子提供良好电磁兼容性的嵌入式开发板，陆续将推出的嵌入式开发板如下：**

- STM32F103 开发板、
- STM32F407 开发板、
- AT91SAM9260 开发板（ARM9 系列）、
- LPC1768 开发板、
- LPC1788 开发板

**敬请关注，同时诚招各地加盟代理商，欢迎联系洽谈。**

## 联系方式

电话：0550-3789700
店铺：UETECH.TAOBAO.COM
邮箱：UE_TECH@126.COM
网站：WWW.UE-TECH.NET　　WWW.UETECH.NET
地址：安徽省滁州市南谯区花园西路 82 号（科技创业中心 1 幢 302 室）