

1.移植准备: 首先准备移植所需的基础工程,我们就拿我们的基础工程 LED 灯实验。

2.UCOSIII 源码: 我们移植 UCOSIII 肯定需要 UCOSIII 源码了,这里我们需要两个文件:一个是 UCOSIII 的源码,一个是 Micrium 官方在 STM32F4xx 上移植好的工程文件。

资料 > 2 UCOS学习资料 > UCOSIII资料

搜索"UCOSIII资料"

名称	修改日期	类型	大小
UCOSIII 3.03	2014/10/30 22:31	文件夹	
UCOSIII 3.04	2014/10/30 22:25	文件夹	
100-uCOS-III-ST-STM32-003.pdf	2014/11/3 22:57	看图王 PDF 文件	19,416 KB
UCOS_III_配置与初始化.pdf	2014/11/3 22:57	看图王 PDF 文件	114 KB
UCOSIII 思维导图.pdf	2014/11/4 10:30	看图王 PDF 文件	442 KB
UCOSIII 源码.zip	2014/11/3 22:57	好压 ZIP 压缩文件	11,739 KB
uCOS-III的任务调度算法研究.pdf	2014/11/3 14:47	看图王 PDF 文件	674 KB
ucos-iii知识点总结.doc	2014/11/3 14:47	Microsoft Word ...	48 KB

解压后的源码

需要注意一下两点!!!!

1、从图 4.2.2 可以看出 Micrium 官方是在 STM32F429 上移植的,并且 UCOSIII 的版本是 3.04。从第一章 UCOSII 的移植教程中我们可以看出有一些中间文件需要我们来实现。UCOSIII 移植也是一样的,既然 Micrium 已经在 STM32F4 上移植好了 UCOSIII,那么为了方便,这些中间文件我们就直接使用 Micrium 已经编写好的, Micrium 官方虽然是在 STM32F429 上移植的,但是完全可以应用在 STM32F407 上!

2、我们在移植的过程中会将 Micrium 官方使用的 3.04 版本的 UCOSIII 用 3.03 版本替换掉。我在移植 3.04 版本 UCOSIII 的时候遇到了这样一个问题:一旦调用 OSStatTaskCPUUsageInit()函数就会进入 hardfault,如果这时选择-O1 或者-O2 优化的话就没有问题,如果选择-O0 优化的话就会出现这种问题,开发环境使用的 MDK 5.11A,不知是 KEIL 问题还是 UCOSIII 3.04 版本的问题,所以为了保险起见我们使用 3.03 版本的 UCOSIII。另外,目前 UCOSIII 的资料基本都是基于 UCOSIII 3.03 版本的,所以这也是我们选择 3.03 版本 UCOSIII 的另一个主要原因。如果一定要使用 UCOSIII 3.04 的话,使用 KEIL 时一定要选择-O1 或者-O2 优化。

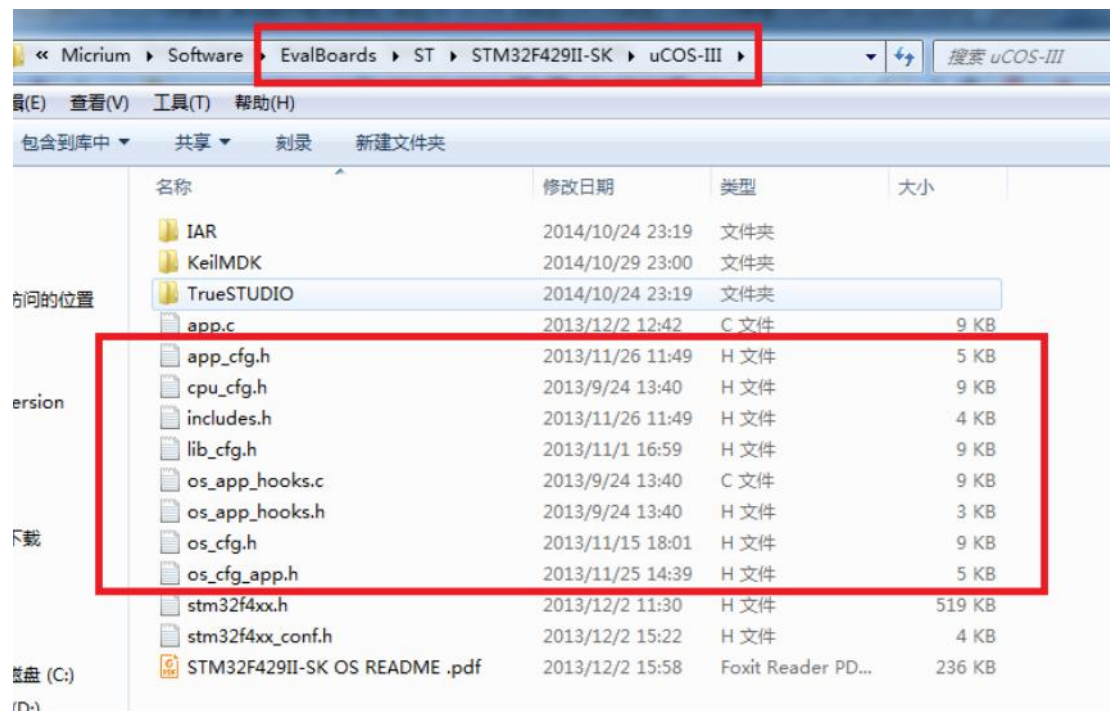
我们打开 Micrium 官方移植好的工程,也就是我们下载下来的 UCOSIII 3.04 源码,打开后如图所示。

UCOSIII资料 > UCOSIII 3.04 > Micrium > Software	搜索"Software"		
名称	修改日期	类型	大小
EvalBoards	2014/10/24 23:19	文件夹	
uC-CPU	2014/10/24 23:19	文件夹	
uC-LIB	2014/10/24 23:19	文件夹	
uCOS-III	2014/10/24 23:19	文件夹	

在图中有四个文件夹: EvalBoards、uC-CPU、uC-LIB 和 uCOS-III,这四个文件的内容如下:

1、EvalBoards 文件夹

这个文件里面就是关于 STM32F429 的工程文件，我们是在 STM32F407 上移植的，我们打开这个文件如图下所示。



在图中红框圈起来的是我们移植时候需要添加到我们的工程中的文件，一共有 8 个文件。

2、uC-CPU 文件夹

这个文件里面是与 CPU 相关的代码，有下面几个文件：

1) cpu_core.c 文件

该文件包含了适用于所有 CPU 架构的 C 代码。该文件包含了用来测量中断关闭事件的函数(中断关闭和打开分别由 CPU_CRITICAL_ENTER()和 CPU_CRITICAL_EXIT()两个宏实现)，还包含一个可模仿前导码零计算的函数(以防止 CPU 不提供这样的指令)，以及一些其他的函数。

2) cpu_core.h 文件

包含 cpu_core.c 中函数的原型声明，以及用来测量中断关闭时间变量的定义。

3) cpu_def.h 文件

包含 uC/CPU 模块使用的各种#define 常量。

4) cpu.h 文件

包含了一些类型的定义，使 UCOSIII 和其他模块可与 CPU 架构和编译器字宽度无关。在该文件中用户能够找到 CPU_INT16U、CPU_INT32U、CPU_FP32 等数据类型的定义。该文件还指定了 CPU 使用的是大端模式还是小端模式，定义了 UCOSIII 使用的 CPU_STK 数据类型，定义了 CPU_CRITICAL_ENTER()和 CPU_CRITICAL_EXIT()，还包括一些与 CPU 架构相关的函数的声明。

5) cpu_a.asm 文件

该文件包含了一些用汇编语言编写的函数，可用来开中断和关中断，计算前导零(如果 CPU 支持这条指令)，以及其他一些只能用汇编语言编写的与 CPU 相关的函数，这个文件中的函数可以从 C 代码中调用。

6) cpu_c.c 文件

包含了一些基于特定 CPU 架构但为了可移植而用 C 语言编写的函数 C 代码，作为一个普通原则，除非汇编语言能显著提高性能，否则尽量用 C 语言编写函数。

注意，上面的 `cpu.h`、`cpu_a.asm` 和 `cpu_c.c` 这三个文件，是在 `uC-CPU` 文件夹中 `ARM-Cortex-M4` 文件夹下的，我们打开 `ARM-Cortex-M4` 文件如图下所示。



从图中可以看出一共有三个文件：`GNU`、`IAR`、`RealView`，这三个文件中都有 `cpu.h`、`cpu_a.asm` 和 `cpu_c.c` 这三个文件。我们使用的是 `KEIL`，所以我们在移植 `UCOSIII` 的时候选择 `RealView` 中的文件。在我们接下来的讲解中会看到同样的设计，根据不同的编译平台有不同的处理。

3 、uC-LIB 文件

`uC-LIB` 是由一些可移植并且与编译器无关的函数组成，`UCOS III` 不使用 `uC-LIB` 中的函数，但是 `UCOS III` 和 `uC-CPU` 假定 `lib_def.h` 是存在的，`uC-LIB` 包含以下几个文件：

1) `lib_ascii.h` 和 `lib_ascii.c` 文件

提供 `ASCII_ToLower()`、`ASCII_ToUpper()`、`ASCII_IsAlpha()`和 `ASCII_IsDig()`等函数，它们可以分别替代标准库函数 `tolower()`、`toupper()`、`isalpha()`和 `isdigit()`等。

2) `lib_def.h` 文件

定义了许多常量，如 `RTUE/FALSE`、`YES/NO`、`ENABLE/DISABLE`，以及各种进制的常量。但是，该文件中所有 `#define` 常量都以 `DEF_` 打头，所以上述常量的名字实际上为 `DEF_TRUE/DEF_FALSE`、`DEF_YES/DEF_NO`、`DEF_ENABLE/DEF_DISABLE` 等。该文件还为常用数学计算定义了宏。

3) `lib_math.h` 和 `lib_math.c` 文件

包含了 `Math_Rand()`、`Math_SetRand()`等函数的源代码，可用来替代标准库函数 `rand()`、`srand()`。

4) `lib_mem.c` 和 `lib_mem.h` 文件

包含了 `Mem_Clr()`、`Mem_Set()`、`Mem_Copy()`和 `Mem_Cmp()`等函数的源代码，可用来替代标准库函数 `memset()`、`memcpy()`和 `memcmp()`等。

5) `lib_str.c` 和 `lib_str.h` 文件

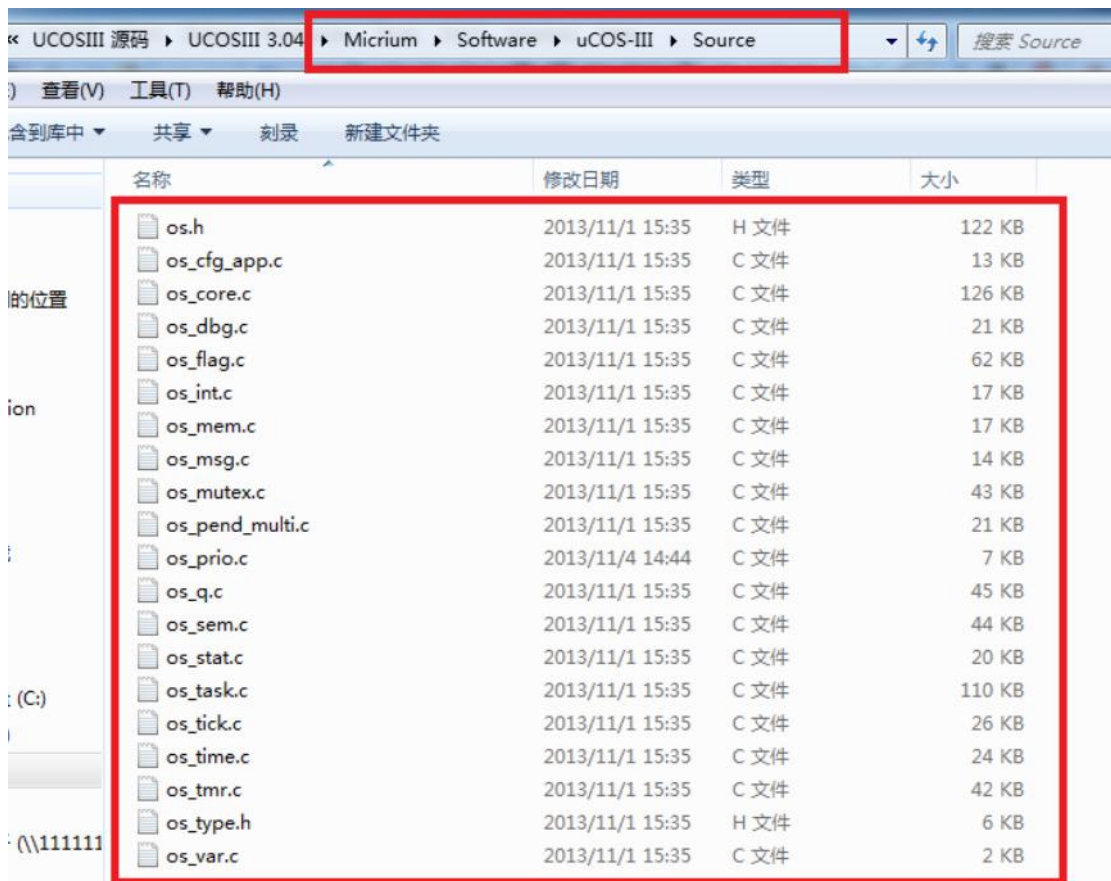
包含了 `Str_Lenr()`、`Str_Copy()`和 `Str_Cmp()`等函数的源代码，可用于替代标准库函数 `strlen()`、`strcpy()`和 `strcmp()`等。

6) `lib_mem_a.asm` 文件

包含了 `lib_mem.c` 函数的汇编优化版。

4 、uCOS-III 文件

这个文件夹中有两个文件 `Ports` 和 `Source`，`Ports` 文件为与 `CPU` 平台有关的文件，`Source` 文件夹里面为 `UCOSIII 3.04` 的源码，我们打开 `Source` 文件夹如图所示。



UCOSIII 3.04 和 UCOSIII 3.03 源码的文件都是一样的，不同的是各个文件里面的有些函数做了修改，UCOSIII 源码各个文件内容如表所示。

文件	描述
os.h	包含 UCOSIII 的主要头文件，声明了常量、宏、全局变量、函数原型等
os_cfg_app.c	根据 os_cfg_app.h 中的宏定义声明变量和数组
os_core.c	UCOSIII 的内核功能模块
os_dbg.c	包含内核调试或 uC/Probe 使用的常量的声明
os_flag.c	包含事件标志的管理代码。
os_int.c	包含中断处理任务的代码。
os_mem.c	包含 UCOSIII 固定大小的存储分区的管理代码。
os_msg.c	包含消息处理的代码。
os_mutex.c	包含互斥型信号量的管理代码
os_pend_multi.c	包含允许任务同时等待多个信号量或多个消息队列的代码。
os_prio.c	包含位映射表的管理代码，用于追踪那些已经就绪的任务。
os_q.c	包含消息队列的管理代码。
os_sem.c	包含信号量的管理代码。
os_stat.c	包含统计任务的代码。

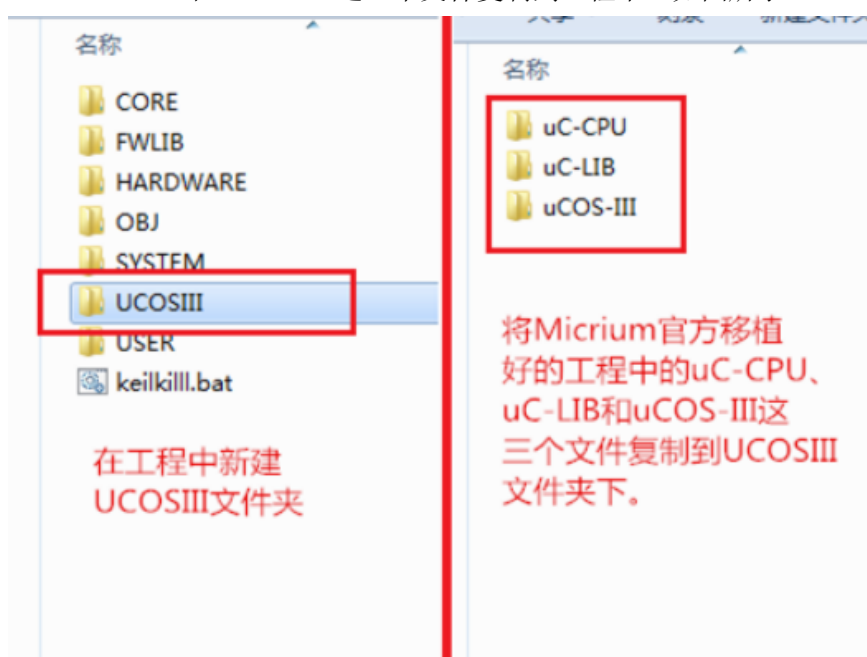
os_task.c	包含任务的管理代码。
os_tick.c	包含可管理正在延时和超时等待的任务的代码。
os_time.c	包含可是任务延时一段时间的代码。
os_tmr.c	包含软件定时器的管理代码。
os_type.h	包含 UCOSIII 数据类型的声明。
os_var.c	包含 UCOSIII 的全局变量。

3.UCOS III

1、向工程中添加相应的文件

1) 新建相应的文件夹

在工程目录中新建一个 UCOSIII 文件夹，然后将我们下载的 Micrium 官方移植工程中的 uC-CPU、uC-LIB 和 UCOS-III 这三个文件复制到工程中，如图所示。



我们还需要在 UCOSIII 文件中再新建两个文件：UCOS_BSP 和 UCOS_CONFIG，如图所示。



2) 向 UCOS_CONFIG 添加文件

复制 Micrium 官方移植好的工程中的相关文件复制到 UCOS_CONFIG 文件夹下，这些文件如图所示，这些文件在 Micrium 官方移植工程中的路径为：

Micrium->Software->EvalBoards->ST->STM32F429II-SK->uCOS-III

名称	修改日期	类型	大小
app_cfg.h	2013/11/26 11:49	H 文件	5 KB
cpu_cfg.h	2013/9/24 13:40	H 文件	9 KB
includes.h	2013/11/26 11:49	H 文件	4 KB
lib_cfg.h	2013/11/1 16:59	H 文件	9 KB
os_app_hooks.c	2013/9/24 13:40	C 文件	9 KB
os_app_hooks.h	2013/9/24 13:40	H 文件	3 KB
os_cfg.h	2013/11/15 18:01	H 文件	9 KB
os_cfg_app.h	2017/9/18 10:39	H 文件	5 KB

3) 向 UCOS_BSP 添加文件

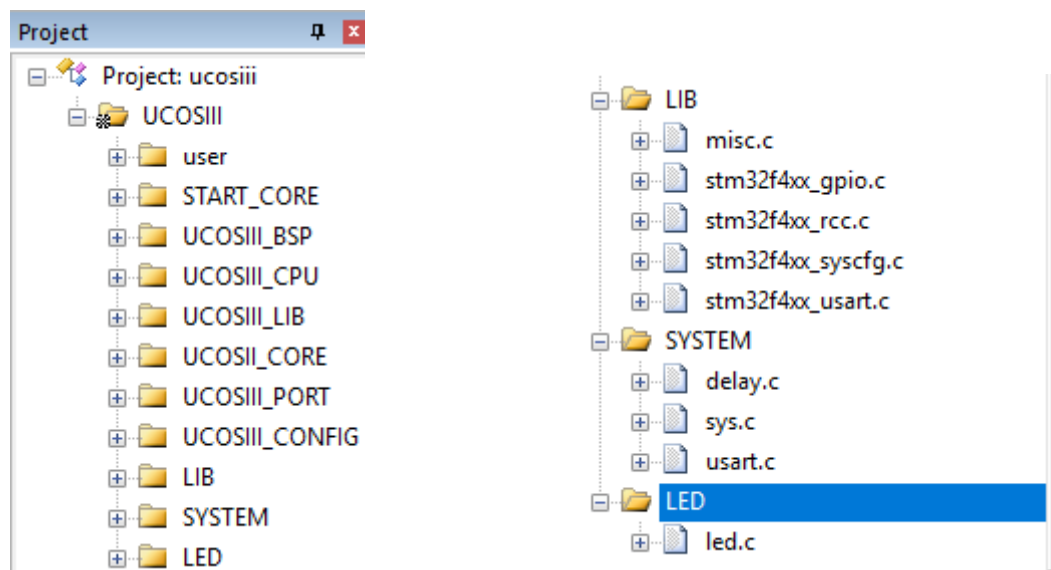
同样复制 Micrium 官方移植好的工程中的相关文件到 UCOS_BSP 文件下，需要复制的文件如图所示，这些文件在 Micrium 官方移植工程中的路径为：

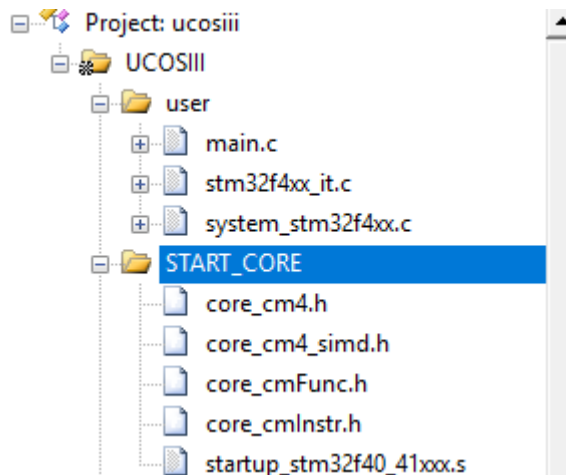
Micrium->Software->EvalBoards->ST->STM32F429II-SK->BSP

名称	修改日期	类型
bsp.c	2017/9/18 10:36	C 文件
bsp.h	2017/9/18 10:10	H 文件

4) 向工程中添加 分组

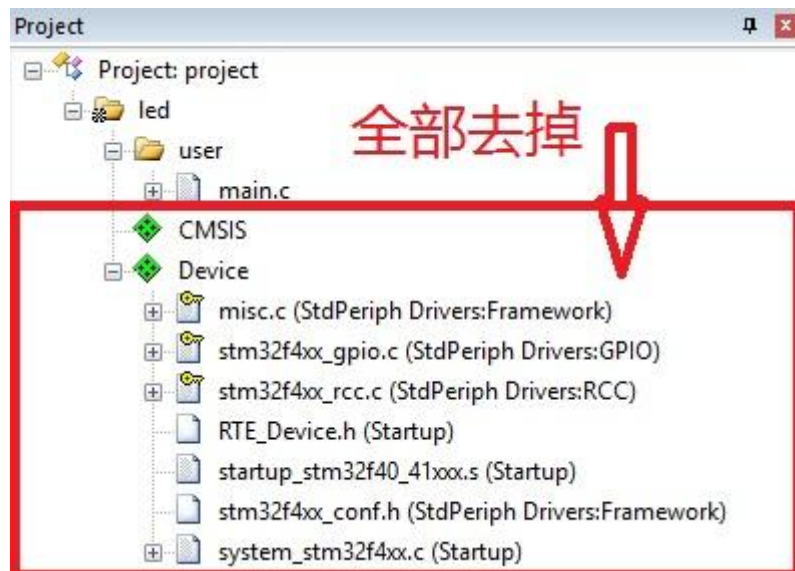
我们在上面已经准备好了所需的文件，我们还要将这些文件添加到我们的工程中，我们在 KEIL 工程中新建如图所示的分组。



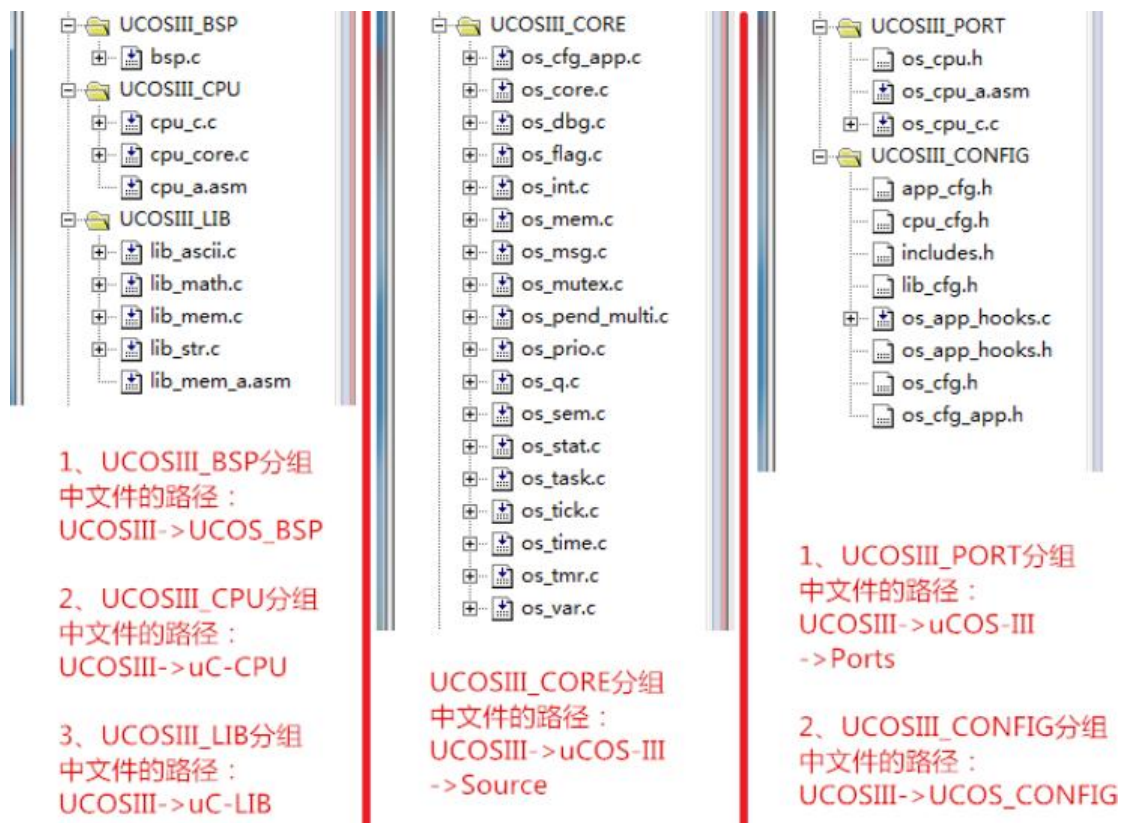


注意！上图这些是该工程的所有文件夹！LIB 下放的是需要使用的库文件，START_CORE 放的是启动代码！如果大家也和我一样用的是之前 LED 的工程则需要将下面这些去掉。因为我们的工程文件夹已经有了所有需要的文件了！

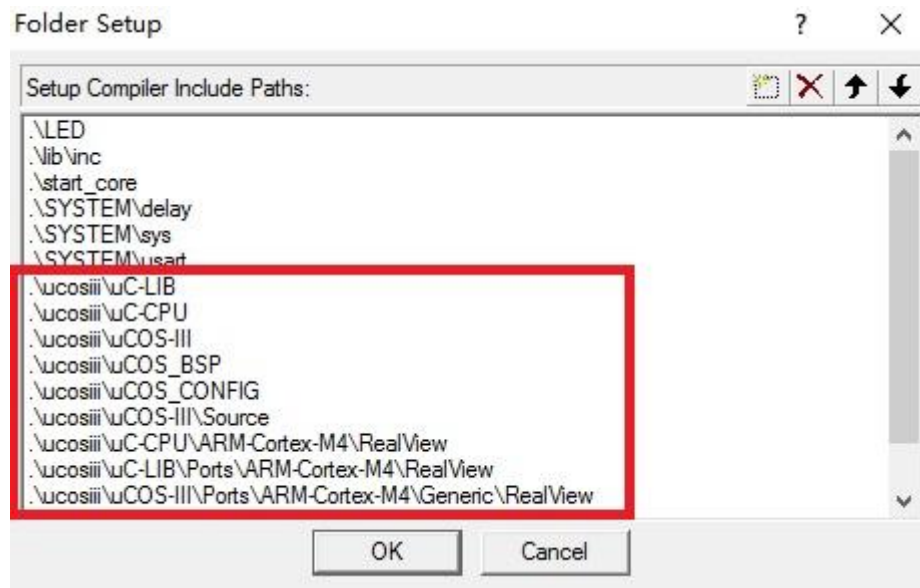
我用的是：笔记->002->代码->led



工程中分组建立完成以后我们就需要向新建的各个分组中添加文件了，按照如图所示向各个分组中添加文件。



向各个分组添加完文件以后我们还需要添加相应的头文件路径，按照图所示添加头文件路径。



到这一步我们编译一下工程，会有如图 4.3.8 所示的错误提示，提示我们在 bsp.c 文件中 BSP_IntInit()和 BSP_PeriphEn()这两个函数未定义，这里我们先不管这两个错误。

```
linking...
.\Objects\UCOSIII.axf: Error: L6218E: Undefined symbol BSP_IntInit (referred from bsp.o).
.\Objects\UCOSIII.axf: Error: L6218E: Undefined symbol BSP_PeriphEn (referred from bsp.o).
```

4.修改 bsp.c 和 和 bsp.h

我们打开 bsp.c 文件，里面有很多的代码我们只需要其中的很少一部分关于 DWT 的

代码,因此我们要做相应的修改,在修改之前我们稍微讲解一下 Cortex-M3/M4 的跟踪组件。

在 CM3/CM4 中有 3 种跟踪源: ETM、ITM 和 DWT,要想使用 ETM、ITM 和 DWT 的话要将 DEMCR 寄存器的 TRCENA 位(bit24)置 1, DEMCR 寄存器地址为: 0XE000EDFC, 在我们的光盘中 STM32 参考资料下的《Cortex-M3 与 M4 权威指南》(英文名为《The DefinitiveGuide to ARM Cortex-M3 and Cortex-M4 Processors, 3rd Edition》)的 501 页有详细的讲解,感兴趣的朋友可以自行查阅一下。在 DWT 组件中有一个 CYCCNT 寄存器,这个寄存器用来对时钟周期计数,我们可以使用这个寄存器来测量执行某个任务所花费的时间。

DWT 组件有多个寄存器,我们这里只使用 DWT 的控制寄存器 CTRL、CYCCNT 寄存器, CTRL 寄存器地址为 0XE0001000、CYCCNT 寄存器地址为 0XE0001004。如果我们要使用时钟计数功能需要将 CTRL 寄存器的 bit0 置 1。

这里大家可以直接使用的我替共的 UCOSIII->ucosiii->uCOS_BSP 下的 bsp.c 文件,修改后的 bsp.c 可以自行参考,这里为了方便讲解我们删掉了一些函数的英文注释,改为中文注释。

我们还要对 bsp.h 文件做相应的修改,修改后的 bsp.h 的文件如下所示, bsp.h 文件很简单,就是添加一些头文件。

```
#ifndef BSP_PRESENT
#define BSP_PRESENT
#ifdef BSP_MODULE
#define BSP_EXT
#else
#define BSP_EXT extern
#endif
#include <stdio.h>
#include <stdarg.h>
#include <cpu.h>
#include <cpu_core.h>
#include <lib_def.h>
#include <lib_ascii.h>
#include <stm32f4xx_conf.h>
#endif
```

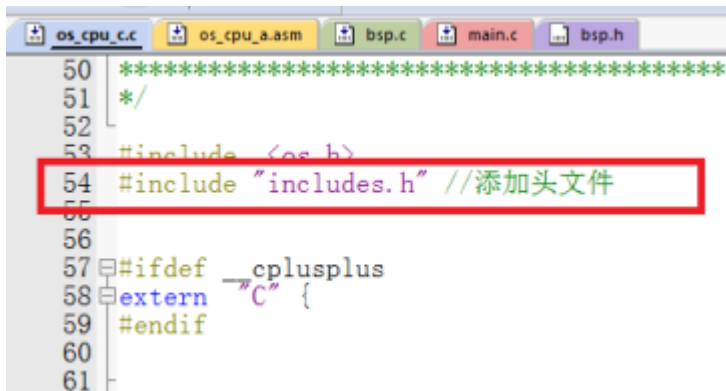
5.修改 os_cpu_a.asm

os_cpu_a.asm 文件大家根据我提供的 os_cpu_a.asm 文件来修改自己工程中的 os_cpu_a.asm 文件,修改完 os_cpu_a.asm 文件后我们编译一下工程提示如图所示错误,提示我们在 os_cpu_c.c 使用到的 OS_CPU_FP_Reg_Pop()和 OS_CPU_FP_Reg_Push()这两个函数未定义,这个错误我们会在下面讲解 os_cpu_c.c 文件修改的时候讲到。

```
linking...
..\OBJ\Template.axf: Error: L6218E: Undefined symbol OS_CPU_FP_Reg_Pop (referred from os_cpu_c.o).
..\OBJ\Template.axf: Error: L6218E: Undefined symbol OS_CPU_FP_Reg_Push (referred from os_cpu_c.o).
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 2 error messages.
```

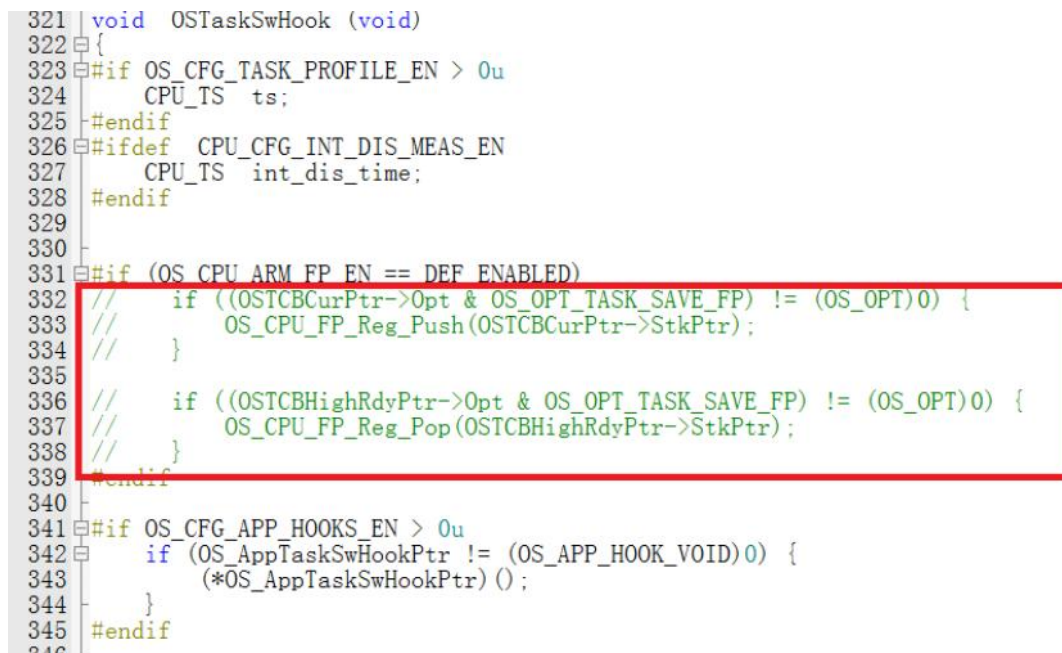
6.修改 os_cpu_c.c

首先我们在 os_cpu_c.c 文件开始部分添加 includes.h 头文件,如图所示



```
50 *****
51 */
52
53 #include <os.h>
54 #include "includes.h" //添加头文件
55
56
57 #ifdef cplusplus
58 extern "C" {
59 #endif
60
61
```

在上面我们知道修改完 `os_cpu_a.asm` 文件后有两个错误，我们需要修改 `os_cpu_c.c` 文件来消除这两个错误。我们找到在 `os_cpu_c.c` 文件中的 `OSTaskSwHook()` 函数使用到这两个函数，这两个函数分别用来对 FPU 寄存器进行出栈和入栈处理的，我们对于 FPU 寄存器的入栈和出栈有相应的处理，这里不使用这两个函数，因此把 `OSTaskSwHook()` 函数中关于 `OS_CPU_FP_Reg_Pop()` 和 `OS_CPU_FP_Reg_Push()` 这两个函数的代码注销掉，如图所示。



```
321 void OSTaskSwHook (void)
322 {
323 #if OS_CFG_TASK_PROFILE_EN > 0u
324     CPU_TS ts;
325 #endif
326 #ifdef CPU_CFG_INT_DIS_MEAS_EN
327     CPU_TS int_dis_time;
328 #endif
329
330
331 #if (OS_CPU_ARM_FP_EN == DEF_ENABLED)
332 // if ((OSTCBCurPtr->Opt & OS_OPT_TASK_SAVE_FP) != (OS_OPT)0) {
333 //     OS_CPU_FP_Reg_Push(OSTCBCurPtr->StkPtr);
334 // }
335
336 // if ((OSTCBHighRdyPtr->Opt & OS_OPT_TASK_SAVE_FP) != (OS_OPT)0) {
337 //     OS_CPU_FP_Reg_Pop(OSTCBHighRdyPtr->StkPtr);
338 // }
339 #endif
340
341 #if OS_CFG_APP_HOOKS_EN > 0u
342 if (OS_AppTaskSwHookPtr != (OS_APP_HOOK_VOID)0) {
343     (*OS_AppTaskSwHookPtr)();
344 }
345 #endif
346
```

注销掉 `OSTaskSwHook()` 函数中调用的 `OS_CPU_FP_Reg_Pop()` 和 `OS_CPU_FP_Reg_Push()` 这两个函数后再次编译一下工程，应该就没有错误了。

同移植 UCOSII 一样，我们还需要修改堆栈初始化函数 `OSTaskStkInit()`，修改后的 `OSTaskStkInit()` 函数如下所示。这个函数基本上和我们移植 UCOSII 时 `os_cpu_c.c` 文件中的 `OSTaskStkInit()` 函数一样

```
CPU_STK *OSTaskStkInit (OS_TASK_PTR p_task,
void *p_arg,
CPU_STK *p_stk_base,
CPU_STK *p_stk_limit,
CPU_STK_SIZE stk_size,
OS_OPT opt)
{
    CPU_STK *p_stk;
```

```

(void)opt;
p_stk = &p_stk_base[stk_size];
p_stk = (CPU_STK *)((CPU_STK)(p_stk) & 0xFFFFF8); //堆栈 8 字节对齐
#if (__FPU_PRESENT==1)&&(__FPU_USED==1)
    *--p_stk = (CPU_STK)0x00000000u;
    *--p_stk = (CPU_STK)0x00001000u; //FPSCR
    *--p_stk = (CPU_STK)0x00000015u; //s15
    *--p_stk = (CPU_STK)0x00000014u; //s14
    *--p_stk = (CPU_STK)0x00000013u; //s13
    *--p_stk = (CPU_STK)0x00000012u; //s12
    *--p_stk = (CPU_STK)0x00000011u; //s11
    *--p_stk = (CPU_STK)0x00000010u; //s10
    *--p_stk = (CPU_STK)0x00000009u; //s9
    *--p_stk = (CPU_STK)0x00000008u; //s8
    *--p_stk = (CPU_STK)0x00000007u; //s7
    *--p_stk = (CPU_STK)0x00000006u; //s6
    *--p_stk = (CPU_STK)0x00000005u; //s5
    *--p_stk = (CPU_STK)0x00000004u; //s4
    *--p_stk = (CPU_STK)0x00000003u; //s3
    *--p_stk = (CPU_STK)0x00000002u; //s2
    *--p_stk = (CPU_STK)0x00000001u; //s1
    *--p_stk = (CPU_STK)0x00000000u; //s0
#endif
    *--p_stk = (CPU_STK)0x01000000u; //xPSR
    *--p_stk = (CPU_STK)p_task; // Entry Point
    *--p_stk = (CPU_STK)OS_TaskReturn; //R14 (LR)
    *--p_stk = (CPU_STK)0x12121212u; // R12
    *--p_stk = (CPU_STK)0x03030303u; // R3
    *--p_stk = (CPU_STK)0x02020202u; // R2
    *--p_stk = (CPU_STK)p_stk_limit; // R1
    *--p_stk = (CPU_STK)p_arg; //R0
#if (__FPU_PRESENT==1)&&(__FPU_USED==1)
    *--p_stk = (CPU_STK)0x00000031u; //s31
    *--p_stk = (CPU_STK)0x00000030u; //s30
    *--p_stk = (CPU_STK)0x00000029u; //s29
    *--p_stk = (CPU_STK)0x00000028u; //s28
    *--p_stk = (CPU_STK)0x00000027u; //s27
    *--p_stk = (CPU_STK)0x00000026u; //s26
    *--p_stk = (CPU_STK)0x00000025u; //s25
    *--p_stk = (CPU_STK)0x00000024u; //s24
    *--p_stk = (CPU_STK)0x00000023u; //s23
    *--p_stk = (CPU_STK)0x00000022u; //s22
    *--p_stk = (CPU_STK)0x00000021u; //s21
    *--p_stk = (CPU_STK)0x00000020u; //s20

```

```

        *(&p_stk) = (CPU_STK)0x00000019u; //s19
        *(&p_stk) = (CPU_STK)0x00000018u; //s18
        *(&p_stk) = (CPU_STK)0x00000017u; //s17
        *(&p_stk) = (CPU_STK)0x00000016u; //s16
    #endif

    *(&p_stk) = (CPU_STK)0x11111111u; //R11
    *(&p_stk) = (CPU_STK)0x10101010u; //R10
    *(&p_stk) = (CPU_STK)0x09090909u; //R9
    *(&p_stk) = (CPU_STK)0x08080808u; // R8
    *(&p_stk) = (CPU_STK)0x07070707u; //R7
    *(&p_stk) = (CPU_STK)0x06060606u; //R6
    *(&p_stk) = (CPU_STK)0x05050505u; // R5
    *(&p_stk) = (CPU_STK)0x04040404u; //R4

    return (p_stk);
}

```

7.修改 os_cfg_app.h

我们还需要修改 os_cfg_app.h 文件，os_cfg_app.h 主要是对 UCOSIII 内部一些系统任务的配置，如任务优先级、任务堆栈、UCOSIII 的系统时钟节拍等等，os_cfg_app.h 文件都是一些宏定义，比较简单，文件代码如下。

```

#ifndef OS_CFG_APP_H
#define OS_CFG_APP_H

#define OS_CFG_MSG_POOL_SIZE 100u //消息数量
#define OS_CFG_ISR_STK_SIZE 128u //中断任务堆栈大小
//任务堆栈深度，比如当定义为 10 时表示当任务堆栈剩余百分之 10%时就说明堆栈为
空

#define OS_CFG_TASK_STK_LIMIT_PCT_EMPTY 10u
//*****空闲任务*****
#define OS_CFG_IDLE_TASK_STK_SIZE 128u //空任务堆栈大小
//*****中断服务管理任务*****
#define OS_CFG_INT_Q_SIZE 10u //中断队列大小
#define OS_CFG_INT_Q_TASK_STK_SIZE 128u //中断服务管理任务大小
//*****统计任务*****
//统计任务优先级，倒数第二个优先级，最后一个优先级是空闲任务的。
#define OS_CFG_STAT_TASK_PRIO (OS_CFG_PRIO_MAX-2u)
// OS_CFG_STAT_TASK_RATE_HZ 用于统计任务计算 CPU 使用率，统计任务通过统计在
//(1/ OS_CFG_STAT_TASK_RATE_HZ)秒的时间内 OSStatTaskCtr 能够达到的最大值来得到
//CPU 使用率，OS_CFG_STAT_TASK_RATE_HZ 值应该在 1~10Hz
#define OS_CFG_STAT_TASK_RATE_HZ 10u
#define OS_CFG_STAT_TASK_STK_SIZE 128u //统计任务堆栈
//*****时钟节拍任务*****
#define OS_CFG_TICK_RATE_HZ 200u //系统时钟节拍频率
//时钟节拍任务，一般设置一个相对较高的优先级
#define OS_CFG_TICK_TASK_PRIO 1u

```

```

#define OS_CFG_TICK_TASK_STK_SIZE 128u //时钟节拍任务堆栈大小
#define OS_CFG_TICK_WHEEL_SIZE 17u //时钟节拍列表大小
//*****定时任务*****
#define OS_CFG_TMR_TASK_PRIO 2u //定时任务优先级
#define OS_CFG_TMR_TASK_RATE_HZ 100u //定时频率，一般为 100Hz
#define OS_CFG_TMR_TASK_STK_SIZE 128u //定时任务堆栈大小
#define OS_CFG_TMR_WHEEL_SIZE 17u //定时器列表数量
#endif

```

在 UCOSIII 中有五个系统任务：空闲任务、时钟节拍任务、统计任务、定时任务和中断服务管理任务，在系统初始化的时候至少要创建两个任务：空闲任务和时钟节拍任务。空闲任务的优先级应该为最低 `OS_CFG_PRIO_MAX-1`，如果使用中断服务管理任务的话那么中断服务管理任务的优先级应该为最高 `0`。其他 `3` 个任务的优先级用户可以自行设置，本手册中我们将统计任务设置为 `OS_CFG_PRIO_MAX-2`，既倒数第二个优先级；时钟节拍任务也需要一个高优先级，我们将优先级 `1` 分配给时钟节拍任务；将优先级 `2` 分配给定时器任务。所以优先级 `0`、`1`、`2`、`OS_CFG_PRIO_MAX-2` 和 `OS_CFG_PRIO_MAX-1` 这 `5` 个优先级用户应用程序是不能使用的！

8.软件设计

移植完成后就需要编写测试软件测试我们移植是否正确，我们建立 `3` 个任务，其中两个任务分别用于 `LED0`、`LED1`、`LED2`、`LED3` 闪烁，另外一个任务用于测试浮点计算，软件代码如下。（`main.c` 中）

```

#include "sys.h"
#include "delay.h"
#include "usart.h"
#include "led.h"
#include "includes.h"
#include "os_app_hooks.h"

//任务优先级
#define START_TASK_PRIO      3
//任务堆栈大小
#define START_STK_SIZE      512
//任务控制块
OS_TCB StartTaskTCB;
//任务堆栈
CPU_STK START_TASK_STK[START_STK_SIZE];
//任务函数
void start_task(void *p_arg);

//任务优先级
#define LED0_TASK_PRIO      4
//任务堆栈大小
#define LED0_STK_SIZE      128
//任务控制块

```



```
OS_TCB Led0TaskTCB;
//任务堆栈
CPU_STK LED0_TASK_STK[LED0_STK_SIZE];
void led0_task(void *p_arg);

//任务优先级
#define LED1_TASK_PRIO      5
//任务堆栈大小
#define LED1_STK_SIZE      128
//任务控制块
OS_TCB Led1TaskTCB;
//任务堆栈
CPU_STK LED1_TASK_STK[LED1_STK_SIZE];
//任务函数
void led1_task(void *p_arg);

//任务优先级
#define LED2_TASK_PRIO      6
//任务堆栈大小
#define LED2_STK_SIZE      128
//任务控制块
OS_TCB Led2TaskTCB;
//任务堆栈
CPU_STK LED2_TASK_STK[LED1_STK_SIZE];
//任务函数
void led2_task(void *p_arg);

//任务优先级
#define LED3_TASK_PRIO      7
//任务堆栈大小
#define LED3_STK_SIZE      128
//任务控制块
OS_TCB Led3TaskTCB;
//任务堆栈
CPU_STK LED3_TASK_STK[LED1_STK_SIZE];
//任务函数
void led3_task(void *p_arg);

//任务优先级
#define FLOAT_TASK_PRIO     8
//任务堆栈大小
#define FLOAT_STK_SIZE     128
//任务控制块
OS_TCB  FloatTaskTCB;
```

```

//任务堆栈
__align(8) CPU_STKFLOAT_TASK_STK[FLOAT_STK_SIZE];
//任务函数
void float_task(void *p_arg);

int main(void)
{
    OS_ERR err;
    CPU_SR_ALLOC();

    delay_init(168);    //时钟初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //中断分组配置
    uart_init(115200);  //串口初始化
    LED_Init();         //LED 初始化

    OSInit(&err);        //初始化 UCOSIII
    OS_CRITICAL_ENTER(); //进入临界区
    //创建开始任务
    OSTaskCreate((OS_TCB *) &StartTaskTCB,    //任务控制块
                (CPU_CHAR *) "start task",    //任务名字
                (OS_TASK_PTR) start_task,      //任务函数
                (void *) 0,                    //传递给任务函数的参数
                (OS_PRIO) START_TASK_PRIO,     //任务优先级
                (CPU_STK *) &START_TASK_STK[0], //任务堆栈基地址
                (CPU_STK_SIZE) START_STK_SIZE/10, //任务堆栈深度限位
                (CPU_STK_SIZE) START_STK_SIZE,  //任务堆栈大小
                (OS_MSG_QTY) 0,                 //任务内部消息队列能够
接收的最大消息数目,为 0 时禁止接收消息
                (OS_TICK) 0,                   //当使能时间片轮转时的
时间片长度, 为 0 时为默认长度,
                (void *) 0,                    //用户补充的存储区
                (OS_OPT) OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR, //
任务选项
                (OS_ERR *) &err);              //存放该函数错误时的返回值
    OS_CRITICAL_EXIT();    //退出临界区
    OSStart(&err); //开启 UCOSIII
    while(1);
}

//开始任务函数
void start_task(void *p_arg)
{
    OS_ERR err;

```

```

    CPU_SR_ALLOC();
    p_arg = p_arg;

    CPU_Init();
    #if OS_CFG_STAT_TASK_EN > 0u
        OSStatTaskCPUUsagInit(&err);    //统计任务
    #endif

    #ifndef CPU_CFG_INT_DIS_MEAS_EN      //如果使能了测量中断关闭时间
        CPU_IntDisMeasMaxCurReset();
    #endif

    #if OS_CFG_SCHED_ROUND_ROBIN_EN    //当使用时间片轮转的时候
        //使能时间片轮转调度功能,时间片长度为 1 个系统时钟节拍, 既 1*5=5ms
        OSSchedRoundRobinCfg(DEF_ENABLED,1,&err);
    #endif

    OS_CRITICAL_ENTER(); //进入临界区
    //创建 LED0 任务
    OSTaskCreate((OS_TCB *) &Led0TaskTCB,
        (CPU_CHAR *) "led0 task",
        (OS_TASK_PTR) led0_task,
        (void *) 0,
        (OS_PRIO) LED0_TASK_PRIO,
        (CPU_STK *) &LED0_TASK_STK[0],
        (CPU_STK_SIZE) LED0_STK_SIZE/10,
        (CPU_STK_SIZE) LED0_STK_SIZE,
        (OS_MSG_QTY) 0,
        (OS_TICK) 0,
        (void *) 0,
        (OS_OPT) OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
        (OS_ERR *) &err);

    //创建 LED1 任务
    OSTaskCreate((OS_TCB *) &Led1TaskTCB,
        (CPU_CHAR *) "led1 task",
        (OS_TASK_PTR) led1_task,
        (void *) 0,
        (OS_PRIO) LED1_TASK_PRIO,
        (CPU_STK *) &LED1_TASK_STK[0],
        (CPU_STK_SIZE) LED1_STK_SIZE/10,
        (CPU_STK_SIZE) LED1_STK_SIZE,
        (OS_MSG_QTY) 0,
        (OS_TICK) 0,

```

```
(void * )0,  
(OS_OPT )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,  
(OS_ERR * )&err);
```

//创建 LED2 任务

```
OSTaskCreate((OS_TCB *)&Led2TaskTCB,  
             (CPU_CHAR *)"led2 task",  
             (OS_TASK_PTR )led2_task,  
             (void * )0,  
             (OS_PRIO )LED2_TASK_PRIO,  
             (CPU_STK * )&LED2_TASK_STK[0],  
             (CPU_STK_SIZE)LED2_STK_SIZE/10,  
             (CPU_STK_SIZE)LED2_STK_SIZE,  
             (OS_MSG_QTY )0,  
             (OS_TICK )0,  
             (void * )0,  
             (OS_OPT )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,  
             (OS_ERR * )&err);
```

//创建 LED3 任务

```
OSTaskCreate((OS_TCB *)&Led3TaskTCB,  
             (CPU_CHAR *)"led3 task",  
             (OS_TASK_PTR )led3_task,  
             (void * )0,  
             (OS_PRIO )LED3_TASK_PRIO,  
             (CPU_STK * )&LED3_TASK_STK[0],  
             (CPU_STK_SIZE)LED3_STK_SIZE/10,  
             (CPU_STK_SIZE)LED3_STK_SIZE,  
             (OS_MSG_QTY )0,  
             (OS_TICK )0,  
             (void * )0,  
             (OS_OPT )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,  
             (OS_ERR * )&err);
```

//创建浮点测试任务

```
OSTaskCreate((OS_TCB *)&FloatTaskTCB,  
             (CPU_CHAR *)"float test task",  
             (OS_TASK_PTR )float_task,  
             (void * )0,  
             (OS_PRIO )FLOAT_TASK_PRIO,  
             (CPU_STK * )&FLOAT_TASK_STK[0],  
             (CPU_STK_SIZE)FLOAT_STK_SIZE/10,  
             (CPU_STK_SIZE)FLOAT_STK_SIZE,  
             (OS_MSG_QTY )0,
```

```

        (OS_TICK      )0,
        (void          *)0,
        (OS_OPT        )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
        (OS_ERR        *)&err);
    OS_TaskSuspend((OS_TCB*)&StartTaskTCB,&err);    //挂起开始任务
    OS_CRITICAL_EXIT();    //进入临界区
}

//led0 任务函数
void led0_task(void *p_arg)
{
    OS_ERR err;
    p_arg = p_arg;
    while(1)
    {
        LED0=0;
        OSTimeDlyHMSM(0,0,0,200,OS_OPT_TIME_HMSM_STRICT,&err); //延时 200ms
        LED0=1;
        OSTimeDlyHMSM(0,0,0,500,OS_OPT_TIME_HMSM_STRICT,&err); //延时 500ms
    }
}

//led1 任务函数
void led1_task(void *p_arg)
{
    OS_ERR err;
    p_arg = p_arg;
    while(1)
    {
        LED1=~LED1;
        OSTimeDlyHMSM(0,0,0,500,OS_OPT_TIME_HMSM_STRICT,&err); //延时 500ms
    }
}

//led2 任务函数
void led2_task(void *p_arg)
{
    OS_ERR err;
    p_arg = p_arg;
    while(1)
    {
        LED2=~LED2;
        OSTimeDlyHMSM(0,0,0,800,OS_OPT_TIME_HMSM_STRICT,&err); //延时 500ms
    }
}

```



```

}

//led3 任务函数
void led3_task(void *p_arg)
{
    OS_ERR err;
    p_arg = p_arg;
    while(1)
    {
        LED3=~LED3;
        OSTimeDlyHMSM(0,0,0,200,OS_OPT_TIME_HMSM_STRICT,&err); //延时 500ms
    }
}

//浮点测试任务
void float_task(void *p_arg)
{
    CPU_SR_ALLOC();
    static float float_num=0.01;
    while(1)
    {
        float_num+=0.02;
        OS_CRITICAL_ENTER(); //进入临界区
        printf("float_num 的值为: %.4f\r\n",float_num);
        OS_CRITICAL_EXIT();   //退出临界区
        delay_ms(500);        //延时 500ms
    }
}

```

10. 下载验证



显示上面画面，代表移植成功。