FreeRTOS 移植到 STM32F103 步骤与注意事项

前言：

由于之前听过太多人抱怨移植 FreeRTOS 到 STM32 有各种各样的问题，小灯经过一年多对 FreeRTOS 的研究并在公司产品中应用，多少有些心得，接下来就由小灯以最新版的 FreeRTOS 为例一步一步移植到 STM32F103 上，并提醒大家某些需要注意的事项。本文档为非正式技术文档，故排版会有些凌乱，希望大家能提供宝贵意见以供小灯参考改进。

下面先以 IAR 移植为例，说明移植过程中的诸多注意事项，最后再以 MDK 移植时不再重复说明，所以还是建议大家先花些时间看 IAR 的移植过程，哪怕你不使用 IAR，最好也注意下那一大堆注意事项！

一、从官网下载最新版的 FreeRTOS 源码

下面的网址是官方最新源码的下载地址：

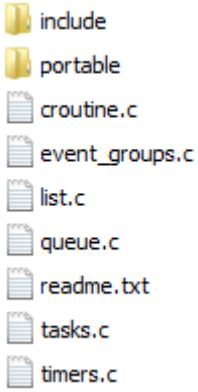
https://sourceforge.net/projects/freertos/files/latest/download?source=files

目前官方提供的最新版本是 v9.0.0， FreeRTOS 源码在解压目录下的路径为 FreeRTOS_V9.0.0rc2\FreeRTOS\Source

FreeRTOS 组织为了抢用户也是拼了命的，不信你打开 Demo 文件夹看看，里面提供了 FreeRTOS 在各种单片机上已经移植好的工程，如果建工程时遇到什么问题，可以参考下这些 Demo。

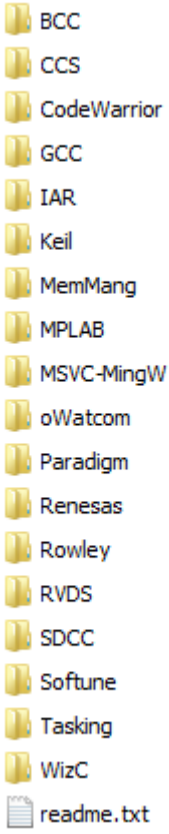
不过小灯现在着重于自己动手移植 FreeRTOS，考虑到原子哥@正点原子的用户比较多，绝大多数习惯了使用 MDK 来开发 STM32，因此小灯分别以 IAR 和 MDK 两种使用比较广泛的开发环境来移植 FreeRTOS。说到 IAR 和 MDK，不得不提的是小灯自从用了 IAR 之后就果断放弃了 MDK，相信很多人有这个经历，哈哈！

在开始移植 FreeRTOS 之前，先介绍下 FreeRTOS 的源码：



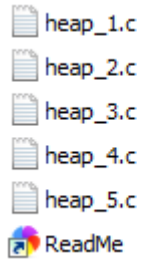
FreeRTOS 的源码比较少，源文件也远没有 UCOS 多，不过麻雀虽小五脏俱全，FreeRTOS 的短小精悍也是最令小灯着迷的，虽然缺少了很多组成部分，例如 GUI、网络协议栈、文件系统等，不过这些统统都不是问题，因为完全可以移植第三方的组件！

一不小心牛逼又吹大了，哈哈！回归正题，FreeRTOS 的源码核心部分是 tasks.c 和 list.c，其余的几个文件功能都是可选的，例如软件定时器、队列、协程等等，小灯就不介绍了，有兴趣的话可以到官网上看介绍。include 文件夹里面的文件是操作系统相关的头文件，而 portable 这个文件夹有些奇葩，先看看里面有啥：



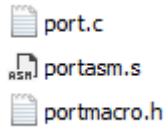
这里的文件几乎都是与平台相关的，如果你要删掉这里的文件时就必须小心了，因为不是所有文件都能删除的。

注意文件夹 MemMang，里面存放的是 FreeRTOS 自带的内存管理方案的源文件：

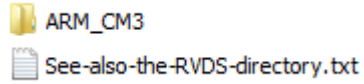


关于内存管理方案的选择，小灯以后再跟大家讨论，现在只需要知道这些文件不能删就好。

接下来看看 IAR 文件夹的内容，里面都是跟单片机底层相关的，由于我们以 STM32F103 为例，因此只需要保留 ARM_CM3 文件夹即可，其余可选择性删除。ARM_CM3 文件夹里只有几个文件，这几个文件是操作系统最最底层的部分：



接下来再看看 Keil 文件夹的内容，里面只有一个文件，文件提示 See-also-the-RVDS-directory，意思是让我们参照 RVDS 目录下的文件。其实我们以 MDK 建工程时，就是拿 RVDS 目录下的文件来替代的，因此我们应该把 RVDS 目录下的文件拷贝到 Keil 目录下，跟上面 IAR 文件夹一样我们只拷贝 ARM_CM3 文件夹即可：

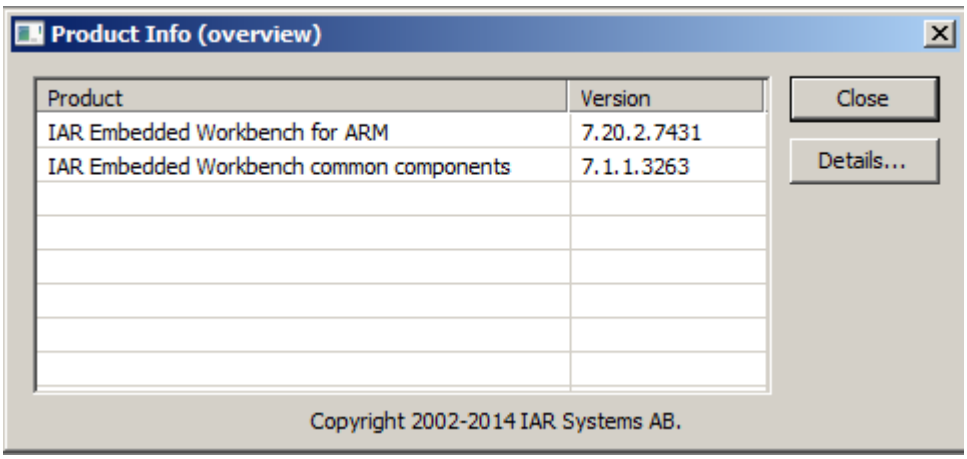


到这里我们可以把其他无用的文件统统删掉了，portable 目录下只保留下面几个文件夹的文件即可：

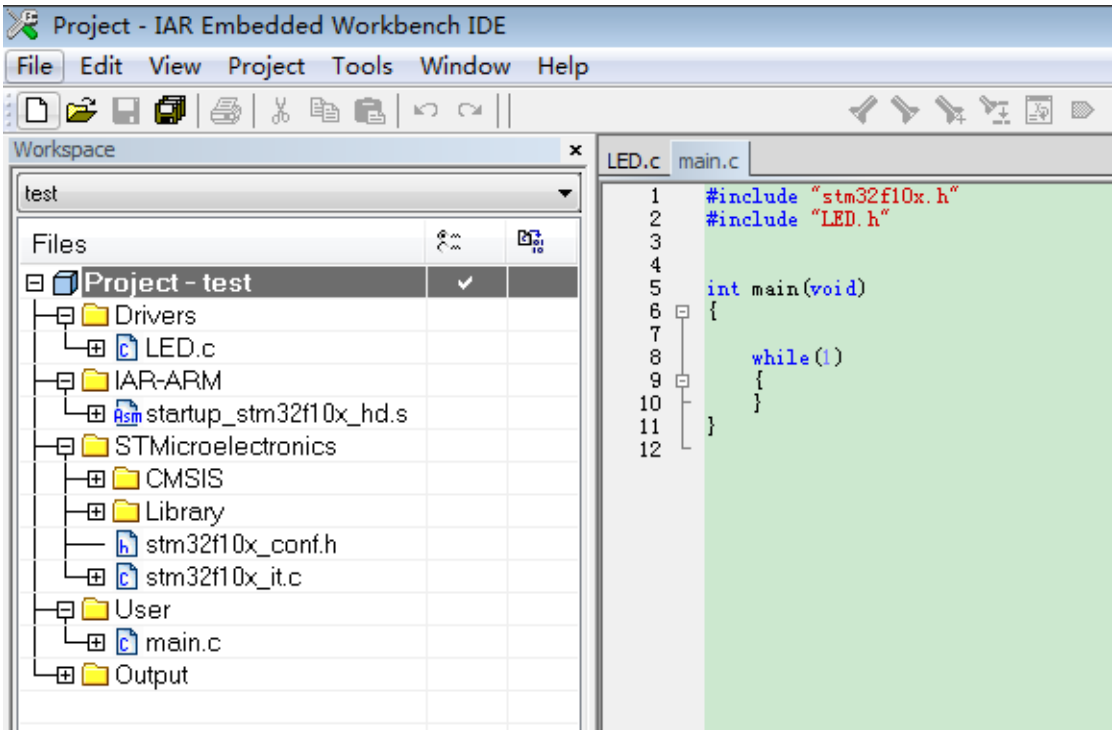
现在已经把源码整理好了，接下来就开始移植工作吧！

二、IAR 下移植 FreeRTOS

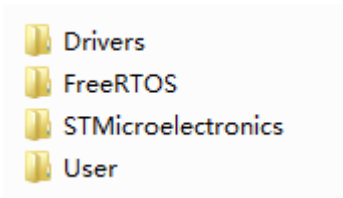
事先说明下，小灯使用的 IAR 版本是



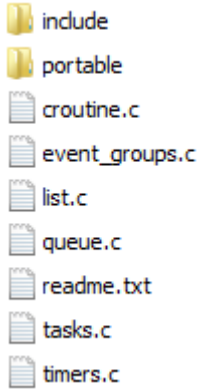
关于 IAR 下如何创建 STM32 基础工程，小灯就偷下懒不介绍了，这入门级的知识还是交给卖开发板的人来传播吧，小灯就以自己平常用的简单工程为例：



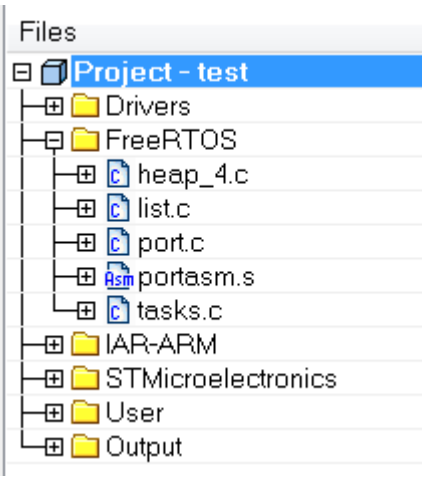
工程当中只有一个 LED.c 是小灯额外添加的，小灯一直停留在跑灯的水平，习惯用 LED 来观察现象，希望各位大神莫怪。工程源码结构如下：



其中 FreeRTOS 文件夹下就是 FreeRTOS 的源码：



接下来在工程里面添加 FreeRTOS 文件：



文件清单如下：

FreeRTOS\tasks.c

FreeRTOS\list.c

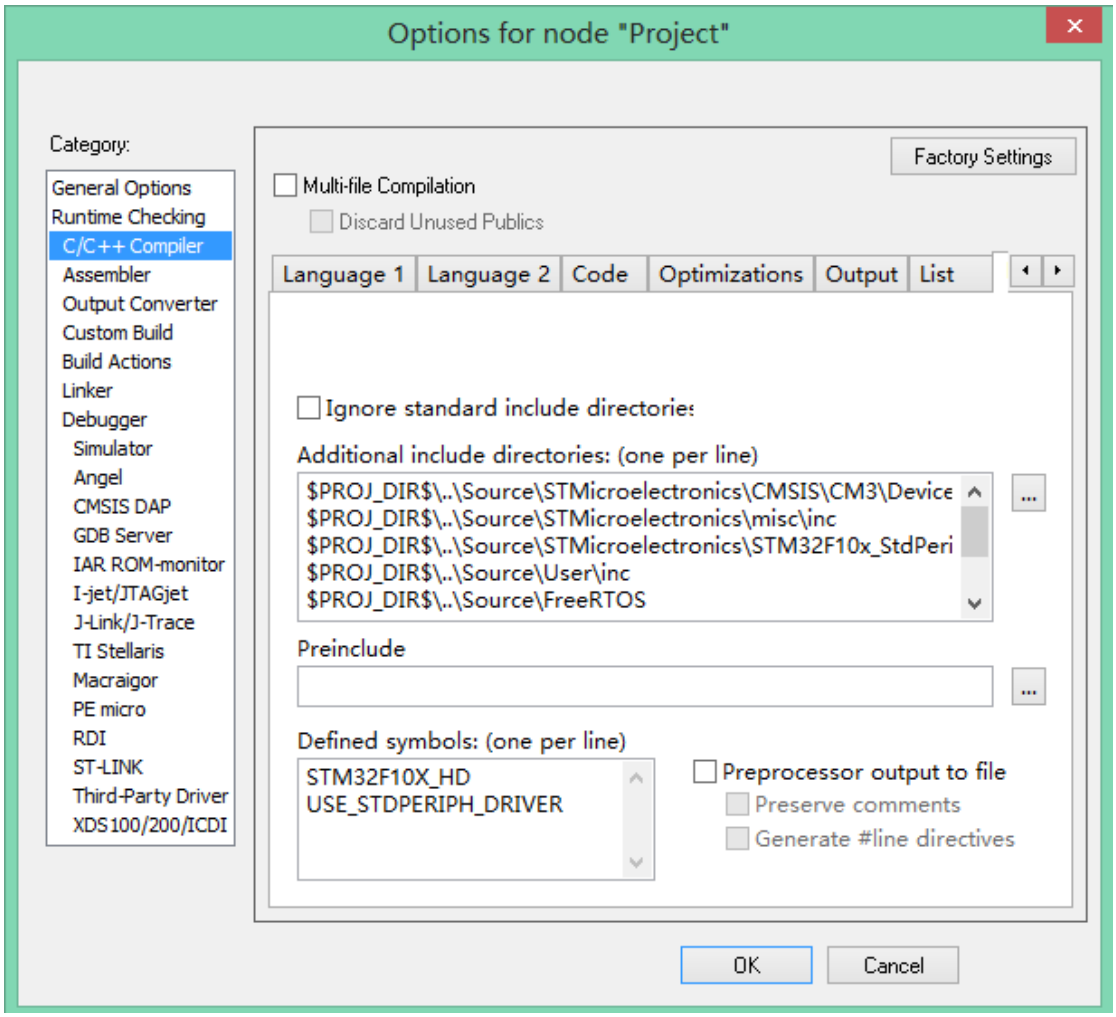
FreeRTOS\portable\IAR\ARM_CM3\port.c

FreeRTOS\portable\IAR\ARM_CM3\portasm.s

FreeRTOS\portable\MemMang\heap_4.c

这时有人可要问为何没有把 FreeRTOS 的所有文件都添加进去，原因我上面提过了，FreeRTOS 的核心部分是 tasks.c 和 list.c，其余的几个文件是可选部分，在此小灯就先不添加这些可选部分以简化我们的工程。小灯建议大家使用内存管理的方案四 heap_4.c，因为该方案具有内存块碎片合并功能，比 heap_2.c 的最优内存块分配方案要稳定很多，这是小灯经过很长时间测试对比出来的，公司的产品也是一直使用 heap_4.c，稳定性无懈可击！

接下来非常重要的一步就是添加头文件路径：



头文件路径如下：

$PROJ_DIR$\..\Source\FreeRTOS\include

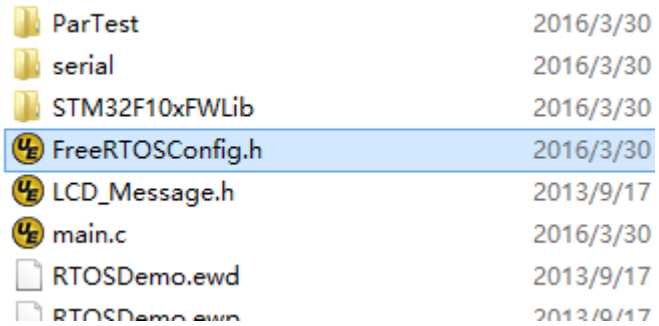
$PROJ_DIR$\..\Source\FreeRTOS\portable\IAR\ARM_CM3

好了这时我们可以尝试编译下整个工程了，编译结果提示缺少了个头文件：

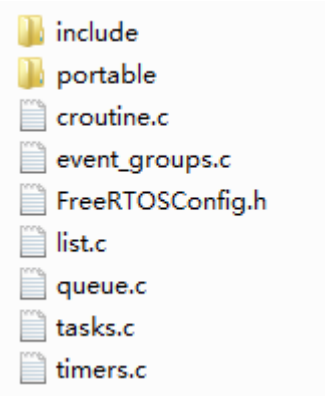
```
Fatal Error[Pe1696]: cannot open source file "FreeRTOSConfig.h"
```

FreeRTOS 组织也真是奇葩，居然连这么重要的文件都不提供在源码里面！！！前面提醒过大家，新建工程时碰到问题一定要参考官方提供的 Demo，既然 Demo 是一堆成品的工程，那么里面绝对有我们所需的这个 FreeRTOSConfig.h

我们就选择打开 Demo\CORTEX_STM32F103_IAR 下的这个工程吧，果不其然里面真的有我们需要的这个头文件：



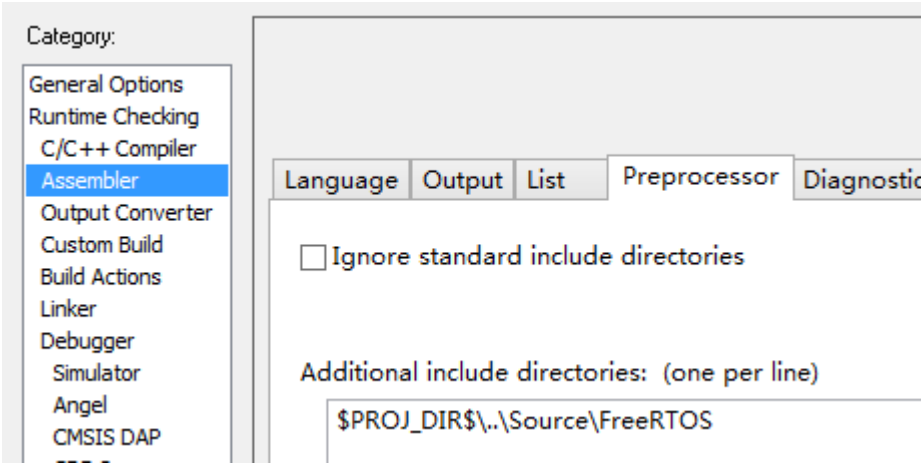
把这文件放哪里好呢，这是一直纠结小灯的问题，官方直接把这文件放在工程目录下，但这么重要的配置文件这么随便放似乎不太好吧。在小灯看来，这个文件的重要性和打开的概率绝不比 FreeRTOS 内核文件低，所以还是把它放在源码里面比较合理：



在 C/C++ Compiler 下添加头文件路径：

$PROJ_DIR$\..\Source\FreeRTOS

还有一个地方一定要十分注意，因为操作系统的最最底层的几个文件也需要用到 FreeRTOSConfig.h 头文件，而底层文件是用汇编来写的，因此**必须在 Assembler 下添加 FreeRTOSConfig.h 头文件路径**：



好了再编译一次：

```
Total number of errors: 0
Total number of warnings: 0
```

0 个错误 0 个警告，程序员最欢喜的莫过于看见这个结果了，哈哈！

在编写系统任务前，有必要对配置文件 FreeRTOSConfig.h 进行检查。FreeRTOSConfig.h 里面几乎都是一些宏定义，关于这些宏定义的具体用法，可以在官网上查阅：http://www.freertos.org/a00110.html

小灯只以其中几个比较重要的参数作特别说明，下面以小灯修改过的 FreeRTOSConfig.h 为例作为说明：

```
#define configUSE_PREEMPTION            1                                       // 使用可剥夺内核(可抢
#define configUSE_IDLE_HOOK             0                                       // 不使用空闲任务钩子函
#define configUSE_TICKLESS_IDLE         0                                       // 使用低功耗机制
#define configEXPECTED_IDLE_TIME_BEFORE_SLEEP   3                               // 最小唤醒节拍设为5个
#define configUSE_TICK_HOOK             0
#define configCPU_CLOCK_HZ              ( ( unsigned long ) 72000000 )          // CPU主频为72MHz
#define configTICK_RATE_HZ              ( ( TickType_t ) 1000 )                 // SysTick节拍频率
#define configMAX_PRIORITIES            ( 5 )                                   // 最大任务优先级的数目
#define configMINIMAL_STACK_SIZE        ( ( unsigned short ) 40 )               // 最小任务堆栈分配
#define configTOTAL_HEAP_SIZE           ( ( size_t ) ( 10 * 1024 ) )             // 内存堆管理的内存空
#define configMAX_TASK_NAME_LEN         ( 8 )                                   // 任务名称最大长度
#define configUSE_TRACE_FACILITY        0
#define configUSE_16_BIT_TICKS          0                                       // 不使用16位节拍数，使
#define configIDLE_SHOULD_YIELD         0                                       // 空闲任务不主动让出CP
#define configUSE_MUTEXES               1                                       // 使用互斥信号量
#define configQUEUE_REGISTRY_SIZE       8
#define configCHECK_FOR_STACK_OVERFLOW  0                                       // 不使用堆栈溢出检测
#define configUSE_RECURSIVE_MUTEXES     0
#define configUSE_MALLOC_FAILED_HOOK    0
#define configUSE_APPLICATION_TASK_TAG  0
#define configUSE_COUNTING_SEMAPHORES   0                                       // 不使用计数信号量
#define configGENERATE_RUN_TIME_STATS   0


/* Co-routine设置 */
#define configUSE_CO_ROUTINES           0           // 不使用协程
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )       // 协程优先级


/* 软件定时器使用设置 */
#define configUSE_TIMERS                0           // 不使用软件定时器
#define configTIMER_TASK_PRIORITY       ( 2 )
#define configTIMER_QUEUE_LENGTH        5
#define configTIMER_TASK_STACK_DEPTH    ( 50 )


/* 任务相关API函数包含设置 */
#define INCLUDE_vTaskPrioritySet        0           // 是否允许重设任务优先级
#define INCLUDE_uxTaskPriorityGet       0           // 是否允许获取任务优先级
#define INCLUDE_vTaskDelete             0           // 是否允许任务删除
#define INCLUDE_vTaskCleanUpResources   0
#define INCLUDE_vTaskSuspend            1           // 是否允许调度器挂起
#define INCLUDE_vTaskDelayUntil         1
#define INCLUDE_vTaskDelay              1
```

（1） 定义系统底层相关的函数

```
/* 定义操作系统中断为STM32底层中断 */
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler
```

其中 SVC 中断时操作系统启动时进入的中断，而 PendSV 中断手动切换任务时进入的中断，SysTick 中断不用我多说了，@正点原子的基础例程几乎都使用这个定时器定时，在这里 SysTick 是作为操作系统的心脏。由于 FreeRTOS 对这几个中断的名称做了自己的定义，因此必须要重定义这几个函数才能正常进入中断，但这么做又会跟 ST 提供的 stm32f10x_it.c 文件当中定义的中断相冲突，因此必须将 stm32f10x_it.c 下对应的几个中断服务函数屏蔽掉，否则编译会提示函数重定义：

```
////**
////  * @brief  This function handles SVCall exception.
////  * @param  None
////  * @retval None
////  */
////void SVC_Handler(void)
////{
////}

/**
  * @brief  This function handles Debug Monitor exception.
  * @param  None
  * @retval None
  */
void DebugMon_Handler(void)
{
}

////**
////  * @brief  This function handles PendSVC exception.
////  * @param  None
////  * @retval None
////  */
////void PendSV_Handler(void)
////{
////}
////
////**
////  * @brief  This function handles SysTick Handler.
////  * @param  None
////  * @retval None
////  */
////void SysTick_Handler(void)
////{
////}
```

（2） 修改系统可屏蔽的中断优先级阈值

FreeRTOS 提供的可屏蔽中断优先级阈值是 191，对应的十六进制数是 0xBF：

#define configMAX_SYSCALL_INTERRUPT_PRIORITY　　191

由于 STM32F103 的优先级分组只有 4 个位，而 CM3 的优先级是以 MSB 对齐的，也就是说 STM32F103 的优先级寄存器只有最高 4 位有效，低四位是无效的。当操作系统进入临界区时，会把上面的可屏蔽中断优先级阈值写入 BASEPRI 寄存器以屏蔽部分中断：

```
#define portDISABLE_INTERRUPTS()                                \
{                                                               \
    __set_BASEPRI( configMAX_SYSCALL_INTERRUPT_PRIORITY );      \
    __DSB();                                                    \
    __ISB();                                                    \
}
```

因此当进入临界区时，优先级对应 0xB~0xF 的中断均被屏蔽，而优先级处于 0xB 前面的中断不受影响。这个跟 CM0 有区别，也是最值得注意的地方。

上述的基础知识不是小灯要重点提的，对 CM3 的优先级不熟悉的朋友建议查阅《Cortex-M3 权威指南》，接下来才是小灯要重点提的。由于使用@正点原子的开发板的用户比较多，很多人直接把 FreeRTOS 移植到原子哥的工程下，然后出现了各种各样的诡异问题，一直无解。其中一个非常严重的问题就是小灯上面提及到的中断屏蔽的问题，下面小灯就这个问题进行一个简单的分析，先贴上@正点原子的部分代码：

```
void NVIC_Configuration(void)
{

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置NVIC中断分组2:2位抢占优先级，2位响应优先级

}
```

这是原子哥最常用的中断优先级分组方式，采用分组方式 2，2 位抢占优先级 2 位亚优先级。但是在移植 FreeRTOS 时必须要修改成优先级分组方式 4：

```
#define NVIC_PriorityGroup_4         ((uint32_t)0x300) /*!< 4 bits for pre-emption priority
                                                            0 bits for subpriority */
```

把 STM32 的优先级分组的 4 个位均设成抢占优先级，也就是说完全放弃亚优先级。为何要这么设置？其实这得怪 FreeRTOS 机构里面被驴踢过的逗逼，这些逗逼为了自己省事，直接默认不使用亚优先级，下面让大家见识下这群逗逼的解释：

```
/* Priority grouping:  The interrupt controller (NVIC) allows the bits
that define each interrupt's priority to be split between bits that
define the interrupt's pre-emption priority bits and bits that define
the interrupt's sub-priority.  For simplicity all bits must be defined
to be pre-emption priority bits.  The following assertion will fail if
this is not the case (if some bits represent a sub-priority).

If the application only uses CMSIS libraries for interrupt
configuration then the correct setting can be achieved on all Cortex-M
devices by calling NVIC_SetPriorityGrouping( 0 ); before starting the
scheduler.  Note however that some vendor specific peripheral libraries
assume a non-zero priority group setting, in which cases using a value
of zero will result in unpredicable behaviour. */
configASSERT( ( portAIRCR_REG & portPRIORITY_GROUP_MASK ) <= ulMaxPRIGROUPValue );
```

不过站在他们的角度来思考，其实我们也应该给他们那么一丁点儿理解，毕竟他们采取了一个比较简单的方法来获取不同厂商的单片机的有效优先级位数，算法是通过对优先级寄存器组的某一个寄存器写入 0xFF，然后再读出来看实际上有多少位写入成功，然后根据实际的有效位数来重设优先级分组寄存器的分组方式（上面提到的分组方式 4）。有兴趣的可以研究下他们的算法，代码截图在下面：

```
/* Determine the maximum priority from which ISR safe FreeRTOS API
functions can be called.  ISR safe functions are those that end in
"FromISR".  FreeRTOS maintains separate thread and ISR API functions to
ensure interrupt entry is as fast and simple as possible.

Save the interrupt priority value that is about to be clobbered. */
ulOriginalPriority = *pucFirstUserPriorityRegister;

/* Determine the number of priority bits available.  First write to all
possible bits. */
*pucFirstUserPriorityRegister = portMAX_8_BIT_VALUE;

/* Read the value back to see how many bits stuck. */
ucMaxPriorityValue = *pucFirstUserPriorityRegister;

/* Use the same mask on the maximum system call priority. */
ucMaxSysCallPriority = configMAX_SYSCALL_INTERRUPT_PRIORITY & ucMaxPriorityValue;

/* Calculate the maximum acceptable priority group value for the number
of bits read back. */
ulMaxPRIGROUPValue = portMAX_PRIGROUP_BITS;
while( ( ucMaxPriorityValue & portTOP_BIT_OF_BYTE ) == portTOP_BIT_OF_BYTE )
{
    ulMaxPRIGROUPValue--;
    ucMaxPriorityValue <<= ( uint8_t ) 0x01;
}

/* Shift the priority group value back to its position within the AIRCR
register. */
ulMaxPRIGROUPValue <<= portPRIGROUP_SHIFT;
ulMaxPRIGROUPValue &= portPRIORITY_GROUP_MASK;

/* Restore the clobbered interrupt priority register to its original
value. */
*pucFirstUserPriorityRegister = ulOriginalPriority;
```

不知不觉越扯越远了，回归正题，到底为何要重设可屏蔽的中断优先级阈值，我们重新把思路理一下。根据 STM32 的中断优先级的设计，只有高 4 位有效，还有 FreeRTOS 默认将 4 个优先级位均划分为抢占优先级。由于 FreeRTOS 官方提供的中断优先级阈值是 191（对应实际的 0xB），也就是 11~15 的优先级均可被操作系统屏蔽。但我们实际使用时设置的中断优先级一般不会使用到 11 打后的，例如@正点原子的基础例程里面使用最多的 1~3，所以我们必须要修改这个值，否则我们要重新修改所有底层驱动的优先级。

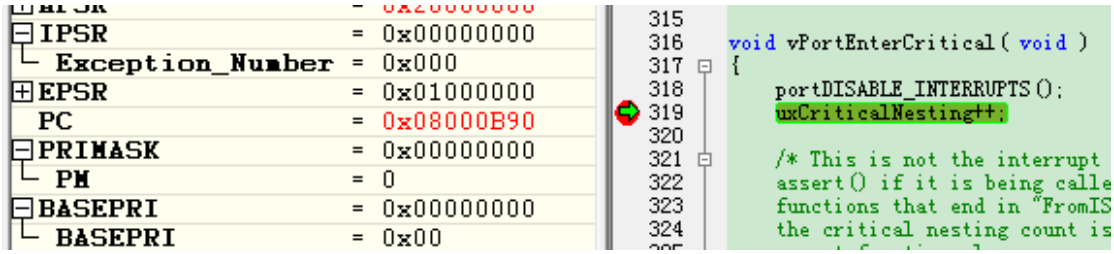
那么怎么修改比较合理？这个就得看实际应用需要了，其实使用宏 configMAX_SYSCALL_INTERRUPT_PRIORITY 来屏蔽部分中断是比较合理的，相对于 CM3，CM0 没有中断优先级阈值寄存器，只能简单粗暴的开启全局中断和关闭全局中断。操作系统在执行十分重要的工作时一般不能打断这个工作，尤其是这时在中断里面调用了操作系统的 API 函数，这都会严重影响系统的稳定性。小灯对 configMAX_SYSCALL_INTERRUPT_PRIORITY 的理解是，这个数值打后的所有中断均划入操作系统管理，而这个数值打前的中断则归由用户自己管理，但用户必须十分小心地处理这些中断，用户可以使用这些中断来处理一些跟操作系统无关的工作。这纯属个人理解，如有错误之处还请大家指出，小灯会尽快修改！

分析完 configMAX_SYSCALL_INTERRUPT_PRIORITY 的作用后，下面小灯提供一个参考值：

```
/* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY    0x1F
```

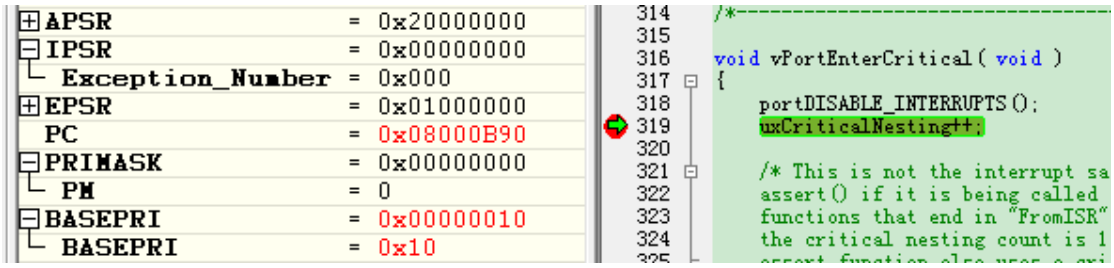
由于优先级寄存器是高四位有效，因此上述的屏蔽阈值实际上是 0x1，也就是说优先级在 1~15 之间的中断均可被操作系统屏蔽，而优先级 0 归由用户自己控制。值得注意的是 configMAX_SYSCALL_INTERRUPT_PRIORITY 的高四位绝对不能设为 0，下面小灯给出 0x0F 的仿真结果：

```
#define configMAX_SYSCALL_INTERRUPT_PRIORITY    0x0F
```



当 configMAX_SYSCALL_INTERRUPT_PRIORITY 设为 0x0F 时系统在进入临界区时 BASEPRI 寄存器的值一直为 0，没有更新。下面给出 0x1F 时的仿真结果：

```
#define configMAX_SYSCALL_INTERRUPT_PRIORITY    0x1F
```



当 configMAX_SYSCALL_INTERRUPT_PRIORITY 设为 0x1F 时系统在进入临界区时 BASEPRI 寄存器的值更新为 0x10，中断优先级阈值寄存器起作用了！原因在上面也解释过了，因为 STM32F103 的优先级寄存器是高四位有效的，对应的 BASEPRI 也是高四位能够写入而低四位无法写入，而 BASEPRI 有一个特点——对它写 0 会取消屏蔽所有中断（相当于禁用了该寄存器），因此 BASEPRI 的高四位一定不能设为 0，否则不会屏蔽任何中断，这点注意下就好了。

（3） 添加参数检测功能

```
/* 函数参数判断 */
#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ); }
```

该参数检测功能是官方提供的一个参考，很多人为了省事不使用参数检测，但小灯劝诫各位千万不能干这蠢事！说了半天想必大家也累了，那么接下来就开始我们跑灯之旅吧！

下面就以 LED 创建两个闪烁任务：

```
#include "stm32f10x.h"
#include "LED.h"

/* FreeRTOS头文件 */
#include "FreeRTOS.h"
#include "task.h"



/**************************************************/
/*                                                */
/*            LED周期性闪烁任务1                  */
/*                                                */
/**************************************************/
void LedTask1(void)
{
    while(1)
    {
        Led_Toggle(LED1);
        vTaskDelay(500);     // 睡眠500ms
    }
}

/**************************************************/
/*                                                */
/*            LED周期性闪烁任务2                  */
/*                                                */
/**************************************************/
void LedTask2(void)
{
    while(1)
    {
        Led_Toggle(LED2);
        vTaskDelay(1000);    // 睡眠1000ms
    }
}
```

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
    GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable , ENABLE);

    LED_Init();

    /* 创建LED1任务 */
    xTaskCreate( LedTask1,       // 任务函数指针
                 "LED_TASK1",    // 任务名称
                 40,             // 堆栈深度(字)
                 NULL,           // 任务参数为空
                 1,              // 任务优先级
                 NULL );         // 任务句柄为空

    /* 创建LED2任务 */
    xTaskCreate( LedTask2, "LED_TASK2", 40, NULL, 2, NULL );

    /* 启动任务调度器(操作系统开始运行) */
    vTaskStartScheduler();

    /* 不应该运行到这里 */
    while(1)
    {
    }
}
```

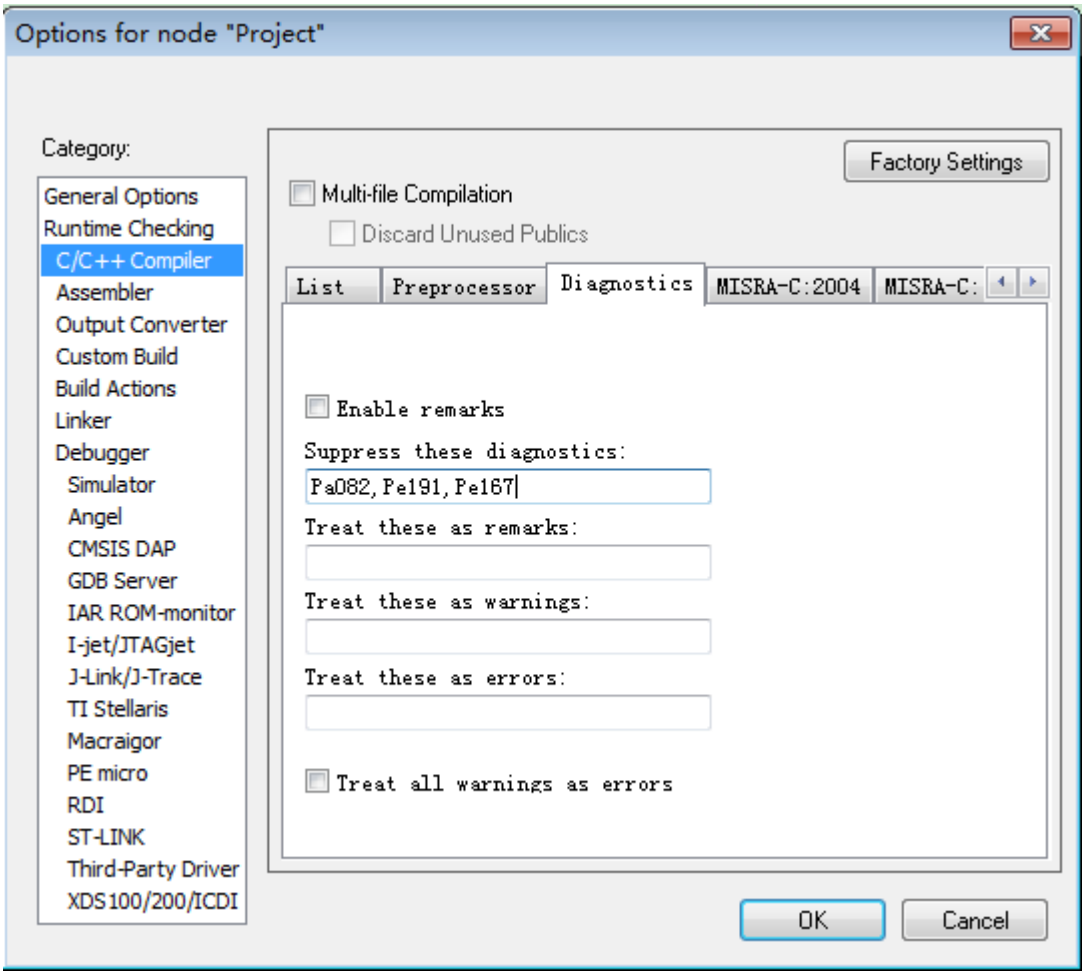
至于 LED 底层驱动代码在工程里面提供了，在这不详细解释，读者可自行修改。在 main 函数里首先设置中断优先级分组方式 4，然后创建 2 个 LED 任务，堆栈深度 40 个字（160 字节）即可，然后启动任务调度器，整个操作系统就能够跑起来了。

重新编译下工程，这时会提示有错误：

```
47      /* 创建LED1任务 */
48      xTaskCreate( LedTask1,       // 任务函数指针
49                   "LED_TASK1",    // 任务名称
50                   40,             // 堆栈深度(字)
51                   NULL,           // 任务参数为空
52                   1,              // 任务优先级
53                   NULL );         // 任务句柄为空
54
55      /* 创建LED2任务 */
56      xTaskCreate( LedTask2, "LED_TASK2", 40, NULL, 2, NULL );
```

```
Error[Pe167]: argument of type "void (*)(void)" is incompatible with parameter of type "TaskFunction_t"
Error[Pe167]: argument of type "void (*)(void)" is incompatible with parameter of type "TaskFunction_t"
Error while running C/C++ Compiler
```

接下来我们需要屏蔽这些编译错误：



在 C/C++ Compiler 的 Diagostics 里面添加 Pa082,Pe191,Pe167

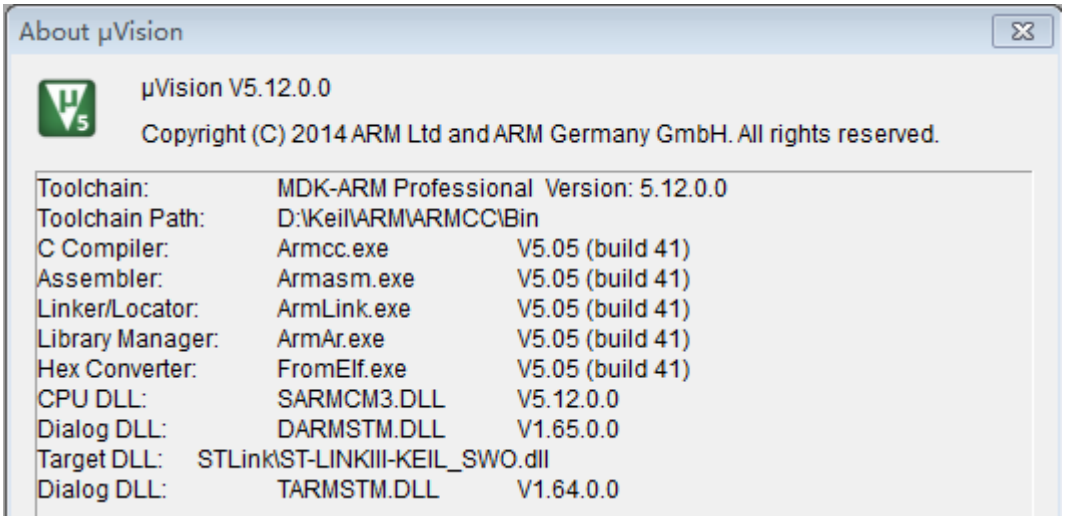
重新编译一次：

```
Total number of errors: 0
Total number of warnings: 0
```

编译通过，下载到板子上后就能够看到 2 个 LED 按照一定的频率闪烁了！

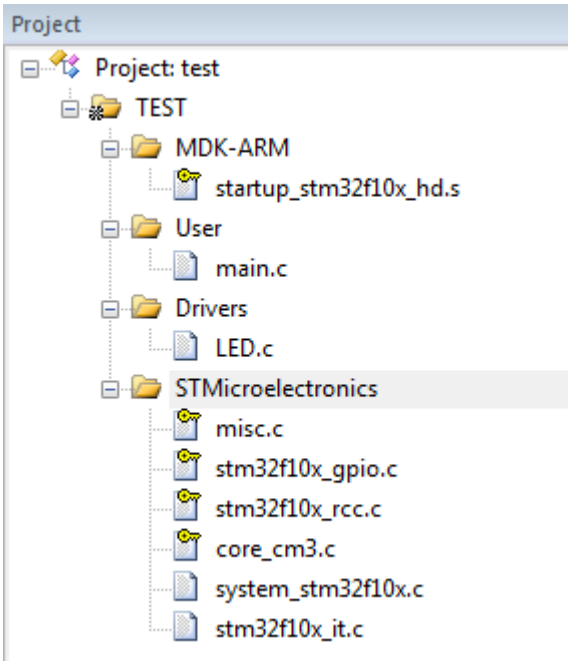
三、MDK 下移植 FreeRTOS

由于在上面 IAR 移植过程中已经把需要注意的事项详细，再次就不累述了，只给出移植过程。小灯使用的 MDK 版本是：

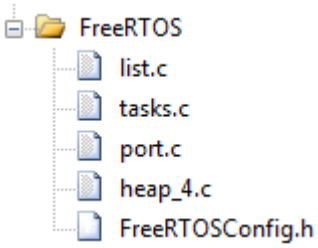


至于 MDK 工程的新建，请参照@正点原子的教程资料，小灯就不抢原子哥风头了，哈哈！

工程沿袭了小灯一贯的简约作风：



（1） 添加 FreeRTOS 源文件：



源文件清单路径如下：

FreeRTOS\tasks.c

FreeRTOS\list.c

FreeRTOS\portable\Keil\ARM_CM3\port.c

FreeRTOS\portable\MemMang\heap_4.c

注意这里的 FreeRTOSConfig.h 的来源小灯在上面 IAR 移植里面已经详细说明了，而且对 FreeRTOSConfig.h 里面的内容做了大篇幅的分析，读者请自行翻查，在此不再累述。

（2） 添加头文件路径：

```
..\Source\FreeRTOS
..\Source\FreeRTOS\include
..\Source\FreeRTOS\portable\Keil\ARM_CM3
```

（3） 修改 stm32f10x_it.c 文件：

```
/////**
////  * @brief  This function handles SVCall exception.
////  * @param  None
////  * @retval None
////  */
////void SVC_Handler(void)
////{
////}

/**
  * @brief  This function handles Debug Monitor exception.
  * @param  None
  * @retval None
  */
void DebugMon_Handler(void)
{
}

/////**
////  * @brief  This function handles PendSVC exception.
////  * @param  None
////  * @retval None
////  */
////void PendSV_Handler(void)
////{
////}
////
/////**
////  * @brief  This function handles SysTick Handler.
////  * @param  None
////  * @retval None
////  */
////void SysTick_Handler(void)
////{
////}
```

至于为了要修改该文件，小灯在上面 IAR 移植里面已经详细说明了，读者请自行翻查，在此不再累述。

（4） 创建两个 LED 闪烁任务：

```
#include "stm32f10x.h"
#include "LED.h"

/* FreeRTOS头文件 */
#include "FreeRTOS.h"
#include "task.h"



/**********************************************/
/*                                            */
/*            LED周期性闪烁任务1              */
/*                                            */
/**********************************************/
void LedTask1(void)
{
    while(1)
    {
        Led_Toggle(LED1);
        vTaskDelay(500);     // 睡眠500ms
    }
}

/**********************************************/
/*                                            */
/*            LED周期性闪烁任务2              */
/*                                            */
/**********************************************/
void LedTask2(void)
{
    while(1)
    {
        Led_Toggle(LED2);
        vTaskDelay(1000);    // 睡眠1000ms
    }
}
```

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
    GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable , ENABLE);

    LED_Init();

    /* 创建LED1任务 */
    xTaskCreate( LedTask1,       // 任务函数指针
                 "LED_TASK1",    // 任务名称
                 40,             // 堆栈深度(字)
                 NULL,           // 任务参数为空
                 1,              // 任务优先级
                 NULL );         // 任务句柄为空

    /* 创建LED2任务 */
    xTaskCreate( LedTask2, "LED_TASK2", 40, NULL, 2, NULL );

    /* 启动任务调度器(操作系统开始运行) */
    vTaskStartScheduler();

    /* 不应该运行到这里 */
    while(1)
    {
    }
}
```

代码的解释在 IAR 移植里面已经详细说明了，读者请自行翻查，在此不再累述。

（5） 编译工程：

```
Build target 'TEST'
".\obj\test.axf" - 0 Error(s), 0 Warning(s).
```

在 MDK 移植 FreeRTOS 相对来说顺利很多，因为不会出现一大堆警告和错误。接下来下载到板子，效果跟 IAR 移植后一样，2 个 LED 按照预定的频率闪烁！

总结：

FreeRTOS 的移植难度并不大，只是有些地方需要注意，否则出问题了也完全找不出问题出在哪里。由于小灯习惯了使用 IAR，习惯使用 MDK 的用户如果看 IAR 的移植步骤可能会很不习惯，但两者移植过程相仿，读者只需要注意对应的事项就好。鉴于小灯水平有限，文档难免会有出错的地方，小灯也很希望各位能指出错误之处或提供宝贵意见，小灯会及时修改，不胜感激！

FreeRTOS 的官方网址为：http://www.freertos.org/