写在前面的话

本系列笔记一共七篇，是我个人学习 FreeRTOS 的实验笔记。

学习过程中写笔记有几个好处：一是可以加深自己对 FreeRTOS 的理解；二是使学习更有成就感。笔记可以作为自己学习进步和知识储备的凭证，当然没人去查，关键是自己真的从中有更多的收获。

在开始学习 FreeRTOS 时，我就已经计划整理出笔记并上传到网上，希望对初学者有所帮助。因为我的学习历程也非常依赖网络资源。

本人在学习 FreeRTOS 之前，已经学过μ C/OS II，也上传了几篇学习笔记。这两个系统非常相似，都是开源的 RTOS，但是一个是免费的，另一个是收费的。

笔记的主要内容就是学习 FreeRTOS 的各种通讯机制。

笔记的结构非常简单，就是通过简单的实例，演示 FreeRTOS 的各种通讯机制的使用方法。

跟随本笔记学习完，能够做到以下几点即可：

1.了解 FreeRTOS 程序的基本架构；

2.能够理解和应用信号量、消息队列、邮箱队列等相关知识。

特别说明：

本笔记以 STM32 为平台，任何 STM32 平台都可以。所有例程只用到简单的硬件资源：最小系统的资源，LED 输出，UART 输出。

为了开发简单，本笔记的例程全部使用 STM32Cube 配置生成，只需要添加很少的代码。如果不熟悉 STM32Cube 的使用，也没关系。只要在网上下载安装 STM32CubeMX 和相应芯片的支持包，然后跟着笔记的步骤操作即可，该笔记没有省略任何步骤。

要学习 STM32Cube，可到 ST 社区论坛 http://www.stmcu.org/module/forum/forum.php，搜索 STM32Cube，即可查看相关帖子。其中比较详细和全面的是微雪电子发布的帖子。

重要参考资料：

FreeRTOS 实时内核实用指南.pdf (由 Zou Changjun 翻译并分享)，建议学习者先通读一遍该文档，这是翻译自 FreeRTOS 作者 Richard Barry 于 2009 年发布的手册。

最新最详细的资料当然是官网 www.freertos.org 发布的信息。

由于本人水平有限，错漏难免，欢迎指正，谢谢！

S.D.Lu 于 深圳

2016 年 8 月

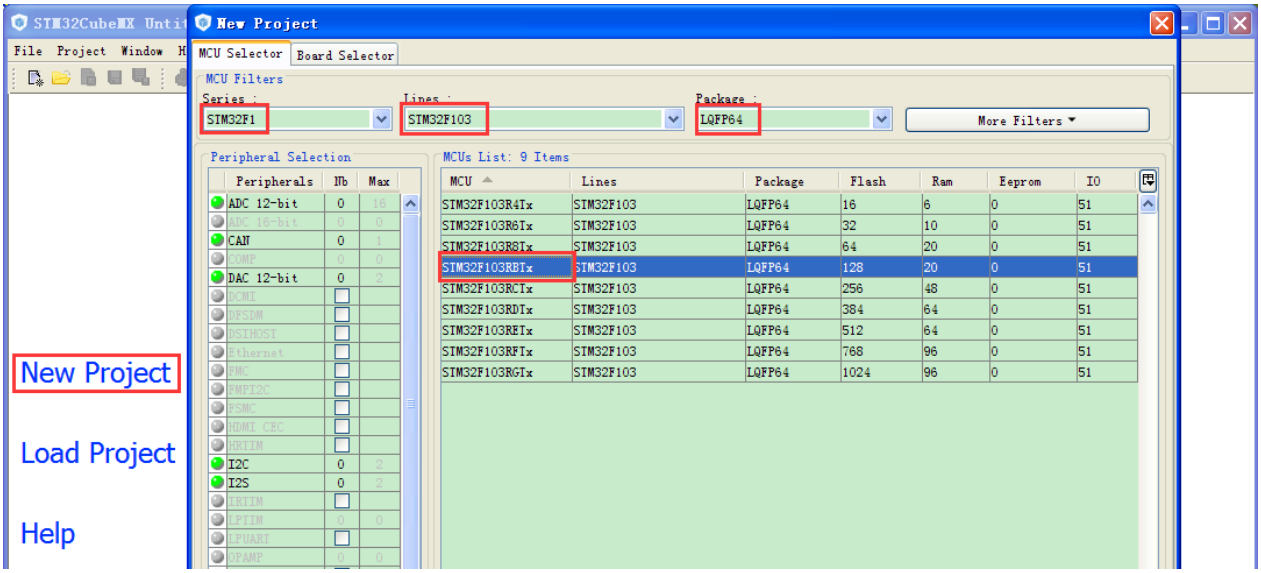
E-mail:547068172@qq.com

FreeRTOS 学习之一：任务的创建

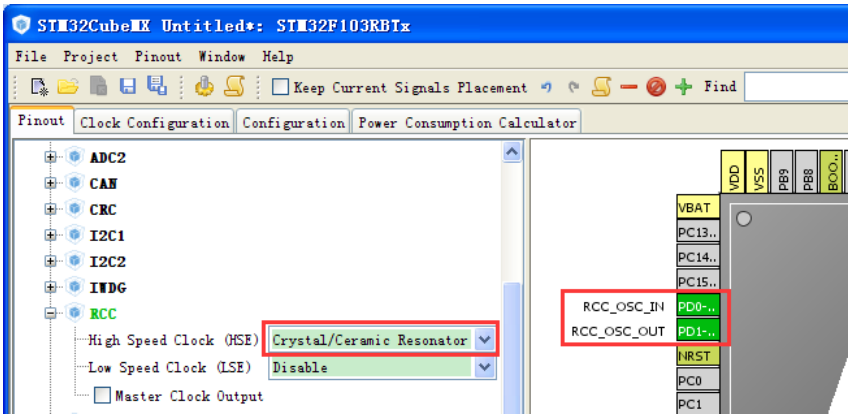
前提：默认已经装好 MDK V5 和 STM32CubeMX，并安装了 STM32F1xx 系列的支持包。
硬件平台：STM32F1xx 系列。
目的：学习 FreeRTOS 任务的创建。

创建任务是使用 FreeRTOS 的必要步骤，本文通过实例描述怎样使用 STM32CubeMX 配置创建 FreeRTOS 的任务。本文例子将创建两个任务，每个任务分别控制一个 LED 的闪烁。

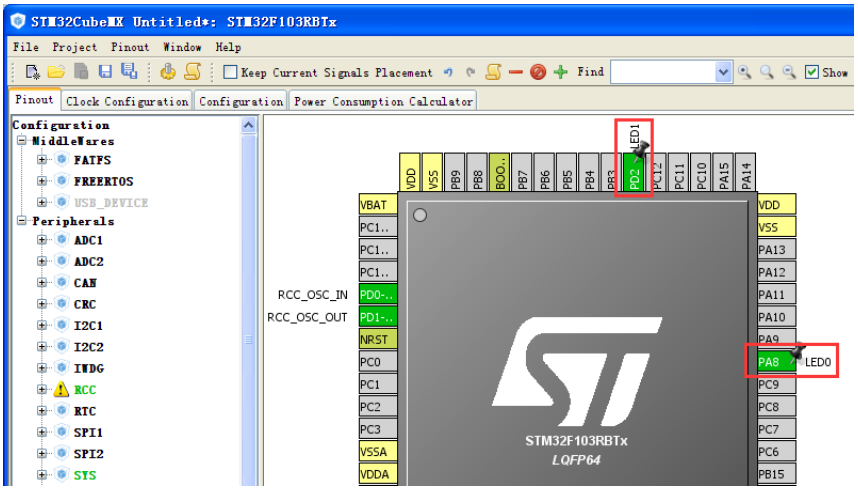
Step1.打开 STM32CubeMX，点击“New Project”，选择芯片型号，STM32F103RBTx。



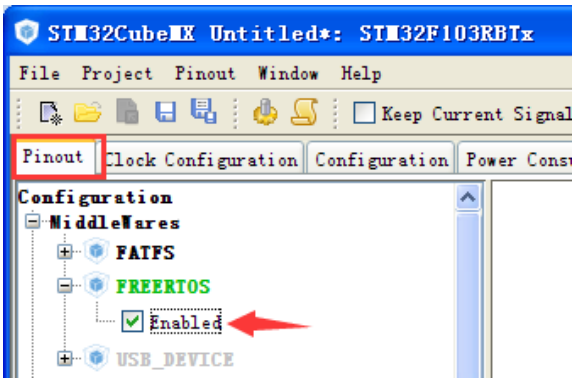
Step2.配置时钟引脚。



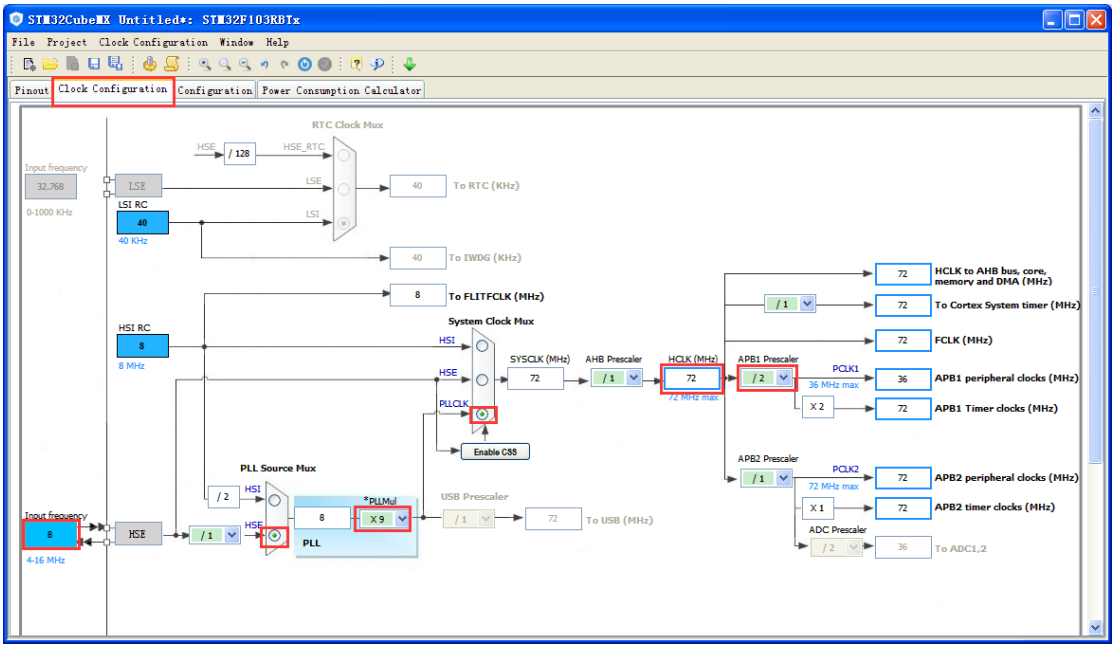
Step3.配置 PA8 和 PD2 为 Output，并把用户标签分别改为 LED0，LED1。



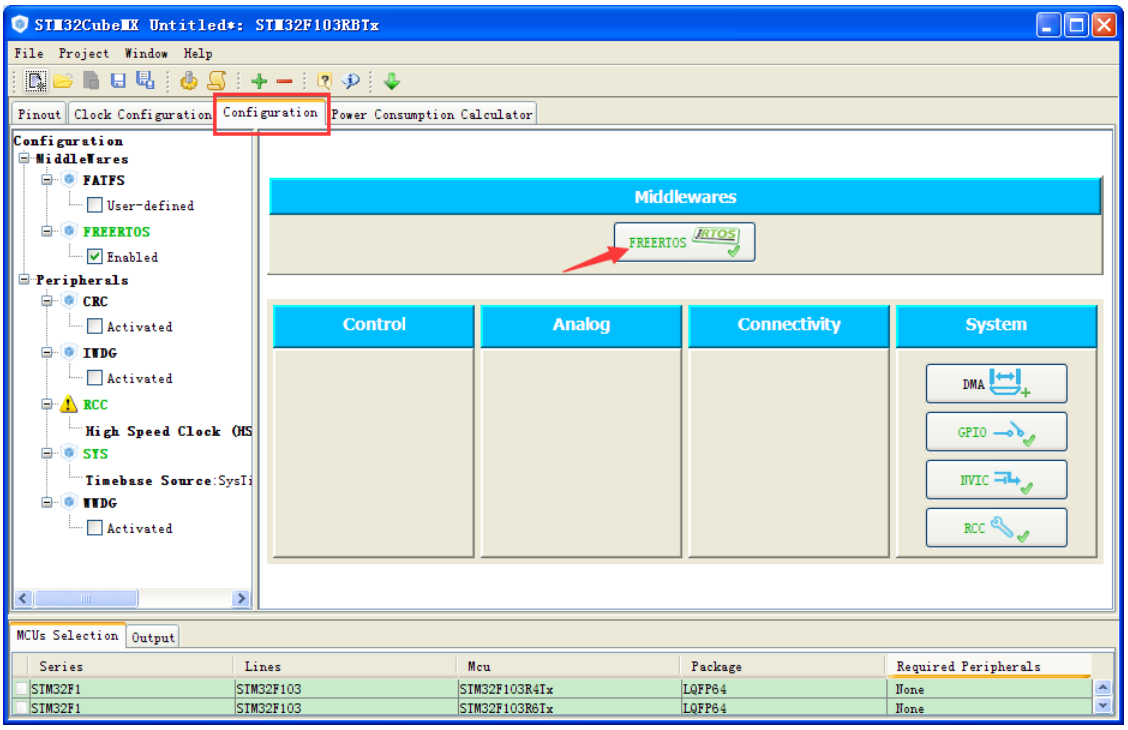
Step4.使能 FreeRTOS。



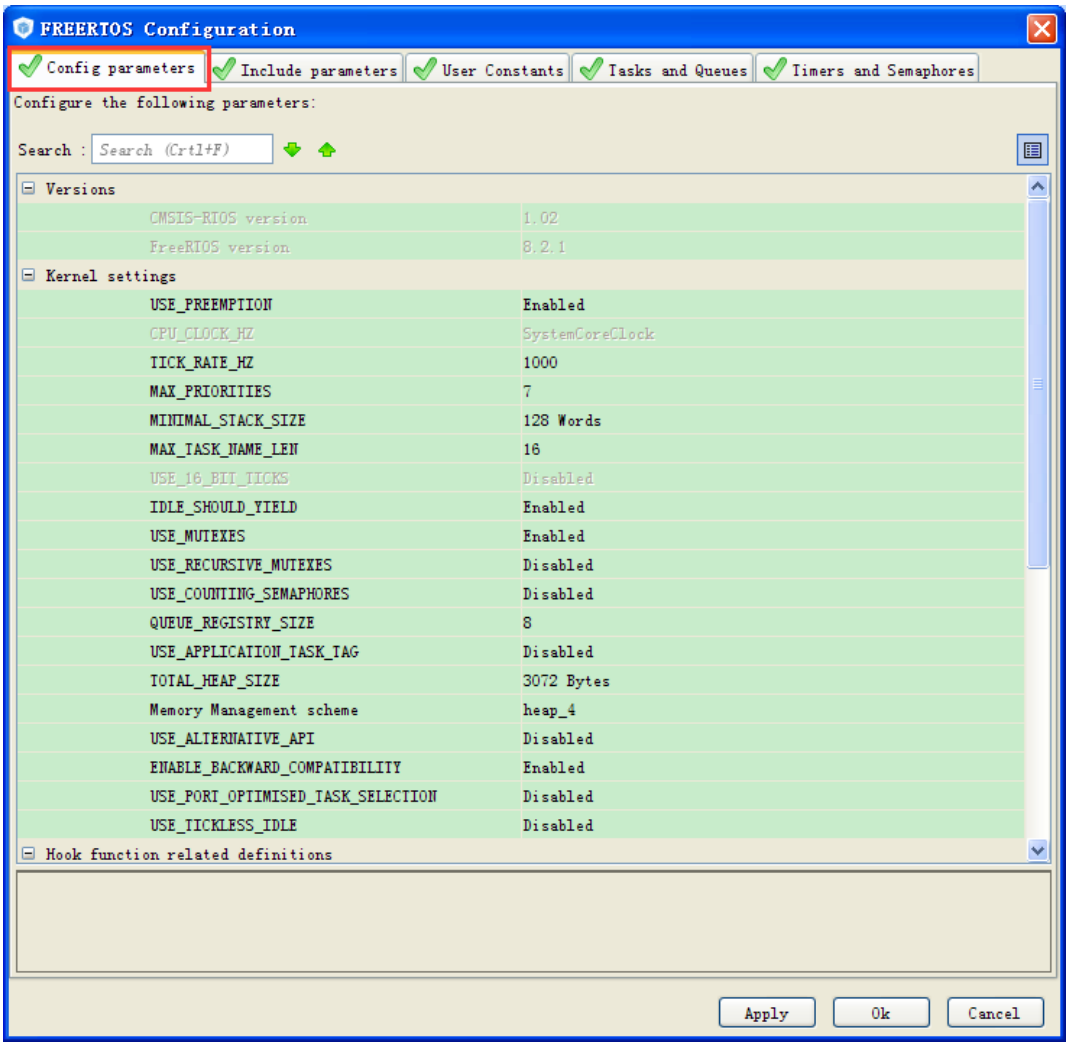
Step5.配置时钟树。8M 输入时，通过 PLL 得到 72M 内部时钟。



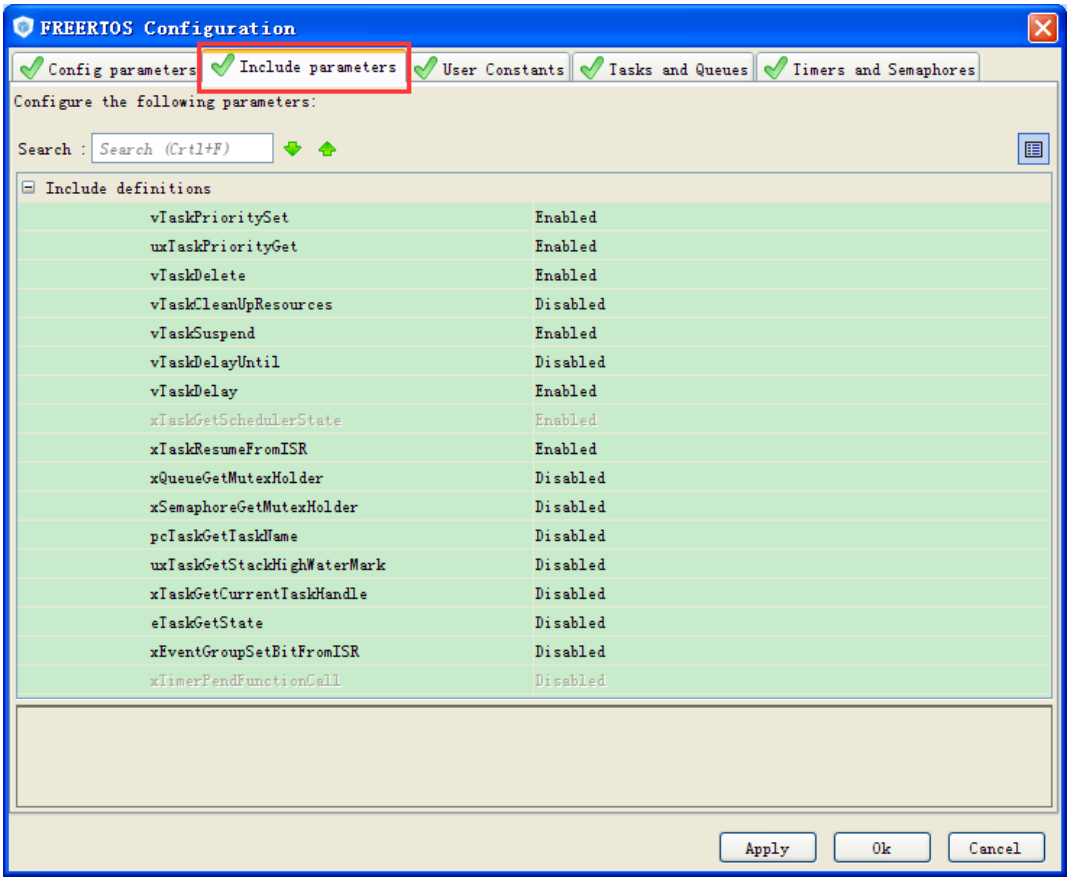
Step6.配置 FreeRTOS。



Config parameters 选项卡中是配置参数，其中列出了 FreeRTOS 的可配置参数，对应于 FreeRTOSConfig.h 文件中的配置参数。

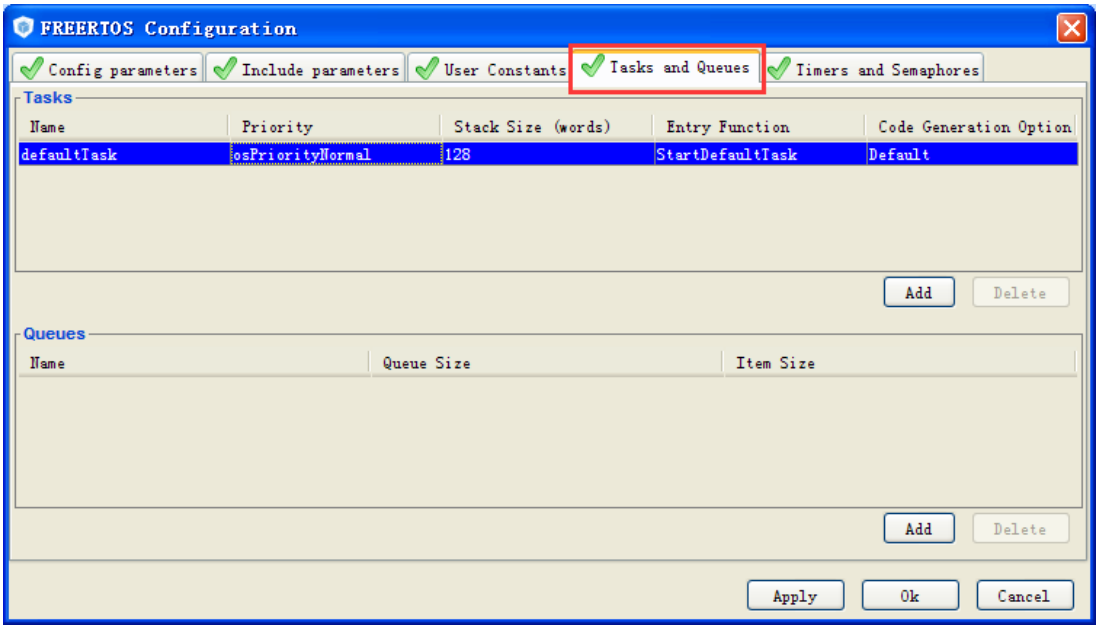


Include parameters 选项卡的参数则是用来配置裁剪 FreeRTOS 的。

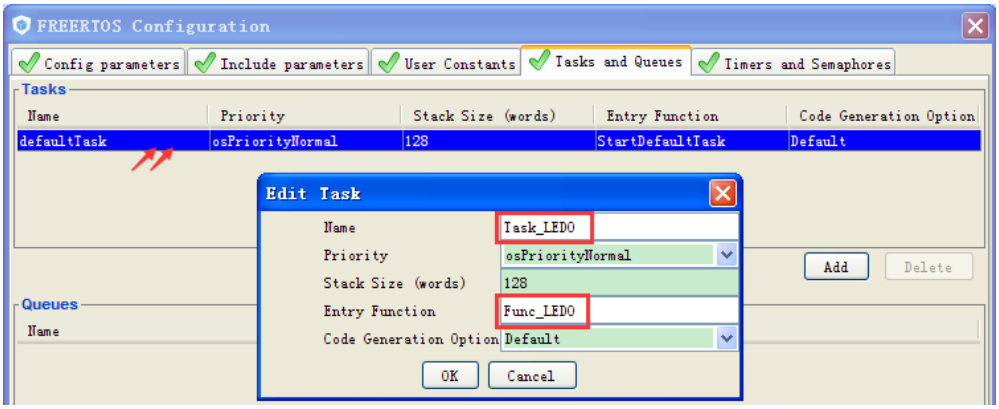


Tasks and Queues 用于添加任务和队列。

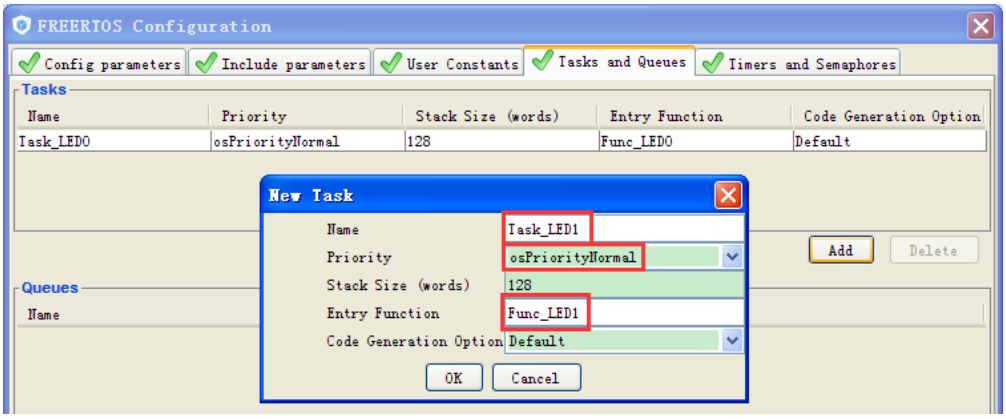
默认配置了一个名为 defaultTask 的任务，其优先级为普通，任务堆栈大小为 128 字，任务函数名为 StartDefaultTask。



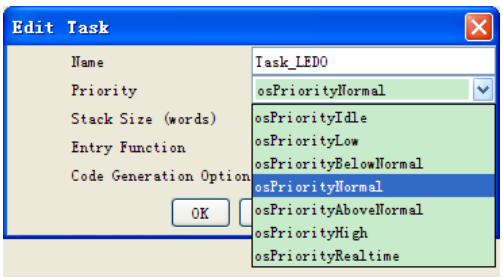
双击蓝色的地方，弹出对话框，将任务名修改为 Task_LED0，将任务函数名修改为 Func_LED0。



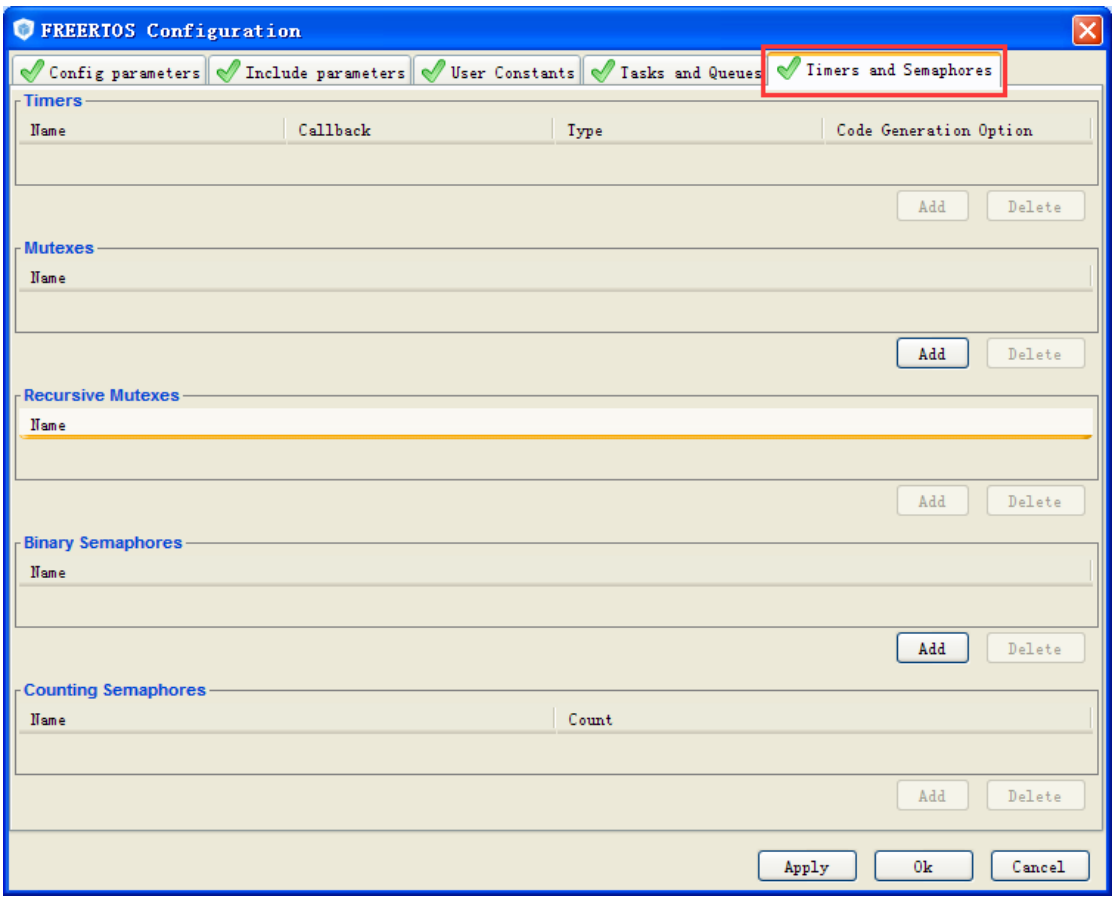
点击 Add 按钮，增加一个任务 Task_LED1，优先级设置为 Normal，函数名为 Func_LED1。



需要注意的是，STM32Cube 对 FreeRTOS 进行了一些修改，比如优先级只有 7 个，如下图。

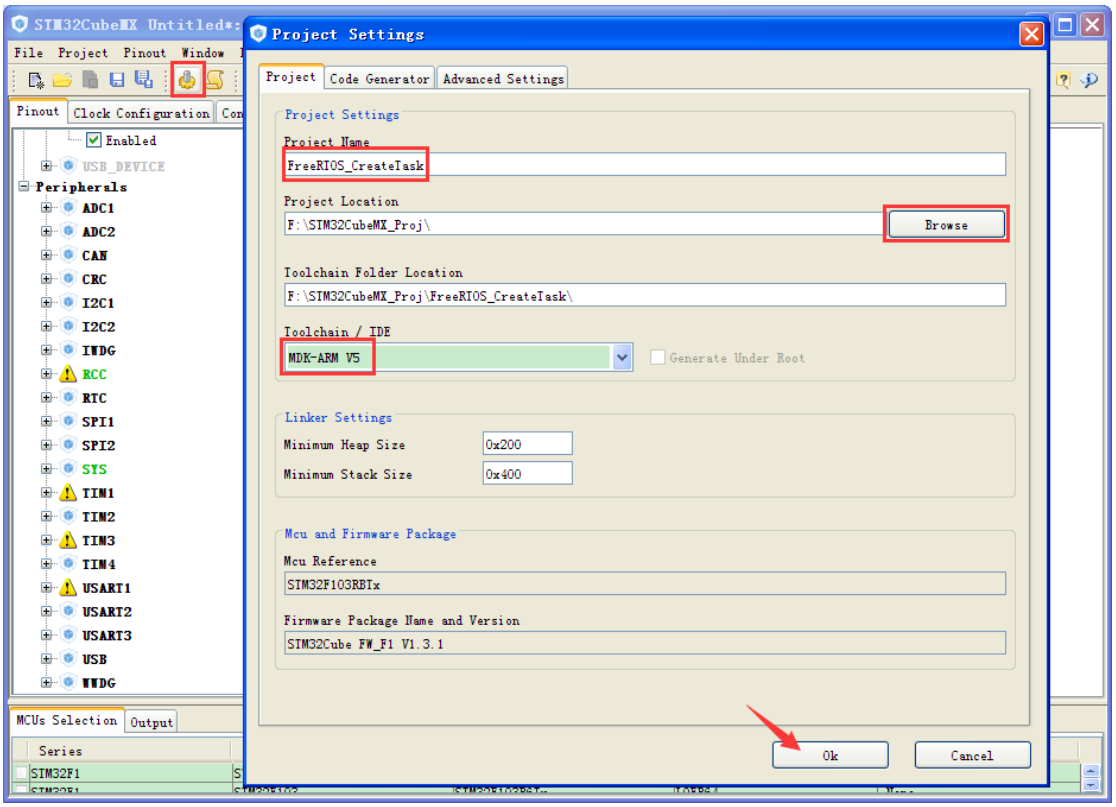


Timers and Semaphores 是添加软件定时器和信号量的选项。

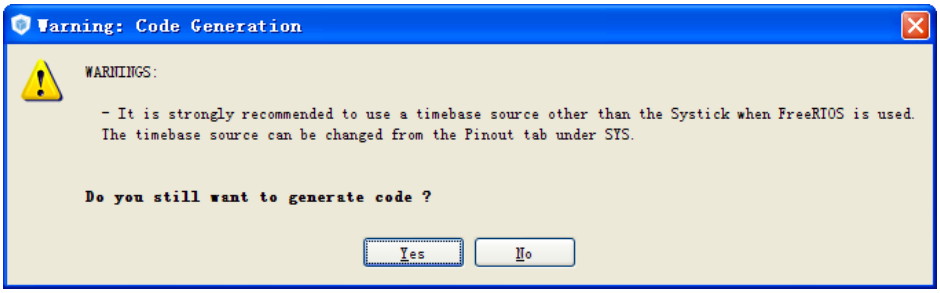


注：该步骤中，除了添加任务，其他的都使用默认参数。

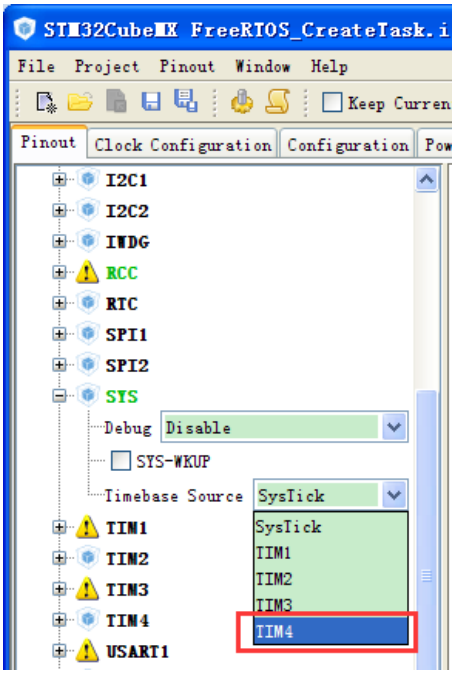
Step7.生成代码。



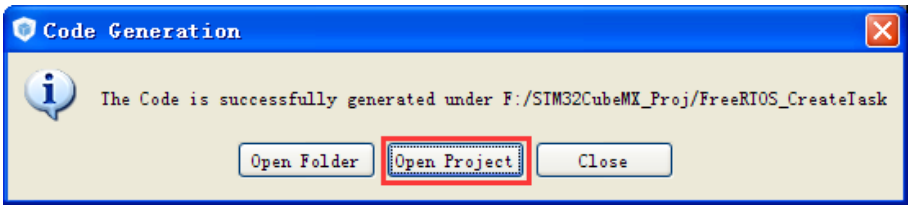
这时候会弹出一个警告。原因是 FreeRTSO 使用了 Systick 作为时钟节拍，而 HAL 库也使用了 Systick 作为 HAL_Delay()和各种 timeout 的时钟基准。因此需要将 HAL 的时钟基准改为其他 TIMER。一般使用一个基本定时器。



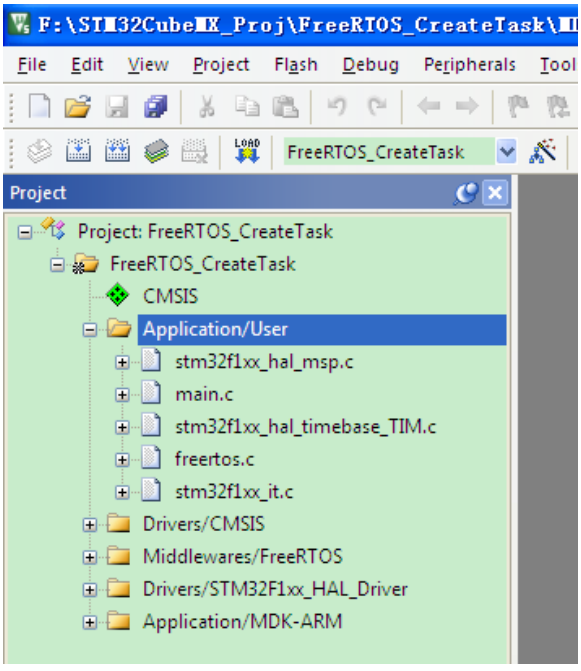
点击“No”按钮，然后在 Pinout 设置页面选择时基源为 TIM4



再次点击代码生成按钮，等完成后直接打开工程。



工程基本组织结构如下图，其中 Application/User 组中的文件是用户可以修改的，而其他组中的文件一般不进行修改。



Step8.分析程序结构。

在进入 main 函数之前，先定义了两个变量，声明了几个函数。

```
/* Private variables ---------------------
osThreadId Task_LED0Handle;
osThreadId Task_LED1Handle;

/* USER CODE BEGIN PV */
/* Private variables ---------------------

/* USER CODE END PV */

/* Private function prototypes -----------
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
void Func_LED0(void const * argument);
void Func_LED1(void const * argument);

```

再看 main 函数。将 main 函数整理，删除很多注释之后，得到下图所示内容。

```
int main(void)
{
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* Configure the system clock */          ①
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();

  /* Create the thread(s) */
  /* definition and creation of Task_LED0 */                    ②
  osThreadDef(Task_LED0, Func_LED0, osPriorityNormal, 0, 128);
  Task_LED0Handle = osThreadCreate(osThread(Task_LED0), NULL);

  /* definition and creation of Task_LED1 */
  osThreadDef(Task_LED1, Func_LED1, osPriorityNormal, 0, 128);
  Task_LED1Handle = osThreadCreate(osThread(Task_LED1), NULL);

  /* Start scheduler */
  osKernelStart();   ③
  /* We should never get here as control is now taken by the scheduler */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
  }
}
```

其中第①部分，是硬件配置；第②部分，创建两个线程(或称任务)；第③部分，启动调度器。这就是程序的基本结构。

启动调度器后，程序就由 FreeRTOS 的调度器管理了，将会被执行的是两个已经创建的任务函数 Func_LED0 和 Func_LED1，后面的 while(1)是不会执行到的。

Step9.添加代码。

在 main.c 文件中，找到前面配置添加的两个任务函数，Func_LED0 和 Func_LED1，然后在里面分别添加 LED0 和 LED1 的控制代码。

```
/* Func_LED0 function */
void Func_LED0(void const * argument)
{

  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
      osDelay(500);
      HAL_GPIO_WritePin(LED0_GPIO_Port, LED0_Pin, GPIO_PIN_RESET);  // LED0 = 0;
      osDelay(500);
      HAL_GPIO_WritePin(LED0_GPIO_Port, LED0_Pin, GPIO_PIN_SET);  // LED0 = 1;
  }
  /* USER CODE END 5 */
}

```

```
/* Func_LED1 function */
void Func_LED1(void const * argument)
{
  /* USER CODE BEGIN Func_LED1 */
  /* Infinite loop */
  for(;;)
  {
      osDelay(1000);
      HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET);  // LED1 = 0;
      osDelay(1000);
      HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_SET);   // LED1 = 1;
  }
  /* USER CODE END Func_LED1 */
}
```

Step10.编译下载运行。LED0 和 LED1 分别闪烁，LED0 闪烁周期是 1 秒，LED1 的周期是 2 秒。

程序分析：

1.分析语句：osThreadDef(Task_LED0, Func_LED0, osPriorityNormal, 0, 128);

osThreadDef(…)并不是一个函数，而是一个宏。

其定义在 cmsis_os.h 文件中，作用是定义一个 osThreadDef_t 结构体。

```
#define osThreadDef(name, thread, priority, instances, stacksz)  \
const osThreadDef_t os_thread_def_##name = \
{ #name, (thread), (priority), (instances), (stacksz)  }
```

在 cmsis_os.h 文件中，osThreadDef_t 结构体的定义如下：

```
/// Thread Definition structure contains startup information of a thread.
/// \note CAN BE CHANGED: \b os_thread_def is implementation specific in every CMSIS-RTOS.
typedef struct os_thread_def  {
  char                  *name;        ///< Thread name
  os_pthread            pthread;      ///< start address of thread function
  osPriority            tpriority;    ///< initial thread priority
  uint32_t              instances;    ///< maximum number of instances of that thread function
  uint32_t              stacksize;    ///< stack size requirements in bytes; 0 is default stack size
} osThreadDef_t;
```

因此，将 osThreadDef(Task_LED0, Func_LED0, osPriorityNormal, 0, 128);展开结果就是

const osThreadDef_t os_thread_def_Task_LED0 =

{ Task_LED0, (Func_LED0), (osPriorityNormal), (0), (128) };

即，定义了一个名为 os_thread_def_Task_LED0 的 osThreadDef_t 类型结构体，并赋值给各个成员变量。

2.分析语句：Task_LED0Handle = osThreadCreate(osThread(Task_LED0), NULL);

同样的，osThread(…)也是一个宏定义，在 cmsis_os.h 文件中可查到。

```
/// Access a Thread definition.
/// \param          name
/// \note CAN BE CHANGED: The par
///       macro body is implement
#define osThread(name)  \
&os_thread_def_##name
```

osThread(Task_LED0)展开的结果就是 &os_thread_def_ Task_LED0。

因此，将 Task_LED0Handle = osThreadCreate(osThread(Task_LED0), NULL);展开结果就是 Task_LED0Handle = osThreadCreate(&os_thread_def_Task_LED0, NULL);

所以上面分析的两句话，其过程就是定义一个结构体变量，然后将结构体作为参数传递给 osThreadCreate()函数，创建一个任务。

3.分析 osThreadCreate()函数。

```
/**
* @brief  Create a thread and add it to Active Threads and set it to state READY.
* @param  thread_def    thread definition referenced with \ref osThread.
* @param  argument      pointer that is passed to the thread function as start argument.
* @retval thread ID for reference by other functions or NULL in case of error.
* @note   MUST REMAIN UNCHANGED: \b osThreadCreate shall be consistent in every CMSIS-RTOS.
*/
osThreadId osThreadCreate (const osThreadDef_t *thread_def, void *argument)
{
  TaskHandle_t handle;


  if (xTaskCreate((TaskFunction_t)thread_def->pthread,(const portCHAR *)thread_def->name,
              thread_def->stacksize, argument, makeFreeRtosPriority(thread_def->tpriority),
              &handle) != pdPASS)  {
    return NULL;
  }

  return handle;
}
```

查看其源码，可以发现，这个函数实际上调用了 xTaskCreate()函数，这才是原生 FreeRTOS 的 API 函数。

STM32CubeMX 的工程师根据 CMSIC 接口标准对 FreeRTOS 的 API 函数进行了二次封装，使用户开发更加容易。封装后的函数接口都放在 cmsis_os.h 文件中。

其实，在开发过程中，不需要像上面的分析过程那样，将函数或者宏定义展开进行详细分析。我们知道每个接口参数的意义，并会使用该接口就行了。

附加内容：FreeRTOS 任务调度策略探讨

本例中的两个任务函数 Func_LED0 和 Func_LED1，他们实际占用 CPU 的时间很少，在调用 osDelay()函数之后，它们就进入阻塞状态了，它们在等待“定时时间到”事件。在用户任务都进入阻塞状态时，运行的是空闲任务。空闲任务是启动调度器时自动创建的。

本例中，两个任务的优先级是一样的，都是 osPriorityNormal。但是由于调用了 osDelay()函数，它们进入阻塞状态时就让出了 CPU 的使用权。因此，两个任务看上去就像并行执行的一样。

如果把其中的一任务的优先级设置成 osPriorityLow 或者 osPriorityHigh，让两个任务的优先级不同，会怎样呢？结果是，运行起来还是像并行执行的一样。

我们都知道，如果两个就绪的任务优先级不同，那么优先级高的任务得到运行。那么，如果两个就绪任务优先级相同，且一直处于就绪状态，那么在 FreeRTOS 中将如何运行呢？下面将通过实验检验其运行过程。

写一个用 for 循环实现的延时函数：

```
void my_delay(uint16_t n)
{
    uint16_t i;
    for (;n;n--){
        for(i=0;i<5000;i++);
    }
}
```

然后在任务中替代原来的 osDelay()函数函数。这样，两个任务会一直处于可运行的状态，因为他们“总是有事情要做”。

```c
/* Func_LED0 function */
void Func_LED0(void const * argument)
{

  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
      my_delay(500);
      HAL_GPIO_WritePin(LED0_GPIO_Port, LED0_Pin, GPIO_PIN_RESET);  // LED0 = 0;
      my_delay(500);
      HAL_GPIO_WritePin(LED0_GPIO_Port, LED0_Pin, GPIO_PIN_SET);  // LED0 = 1;
  }
  /* USER CODE END 5 */
}
```

```c
/* Func_LED1 function */
void Func_LED1(void const * argument)
{
  /* USER CODE BEGIN Func_LED1 */
  /* Infinite loop */
  for(;;)
  {
      my_delay(500);
      HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET);  // LED1 = 0;
      my_delay(500);
      HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_SET);  // LED1 = 1;
  }
  /* USER CODE END Func_LED1 */
}
```

a).如果两个任务的优先级不同，则运行结果是：只有优先级高的任务得到运行。

b).如果两个任务的优先级相同，则运行结果是：两个任务都得到运行，但是 LED 的闪烁频率比单独运行时的低。按照上述代码，两个任务调用的都是 my_delay(500);，结果就是 LED 闪烁频率减半，相当于单个任务运行时的 my_delay(1000);。

从 b)的情况可知，FreeRTOS 在任务优先级相同时，会分配给各任务相同的 CPU 时间，即任务会轮流执行相等的时间片。

S.D.Lu 于 深圳
2016 年 8 月