FreeRTOS 实战教程 - 良许 - 2023版

此教程不会讲过多理论，以动手实操为主，解决大伙学了半天 FreeRTOS 操作系统不知道干什么的问题。

为什么要学 FreeRTOS？

更有钱途！！

- 只会裸机开发的单片机工程师，薪资注定不会高于会 FreeRTOS 的工程师；
- 有了 FreeRTOS 基础，对于将来学习 Linux 操作系统会更加有帮助；

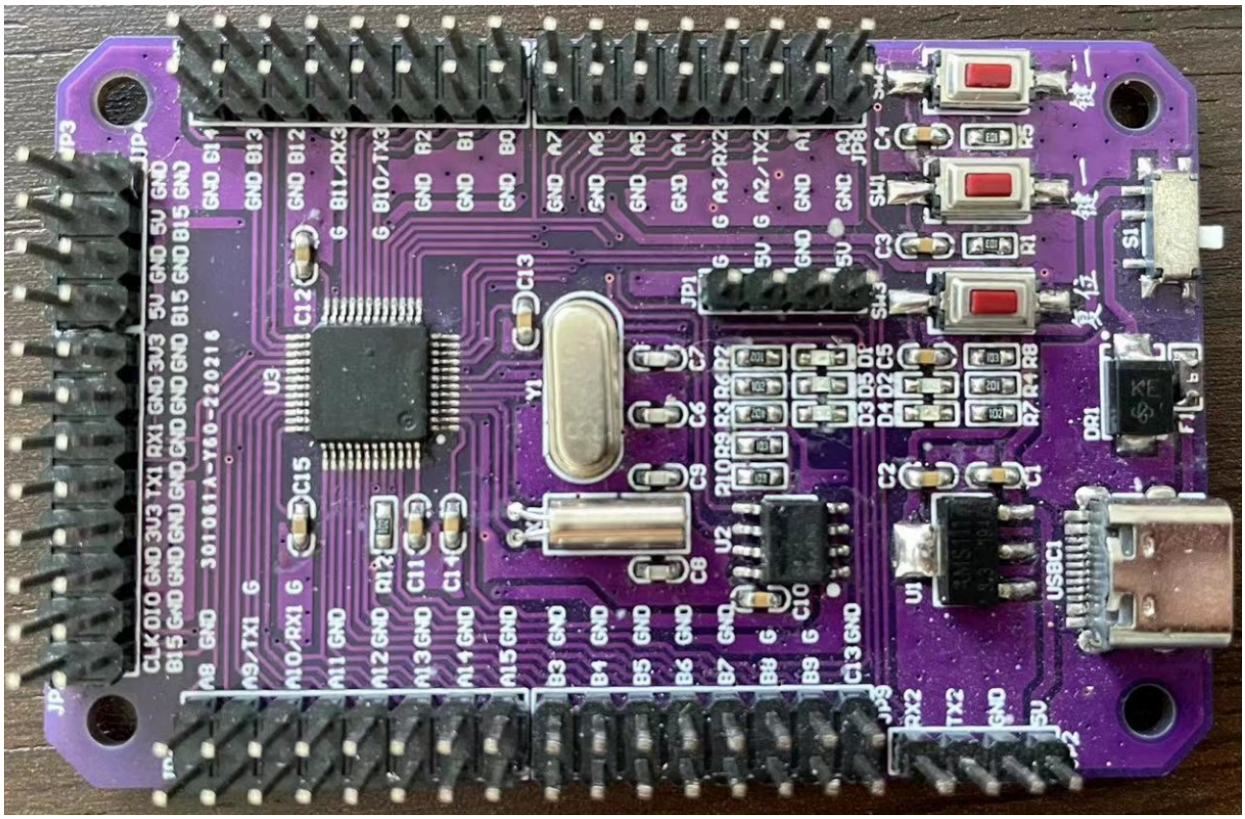
如何学好 FreeRTOS？

无它，多写代码，多做项目！

一定要把本课程里所有项目全部自己动手做一遍，加深理解。**光看不练假把式！！**

学习本课程前置要求

- C 语言熟练；
- 上官二号课程一定要好好学一遍，本课程依然基于上官二号。



准备好了吗？快乐启程~

一、FreeRTOS 介绍

什么是 FreeRTOS？

Free即免费的，RTOS的全称是Real time operating system，中文就是实时操作系统。

注意：RTOS不是指某一个确定的系统，而是指一类操作系统。比如：uc/OS，FreeRTOS，RTX，RT-Thread等这些都是RTOS类操作系统。

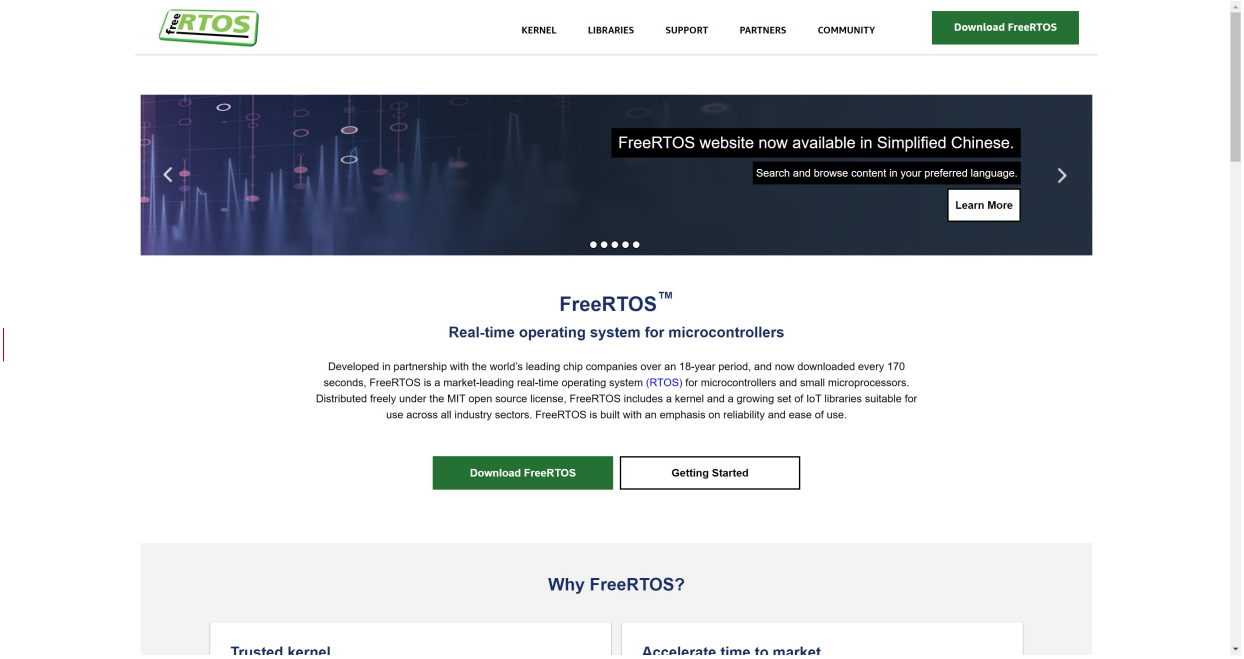
> FreeRTOS是一个迷你的实时操作系统内核。作为一个轻量级的操作系统，功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能、软件定时器、协程等，可基本满足较小系统的需要。
> 由于RTOS需占用一定的系统资源(尤其是RAM资源)，只有μC/OS-II、embOS、salvo、FreeRTOS等少数实时操作系统能在小RAM单片机上运行。相对μC/OS-II、embOS等商业操作系统，FreeRTOS操作系统是完全免费的操作系统，具有源码公开、可移植、可裁减、调度策略灵活的特点，可以方便地移植到各种单片机上运行，其最新版本为10.4.4版。
>
> （以上来自百度百科）

为什么选择 FreeRTOS？

- FreeRTOS 是免费的；
- 很多半导体厂商产品的SDK（Software Development Kit）软件开发工具包，就使用FreeRTOS作为其操作系统，尤其是WIFI、蓝牙这些带有协议栈的芯片或模块。
- 简单，因为FreeRTOS的文件数量很少。

FreeRTOS 资料与源码下载

最好的资料就是官网提供的资料！[点击直达官网]



裸机开发与 FreeRTOS

众所周知，游戏与女朋友不可兼得！

游戏　　女朋友

裸机开发：

- 玩游戏，结果女朋友生气，分手！
- 陪女朋友，无法玩游戏，抑郁症！

FreeRTOS：

玩1秒游戏 -- 陪1秒女朋友 -- 玩1秒游戏 -- 陪1秒女朋友 -- 玩1秒游戏 -- 陪1秒女朋友 ....

（累死……）

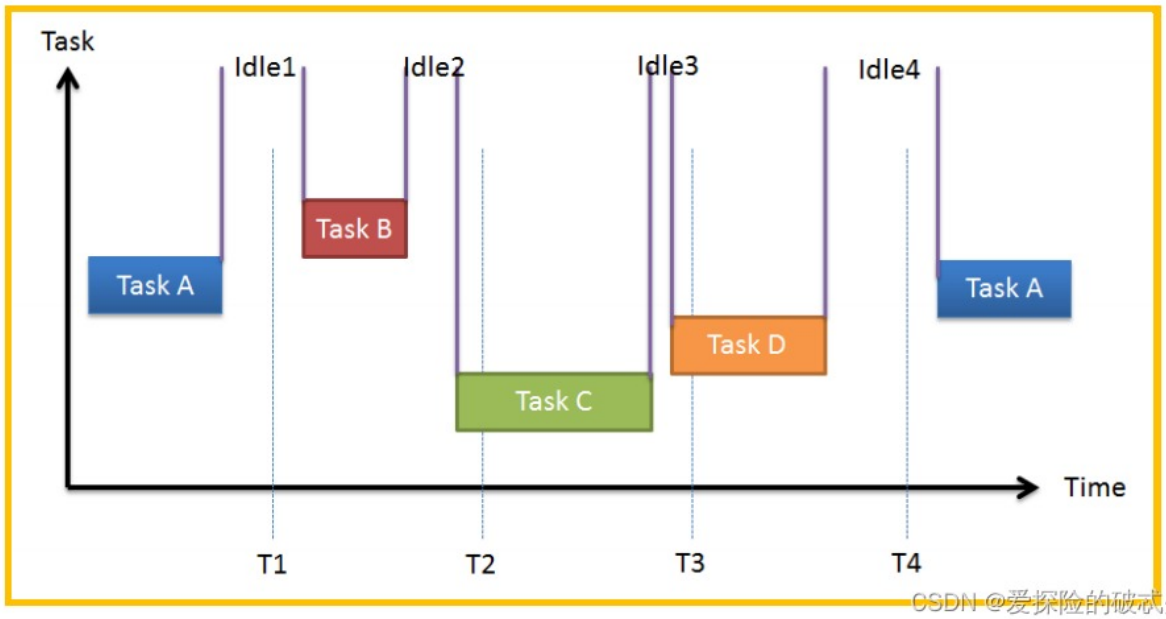
但 CPU 是个无情的战斗机器，可以快速在两个乃至多个任务间快速切换，并且不觉得劳累，实现二者兼顾。

FreeRTOS 实现多任务的原理

严格来说 FreeRTOS 并不是实时操作系统，因为它是**分时复用**的。

系统将时间分割成很多时间片，然后轮流执行各个任务。

每个任务都是独立运行的，互不影响，由于切换的频率很快，就感觉像是同时运行的一样。



二、移植 FreeRTOS 到上官二号平台

手动移植

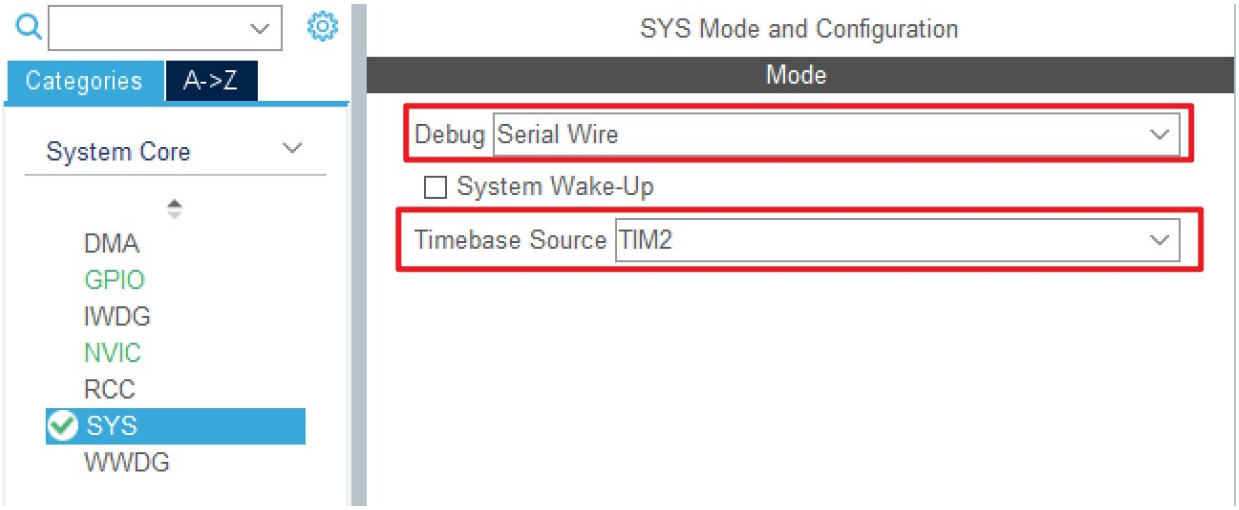
过程复杂且繁琐，对新手不友好。如有需要手动移植，可参照以下文章：

FreeRTOS移植到STM32教程（点击直达）

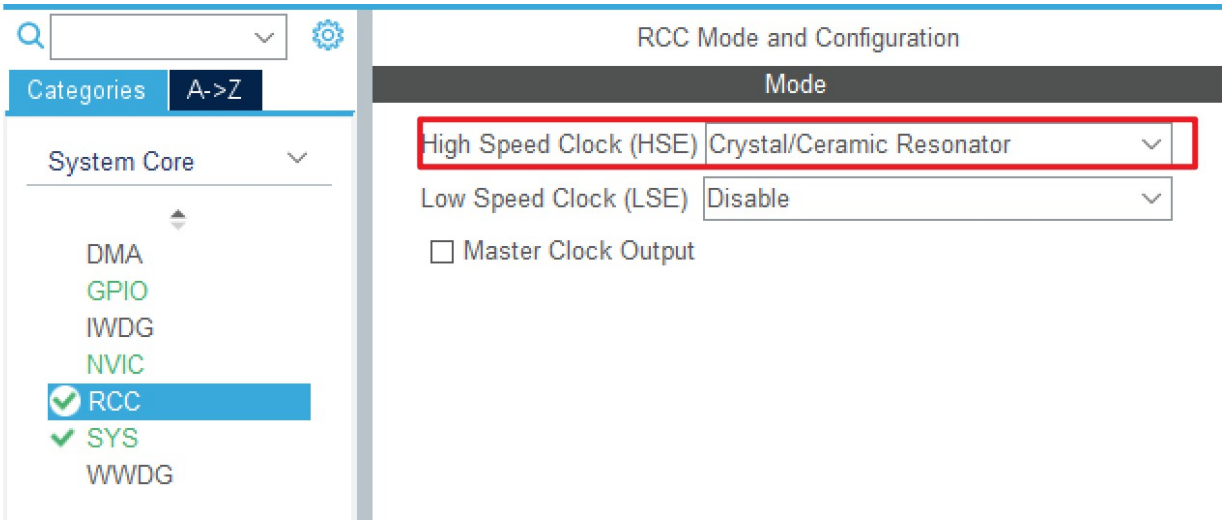
使用CubeMX快速移植

快速移植流程

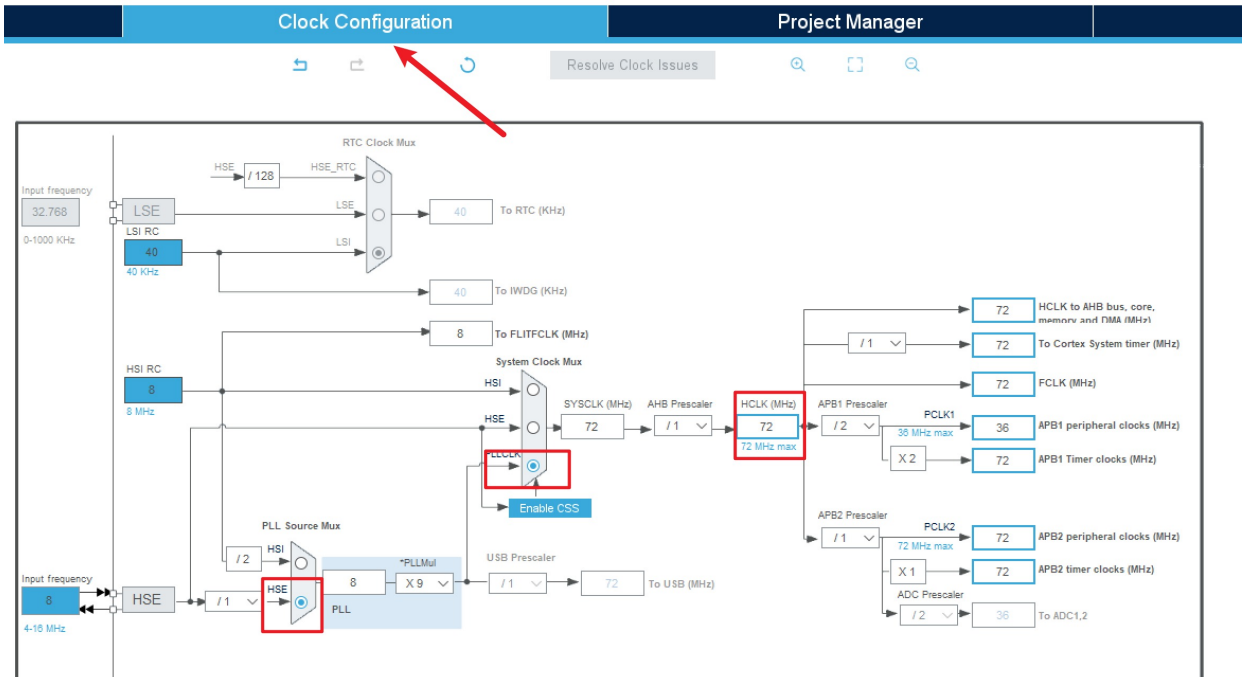
1. 在 SYS 选项里，将 Debug 设为 Serial Wire ，并且将 Timebase Source 设为 TIM2 （其它定时器也行）。为何要如此配置？下文解说。



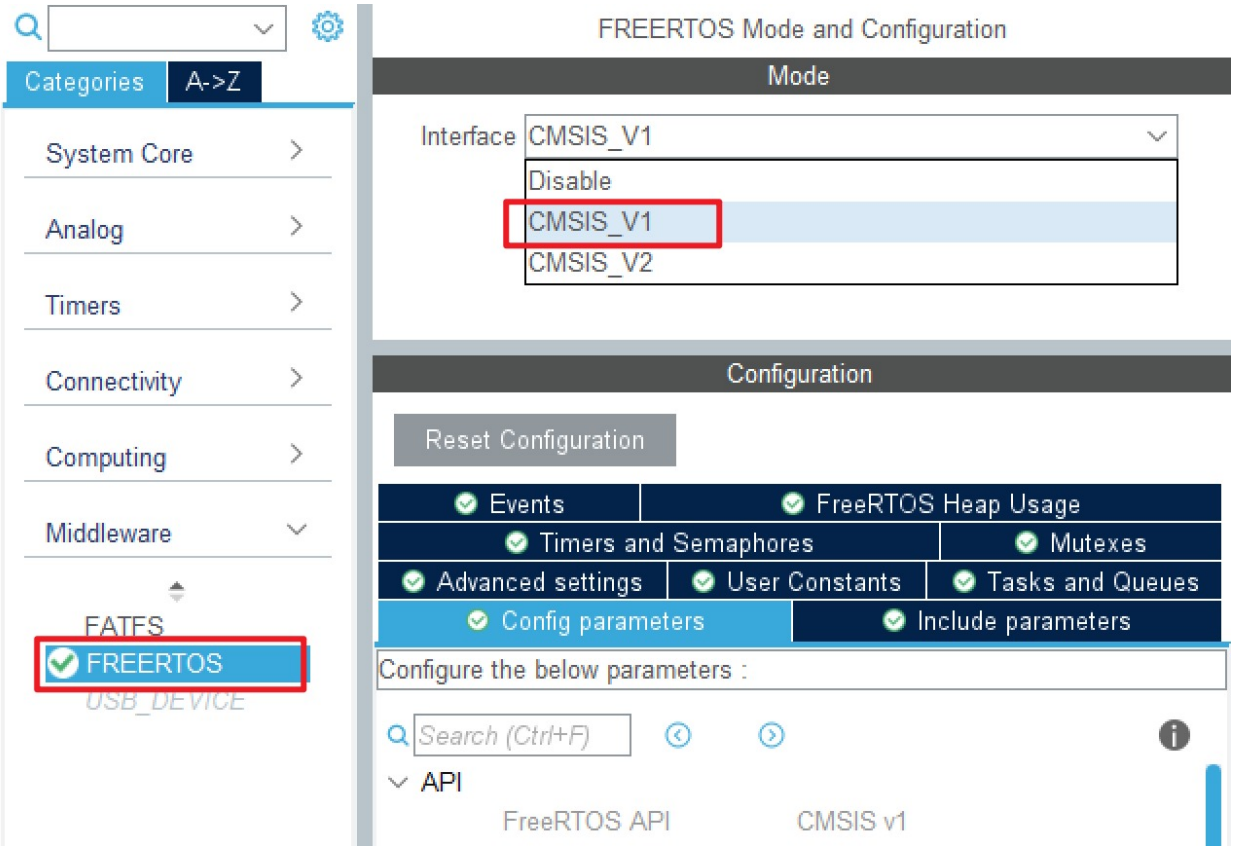
2. 将 RCC 里的 HSE 设置为 Crystal/Ceramic Resonator 。



3. 时钟按下图配置



4. 选择 FREERTOS 选项，并将 Interface 改为 CMSIS_V1 。V1 和 V2 有啥区别？下文解释。



5. 配置项目信息，并导出代码。

一些常见问题

1. Timebase Source 为什么不能设置为 SysTick？

裸机的时钟源默认是 SysTick，但是开启 FreeRTOS 后，FreeRTOS会占用 SysTick （用来生成1ms 定时，用于任务调度），所以需要需要为其他总线提供另外的时钟源。

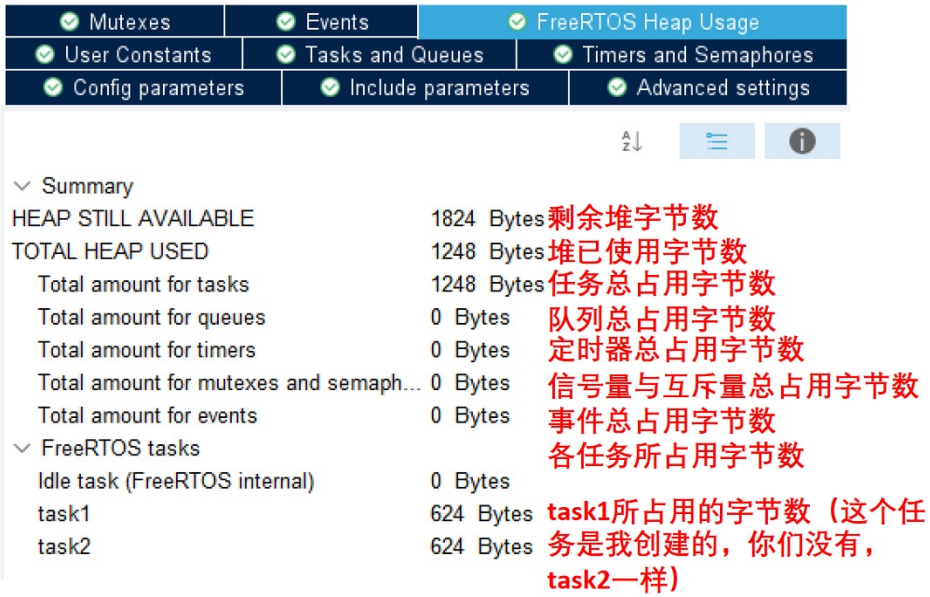
2. FreeRTOS 版本问题

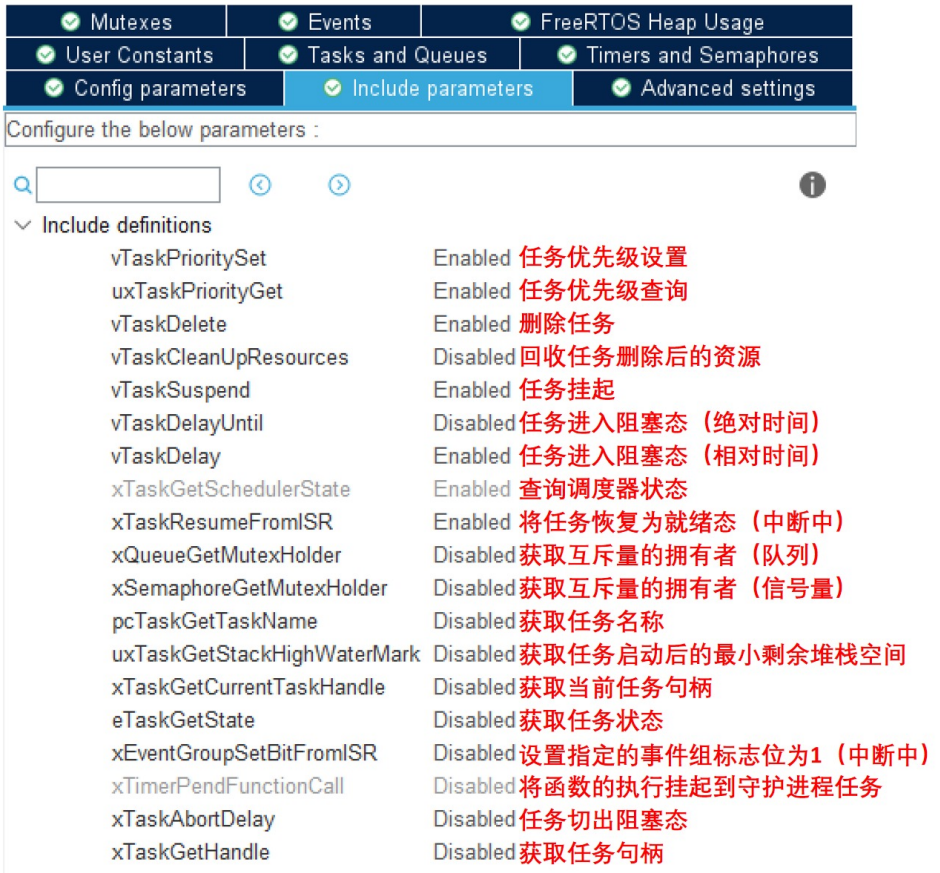
V2 的内核版本更高，功能更多，在大多数情况下 V1 版本的内核完全够用。

3. FreeRTOS 各配置选项卡的解释

- Events：事件相关的创建
- Task and Queues： 任务与队列的创建
- Timers and Semaphores： 定时器和信号量的创建
- Mutexes： 互斥量的创建
- FreeRTOS Heap Usage： 用于查看堆使用情况
- config parameters： 内核参数设置，用户根据自己的实际应用来裁剪定制 FreeRTOS 内核
- Include parameters： FreeRTOS 部分函数的使能
- User Constants： 相关宏的定义，可以自建一些常量在工程中使用
- Advanced settings：高级设置

4. 内核配置、函数使能的一些翻译





内核参数的理解内容非常多，可以参考以下文章：

FreeRTOS内核配置说明（点击直达）

三、任务的创建与删除

1. 什么是任务?

任务可以理解为进程/线程，创建一个任务，就会在内存开辟一个空间。

比如：

玩游戏、陪女朋友，都可以视为任务

Windows 系统中的 MarkText 、谷歌浏览器、记事本，都是任务。

任务通常都含有 `while(1)` 死循环。

2. 任务创建与删除相关函数

任务创建与删除相关函数有如下三个：

函数名称	函数作用
xTaskCreate()	动态方式创建任务

函数名称	函数作用
xTaskCreateStatic()	静态方式创建任务
vTaskDelete()	删除任务

任务动态创建与静态创建的区别：

动态创建任务的堆栈由系统分配，而静态创建任务的堆栈由用户自己传递。

通常情况下使用动态方式创建任务。

xTaskCreate 函数原型

```
BaseType_t xTaskCreate
(
      TaskFunction_t                pxTaskCode,       /* 指向任务函数的指针 */
      const char * const            pcName,           /* 任务名字，最大长度configMAX_TASK_NAME_LEN */
      const  configSTACK_DEPTH_TYPE usStackDepth,     /* 任务堆栈大小，注意字为单位 */
      void * const                  pvParameters,     /* 传递给任务函数的参数 */
      UBaseType_t                   uxPriority,       /* 任务优先级，范围：0 ~ configMAX_PRIORITIES - 1 */
      TaskHandle_t * const          pxCreatedTask     /* 任务句柄，就是任务的任务控制块 */
)
```

1. pvTaskCode：指向任务函数的指针，任务必须实现为永不返回（即连续循环）；
2. pcName：任务的名字，主要是用来调试，默认情况下最大长度是16；
3. pvParameters：指定的任务栈的大小；
4. uxPriority：任务优先级，**数值越大，优先级越大**；
5. pxCreatedTask：用于返回已创建任务的句柄可以被引用。

返回值	描述
pdPASS	任务创建成功
errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY	任务创建失败

官方案例：

```
/* Task to be created. */
void vTaskCode( void * pvParameters )
{
    /* The parameter value is expected to be 1 as 1 is passed in the
    pvParameters value in the call to xTaskCreate() below.
    configASSERT( ( ( uint32_t ) pvParameters ) == 1 );

    for( ;; )
    {
        /* Task code goes here. */
    }
}

/* Function that creates a task. */
```

```
void vOtherFunction( void )
{
BaseType_t xReturned;
TaskHandle_t xHandle = NULL;

    /* Create the task, storing the handle. */
    xReturned = xTaskCreate(
                    vTaskCode,       /* Function that implements the task. */
                    "NAME",          /* Text name for the task. */
                    STACK_SIZE,      /* Stack size in words, not bytes. */
                    ( void * ) 1,    /* Parameter passed into the task. */
                    tskIDLE_PRIORITY,/* Priority at which the task is created. */
                    &xHandle );      /* Used to pass out the created task's handle.
*/

    if( xReturned == pdPASS )
    {
        /* The task was created.  Use the task's handle to delete the task. */
        vTaskDelete( xHandle );
    }
}
```

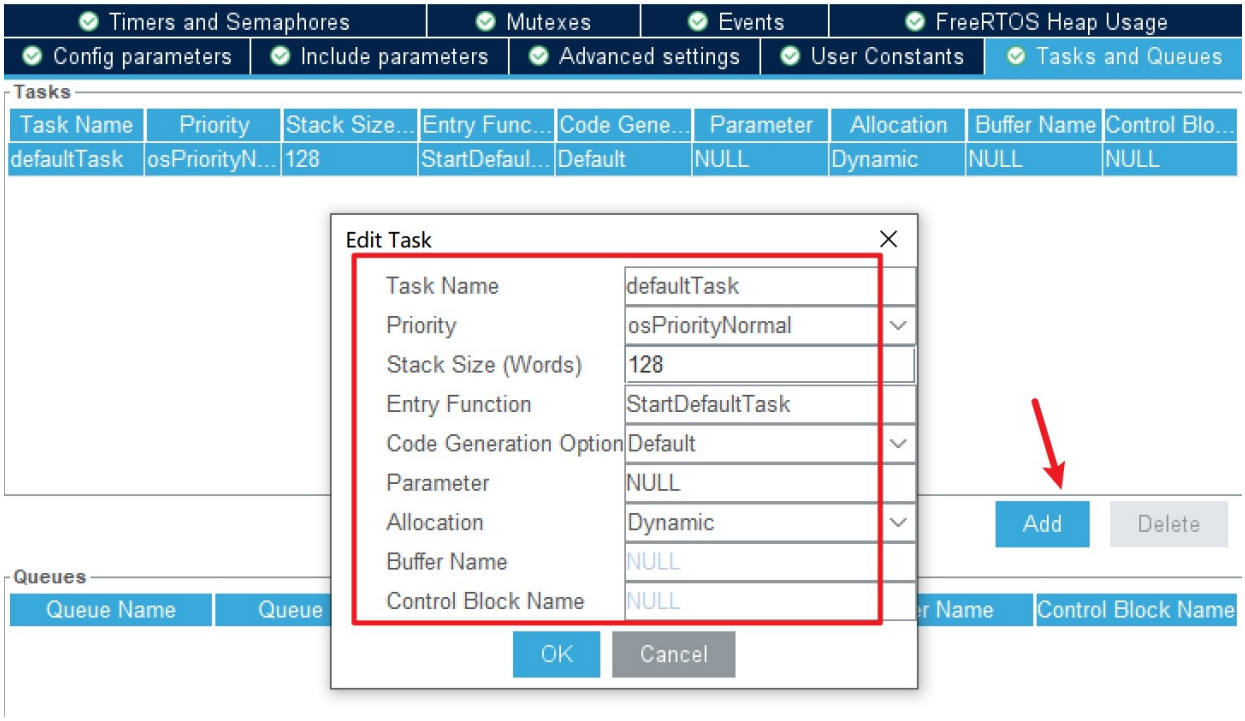
vTaskDelete 函数原型

```
void vTaskDelete(TaskHandle_t xTaskToDelete);
```

只需将待删除的任务句柄传入该函数，即可将该任务删除。

当传入的参数为NULL，则代表删除任务自身（当前正在运行的任务）。

3. 实操



四、任务调度

什么是任务调度?

调度器就是使用相关的调度算法来决定当前需要执行的哪个任务。

FreeRTOS中开启任务调度的函数是 `vTaskStartScheduler()` ，但在 CubeMX 中被封装为 `osKernelStart()` 。

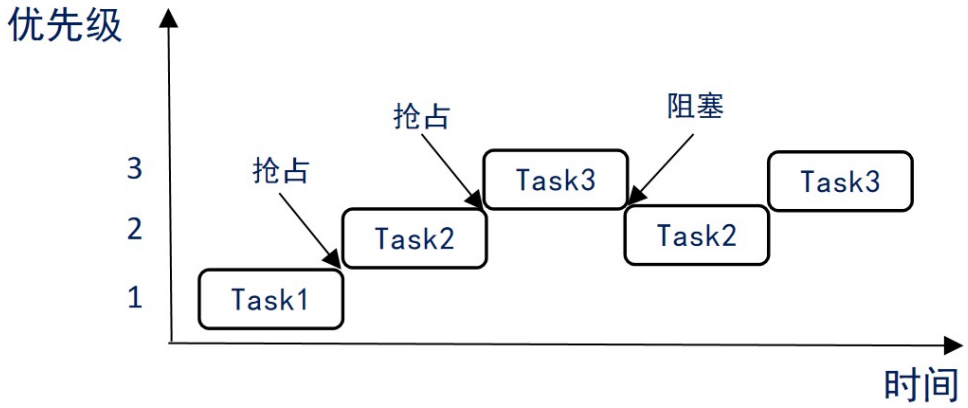
FreeRTOS的任务调度规则是怎样的?

FreeRTOS 是一个实时操作系统，它所奉行的调度规则：

1. 高优先级抢占低优先级任务，系统永远执行最高优先级的任务（即**抢占式调度**）
2. 同等优先级的任务轮转调度（即**时间片调度**）

还有一种调度规则是协程式调度，但官方已明确表示不更新，主要是用在小容量的芯片上，用得也不多。

抢占式调度运行过程



Task 1：玩游戏

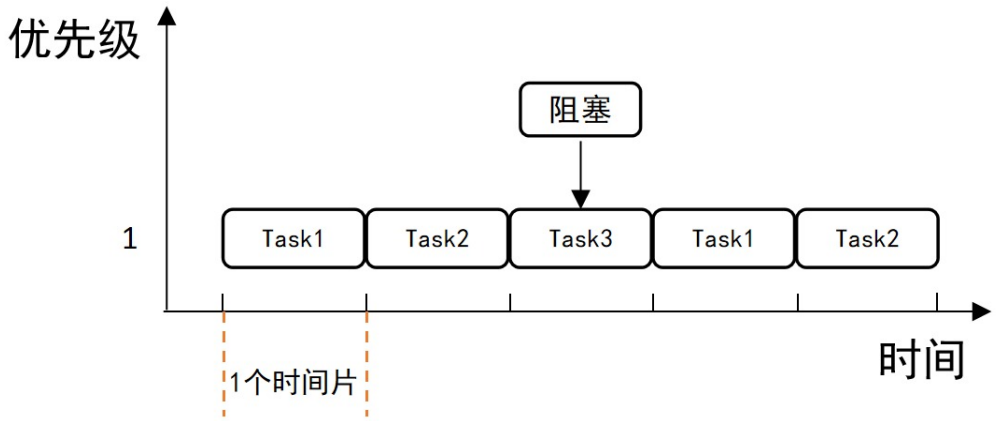
Task 2：老妈喊你吃饭

Task 3：女朋友喊你看电视

总结：

1. 高优先级任务，优先执行；
2. 高优先级任务不停止，低优先级任务无法执行；
3. 被抢占的任务将会进入就绪态

时间片调度运行过程



总结：

1. 同等优先级任务，轮流执行，时间片流转；
2. 一个时间片大小，取决为滴答定时器中断周期；
3. 注意没有用完的时间片不会再使用，下次任务 Task3 得到执行，还是按照一个时间片的时钟节拍运行

五、任务的状态

FreeRTOS中任务共存在4种状态：

- **Running 运行态**

当任务处于实际运行状态称之为运行态，即CPU的使用权被这个任务占用（同一时间仅一个任务处于运行态）。

- **Ready 就绪态**

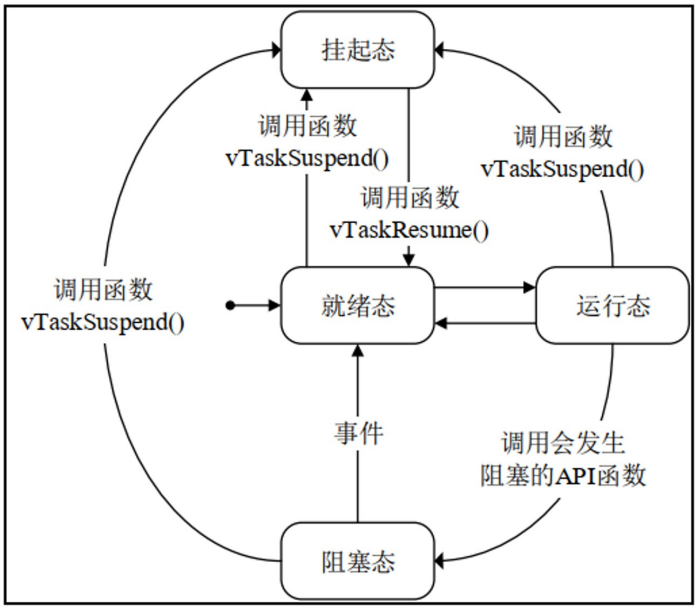
处于就绪态的任务是指那些能够运行（没有被阻塞和挂起），但是当前没有运行的任务，因为同优先级或更高优先级的任务正在运行。

- **Blocked 阻塞态**

如果一个任务因延时，或等待信号量、消息队列、事件标志组等而处于的状态被称之为阻塞态。

- **Suspended 挂起态**

类似暂停，通过调用函数 `vTaskSuspend()` 对指定任务进行挂起，挂起后这个任务将不被执行，只有调用函数 `xTaskResume()` 才可以将这个任务从挂起态恢复。



总结：

1. 仅就绪态可转变成运行态
2. 其他状态的任务想运行，必须先转变成就绪态

任务综合小实验

实验需求

创建 4 个任务：taskLED1，taskLED2，taskKEY1，taskKEY2，任务要求如下：

taskLED1：间隔 500ms 闪烁 LED1；

taskLED2：间隔 1000ms 闪烁 LED2；

taskKEY1：如果 taskLED1 存在，则按下 KEY1 后删除 taskLED1 ，否则创建 taskLED1 ；

taskKEY2：如果 taskLED2 正常运行，则按下 KEY2 后挂起 taskLED2 ，否则恢复 taskLED2

cubeMX配置

Reset Configuration

Mutexes | Events | FreeRTOS Heap Usage
User Constants | Tasks and Queues | Timers and Semaphores
Config parameters | Include parameters | Advanced settings

Tasks

Task Name	Priority	St...	Entry Function	Code ...	Par...	Allocati...	Buff...	Control Block Name
taskLED1	osPriorityNormal	128	StartTaskLED1	Default	NULL	Dynamic	NULL	NULL
taskLED2	osPriorityNormal	128	StartTaskLED2	Default	NULL	Dynamic	NULL	NULL
taskKEY1	osPriorityNormal	128	StartTaskKEY1	Default	NULL	Dynamic	NULL	NULL
taskKEY2	osPriorityNormal	128	StartTaskKEY2	Default	NULL	Dynamic	NULL	NULL

代码实现

```c
/* USER CODE BEGIN Header_StartTaskLED1 */
void StartTaskLED1(void const * argument)
{
  /* USER CODE BEGIN StartTaskLED1 */
  /* Infinite loop */
  for(;;)
  {
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_8);
    osDelay(500);
  }
  /* USER CODE END StartTaskLED1 */
}

/* USER CODE END Header_StartTaskLED2 */
void StartTaskLED2(void const * argument)
{
  /* USER CODE BEGIN StartTaskLED2 */
  /* Infinite loop */
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_9);
    osDelay(1000);
  }
  /* USER CODE END StartTaskLED2 */
}

/* USER CODE END Header_StartTaskKEY1 */
void StartTaskKEY1(void const * argument)
{
  /* USER CODE BEGIN StartTaskKEY1 */
  /* Infinite loop */
  for(;;)
  {
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
    {
      osDelay(20);
      if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
      {
        printf("KEY1按下! \r\n");
        if (taskLED1Handle == NULL)
        {
          printf("任务1不存在，准备创建任务1\r\n");
          osThreadDef(taskLED1, StartTaskLED1, osPriorityNormal, 0, 128);
          taskLED1Handle = osThreadCreate(osThread(taskLED1), NULL);
          if (taskLED1Handle != NULL)
            printf("任务1创建完成\r\n");
        }
        else
        {
          printf("删除任务1\r\n");
          osThreadTerminate(taskLED1Handle);
          taskLED1Handle = NULL;
```

```
        }
      }
      while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET);
    }
    osDelay(10);
  }
  /* USER CODE END StartTaskKEY1 */
}

/* USER CODE END Header_StartTaskKEY2 */
void StartTaskKEY2(void const * argument)
{
  /* USER CODE BEGIN StartTaskKEY2 */
  static int flag = 0;
  /* Infinite loop */
  for(;;)
  {
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
    {
      osDelay(20);
      if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
      {
        printf("KEY2按下! \r\n");
        if (flag == 0)
        {
          osThreadSuspend(taskLED2Handle);
          printf("任务2已暂停\r\n");
          flag = 1;
        }
        else
        {
          osThreadResume(taskLED2Handle);
          printf("任务2已恢复\r\n");
          flag = 0;
        }
      }
      while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET);
    }
    osDelay(10);
  }
  /* USER CODE END StartTaskKEY2 */
}
```

五、队列

什么是队列?

队列又称消息队列，是一种常用于任务间通信的数据结构，队列可以在任务与任务间、中断和任务间传递信息。

为什么不使用全局变量?

如果使用全局变量，兔子（任务1）修改了变量 a ，等待树獭（任务3）处理，但树獭处理速度很慢，在处理数据的过程中，狐狸（任务2）有可能又修改了变量 a ，导致树獭有可能得到的不是正确的数据。

在这种情况下，就可以使用队列。兔子和狐狸产生的数据放在流水线上，树獭可以慢慢一个个依次处理。

关于队列的几个名词：

队列项目：队列中的每一个数据；

队列长度：队列能够存储队列项目的最大数量；

创建队列时，需要指定队列长度及队列项目大小。

队列特点

1. 数据入队出队方式

通常采用**先进先出**（FIFO）的数据存储缓冲机制，即先入队的数据会先从队列中被读取。

也可以配置为后进先出（LIFO）方式，但用得比较少。

2. 数据传递方式

采用实际值传递，即将数据拷贝到队列中进行传递，也可以传递指针，在传递较大的数据的时候采用指针传递。

3. 多任务访问

队列不属于某个任务，任何任务和中断都可以向队列发送/读取消息

4. 出队、入队阻塞

当任务向一个队列发送消息时，可以指定一个阻塞时间，假设此时当队列已满无法入队。

阻塞时间如果设置为：

- 0：直接返回不会等待；

- 0~port_MAX_DELAY：等待设定的阻塞时间，若在该时间内还无法入队，超时后直接返回不再等待；
- port_MAX_DELAY：死等，一直等到可以入队为止。出队阻塞与入队阻塞类似；

队列相关 API 函数

1. 创建队列

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

参数:

- uxQueueLength：队列可同时容纳的最大项目数 。
- uxItemSize：存储队列中的每个数据项所需的大小（以字节为单位）。

返回值:
如果队列创建成功，则返回所创建队列的句柄 。 如果创建队列所需的内存无法分配 ， 则返回 NULL。

2. 写队列

写队列总共有以下几个函数:

函数	描述
xQueueSend()	往队列的尾部写入消息
xQueueSendToBack()	同 xQueueSend()
xQueueSendToFront()	往队列的头部写入消息
xQueueOverwrite()	覆写队列消息（只用于队列长度为 1 的情况）
xQueueSendFromISR()	在中断中往队列的尾部写入消息
xQueueSendToBackFromISR()	同 xQueueSendFromISR()
xQueueSendToFrontFromISR()	在中断中往队列的头部写入消息
xQueueOverwriteFromISR()	在中断中覆写队列消息（只用于队列长度为 1 的情况）

```
BaseType_t xQueueSend(
                          QueueHandle_t xQueue,
                          const void * pvItemToQueue,
                          TickType_t xTicksToWait
                      );
```

参数:

- xQueue：队列的句柄，数据项将发送到此队列。
- pvItemToQueue：待写入数据

- xTicksToWait：阻塞超时时间

返回值：

如果成功写入数据，返回 pdTRUE，否则返回 errQUEUE_FULL。

3. 读队列

读队列总共有以下几个函数：

函数	描述
xQueueReceive()	从队列头部读取消息，并删除消息
xQueuePeek()	从队列头部读取消息，但是不删除消息
xQueueReceiveFromISR()	在中断中从队列头部读取消息，并删除消息
xQueuePeekFromISR()	在中断中从队列头部读取消息

```
BaseType_t xQueueReceive(
                              QueueHandle_t xQueue,
                              void *pvBuffer,
                              TickType_t xTicksToWait
                            );
```

参数：

- xQueue：待读取的队列
- pvItemToQueue：数据读取缓冲区
- xTicksToWait：阻塞超时时间

返回值：

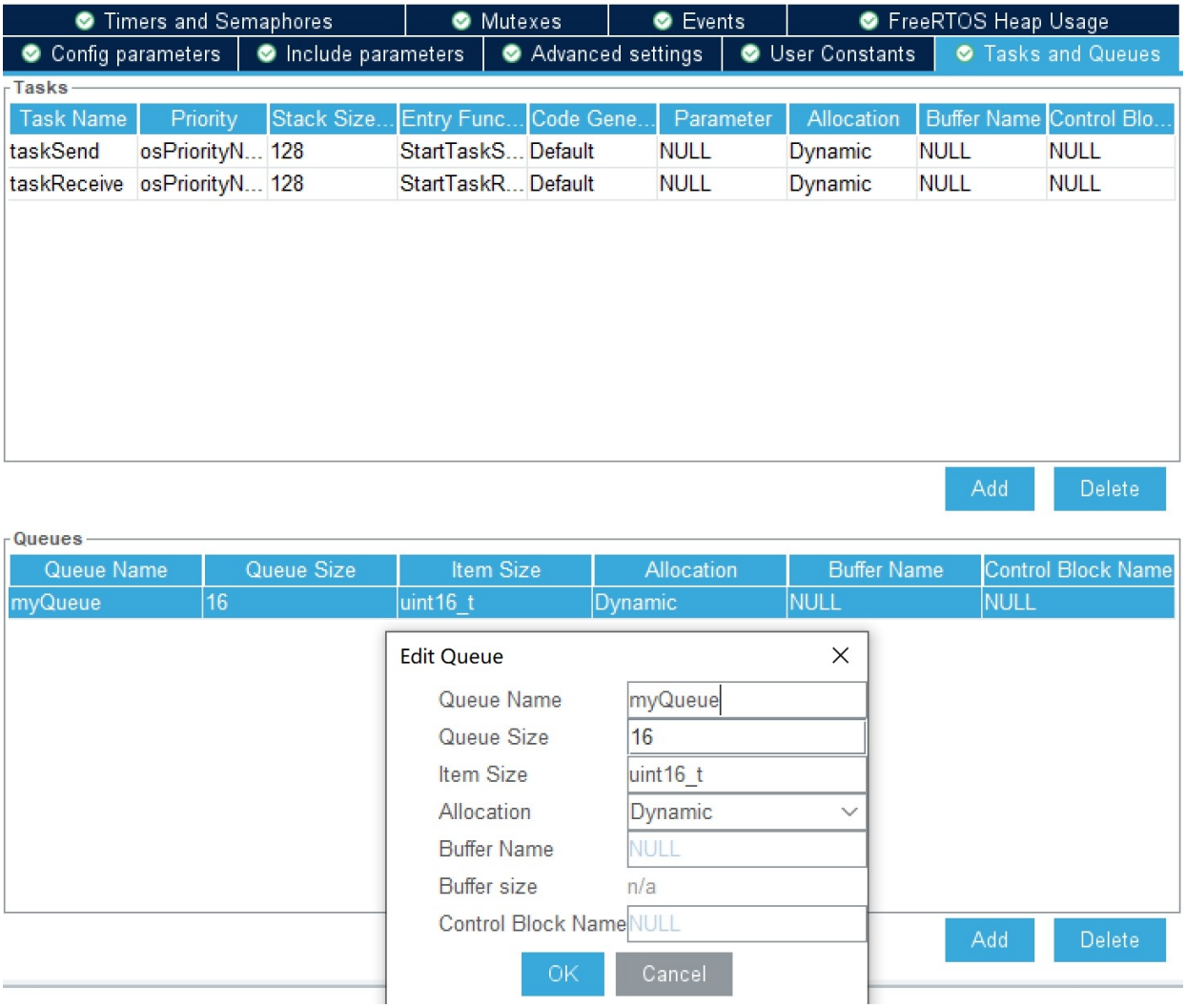
成功返回 pdTRUE，否则返回 pdFALSE。

实操

实验需求

创建一个队列，按下 KEY1 向队列发送数据，按下 KEY2 向队列读取数据。

cubeMX配置



代码实现

```
void StartTaskSend(void const * argument)
{
  /* USER CODE BEGIN StartTaskSend */
    uint16_t buf = 100;
    BaseType_t status;
  /* Infinite loop */
  for(;;)
  {
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
        {
            osDelay(20);
            if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
            {
                status = xQueueSend(myQueueHandle, &buf, 0);
                if (status == pdTRUE)
                    printf("写入队列成功，写入值为%d\r\n", buf);
                else
                    printf("写入队列失败\r\n");
            }
            while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET);
```

```
        }
    osDelay(10);
  }
  /* USER CODE END StartTaskSend */
}

void StartTaskReceive(void const * argument)
{
  /* USER CODE BEGIN StartTaskReceive */
    uint16_t buf;
    BaseType_t status;
  /* Infinite loop */
  for(;;)
  {
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
        {
            osDelay(20);
            if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
            {
                status = xQueueReceive(myQueueHandle, &buf, 0);
                if (status == pdTRUE)
                    printf("队列数据读取成功，读出值为%d\r\n", buf);
                else
                    printf("队列读取失败\r\n");
            }
            while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET);
        }
    osDelay(10);
  }
  /* USER CODE END StartTaskReceive */
}
```

六、二值信号量

什么是信号量?

信号量（Semaphore），是在多任务环境下使用的一种机制，是可以用来保证两个或多个关键代码段不被并发调用。

信号量这个名字，我们可以把它拆分来看，信号可以起到通知信号的作用，然后我们的量还可以用来表示资源的数量，当我们的量只有0和1的时候，它就可以被称作二值信号量，只有两个状态，当我们的那个量没有限制的时候，它就可以被称作为计数型信号量。

信号量也是队列的一种。

什么是二值信号量?

二值信号量其实就是一个长度为1，大小为零的队列，只有0和1两种状态，通常情况下，我们用它来进行互斥访问或任务同步。

互斥访问：比如门钥匙，只有获取到钥匙才可以开门

任务同步：比如我录完视频你才可以看视频

二值信号量相关 API 函数

函数	描述
xSemaphoreCreateBinary()	使用动态方式创建二值信号量
xSemaphoreCreateBinaryStatic()	使用静态方式创建二值信号量
xSemaphoreGive()	释放信号量
xSemaphoreGiveFromISR()	在中断中释放信号量
xSemaphoreTake()	获取信号量
xSemaphoreTakeFromISR()	在中断中获取信号量

1. 创建二值信号量

```
SemaphoreHandle_t xSemaphoreCreateBinary( void )
```

参数:

无

返回值:

成功，返回对应二值信号量的句柄；

失败，返回 NULL 。

2. 释放二值信号量

```
BaseType_t   xSemaphoreGive( SemaphoreHandle_t xSemaphore )
```

参数:

xSemaphore：要释放的信号量句柄

返回值:

成功，返回 pdPASS ；

失败，返回 errQUEUE_FULL 。

3. 获取二值信号量

```
BaseType_t   xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                TickType_t xTicksToWait );
```

参数:

xSemaphore：要获取的信号量句柄

xTicksToWait：超时时间，0 表示不超时，portMAX_DELAY表示卡死等待；

返回值:

成功，返回 pdPASS ；

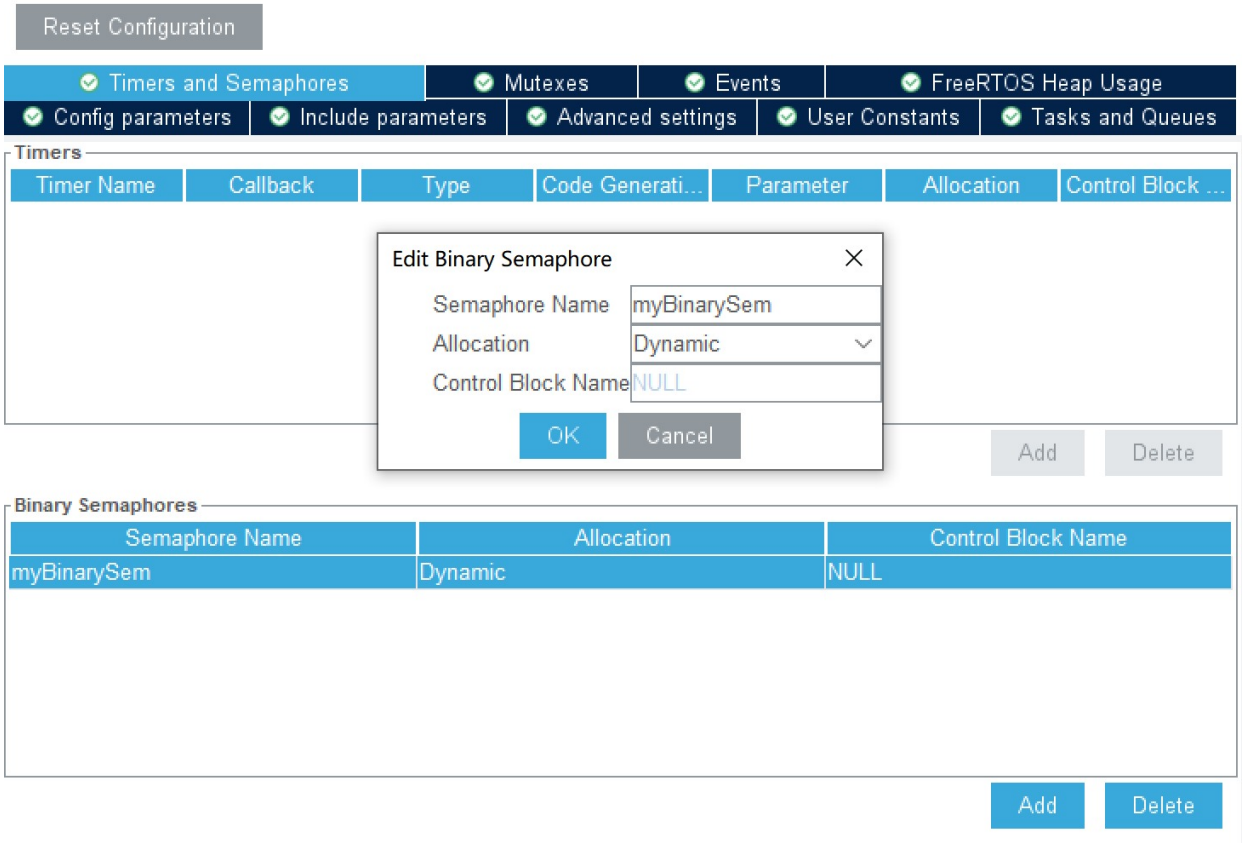
失败，返回 errQUEUE_FULL 。

实操

实验需求

创建一个二值信号量，按下 KEY1 则释放信号量，按下 KEY2 获取信号量。

cubeMX配置



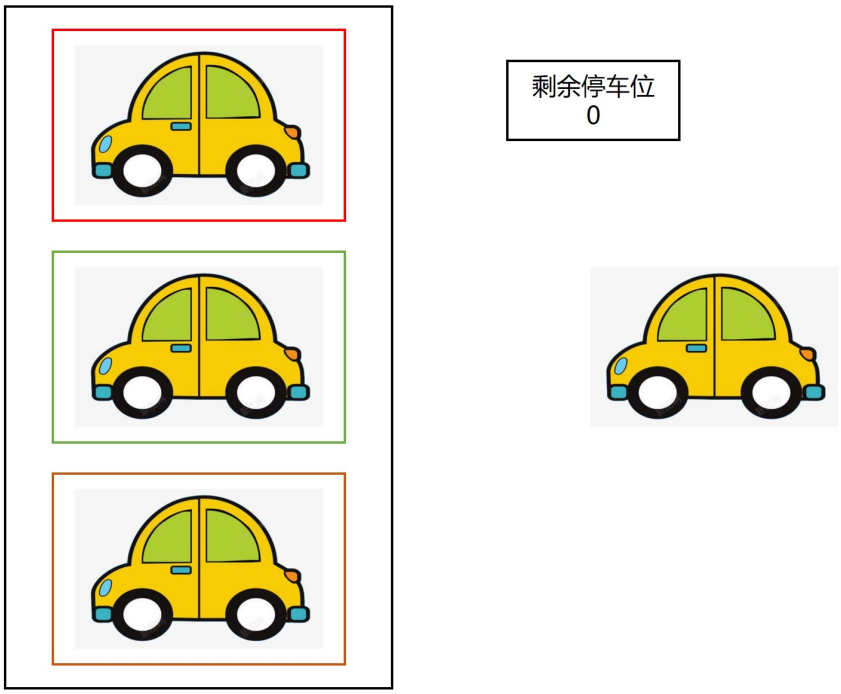
代码实现

```
myBinarySemHandle = xSemaphoreCreateBinary();

void StartTaskGive(void const * argument)
{
  /* USER CODE BEGIN StartTaskGive */
  /* Infinite loop */
  for(;;)
  {
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
        {
            osDelay(20);
            if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
            {
                if (xSemaphoreGive(myBinarySemHandle) == pdTRUE)
                    printf("二值信号量放入成功\r\n");
                else
                    printf("二值信号量放入失败\r\n");
            }
            while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET);
        }
    osDelay(10);
  }
  /* USER CODE END StartTaskGive */
}

void StartTaskTake(void const * argument)
{
  /* USER CODE BEGIN StartTaskTake */
  /* Infinite loop */
  for(;;)
  {
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
        {
            osDelay(20);
            if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
            {
                if (xSemaphoreTake(myBinarySemHandle, portMAX_DELAY ) == pdTRUE)
                    printf("二值信号量取出成功\r\n");
                else
                    printf("二值信号量取出失败\r\n");
            }
            while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET);
        }
    osDelay(10);
  }
  /* USER CODE END StartTaskTake */
}
```

七、计数型信号量

什么是计数型信号量?

计数型信号量相当于队列长度大于1 的队列，因此计数型信号量能够容纳多个资源，这在计数型信号量被创建的时候确定的。



计数型信号量相关 API 函数

函数	描述
xSemaphoreCreateCounting()	使用动态方法创建计数型信号量。
xSemaphoreCreateCountingStatic()	使用静态方法创建计数型信号量
uxSemaphoreGetCount()	获取信号量的计数值

计数型信号量的释放和获取与二值信号量完全相同！

```
SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount,
                                            UBaseType_t uxInitialCount);
```

参数：

uxMaxCount：可以达到的最大计数值
uxInitialCount：创建信号量时分配给信号量的计数值

返回值：

成功，返回对应计数型信号量的句柄；

失败，返回 NULL 。

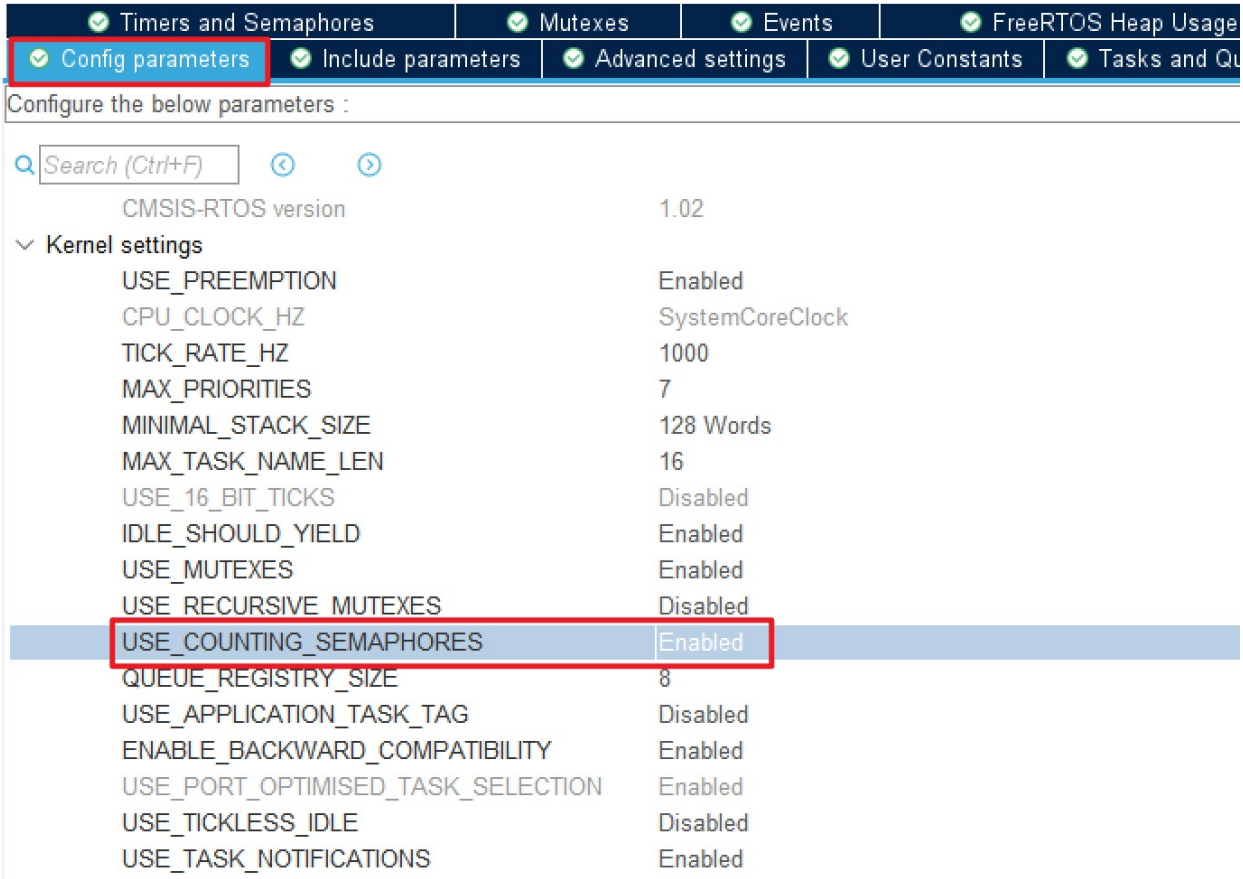
实操

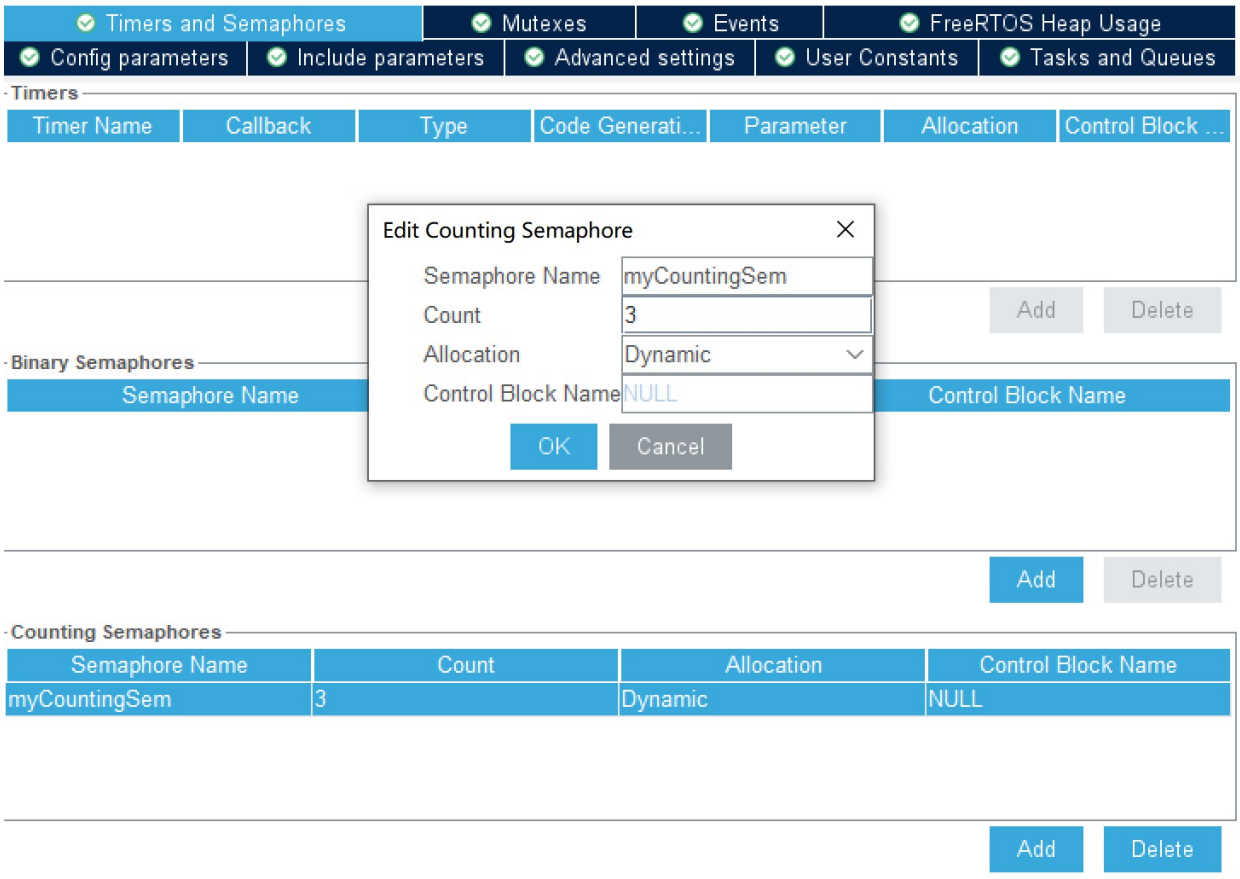
实验需求

创建一个计数型信号量，按下 KEY1 则释放信号量，按下 KEY2 获取信号量。

cubeMX配置

将 Config parameters 标签里的 USE_COUNTING_SEMAPHORES 设置为 Enabled 。





代码实现

```
myCountingSemHandle = xSemaphoreCreateCounting(3, 0);

void StartTaskGive(void const * argument)
{
  /* USER CODE BEGIN StartTaskGive */
  /* Infinite loop */
  for(;;)
  {
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
        {
            osDelay(20);
            if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
            {
                if (xSemaphoreGive(myCountingSemHandle) == pdTRUE)
                    printf("计数信号量放入成功\r\n");
                else
                    printf("计数信号量放入失败\r\n");
            }
            while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET);
        }
    osDelay(10);
  }
  /* USER CODE END StartTaskGive */
}
```

```
void StartTaskTake(void const * argument)
{
  /* USER CODE BEGIN StartTaskTake */
  /* Infinite loop */
  for(;;)
  {
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
        {
            osDelay(20);
            if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
            {
                if (xSemaphoreTake(myCountingSemHandle, 0 ) == pdTRUE)
                    printf("计数信号量获取成功\r\n");
                else
                    printf("计数信号量获取失败\r\n");
            }
            while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET);
        }
    osDelay(10);
  }
  /* USER CODE END StartTaskTake */
}
```

八、互斥量

什么是互斥量?

在多数情况下，互斥型信号量和二值型信号量非常相似，但是从功能上二值型信号量用于同步，而互斥型信号量用于资源保护。

互斥型信号量和二值型信号量还有一个最大的区别，互斥型信号量可以有效解决优先级反转现象。

什么是优先级翻转?

以上图为例，系统中有3个不同优先级的任务H/M/L，最高优先级任务H和最低优先级任务L通过信号量机制，共享资源。目前任务L占有资源，锁定了信号量，Task H运行后将被阻塞，直到Task L释放信号量后，Task H才能够退出阻塞状态继续运行。但是Task H在等待Task L释放信号量的过程中，中等优先级任务M抢占了任务L，从而延迟了信号量的释放时间，导致Task H阻塞了更长时间，这种现象称为优先级倒置或反转。

优先级继承：当一个互斥信号量正在被一个低优先级的任务持有时， 如果此时有个高优先级的任务也尝试获取这个互斥信号量，那么这个高优先级的任务就会被阻塞。**不过这个高优先级的任务会将低优先级任务的优先级提升到与自己相同的优先级。**

优先级继承并不能完全的消除优先级翻转的问题，它只是尽可能的降低优先级翻转带来的影响。

互斥量相关 API 函数

互斥信号量不能用于中断服务函数中！

函数	描述
xSemaphoreCreateMutex()	使用动态方法创建互斥信号量。
xSemaphoreCreateMutexStatic()	使用静态方法创建互斥信号量。

```
SemaphoreHandle_t xSemaphoreCreateMutex( void )
```

参数：

无

返回值：

成功，返回对应互斥量的句柄；

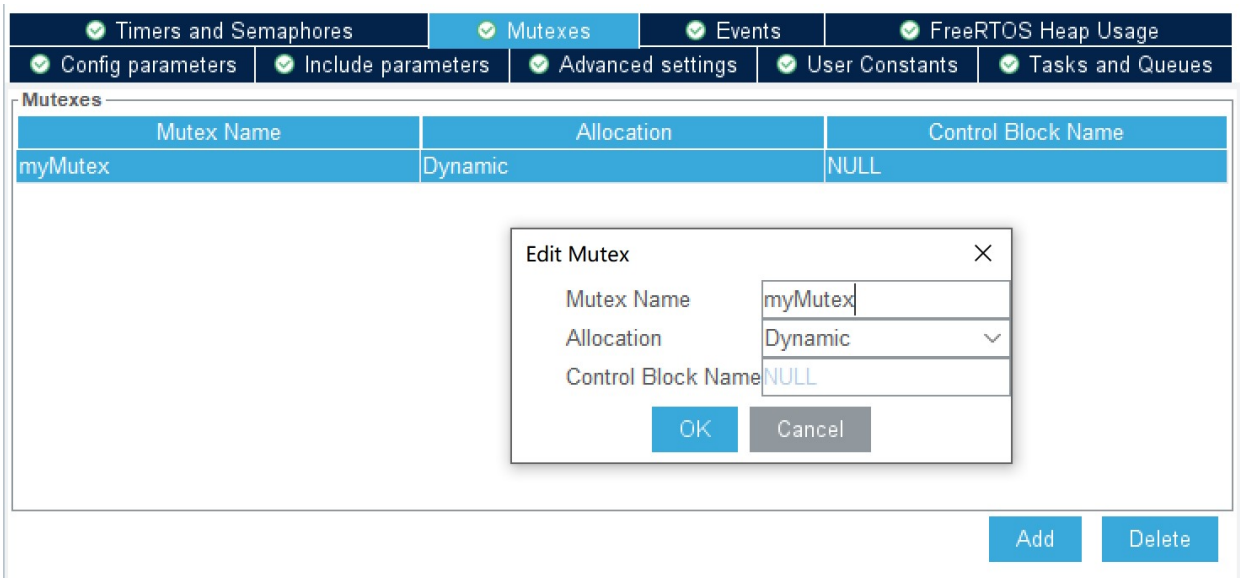
失败，返回 NULL 。

实操

实验需求

1. 演示优先级翻转
2. 使用互斥量优化优先级翻转问题

cubeMX配置



代码实现

```
void StartTaskH(void const * argument)
{
  /* USER CODE BEGIN StartTaskH */
  /* Infinite loop */
  for(;;)
  {
        xSemaphoreTake(myMutexHandle, portMAX_DELAY);
        printf("TaskH：我开始进入厕所，发功中~~~\r\n");
        HAL_Delay(1000);
    //osDelay(1000);
        printf("TaskH：我上完厕所了，真舒服~~~\r\n");
        xSemaphoreGive(myMutexHandle);
        osDelay(1000);
  }
  /* USER CODE END StartTaskH */
}

void StartTaskM(void const * argument)
{
  /* USER CODE BEGIN StartTaskM */
  /* Infinite loop */
  for(;;)
  {
        printf("TaskM：我就是为了占用CPU资源，带女朋友兜风去~~\r\n");
    osDelay(1000);
  }
  /* USER CODE END StartTaskM */
}

void StartTaskL(void const * argument)
{
  /* USER CODE BEGIN StartTaskL */
  /* Infinite loop */
```

```
  for(;;)
  {
        xSemaphoreTake(myMutexHandle, portMAX_DELAY);
        printf("TaskL：我开始进入厕所，发功中~~~\r\n");
        HAL_Delay(3000);
    //osDelay(3000);
        printf("TaskL：我上完厕所了，真舒服~~~\r\n");
        xSemaphoreGive(myMutexHandle);
        osDelay(1000);
  }
  /* USER CODE END StartTaskL */
}
```

九、事件标志组

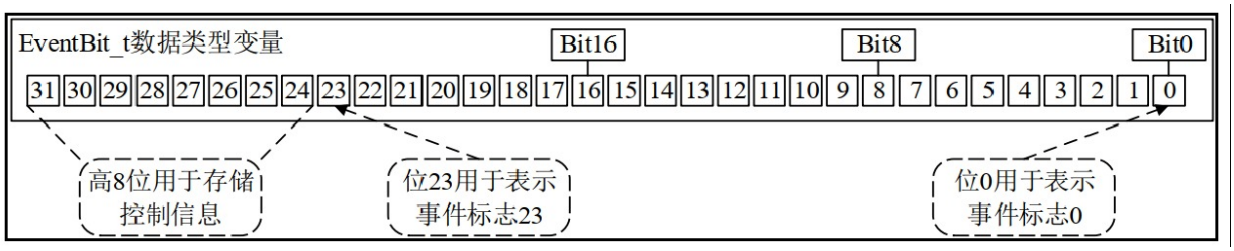
什么是事件标志组?

事件标志位：表明某个事件是否发生，联想：全局变量 flag。通常按位表示，每一个位表示一个事件（高8位不算）

事件标志组是一组事件标志位的集合， 可以简单的理解事件标志组，就是一个整数。

事件标志组本质是一个 16 位或 32 位无符号的数据类型 EventBits_t ， 由 configUSE_16_BIT_TICKS 决定。

虽然使用了 32 位无符号的数据类型变量来存储事件标志， 但其中的高8位用作存储事件标志组的控制信息，低 24 位用作存储事件标志 ， 所以说一个事件组最多可以存储 24 个事件标志!



事件标志组相关 API 函数

函数	描述
xEventGroupCreate()	使用动态方式创建事件标志组
xEventGroupCreateStatic()	使用静态方式创建事件标志组
xEventGroupClearBits()	清零事件标志位
xEventGroupClearBitsFromISR()	在中断中清零事件标志位
xEventGroupSetBits()	设置事件标志位
xEventGroupSetBitsFromISR()	在中断中设置事件标志位
xEventGroupWaitBits()	等待事件标志位

1. 创建事件标志组

```
EventGroupHandle_t xEventGroupCreate( void );
```

参数:

无

返回值:

成功，返回对应事件标志组的句柄;

失败，返回 NULL 。

2. 设置事件标志位

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup,
                                const EventBits_t uxBitsToSet );
```

参数:

xEventGroup：对应事件组句柄。
uxBitsToSet：指定要在事件组中设置的一个或多个位的按位值。

返回值:

设置之后事件组中的事件标志位值。

3. 清除事件标志位

```
EventBits_t xEventGroupClearBits(EventGroupHandle_t xEventGroup,
                                 const EventBits_t uxBitsToClear );
```

参数:

xEventGroup：对应事件组句柄。
uxBitsToClear：指定要在事件组中清除的一个或多个位的按位值。

返回值:

清零之前事件组中事件标志位的值。

4. 等待事件标志位

```
EventBits_t xEventGroupWaitBits(
                        const EventGroupHandle_t xEventGroup,
                        const EventBits_t uxBitsToWaitFor,
                        const BaseType_t xClearOnExit,
                        const BaseType_t xWaitForAllBits,
                        TickType_t xTicksToWait );
```

参数:

xEventGroup：对应的事件标志组句柄
uxBitsToWaitFor：指定事件组中要等待的一个或多个事件位的按位值
xClearOnExit：pdTRUE——清除对应事件位，pdFALSE——不清除
xWaitForAllBits：pdTRUE——所有等待事件位全为1（逻辑与），pdFALSE——等待的事件位有一个为1（逻辑或）
xTicksToWait：超时

返回值：

等待的事件标志位值：等待事件标志位成功，返回等待到的事件标志位
其他值：等待事件标志位失败，返回事件组中的事件标志位

实操

实验需求

创建一个事件标志组和两个任务（ task1 和 task2），task1 检测按键，如果检测到 KEY1 和 KEY2 都按过，则执行 task2 。

代码实现

```
EventGroupHandle_t  eventgroup_handle;

eventgroup_handle = xEventGroupCreate();

void StartTask1(void const * argument)
{
  /* USER CODE BEGIN StartTask1 */
  /* Infinite loop */
  for(;;)
  {
        // 等待 KEY1 按下
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
        {
            osDelay(20);
            if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
            {
                xEventGroupSetBits(eventgroup_handle, 0x01);
            }
            while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET);
        }

        // 等待 KEY2 按下
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
        {
            osDelay(20);
            if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
            {
                xEventGroupSetBits(eventgroup_handle, 0x02);
            }
            while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET);
```

```
        }
    osDelay(10);
  }
  /* USER CODE END StartTask1 */
}

void StartTask2(void const * argument)
{
  /* USER CODE BEGIN StartTask2 */
    EventBits_t event_bit = 0;
  /* Infinite loop */
  for(;;)
  {
        event_bit = xEventGroupWaitBits( eventgroup_handle, 0x01 | 0x02, pdTRUE,
pdTRUE, portMAX_DELAY );
        printf("返回值：%#x，请假成功，可以去大保健了！\r\n",event_bit);
    osDelay(1);
  }
  /* USER CODE END StartTask2 */
}
```

十、任务通知

什么是任务通知?

FreeRTOS 从版本 V8.2.0 开始提供任务通知这个功能，每个任务都有一个 32 位的通知值。按照 FreeRTOS 官方的说法，使用消息通知比通过二进制信号量方式解除阻塞任务快 45%， 并且更加省内存（无需创建队列）。

在大多数情况下，任务通知可以替代二值信号量、计数信号量、事件标志组，可以替代长度为 1 的队列（可以保存一个 32 位整数或指针值），并且任务通知速度更快、使用的RAM更少！

任务通知值的更新方式

FreeRTOS 提供以下几种方式发送通知给任务：

- 发送消息给任务，如果有通知未读， 不覆盖通知值
- 发送消息给任务，直接覆盖通知值
- 发送消息给任务，设置通知值的一个或者多个位
- 发送消息给任务，递增通知值

通过对以上方式的合理使用，可以在一定场合下替代原本的队列、信号量、事件标志组等。

任务通知的优势和劣势

任务通知的优势

1. 使用任务通知向任务发送事件或数据，比使用队列、事件标志组或信号量快得多。
2. 使用其他方法时都要先创建对应的结构体，使用任务通知时无需额外创建结构体。

任务通知的劣势

1. 只有任务可以等待通知，中断服务函数中不可以，因为中断没有 TCB 。
2. 通知只能一对一，因为通知必须指定任务。
3. 等待通知的任务可以被阻塞， 但是发送消息的任务，任何情况下都不会被阻塞等待。
4. 任务通知是通过更新任务通知值来发送数据的，任务结构体中只有一个任务通知值，只能保持一个数据。

任务通知相关 API 函数

1. 发送通知

函数	描述
xTaskNotify()	发送通知，带有通知值
xTaskNotifyAndQuery()	发送通知，带有通知值并且保留接收任务的原通知值
xTaskNotifyGive()	发送通知，不带通知值
xTaskNotifyFromISR()	在中断中发送任务通知
xTaskNotifyAndQueryFromISR()	在中断中发送任务通知
vTaskNotifyGiveFromISR()	在中断中发送任务通知

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify,
                        uint32_t ulValue,
                        eNotifyAction eAction );
```

参数:

xTaskToNotify：需要接收通知的任务句柄;

ulValue：用于更新接收任务通知值， 具体如何更新由形参 eAction 决定;

eAction：一个枚举，代表如何使用任务通知的值;

枚举值	描述
eNoAction	发送通知，但不更新值（参数ulValue未使用）
eSetBits	被通知任务的通知值按位或ulValue。（某些场景下可代替事件组，效率更高）

枚举值	描述
eIncrement	被通知任务的通知值增加1（参数ulValue未使用），相当于 `xTaskNotifyGive`
eSetValueWithOverWrite	被通知任务的通知值设置为 ulValue。（某些场景下可代替 `xQueueOverwrite`，效率更高）
eSetValueWithoutOverwrite	如果被通知的任务当前没有通知，则被通知的任务的通知值设为 ulValue。 如果被通知任务没有取走上一个通知，又接收到了一个通知，则这次通知值丢弃，在这种情况下视为调用失败并返回 `pdFALSE` （某些场景下可代替 `xQueueSend`，效率更高）

返回值：

如果被通知任务还没取走上一个通知，又接收了一个通知，则这次通知值未能更新并返回 pdFALSE， 而其他情况均返回pdPASS。

```
BaseType_t xTaskNotifyAndQuery( TaskHandle_t xTaskToNotify,
                                uint32_t ulValue,
                                eNotifyAction eAction,
                                uint32_t *pulPreviousNotifyValue );
```

参数：

xTaskToNotify：需要接收通知的任务句柄；
ulValue：用于更新接收任务通知值， 具体如何更新由形参 eAction 决定；
eAction：一个枚举，代表如何使用任务通知的值；
pulPreviousNotifyValue：对象任务的上一个任务通知值，如果为 NULL， 则不需要回传， 这个时候就等价于函数 xTaskNotify()。

返回值：

如果被通知任务还没取走上一个通知，又接收了一个通知，则这次通知值未能更新并返回 pdFALSE， 而其他情况均返回pdPASS。

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

参数：

xTaskToNotify：接收通知的任务句柄， 并让其自身的任务通知值加 1。

返回值：

总是返回 pdPASS。

2. 等待通知

等待通知API函数只能用在任务，不可应用于中断中！

函数	描述
ulTaskNotifyTake()	获取任务通知，可以设置在退出此函数的时候将任务通知值清零或者减一。当任务通知用作二值信号量或者计数信号量的时候，使用此函数来获取信号量。
xTaskNotifyWait()	获取任务通知，比 ulTaskNotifyTak()更为复杂，可获取通知值和清除通知值的指定位

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit,
                           TickType_t xTicksToWait );
```

参数:

xClearCountOnExit：指定在成功接收通知后，将通知值清零或减 1，pdTRUE：把通知值清零（二值信号量）；pdFALSE：把通知值减一（计数型信号量）；
xTicksToWait：阻塞等待任务通知值的最大时间；

返回值:

0：接收失败
非0：接收成功，返回任务通知的通知值

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,
                            uint32_t ulBitsToClearOnExit,
                            uint32_t *pulNotificationValue,
                            TickType_t xTicksToWait );
```

ulBitsToClearOnEntry：函数执行前清零任务通知值那些位 。
ulBitsToClearOnExit：表示在函数退出前，清零任务通知值那些位，在清 0 前，接收到的任务通知值会先被保存到形参*pulNotificationValue 中。
pulNotificationValue：用于保存接收到的任务通知值。 如果 不需要使用，则设置为 NULL 即可 。
xTicksToWait：等待消息通知的最大等待时间。

实操

1. 模拟二值信号量
2. 模拟计数型信号量
3. 模拟事件标志组
4. 模拟邮箱

十一、延时函数

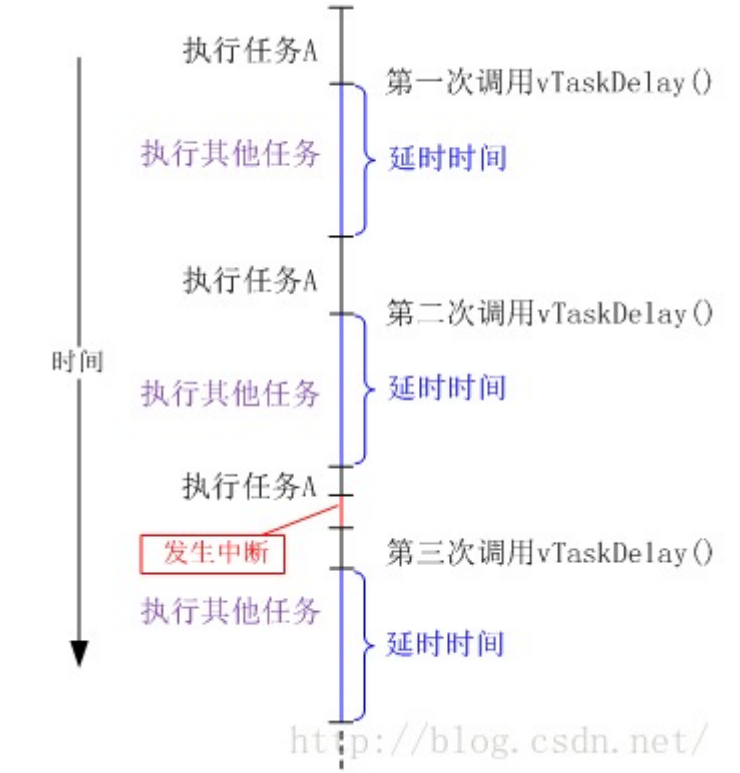
什么是延时函数?

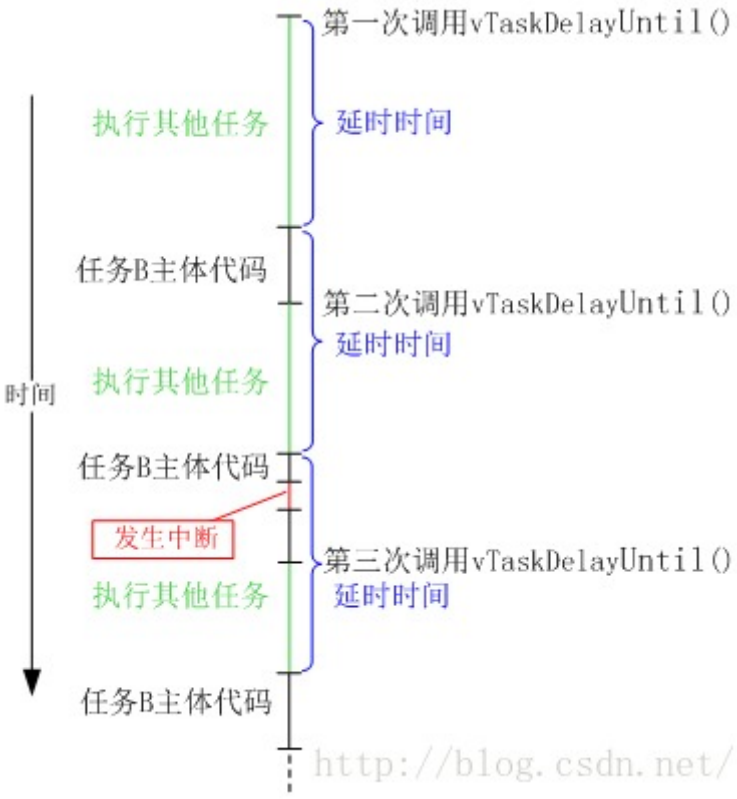
这还需要解释吗?

延时函数分类

相对延时：vTaskDelay

绝对延时：vTaskDelayUntil





vTaskDelay 与 HAL_Delay 的区别

```
void vTaskDelay( const TickType_t xTicksToDelay )
{
BaseType_t xAlreadyYielded = pdFALSE;

  /* A delay time of zero just forces a reschedule. */
  if( xTicksToDelay > ( TickType_t ) 0U )
  {
    configASSERT( uxSchedulerSuspended == 0 );
    vTaskSuspendAll();
    {
      traceTASK_DELAY();

      /* A task that is removed from the event list while the
      scheduler is suspended will not get placed in the ready
      list or removed from the blocked list until the scheduler
      is resumed.

      This task cannot be in an event list as it is the currently
      executing task. */
      prvAddCurrentTaskToDelayedList( xTicksToDelay, pdFALSE );
    }
    xAlreadyYielded = xTaskResumeAll();
  }
  else
  {
    mtCOVERAGE_TEST_MARKER();
  }

  /* Force a reschedule if xTaskResumeAll has not already done so, we may
  have put ourselves to sleep. */
  if( xAlreadyYielded == pdFALSE )
  {
    portYIELD_WITHIN_API();
  }
  else
  {
    mtCOVERAGE_TEST_MARKER();
  }
}
```

```
__weak void HAL_Delay(uint32_t Delay)
{
  uint32_t tickstart = HAL_GetTick();
  uint32_t wait = Delay;

  /* Add a freq to guarantee minimum wait */
  if (wait < HAL_MAX_DELAY)
  {
    wait += (uint32_t)(uwTickFreq);
  }

  while ((HAL_GetTick() - tickstart) < wait)
  {
  }
}
```

vTaskDelay 作用是让任务阻塞，任务阻塞后，RTOS系统调用其它处于就绪状态的优先级最高的任务来执行。

HAL_Delay 一直不停的调用获取系统时间的函数，直到指定的时间流逝然后退出，故其占用了全部CPU时间。

十二、软件定时器

什么是定时器?

简单可以理解为闹钟，到达指定一段时间后，就会响铃。

STM32 芯片自带硬件定时器，精度较高，达到定时时间后会触发中断，也可以生成 PWM 、输入捕获、输出比较，等等，功能强大，但是由于硬件的限制，个数有限。

软件定时器也可以实现定时功能，达到定时时间后可调用回调函数，可以在回调函数里处理信息。

软件定时器优缺点

优点:

1. 简单、成本低;
2. 只要内存足够，可创建多个;

缺点:

精度较低，容易受中断影响。在大多数情况下够用，但对于精度要求比较高的场合不建议使用。

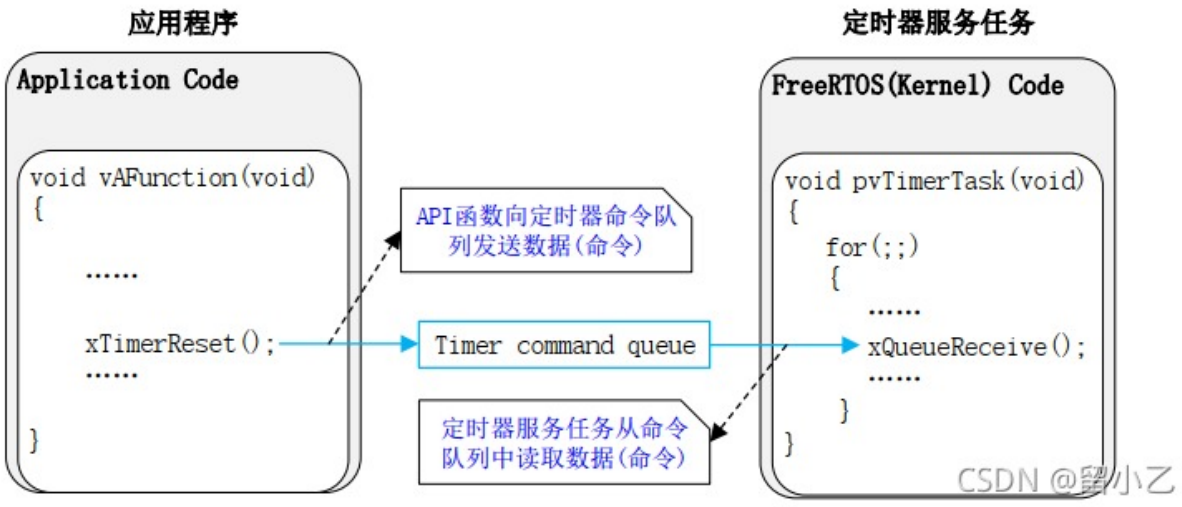
软件定时器原理

定时器是一个可选的、不属于 FreeRTOS 内核的功能，它是由定时器服务任务来提供的。

在调用函数 vTaskStartScheduler() 开启任务调度器的时候，会创建一个用于管理软件定时器的任务，这个任务就叫做软件定时器服务任务。

1. 负责软件定时器超时的逻辑判断
2. 调用超时软件定时器的超时回调函数
3. 处理软件定时器命令队列

FreeRTOS提供了很多定时器有关的API函数，这些API函数大多都使用FreeRTOS的队列发送命令给定时器服务任务。**这个队列叫做定时器命令队列。**定时器命令队列是提供给FreeRTOS的软件定时器使用的，**用户不能直接访问**!



软件定时器相关配置

软件定时器有一个定时器服务任务和定时器命令队列，这两个东西肯定是要配置的，相关的配置也是放到文件FreeRTOSConfig.h中的，涉及到的配置如下：

1、configUSE_TIMERS

如果要使用软件定时器的话宏configUSE_TIMERS一定要设置为1，当设置为1的话定时器服务任务就会在启动FreeRTOS调度器的时候自动创建。

2、configTIMER_TASK_PRIORITY

设置软件定时器服务任务的任务优先级，可以为0~(configMAX_PRIORITIES-1)。优先级一定要根据实际的应用要求来设置。如果定时器服务任务的优先级设置的高的话，定时器命令队列中的命令和定时器回调函数就会及时的得到处理。

3、configTIMER_QUEUE_LENGTH

此宏用来设置定时器命令队列的队列长度。

4、configTIMER_TASK_STACK_DEPTH

此宏用来设置定时器服务任务的任务堆栈大小。

单次定时器和周期定时器

单次定时器： 只超时一次，调用一次回调函数。可手动再开启定时器；

周期定时器： 多次超时，多次调用回调函数。

软件定时器相关 API 函数

函数	描述
xTimerCreate()	动态方式创建软件定时器
xTimerCreateStatic()	静态方式创建软件定时器
xTimerStart()	开启软件定时器定时
xTimerStop()	停止软件定时器定时
xTimerReset()	复位软件定时器定时
xTimerChangePeriod()	更改软件定时器的定时超时时间
xTimerStartFromISR()	在中断中开启软件定时器定时
xTimerStopFromISR()	在中断中停止软件定时器定时
xTimerResetFromISR()	在中断中复位软件定时器定时
xTimerChangePeriodFromISR()	在中断中更改定时超时时间

1. 创建软件定时器

```
TimerHandle_t xTimerCreate
                  ( const char * const pcTimerName,
                    const TickType_t xTimerPeriod,
                    const UBaseType_t uxAutoReload,
                    void * const pvTimerID,
                    TimerCallbackFunction_t pxCallbackFunction );
```

参数:

pcTimerName：软件定时器名称
xTimerPeriodInTicks：定时超时时间，单位：系统时钟节拍。宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为以 tick 为单位指定的时间。
uxAutoReload：定时器模式， pdTRUE：周期定时器， pdFALSE：单次定时器
pvTimerID：软件定时器 ID，用于多个软件定时器公用一个超时回调函数
pxCallbackFunction：软件定时器超时回调函数

返回值:

成功：定时器句柄

失败：NULL

2. 开启软件定时器

```
BaseType_t xTimerStart( TimerHandle_t xTimer,
                            TickType_t xBlockTime );
```

参数:

xTimer：待开启的软件定时器的句柄
xTickToWait：发送命令到软件定时器命令队列的最大等待时间

返回值:

pdPASS：开启成功
pdFAIL：开启失败

3. 停止软件定时器

```
BaseType_t xTimerStop( TimerHandle_t xTimer,
                           TickType_t xBlockTime );
```

参数与返回值同上。

4. 复位软件定时器

```
BaseType_t xTimerReset( TimerHandle_t xTimer,
                        TickType_t xBlockTime );
```

参数与返回值同上。

该功能将使软件定时器的重新开启定时，复位后的软件定时器以复位时的时刻作为开启时刻重新定时。

5. 更改软件定时器定时时间

```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer,
                               TickType_t xNewPeriod,
                               TickType_t xBlockTime );
```

xNewPeriod：新的定时超时时间，单位：系统时钟节拍。

其余参数与返回值同上。

实操

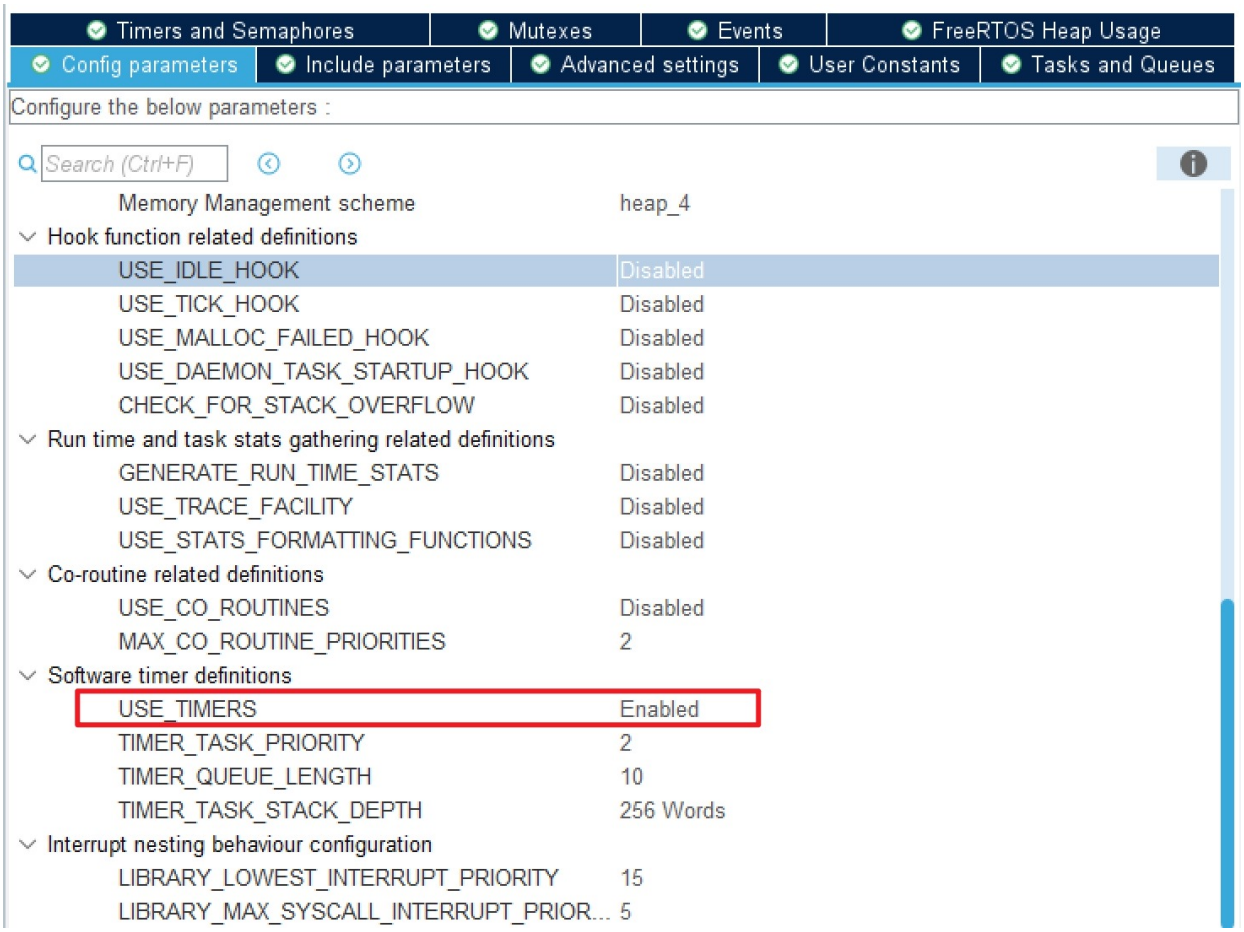
实验需求

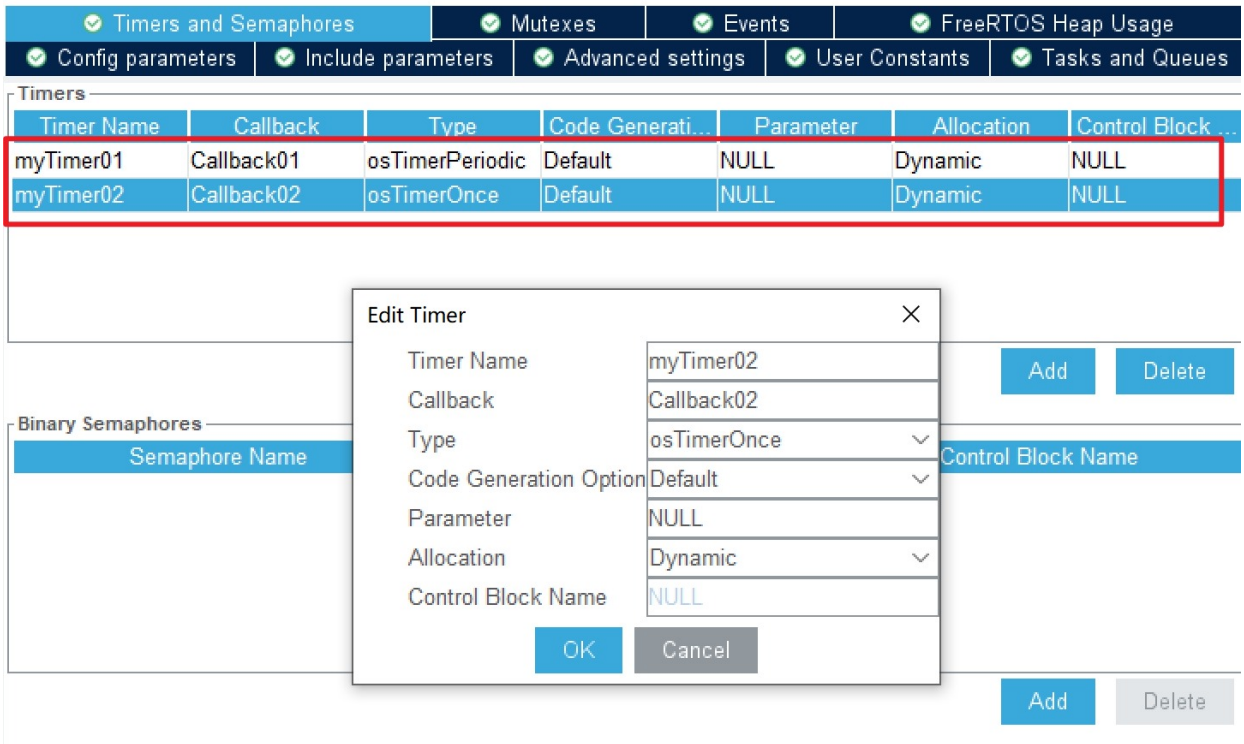
创建两个定时器：

定时器1，周期定时器，每1秒打印一次 liangxu shuai

定时器2，单次定时器，启动后 2 秒打印一次 laochen shuai

cubeMX配置


Timers and Semaphores
Mutexes
Events
FreeRTOS Heap Usage
Config parameters
Include parameters
Advanced settings
User Constants
Tasks and Queues
Configure the below parameters :
Search (Ctrl+F)
Memory Management scheme
heap_4
Hook function related definitions
USE_IDLE_HOOK
Disabled
USE_TICK_HOOK
Disabled
USE_MALLOC_FAILED_HOOK
Disabled
USE_DAEMON_TASK_STARTUP_HOOK
Disabled
CHECK_FOR_STACK_OVERFLOW
Disabled
Run time and task stats gathering related definitions
GENERATE_RUN_TIME_STATS
Disabled
USE_TRACE_FACILITY
Disabled
USE_STATS_FORMATTING_FUNCTIONS
Disabled
Co-routine related definitions
USE_CO_ROUTINES
Disabled
MAX_CO_ROUTINE_PRIORITIES
2
Software timer definitions
USE_TIMERS
Enabled
TIMER_TASK_PRIORITY
2
TIMER_QUEUE_LENGTH
10
TIMER_TASK_STACK_DEPTH
256 Words
Interrupt nesting behaviour configuration
LIBRARY_LOWEST_INTERRUPT_PRIORITY
15
LIBRARY_MAX_SYSCALL_INTERRUPT_PRIOR...
5


Timers and Semaphores
Mutexes
Events
FreeRTOS Heap Usage
Config parameters
Include parameters
Advanced settings
User Constants
Tasks and Queues
Timers
Timer Name
Callback
Type
Code Generati...
Parameter
Allocation
Control Block ...
myTimer01
Callback01
osTimerPeriodic
Default
NULL
Dynamic
NULL
myTimer02
Callback02
osTimerOnce
Default
NULL
Dynamic
NULL
Add
Delete
Binary Semaphores
Semaphore Name
Control Block Name
Add
Delete
Edit Timer
Timer Name
myTimer02
Callback
Callback02
Type
osTimerOnce
Code Generation Option
Default
Parameter
NULL
Allocation
Dynamic
Control Block Name
NULL
OK
Cancel

代码实现

```
void StartDefaultTask(void const * argument)
{
  /* USER CODE BEGIN StartDefaultTask */
    osTimerStart(myTimer01Handle, 1000);
    // xTimerChangePeriod(myTimer01Handle, pdMS_TO_TICKS(1000), 0);
    osTimerStart(myTimer02Handle, 2000);
  /* Infinite loop */
  for(;;)
  {
    osDelay(1);
  }
  /* USER CODE END StartDefaultTask */
}

void Callback01(void const * argument)
{
  /* USER CODE BEGIN Callback01 */
    printf("周期定时器：liangxu shuai\r\n");
  /* USER CODE END Callback01 */
}

void Callback02(void const * argument)
{
  /* USER CODE BEGIN Callback02 */
    printf("单次定时器：laochen shuai\r\n");
  /* USER CODE END Callback02 */
}
```

十三、中断管理

中断定义

请参考 51 及 STM32 中断相关课程。

中断优先级

任何中断的优先级都大于任务！

在我们的操作系统，中断同样是具有优先级的，并且我们也可以设置它的优先级，但是他的优先级并不是从 0~15 ，默认情况下它是从 5~15 ， 0~4 这 5 个中断优先级不是 FreeRTOS 控制的（5是取决于 configMAX_SYSCALL_INTERRUPT_PRIORITY）。

相关注意

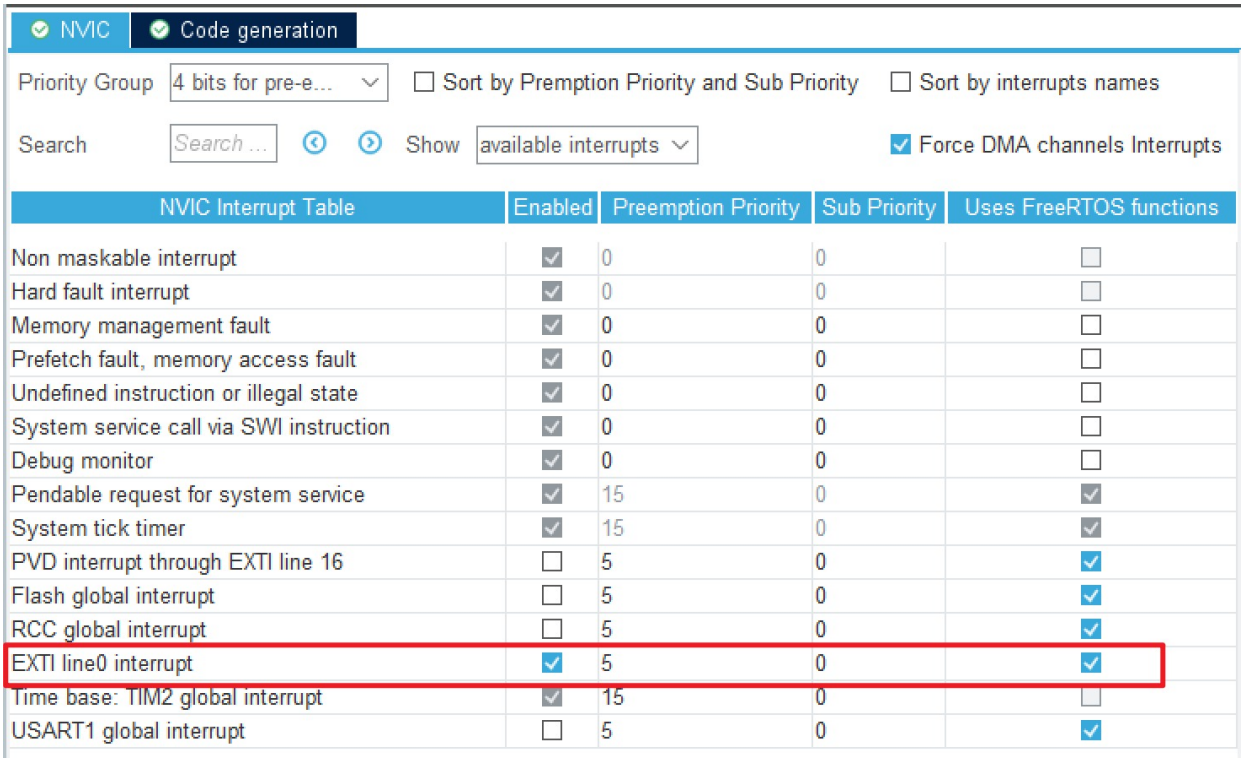
1. 在中断中必需使用中断相关的函数；
2. 中断服务函数运行时间越短越好。

实操

实验需求

创建一个队列及一个任务，按下按键 KEY1 触发中断，在中断服务函数里向队列里发送数据，任务则阻塞接收队列数据。

cubeMX配置



NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority	Uses FreeRTOS functions
Non maskable interrupt	☑	0	0	☐
Hard fault interrupt	☑	0	0	☐
Memory management fault	☑	0	0	☐
Prefetch fault, memory access fault	☑	0	0	☐
Undefined instruction or illegal state	☑	0	0	☐
System service call via SWI instruction	☑	0	0	☐
Debug monitor	☑	0	0	☐
Pendable request for system service	☑	15	0	☑
System tick timer	☑	15	0	☑
PVD interrupt through EXTI line 16	☐	5	0	☑
Flash global interrupt	☐	5	0	☑
RCC global interrupt	☐	5	0	☑
EXTI line0 interrupt	☑	5	0	☑
Time base: TIM2 global interrupt	☑	15	0	☐
USART1 global interrupt	☐	5	0	☑

代码实现

stm32f1xx_it.c

```
#include "cmsis_os.h"

extern osMessageQId myQueue01Handle;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    uint32_t snd = 1;
    xQueueSendFromISR(myQueue01Handle, &snd, NULL);
}
```

freertos.c

```
void StartDefaultTask(void const * argument)
{
  /* USER CODE BEGIN StartDefaultTask */
    uint32_t rev = 0;
  /* Infinite loop */
  for(;;)
  {
        if (xQueueReceive(myQueue01Handle, &rev, portMAX_DELAY) == pdTRUE)
            printf("rev = %d\r\n", rev);
    osDelay(1);
  }
  /* USER CODE END StartDefaultTask */
}
```

完结！撒花！

讲到这边，FreeRTOS 基本内容都讲完了，留给大家一个作业：使用 FreeRTOS 实现上官二号八个项目中的任意一个！