STM32Cube高效开发教程（高级篇）

第6章 互斥量

王维波
中国石油大学（华东）控制科学与工程学院

STM32Cube高效开发教程（高级篇）

作者：王维波，鄢志丹，王钊

人民邮电出版社

2022年2月出版

如果有读者需要本书课件的PPT版本用于备课，可以给作者发邮件免费获取，并可加入专门的教学和技术交流QQ群

邮箱：wangwb@upc. edu. cn

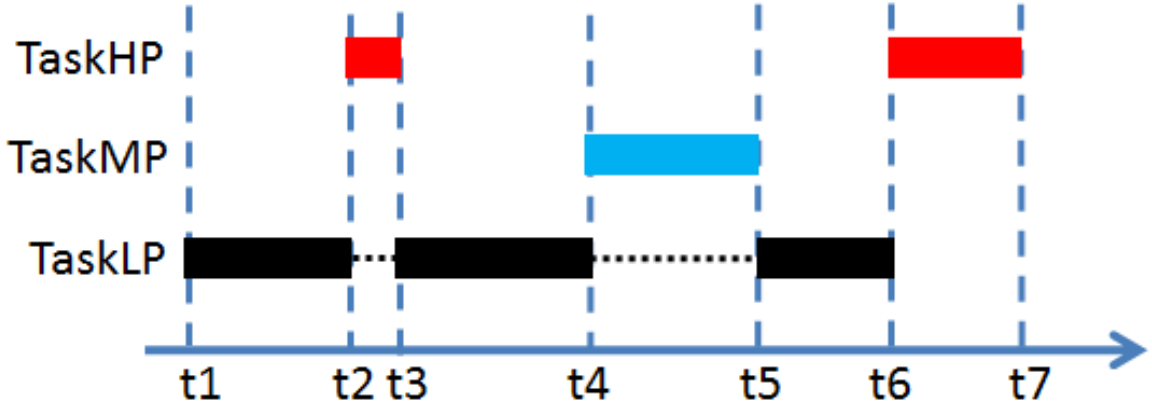
6.1 优先级翻转问题

6.2 互斥量工作原理

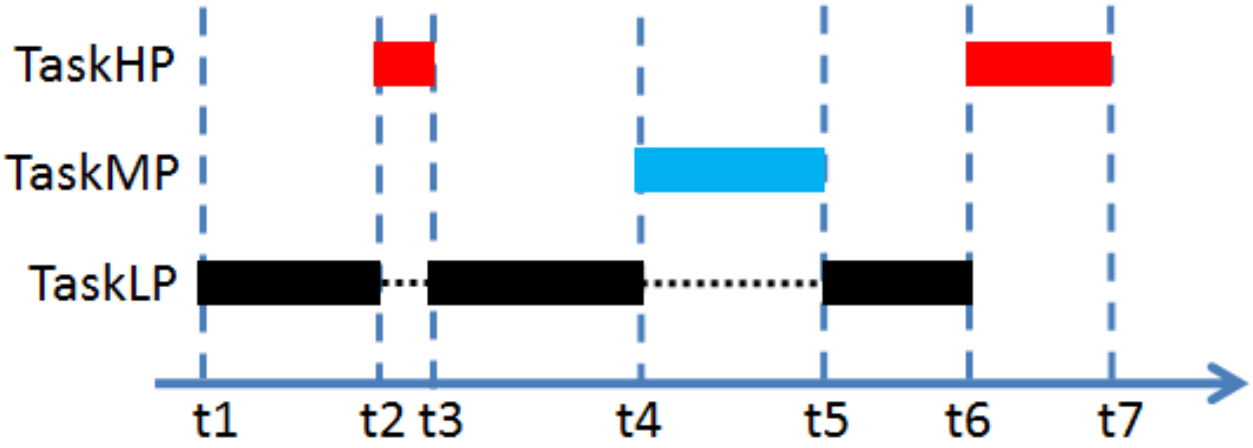
6.3 优先级翻转示例

6.4 互斥量使用示例

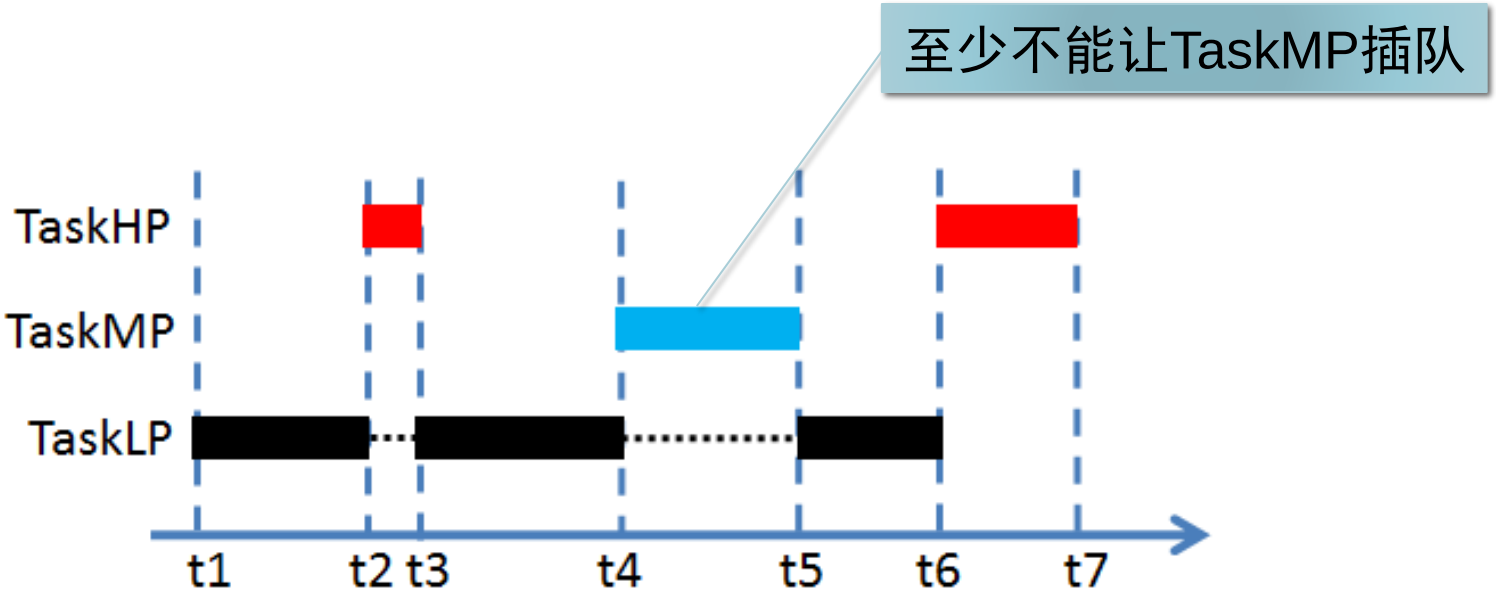
二值信号量也可以用于互斥型资源访问控制，但是容易出现优先级翻转（Priority Inversion）问题。



- 低优先级任务TaskLP在t1时刻开始处于运行状态，并且获取了一个二值信号量semp。
- 在时刻t2，高优先级任务TaskHP进入运行状态，它申请二值信号量semp，但是二值信号量被任务TaskLP占用，所以，TaskHP在时刻t3进入阻塞等待状态，TaskLP进入运行状态。



- 在时刻t4，中等优先级任务TaskMP抢占了TaskLP的CPU使用权，TaskMP不使用二值信号量，所以它一直运行到时刻t5才进入阻塞状态。
- 从t5时刻开始TaskLP又进入运行状态，直到t6时刻释放二值信号量semp，TaskHP才能进入运行状态。



高优先级的任务TaskHP需要等待低优先级的任务TaskLP释放二值信号量之后才可以运行，这也是期望的运行效果。

但是在t4时刻，虽然任务TaskMP的优先级比TaskHP低，但是它先于TaskHP抢占了CPU的使用权，这破坏了基于优先级抢占式执行的原则，对系统的实时性是有不利影响的。

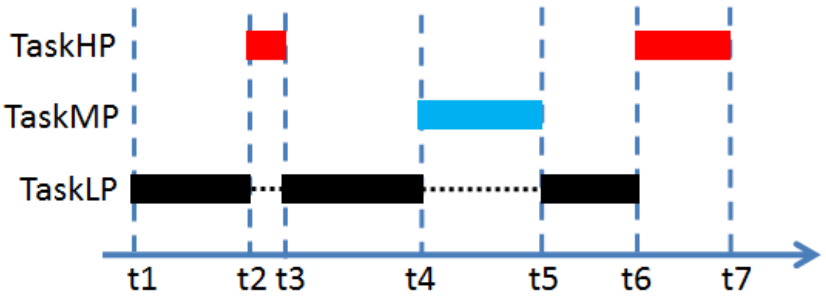
6.2 互斥量工作原理

6.2.1 优先级继承

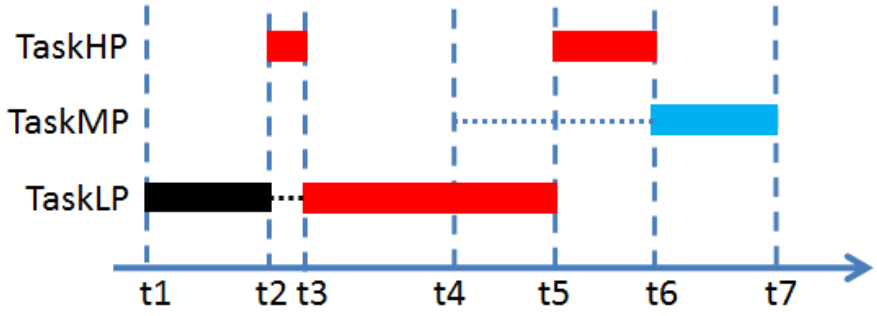
6.2.2 互斥量相关函数详解

6.2.1 优先级继承

在二值信号量的功能上引入了优先级继承（Priority Inheritance）机制，这就是互斥量（Mutex）

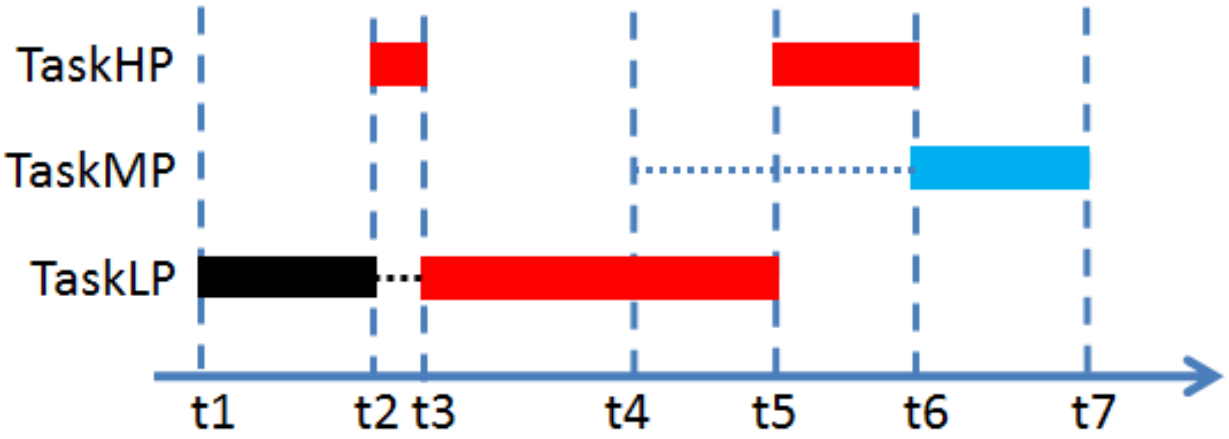


使用二值信号量

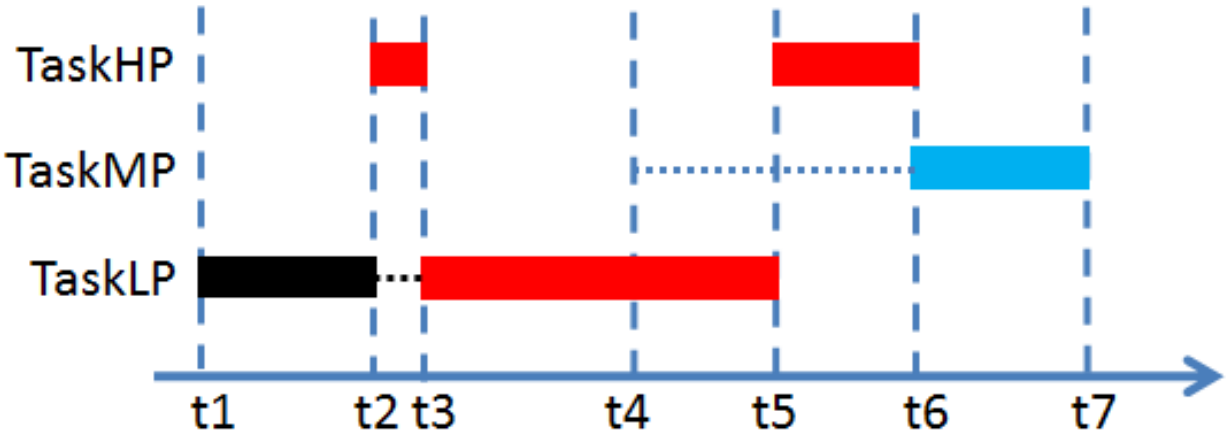


使用互斥量

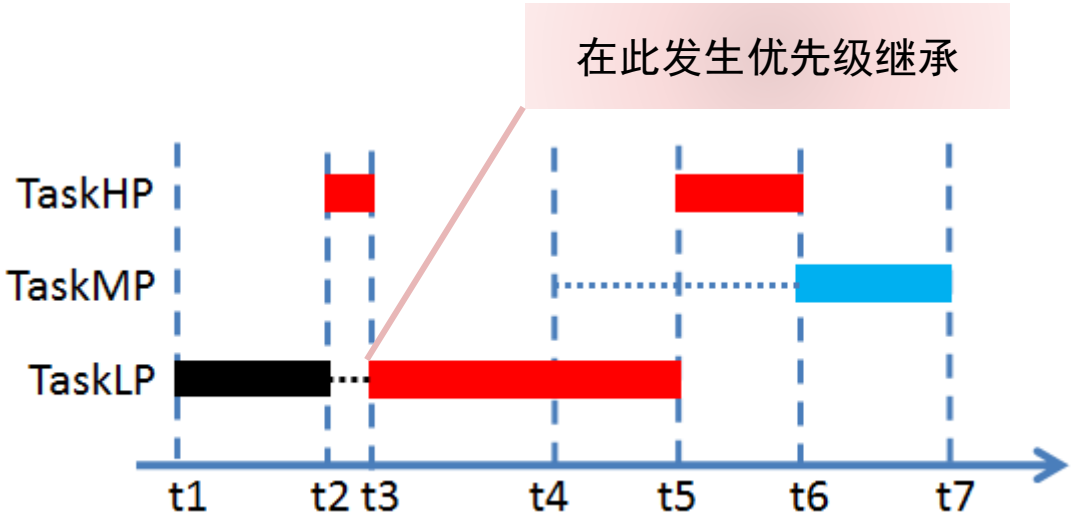
没有被低优先级的TaskMP插队



- t1时刻：TaskLP处于运行状态，并且获得了一个互斥量mutex
- t2时刻：TaskHP进入运行状态
- t3时刻： TaskHP申请互斥量mutex，但是互斥量被TaskLP占用，TaskHP进入阻塞等待状态，TaskLP进入运行状态。但是在t3时刻，RTOS将TaskLP的优先级临时提高到与TaskHP相同的级别，这就是优先级继承。



- t4时刻，TaskMP进入就绪状态，但是因为TaskLP的临时优先级高于TaskMP，所以TaskMP无法获得CPU的使用权。
- t5时刻：TaskLP释放互斥量，任务TaskHP立刻抢占CPU的使用权，并恢复TaskLP原来的优先级。
- t6时刻：TaskHP进入阻塞状态后，TaskMP才进入运行状态。



互斥量引入了优先级继承机制，可以临时提升占用互斥量的低优先级任务的优先级，与申请互斥量的高优先级任务的优先级相同，这样就避免了被中间优先级的任务抢占CPU的使用权，保证了高优先级任务运行的实时性。

使用互斥量可以减缓优先级翻转的影响，但是不能完全消除优先级翻转的问题。互斥量特别适用于互斥型资源访问控制。

6.2.2 互斥量相关函数详解

1. 创建互斥量

xSemaphoreCreateMutex()的定义是

```
#define xSemaphoreCreateMutex()
        xQueueCreateMutex( queueQUEUE_TYPE_MUTEX )
```

它调用了函数xQueueCreateMutex()，这个函数的原型定义是

```
QueueHandle_t  xQueueCreateMutex( const uint8_t  ucQueueType )
```

参数ucQueueType表示要创建的对象类型，

- 常量queueQUEUE_TYPE_MUTEX，用于创建互斥量
- 常量queueQUEUE_TYPE_RECURSIVE_MUTEX，用于创建递归互斥量

2. 获取和释放互斥量

获取互斥量使用函数xSemaphoreTake()，释放信号量使用函数xSemaphoreGive()，这两个函数的用法与获取和释放二值信号量一样。

注意，互斥量不能在ISR函数中使用，因为互斥量具有针对任务的优先级继承机制，而ISR函数不是任务。所以，函数xSemaphoreGiveFromISR()和xSemaphoreTakeFromISR()不能应用于互斥量。

6.3 优先级翻转示例

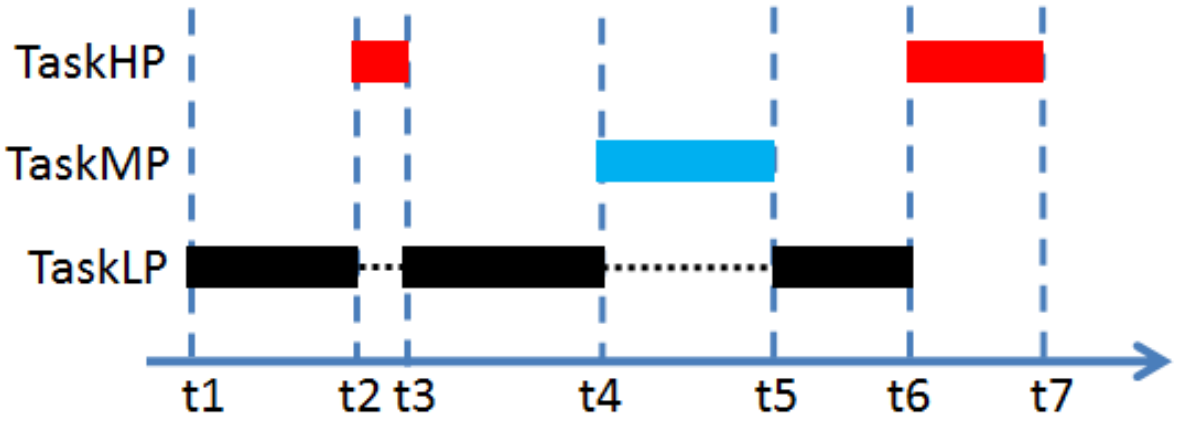
6.3.1 示例功能和CubeMX项目设置

6.3.2 程序功能实现

6.3.1 示例功能和CubeMX项目设置

演示使用二值信号量时出现的优先级翻转问题。

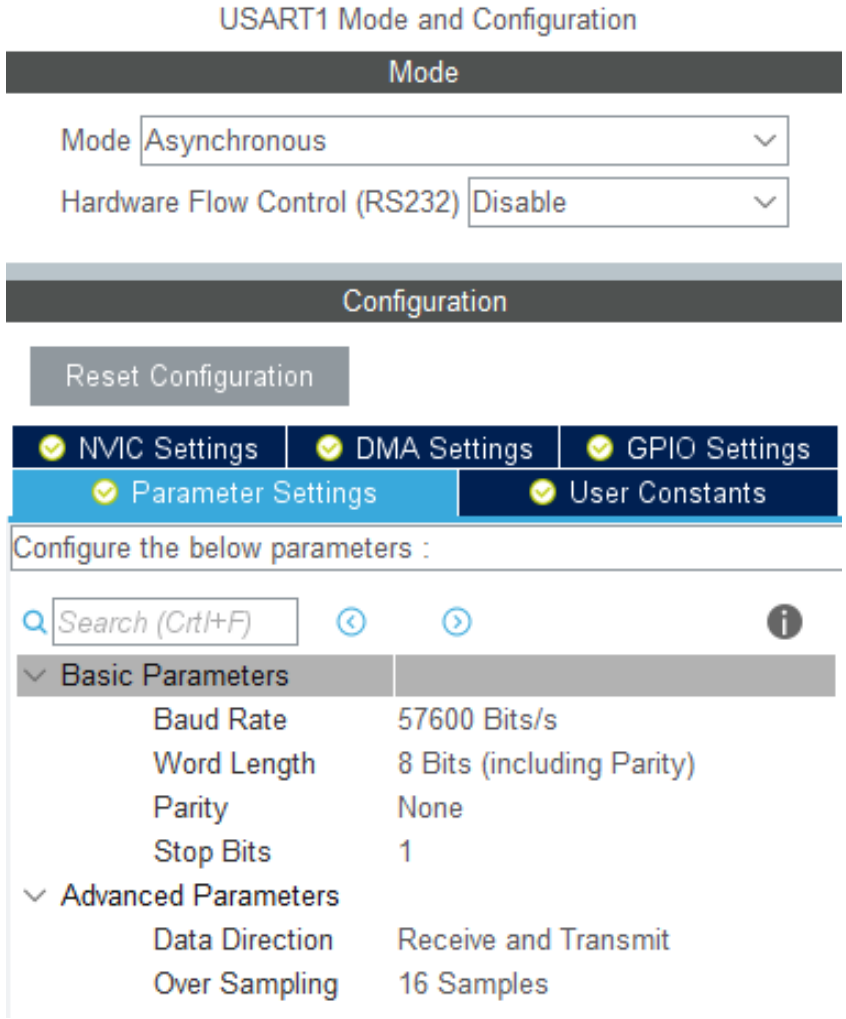
- 使用USART1向PC上传字符串信息，USART1作为一个互斥性访问的资源。
- 在FreeRTOS中创建3个不同优先级的任务，模拟图6-1 的工作过程，演示优先级翻转的问题。



优先级翻转现象

（1）USART1的设置

开发板上有一个USB到串口的转换芯片CH340，将STM32F407的USART1转换为USB，可以通过MicroUSB数据线与PC的USB接口直接相连



（2）FreeRTOS的设置

创建3个不同优先级的任务

Tasks

Task Name	Priority	Entry Function	Stack Size (Words)	Allocation
Task_High	osPriorityHigh	AppTask_High	128	Dynamic
Task_Middle	osPriorityNormal	AppTask_Middle	128	Dynamic
Task_Low	osPriorityLow	AppTask_Low	128	Dynamic

创建一个二值信号量，就命名为token，这是为了在下一个示例中直接创建一个同名的互斥量，减少代码的修改量。

Binary Semaphores

Semaphore Name	Allocation	Control Block Name
token	Dynamic	NULL

Add Delete

6.3.2 程序功能实现

1.主程序

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_FSMC_Init();
  MX_USART1_UART_Init();                    //USART1初始化

  /* USER CODE BEGIN 2 */
  TFTLCD_Init();                        //LCD 初始化
  LCD_ShowString(10, 10, (uint8_t *)"Demo6_1:Priority inversion");
  LCD_ShowString(10, 30, (uint8_t *)"Using binary semaphore");
  LCD_ShowString(10, 60, (uint8_t *)"1.Connect USART1 to PC via USB2");
  LCD_ShowString(10, 80, (uint8_t *)"2.View result on PC via COM");
  /* USER CODE END 2 */

  osKernelInitialize();
  MX_FREERTOS_Init();
  osKernelStart();
}
```

2. FreeRTOS初始化和对象的创建

3个任务的定义

```
/* 任务 Task_High相关定义 */
osThreadId_t Task_HighHandle;            //任务Task_High句柄变量
const osThreadAttr_t Task_High_attributes = {
                .name = "Task_High",
                .priority = (osPriority_t) osPriorityHigh,
                .stack_size = 128 * 4
};

/* 任务Task_Middle相关定义  */
osThreadId_t Task_MiddleHandle;          //任务Task_Middle句柄变量
const osThreadAttr_t Task_Middle_attributes = {
                .name = "Task_Middle",
                .priority = (osPriority_t) osPriorityNormal,
                .stack_size = 128 * 4
};

/* 任务Task_Low相关定义 */
osThreadId_t Task_LowHandle;             //任务Task_Low句柄变量
const osThreadAttr_t Task_Low_attributes = {
                .name = "Task_Low",
                .priority = (osPriority_t) osPriorityLow,
                .stack_size = 128 * 4
};
```

二值信号量的定义，函数MX_FREERTOS_Init()

```
/* 二值信号量token 相关定义*/
osSemaphoreId_t tokenHandle;                    /二值信号量token的句柄变量
const osSemaphoreAttr_t token_attributes = {
        .name = "token"
};

void MX_FREERTOS_Init(void)
{
    /* 创建二值信号量 token */
    tokenHandle = osSemaphoreNew(1, 1, &token_attributes);

    /* 创建任务 Task_High */
    Task_HighHandle = osThreadNew(AppTask_High, NULL, &Task_High_attributes);

    /*  创建任务Task_Middle */
    Task_MiddleHandle = osThreadNew(AppTask_Middle, NULL,
    &Task_Middle_attributes);

    /*  创建任务Task_Low */
    Task_LowHandle = osThreadNew(AppTask_Low, NULL, &Task_Low_attributes);
}
```

3. 三个任务的功能实现

```
void AppTask_Low(void *argument)     //任务Task_Low，低优先级
{
  /* USER CODE BEGIN AppTask_Low */
        uint8_tstr1[]="Task_Low take it\n";
        uint8_tstr2[]="Task_Low give it\n";
        for(;;)
        {
         if (xSemaphoreTake(tokenHandle, pdMS_TO_TICKS(200))==pdTRUE)  //获取
         {
            HAL_UART_Transmit(&huart1,str1,sizeof(str1),300);  //阻塞模式发送
            HAL_Delay(1000);  //连续延时，但是不释放信号量，期间会被Task_Middle抢占
            HAL_UART_Transmit(&huart1,str2,sizeof(str2),300);   //阻塞模式发送
            HAL_Delay(10);     //避免换行符 \n不能被正常输出
            xSemaphoreGive(tokenHandle);      //释放信号量
         }
         vTaskDelay(20);
        }
  /* USER CODE END AppTask_Low */
}
```

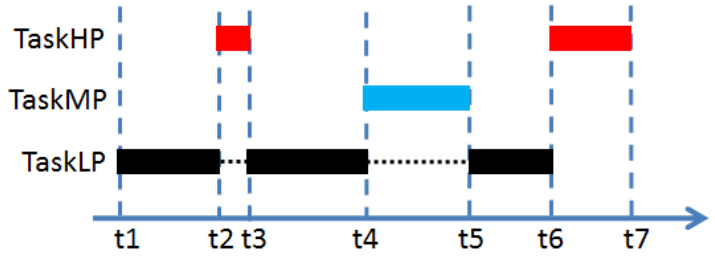
Task_Low占用信号量的时间长达1000ms

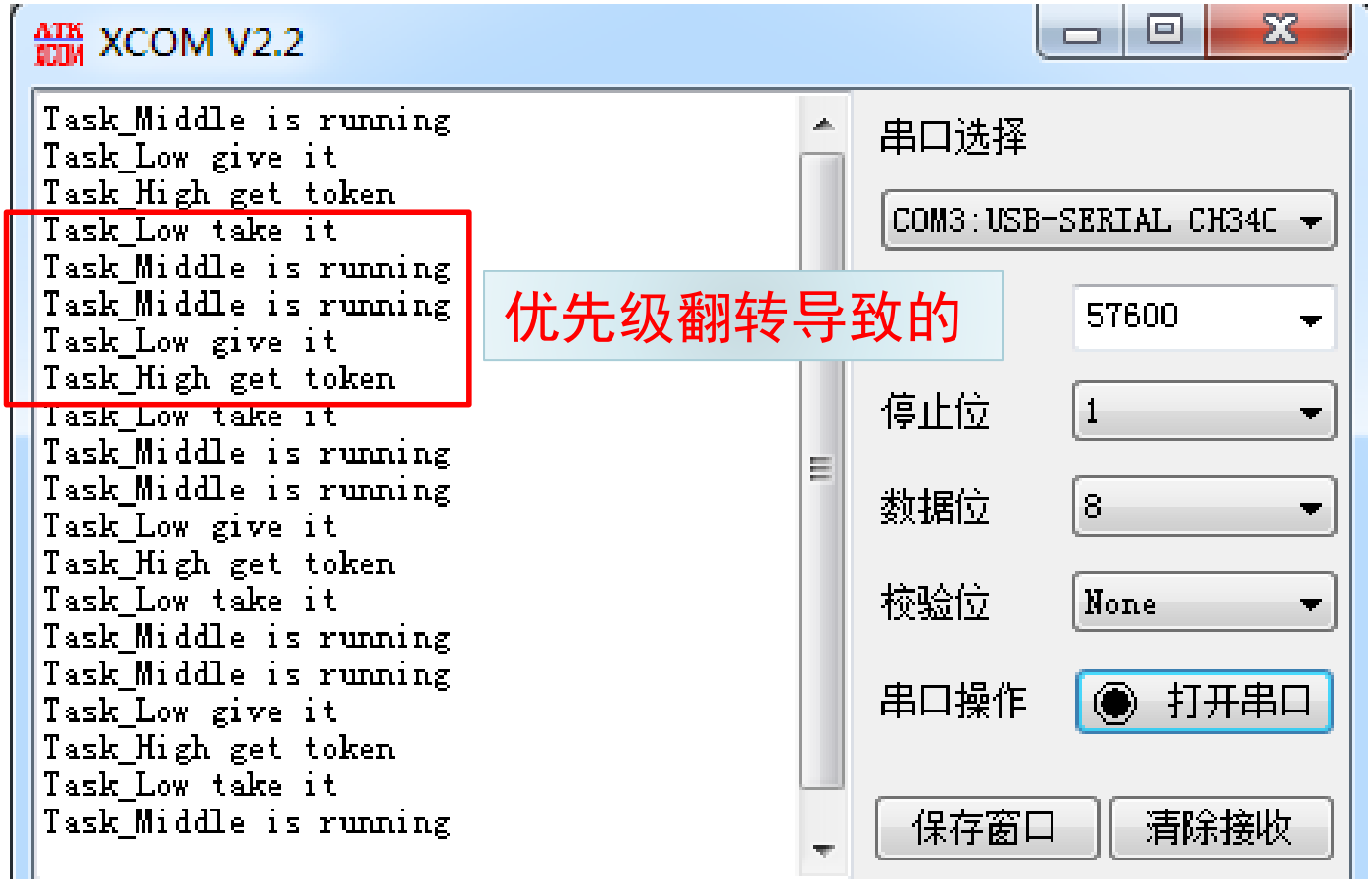
```
void AppTask_Middle(void *argument)    //任务Task_Middle，中优先级
{
  /* USER CODE BEGIN AppTask_Middle */
        uint8_t  strMid[]="Task_Middle is running\n";
        for(;;)
        {
         HAL_UART_Transmit(&huart1,strMid,sizeof(strMid),300); //阻塞模式发送
         HAL_Delay(10);     //避免换行符 \n不能被正常输出
         vTaskDelay(500);   //延时，进入阻塞状态
        }
  /* USER CODE END AppTask_Middle */
}
```

Task_Middle是一个正常的任务，每次循环有500ms时间处于阻塞状态，Task_Low循环内运行时间1000ms，所以，Task_Middle可以抢占Task_Low的运行。

```
void AppTask_High(void *argument)   //任务Task_High，高优先级
{
  /* USER CODE BEGIN AppTask_High */
        uint8_tstrHigh[]="Task_High get token\n";
        for(;;)
        {
         if (xSemaphoreTake(tokenHandle, portMAX_DELAY)==pdTRUE)  //获取信号量
         {
            HAL_UART_Transmit(&huart1,strHigh,sizeof(strHigh),300); //阻塞模式发送
            HAL_Delay(10);      //避免换行符 \n不能被正常输出
            xSemaphoreGive(tokenHandle);      //释放信号量
         }
         vTaskDelay(500);
        }
  /* USER CODE END AppTask_High */
}
```

Task_High要获取信号量，但是在Task_Low占用信号量时，Task_High只有等待。而Task_Low会被Task_Middle抢占，就出现了优先级翻转问题。





从运行结果可以看到明显的优先级翻转问题。优先级翻转问题导致高优先级任务不能及时运行，违背了抢占式任务调度系统的设计初衷。

6.4 互斥量使用示例

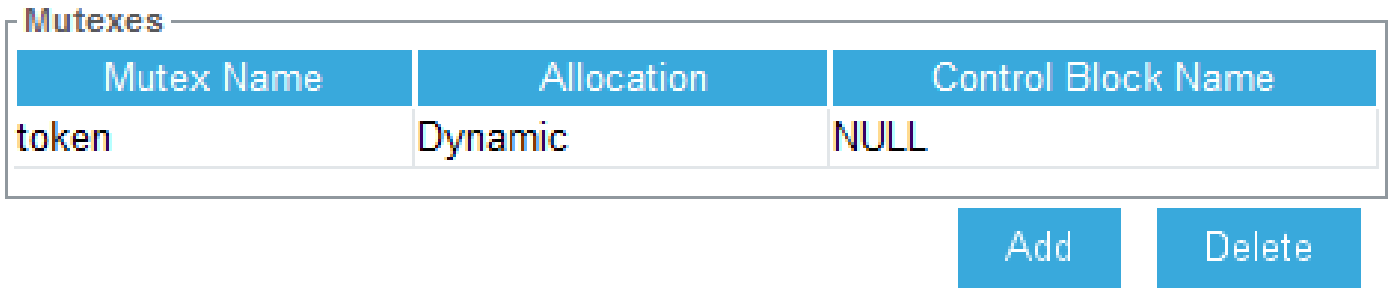
6.4.1 示例功能和CubeMX项目设置

6.4.2 程序功能实现

6.4.1 示例功能和CubeMX项目设置

本示例主要功能与前一示例相同，只是将其中的二值信号量换成了互斥量。

在FreeRTOS的设置中，删除原来的二值信号量token，在Mutex页面创建一个名称为token的互斥量，如图所示。



6.4.2 程序功能实现

1. 主程序

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_FSMC_Init();
  MX_USART1_UART_Init();

  /* USER CODE BEGIN 2 */
  TFTLCD_Init();                    //LCD 初始化
  LCD_ShowString(10, 10, (uint8_t *)"Demo6_2:Using Mutex");
  LCD_ShowString(10, 30, (uint8_t *)"To avoid priority inversion");
  LCD_ShowString(10, 60, (uint8_t *)"1.Connect USART1 to PC via USB2");
  LCD_ShowString(10, 80, (uint8_t *)"2.View result on PC via COM");
  /* USER CODE END 2 */

  osKernelInitialize();
  MX_FREERTOS_Init();
  osKernelStart();
}
```

2. FreeRTOS初始化和对象的创建

3个任务的定义代码与前一实例完全相同

```
/*   互斥量token相关定义   */
osMutexId_t  tokenHandle;      //互斥量token句柄变量
const osMutexAttr_t token_attributes = {  //互斥量token的属性
                .name = "token"
};

void MX_FREERTOS_Init(void)
{
        /* 创建互斥量 token */
        tokenHandle = osMutexNew(&token_attributes);

        /* 创建任务 Task_High */
        Task_HighHandle = osThreadNew(AppTask_High, NULL, &Task_High_attributes);

        /* 创建任务Task_Middle */
        Task_MiddleHandle = osThreadNew(AppTask_Middle, NULL,
        &Task_Middle_attributes);

        /* 创建任务 Task_Low */
        Task_LowHandle = osThreadNew(AppTask_Low, NULL, &Task_Low_attributes);
}
```

```
const osMutexAttr_t token_attributes = {   //互斥量token的属性
              .name = "token"
};
       tokenHandle = osMutexNew(&token_attributes);
```

结构体类型osMutexAttr_t用于描述互斥量的属性，定义如下

```
typedef struct {
  const char        *name;      //互斥量的名称字符串
  uint32_t           attr_bits;    //属性位
  void              *cb_mem;    //控制块的存储空间
  uint32_t           cb_size;      //控制块的大小，单位：字节
} osMutexAttr_t;
```

函数osMutexNew()创建互斥量，这是CMSIS RTOS标准接口函数。根据传递的互斥量属性，osMutexNew()自动判别创建互斥量或递归互斥量。在创建互斥量时，会根据属性设置，自动用动态分配内存或静态分配内存的函数。

3. 三个任务的功能实现

获取和释放互斥量的函数与操作信号量的一样

```
void AppTask_Low(void *argument)     //任务Task_Low，低优先级
{
  /* USER CODE BEGIN AppTask_Low */
        uint8_tstr1[]="Task_Low take it\n";
        uint8_tstr2[]="Task_Low give it\n";
        for(;;)
        {
         if (xSemaphoreTake(tokenHandle, pdMS_TO_TICKS(200))==pdTRUE)  //获取互斥量
         {
            HAL_UART_Transmit(&huart1,str1,sizeof(str1),300);  //阻塞模式发送
            HAL_Delay(1000);//延时，但是不释放信号量
            HAL_UART_Transmit(&huart1,str2,sizeof(str2),300);  //阻塞模式发送
            HAL_Delay(10);   //避免换行符 \n 不能被正常输出
            xSemaphoreGive(tokenHandle);
         }
         vTaskDelay(20);
        }
  /* USER CODE END AppTask_Low */
}
```

现在这个是互斥量

Task_Middle的代码没有任何变化

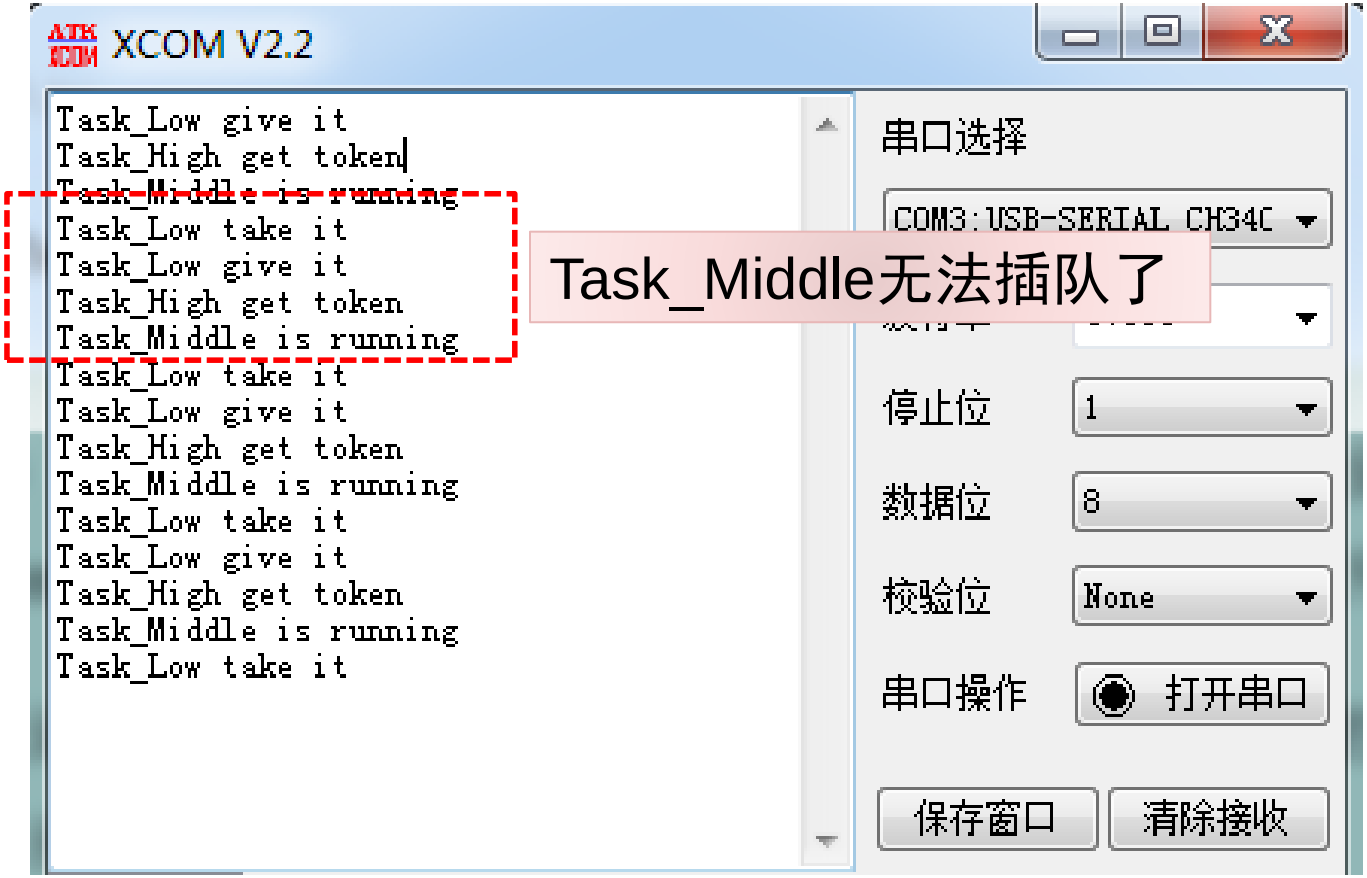
```
void AppTask_Middle(void *argument)     //任务Task_Middle，中优先级
{
  /* USER CODE BEGIN AppTask_Middle */
       uint8_t  strMid[]="Task_Middle is running\n";
       for(;;)
       {
        HAL_UART_Transmit(&huart1,strMid,sizeof(strMid),300);  //阻塞模式发送
        HAL_Delay(10);   //避免换行符 \n 不能被正常输出
        vTaskDelay(500);//延时, 进入阻塞状态
       }
  /* USER CODE END AppTask_Middle */
}
```

如果Task_Low没有被提升优先级，Task_Middle仍然可以抢占Task_Low的运行，只有在Task_Low的优先级被提升之后，Task_Middle才不能抢占Task_Low的运行。

Task_High的代码没有任何变化

```
void AppTask_High(void *argument)   //任务Task_High，高优先级
{
  /* USER CODE BEGIN AppTask_High */
        uint8_tstrHigh[]="Task_High get token\n";
        for(;;)
        {
         if (xSemaphoreTake(tokenHandle, portMAX_DELAY)==pdTRUE)  //获取互斥量
         {
            HAL_UART_Transmit(&huart1,strHigh,sizeof(strHigh),300); //阻塞模式发送
            HAL_Delay(10);   //避免换行符 \n 不能被正常输出
            xSemaphoreGive(tokenHandle);      //释放互斥量
         }
         vTaskDelay(500);
        }
  /* USER CODE END AppTask_High */
}
```

Task_High在申请互斥量时，如果互斥量被Task_Low占用，就会临时提升Task_Low的优先级。



互斥量并不能在所有的情况下彻底解决优先级翻转问题，但是至少可以减缓优先级翻转问题的出现。另外，因为互斥量使用了优先级继承机制，所以不能在ISR函数中使用互斥量。

练习任务

1. 看教材，练习本章的示例。