FreeRTOS实现

**作者: 向仔州**

FreeRTOS各个文件组成..........................................................................................................................................................................3

FreeRTOS移植到STM32F1……………………………………………………………………………………………………………………………....10

详解FreeRTOSConfig.h文件……………………………………………………………………………………………………………………..12

在systick中断中调用FreeRTOS的中断处理函数………………………………………………………………………………………...14

创建静态任务…………………………………………………………………………………………………………………………………………………....17

xTaskCreateStatic(....) //静态任务创建函数...............................................................................................................................17

动态创建任务...............................................................................................................................................................................................19

xTaskCreate(....) //动态创建任务………………………………………………………………………………………………………………….19

杀死任务.......................................................................................................................................................................................................21

vTaskDelete(......) //杀死任务函数………………………………………………………………………………………………………………..21

任务状态转换表，挂起，阻塞，就绪，运行的含义............................................................................................................................22

vTaskSuspend(.....) //挂起任务.....................................................................................................................................................26

vTaskResume(.....) //恢复任务......................................................................................................................................................26

xTaskGetTickCount( void ) //获取系统开始到现在执行了多少时间，以1毫秒为单位 ...............................................27

vTaskDelay(...) //阻塞延时函数…………………………………………………………………………………………………………………...27

vTaskDelayUntil(.....)//获取精确的延时时间(绝对延时函数)，延时时间到，任务马上执行，不受中断和优先级打断…..28

pdMS\_TO\_TICKS //将毫秒数转换成节拍数................................................................................................................................28

消息队列........................................................................................................................................................................................................28

xQueueCreate(......) //创建消息队列，返回句柄.........................................................................................................................30

xQueueSend(.....) //写队列，把数据写入队列尾部....................................................................................................................30

xQueueSendToFront(.....) //写队列，把数据写入队列头部，如果你的任务有紧急的事件就需要插队，就用这个函数...30

xQueueSendFromISR(......) //在中断中用非阻塞发送消息的函数，这个函数不会死等，没有数据也会执行结束........30

xQueueReceive(.....) //接收发送过来的消息...............................................................................................................................30

解决printf打印字符乱序，采用消息队列，修改printf…………………………………………………………………………………...31

信号量............................................................................................................................................................................................................34

二值信号量..........................................................................................................................................................................................34

QueueHandle\_t //信号量句柄定义...............................................................................................................................................34

xSemaphoreCreateBinary() //创建二值信号量变量，默认为0 ...........................................................................................35

xSemaphoreGive() //发送信号量，向信号量变量写1 ............................................................................................................35

xSemaphoreTake() //信号量接收函数 .......................................................................................................................................35

计数信号量..........................................................................................................................................................................................36

xSemaphoreCreateCounting(....) //创建计数信号量...............................................................................................................37

互斥量.............................................................................................................................................................................................................41

xSemaphoreCreateMutex() //创建互斥量 ………………………………………………………………………….……………………....43

xSemaphoreGive(.....) //发送互斥量………………………………………………………………………………………………………….....43

xSemaphoreTake(.....) //接收互斥量............................................................................................................................................43

事件使用，解决任务同步...........................................................................................................................................................................46

EventGroupHandle\_t //定义事件句柄........................................................................................................................................48

xEventGroupCreate( void ) //创建事件组.................................................................................................................................48

xEventGroupSetBits(....) //发送事件函数..................................................................................................................................48

xEventGroupWaitBits(...) //阻塞等待事件到来………………………………………………………………………………………….....48

多事件触发才能执行案例................................................................................................................................................................50

多事件也可以多个任务分开来发送...............................................................................................................................................51

软件定时器...................................................................................................................................................................................................52

TimerHandle\_t //定义定时器句柄............................................................................................................................................52

xTimerCreate(....) //创建定时器................................................................................................................................................52

xTimerStart(....) //启动定时器...................................................................................................................................................52

xTimerStartFromISR(.....) //中断中启动定时器........................................................................................................................52

xTimerStop(.....) //软件定时器停止函数.....................................................................................................................................52

任务通知......................................................................................................................................................................................................54

xTaskNotify(....) //发送带参数的通知给指定的任务................................................................................................................55

xTaskNotifyWait(.....) //阻塞程序，接收带参数的任务通知，放弃阻塞，继续向下执行................................................55

XTaskNotifyGive(.....) //发送二值信号量类型的任务通知......................................................................................................56

ulTaskNotifyTake(.....) //接收二值信号量类型的任务通知.....................................................................................................56

ulTaskNotifyTake(...) //将接收任务改成计数信号量任务通知模式，pdFALSE.................................................................57

内存管理.....................................................................................................................................................................................................58

FreeRTOS内存申请4种方式

heap\_1.c...........................................................................................................................................................................................58

heap\_2.c...........................................................................................................................................................................................60

Heap\_3.c malloc函数和free函数.............................................................................................................................................61

Heap\_4.c 使用最多的方式............................................................................................................................................................61

xPortGetFreeHeapSize( void ) //获取当前堆内存剩余大小.......................................................................................61

void \*pvPortMalloc( size\_t xWantedSize ) //内存分配函数，按字节分配............................................................61

void vPortFree( void \*pv ) //内存释放...........................................................................................................................62

中断管理，主要是FreeRTOS处理硬件中断的使用...........................................................................................................................63

taskENTER\_CRITICAL\_FROM\_ISR() //进入临界区.................................................................................................................63

taskEXIT\_CRITICAL\_FROM\_ISR( .... ) //退出临界区...............................................................................................................63

xQueueSendFromISR(….，….，….) //中断里发送消息队列函数使用，在消息队列章节有介绍………………………….63

CPU使用率统计.........................................................................................................................................................................................65

查询系统堆大小，用于检测程序是否长时间正常运行.........................................................................................68

size\_t xPortGetFreeHeapSize( void ) //系统剩余堆大小检测................................................................68

size\_t xPortGetMinimumEverFreeHeapSize( void )//查询堆最小空间，其实和剩余堆大小查询一样的功能....................................................................................................................................................................70

void vApplicationMallocFailedHook( void ); //堆申请失败回调函数……………………………………..70

查询每个任务栈使用大小，用于检测系统是否长时间正常运行…………………………………………………….....70

UBaseType\_t uxTaskGetStackHighWaterMark(.....) //查看当前任务栈还剩多少…………………......70

**FreeRTOS各个文件组成**

portmacro.h 文件

#define portCHAR char

#define portFLOAT float

#define portDOUBLE double

#define portLONG long

#define portSHORT short

#define portSTACK\_TYPE uint32\_t

#define portBASE\_TYPE long

typedef portSTACK\_TYPE StackType\_t;

typedef long BaseType\_t;

typedef unsigned long UBaseType\_t;

#if( configUSE\_16\_BIT\_TICKS == 1 )

typedef uint16\_t TickType\_t;

#define portMAX\_DELAY ( TickType\_t ) 0xffff

#else

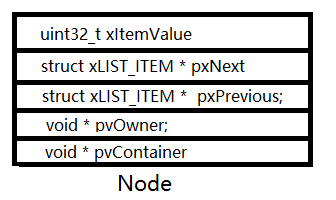
typedef uint32\_t TickType\_t;

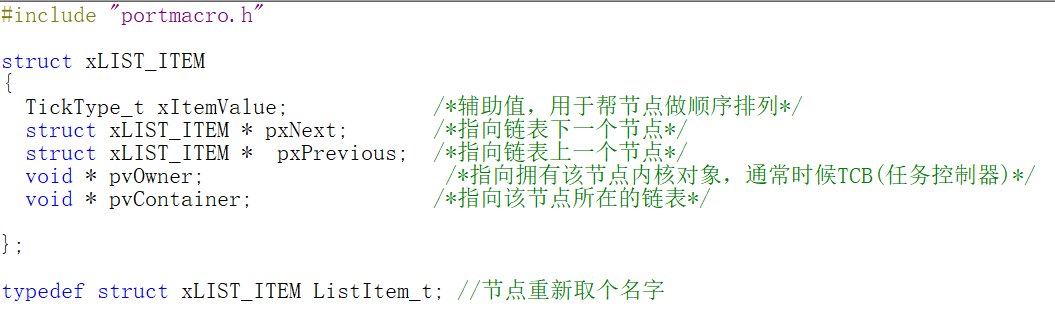
#define portMAX\_DELAY ( TickType\_t ) 0xffffffffUL

#endif

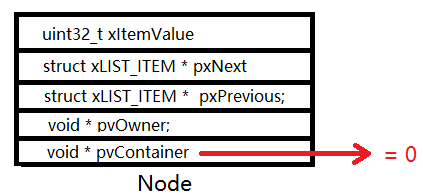
需要用到的变量类型重新定义，以符合MCU型号。

list.h

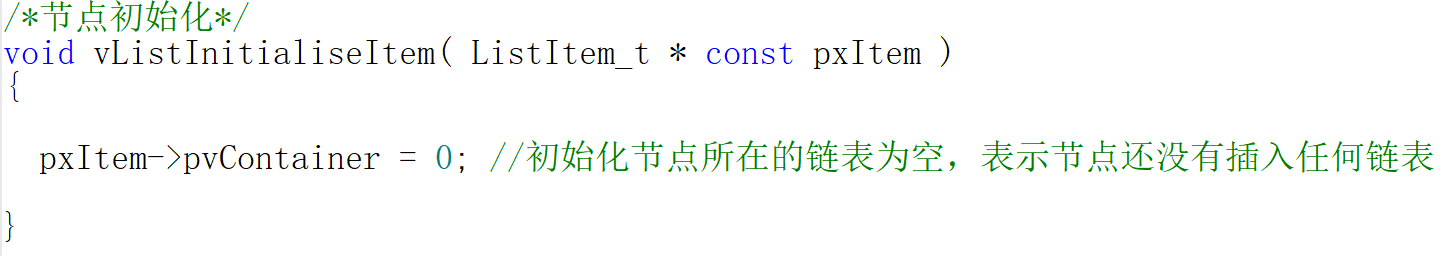




一个节点是什么样的

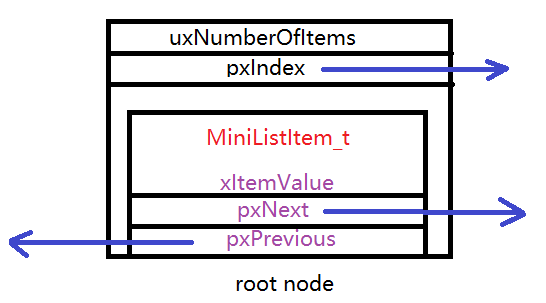


list.c

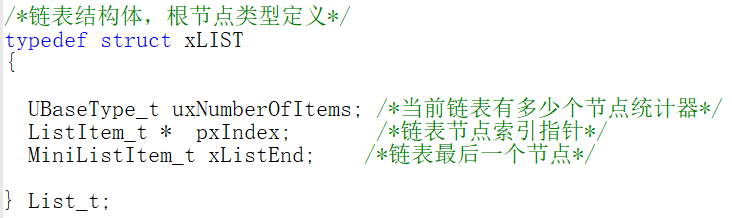




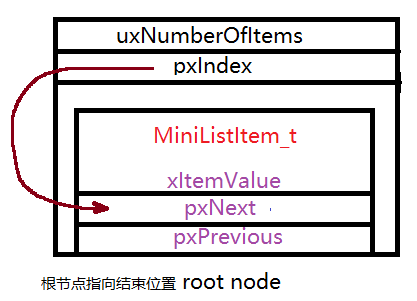
根节点定义



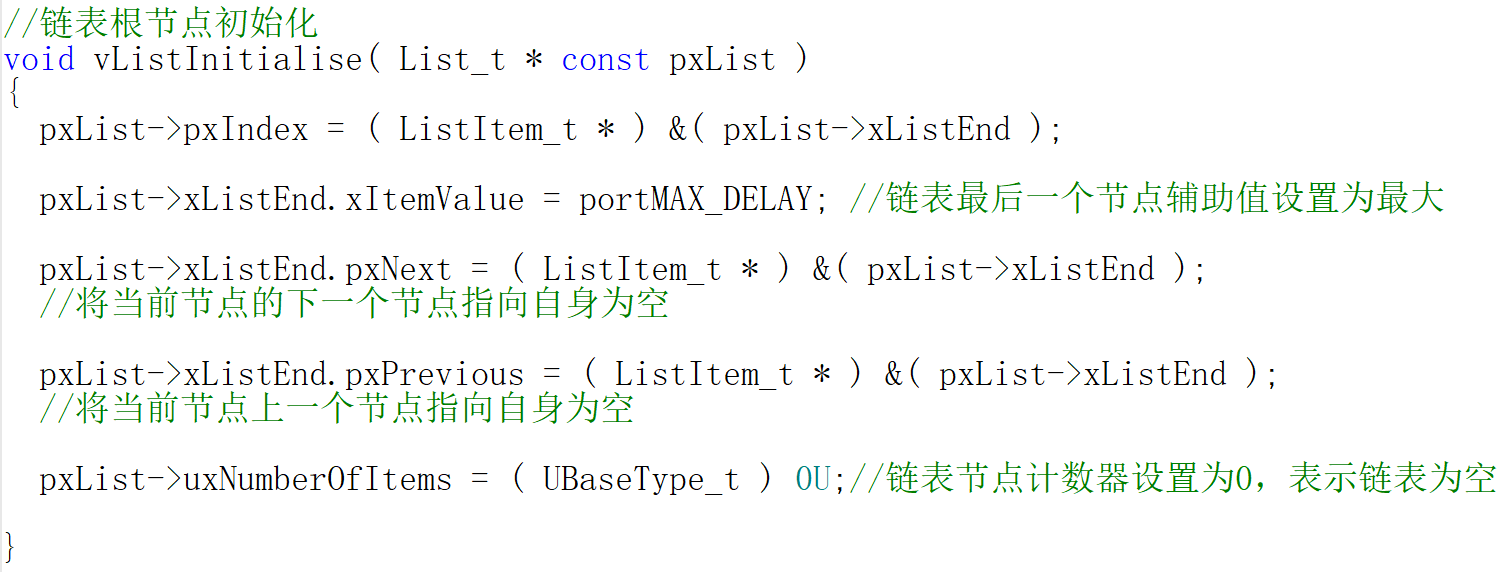
list.h

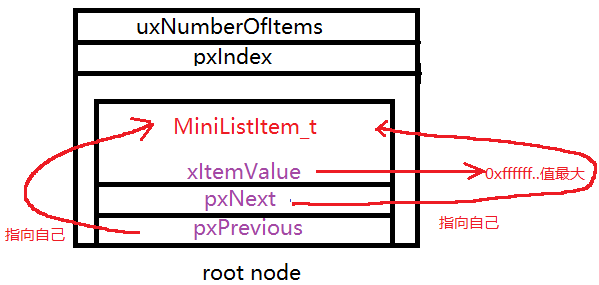


链表根节点初始化

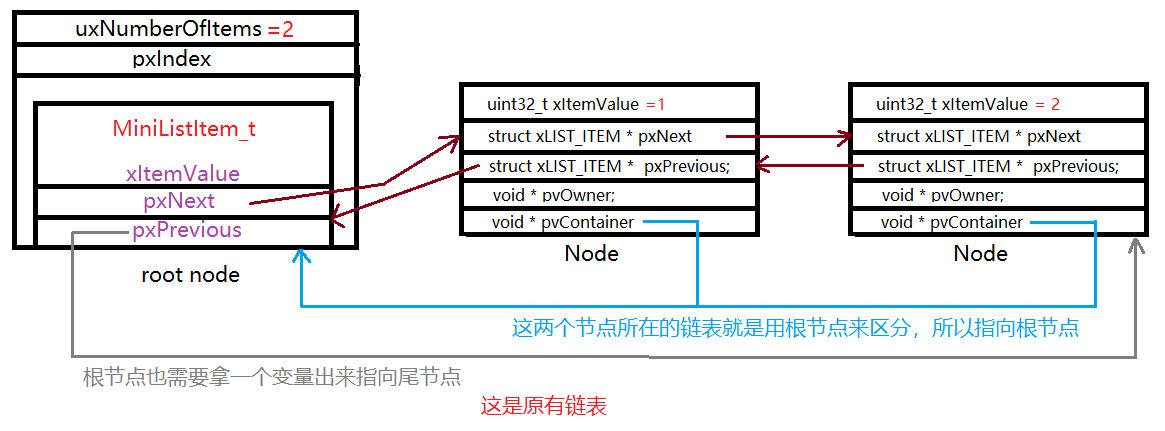


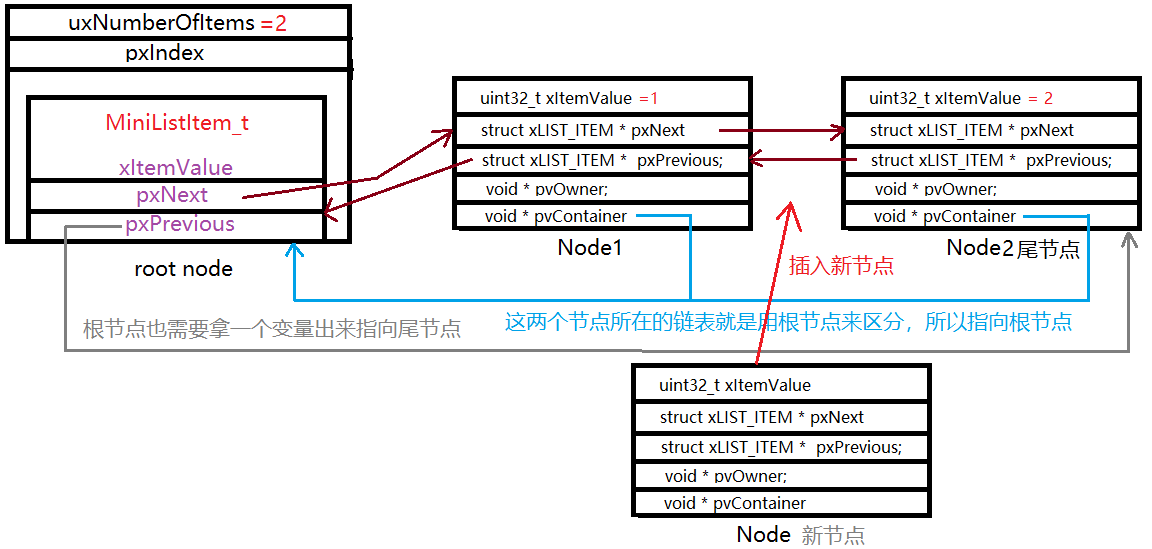
list.c



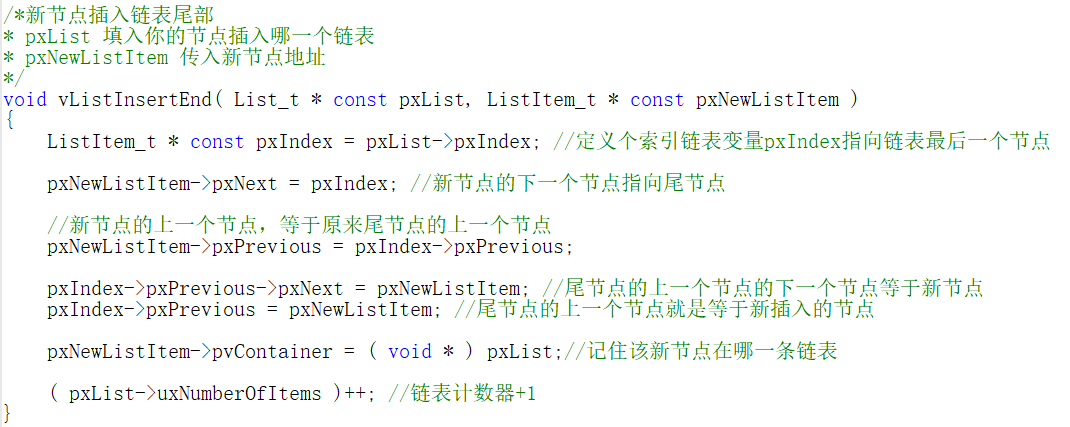


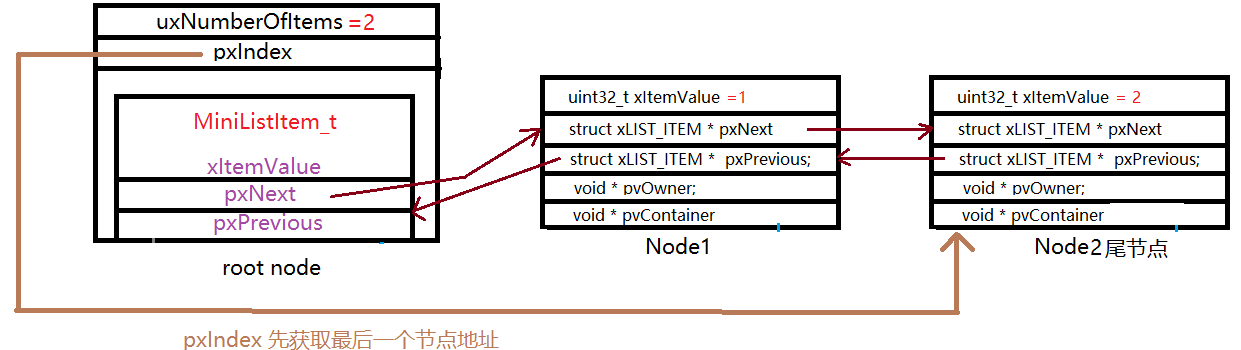
节点插入链表

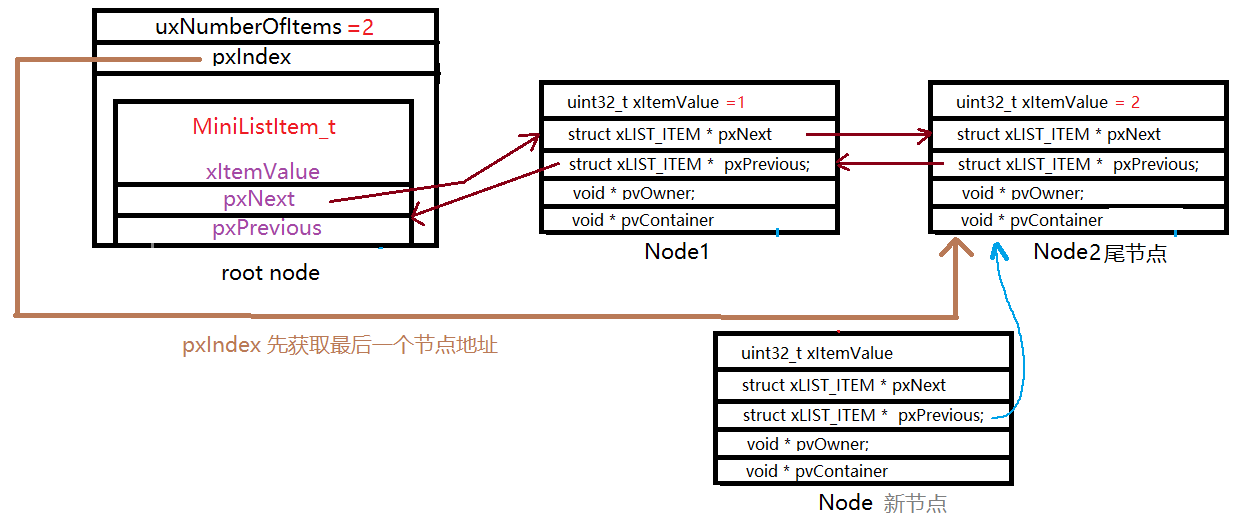


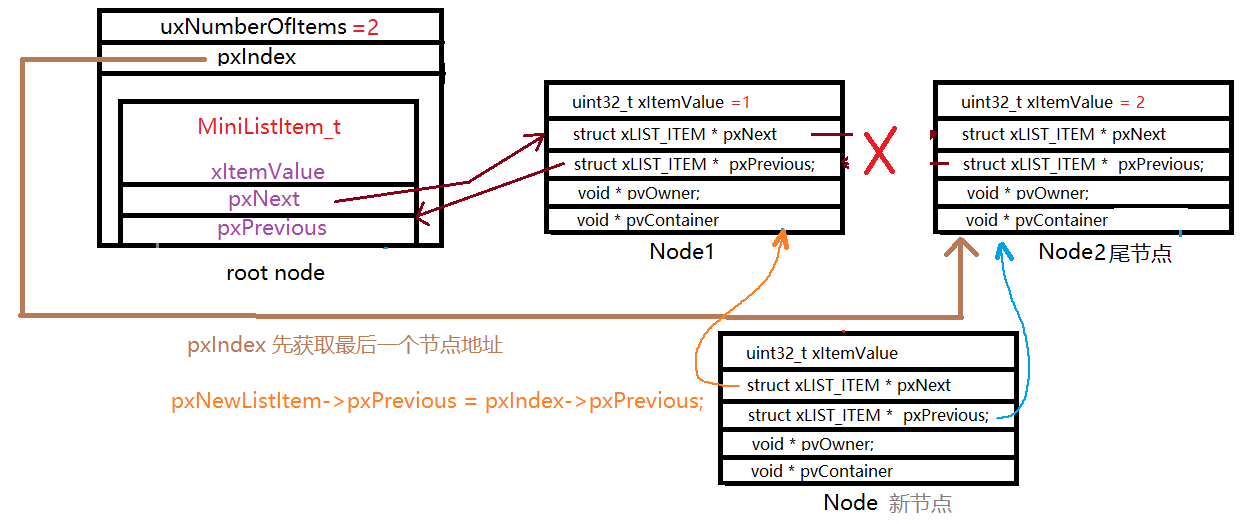


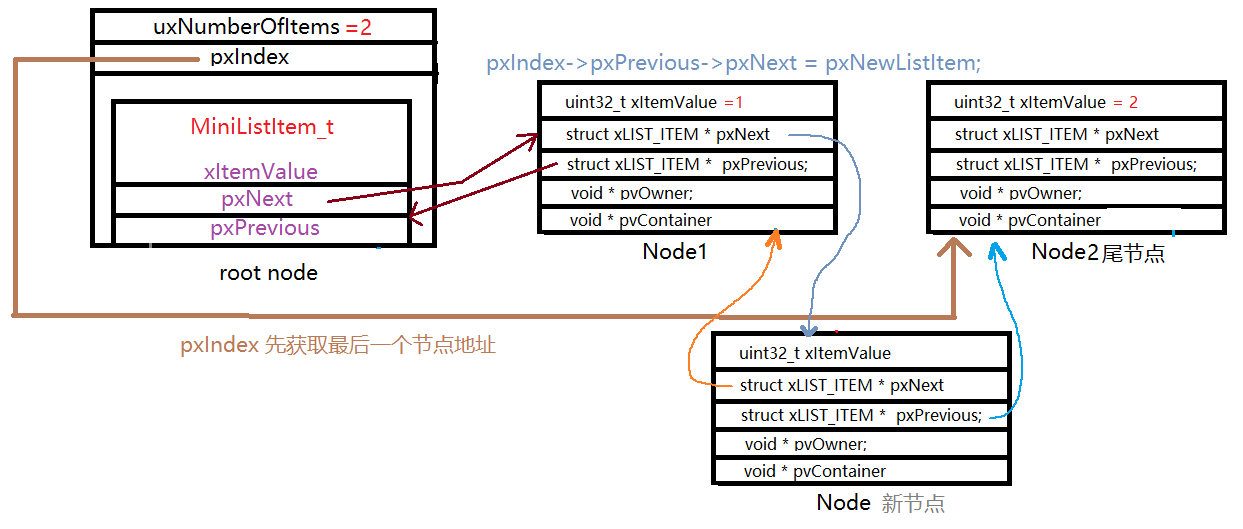
list.c



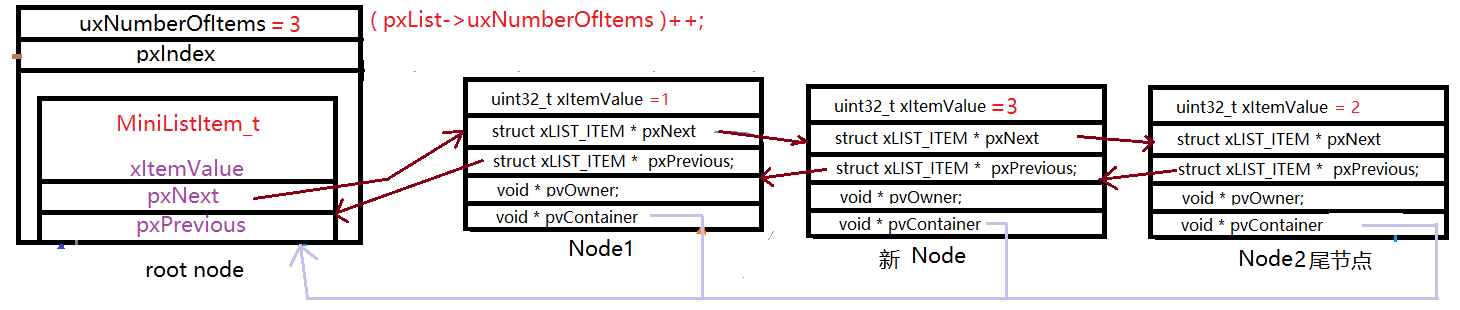






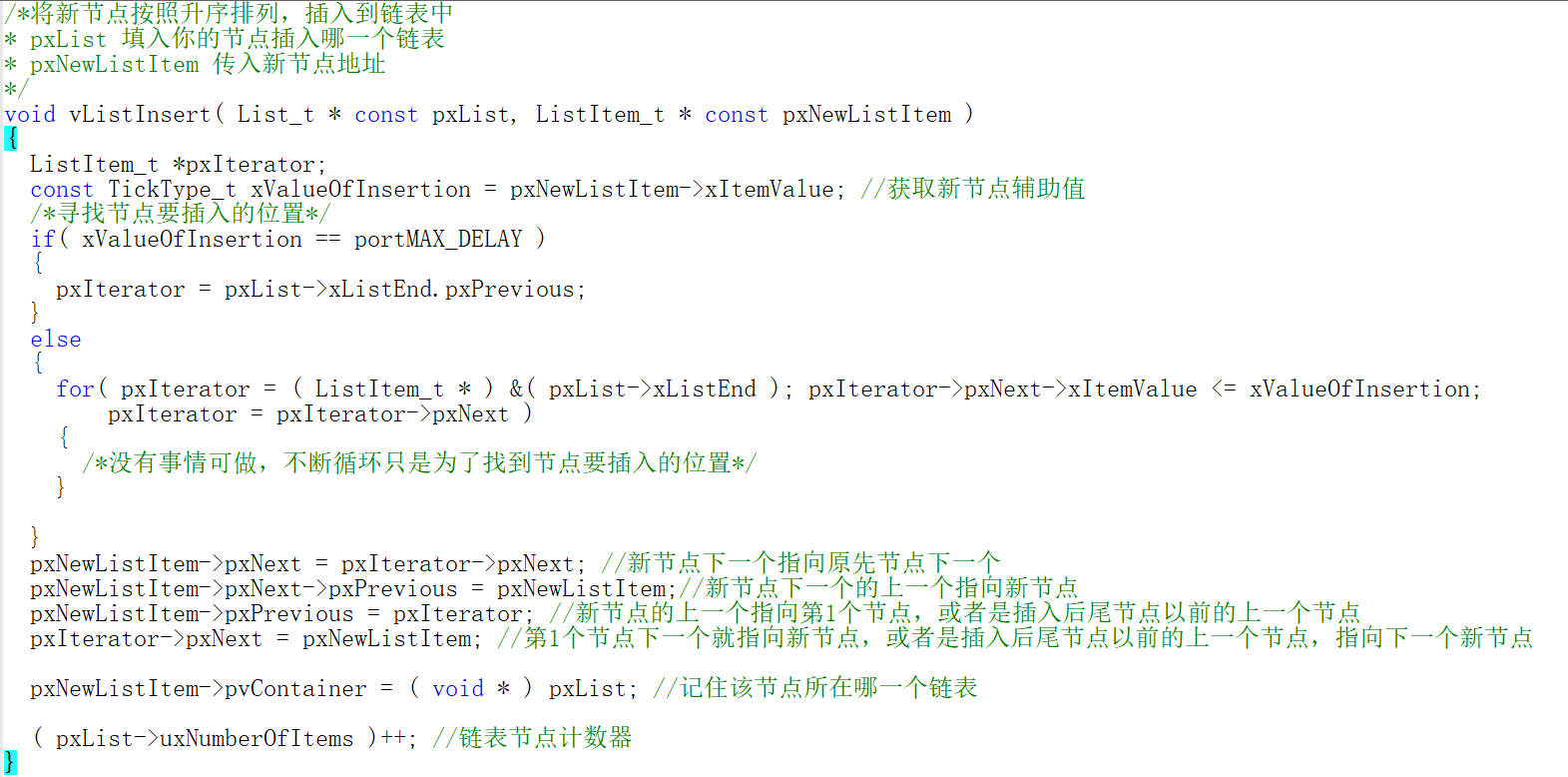


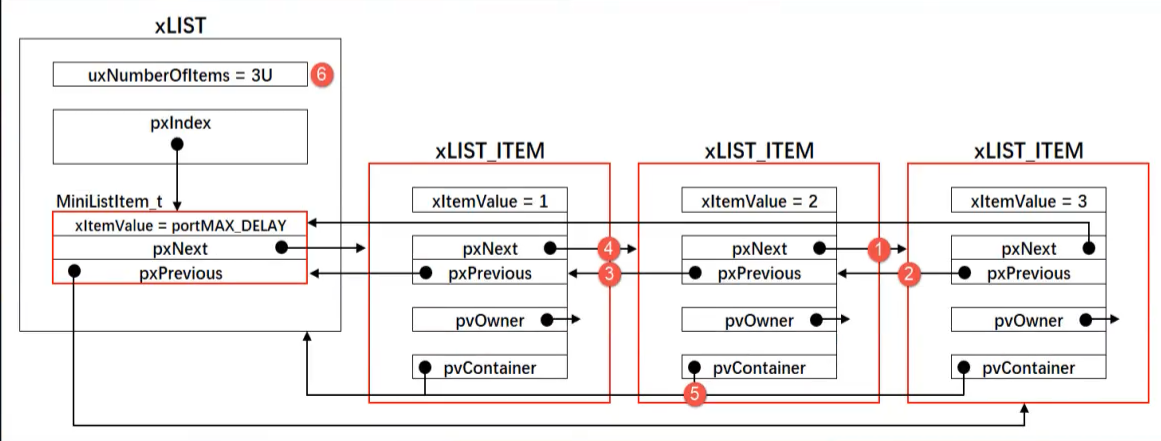




除了尾部插入新节点，也可以按照升序的方式插入新节点，这样链表就是按照1,2,3….节点顺序排列

list.c





pxIterator->pxNext = pxNewListItem

pxNewListItem->pxNext = pxIterator->pxNext;

pxNewListItem->pxPrevious = pxIterator

pxNewListItem->pxNext->pxPrevious = pxNewListItem;

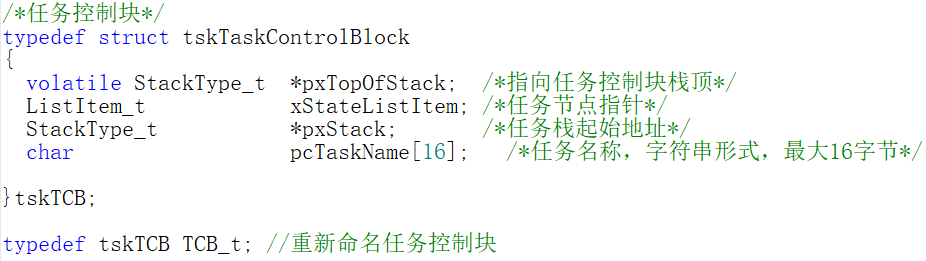
删除节点

list.c

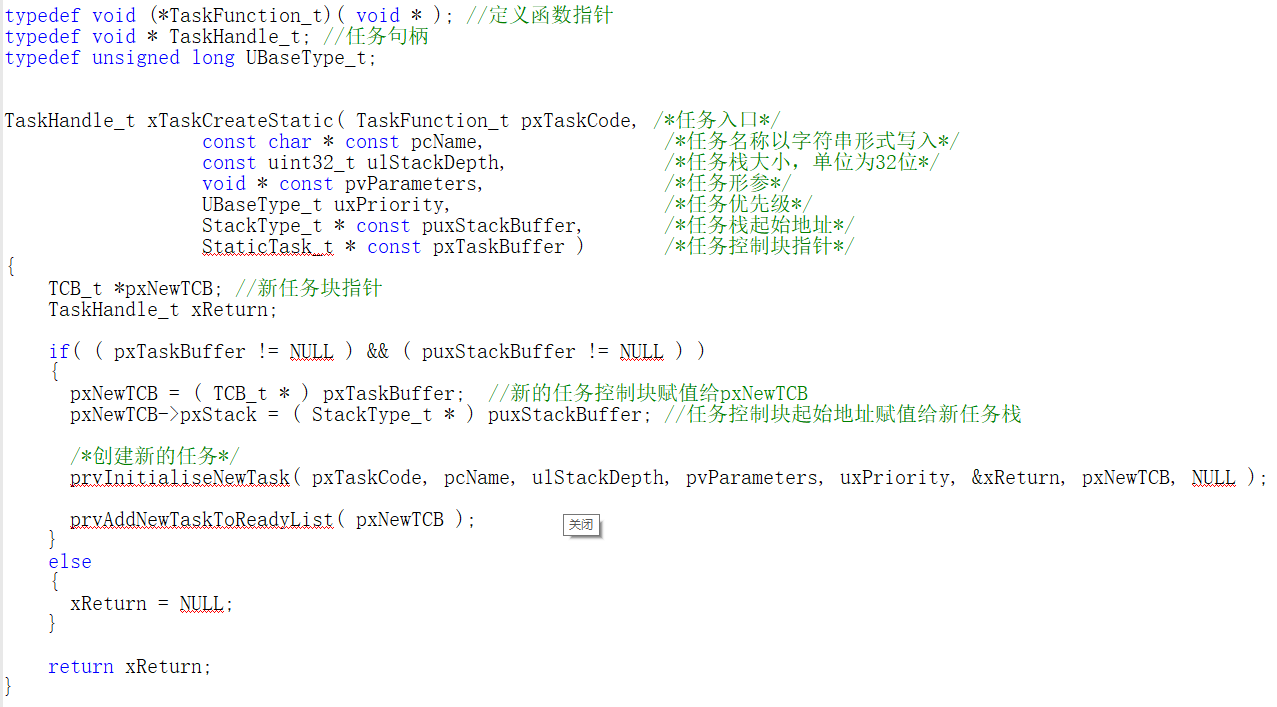


任务控制块创建

在FreeRTOS.h中定义



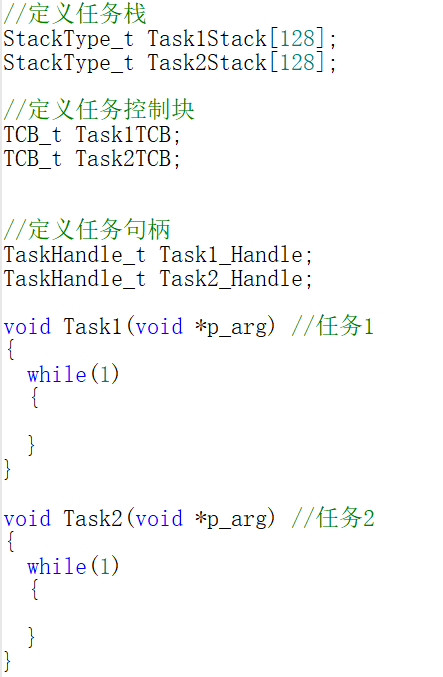
task.c 任务创建函数



具体任务创建函数实现细节我们暂时不讲，先使用起来

任务就绪表实现

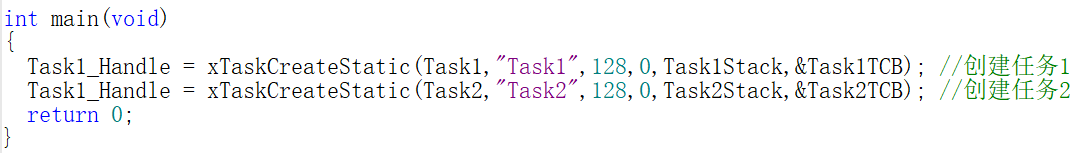
创建任务 第1.需要任务栈，这个任务栈就是用来存放任务全局变量和临时变量的



第4.任务执行函数

第3.定义任务句柄

第2.需要任务控制块，有多少个任务就需要多少个任务控制块



创建成功后返回任务句柄

任务优先级

任务控制块

任务栈

任务栈大小，必须和上面任务栈数组大小一样

填入任务函数地址

填入任务名称(自定义)

以上过程都会和任务就绪表有关系

未完待续……

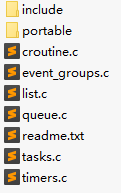
**FreeRTOS移植到STM32F1**

去托管网站下载freertos源码https://sourceforge.net/projects/freertos/files/FreeRTOS/

下载9.0.0版本

下载FreeRTOSv9.0.0.zip压缩包

压缩包



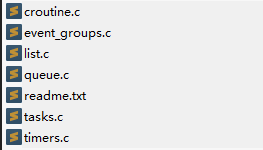


Source就是源文件

Portable里面就是与硬件相关的文件

自己创建工程文件，将STM32F1的串口和固件库移植完成，再创建FreeRTOS工程文件。

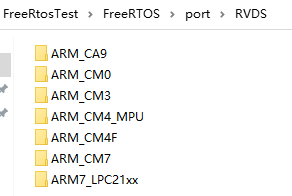
 

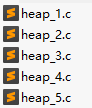
 1.将源码拷贝进FreeRTOS/src

 其实keil目录和RVDS目录是一样的，所以我们拷贝RVDS

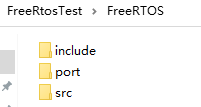
2.portable文件里与keil相关的RVDS拷贝进自己工程FreeRTOS/port

这就是RVDS内容，不同的内核

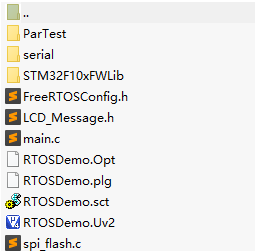
  将MemMang目录里面的拷贝进port目录

工程里面只需要加入heap\_4.c

 3.将include目录拷贝进FreeRTOS

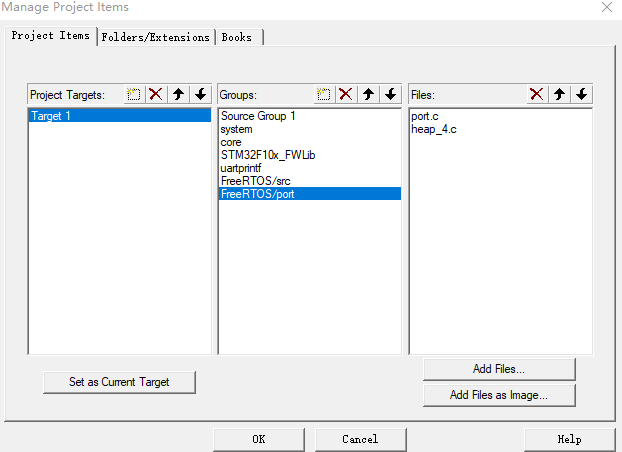
这就是FreeRTOS目录下所有拷贝得到的内容。

4.还要记得将配置文件拷贝到你的根目录下，这个配置文件FreeRTOSconfig.h去demo目录里面找

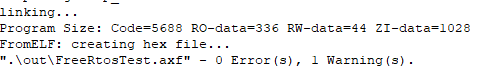
  

将FreeRTOSconfig.h 拷贝到自己工程根目录下，或者FreeRTOS目录下。

现在添加源码到自己工程里面

port导入ARM\_CM3的文件和MemMang的文件

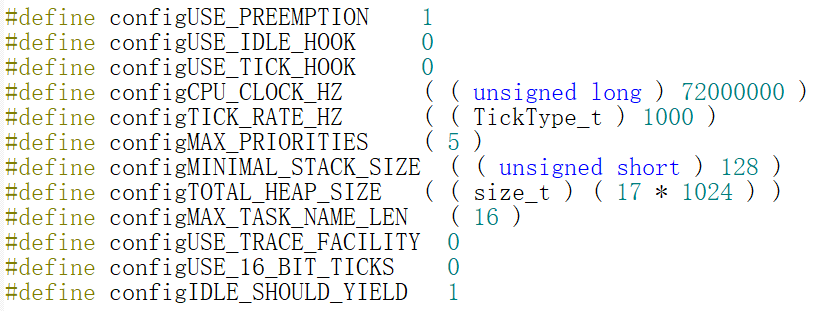
加入进头文件后进行编译。



第一次编译完全通过，但是实际是有问题的，很多配置没有打开和移植，下面进行配置和移植。

详解FreeRTOSConfig.h文件

我们需要明确一个问题，**FreeRTOSConfig.h是一个用户级别的文件，不属于内核文件。每个用户可以有不同的FreeRTOSConfig.h**。所以我们在 **FreeRTOSConfig.h 看到有些宏没有，并不是系统本身没有。**其绝大多数配置选项都体现在**FreeRTOS.h**（注意是FreeRTOS.h不是FreeRTOSConfig.h）中。



#define configUSE\_PREEMPTION 1 //支持抢占式调度，写1支持抢占调度

#define configUSE\_TIME\_SLICING 1 //支持时间片功能，这个宏其实是在FreeRTOS.h里面，而不是在

**FreeRTOSConfig.h里。我在FreeRTOSConfig.h**里定义了之后，也就等同于**FreeRTOS.h使能时间片功能**

#define configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 1 //在执行下一个任务的时候有两种方法可以选择，我选择特殊方法

通用方法

1.configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 为 0 取消特殊方法使用通用方法，或者硬件本身不支持这种特殊方法

2.可以用于所有 FreeRTOS 支持的硬件

3.完全用 C 实现，效率略低于特殊方法

4.不强制要求限制最大可用优先级数目

特殊方法

1.必须将 configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 设置为 1 特殊方法

2.必须将 configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 设置为 1。

3.比通用方法更高效

4.一般强制限定最大可用优先级数目为 32

#define configUSE\_TICKLESS\_IDLE 0 //低功耗模式

置 1：使能低功耗 tickless 模式；置 0：保持系统节拍（tick）中断一直运行

如果保持低功耗模式会导致下载程序出问题，因为下载程序时说不准芯片进入低功耗了。当然直接擦除flash再次下载可以解决这个IDLE下载问题。

#define configCPU\_CLOCK\_HZ ( ( unsigned long ) 72000000 ) //系统时钟我们定义72Mhz

#define configTICK\_RATE\_HZ ( ( TickType\_t ) 1000 ) //中断频率，1秒钟中断次数

RTOS 系统节拍中断的频率。即一秒中断的次数，每次中断 RTOS 都会进行任务调度，这里是1000，也就是1秒钟中断1000次。也就是每1毫秒中断一次。

#define configMAX\_PRIORITIES ( 32 ) //最大可用优先级，因为前面使用的特殊方法，所以只能设定到32

#define configMINIMAL\_STACK\_SIZE ( ( unsigned short ) 128 ) //空闲任务栈，定义为128字

如果空闲任务没有做什么事情，可以将空闲任务设置小点，比如64,32节省内存。

#define configMAX\_TASK\_NAME\_LEN ( 16 ) //任务名字长度16字节，也可自定义长度。

#define configUSE\_16\_BIT\_TICKS 0 //系统节拍最大计数，1表示16位，0表示32位

我们systick滴答定时器是32位的，所以最大计数到0xffffffff, 所以我们填0。

#define configIDLE\_SHOULD\_YIELD 1 //填1，空闲任务和其它任务同时执行时，空闲任务放弃优先权，让其它任务先执行。填0，空闲任务不放弃当前执行机会。

#define configUSE\_QUEUE\_SETS 0 //是否启用消息队列，填0不启用，填1启用。

#define configUSE\_TASK\_NOTIFICATIONS 1 //任务通知，默认开启

#define configUSE\_MUTEXES 0 //互斥量，填0不使用，填1使用。

#define configUSE\_RECURSIVE\_MUTEXES 0 //递归互斥信号量，填0不使用，填1使用。

#define configUSE\_COUNTING\_SEMAPHORES 0 //填0不使用计数信号量

#define configQUEUE\_REGISTRY\_SIZE 10 //设置可以注册信号量和消息队列的个数，这里注册10个

#define configSUPPORT\_DYNAMIC\_ALLOCATION 1 //设置为1支持动态内存分配

#define configSUPPORT\_STATIC\_ALLOCATION 0 //设置为0 取消静态内存分配。

一般使用动态内存分配，静态内存分配要自己手动写死任务栈大小，前面重新写FreeRTOS内核章节讲过。

#define configTOTAL\_HEAP\_SIZE ( ( size\_t ) ( 36 \* 1024 ) ) //系统所有总的堆大小 36K RAM

#define configUSE\_IDLE\_HOOK 0

#define configUSE\_MALLOC\_FAILED\_HOOK 0 //使用内存申请失败钩子函数，这种就是如果内存分配失败，你需要执行什么相关的程序，可以放入这个钩子函数中执行。

#define configCHECK\_FOR\_STACK\_OVERFLOW 0 //堆栈检测钩子函数

在测试项目的时候可以将其置1，实现堆栈检测钩子函数，看看有没有堆栈溢出情况，成品出货可以关闭该功能。

#define xPortPendSVHandler PendSV\_Handler //任务切换宏必须加入，会去调用port.c下面的汇编，该汇编有切换STM32F1内部寄存器的功能，所以就不用我们再去移植汇编了

#define vPortSVCHandler SVC\_Handler //任务切换宏必须加入，会去调用汇编，同上功能

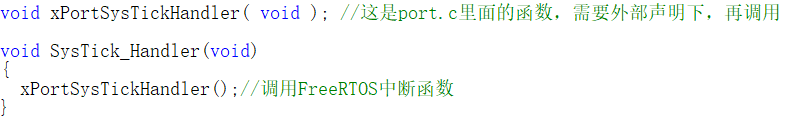
这两个任务切换宏很重要

同时这两个宏xPortPendSVHandler PendSV\_Handler，vPortSVCHandler SVC\_Handler 会和STM32 ….it.c里面的SVC和PendSV中断函数重复定义，所以去屏蔽….it.c里面的void SVC\_Handler(void)和void PendSV\_Handler(void)

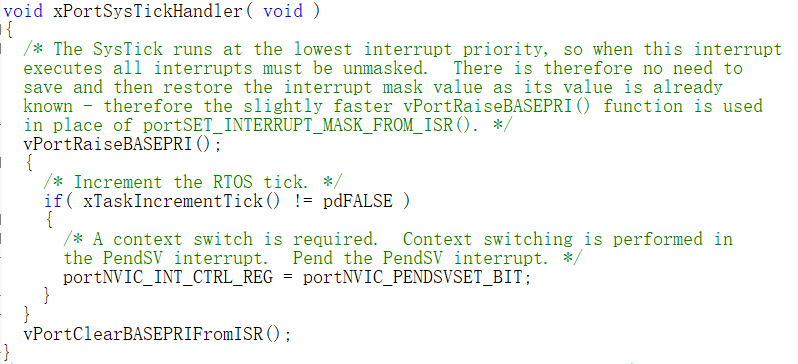
 这样你的任务切换功能就算实现了。

在systick中断中调用FreeRTOS的中断处理函数





FreeRTOS/port/port.c



编译通过。

但是还是需要再做点修改，主要修改stm32f10x\_it.c

#include "FreeRTOS.h"

#include "task.h"

void SysTick\_Handler(void)

{

#if(INCLUDE\_xTaskGetSchedulerState == 1) //当调度器启动后 我们才能执行这个函数

{

#endif

xPortSysTickHandler();//调用FreeRTOS中断函数

#if(INCLUDE\_xTaskGetSchedulerState == 1)

}

#endif

}

改成这样最好，编译通过。

FreeRTOSconfig.h里面我自定义的初步宏，这些基本宏定义了才可以实现任务切换和一些基本功能

#define configUSE\_PREEMPTION 1

#define configUSE\_IDLE\_HOOK 0

#define configUSE\_TICK\_HOOK 0

#define configCPU\_CLOCK\_HZ ( ( unsigned long ) 72000000 )

#define configTICK\_RATE\_HZ ( ( TickType\_t ) 1000 )

#define configMAX\_PRIORITIES ( 32 ) //最大优先级

#define configMINIMAL\_STACK\_SIZE ( ( unsigned short ) 128 ) //空闲任务栈128字

#define configTOTAL\_HEAP\_SIZE ( ( size\_t ) ( 17 \* 1024 ) )

#define configMAX\_TASK\_NAME\_LEN ( 16 ) //任务名长度16字

#define configUSE\_TRACE\_FACILITY 0

#define configUSE\_16\_BIT\_TICKS 0 //系统最大节拍数32位

#define configIDLE\_SHOULD\_YIELD 1 //空闲任务放弃优先权

/\* Co-routine definitions. \*/

#define configUSE\_CO\_ROUTINES 0

#define configMAX\_CO\_ROUTINE\_PRIORITIES ( 2 )

/\* Set the following definitions to 1 to include the API function, or zero

to exclude the API function. \*/

#define INCLUDE\_vTaskPrioritySet 1

#define INCLUDE\_uxTaskPriorityGet 1

#define INCLUDE\_vTaskDelete 1

#define INCLUDE\_vTaskCleanUpResources 0

#define INCLUDE\_vTaskSuspend 1

#define INCLUDE\_vTaskDelayUntil 1

#define INCLUDE\_vTaskDelay 1

#define configKERNEL\_INTERRUPT\_PRIORITY 255

#define configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 191 /\* equivalent to 0xb0, or priority 11. \*/

/\* This is the value being used as per the ST library which permits 16

priority values, 0 to 15. This must correspond to the

configKERNEL\_INTERRUPT\_PRIORITY setting. Here 15 corresponds to the lowest

NVIC value of 255. \*/

#define configLIBRARY\_KERNEL\_INTERRUPT\_PRIORITY 15

#define configSUPPORT\_STATIC\_ALLOCATION 1 //我先使用静态方式创建任务，所以要打开静态分配内存功能

//#define configSUPPORT\_DYNAMIC\_ALLOCATION 1

#define configUSE\_TIME\_SLICING 1

#define configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 1

#define configUSE\_QUEUE\_SETS 0 //暂不启用消息队列

#define configUSE\_TASK\_NOTIFICATIONS 1 //任务通知开启

#define configUSE\_MUTEXES 0 //互斥量暂不使用

#define configUSE\_RECURSIVE\_MUTEXES 0

#define configUSE\_COUNTING\_SEMAPHORES 0 //计数信号量暂不使用

#define configQUEUE\_REGISTRY\_SIZE 10 //信号量消息队列个数可以注册10个

#define xPortPendSVHandler PendSV\_Handler //任务切换函数必须加入，会去调用汇编

#define vPortSVCHandler SVC\_Handler //任务切换函数必须加入，会去调用汇编

因为我们使用的是静态方式创建任务，所以需要实现空闲任务的堆栈和空闲任务控制块

/\* 空闲任务任务堆栈 \*/

static StackType\_t Idle\_Task\_Stack[128];

/\* 空闲任务控制块 \*/

static StaticTask\_t Idle\_Task\_TCB;

void vApplicationGetIdleTaskMemory(StaticTask\_t \*\*ppxIdleTaskTCBBuffer,

StackType\_t \*\*ppxIdleTaskStackBuffer,

uint32\_t \*pulIdleTaskStackSize )

{

\*ppxIdleTaskTCBBuffer=&Idle\_Task\_TCB;/\* 任务控制块内存 \*/

\*ppxIdleTaskStackBuffer=Idle\_Task\_Stack;/\* 任务堆栈内存 \*/

\*pulIdleTaskStackSize=configMINIMAL\_STACK\_SIZE;/\* 任务堆栈大小 \*/

}

将这个空闲任务控制块放在main文件下，或者其它文件都可以，系统会自动去调用

在config文件定义#define configTIMER\_TASK\_STACK\_DEPTH 128 //定时器任务堆栈大小

/\* 定时器任务堆栈 \*/

static StackType\_t Timer\_Task\_Stack[128];

/\* 定时器任务控制块 \*/

static StaticTask\_t Timer\_Task\_TCB;

void vApplicationGetTimerTaskMemory(StaticTask\_t \*\*ppxTimerTaskTCBBuffer,

StackType\_t \*\*ppxTimerTaskStackBuffer,

uint32\_t \*pulTimerTaskStackSize)

{

\*ppxTimerTaskTCBBuffer=&Timer\_Task\_TCB;/\* 任务控制块内存 \*/

\*ppxTimerTaskStackBuffer=Timer\_Task\_Stack;/\* 任务堆栈内存 \*/

\*pulTimerTaskStackSize=configTIMER\_TASK\_STACK\_DEPTH;/\* 任务堆栈大小 \*/

}

在main文件或者其它文件下实现定时器任务控制函数, 系统会自动去调用

**创建静态任务**

TaskHandle\_t xTaskCreateStatic( TaskFunction\_t pxTaskCode,

const char \* const pcName,

const uint32\_t ulStackDepth,

void \* const pvParameters,

UBaseType\_t uxPriority,

StackType\_t \* const puxStackBuffer,

StaticTask\_t \* const pxTaskBuffer ) //静态任务创建函数

pxTaskCode: 填入任务函数的名称，也就是函数地址

\* const pcName: 传入任务名称字符串

ulStackDepth: 任务堆栈大小，必须定义的任务栈数组大小一样

pvParameters: 外部参数传递给任务内部使用

uxPriority: 任务优先级

puxStackBuffer: 传入定义的任务堆栈数组

pxTaskBuffer: 传入定义的任务控制块

#include "stm32f10x.h"

#include "sysclock.h"

#include "uartprintf.h"

#include "stdio.h"

#include "FreeRTOS.h" //使用FreeRTOS里面的函数一定要包含这个有文件

#include "task.h" //使用FreeRTOS里面的函数一定要包含这个有文件

//定义任务栈

StackType\_t Task1Stack[128];

StackType\_t Task2Stack[128];

//定义任务控制块

static StaticTask\_t Task1TCB;

static StaticTask\_t Task2TCB;

//定义任务句柄

static TaskHandle\_t Task1\_Handle;

static TaskHandle\_t Task2\_Handle;

void Task1(void)

{

while(1)

{

printf("Task1....\r\n");

vTaskDelay(500);

}

}

void Task2(void)

{

while(1)

{

printf("Task2....\r\n");

vTaskDelay(500);

}

}

int main(void)

{

RCC\_configuration();//初始化时钟

USART\_config(115200);//初始化串口

NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4); //STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为： 0~15 //优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，

//都统一用这个优先级分组，千万不要再分组，切忌。

printf("xxxzzzzzz\r\n");

Task1\_Handle = xTaskCreateStatic((TaskFunction\_t)Task1,

(const char\* )"Task1",

(uint32\_t)128,

(void\*)NULL,

(UBaseType\_t)4,

(StackType\_t\* )Task1Stack,

(StaticTask\_t\* )&Task1TCB);

Task2\_Handle = xTaskCreateStatic((TaskFunction\_t)Task2,

(const char\* )"Task1",

(uint32\_t)128,

(void\*)NULL,

(UBaseType\_t)4,

(StackType\_t\* )Task2Stack,

(StaticTask\_t\* )&Task2TCB);

if ((NULL != Task1\_Handle) && (NULL != Task2\_Handle)) //如果Task1任务和Task2任务创建成功就执行启动任务

{

vTaskStartScheduler(); /\* 启动任务，开启调度 \*/

}

else

{

printf("task creat failed..\r\n");

}

while(1)

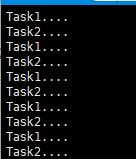
{

delay\_ms(500);

}

return 0;

}

任务成功同时执行

**动态创建任务**

比如有些任务只是在某时间段执行一次就不再执行了，这种就可以用动态创建任务方法。

FreeRTOS堆空间大小是专门定义了一个全局数组来当做堆使用的。

在FreeRTOSconfig.h中我们可以定义堆大小





我们定义36K堆大小，我们编译后SRAM就被占用了40K左右，如果我只定义1024字节堆大小，我们的SRAM编译出来就差不多5K，所以这个堆大小你可以自己定义。



动态分配内存置1，这样静态动态都可以使用了。

BaseType\_t xTaskCreate( TaskFunction\_t pxTaskCode,

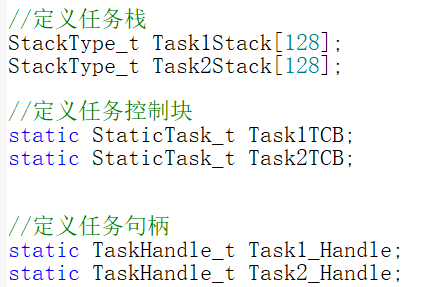
const char \* const pcName,

const uint16\_t usStackDepth,

void \* const pvParameters,

UBaseType\_t uxPriority,

TaskHandle\_t \* const pxCreatedTask ) //动态创建任务



这里要把静态StaticTask\_t改成TaskHandle\_t

取消静态任务需要使用的任务栈数组和任务句柄。

#include "stm32f10x.h"

#include "sysclock.h"

#include "uartprintf.h"

#include "stdio.h"

#include "FreeRTOS.h" //使用FreeRTOS里面的函数一定要包含这个有文件

#include "task.h" //使用FreeRTOS里面的函数一定要包含这个有文件

//定义任务控制块

static TaskHandle\_t Task1TCB;

static TaskHandle\_t Task2TCB;

void Task1(void)

{

while(1)

{

printf("Task1....\r\n");

vTaskDelay(500);

}

}

void Task2(void)

{

while(1)

{

printf("Task2....\r\n");

vTaskDelay(500);

}

}

int main(void)

{

BaseType\_t xReturn1,xReturn2;

RCC\_configuration();//初始化时钟

USART\_config(115200);//初始化串口

NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4); //STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为： 0~15

//优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断， //都统一用这个优先级分组，千万不要再分组，切忌。

printf("xxxzzzzzz\r\n");

xReturn1 = xTaskCreate((TaskFunction\_t)Task1, //任务函数地址

(const char\* )"Task1", //任务名

(uint32\_t)128, //任务栈大小

(void\*)NULL, //向函数传参

(UBaseType\_t)4, //优先级

(TaskHandle\_t\* )&Task1TCB); //任务控制块

xReturn2 = xTaskCreate((TaskFunction\_t)Task2,

(const char\* )"Task1",

(uint32\_t)128,

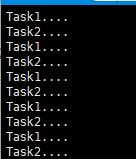
(void\*)NULL,

(UBaseType\_t)4,

(TaskHandle\_t\* )&Task2TCB);

if((xReturn1 == pdPASS) && (xReturn2 == pdPASS)) //判断任务1和任务2是否 == pdPASS 如果等于证明任务创建成功

{

执行成功

vTaskStartScheduler(); /\* 启动任务，开启调度 \*/

}

else

{

printf("task creat failed..\r\n");

}

while(1)

{

delay\_ms(500);

}

return 0;

}

**杀死任务**

void vTaskDelete( TaskHandle\_t xTaskToDelete ) //杀死任务函数

无法杀死任务的原因

int main(void)

{

BaseType\_t xReturn1,xReturn2;

RCC\_configuration();//初始化时钟

USART\_config(115200);//初始化串口

NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4);

printf("xxxzzzzzz\r\n");

xReturn1 = xTaskCreate((TaskFunction\_t)Task1, //任务函数地址

(const char\* )"Task1", //任务名

(uint32\_t)128, //任务栈大小

(void\*)NULL, //向函数传参

(UBaseType\_t)4, //优先级

(TaskHandle\_t\* )&Task1TCB); //任务控制块

xReturn2 = xTaskCreate((TaskFunction\_t)Task2,

(const char\* )"Task1",

(uint32\_t)128,

(void\*)NULL,

(UBaseType\_t)4,

(TaskHandle\_t\* )&Task2TCB);

if((xReturn1 == pdPASS) && (xReturn2 == pdPASS)) //判断任务1和任务2是否 == pdPASS 如果等于证明任务创建成功

{

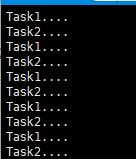
vTaskStartScheduler(); /\* 启动任务，开启调度 \*/

}

else

{

printf("task creat failed..\r\n");

任务2还是没有被杀死

}

vTaskDelay(1000);

vTaskDelete(Task2TCB); //杀死任务，传入要杀死任务的任务控制块

printf("vTaskDelete..\r\n");

while(1)

{

delay\_ms(500);

}

return 0;

}

执行杀死任务2之后，任务2还在继续执行？这是为什么呢？

if((xReturn1 == pdPASS) && (xReturn2 == pdPASS))

{

vTaskStartScheduler(); /\* 启动任务，开启调度 \*/

}

当任务启动之后vTaskStartScheduler()，主函数main就不会再向下执行了，两个任务把优先级占用完了，这样vTaskDelete(Task2TCB)也就执行不到

else

{

printf("task creat failed..\r\n");

}

vTaskDelay(1000);

vTaskDelete(Task2TCB); //杀死任务，传入要杀死任务的任务控制块

printf("vTaskDelete..\r\n");

while(1)

{

delay\_ms(500);

}

return 0;

现在唯一的方法就是把任务1用来做杀死任务2的方法。

//定义任务控制块

static TaskHandle\_t Task1TCB;

static TaskHandle\_t Task2TCB;

void Task2(void);//任务2声明

void Task1(void)

{

int i = 0;

任务1执行10次之后就杀死任务2

while(1)

{

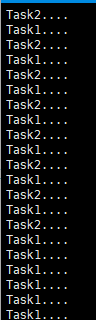
i++;

if(i == 10)

vTaskDelete(Task2TCB); //杀死任务，传入要杀死任务的任务控制块

printf("Task1....\r\n");

vTaskDelay(500);

任务2被杀死后面都是任务1

}

}

void Task2(void)

{

while(1)

{

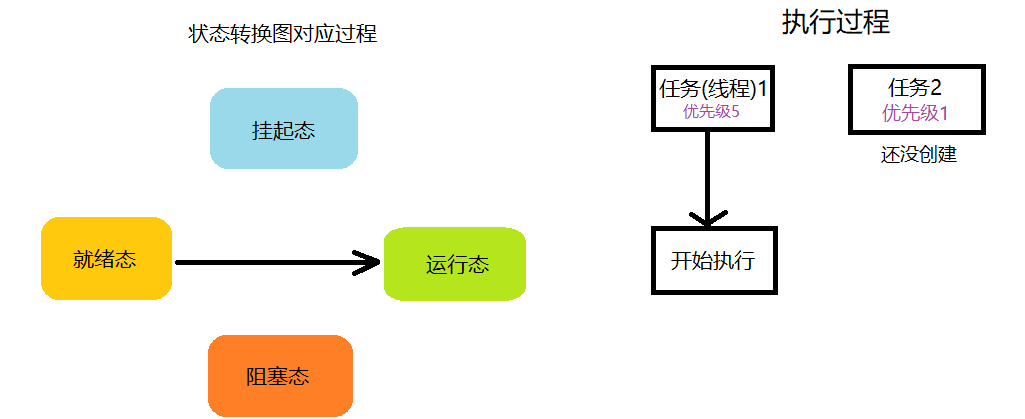
printf("Task2....\r\n");

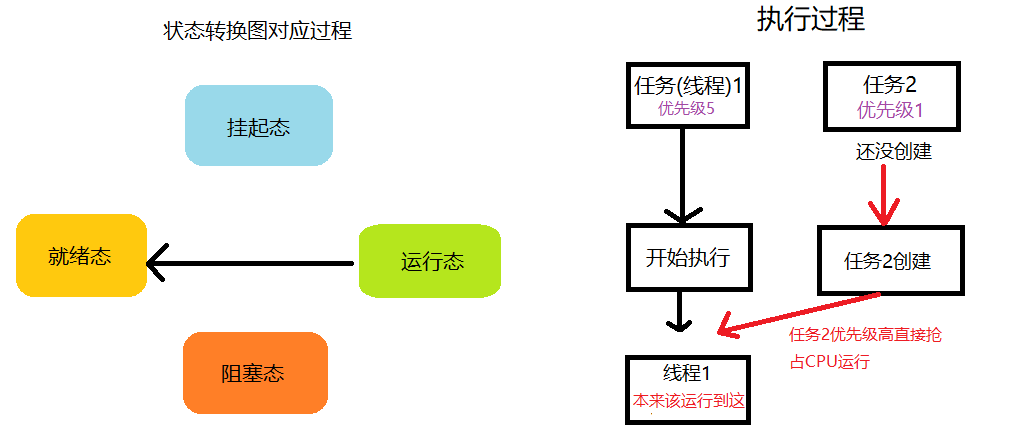
vTaskDelay(500);

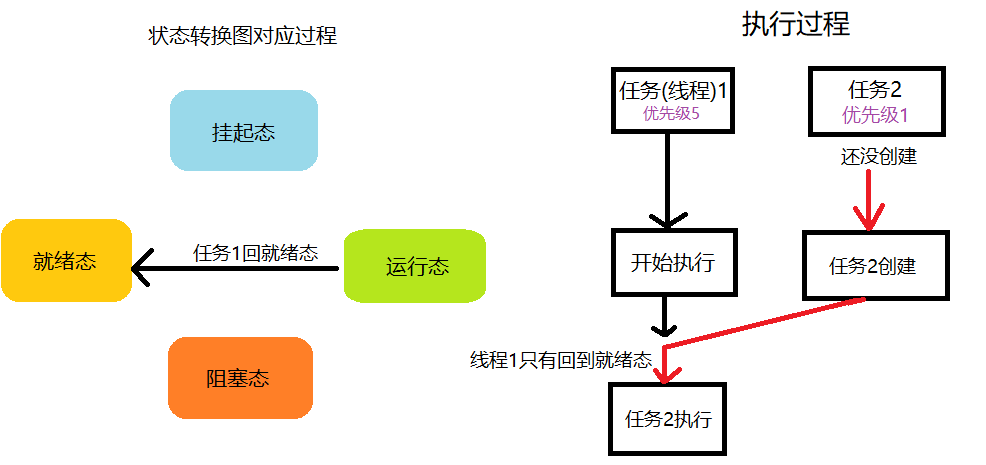
}

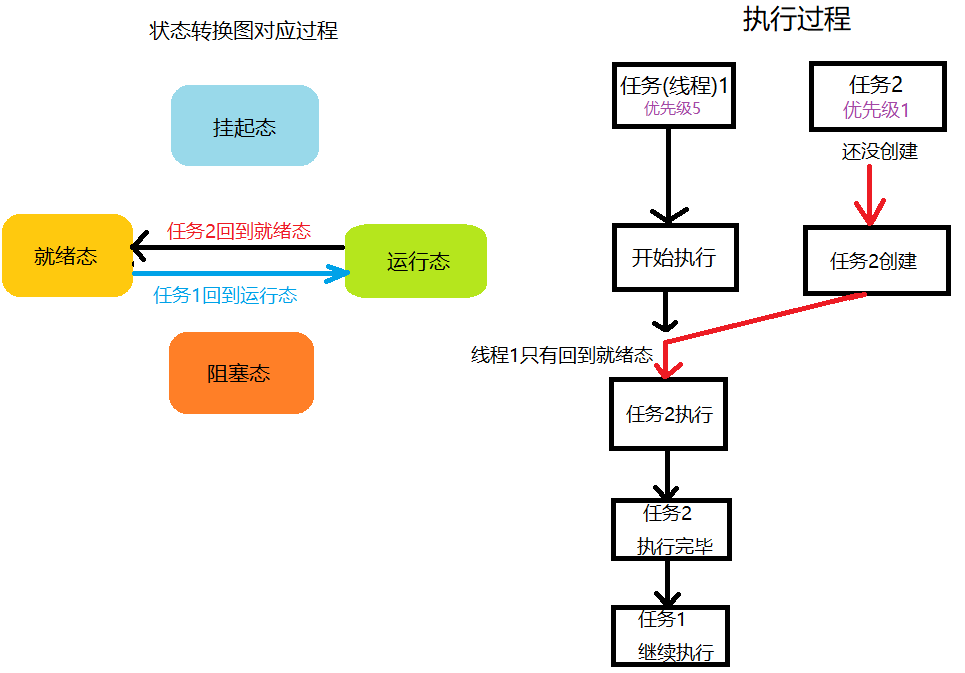
}

**任务状态转换表，挂起，阻塞，就绪，运行的含义**



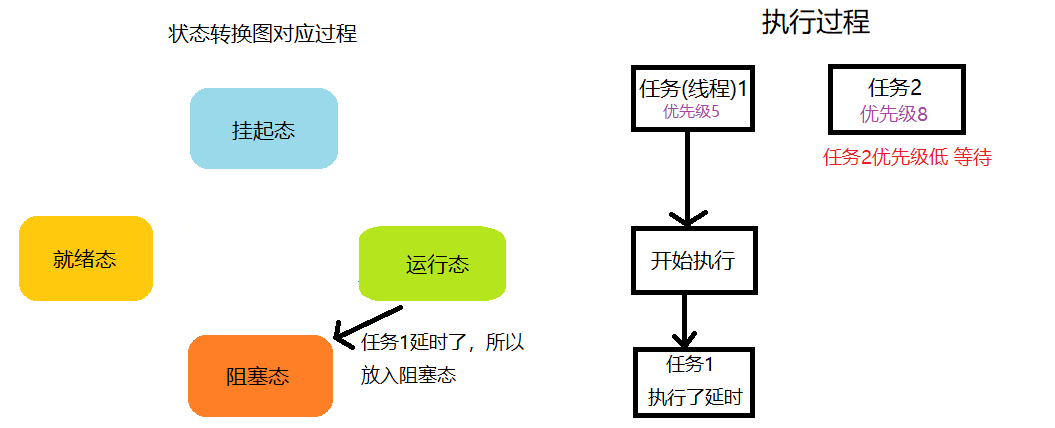


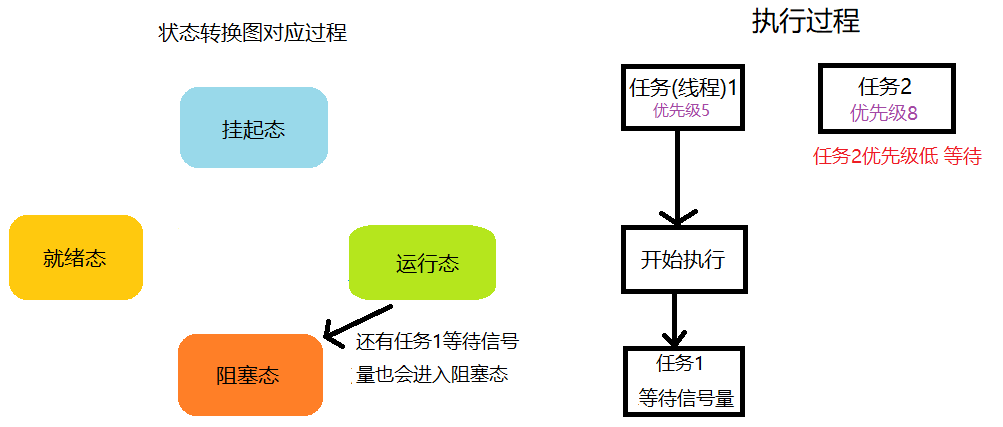


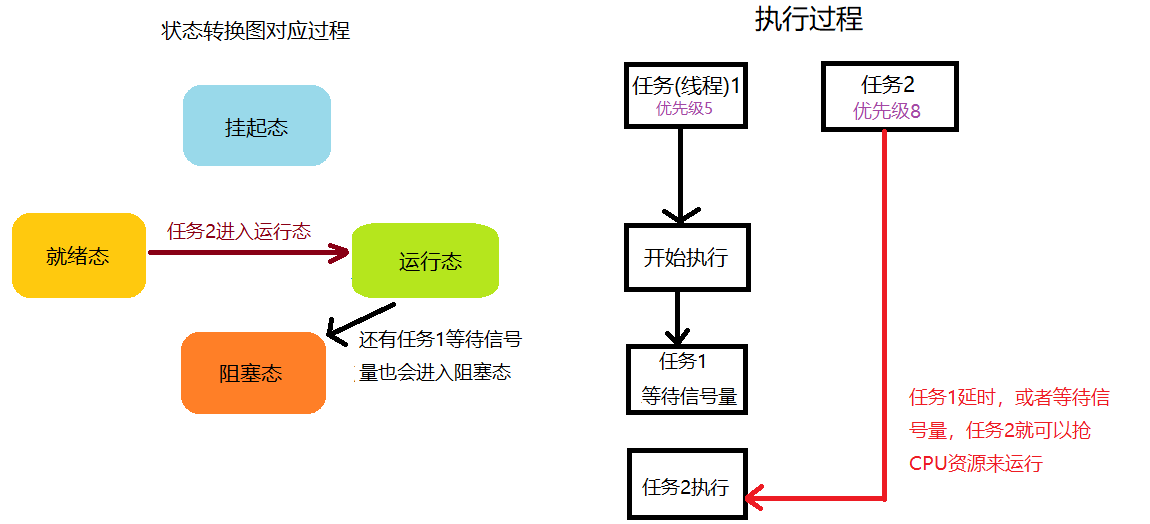


这是在常规情况下有CPU安排调度的过程

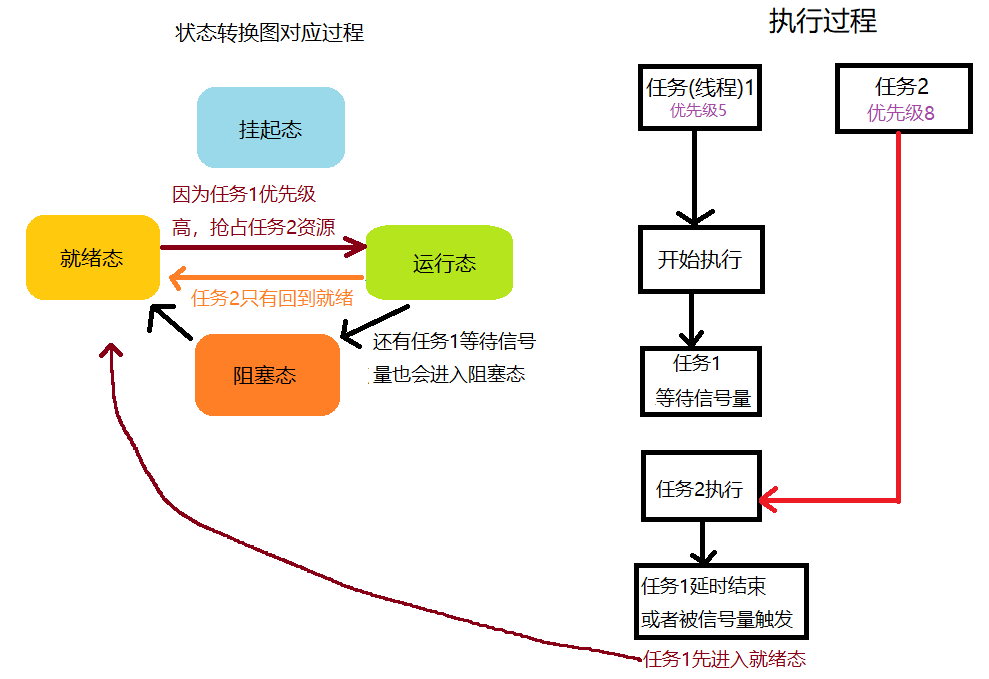
下面是其它情况会出现的状态





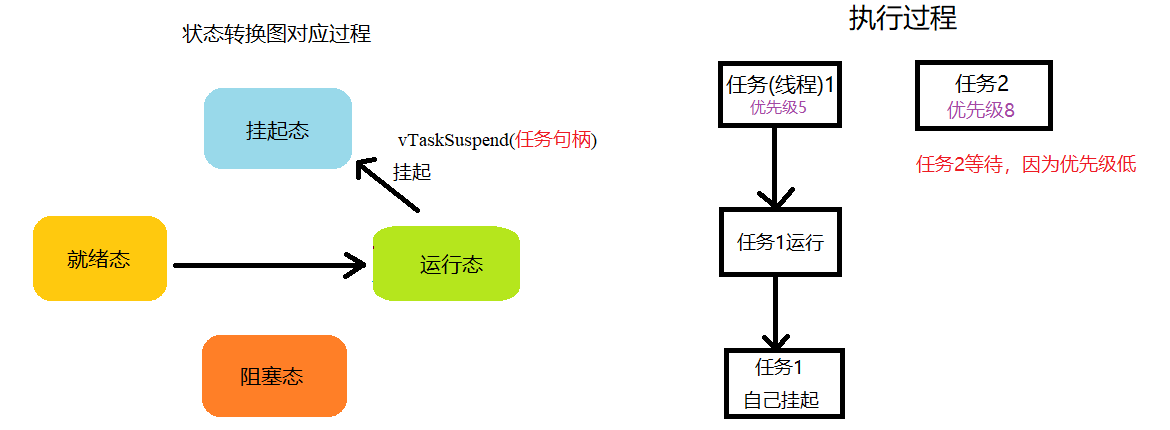


等待任务1延时结束，或者被其它中断发过来的信号量触发，那么任务1会进入就绪态。

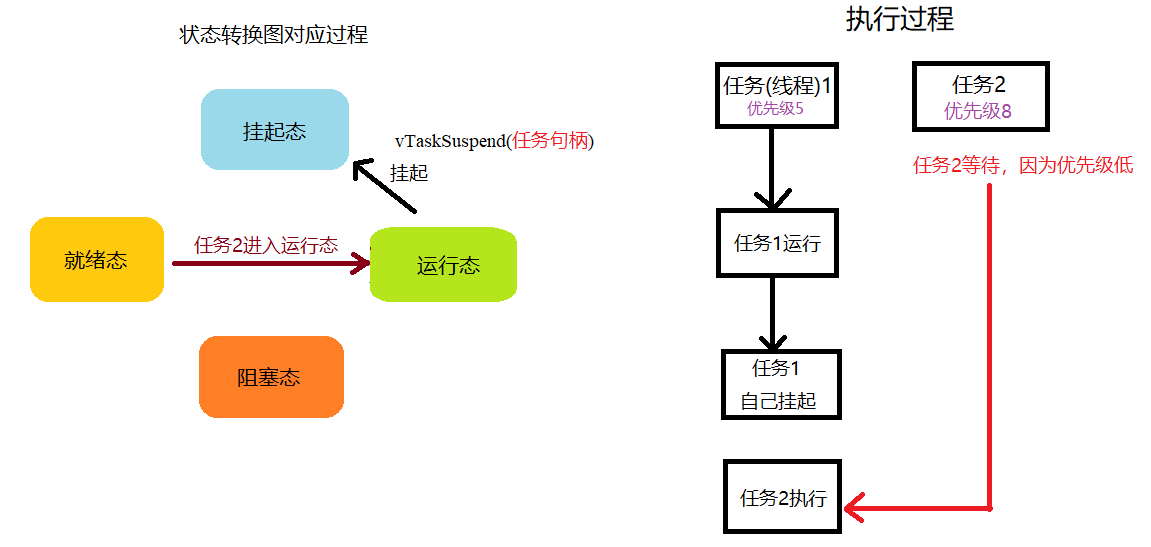


任务挂起(挂起态)，并不是其它任务造成的，是任务执行了某个函数，自觉自愿挂起的。

挂起的好处是该任务不管优先级多高，都不能抢占其它低优先级任务的CPU进行运行。



任务1挂起后任务2就可以执行了。

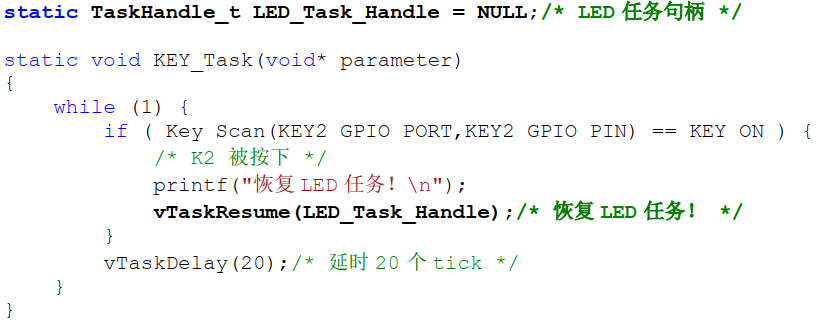


这种任务挂起的应用场景，比如按键被按下，执行该任务挂起。



所以在这种突发事件的情况下，不管是任务延时，还是任务信号量，都无法让任务挂起，只有任务收到突发事件，自己将自己挂起，把CPU使用权交出来给其它任务使用，这就是一个应用场景。



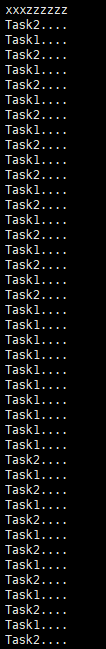


这就是恢复挂起任务的应用场景，所以如果想让其它任务来，解除任务1的挂起状态，那么任务1的任务句柄最好用全局变量。

void vTaskSuspend( TaskHandle\_t xTaskToSuspend ) //挂起任务

xTaskToSuspend: 填入要挂起任务的任务控制块。

void vTaskResume( TaskHandle\_t xTaskToResume )//恢复任务



xTaskToSuspend: 填入要恢复任务的任务控制块。

//定义任务控制块

任务1任务2执行10次之后，任务1将任务2挂起

static TaskHandle\_t Task1TCB;

static TaskHandle\_t Task2TCB;

void Task2(void);//任务2声明

void Task1(void)

{

int i = 0;

while(1)

{

i++;

if(i == 10)

任务2挂起之后就不在就绪态和运行态了，所以任务2不执行，等任务1计数20次之后恢复任务2

vTaskSuspend(Task2TCB); //挂起任务2，任务1运行10次之后主动挂起任务2

else if(i == 20)

vTaskResume(Task2TCB); //恢复任务2

else

{

}

printf("Task1....\r\n");

vTaskDelay(500);

}

}

任务2再次运行

void Task2(void)

{

while(1)

{

printf("Task2....\r\n");

vTaskDelay(500);

}

}

int main(void)

{

创建任务1，任务2…

}

TickType\_t xTaskGetTickCount( void ) //获取系统开始到现在执行了多少时间，以1毫秒为单位

TickType\_t : uint32\_t类型，获得毫秒数



这个数字是不断在累加的，不会清0，除非重启板子

void Task1(void)

{

while(1)

{

printf("time1 = %d\r\n",xTaskGetTickCount()); //获取线程1执行了多少ms

printf("Task1....\r\n");

vTaskDelay(500);

}

}

void Task2(void)

{

while(1)

{

printf("time2 = %d\r\n",xTaskGetTickCount()); //获取线程2执行了多少ms

printf("Task2....\r\n");

vTaskDelay(1000);

}

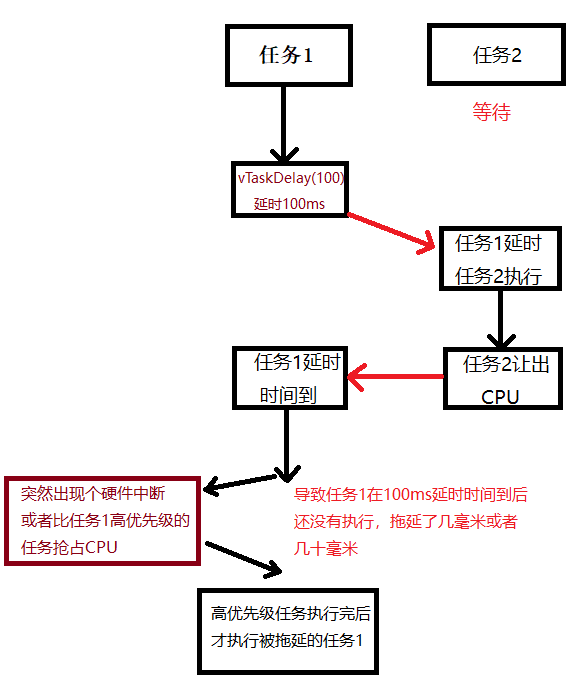
}

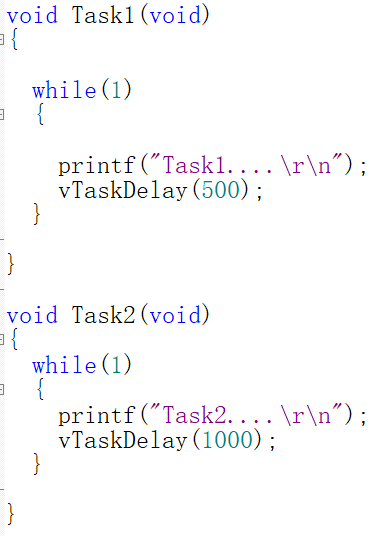
void vTaskDelay( const TickType\_t xTicksToDelay ) //阻塞延时函数

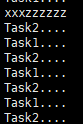
xTicksToDelay : 延时多少毫秒

单位为系统节拍周期， 比如系统的时钟节拍周期为 1ms，那么调用 vTaskDelay(1)的延时时间则为 1ms。

vTaskDelay函数不能准确的延时时间到，就可以马上执行任务





500ms延时执行

这就是vTaskDelay 延时函数的弊端，一般延时可以用，但是有些那种死脑筋的任务必须延时几百毫秒后必须马上执行，那么这种任务就不要用vTaskDelay 做延时

用vTaskDelayUntil作精确延时是个好办法，看下面…..

void vTaskDelayUntil( TickType\_t \* const pxPreviousWakeTime, const TickType\_t xTimeIncrement )

//获取精确的延时时间(绝对延时函数)，延时时间到，任务马上执行，不受中断和优先级打断

pxPreviousWakeTime: 填入上次唤醒的时间

xTimeIncrement: 设置延时时间，只不过这个参数得传入节拍数才行

返回节拍 = pdMS\_TO\_TICKS(传入延时的毫秒) //将毫秒数转换成节拍数

在config中 #define INCLUDE\_vTaskDelayUntil 1 //启用绝对延时

绝对延时vTaskDelayUntil函数使用要麻烦些，但是精确。

void Task1(void)

{

/\* 用于保存上次时间。调用后系统自动更新 \*/

static TickType\_t PreviousWakeTime;

/\* 设置延时时间，将时间转为节拍数 \*/

延时多少毫秒?

const TickType\_t TimeIncrement = pdMS\_TO\_TICKS(500);

/\* 获取当前系统时间 \*/

PreviousWakeTime = xTaskGetTickCount();

while(1)

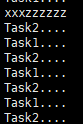
{

printf("Task1....\r\n");

vTaskDelayUntil(&PreviousWakeTime,TimeIncrement ); //当前系统时间PreviousWakeTime+ TimeIncrement节拍时间到达后，延时结束

}

}

任务1， 500ms精确延时成功

void Task2(void)

{

while(1)

{

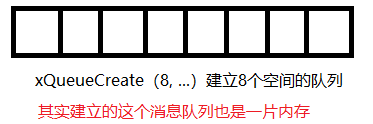
printf("Task2....\r\n");

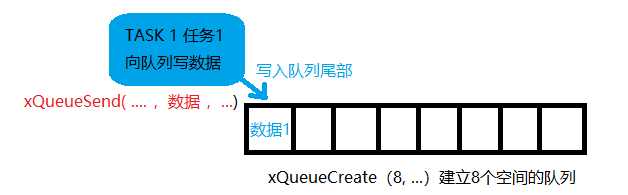
vTaskDelay(500);

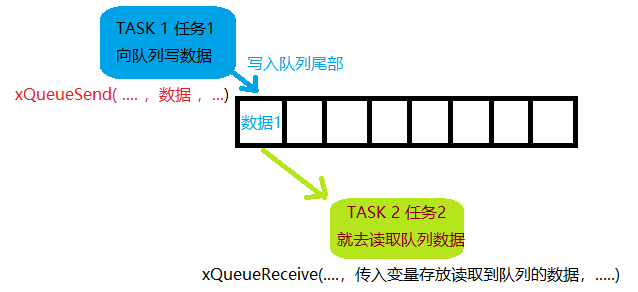
}

}

**消息队列**

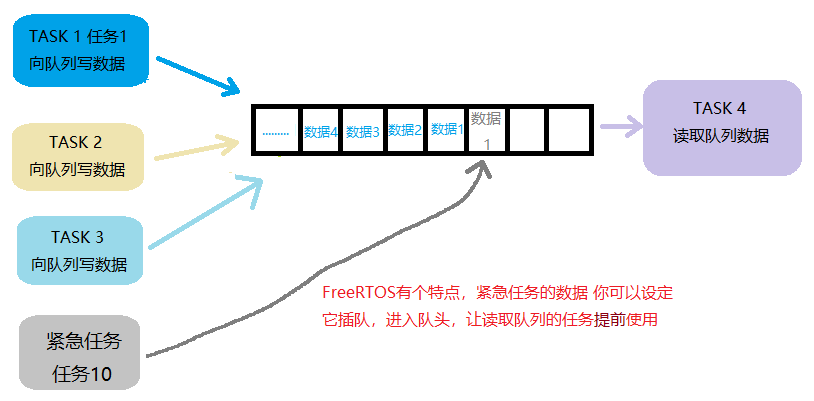


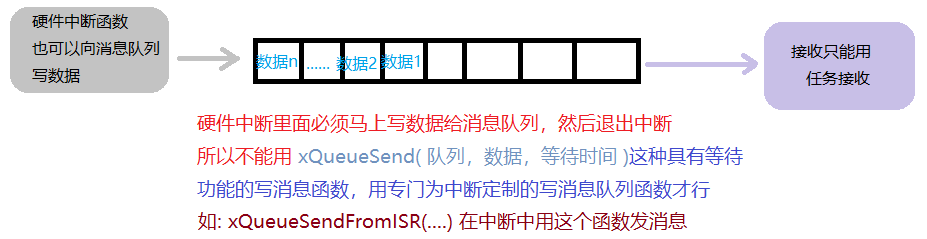




因为队列只有一个数据，所以即使队头，也是队列尾。







在config中 #define configUSE\_QUEUE\_SETS 1 //启用消息队列

typedef void \* QueueHandle\_t //创建句柄变量

QueueHandle\_t xQueueCreate( uxQueueLength, uxItemSize ) //创建消息队列，返回句柄

uxQueueLength: 消息队列空间大小，就是可以存放几个值

uxItemSize: 每个值的数据类型大小，可以是char 1字节，也可以是short 16字节，int 32字节，也可能是结构体，所以建议用sizeof来计算。

QueueHandle\_t: 返回非NULL创建成功，返回NULL创建失败

BaseType\_t xQueueSend( QueueHandle\_t xQueue, const void \*pvItemToQueue, TickType\_t xTicksToWait ) //写队列，把数据写入队列尾部

xQueue : 传入队列句柄，因为接受队列数据的函数也是通过这个句柄来判断是哪一个队列发的

\*pvItemToQueue: 传入要发送的数据

BaseType\_t: 返回pdPASS 表示发送成功，非pdPASS发送失败

BaseType\_t xQueueSendToFront( QueueHandle\_t xQueue, const void \*pvItemToQueue, TickType\_t xTicksToWait ) //写队列，把数据写入队列头部，如果你的任务有紧急的事件，就需要插队，就用这个函数

参数同上

xQueueSendFromISR(QueueHandle\_t xQueue, const void \*pvItemToQueue, 填NULL) //在中断中用非阻塞发送消息的函数，这个函数不会死等，没有数据也会执行结束。

xQueue : 传入队列句柄

\*pvItemToQueue: 传入要发送的数据

填NULL

BaseType\_t: 返回pdPASS 表示发送成功，非pdPASS发送失败

BaseType\_t xQueueReceive( QueueHandle\_t xQueue, void \*pvBuffer, TickType\_t xTicksToWait ) //接收发送过来的消息

xQueue : 传入队列句柄，要传入发送队列的句柄，我才知道接收某个队列的数据，在xQueueSend函数段说了的

\*pvBuffer: 接收发送过来的数据，如果是char，就要定义char变量来接收，如果是int就要定义int变量来接收，如果是结构体，就要定义结构体变量来接收。

xTicksToWait: 阻塞等待时间，如果我在任务中使用xQueueReceive，其它线程没有发响应的消息过来，我就会将任务卡死在这里，一直等待，把CPU让出来。如果有消息突然发过来我就不用等待，继续向下执行。等待时间最大portMAX\_DELAY(这是个很长的时间)，如果你不想等待，可以填NULL。

#include "stm32f10x.h"

#include "sysclock.h"

#include "uartprintf.h"

#include "stdio.h"

#include "FreeRTOS.h" //使用FreeRTOS里面的函数一定要包含这个有文件

#include "task.h" //使用FreeRTOS里面的函数一定要包含这个有文件

#include "queue.h" //消息队列要包含该头文件

//定义任务控制块

static TaskHandle\_t Task1TCB;

static TaskHandle\_t Task2TCB;

QueueHandle\_t xQueue; //创建一个消息队列句柄

void Task1(void)

{

uint8\_t Receive\_data; //任务2发来的数据需要存放，该变量就是用来存放的

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{

xReturn = xQueueReceive(xQueue,&Receive\_data,portMAX\_DELAY); //接收任务2发来的数据

if (pdTRUE == xReturn)

printf("Task1 Receive = %d\r\n",Receive\_data);

else

printf("Task1 Receive failed\r\n");

}

}

void Task2(void)

{

uint8\_t send\_data = 50; //要发送给任务1的数据

BaseType\_t xReturn = pdPASS;// 定义一个信息返回值来判断消息是否发送成功，默认为 pdPASS

xQueue = xQueueCreate(10,sizeof(char));// 创建消息队列，消息队列有10个空间，每个空间存放一个char大小的数据

while(1)

{

xReturn = xQueueSend( xQueue , &send\_data, 0 ); // 消息发送 等待时间 0

if (pdPASS == xReturn)

printf("Task2 Queue send success\r\n");

else

printf("Task2 Queue send failed\r\n");

vTaskDelay(1000);

}

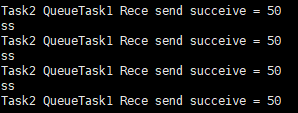
}

int main(void)

{

初始化串口，创建两个任务…..

}

从打印的数字来看，消息队列确实发生成功了的。

但是打印字符串乱序了，这是因为引入操作系统后，多线程任务切换太快的原因，需要改进printf

解决printf打印字符乱序，采用消息队列，修改printf，

需要下一章信号量的知识，可以先学习下信号量

printf函数是不可重入的，也就是在多任务情况下不能多个任务同时调用printf，所以像这种不可重入的函数，要在里面进行上锁或者用信号量什么的，保证printf不会同时被多个任务调用。

#include "semphr.h" //二值信号量头文件

#include <stdarg.h> //printf调用的是vprintf，而vprintf需要va\_list类型，所以包含该头文件

给printf专门创建个二值信号量

SemaphoreHandle\_t semDebug = NULL;

void debug\_printf(const char \*fmt, ...)

{

给printf是调用的vprintf，vprintf才去调用fput(),所以干脆直接把vprintf拿出来修改，给vprintf专门定义个打印函数

va\_list args;

if(xSemaphoreTake(semDebug,5)==pdFALSE)return;

va\_start(args,fmt);

vprintf(fmt,args);

va\_end(args);

xSemaphoreGive(semDebug);

}

这个vprintf使用主要的区别就是我加入了信号量，但是你发现没，debug\_printf执行的第2行就要获取信号量，如果这时候我没有信号量，程序就会阻塞，所以我们在程序初始化的时候一定要先释放信号量(信号量写1)

//定义任务控制块

static TaskHandle\_t Task1TCB;

static TaskHandle\_t Task2TCB;

QueueHandle\_t xQueue; //创建一个消息队列句柄

void Task1(void)

{

uint8\_t Receive\_data; //任务2发来的数据需要存放，该变量就是用来存放的

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{

xReturn = xQueueReceive(xQueue,&Receive\_data,portMAX\_DELAY); //接收任务2发来的数据

if (pdTRUE == xReturn)

debug\_printf("Task1 Receive = %d\r\n",Receive\_data);

else

printf全部改为debug\_printf就能解决打印乱序的问题

debug\_printf("Task1 Receive failed\r\n");

}

}

void Task2(void)

{

uint8\_t send\_data = 50; //要发送给任务1的数据

BaseType\_t xReturn = pdPASS;// 定义一个信息返回值来判断消息是否发送成功，默认为 pdPASS

xQueue = xQueueCreate(10,sizeof(char));// 创建消息队列，消息队列有10个空间，每个空间存放一个char大小的数据

while(1)

{

xReturn = xQueueSend( xQueue , &send\_data, 0 ); // 消息发送 等待时间 0

if (pdPASS == xReturn)

debug\_printf("Task2 Queue send success\r\n");

else

debug\_printf("Task2 Queue send failed\r\n");

vTaskDelay(1000);

}

}

int main(void)

{

BaseType\_t xReturn1,xReturn2;

RCC\_configuration();//初始化时钟

USART\_config(115200);//初始化串口

NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4); //STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为： 0~15

//优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，

//都统一用这个优先级分组，千万不要再分组，切忌。

这里就是程序初始化释放信号量的地方

printf("xxxzzzzzz\r\n");

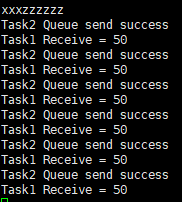
semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量

xSemaphoreGive(semDebug); //必须在初始化前就给先向semDebug变量里面写入1，因为debug\_printf函数最后一行才会给semDebug写1

创建任务1….

创建任务2….

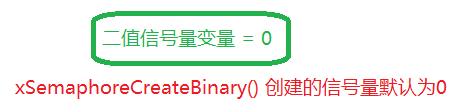
}

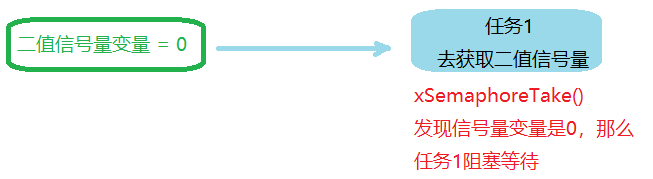
打印输出正常了。

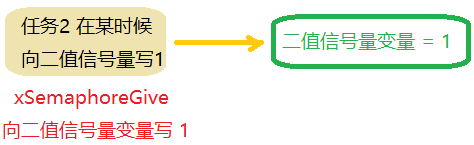
所以以后如果加入了FreeRTOS或者其它什么操作系统，使用debug\_printf(….)函数来打印。

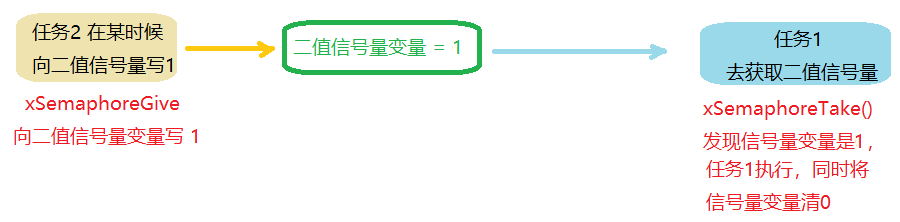
**信号量**

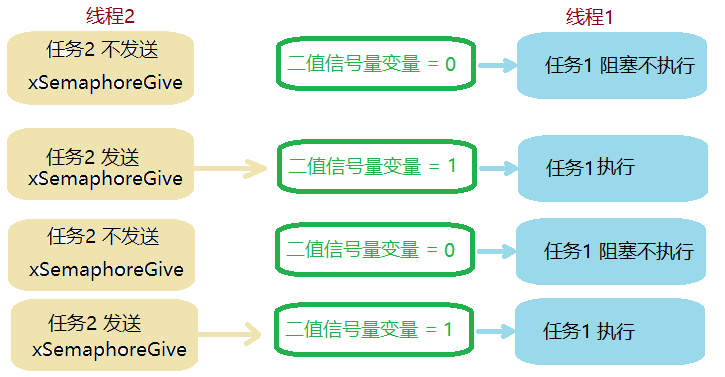
二值信号量











所以二值信号量就是实现任务间同步执行的功能，也就是任务2夺一下，任务1跳一下，任务2夺一下，任务1跳一下，就是这个意思。

typedef QueueHandle\_t SemaphoreHandle\_t; //定义信号量句柄，信号量发送和接收都靠这个句柄识别

SemaphoreHandle\_t xSemaphoreCreateBinary() //创建二值信号量变量，默认为0

BaseType\_t xSemaphoreGive( QueueHandle\_t xSemaphore ) //发送信号量，向信号量变量写1

xSemaphore: 传入信号量句柄，就表示信号量是发送给这个句柄的

BaseType\_t: 返回pdTRUE发送成功，返回pdFALSE发送失败

BaseType\_t xSemaphoreTake( SemaphoreHandle\_t xSemaphore, TickType\_t xBlockTime )

//信号量接收函数

xSemaphore: 传入信号量句柄，查看这个句柄里面的信号量变量是否为1，为1该任务不阻塞，为0该任务阻塞。

XBlockTime: 如果信号量变量为0，是否要一直阻塞下去，portMAX\_DELAY就是无限阻塞，如果xBlockTime 填0就是不阻塞，程序继续向下执行，如果填20就是阻塞20个时钟周期然后程序继续向下执行。

代码示例：

#include "semphr.h" //二值信号量头文件

SemaphoreHandle\_t xSemaphore = NULL; //创建一个空信号量句柄

void Task1(void)

{

BaseType\_t xReturn = pdPASS; //信号量获取状态返回值

while(1)

{

xReturn = xSemaphoreTake(xSemaphore , portMAX\_DELAY); //portMAX\_DELAY该任务永久阻塞，等待任务2发送过来信号量才执行

if (pdTRUE == xReturn)

debug\_printf("Task1 Sem get success\r\n"); //任务1获取到信号量

else

debug\_printf("Task1 Sem get failed\r\n");

}

}

void Task2(void)

{

BaseType\_t xReturn = pdFALSE; //返回二值信号量是否发送成功

xSemaphore = xSemaphoreCreateBinary(); //创建1个二值信号量，返回句柄，现在变量为0

if(NULL !=xSemaphore)

printf("Task2 Binar Sem Creat success\r\n"); //二值信号量创建成功

else

printf("Task2 Binar Sem Creat failed\r\n");

while(1)

{

xReturn = xSemaphoreGive( xSemaphore ); //发送信号量，也就是向变量+1

if ( xReturn == pdTRUE )

debug\_printf("Task2 Sem send success\r\n"); //信号量发送成功

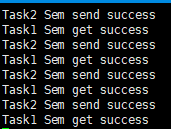
else

debug\_printf("Task2 Sem send failed\r\n"); //debug\_printf是经过处理的printf，上一节有讲

vTaskDelay(1000);

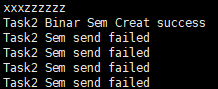
}

}

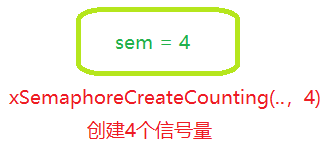
这就是二值信号量，Task2任务夺一下，Task1任务跳一下。

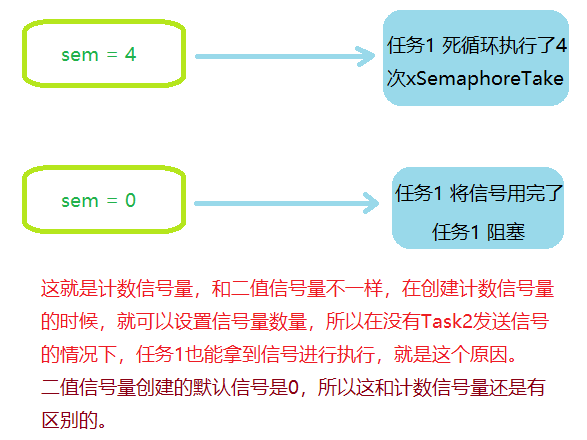


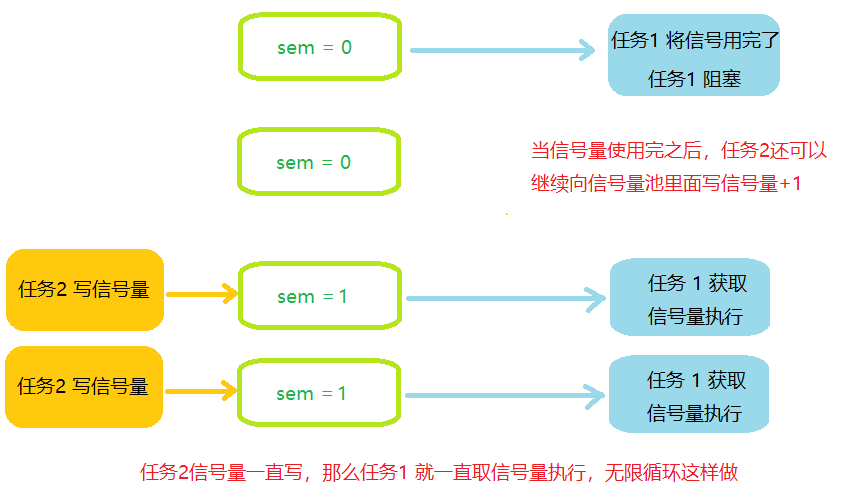
任务2停止发送信号量

 Task1 任务就会卡在xSemaphoreTake阻塞等待。

计数信号量







使用计数信号量记得在FreeRTOSconfig.h里#define configSUPPORT\_DYNAMIC\_ALLOCATION 1//打开动态内存分配，和 #define configUSE\_COUNTING\_SEMAPHORES 1 //启用计数信号量

SemaphoreHandle\_t xSemaphoreCreateCounting(UBaseType\_t uxMaxCount , UBaseType\_t uxInitialCount ) //创建计数信号量

uxMaxCount: 计数信号量最大值，主要是在信号量使用完之后，在没有程序使用xSemaphoreGive 消费信号量的情况下，xSemaphoreGive发送信号最多能向信号量池写几次。

uxInitialCount: 创建计数信号量时，向信号量池准备几个信号

发送信号量和接收信号量函数和上一节一样

#include "semphr.h" //二值信号量头文件

SemaphoreHandle\_t CountSem; //计数信号量创建

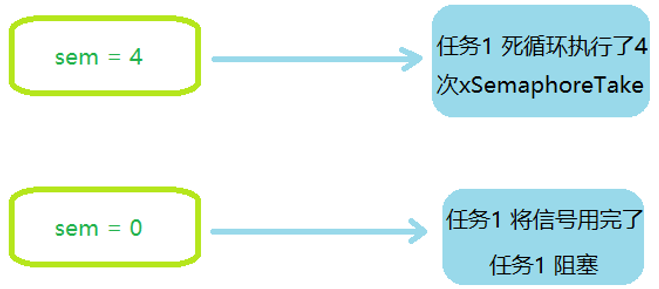
void Task1(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{



xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task1 get Count sem success \r\n");

else

debug\_printf("Task1 get Count sem failed\r\n");

}

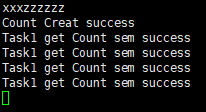
}

void Task2(void)

{

CountSem = xSemaphoreCreateCounting(5,4); //计数信号量最大值是5，初始值为4

if(CountSem != NULL)



debug\_printf("Count Creat success\r\n");//计数信号量创建成功

else

debug\_printf("Count Creat failed\r\n");

while(1)

{

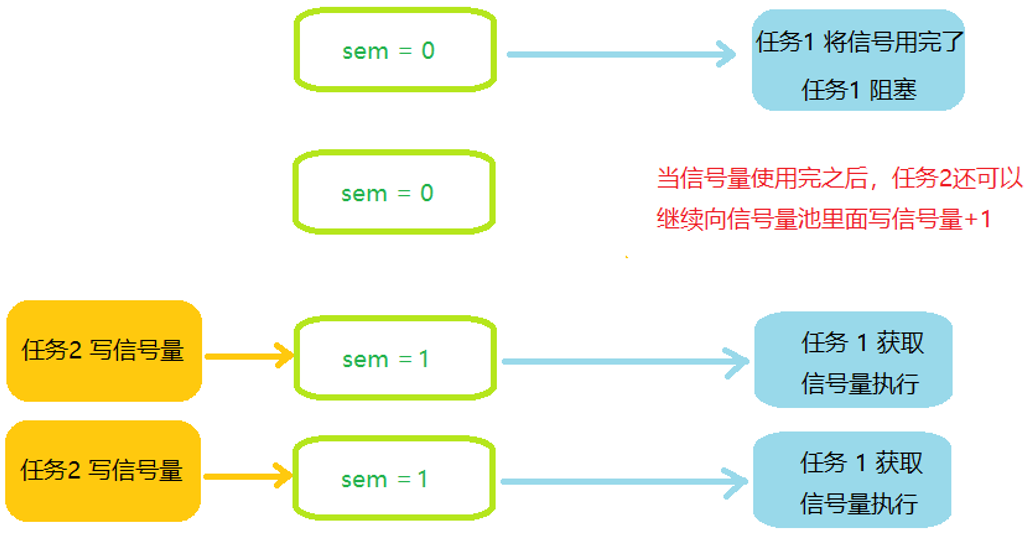
vTaskDelay(1000);

}

}

在没有Task2任务发送信号的情况下，Task1任务可以直接获取创建好的信号量。

SemaphoreHandle\_t CountSem; //计数信号量创建



void Task1(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{

xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task1 get Count sem success \r\n");

else

debug\_printf("Task1 get Count sem failed\r\n");

}

}

void Task2(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断发送是否会成功

CountSem = xSemaphoreCreateCounting(5,4); //计数信号量最大值是5，初始值为4



if(CountSem != NULL)

debug\_printf("Count Creat success\r\n");//计数信号量创建成功

else

任务1一次性消费完信号量之后，如果任务2继续发送，那么任务1还可以继续消费信号量

debug\_printf("Count Creat failed\r\n");

while(1)

{

xReturn = xSemaphoreGive(CountSem);

if (pdTRUE == xReturn)

debug\_printf("Task2 send Count sem success \r\n");

else

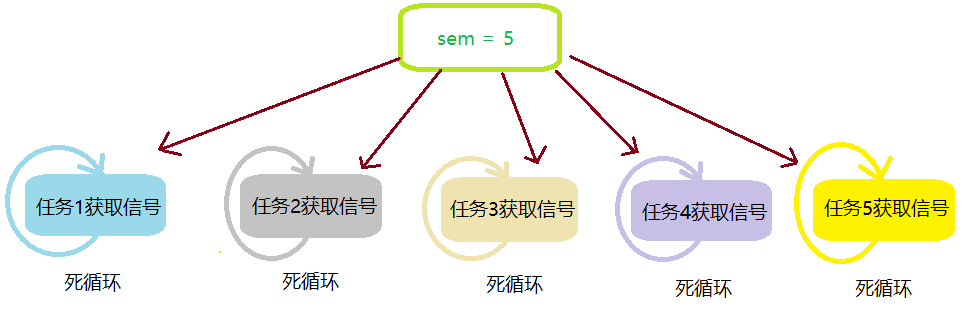
debug\_printf("Task2 send Count sem failed\r\n");

vTaskDelay(1000);

}

}

这样看来其实计数信号量意义不是很大？计数信号量主要是用于多任务。



计数信号量主要用于多任务同时获取同一个信号来处理同一个事件，但是这样死循环真的能每个任务都能获取到信号吗？下面代码验证下

SemaphoreHandle\_t CountSem; //计数信号量创建

void Task1(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{

xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task1 get Count sem success \r\n");

else

debug\_printf("Task1 get Count sem failed\r\n");

}

}

void Task3(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{

xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task3 get Count sem success \r\n");

else

debug\_printf("Task3 get Count sem failed\r\n");

}

}

void Task4(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{

xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task4 get Count sem success \r\n");

else

debug\_printf("Task4 get Count sem failed\r\n");

}

}

void Task5(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{

xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task5 get Count sem success \r\n");

else

debug\_printf("Task5 get Count sem failed\r\n");

}

}

void Task2(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断发送是否会成功

CountSem = xSemaphoreCreateCounting(5,6); //计数信号量最大值是5，初始值为4

if(CountSem != NULL)

debug\_printf("Count Creat success\r\n");//计数信号量创建成功

else

debug\_printf("Count Creat failed\r\n");

while(1)

{

xReturn = xSemaphoreGive(CountSem);

if (pdTRUE == xReturn)

debug\_printf("Task2 send Count sem success \r\n");

else

debug\_printf("Task2 send Count sem failed\r\n");

vTaskDelay(1000);

}

}

int main(void)

{

BaseType\_t xReturn1,xReturn2,xReturn3,xReturn4,xReturn5;

RCC\_configuration();//初始化时钟

USART\_config(115200);//初始化串口

NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4); //STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为： 0~15

//优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，

//都统一用这个优先级分组，千万不要再分组，切忌。

printf("xxxzzzzzz\r\n");

semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量

xSemaphoreGive(semDebug); //必须在初始化前就给先向semDebug变量里面写入1，因为debug\_printf函数最后一行才会给semDebug写1

xReturn1 = xTaskCreate((TaskFunction\_t)Task1, //任务函数地址

(const char\* )"Task1", //任务名

(uint32\_t)128, //任务栈大小

(void\*)NULL, //向函数传参

(UBaseType\_t)1, //优先级

(TaskHandle\_t\* )&Task1TCB); //任务控制块

xReturn2 = xTaskCreate((TaskFunction\_t)Task2,

(const char\* )"Task2",

(uint32\_t)128,

(void\*)NULL,

(UBaseType\_t)7,

(TaskHandle\_t\* )&Task2TCB); //因为任务2是发送消息的，所以优先级一定要高于其它任务

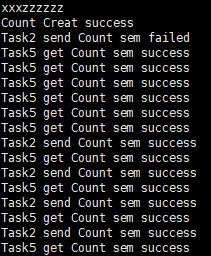
不然你会发现任务启动后程序卡死了，有可能是接收消息的任务优先级高了，导致先执行，但是这时候Task2没有先执行，没有创建信号量，所以程序卡死。

创建任务3

创建任务4

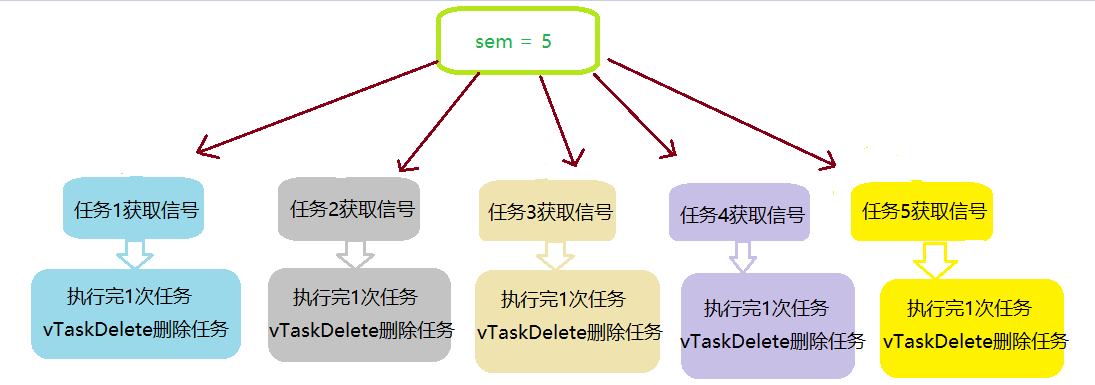
创建任务5

}

 发现只有Task2和Task5在运行。这是为什么呢？因为Task5是接收信号量的任务，除了Task2优先级最高外，Task5优先级比Task1,3,4都高，所以Task5一直获取信号。

如果取消while循环呢？其实也不行，因为任务函数就是要求无限循环的,所以你会发现启动之后程序又再次卡死到Task5线程了。除非将任务删除。

SemaphoreHandle\_t CountSem; //计数信号量创建



void Task1(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task1 get Count sem success \r\n");

else

debug\_printf("Task1 get Count sem failed\r\n");

vTaskDelete(Task1TCB); //删除任务

}

void Task3(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task3 get Count sem success \r\n");

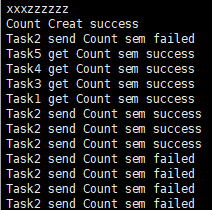
else

debug\_printf("Task3 get Count sem failed\r\n");

vTaskDelete(Task3TCB); //删除任务

任务创建后每个任务都能获得一次信号进行执行，这是因为每个任务执行后都把自己删除了

}



void Task4(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task4 get Count sem success \r\n");

else

debug\_printf("Task4 get Count sem failed\r\n");

vTaskDelete(Task4TCB); //删除任务

}

void Task5(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

xReturn = xSemaphoreTake(CountSem,portMAX\_DELAY);

if (pdTRUE == xReturn)

debug\_printf("Task5 get Count sem success \r\n");

else

debug\_printf("Task5 get Count sem failed\r\n");

vTaskDelete(Task5TCB); //删除任务

}

void Task2(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断发送是否会成功

CountSem = xSemaphoreCreateCounting(5,6); //计数信号量最大值是5，初始值为4

if(CountSem != NULL)

debug\_printf("Count Creat success\r\n");//计数信号量创建成功

else

debug\_printf("Count Creat failed\r\n");

while(1)

{

xReturn = xSemaphoreGive(CountSem);

if (pdTRUE == xReturn)

debug\_printf("Task2 send Count sem success \r\n");

else

debug\_printf("Task2 send Count sem failed\r\n");

vTaskDelay(1000);

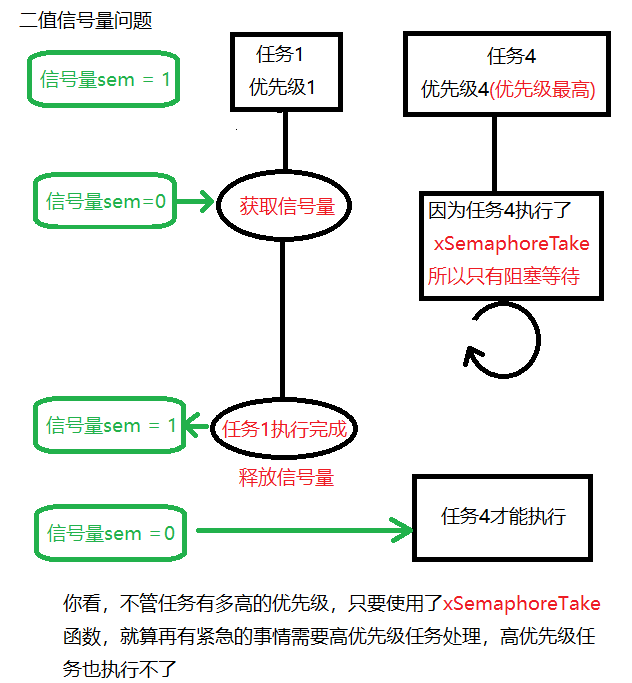
}

}

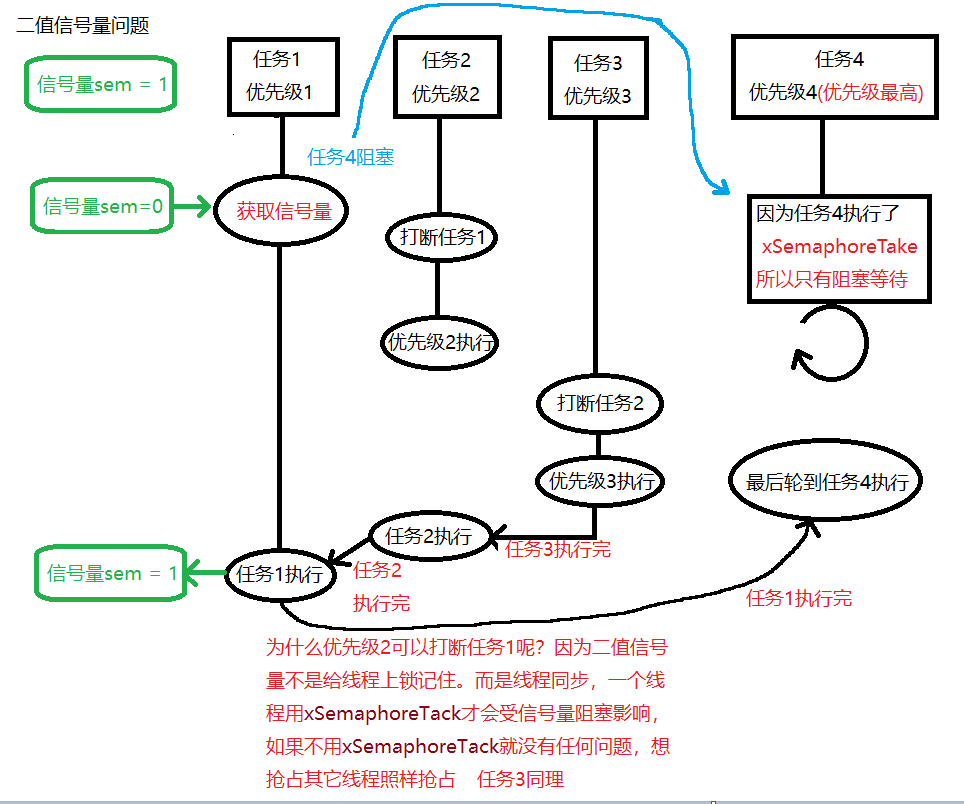
这种创建多个任务，每个任务只执行一次的场景，而且必须在某时某刻准点执行的场景，用计数信号量才有意义。一般都是先创建任务后就进行阻塞，等待某时某刻发送信号量，一次性执行完多个任务。

**互斥量**

解决二值信号量优先级翻转问题

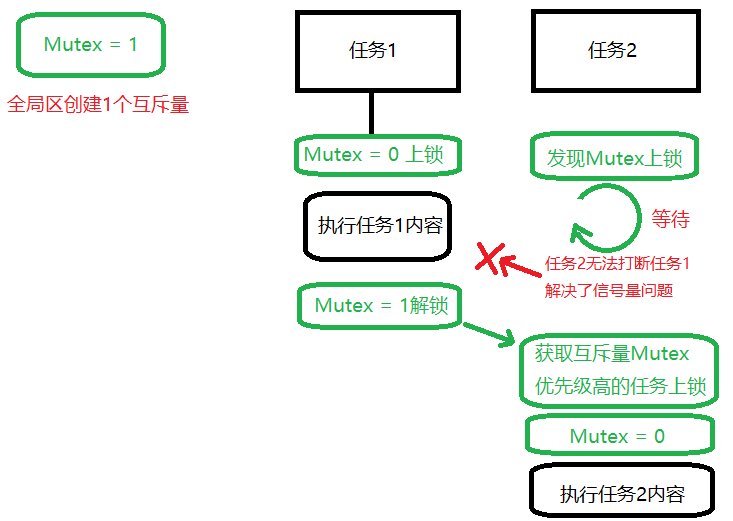


还有一种更恶劣的情况



这种一个最高优先级的紧急任务要等前面3个低优先级任务执行完才能执行，这就是优先级翻转，更恐怖。

所以引入了互斥锁(互斥量)



如果任务2再定义一个新的互斥量那么就不用等待任务1了，但是这也没有意义了啊，互斥量一定是一个变量和多个任务使用才有临界区保护能力

在FreeRTOSconfig.h里#define configUSE\_MUTEXES 1 //互斥量使用

SemaphoreHandle\_t xSemaphoreCreateMutex() //创建互斥量，返回句柄，句柄就是要操作的互斥量

BaseType\_t xSemaphoreGive( SemaphoreHandle\_t xSemaphore ) //发送互斥量(用于解锁)

xSemaphore： 传入要发送的互斥量

BaseType\_t xSemaphoreTake(SemaphoreHandle\_t xSemaphore,xTicksToWait) //接收互斥量

XSemaphore: 传入要接收的互斥量

XTicksToWait: 超时时间设置

代码示例:

#include "semphr.h" //互斥量就是继承的二值信号量，所以用信号量头文件

SemaphoreHandle\_t xMutex; //定义互斥量句柄

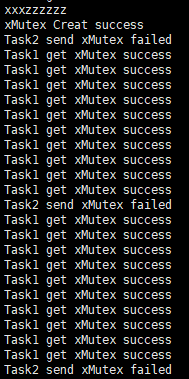
void Task1(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{ xReturn = xSemaphoreTake(xMutex,portMAX\_DELAY); //获取互斥量上锁



if (pdTRUE == xReturn)

debug\_printf("Task1 get xMutex success \r\n");

为什么任务2发送失败呢？因为互斥量创建之后就产生了信号，任务1直接使用就是了。又不是信号量创建时候为空，需要先发送信号

else

debug\_printf("Task1 get xMutex failed\r\n");

xReturn = xSemaphoreGive(xMutex); //发送互斥量解锁

vTaskDelay(100);

}

}

void Task2(void)

{ BaseType\_t xReturn = pdTRUE; //用来判断发送是否会成功

xMutex = xSemaphoreCreateMutex();//一定要先创建互斥量，其它任务才可以使用

if(xMutex != NULL)

debug\_printf("xMutex Creat success\r\n");//互斥量创建成功

else

debug\_printf("xMutex Creat failed\r\n");

while(1)

{ xReturn = xSemaphoreGive(xMutex); //一定要先发送1次互斥量

if (pdTRUE == xReturn)

debug\_printf("Task2 send xMutex success \r\n");

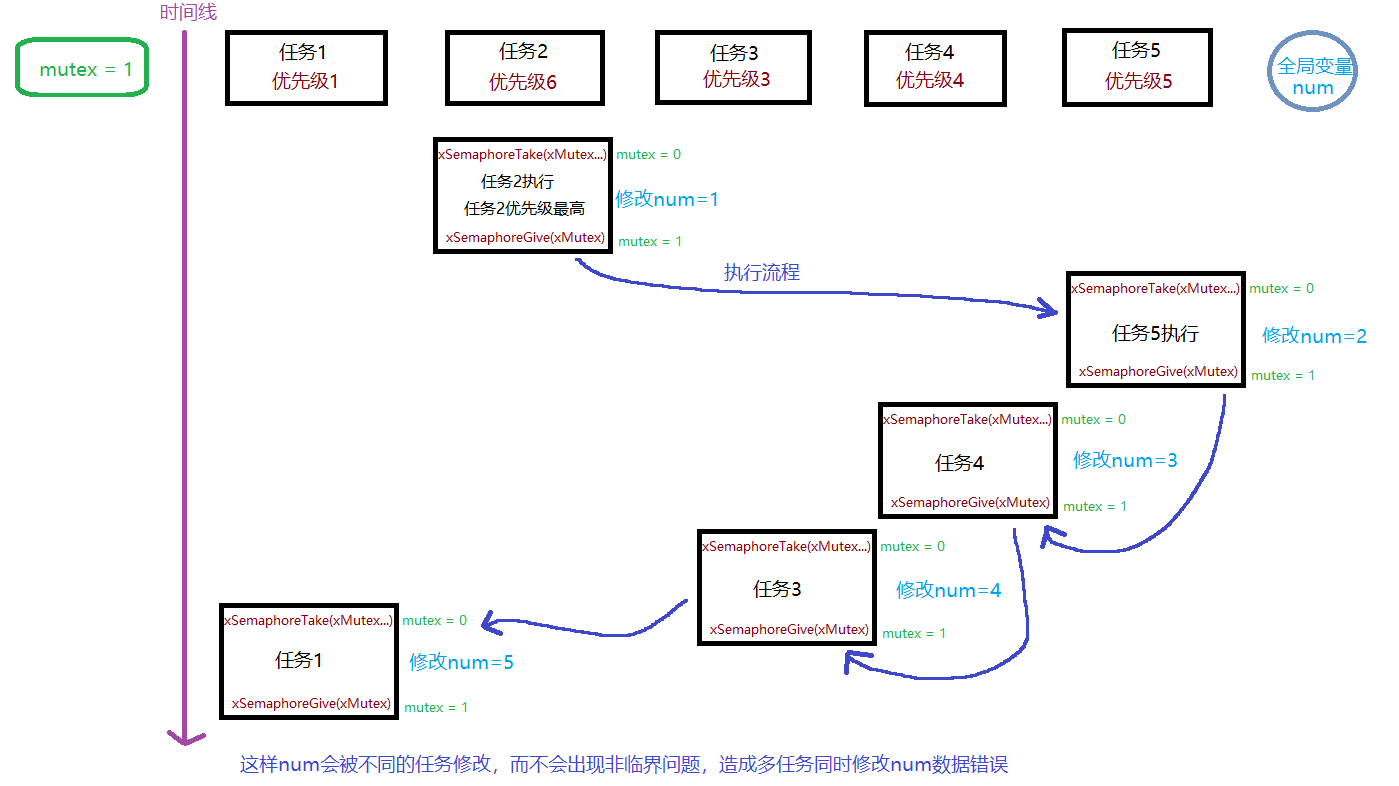
else

debug\_printf("Task2 send xMutex failed\r\n");

vTaskDelay(1000);

}

}



代码示例:

SemaphoreHandle\_t xMutex; //定义互斥量句柄

static int num = 0; //所有任务都要修改这个变量

void Task3(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1){

xReturn = xSemaphoreTake(xMutex,portMAX\_DELAY); //获取互斥量

num++;

if (pdTRUE == xReturn)

debug\_printf("Task3 get xMutex success num = %d \r\n",num);

else

debug\_printf("Task3 get xMutex failed\r\n");

xReturn = xSemaphoreGive(xMutex); //发送互斥量解锁

vTaskDelay(1000);

}

}

void Task4(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1){

xReturn = xSemaphoreTake(xMutex,portMAX\_DELAY); //获取互斥量

num++;

if (pdTRUE == xReturn)

debug\_printf("Task4 get xMutex success num = %d \r\n",num);

else

debug\_printf("Task4 get xMutex failed\r\n");

xReturn = xSemaphoreGive(xMutex); //发送互斥量解锁

vTaskDelay(1000);

}

}

void Task5(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{

xReturn = xSemaphoreTake(xMutex,portMAX\_DELAY); //获取互斥量

num++;

if (pdTRUE == xReturn)

debug\_printf("Task5 get xMutex success num = %d \r\n",num);

else

debug\_printf("Task5 get xMutex failed\r\n");

xReturn = xSemaphoreGive(xMutex); //发送互斥量解锁

vTaskDelay(1000);

}

}

void Task1(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断接收是否会成功

while(1)

{

xReturn = xSemaphoreTake(xMutex,portMAX\_DELAY); //获取互斥量上锁

num++;

if (pdTRUE == xReturn)

debug\_printf("Task1 get xMutex success num = %d \r\n",num);

else

debug\_printf("Task1 get xMutex failed\r\n");

xReturn = xSemaphoreGive(xMutex); //发送互斥量解锁

vTaskDelay(1000);

}

}

void Task2(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断发送是否会成功

xMutex = xSemaphoreCreateMutex();//创建1个互斥量，多个任务使用

if(xMutex != NULL)

debug\_printf("xMutex Creat success\r\n");//互斥量创建成功

else

debug\_printf("xMutex Creat failed\r\n");

while(1)

{

xReturn = xSemaphoreTake(xMutex,portMAX\_DELAY); //获取互斥量上锁

num++;

if (pdTRUE == xReturn)

debug\_printf("Task2 send xMutex success num = %d \r\n",num);

else

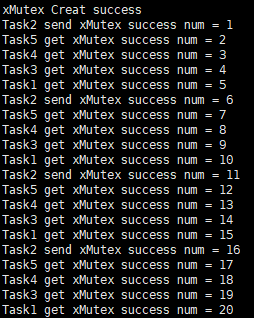
debug\_printf("Task2 send xMutex failed\r\n");

xReturn = xSemaphoreGive(xMutex); //发送互斥量

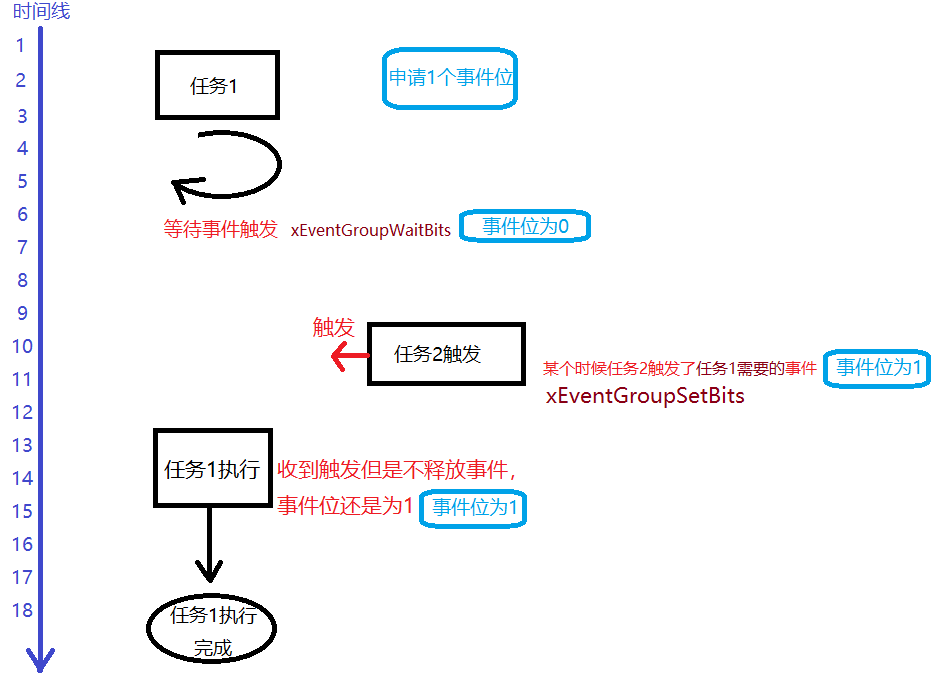
vTaskDelay(1000);

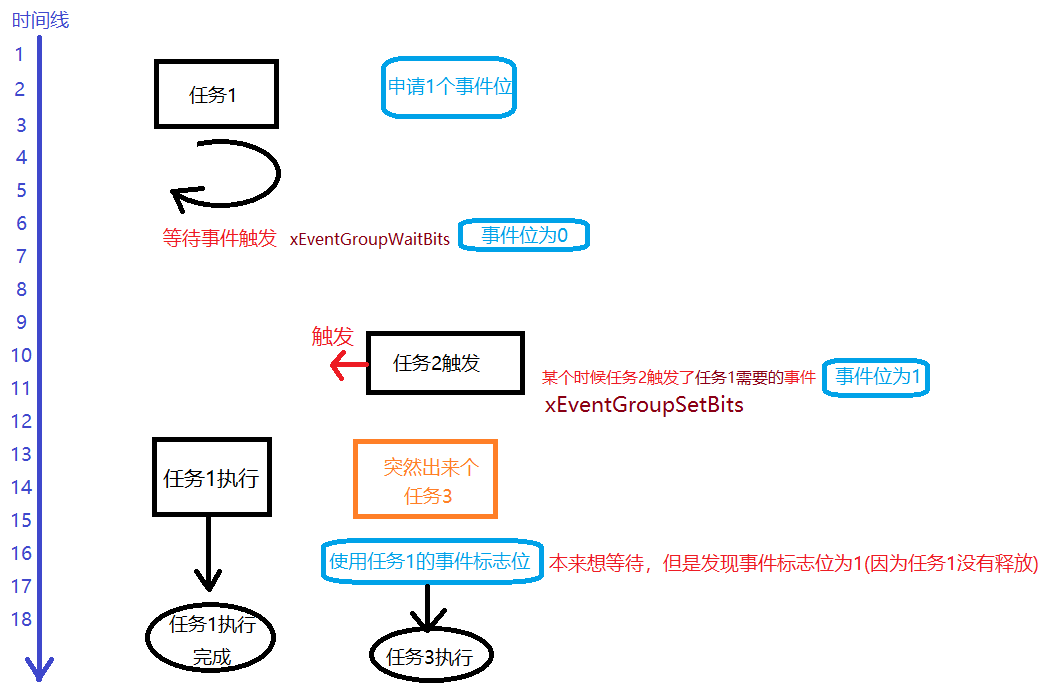
}

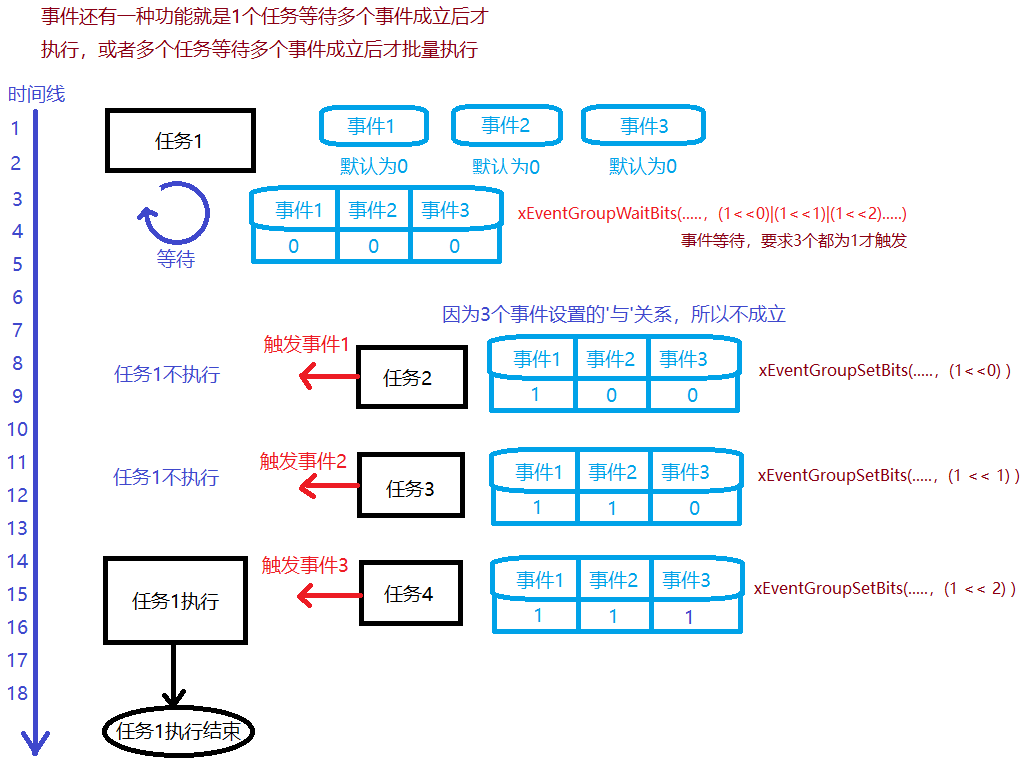
}



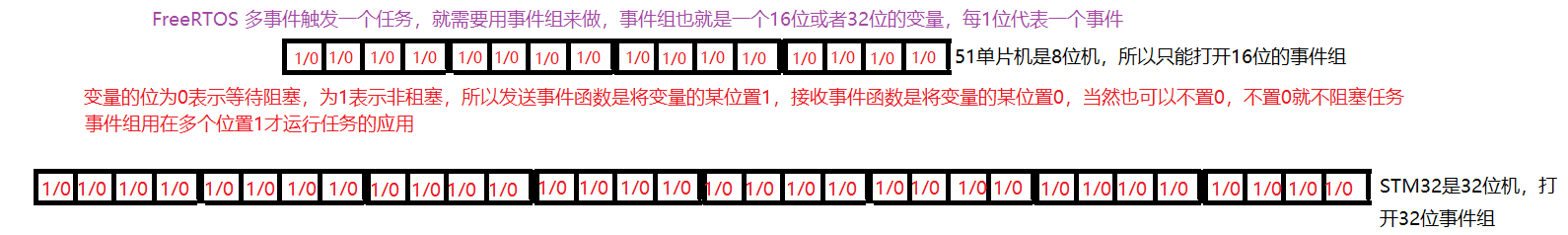
**事件使用，解决任务同步**







这就是事件 ’与 ’ 实现的多事件触发一个任务。比如有些系统要求，电源事件稳定，环境事件稳定，时钟事件稳定，三个变量同时稳定之后才能执行，这个时候就可以用事件。事件’或 ’就是任何1个事件置1任务都可以运行。



事件和信号量区别:

信号量可以传递参数，但是信号量只能一对一触发。

事件无法传递参数，但是可以做为多个事件触发。当多个事件同时满足时才执行某个人物，用于多情况判定。

那为什么不用变量做事件呢？，因为变量你需要去轮询查询变量的状态，如果我while(1)循环一次轮询为20ms毫秒一个周期，如果紧急事件需要1ms之内就必须处理，你怎么办? 而且变量同一时间多个任务都在写该变量，那么变量的值到底是多少呢？所以事件是最好的选择。

注意: 一个16位的事件组只能用低8位，

注意: 一个32位的事件组只能用低24位。

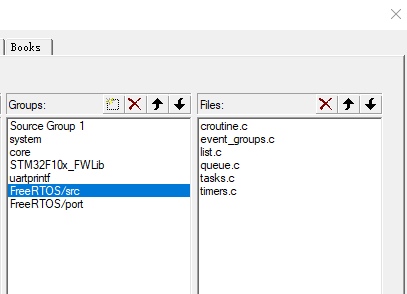


#define configUSE\_16\_BIT\_TICKS 0 //系统最大节拍数32位 ，也就是uxEventBits事件标志位有32位

#define configUSE\_16\_BIT\_TICKS 1//系统最大节拍数16位 , ，也就是uxEventBits事件标志位有16位

我是STM32单片机，32位的，我选着0,事件标志位32位，如果是51单片机可以选择1，16位事件位。

要想使用事件必须加入FreeRTOS/source/event\_groups.c这个文件到工程

我这里加入了的

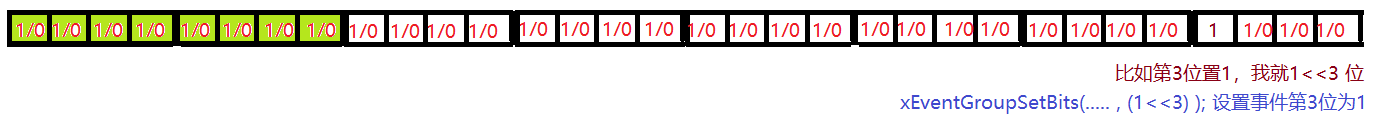
typedef void \* EventGroupHandle\_t //定义事件句柄

EventGroupHandle\_t xEventGroupCreate( void ) //创建事件组

EventBits\_t xEventGroupSetBits( EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToSet ) //发送事件函数

XEventGroup: 放入事件组，就是EventGroupHandle\_t定义的句柄

uxBitsToSet: 事件组哪一位置1?



EventBits\_t xEventGroupWaitBits( EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToWaitFor, const BaseType\_t xClearOnExit, const BaseType\_t xWaitForAllBits, TickType\_t xTicksToWait )//阻塞等待事件到来

xEventGroup: 事件句柄，用来区分谁发送的事件

uxBitsToWaitFor: 指定需要等待事件组哪些位为1才能越过阻塞状态。

如果需要3位为1，就填入1<<3位

如果需要多位为1才取消阻塞，比如bit5，bit6为1才取消阻塞，就填入（1<<5）| (1<<6)

xClearOnExit: 等待事件到来之后，要不要把相应的位清0，pdTRUE系统清除事件标志位，pdFALSE系统不清楚事件标志位。

xWaitForAllBits: 在多事件位触发中，选择pdTRUE 就是uxBitsToWaitFor里面的位都满足了(与关系)，才取消阻塞。选择pdFALSE就是uxBitsToWaitFor里面的位只要满足1位置1就取消阻塞(或关系)

xTicksToWait: 填入阻塞的时间，时间到了，就算事件没有触发，也要取消阻塞继续执行。

static EventGroupHandle\_t Event\_Handle = NULL; //事件句柄

#define EVENTbit\_0 (1<<0)

#define EVENTbit\_1 (1<<1)

#define EVENTbit\_2 (1<<2)

void Task1(void)

{

EventBits\_t EventValue; //获取事件等待返回的结果

while(1)

{

EventValue = xEventGroupWaitBits(Event\_Handle,EVENTbit\_1,pdTRUE,pdTRUE,portMAX\_DELAY);

//阻塞等待事件1 EVENTbit\_1被触发

debug\_printf("Task1 wait EVENTbit\_1 Event arrival = 0x%x\r\n",EVENTbit\_1);

//等待事件1 EVENTbit\_1发生成功，执行打印

vTaskDelay(1000);

}

}

void Task2(void)

{

BaseType\_t xReturn = pdTRUE; //用来判断发送是否会成功

Event\_Handle = xEventGroupCreate();//创建事件

if(Event\_Handle != NULL)

debug\_printf("Event success\r\n");//事件创建成功

else

debug\_printf("Event failed\r\n");

xEventGroupSetBits(Event\_Handle,EVENTbit\_1); //向事件组第1位置1

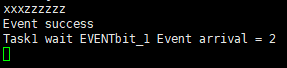
while(1)

{

vTaskDelay(1000);

}

}



如果要Task1不停的执行，那么Task2要不停的发送事件

void Task2(void)

{

Event\_Handle = xEventGroupCreate();//创建事件

if(Event\_Handle != NULL)

debug\_printf("Event success\r\n");//事件创建成功

else

debug\_printf("Event failed\r\n");

while(1)

{

xEventGroupSetBits(Event\_Handle,EVENTbit\_1); //放入循环，向事件组第1位置1

vTaskDelay(1000);

}

}



多事件触发才能执行案例

void Task1(void)

{

EventBits\_t EventValue; //获取事件等待返回的结果

while(1)

{

EventValue = xEventGroupWaitBits(Event\_Handle,EVENTbit\_0|EVENTbit\_1,pdTRUE,pdTRUE,portMAX\_DELAY);

//阻塞等待事件1 EVENTbit\_1, 和事件0 EVENTbit\_0 同时触发才结束阻塞

debug\_printf("Task1 wait EVENTbit\_1 Event arrival = 0x%x\r\n",EVENTbit\_1);

//等待事件1 EVENTbit\_1发生成功，执行打印

vTaskDelay(1000);

}

}

void Task2(void)

{

Event\_Handle = xEventGroupCreate();//创建事件

if(Event\_Handle != NULL)

debug\_printf("Event success\r\n");//事件创建成功

else

debug\_printf("Event failed\r\n");

while(1)

{

xEventGroupSetBits(Event\_Handle,EVENTbit\_1); //向事件组第1位置1

vTaskDelay(1000);

}

}



就算Task2发送事件也没有用，因为只发送了EVENTbit\_1 事件1没有发送事件0

void Task2(void)

{

Event\_Handle = xEventGroupCreate();//创建事件

if(Event\_Handle != NULL)

debug\_printf("Event success\r\n");//事件创建成功

else

debug\_printf("Event failed\r\n");

while(1)

{

xEventGroupSetBits(Event\_Handle,EVENTbit\_0 | EVENTbit\_1); //向事件组第0位置1，第1位置1，发送两个事件

vTaskDelay(1000);

}

}



多事件也可以多个任务分开来发送

static EventGroupHandle\_t Event\_Handle = NULL; //事件句柄

#define EVENTbit\_0 (1<<0)

#define EVENTbit\_1 (1<<1)

#define EVENTbit\_2 (1<<2)

void Task3(void)

{

while(1){

xEventGroupSetBits(Event\_Handle,EVENTbit\_0); //向事件组第0位置1

vTaskDelay(1000);

}

}

void Task1(void)

{

EventBits\_t EventValue; //获取事件等待返回的结果

while(1)

{

EventValue = xEventGroupWaitBits(Event\_Handle,EVENTbit\_0|EVENTbit\_1,pdTRUE,pdTRUE,portMAX\_DELAY);

//阻塞等待事件1 EVENTbit\_1, 和事件0 EVENTbit\_0 同时触发才结束阻塞

debug\_printf("Task1 wait EVENTbit\_1 Event arrival = 0x%x\r\n",EVENTbit\_1);

//打印有多少个事件被置1，用16进制表示，可以翻译成2进制

//等待事件1 EVENTbit\_1发生成功，执行打印

vTaskDelay(1000);

}

}

void Task2(void)

{

Event\_Handle = xEventGroupCreate();//创建事件

if(Event\_Handle != NULL)

debug\_printf("Event success\r\n");//事件创建成功

else

debug\_printf("Event failed\r\n");

while(1)

{

xEventGroupSetBits(Event\_Handle,EVENTbit\_1); //向事件组第1位置1

vTaskDelay(1000);

}

}



**软件定时器**

软件定时器只适用于1ms毫秒左右的定时单位，对定时精度要求不高的可以用软件定时器。

定时要求高的，比如1us微妙的定时就不适合软件定时器。

在FreeRTOSconfig.h中启动定时器配置选项

#define configUSE\_TIMERS 1 //启动软件定时器

#define configTIMER\_TASK\_STACK\_DEPTH 128 //定时器任务堆栈大小

#define configTIMER\_TASK\_PRIORITY (configMAX\_PRIORITIES - 1) //软件定时器优先级最大值-1，优先级最高

#define configTIMER\_QUEUE\_LENGTH 10 //软件定时器队列长度

typedef void \* TimerHandle\_t //定义定时器句柄

typedef void (\*TimerCallbackFunction\_t)( TimerHandle\_t xTimer ) //定时器回调函数类型

TimerHandle\_t xTimerCreate( const char \* const pcTimerName,

const TickType\_t xTimerPeriodInTicks,

const UBaseType\_t uxAutoReload,

void \* const pvTimerID,

TimerCallbackFunction\_t pxCallbackFunction ) //创建定时器

pcTimerName: 给定时器取名字

xTimerPeriodInTicks: 定时器延时周期，最小单位1ms毫秒

uxAutoReload: 定时器工作模式 pdTRUE为周期模式，就是定时器一直定时执行，pdFALSE为单次模式，定时器只定时执行一次。

pvTimerID: 为定时器分配一个索引ID，随便取，只要不重复就行

pxCallbackFunction: 定时器回调函数，定时时间到，执行的函数体

xTimerStart( xTimer, xTicksToWait ) //启动定时器

xTimer: 传入定时器句柄

xTicksToWait: 默认值为0，定时器为正常工作模式

xTimerStartFromISR( xTimer, pxHigherPriorityTaskWoken ) //中断中启动定时器，该函数是xTimerStart中断版本

BaseType\_t xTimerStop( TimerHandle\_t xTimer, TickType\_t xBlockTime ) //软件定时器停止函数

xTimer: 软件定时器句柄，要停哪一个定时器就填入哪一个定时器句柄

xBlockTime: 用户指定超时时间，默认填0就是马上停止定时器工作。

xTimerStopFromISR()是函数 xTimerStop()的中断版本，在中断函数中停止软件定时器，自行查阅用法。

代码例程

#include "event\_groups.h" //加入事件头文件，事件也使用了软件定时器，所以包含事件就包含了软件定时器

static TimerHandle\_t Stimr1\_Handle =NULL; //定义软件定时器1句柄

static TimerHandle\_t Stimr2\_Handle =NULL; //定义软件定时器2句柄

static void Stimr1\_Callback(void\* parameter) //软件定时器1回调函数

{

debug\_printf("Stimr1\_Callback\r\n");

}

static void Stimr2\_Callback(void\* parameter) //软件定时器2回调函数

{

debug\_printf("Stimr2\_Callback\r\n");

}

int main(void)

{

BaseType\_t xReturn1,xReturn2,xReturn3,xReturn4,xReturn5;

RCC\_configuration();//初始化时钟

USART\_config(115200);//初始化串口

NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4); //STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为： 0~15

//优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，

//都统一用这个优先级分组，千万不要再分组，切忌。

printf("xxxzzzzzz\r\n");

semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量

xSemaphoreGive(semDebug); //必须在初始化前就给先向semDebug变量里面写入1，因为debug\_printf函数最后一行才会给semDebug写1

Stimr1\_Handle = xTimerCreate( "SFtimer1", //给软件定时器命名

(TickType\_t)5000, //定时器周期 5000(tick) 就是5000ms毫秒

(UBaseType\_t )pdTRUE, //周期模式

(void\* )1, //为定时器分配一个索引ID，随便写，不重复就是

(TimerCallbackFunction\_t)Stimr1\_Callback //回调函数

);

xTimerStart(Stimr1\_Handle,0); //启动定时器1

Stimr2\_Handle = xTimerCreate( "SFtimer2", //给软件定时器命名

(TickType\_t)1000, //定时器周期 1000(tick) 就是1000ms毫秒

(UBaseType\_t )pdTRUE, //周期模式

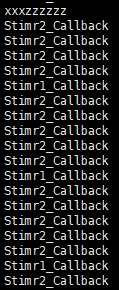
(void\* )1, //为定时器分配一个索引ID，随便写，不重复就是

(TimerCallbackFunction\_t)Stimr2\_Callback //回调函数

);

xTimerStart(Stimr2\_Handle,0); //启动定时器2

vTaskStartScheduler(); /\* 就算不创建任务，也要启动任务，开启调度，因为任务调度是个死循环，会在里面无限执行FreeRTOS里面各种函数 \*/

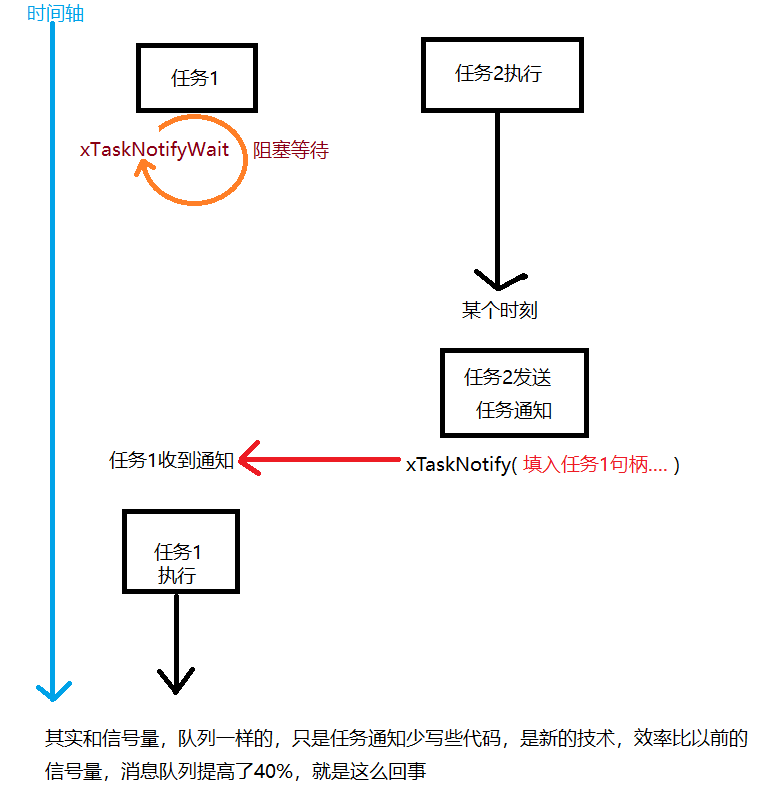
 Stimr2\_Callback 1秒执行1次 执行5次，Stimr1\_Callback就会执行1次，证明Stimr1\_Callback是5秒1次

**任务通知**

任务通知是在FreeRTOS 8.2.0版本才有的，以前版本没有，所以使用时要注意版本。

任务通知可以代替二值信号量，计数信号量，事件组，消息队列。但是不能代替互斥锁机制。

#define configUSE\_TASK\_NOTIFICATIONS 1 //任务通知开启



BaseType\_t xTaskNotify( TaskHandle\_t xTaskToNotify , uint32\_t ulValue , eNotifyAction eAction )

//发送带参数的通知给指定的任务

xtaskToNotify : 传入需要接受本通知的任务句柄

ulValue: 发送参数给指定任务

eAction: 指定任务收到通知后是怎么处理的 ?

eNoAction 指定任务接收到通知后，不接收发给指定任务的参数，返回pdFalse

eSetValueWithOverwrite 指定任务接收到通知后，强制将参数发送给指定任务，指定任务必须接收。

eSetValueWithoutOverwrite 指定任务接收通知后，如果指定任务本身的参数值没有取值，那么久丢弃发送过来的参数值，返回pdFALSE。如果指定任务的参数已经被获取了，那么指定任务就接收发送过来的参数，返回pdTRUE

BaseType\_t xTaskNotifyWait( uint32\_t ulBitsToClearOnEntry, uint32\_t ulBitsToClearOnExit, uint32\_t \*pulNotificationValue, TickType\_t xTicksToWait )

//阻塞程序，接收带参数的任务通知，放弃阻塞，继续向下执行

ulBitsToClearOnEntry: 在进入任务前，将接收的参数哪些位清0。如果设置成0x01，比如pulNotificationValue,接收到0xffffffff，那么就是1111111111111111111111111111111。

那就1111111111111111111111111111101 第1位清0，那么接收的参数就变成了7FFF FFFD。

void Task1(void)

{

uint32\_t RecvValue = 0;

BaseType\_t ret = pdFALSE;

while(1)

{

ret = xTaskNotifyWait(0,0xFFFFFFFF,&RecvValue,portMAX\_DELAY);

if(ret == pdTRUE)

debug\_printf("Task1 Recv success = %d \r\n",RecvValue);

else

debug\_printf("Task1 Recv failed \r\n");

}

}

void Task2(void)

{

BaseType\_t status;

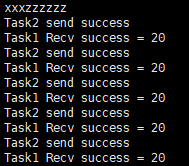
uint32\_t send = 20;

while(1)

{

status = xTaskNotify(Task1TCB,send,eSetValueWithoutOverwrite); //发送任务通知给任务1

if(status == pdPASS)



这就是Task2发送通知,Task1接收通知，并且得到通知值

debug\_printf("Task2 send success\r\n"); //发送给Task1的数据被获取

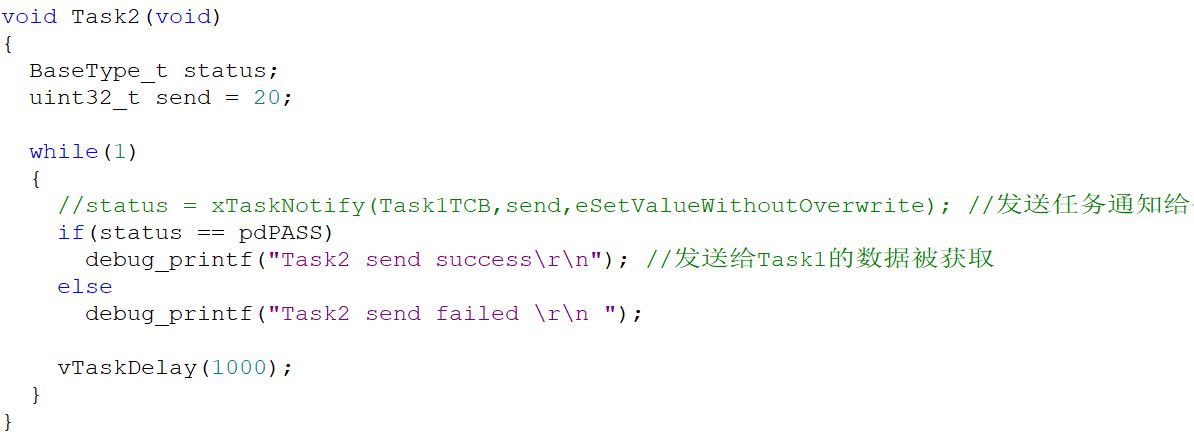
else

debug\_printf("Task2 send failed \r\n ");

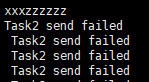
vTaskDelay(1000);

}

}



如果Task2不发送通知，那么Task1就只有阻塞，和消息队列一样的逻辑，只是任务通知不用定义太多参数，就在任务里面直接调用填参数即可。

任务1阻塞

BaseType\_t XTaskNotifyGive(TaskHandle\_t xTaskToNotify) //发送二值信号量类型的任务通知

xTaskToNotify： 传入发送给指定任务的任务句柄

uint32\_t ulTaskNotifyTake( BaseType\_t xClearCountOnExit, TickType\_t xTicksToWait )

//接收二值信号量类型的任务通知

XClearCountOnExit:设置为pdTRUE时，阻塞函数ulTaskNotifyTake退出前可以将任务通知值清0，实现二值信号量

设置为pdFALSE时，阻塞函数ulTaskNotifyTake退出前，任务通知值减1，实现计数信号量。

xTicksToWait: 设置阻塞时间

uint32\_t 返回计数值

void Task1(void)

{

uint32\_t NotifyValue = 0;

BaseType\_t ret = pdFALSE;

while(1)

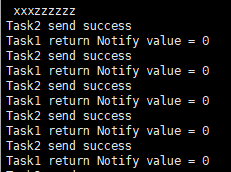
{

ret = ulTaskNotifyTake(pdTRUE,portMAX\_DELAY); //接收二值信号量类的任务通知

if(ret == pdTRUE)

debug\_printf("Task1 return Notify value = %d \r\n",NotifyValue);

else



任务1接收任务2的通知，因为是二值信号量模式，所以只有通知功能

debug\_printf("Task1 return failed \r\n");

}

}

void Task2(void)

{

BaseType\_t status;

uint32\_t send = 20;

while(1)

{

status = xTaskNotifyGive(Task1TCB); //发送类似二值信号量的任务通知给任务1

if(status == pdPASS)

debug\_printf("Task2 send success\r\n"); //发送二值任务通知给Task1

else

debug\_printf("Task2 send failed \r\n ");

vTaskDelay(1000);

}

}

ulTaskNotifyTake( pdFALSE , 0 ); //将接收任务改成计数信号量任务通知模式，pdFALSE

void Task1(void)

{

uint32\_t NotifyValue = 0; //计算Task2已经发送了多少个信号了，但是Task1还没有去获取

int i = 0;

for(i = 0; i<10; i++) //先让Task2 发10次任务通知

{

vTaskDelay(1000);

}

while(1)

{

NotifyValue = ulTaskNotifyTake(pdFALSE,0); //接收二值信号量类的任务通知，应该累计了10个通知返回

if(NotifyValue > 0)

debug\_printf("Task1 return Notify value = %d \r\n",NotifyValue);

else

debug\_printf("Task1 return failed \r\n");

vTaskDelay(1000);

}

}

void Task2(void)

{

BaseType\_t status;

uint32\_t send = 20;

while(1)

{

status = xTaskNotifyGive(Task1TCB); //发送类似二值信号量的任务通知给任务1

if(status == pdPASS)

debug\_printf("Task2 send success\r\n"); //发送二值任务通知给Task1

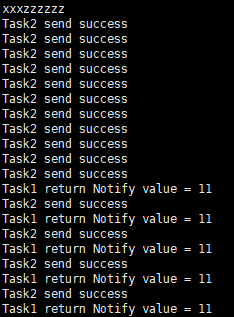
else

debug\_printf("Task2 send failed \r\n ");

vTaskDelay(1000);

}

}



你看发送了11个通知，我Task1一直没有处理，根据这个计数值，Task1在某个时刻可以依次处理，让NotifyValue每次循环-1，就可以处理完

任务通知还实现了事件组的方式，我这里就不写了，我就用传统的事件组一样的。

**内存管理**

FreeRTOS内存申请4种方式

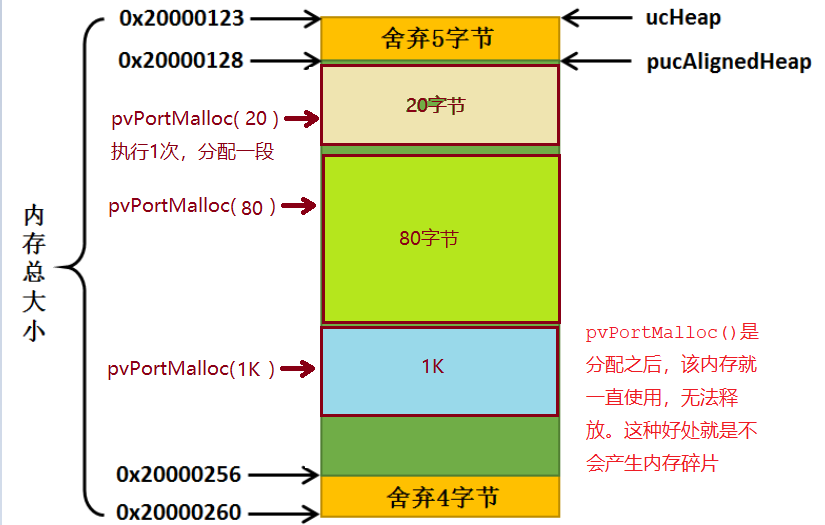
heap\_1.c

这是自己在配置文件设置的堆内存大小

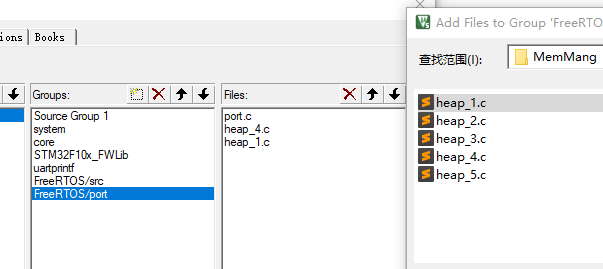
就是前面在FreeRTOSconfig.h里面设置的



void \*pvPortMalloc( size\_t xWantedSize ) //内存一次性分配函数



所以heap\_1适合那些创建好的任务，用于不会删除的任务，使用这种内存方法是比较好的。还有就是变量固定，变量都全部分配好了，不需要删除，只需要使用就行了，类似51单片机这种应用的。

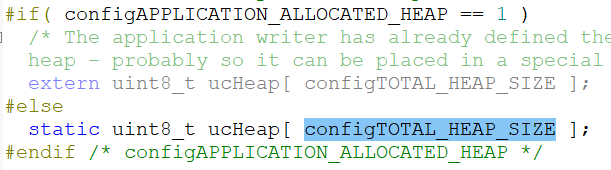
将heap\_1.c添加进工程

在portmacro.h 中



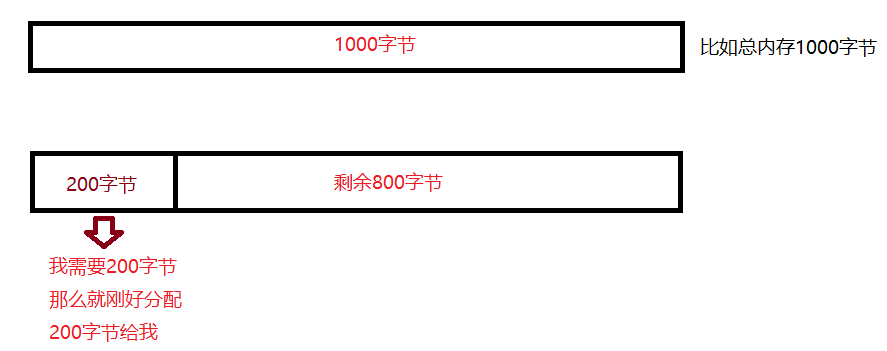
如果是填的8，那么pvPortMalloc内存分配就是8字节对齐

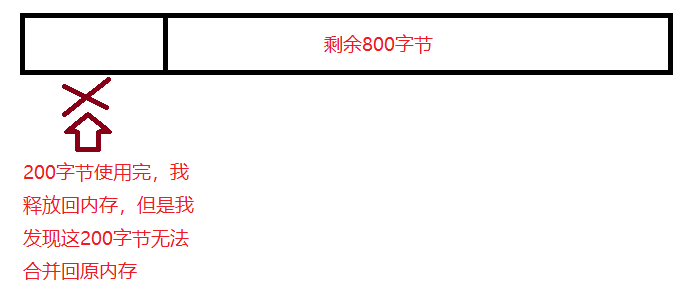
如果你觉得内存不够，需要外接SRAM来存储，那么可以在源码里面heap\_1.c



定义全局数组的时候，在static uint8\_t ucHeap[ configTOTAL\_HEAP\_SIZE ] 后面加\_\_attribute\_\_ & 0x60000,指定SRAM的起始地址0x60000，那么你的堆内存数据就放在SRAM里面了。也可能是其它地址，根据芯片来决定。

heap\_2.c







所以不断的分配和释放就会导致总内存越来越少，内存碎片越来越多。

所以heap2 适合用在反复删除队列，任务，信号量的应用中。

不能用于哪些随机分配内存大小的应用。

Heap\_3.c

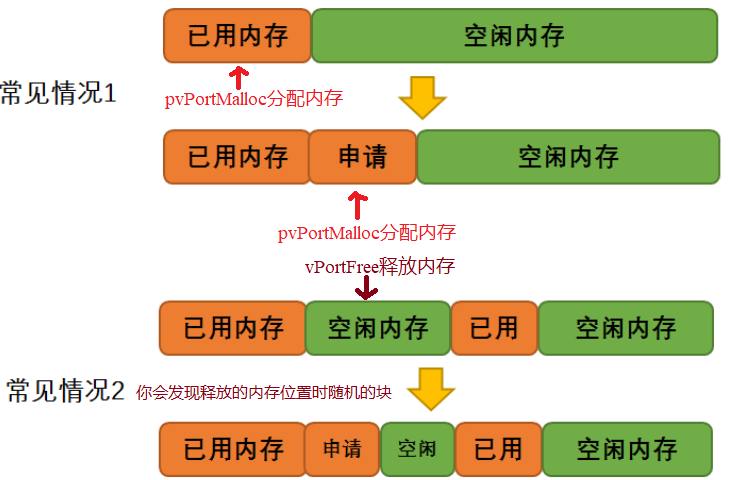
封装了C标准库的malloc()和free()，所以heap3是常规操作。



Heap\_3主要就是要STM32自己的堆内存大小要设置得大。因为heap\_3用的不是全局数组，而是用的系统本身的堆。

Heap\_4.c

使用最多的就是heap4



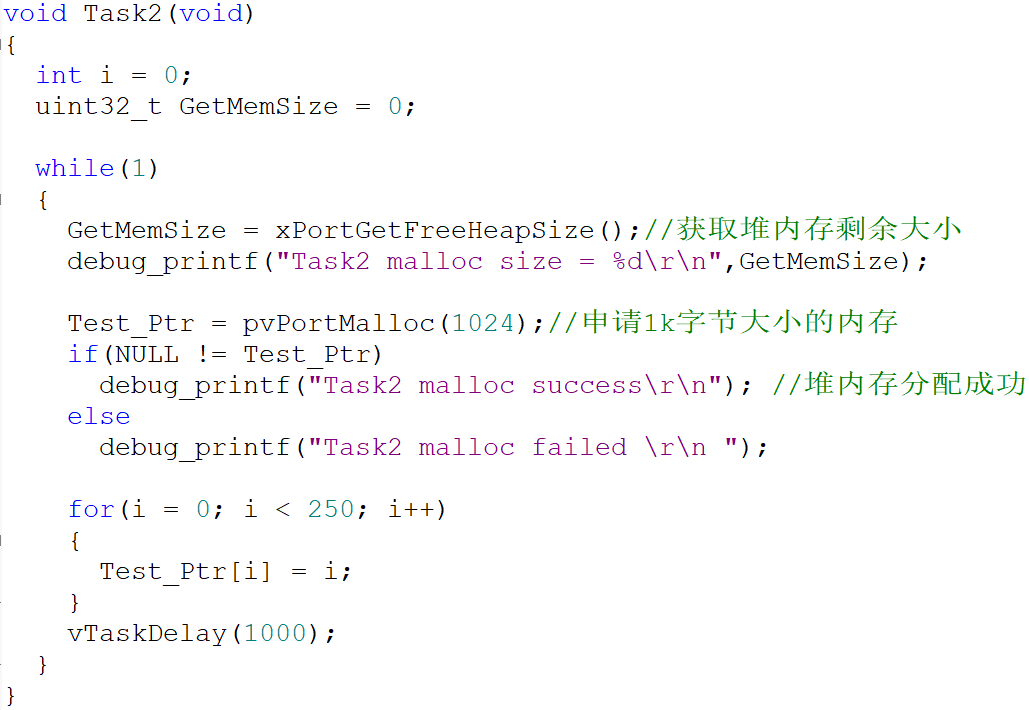
size\_t xPortGetFreeHeapSize( void ) //获取当前堆内存剩余大小

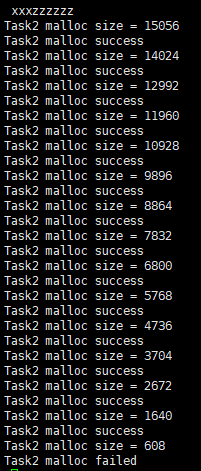
size\_t: 返回内存剩余大小，用uint32\_t变量来接收返回值。

void \*pvPortMalloc( size\_t xWantedSize )//内存分配函数，按字节分配

xWantedSize: 分配多少个字节

void \* 返回分配内存的首地址





你看右图，堆内存，每循环一次执行pvPortMalloc，总内存就会少1024字节，直到总15k内存全部分配完，报内存分配错误

我分配的堆内存是全局的，那么其它任务是否使用到该内存了呢?

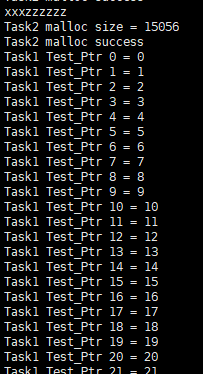
任务2申请内存，写内存，任务1读内存，打印内存

uint8\_t \*Test\_Ptr = NULL; //全局变量，用来做堆内存

void Task2(void) //任务2分配内存，写内存

{

int i = 0;



任务1成功获取到任务2分配的内存数据

uint32\_t GetMemSize = 0;

GetMemSize = xPortGetFreeHeapSize();//获取堆内存剩余大小

debug\_printf("Task2 malloc size = %d\r\n",GetMemSize);

Test\_Ptr = pvPortMalloc(1024);//申请1k字节大小的内存

if(NULL != Test\_Ptr)

debug\_printf("Task2 malloc success\r\n"); //堆内存分配成功

else

debug\_printf("Task2 malloc failed \r\n ");

for(i = 0; i < 250; i++)

{

Test\_Ptr[i] = i;

}

while(1)

{

vTaskDelay(1000);

}

}

void Task1(void) //任务1读内存

{

int i = 0;

for(i = 0; i < 250; i++)

{

debug\_printf("Task1 Test\_Ptr %d = %d \r\n",i,Test\_Ptr[i]);

}

while(1)

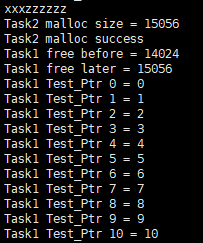
{

vTaskDelay(1000);

}

}

void vPortFree( void \*pv ) //内存释放



从内存容量来看，内存确实被释放了，但是从打印结果看貌似内存没有清0

uint8\_t \*Test\_Ptr = NULL; //全局变量，用来做堆内存

void Task1(void)

{

uint32\_t GetMemSize = 0;

int i = 0;

GetMemSize = xPortGetFreeHeapSize();

debug\_printf("Task1 free before = %d\r\n",GetMemSize);//释放内存之前

vPortFree(Test\_Ptr); //释放内存

GetMemSize = xPortGetFreeHeapSize();//释放内存之后

debug\_printf("Task1 free later = %d\r\n",GetMemSize);

for(i = 0; i < 250; i++)

{

debug\_printf("Task1 Test\_Ptr %d = %d \r\n",i,Test\_Ptr[i]);

}

while(1)

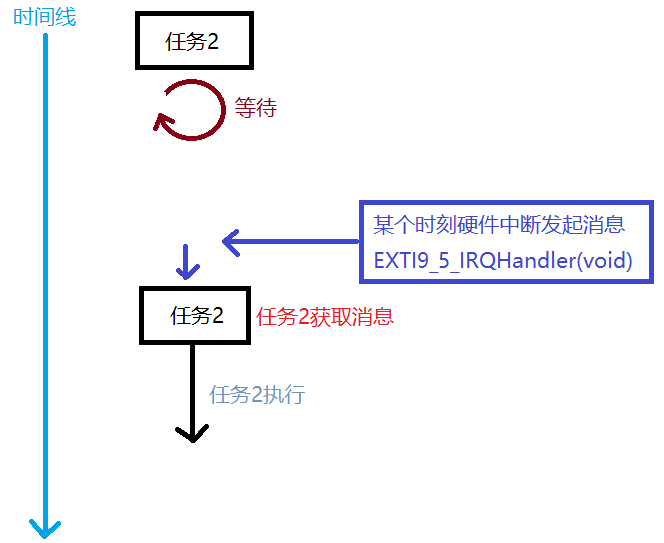
{

vTaskDelay(1000);

}

}

**中断管理，主要是FreeRTOS处理硬件中断的使用**



uint32\_t taskENTER\_CRITICAL\_FROM\_ISR() //进入临界区

返回值属于临界值，用于退出临界区的时候使用

taskEXIT\_CRITICAL\_FROM\_ISR( uint32\_t) //退出临界区

传入进入临界区的返回值，用于解锁退出临界区

xQueueSendFromISR(….，….，….) //中断里发送消息队列函数使用，在消息队列章节有介绍

代码例程:

void key\_Interrupt\_init(void) //PC5按键中断初始化

{

GPIO\_InitTypeDef GPIO\_InitStructure;

EXTI\_InitTypeDef EXTI\_InitStructure;

NVIC\_InitTypeDef NVIC\_InitStructure;

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOC,ENABLE);//使能PORTC时钟

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_5;//PC5

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_IPU; //设置成上拉输入

GPIO\_Init(GPIOC, &GPIO\_InitStructure);//初始化GPIOC5

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_AFIO,ENABLE);//使能PORTC复用时钟

EXTI\_InitStructure.EXTI\_Line = EXTI\_Line5;//PC5 就是中断线5

EXTI\_InitStructure.EXTI\_Mode = EXTI\_Mode\_Interrupt; //外部中断

EXTI\_InitStructure.EXTI\_Trigger = EXTI\_Trigger\_Falling;//下降沿触发

EXTI\_InitStructure.EXTI\_LineCmd = ENABLE;

EXTI\_Init(&EXTI\_InitStructure);

GPIO\_EXTILineConfig(GPIO\_PortSourceGPIOC,GPIO\_PinSource5); //PC5作为中断线

/\*因为FreeRTOS 初始化分配了中断组，所以其它任何时候都再分配中断组，这是FreeRTOS要求\*/

NVIC\_InitStructure.NVIC\_IRQChannel = EXTI9\_5\_IRQn;

NVIC\_InitStructure.NVIC\_IRQChannelSubPriority = 2;//设置响应优先级

NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE;

NVIC\_Init(&NVIC\_InitStructure);

}

QueueHandle\_t Test\_Queue =NULL; //消息队列句柄是全局的，方便硬件中断写入，任务2接收

void Task2(void)

{

uint32\_t data; //接收消息队列数据

BaseType\_t xReturn = pdPASS;

while(1)

{

xReturn = xQueueReceive( Test\_Queue , &data , portMAX\_DELAY );

if (pdPASS == xReturn)

debug\_printf("Task2 get PC5 key for data = %d\r\n",data);

else

debug\_printf("Task2 get PC5 key failed.. \r\n");

}

}

uint32\_t sendData = 50; //中断要发送的数据

void EXTI9\_5\_IRQHandler(void)

{

uint32\_t ulReturn; //临界段保护

ulReturn = taskENTER\_CRITICAL\_FROM\_ISR();//进入临界段

if(EXTI\_GetITStatus(EXTI\_Line5) != RESET) //判断中断是否发生

{

//debug\_printf("interrupt PC5 \r\n ");//调试的时候可以使用，最好不要在临界区使用

xQueueSendFromISR(Test\_Queue,&sendData,NULL);//使用中断消息发送函数发送消息

EXTI\_ClearFlag(EXTI\_Line5);//清除中断标志位

EXTI\_ClearITPendingBit(EXTI\_Line5);

}

taskEXIT\_CRITICAL\_FROM\_ISR( ulReturn );//退出临界段

}

int main(void)

{

BaseType\_t xReturn1,xReturn2,xReturn3,xReturn4,xReturn5;

RCC\_configuration();//初始化时钟

USART\_config(115200);//初始化串口

NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4); //STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为： 0~15

//优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，

//都统一用这个优先级分组，千万不要再分组，切忌。

key\_Interrupt\_init(); //按键中断初始化

Test\_Queue = xQueueCreate(4,sizeof(int)); //消息队列4个空间，每个空间放1个4字节int数据

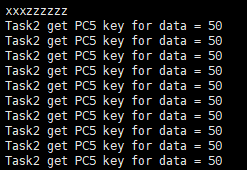
printf("xxxzzzzzz\r\n");

semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量

xSemaphoreGive(semDebug); //必须在初始化前就给先向semDebug变量里面写入1，因为debug\_printf函数最后一行才会给semDebug写1

创建任务2……

}

任务2接收按键触发的消息队列。

**CPU使用率统计**

在FreeRTOSconfig.h中加入以下两个宏

#define configGENERATE\_RUN\_TIME\_STATS 1 //CPU利用率计算需要开启运行时间统计

#define configUSE\_TRACE\_FACILITY 1 //启动可视化跟踪调试

#include "stm32f10x.h"

extern volatile uint32\_t CPU\_RunTime; //定义全局CPU时间

#define configUSE\_STATS\_FORMATTING\_FUNCTIONS 1 //格式化启动

#define portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() (CPU\_RunTime = 0ul)

#define portGET\_RUN\_TIME\_COUNTER\_VALUE() CPU\_RunTime

配置完成后一定要去某个C文件外部实现volatile uint32\_t CPU\_RunTime = 0UL; 这样才能编译过

最好在高级定时器中断外定义volatile uint32\_t CPU\_RunTime = 0UL; 全局变量

高级度定时要求: 比如系统节拍是1000hz，那么定时器节拍数就要是10000~20000hz，我选用20000hz做CPU使用率统计，因为我定义的变量是32位，那么59.6分钟之后就会溢出，导致59.6分钟之后计算的CPU使用率不准，所以可以清0重算。

代码例程

//通用定时器中断初始化

//这里时钟选择为APB1的2倍，而APB1为36M

//arr：自动重装值。

//psc：时钟预分频数

//这里使用的是定时器3!

void TIM3\_Int\_Init(u16 arr,u16 psc)

{

TIM\_TimeBaseInitTypeDef TIM\_TimeBaseStructure;

NVIC\_InitTypeDef NVIC\_InitStructure;

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_TIM3, ENABLE); //时钟使能

TIM\_TimeBaseStructure.TIM\_Period = arr; //设置在下一个更新事件装入活动的自动重装载寄存器周期的值 计数到5000为500ms

TIM\_TimeBaseStructure.TIM\_Prescaler =psc; //设置用来作为TIMx时钟频率除数的预分频值 10Khz的计数频率

TIM\_TimeBaseStructure.TIM\_ClockDivision = 0; //设置时钟分割:TDTS = Tck\_tim

TIM\_TimeBaseStructure.TIM\_CounterMode = TIM\_CounterMode\_Up; //TIM向上计数模式

TIM\_TimeBaseInit(TIM3, &TIM\_TimeBaseStructure); //根据TIM\_TimeBaseInitStruct中指定的参数初始化TIMx的时间基数单位

TIM\_ITConfig( //使能或者失能指定的TIM中断

TIM3, //TIM2

TIM\_IT\_Update ,

ENABLE //使能

);

NVIC\_InitStructure.NVIC\_IRQChannel = TIM3\_IRQn; //TIM3中断

NVIC\_InitStructure.NVIC\_IRQChannelSubPriority = 3; //从优先级3级

NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE; //IRQ通道被使能

NVIC\_Init(&NVIC\_InitStructure); //根据NVIC\_InitStruct中指定的参数初始化外设NVIC寄存器

TIM\_Cmd(TIM3, ENABLE); //使能TIMx外设

}

volatile uint32\_t CPU\_RunTime = 0UL;

void TIM3\_IRQHandler(void) //TIM3中断

{

if (TIM\_GetITStatus(TIM3, TIM\_IT\_Update) != RESET) //检查指定的TIM中断发生与否:TIM 中断源

{

CPU\_RunTime ++ ; //CPU运行时间自加，很重要

TIM\_ClearITPendingBit(TIM3, TIM\_IT\_Update ); //清除TIMx的中断待处理位:TIM 中断源

}

}

void TIM3\_Int\_Init(u16 arr,u16 psc);

int main(void)

{

BaseType\_t xReturn1,xReturn2,xReturn3,xReturn4,xReturn5;

RCC\_configuration();//初始化时钟

USART\_config(115200);//初始化串口

NVIC\_PriorityGroupConfig( NVIC\_PriorityGroup\_4); //STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为： 0~15

//优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，

//都统一用这个优先级分组，千万不要再分组，切忌。

TIM3\_Int\_Init(1,3599);//72M/20000=3600,因为定时器内部+1所以传3599

//20Khz的计数频率，计数到1大概为0.00002(20us)

printf("xxxzzzzzz\r\n");

semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量

xSemaphoreGive(semDebug); //必须在初始化前就给先向semDebug变量里面写入1，因为debug\_printf函数最后一行才会给semDebug写1

xReturn1 = xTaskCreate((TaskFunction\_t)Task1, //任务函数地址

(const char\* )"Task1", //任务名

(uint32\_t)1024, //任务栈大小

(void\*)NULL, //向函数传参

(UBaseType\_t)1, //优先级

(TaskHandle\_t\* )&Task1TCB); //任务控制块

xReturn2 = xTaskCreate((TaskFunction\_t)Task2,

(const char\* )"Task2",

(uint32\_t)1024, //任务2栈大小要改成1024，因为在任务2函数中申请了400的数组已经大于以前的128字节栈了，不然会死机

(void\*)NULL,

(UBaseType\_t)7,

(TaskHandle\_t\* )&Task2TCB); //因为任务2是发送消息的，所以优先级一定要高于其它任务

xReturn3 = xTaskCreate((TaskFunction\_t)Task3,

(const char\* )"Task3",

(uint32\_t)1024,

(void\*)NULL,

(UBaseType\_t)3,

(TaskHandle\_t\* )&Task3TCB);

启动任务1，任务2，任务3，….

}

void Task1(void)

{

while(1)

{

debug\_printf(" TASK1 ..... \r\n");

vTaskDelay(1000);

}

}

void Task3(void)

{

while(1){

debug\_printf(" TASK3 ..... \r\n");

vTaskDelay(500);

}

}

void Task2(void)

{

uint8\_t CPU\_RunInfo[400]; //保存任务运行时间信息

//创建任务的时候记得修改任务栈大小，CPU\_RunInfo都已经超过128字节了

while(1)

{

memset(CPU\_RunInfo,0,400); //信息缓冲区清零

vTaskList((char \*)&CPU\_RunInfo); //获取任务运行时间信息

debug\_printf("-------------------------------- \r\n");

debug\_printf("TaskName status prioritj stack number\r\n");

debug\_printf("%s", CPU\_RunInfo);

debug\_printf("-------------------------------- \r\n");

memset(CPU\_RunInfo,0,400); //信息缓冲区清零

vTaskGetRunTimeStats((char \*)&CPU\_RunInfo);

debug\_printf("TaskName CPUcount UsageRate\r\n");

debug\_printf("%s", CPU\_RunInfo);

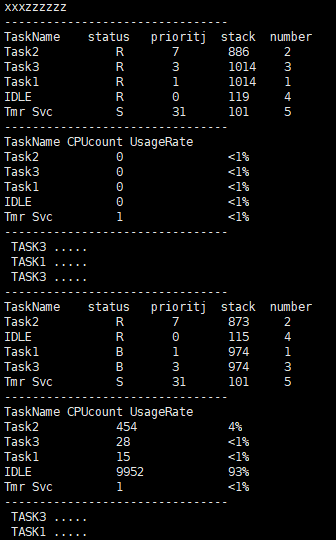
debug\_printf("-------------------------------- \r\n");

vTaskDelay(1000);

}

}

任务名 任务状态 优先级 任务剩余栈 序号



第2次开始就准了

才启动系统，第1次是不准的

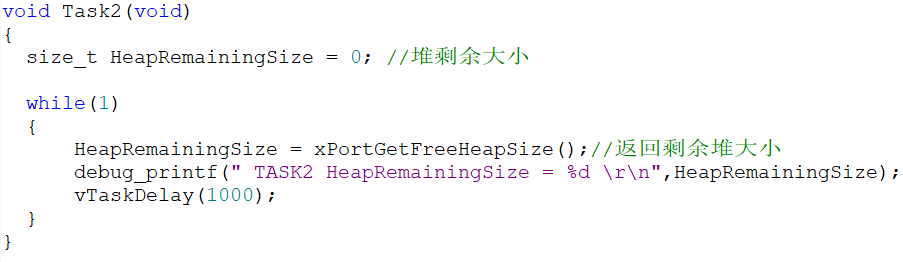
**查询系统堆大小，用于检测程序是否长时间正常运行**

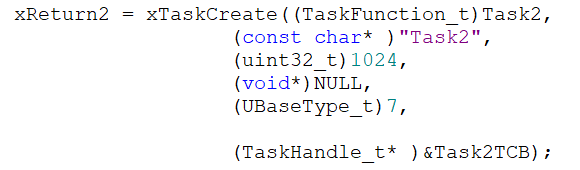
size\_t xPortGetFreeHeapSize( void ) //系统剩余堆大小检测

返回剩余堆大小，size\_t 采用%d打印



在FreeRTOSconfig.h中，系统总共的堆大小我定义的16K

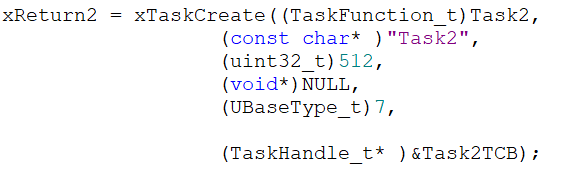




现在只启动Task2任务，这里分配的堆是4K，因为uint32\_t是4字节，所以4 x 1024 = 4K

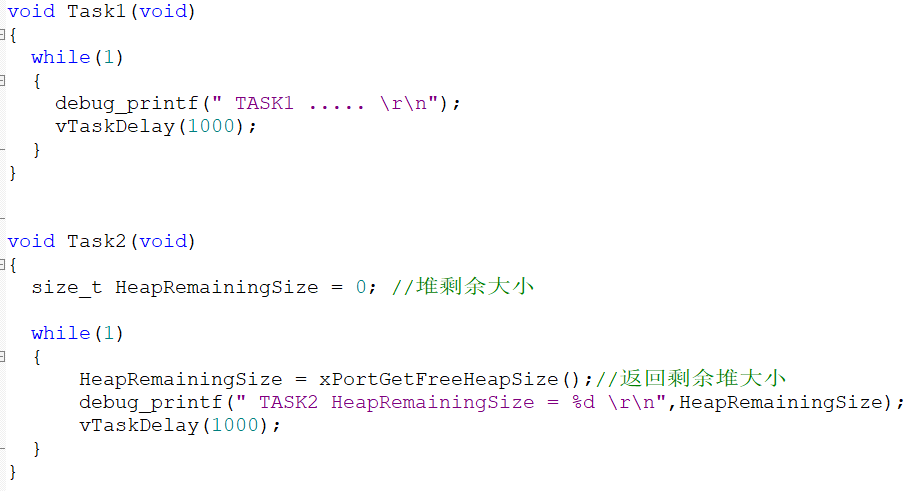
 你看总堆大小16K，运行一个Task2任务消耗4K的堆，还剩12K堆

我将任务堆改成512试试，uint32\_t 就是4，那么4 x 512 = 2048(2k)

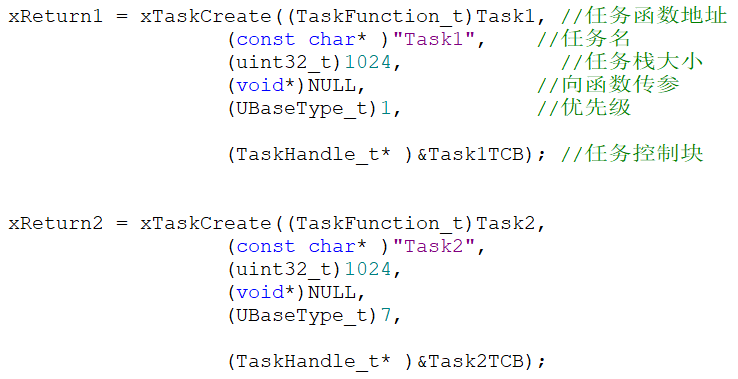


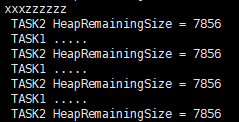
你看，16K总堆只开辟了2K 还剩14K

下面进行多任务堆消耗实验

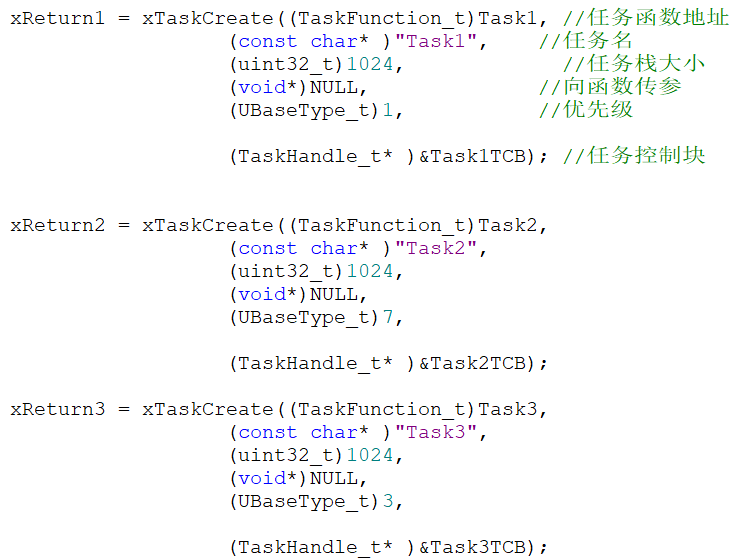


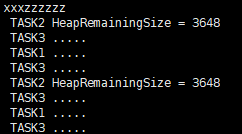
启动任务1，消耗4K





你看，任务2消耗4K ，任务1消耗4K，累计8K ，16K-8K = 8K，还剩接近于8K的堆可以再开辟

启动任务3 就还剩3~4K的堆

确实最后只剩下3K堆可以再分配了

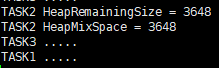
所以要做多线程项目，总堆的空间是越大越好，因为FreeRTOS里面的线程栈，用的是对空间代替的。

size\_t **xPortGetMinimumEverFreeHeapSize**( **void** );

//查询堆最小空间，其实和剩余堆大小查询一样的功能

返回堆最小空间，size\_t 采用%d打印



堆最小空间其实和堆剩余空间一样的功能

**void** **vApplicationMallocFailedHook**( **void** ); //堆申请失败回调函数

这是一个回调函数，需要用户自己实现。如果配置文件中configUSE\_MALLOC\_FAILED\_HOOK 设置为1的话，当堆分配内存失败时会调用此函数。用户可以在此函数中进行错误处理。

**查询每个任务栈使用大小，用于检测系统是否长时间正常运行**

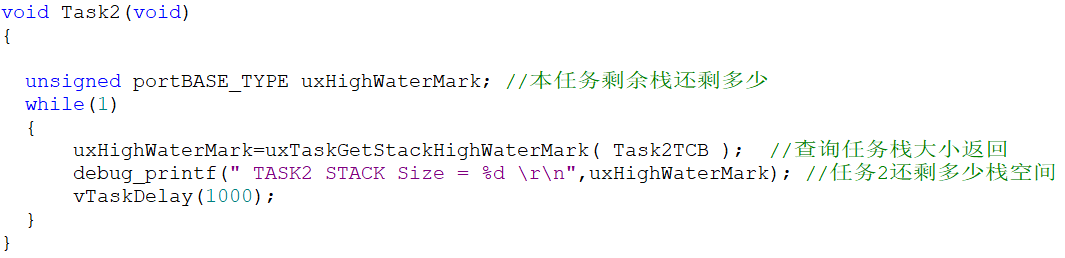
UBaseType\_t uxTaskGetStackHighWaterMark( TaskHandle\_t xTask ) //查看当前任务栈还剩多少

xTask: 你要查询哪一个任务栈的剩余大小，就传入该任务句柄

UBaseType\_t: 返回该任务栈还剩多少,返回值不是计算的字节数，而是计算的uint32\_t个数，所以你要把返回结果 x4才是字节

#define INCLUDE\_uxTaskGetStackHighWaterMark 1 //查询任务栈大小，需要将该宏置1

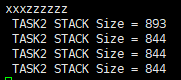
 任务句柄



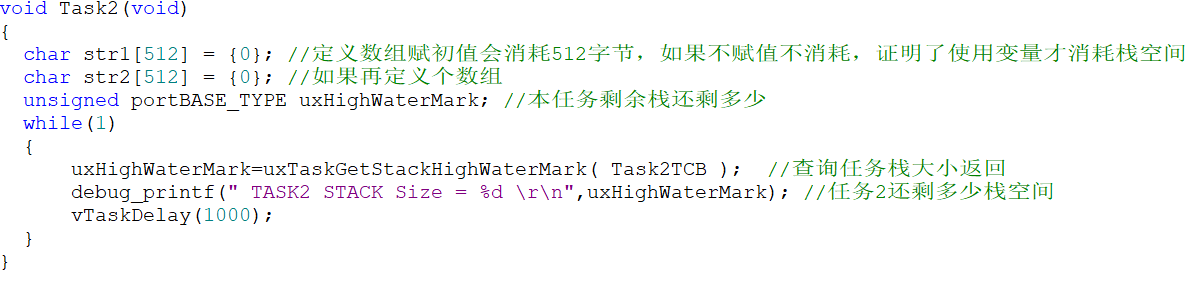
任务开始是1041 x 4 = 4164(4k)栈空间

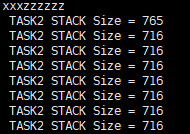
后来函数调用，变量定义消耗了 1014 – 972 = 42,记住42 x 4 =168这才是字节消耗了168字节栈空间

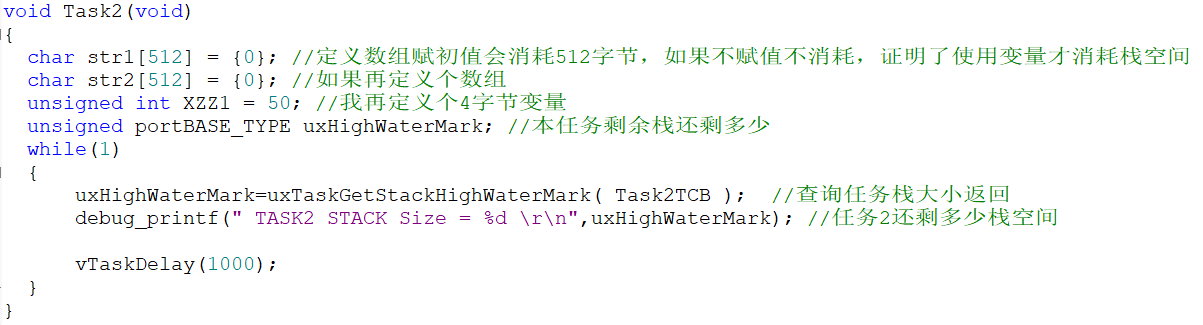


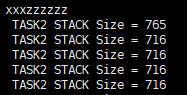
上一页说明了，任务启动剩972 x 4 = 3888字节栈空间

现在 972 – 844 = 128，再128 x 4 = 512字节，所以定义数组消耗了栈空间512字节，还剩844 x 4 =3376字节



 972 – 716 = 256,256 x 4 = 1024，两个数组消耗了1K的栈空间



 定义4字节变量1个，太少了，看不出来，但是我发现只有定义数组才能看出来。unsigned int XZZ1[1] = {0}; 这要就显示714，消耗8字节。