

FreeRTOS实时操作系统复习

笔记基于韦东山FreeRTOS学习手册修改

一、笔记前言

三大要点：

- 讲解FreeRTOS 的常用API：理论、用法
- 选择合适的硬件模块，展示这些API 的实例
- 实现合适的小项目，展示工作中的编程方法

单片机程序设计模式

裸机程序设计模式：

裸机程序的设计模式可以分为：轮询、前后台、定时器驱动、基于状态机。

前面三种方法都无法解决一个问题：假设有A、B 两个都很耗时的函数，无法降低它们相互之间的影响。

第4 种方法可以解决这个问题，但是实践起来有难度。

假设一位职场妈妈需要同时解决2 个问题：给小孩喂饭、回复工作信息

各类模式下如何写程序呢？

轮询模式：

示例代码如下：

```
01 // 经典单片机程序：轮询
02 void main()
03 {
04     while (1)
05     {
06         喂一口饭();
07         回一个信息();
08     }
09 }
```

使用轮询模式编写程序看起来很简单，但是要求while 循环里调用到的函数要执行得非常快，在复杂场景里反而增加了编程难度。

前后台模式：

所谓“前后台”就是使用中断程序。

```
01 // 前后台程序
02 void main()
03 {
04     while (1)
05     {
```

5

百问网

```
06 // 后台程序
07 喂一口饭();
08 }
09 }
10
11 // 前台程序
12 void 滴_中断()
13 {
14     回一个信息();
15 }
```

也可以该进程两个中断的形式：

```
01 // 前后台程序
02 void main()
03 {
04     while (1)
05     {
06         // 后台程序
07     }
08 }
09
10 // 前台程序
11 void 滴_中断()
12 {
13     回一个信息();
14 }
15
16 // 前台程序
17 void 啊_中断()
18 {
19     喂一口饭();
20 }
```

定时器驱动：

```
01 // 定时器驱动：后台程序
02 void main()
03 {
04     while (1)
05     {
06         // 后台程序
07     }
```

6

百问网

```
08 }
09
10 // 前台程序：每1分钟触发一次中断
11 void 定时器_中断()
12 {
13     static int cnt = 0;
14     cnt++;
15     if (cnt % 2 == 0)
16     {
17         喂一口饭();
18     }
19     else if (cnt % 5 == 0)
20     {
21         回一个信息();
22     }
23 }
```

这种模式适合调用周期性的函数，并且每一个函数执行的时间不能超过一个定时器周期。

如果“喂一口饭”很花时间，比如长达10分钟，那么就会耽误“回一个信息”；

反过来也是一样的，如果“回一个信息”很花时间也会影响到“喂一口饭”；

这种场景下程序遭遇到了轮询模式的缺点：函数相互之间有影响。

基于状态机：

主函数：

```
01 // 状态机
02 void main()
03 {
04     while (1)
05     {
06         喂一口饭();
07         回一个信息();
08     }
09 }
```

在main函数里，还是使用轮询模式依次调用2个函数。

关键在于这2个函数的内部实现：使用状态机，每次只执行一个状态的代码，减少每次执行的时间
喂饭：

```
01 void 喂一口饭(void)
02 {
03     static int state = 0;
04     switch (state)
05     {
06         case 0:
07         {
08             /* 酱饭 */
09             /* 进入下一个状态 */
10             state++;
11             break;
12         }
13         case 1:
14         {
15             /* 喂饭 */
16             /* 进入下一个状态 */
17             state++;
18             break;
19         }
20         case 2:
21         {
22             /* 酱菜 */
23             /* 进入下一个状态 */
24             state++;
25             break;
26         }
27         case 3:
28         {
29             /* 喂菜 */
30             /* 恢复到初始状态 */
31             state = 0;
32             break;
33         }
34     }
35 }
36 }
```

回信息：

```
37 void 回一个信息(void)
38 {
39     static int state = 0;
40
41     switch (state)
42     {
43         case 0:
44         {
45             /* 查看信息 */
46             /* 进入下一个状态 */
47             state++;
48             break;
49         }
50         case 1:
51         {
52             /* 打字 */
53             /* 进入下一个状态 */
54             state++;
55             break;
56         }
57         case 2:
58         {
59             /* 发送 */
60             /* 恢复到初始状态 */
61             state = 0;
62             break;
63         }
64     }
65 }
```

扫描

每个函数分成了四部分，每次执行只执行一部分。

多任务系统

RTOS



示例代码：

```
01 // RTOS 程序
02 喂饭任务()
03 {
04     while (1)
05     {
06         喂一口饭();
07     }
08 }
09
10 回信息任务()
11 {
12     while (1)
13     {
14         回一个信息();
15     }
16 }
17
18 void main()
19 {
20     // 创建 2 个任务
21     create_task(喂饭任务);
22     create_task(回信息任务);
23
24     // 启动调度器
25     start_scheduler();
26 }
```

多任务系统会依次给这些任务分配时间：你执行一会，我执行一会，如此循环。

只要切换的间隔足够短，用户会“感觉这些任务在同时运行”

多任务系统中，多个任务可能会“同时”访问某些资源，需要增加保护措施以防止混乱。

比如任务A、B 都要使用串口，能否使用一个全局变量让它们独占地、互斥地使用串口？--> 任务互斥操作

如果任务之间有依赖关系，比如任务A 执行了某个操作之后，需要任务B 进行后续的处理。--> 任务同步操作

二、Freertos工程

FreeRTOS目录结构

以Keil工具下STM32F103芯片为例，它的FreeRTOS的目录如下：

```
|-- FreeRTOS
|   |-- Demo           // 预先制作好的示例工程
|   |   |-- CORTEX_STM32F103_Keil // STM32F103在Keil环境下的工程文件
|   |   |   |-- FreeRTOSConfig.h
|   |   |   |   |-- .....
|   |   |   |-- Common      // 独立于demo的通用代码，大部分已经废弃
|   |   |   |-- Source       // FreeRTOS源码
|   |   |   |   |-- croutine.c
|   |   |   |   |-- event_groups.c
|   |   |   |   |-- list.c
|   |   |   |   |-- queue.c
|   |   |   |   |-- stream_buffer.c
|   |   |   |   |-- tasks.c
|   |   |   |   `-- timers.c
|   |   |   |-- include
|   |   |   |-- portable      // 移植时需要实现的文件
|   |   |   |   |-- RVDS          // IDE为RVDS或Keil
|   |   |   |   |   |-- ARM_CM3      // CortexM3架构
|   |   |   |   |   |   |-- port.c
|   |   |   |   |   |   `-- portmacro.h
|   |   |   |   |   |-- MemMang        // 内存管理
|   |   |   |   |   |   |-- heap_1.c
|   |   |   |   |   |   |-- heap_2.c
|   |   |   |   |   |   |-- heap_3.c
|   |   |   |   |   |   |-- heap_4.c
|   |   |   |   |   |   `-- heap_5.c
|   |-- FreeRTOS-Plus     // FreeRTOS生态的文件，非必需
|   |   |-- Demo
|   |   |-- Source
```

// 核心文件

主要涉及两个目录：

Demo：

Demo目录下是工程文件，以"芯片和编译器"组合成一个名字比如：CORTEX_STM32F103_Keil

Source：

根目录下是核心文件，这些文件是通用的

portable目录下是移植时需要实现的文件

目录名为：[compiler]\[architecture]

比如：RVDS/ARM_CM3，这表示cortexM3架构在RVDS工具上的移植文件

核心文件：

FreeRTOS的最核心文件只有2个：

FreeRTOS/Source/tasks.c

FreeRTOS/Source/list.c

其他文件的作用也一起列表如下：

FreeRTOS/Source/下的文件	作用
tasks.c	必需，任务操作
list.c	必须，列表
queue.c	基本必需，提供队列操作、信号量(semaphore)操作
timer.c	可选，software timer
event_groups.c	可选，提供event group功能
croutine.c	可选，过时了

移植FreeRTOS时涉及的文件放在FreeRTOS/Source/portable/[compiler]\[architecture] 目录下，比如：RVDS/ARM_CM3，这表示cortexM3架构在RVDS或Keil工具上的移植文件。
里面有2个文件：port.c portmacro.h

FreeRTOS需要3个头文件目录：

- FreeRTOS本身的头文件：FreeRTOS/Source/include
- 移植时用到的头文件：FreeRTOS/Source/portable/[compiler]/[architecture]
- 含有配置文件FreeRTOSConfig.h的目录

头文件	作用
FreeRTOSConfig.h	FreeRTOS的配置文件，比如选择调度算法：configUSE_PREEMPTION 每个demo都必定含有FreeRTOSConfig.h 建议去修改demo中的FreeRTOSConfig.h，而不是从头写一个
FreeRTOS.h	使用FreeRTOS API函数时，必须包含此文件。 在FreeRTOS.h之后，再去包含其他头文件，比如： task.h、queue.h、semphr.h、event_group.h

内存管理：

文件在FreeRTOS/Source/portable/MemMang 下，它也是放在portable 目录下，表示你可以提供自己的函数。

源码中默认提供了5个文件，对应内存管理的5种方法。

文件	优点	缺点
heap_1.c	分配简单，时间确定	只分配、不回收
heap_2.c	动态分配、最佳匹配	碎片、时间不定
heap_3.c	调用标准库函数	速度慢、时间不定
heap_4.c	相邻空闲内存可合并	可解决碎片问题、时间不定
heap_5.c	在heap_4基础上支持分隔的内存块	可解决碎片问题、时间不定

FreeRTOS Demo

FreeRTOS 系统文件

名称	修改日期	类型	大小
.vscode	2025/2/10 19:16	文件夹	
DebugConfig	2025/2/10 19:16	文件夹	
freertos	2025/2/10 19:16	文件夹	
Library	2025/2/10 19:16	文件夹	
Listings	2025/2/10 19:16	文件夹	
Objects	2025/2/10 19:16	文件夹	
start	2025/2/10 19:16	文件夹	
System	2025/2/10 19:16	文件夹	
User	2025/2/10 19:16	文件夹	
EventRecorderStub.scvd	2025/2/5 0:11	SCVD 文件	1 KB
keilkill.bat	2011/5/9 18:17	Windows 批处理...	1 KB
Project.uvguix.LENOVO	2025/2/5 0:19	LENOVO 文件	173 KB
Project.uvoptx	2025/2/5 0:15	UVOPTX 文件	40 KB
Project.uvprojx	2025/2/5 0:15	礦ision5 Project	30 KB

名称	修改日期	类型	大小
inc 头文件	2025/2/10 19:16	文件夹	
port 移植文件	2025/2/10 19:16	文件夹	
src 源文件	2025/2/10 19:16	文件夹	
FreeRTOSConfig.h 包含文件	2025/2/4 23:49	DevCpp.h	4 KB

inc:

The screenshot shows a file explorer window with the following details:

Path: T01STM32工程模版 > freertos > inc

Search bar: 在 inc 中搜索

Toolbar: 新建, 剪切, 复制, 粘贴, 打开, 重命名, 删除, 排序, 查看, ...

Table Headers: 名称, 修改日期, 类型, 大小

File List:

名称	修改日期	类型	大小
atomic.h	2023/3/3 23:12	DevCpp.h	13 KB
croutine.h	2023/3/3 23:12	DevCpp.h	29 KB
deprecated_definitions.h	2023/3/3 23:12	DevCpp.h	8 KB
event_groups.h	2023/3/3 23:12	DevCpp.h	32 KB
FreeRTOS.h	2023/3/3 23:12	DevCpp.h	51 KB
list.h	2023/3/3 23:12	DevCpp.h	24 KB
message_buffer.h	2023/3/3 23:12	DevCpp.h	41 KB
mpu_prototypes.h	2023/3/3 23:12	DevCpp.h	18 KB
mpu_wrappers.h	2023/3/3 23:12	DevCpp.h	11 KB
portable.h	2023/3/3 23:12	DevCpp.h	10 KB
projdefs.h	2023/3/3 23:12	DevCpp.h	7 KB
queue.h	2023/3/3 23:12	DevCpp.h	65 KB
semphr.h	2023/3/3 23:12	DevCpp.h	50 KB
stack_macros.h	2023/3/3 23:12	DevCpp.h	9 KB
StackMacros.h	2023/3/3 23:12	DevCpp.h	2 KB
stdint.readme	2023/3/3 23:12	README 文件	3 KB
stream_buffer.h	2023/3/3 23:12	DevCpp.h	42 KB
task.h	2023/3/3 23:12	DevCpp.h	135 KB
timers.h	2023/3/3 23:12	DevCpp.h	62 KB

项目:

port:

The screenshot shows a file explorer window with the following details:

Path: T01STM32工程模版 > freertos > port

Search bar: 在 port 中搜索

Toolbar: 新建, 剪切, 复制, 重命名, 删除, 排序, 查看, ...

Table Headers: 名称, 修改日期, 类型, 大小

File List:

名称	修改日期	类型	大小
heap_1.c	2023/3/3 23:12	C 源文件	6 KB
heap_2.c	2023/3/3 23:12	C 源文件	16 KB
heap_3.c	2023/3/3 23:12	C 源文件	3 KB
heap_4.c	2023/3/3 23:12	C 源文件	21 KB
heap_5.c	2023/3/3 23:12	C 源文件	23 KB
port.c	2023/3/3 23:12	C 源文件	33 KB
portmacro.h	2023/3/3 23:12	DevCpp.h	10 KB
ReadMe	2023/3/3 23:12	Internet 快捷方式	1 KB

Src:

The screenshot shows a file explorer window with the following directory structure:

名称	修改日期	类型	大小
.gitmodules	2023/3/3 23:12	GITMODULES 文件	1 KB
CMakeLists.txt	2023/3/3 23:12	文本文档	17 KB
croutine.c	2023/3/3 23:12	C 源文件	16 KB
event_groups.c	2023/3/3 23:12	C 源文件	32 KB
list.c	2023/3/3 23:12	C 源文件	11 KB
manifest.yml	2023/3/3 23:12	Yaml 源文件	1 KB
queue.c	2023/3/3 23:12	C 源文件	123 KB
sbom.spdx	2023/3/3 23:12	SPDX 文件	54 KB
stream_buffer.c	2023/3/3 23:12	C 源文件	61 KB
tasks.c	2023/3/3 23:12	C 源文件	219 KB
timers.c	2023/3/3 23:12	C 源文件	49 KB

Demo中必须包含FreeRTOS.h

The screenshot shows the `main.c` file with the following content:

```
User > C main.c > ...
1 //include "stm32f10x.h"           // Device header
2 #include "freertos.h"
3 #include "task.h"
4
5 void myTask( void * arg){
6     while(1){
7         GPIO_SetBits(GPIOC,GPIO_Pin_13);
8         vTaskDelay(500);
9         GPIO_ResetBits(GPIOC,GPIO_Pin_13);
10        vTaskDelay(500);
11    }
12 }
13 void myTask2( void * arg){
14     while(1){
15         GPIO_SetBits(GPIOC,GPIO_Pin_15);
16         vTaskDelay(1000);
17         GPIO_ResetBits(GPIOC,GPIO_Pin_15);
18         vTaskDelay(1000);
19    }
20 }
21
22 int main(void){
23
24     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC,ENABLE);
25     GPIO_InitTypeDef GPIO_InitStructure;
26     GPIO_InitStructure.GPIO_Mode=GPIO_Mode_Out_PP;
27     GPIO_InitStructure.GPIO_Pin=GPIO_Pin_13|GPIO_Pin_15;
}
```

The line `#include "freertos.h"` is highlighted with a red rectangle.

编程规范

补充知识:

```
1 typedef 定义函数指针类型:
2 typedef int (*FuncPtr) (int,int);
3 定义了 FuncPtr类型的函数指针, 返回值为int, 参数为int, int
4
5 typedef struct 结构体名 *指针名
6 定义结构体指针变量, 指针名变量是指向struct 结构体的一个指针
```

```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )

    BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
                            const char * const pcName, /*lint !e971 Unqualified char types are
                            const configSTACK_DEPTH_TYPE usStackDepth,
                            void * const pvParameters,
                            UBaseType_t uxPriority,
                            TaskHandle_t * const pxCreatedTask )

    {
        TCB_t * pxNewTCB;
        BaseType_t xReturn;

        /*
         * Defines the prototype to which task functions must conform. Defined in this
         * file to ensure the type is known before portable.h is included.
         */
        typedef void (* TaskFunction_t)( void * );
        */

        struct tskTaskControlBlock; /* The old naming convention is used to prevent breaking kernel
        typedef struct tskTaskControlBlock * TaskHandle_t;

        /*

```

数据类型

每个移植的版本都含有自己的portmacro.h 头文件，里面定义了2个数据类型：

TickType_t: 时钟终端次数的类型

- FreeRTOS配置了一个周期性的时钟中断：Tick Interrupt
- 每发生一次中断，中断次数累加，这被称为tick count
- tick count这个变量的类型就是TickType_t
- TickType_t可以是16位的，也可以是32位的
- FreeRTOSConfig.h中定义configUSE_16_BIT TICKS时，TickType_t就是uint16_t
- 否则TickType_t就是uint32_t
- 对于32位架构，建议把TickType_t配置为uint32_t

BaseType_t: 架构最高效的返回值类型(架构几位我几位)

- 这是该架构最高效的数据类型
- 32位架构中，它就是uint32_t
- 16位架构中，它就是uint16_t
- 8位架构中，它就是uint8_t
- BaseType_t通常用作简单的返回值的类型，还有逻辑值，比如pdTRUE/pdFALSE

变量名

变量名前缀寓意类型：

变量名前缀	含义
c	char
s	int16_t, short
l	int32_t, long
x	BaseType_t, 其他非标准的类型：结构体、task handle、queue handle等
u	unsigned
p	指针
uc	uint8_t, unsigned char
pc	char指针

扫描

函数名

函数名的前缀有2部分：返回值类型、在哪个文件定义。

函数名前缀	含义
vTaskPrioritySet	返回值类型：void 在task.c中定义
xQueueReceive	返回值类型：BaseType_t 在queue.c中定义
pvTimerGetTimerID	返回值类型：pointer to void 在timer.c中定义

宏名

宏的名字是大小，可以添加小写的前缀。

前缀是用来表示：宏在哪个文件中定义。

宏的前缀	含义：在哪个文件里定义
port (比如portMAX_DELAY)	portable.h或portmacro.h
task (比如taskENTER_CRITICAL())	task.h
pd (比如pdTRUE)	projdefs.h
config (比如configUSE_PREEMPTION)	FreeRTOSConfig.h
err (比如errQUEUE_FULL)	projdefs.h

通用宏定义：

通用的宏定义如下：

宏	值
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

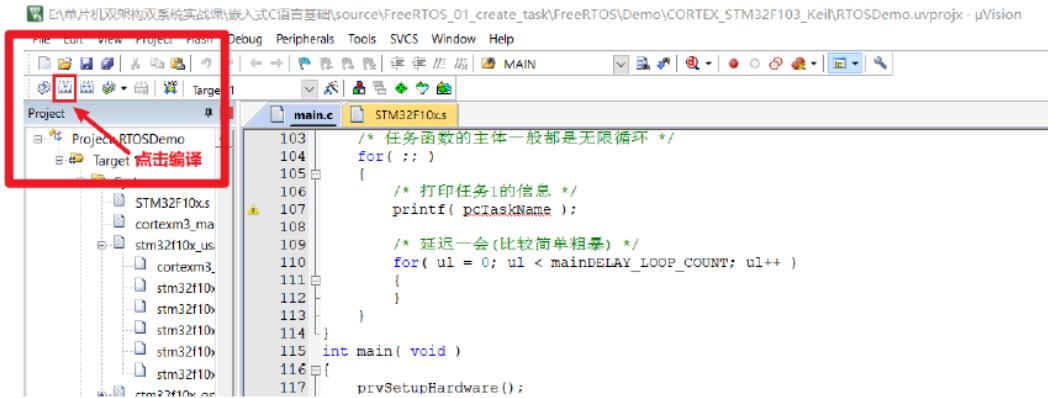
Keil模拟器

编译仿真：

先获取配套示例代码。

双击"FreeRTOS_01_create_task\FreeRTOS\Demo\CORTEX_STM32F103_Keil\RTOSDemo.uvprojx"打开第一个示例。

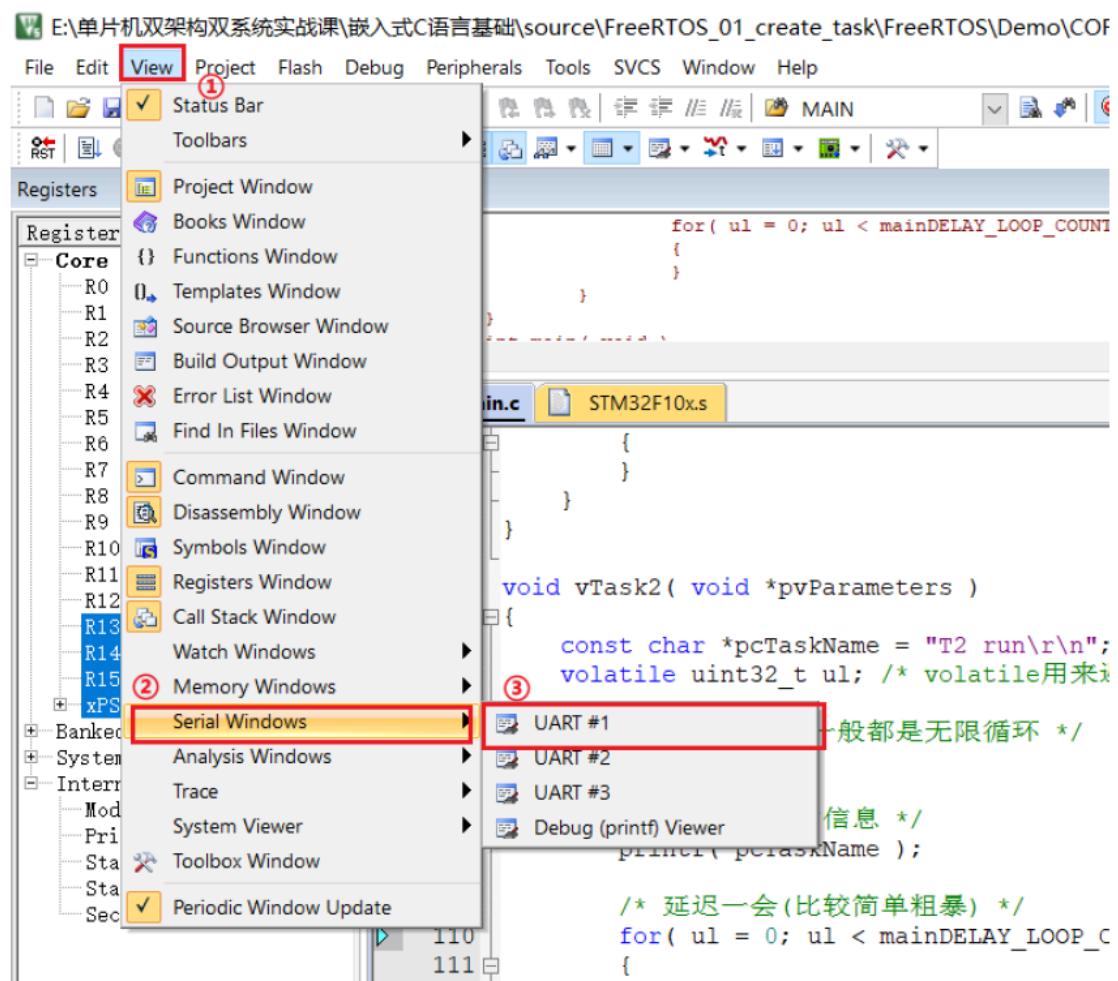
打开之后，首先要**编译工程**，才能使用模拟器运行，点击"Build"图标进行编译，如下图所示：



编译完成后，点击"Debug"按钮进行仿真，如下图所示：

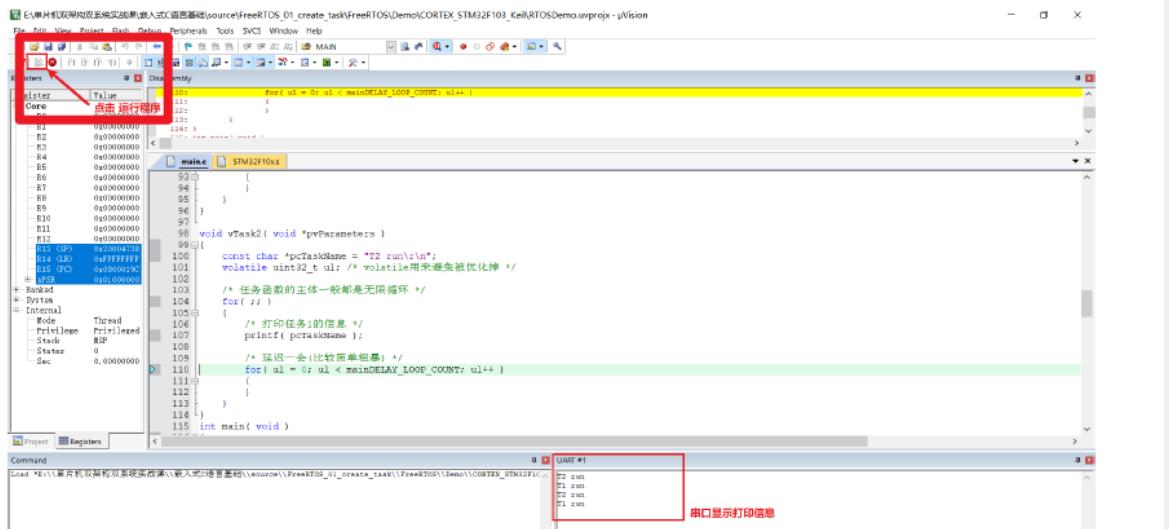


可以在VIEW中看到各种外设状态：



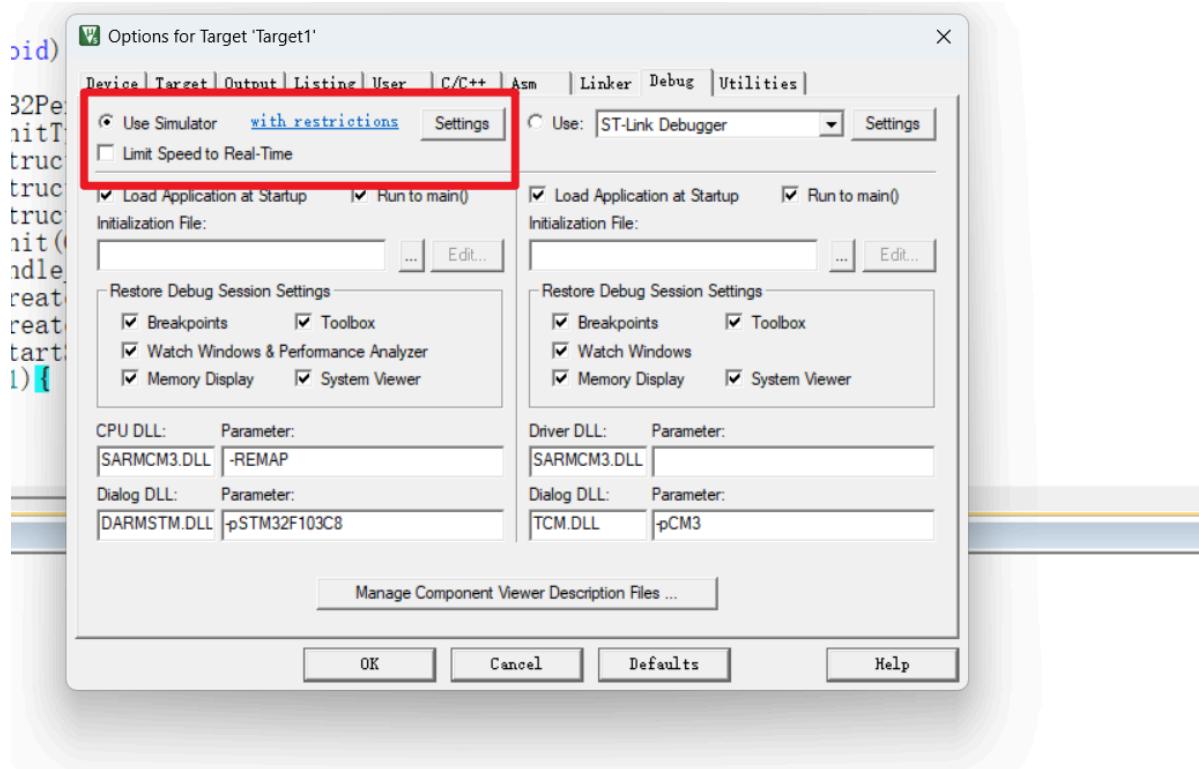
RUN运行即可：

最后，点击“Run”运行程序，右下角串口显示窗口将打印两个任务的信息。



再次点击Debug即可退出。

注意：魔术棒工具Debug中选择仿真器。



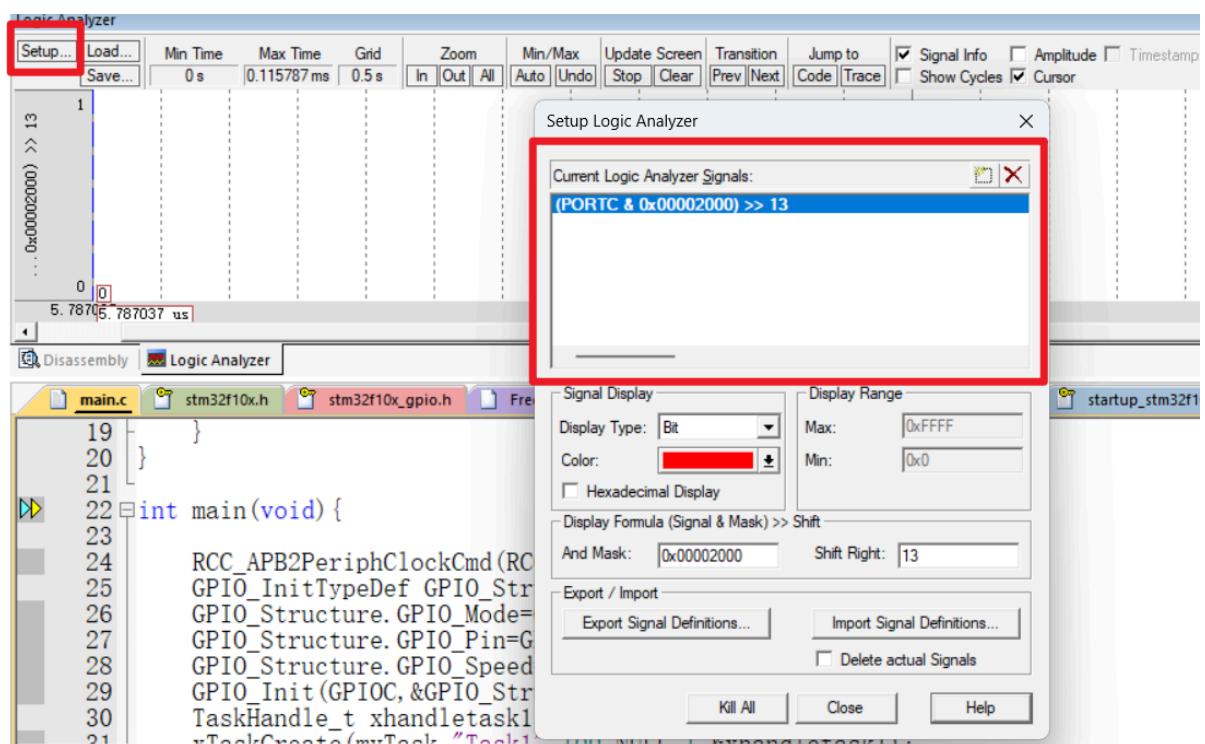
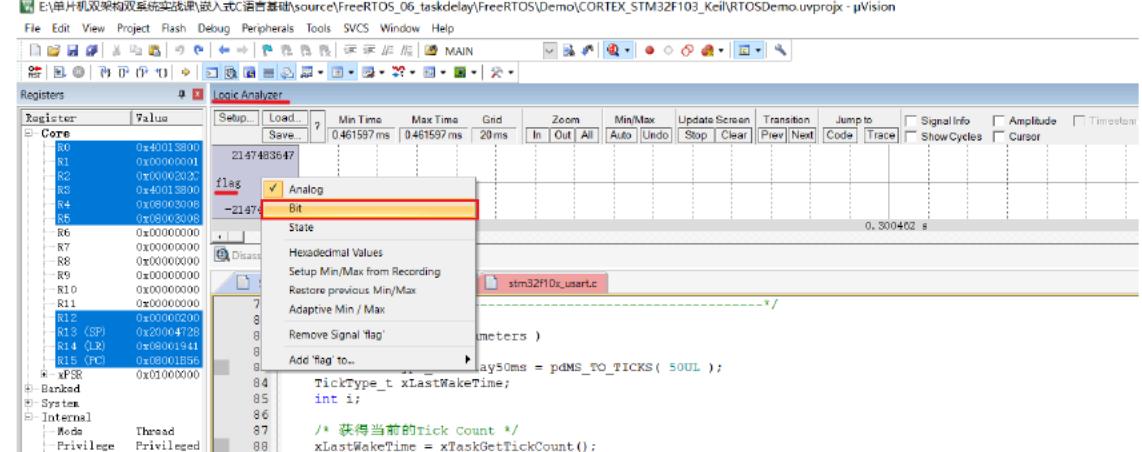
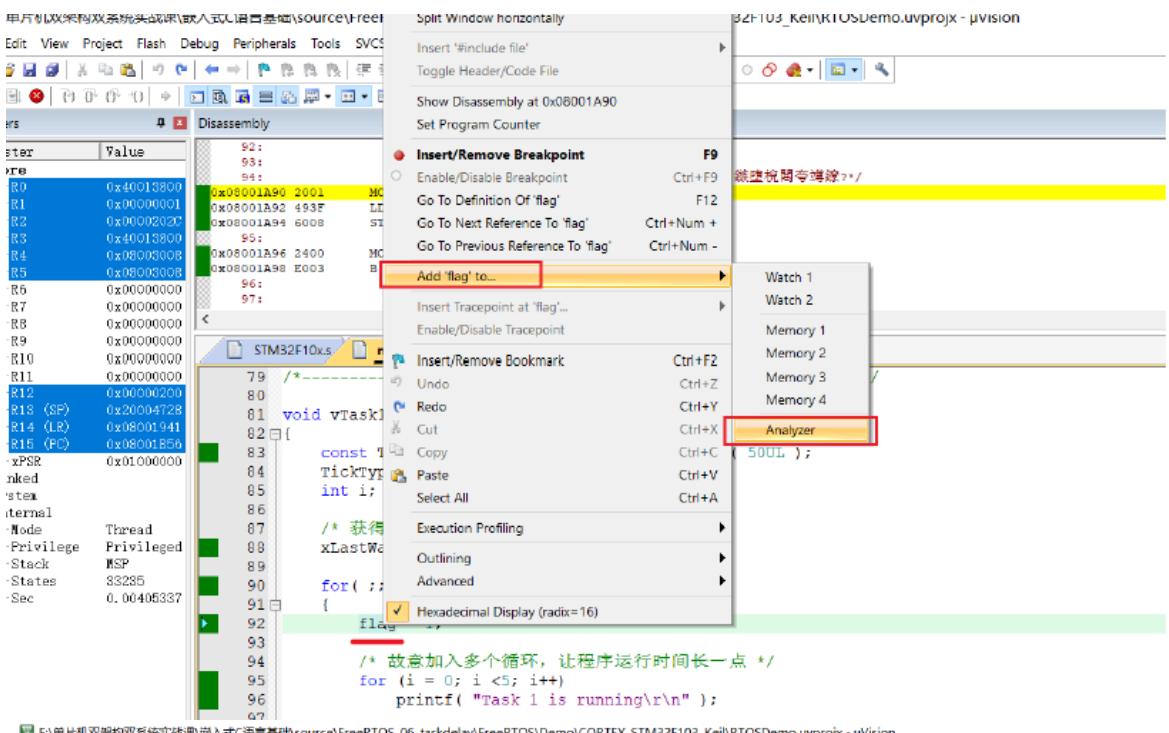
裸机分析仪：

常用的程序输出方式

串口：查看打印信息

逻辑分析仪：观察全局变量的波形，根据波形解析任务调度情况

1. 直接在代码中找到全局变量，右键添加到Analyzer，注意选择Bit观察
2. 也可以Setup创建添加



三、内存管理

后续的章节涉及这些内核对象：task、queue、semaphores和event group等。

为了让FreeRTOS更容易使用，这些内核对象一般都是动态分配：用到时分配，不使用时释放。

使用内存的动态管理功能，简化了程序设计：不再需要小心翼翼地提前规划各类对象，简化API函数的涉及，甚至可以减少内存的使用。

内存的动态管理是C程序的知识范畴，并不属于FreeRTOS的知识范畴，但是它跟FreeRTOS关系是如此紧密，所以我们先讲解它。

在C语言的库函数中，有malloc、free等函数，但是在FreeRTOS中，它们不适用：

- 不适合用在资源紧缺的嵌入式系统中
- 这些函数的实现过于复杂、占据的代码空间太大
- 并非线程安全的(thread-safe)
- 运行有不确定性：每次调用这些函数时花费的时间可能都不相同
- 内存碎片化
- 使用不同的编译器时，需要进行复杂的配置
- 有时候难以调试

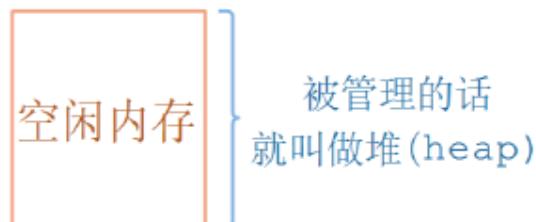
堆与栈

堆

堆就是一块空闲的内存。

堆，heap，就是一块空闲的内存，需要提供管理函数

- malloc：从堆里划出一块空间给程序使用
- free：用完后，再把它标记为空闲的，可以再次使用



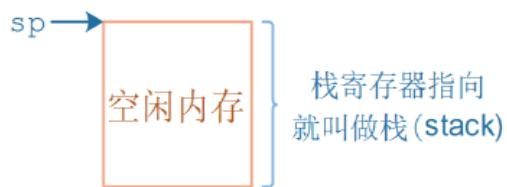
堆的一个简单实现

```
1 char heap_buf[1024]; //空闲内存
2 int pos = 0; //空闲内存首地址
3
4 void* My_malloc(int size){
5     int old_pos = pos;
6     pos += size;
7     return &heap_buf[old_pos];
8 }
9
10 void my_free(void *buf){
11     /*err*/
12 }
13
```

栈

栈，stack，函数调用时局部变量保存在栈中，当前程序的环境也是保存在栈中

- 可以从堆中分配一块空间用作栈



栈的作用：

情景：main 调用 a.fun a.fun 调用 b.fun

1. main调用a.fun时，需要记录函数返回的地址，将下一条指令的地址存入LR(Link Register),然后调用a.fun
 2. a.fun 调用b.fun时同理，将下一条指令存入LR, 调用b.fun

问题：LR会被覆盖 --> LR地址保存入栈

C函数的执行：

1. 划分栈空间(存入LR, 局部变量)
 2. LR, 存入栈
 3. 执行代码

LR存入的是返回地址，SP存入栈顶指针



SP一开始执行初始栈顶, LR(main返回地址), main的局部变量压入栈, 栈顶为SP=SP-N, N字节空间分配给main。

执行到a.fun时, SP=SP-M, LR(a.fun返回地址), a.fun的局部变量压入栈, 执行b.fun。M字节的空间分配给a.fun。

执行到b.fun时, 栈顶SP=SP-P, LR, b.fun压入栈, 执行b.fun, P字节的空间分配给b.fun。

每个任务都有自己的独立的栈空间。(任务即线程)

FreeRTOS中的内存管理

FreeRTOS中内存管理的接口函数为: pvPortMalloc、vPortFree, 对应于C库的malloc、free。

文件在FreeRTOS/Source/portable/MemMang下, 它也是放在portable目录下, 表示你可以提供自己的函数。

源码中默认提供了5个文件, 对应内存管理的5种方法。

文件	优点	缺点
heap_1.c	分配简单, 时间确定	只分配、不回收
heap_2.c	动态分配、最佳匹配	碎片、时间不定
heap_3.c	调用标准库函数	速度慢、时间不定
heap_4.c	相邻空闲内存可合并	可解决碎片问题、时间不定
heap_5.c	在 heap_4 基础上支持分隔的内存块	可解决碎片问题、时间不定

Heap_1

它只实现了pvPortMalloc, 没有实现vPortFree。

如果你的程序不需要删除内核对象, 那么可以使用heap_1:

- 实现最简单
- 没有碎片问题
- 一些要求非常严格的系统里, 不允许使用动态内存, 就可以使用heap_1

它的实现原理很简单，首先定义一个大数组，然后，对于pvPortMalloc调用时，从这个数组中分配空间。

```
/* Allocate the memory for the heap. */
##if ( configAPPLICATION_ALLOCATED_HEAP == 1 )

/* The application writer has already defined the array used for the RTOS
 * heap - probably so it can be placed in a special segment or address. */
extern uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];

##else
```

96

百问网

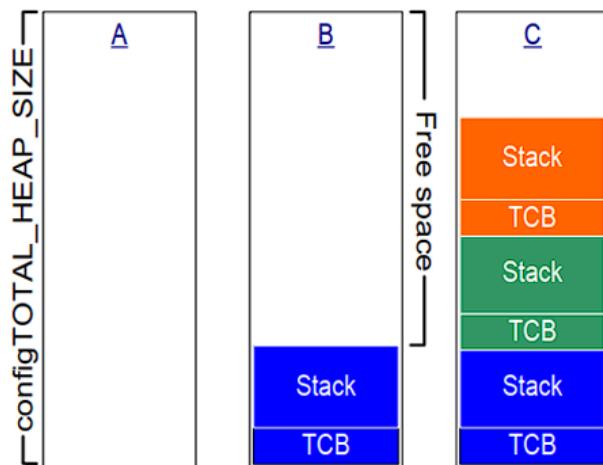
```
static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
##endif /* configAPPLICATION_ALLOCATED_HEAP */
```

分配示意图：

FreeRTOS在创建任务时，需要2个内核对象：task control block(TCB)、stack。

使用heap_1时，内存分配过程如下图所示：

- A：创建任务之前整个数组都是空闲的
- B：创建第1个任务之后，蓝色区域被分配出去了
- C：创建3个任务之后的数组使用情况



RAM being allocated from the heap_1 array each time a task is created

Heap_2

Heap_2之所以还保留，只是为了兼容以前的代码。新设计中不再推荐使用Heap_2。

建议使用Heap_4来替代Heap_2，更加高效。

Heap_2也是在数组上分配内存，跟Heap_1不一样的地方在于：

- Heap_2使用**最佳匹配算法(best fit)**来分配内存

- 它支持vPortFree

最佳匹配算法：

- 假设heap有3块空闲内存：5字节、25字节、100字节
- pvPortMalloc 想申请20字节
- 找出最小的、能满足pvPortMalloc的内存：25字节
- 把它划分为20字节、5字节
- 返回这20字节的地址
- 剩下的5字节仍然是空闲状态，留给后续的pvPortMalloc使用

与Heap_4相比，Heap_2不会合并相邻的空闲内存，所以Heap_2会导致严重的"碎片化"问题。

但是，如果申请、分配内存时大小总是相同的，这类场景下Heap_2没有碎片化的问题。

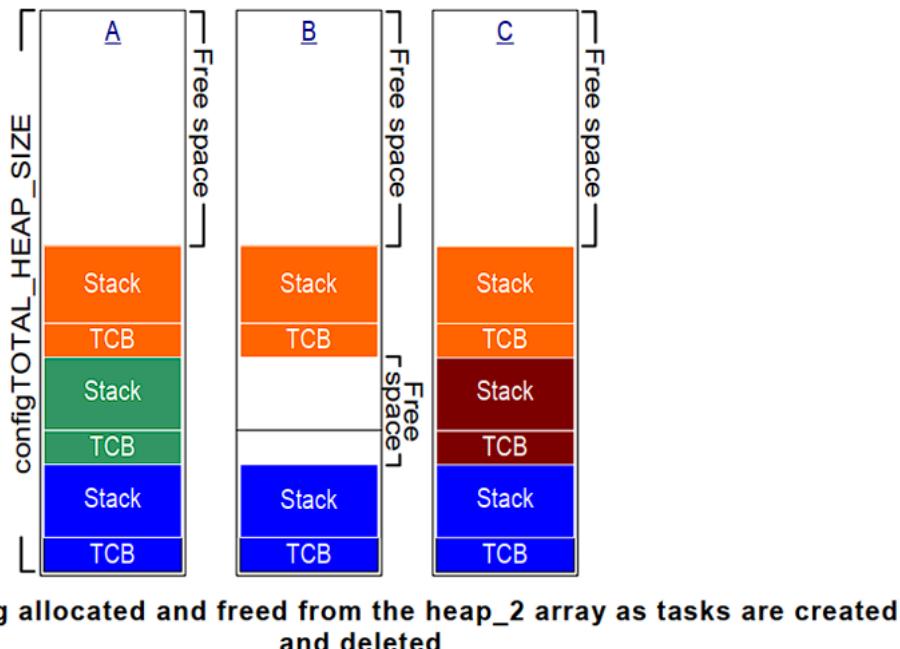
所以它适合这种场景：频繁地创建、删除任务，但是任务的栈大小都是相同的(创建任务时，需要分配TCB和栈，TCB总是一样的)。

虽然不再推荐使用heap_2，但是它的效率还是远高于malloc、free。

分配示意图：

使用heap_2时，内存分配过程如下图所示：

- A：创建了3个任务
- B：删除了一个任务，空闲内存有3部分：顶层的、被删除任务的TCB空间、被删除任务的Stack空间
- C：创建了一个新任务，因为TCB、栈大小跟前面被删除任务的TCB、栈大小一致，所以刚好分配到原来的内存



Heap_3

Heap_3 使用标准C库里的malloc、free函数，所以堆大小由链接器的配置决定，配置项 configTOTAL_HEAP_SIZE 不再起作用。

C库里的malloc、free函数并非线程安全的，Heap_3中先暂停FreeRTOS的调度器，再去调用这些函数，使用这种方法实现了线程安全。

Heap_4

跟Heap_1、Heap_2一样，Heap_4也是使用大数组来分配内存。

Heap_4使用首次适应算法(first fit)来分配内存。

它还会把相邻的空闲内存合并为一个更大的空闲内存，这有助于较少内存的碎片问题。

首次适应算法：

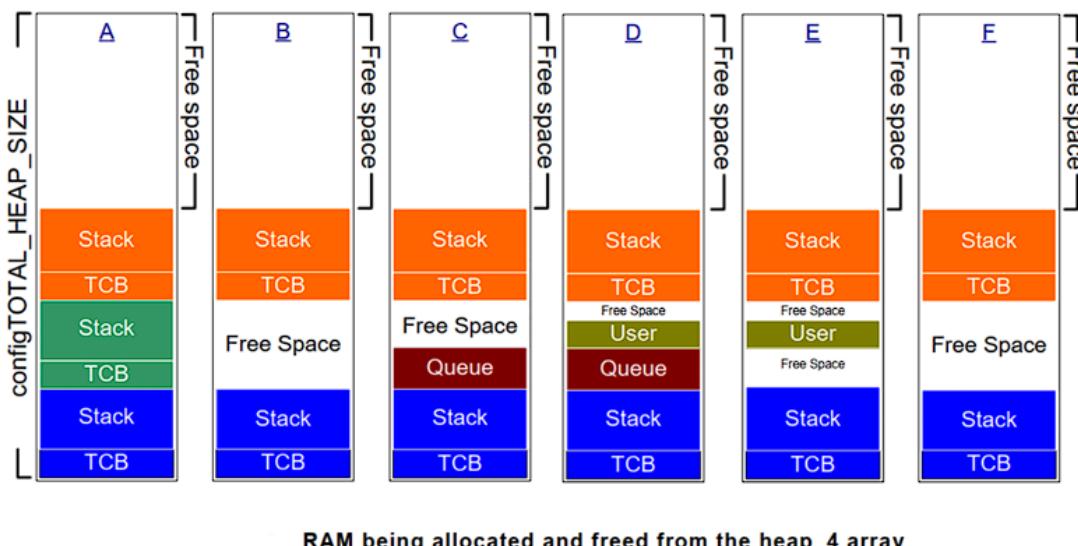
- 假设堆中有3块空闲内存：5字节、200字节、100字节
- pvPortMalloc 想申请20字节
- 找出第1个能满足pvPortMalloc的内存：200字节
- 把它划分为20字节、180字节
- 返回这20字节的地址
- 剩下的180字节仍然是空闲状态，留给后续的pvPortMalloc使用

Heap_4会把相邻空闲内存合并为一个大的空闲内存，可以较少内存的碎片化问题。

适用于这种场景：频繁地分配、释放不同大小的内存。

Heap_4的使用过程举例如下：

- A: 创建了3个任务
- B: 删除了一个任务，空闲内存有2部分：
 - 顶层的
 - 被删除任务的TCB空间、被删除任务的Stack空间合并起来的
- C: 分配了一个Queue，从第1个空闲块中分配空间
- D: 分配了一个User数据，从Queue之后的空闲块中分配
- E: 释放的Queue，User前后都有一块空闲内存
- F: 释放了User数据，User前后的内存、User本身占据的内存，合并为一个大的空闲内存



RAM being allocated and freed from the heap_4 array

Heap_4执行的时间是不确定的，但是它的效率高于标准库的malloc、free。

Heap_5

Heap_5 分配内存、释放内存的算法跟Heap_4是一样的。

相比于Heap_4，Heap_5并不局限于管理一个大数组：它可以管理多块、分隔开的内存。

在嵌入式系统中，内存的地址可能并不连续，这种场景下可以使用Heap_5。

既然内存时分隔开的，那么就需要进行初始化：确定这些内存块在哪、多大：

- 在使用pvPortMalloc之前，必须先指定内存块的信息
- 使用vPortDefineHeapRegions 来指定这些信息

指定一块内存：

怎么指定一块内存？使用如下结构体：

```
typedef struct HeapRegion
{
    uint8_t * pucStartAddress; // 起始地址
    size_t xSizeInBytes;      // 大小
} HeapRegion_t;
```

指定多块内存：

怎么指定多块内存？使用一个HeapRegion_t数组，在这个数组中，低地址在前、高地址在后。

比如：

```
HeapRegion_t xHeapRegions[] =
{
    { ( uint8_t * ) 0x80000000UL, 0x10000 }, // 起始地址0x80000000, 大小0x10000
    { ( uint8_t * ) 0x90000000UL, 0xa0000 }, // 起始地址0x90000000, 大小0xa0000
    { NULL, 0 } // 表示数组结束
```

100

百问网

```
};
```

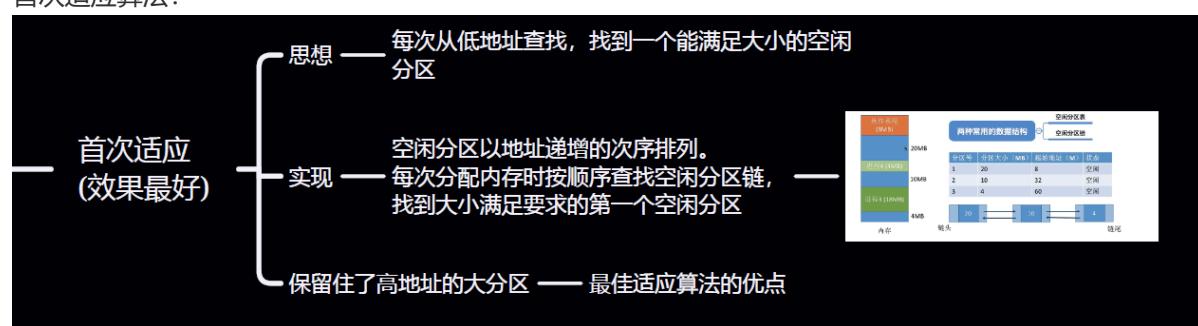
vPortDefineHeapRegions函数原型如下：

```
void vPortDefineHeapRegions( const HeapRegion_t * const pxHeapRegions );
```

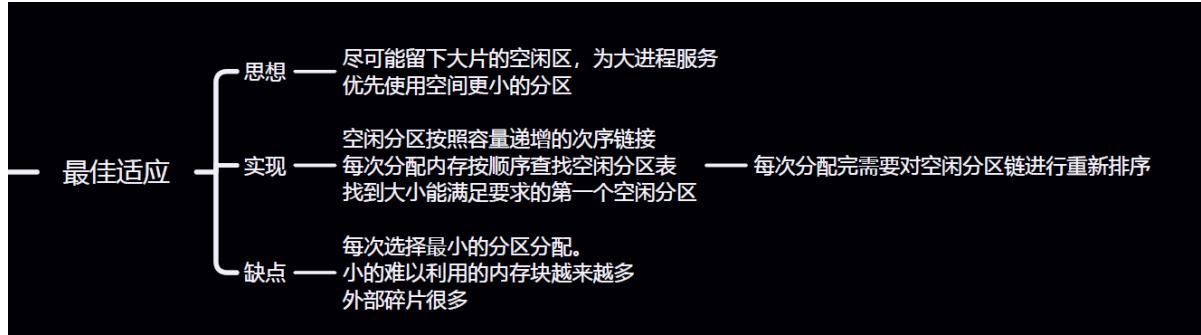
把 xHeapRegions 数组传给 vPortDefineHeapRegions 函数，即可初始化 Heap_5。

Os中动态内存管理办法

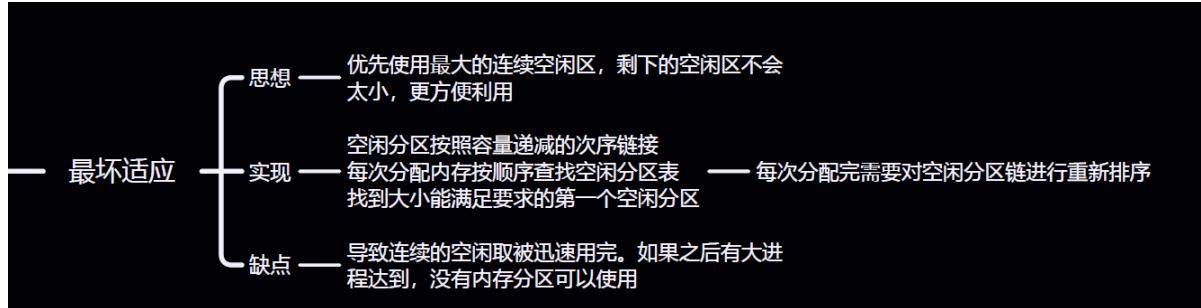
首次适应算法：



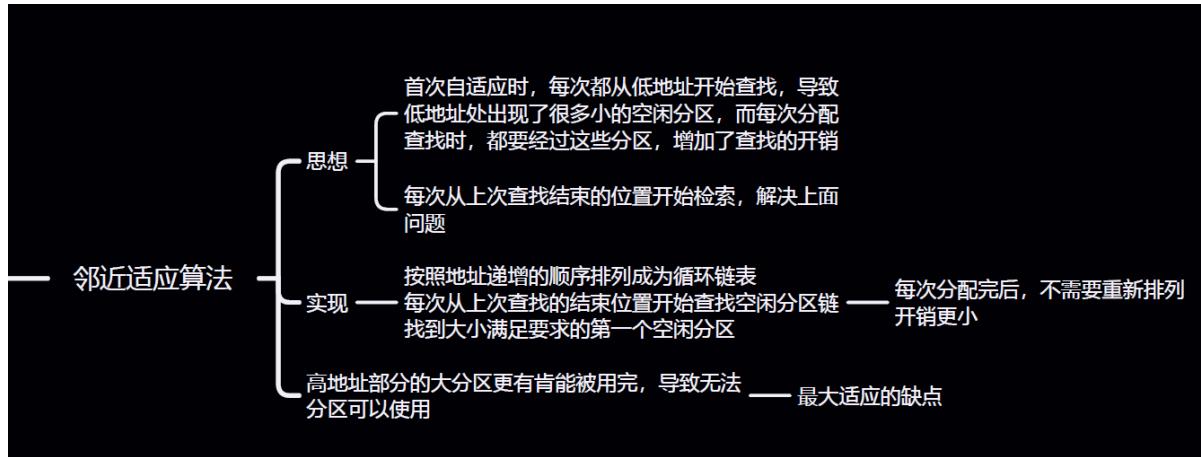
最佳适应法：



最坏适应：



邻近适应算法：



Heap相关函数

pvPortMalloc与vPortFree

```
1 | void * pvPortMalloc( size_t xWantedSize );
2 | void vPortFree( void * pv );
```

作用：分配内存、释放内存。

如果分配内存不成功，则返回值为NULL。

xPortGetFreeHeapSize

```
1 | size_t xPortGetFreeHeapSize( void );
```

当前还有多少空闲内存，这函数可以用来优化内存的使用情况。比如当所有内核对象都分配好后，执行此函数返回2000，那么configTOTAL_HEAP_SIZE就可减小2000。

xPortGetMinimumEverFreeHeapSize

```
1 | size_t xPortGetMinimumEverFreeHeapSize( void );
```

返回：程序运行过程中，空闲内存容量的最小值。

注意：只有heap_4、heap_5支持此函数。

malloc 失败的钩子函数

```
1 | void * pvPortMalloc( size_t xWantedSize )vPortDefineHeapRegions
2 | {
3 | .....
4 | #if ( configUSE_MALLOC_FAILED_HOOK == 1 )
5 |
6 | {
7 | if( pvReturn == NULL )
8 | {
9 | extern void vApplicationMallocFailedHook( void );
10| vApplicationMallocFailedHook();
11| }
12| }
13| #endif
14| return pvReturn;
15| }
```

所以，如果想使用这个钩子函数：

- 在FreeRTOSConfig.h 中，把configUSE_MALLOC_FAILED_HOOK 定义为1
- 提供vApplicationMallocFailedHook 函数
- pvPortMalloc 失败时，才会调用此函数

四、任务管理

简介：

在本章中，会涉及如下内容：

- FreeRTOS如何给每个任务分配CPU时间
- 如何选择某个任务来运行
- 任务优先级如何起作用
- 任务有哪些状态
- 如何实现任务
- 如何使用任务参数
- 怎么修改任务优先级
- 怎么删除任务
- 怎么实现周期性的任务
- 如何使用空闲任务

任务创建与删除

在FreeRTOS 中，任务就是一个函数，原型如下：

```
void ATTaskFunction( void *pvParameters );
```

要注意的是：

- 这个函数不能返回
- 同一个函数，可以用来创建多个任务；换句话说，多个任务可以运行同一个函数
- 函数内部，尽量使用局部变量：
 - 每个任务都有自己的栈
 - 每个任务运行这个函数时

- | | |
|---|-------------------------------------|
| 1 | ◆ 任务A的局部变量放在任务A的栈里、任务B的局部变量放在任务B的栈里 |
| 2 | ◆ 不同任务的局部变量，有自己的副本 |

■ 函数使用全局变量、静态变量的话

- | | |
|---|------------------------|
| 1 | ◆ 只有一个副本：多个任务使用的是同一个副本 |
| 2 | ◆ 要防止冲突(后续会讲) |

示例：

```
1 void ATaskFunction( void *pvParameters )
2 {
3     /* 对于不同的任务，局部变量放在任务的栈里，有各自的副本 */
4     int32_t lVariableExample = 0;
5     /* 任务函数通常实现为一个无限循环 */
6     for( ;; )
7     {
8         /* 任务的代码 */
9     }
10    /* 如果程序从循环中退出，一定要使用vTaskDelete删除自己
11    * NULL表示删除的是自己
12    */
13    vTaskDelete( NULL );
14    /* 程序不会执行到这里，如果执行到这里就出错了 */
15 }
```

任务创建

创建任务时可以使用2个函数：动态分配内存、静态分配内存。

使用动态分配内存的函数如下：

```
BaseType_t xTaskCreate(
    TaskFunction_t pxTaskCode, // 函数指针，任务函数
    const char * const pcName, // 任务的名字
    const configSTACK_DEPTH_TYPE usStackDepth, // 栈大小，单位为word, 10表示40字节
    void * const pvParameters, // 调用任务函数时传入的参数
    UBaseType_t uxPriority, // 优先级
    TaskHandle_t * const pxCreatedTask ); // 任务句柄，以后使用它来操作这个任务
```

一字代表32位，4B

参数说明：

参数	描述
pvTaskCode	函数指针，可以简单地认为任务就是一个 C 函数。 它稍微特殊一点：永远不退出，或者退出时要调用 “vTaskDelete(NULL)”
pcName	任务的名字，FreeRTOS 内部不使用它，仅仅起调试作用。 长度为： configMAX_TASK_NAME_LEN
usStackDepth	每个任务都有自己的栈，这里指定栈大小。 单位是 word，比如传入 100，表示栈大小为 100 word，也就是 400 字节。 最大值为 uint16_t 的最大值。 怎么确定栈的大小，并不容易，很多时候是估计。 精确的办法是看反汇编码。
pvParameters	调用 pvTaskCode 函数指针时用到：pvTaskCode(pvParameters)
uxPriority	优先级范围：0~(configMAX_PRIORITIES - 1) 数值越小优先级越低， 如果传入过大的值，xTaskCreate 会把它调整为 (configMAX_PRIORITIES - 1)
pxCreatedTask	用来保存 xTaskCreate 的输出结果：task handle。 以后如果想操作这个任务，比如修改它的优先级，就需要这个 handle。 如果不使用该 handle，可以传入 NULL。
返回值	成功：pdPASS； 失败：errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY(失败原因只有内存 不足)

使用静态分配内存的函数如下：

```
TaskHandle_t xTaskCreateStatic (
    TaskFunction_t pxTaskCode,      // 函数指针，任务函数
    const char * const pcName,      // 任务的名字
    const uint32_t ulStackDepth,    // 栈大小，单位为word, 10表示40字节
    void * const pvParameters,     // 调用任务函数时传入的参数
    UBaseType_t uxPriority,        // 优先级
    StackType_t * const puxStackBuffer, // 静态分配的栈，就是一个buffer
    StaticTask_t * const pxTaskBuffer // 静态分配的任务结构体的指针，用它来操作这个任务
);
```

参数说明：

相比于使用动态分配内存创建任务的函数，最后2个参数不一样：

参数	描述
pvTaskCode	函数指针，可以简单地认为任务就是一个 C 函数。 它稍微特殊一点：永远不退出，或者退出时要调用 “vTaskDelete(NULL)”
pcName	任务的名字，FreeRTOS 内部不使用它，仅仅起调试作用。 长度为：configMAX_TASK_NAME_LEN
usStackDepth	每个任务都有自己的栈，这里指定栈大小。 单位是 word，比如传入 100，表示栈大小为 100 word，也就是 400 字节。 最大值为 uint16_t 的最大值。 怎么确定栈的大小，并不容易，很多时候是估计。 精确的办法是看反汇编码。
pvParameters	调用 pvTaskCode 函数指针时用到：pvTaskCode(pvParameters)
uxPriority	优先级范围：0~(configMAX_PRIORITIES - 1) 数值越小优先级越低， 如果传入过大的值，xTaskCreate 会把它调整为 (configMAX_PRIORITIES - 1)
pxxStackBuffer	静态分配的栈内存，比如可以传入一个数组， 它的大小是 usStackDepth*4。
pxTaskBuffer	静态分配的 StaticTask_t 结构体的指针
返回值	成功：返回任务句柄； 失败：NULL

静态创建，需要TCB结构体与栈提前分配

采用静态分配需要增加宏开关， configSUPPORT_STATIC_ALLOCATION

还需要加写vApplicationGetIdleTaskMemory()函数

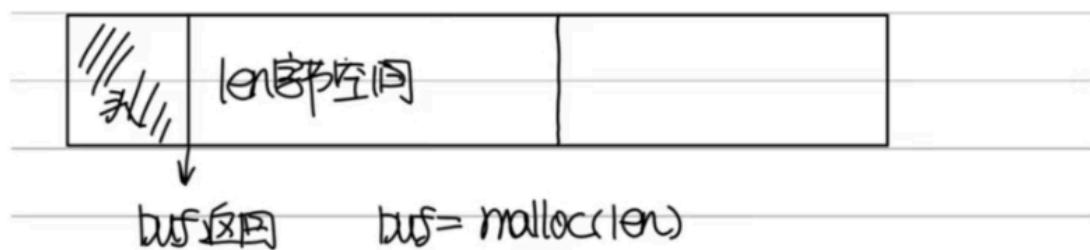
对于xTaskCreateStatic()静态创建任务，若令其回收必须记录其返回值。

动态创建将TCB存入handler参数中。

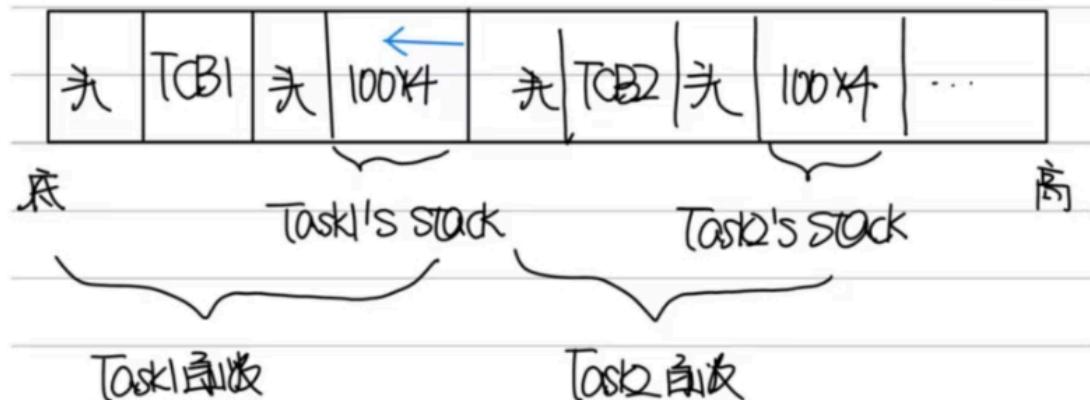
多个任务可以使用同一个函数，怎么体现它们的差别？

- 栈不同
- 创建任务时可以传入不同的参数

malloc分配内存：



头部部分有长度信息，可用头部中 len 中 free 空间



被溢出后会覆盖头 TCB ... 寻找释放前缀

volatile 变量可变关键字，不允许编译器优化。

补充：静态创建与动态创建的区别

- 1 动态创建任务使用 `xTaskCreate` 函数来创建任务。
- 2 该函数在运行时从堆中分配内存来存储任务控制块（TCB）和任务堆栈。
- 3 优点：
 - 4 简单易用，代码更简洁。
 - 5 内存分配在运行时进行，灵活性更高。
- 6 缺点：
 - 7 依赖于堆内存管理，可能会导致内存碎片问题。
 - 8 如果堆内存不足，任务创建会失败。
- 9
- 10 静态创建任务使用 `xTaskCreateStatic` 函数来创建任务。
- 11 该函数需要在编译时分配内存来存储任务控制块（TCB）和任务堆栈。
- 12 优点：
 - 13 不依赖于堆内存管理，避免了内存碎片问题。
 - 14 更适合内存受限的嵌入式系统。
- 15 缺点：
 - 16 需要在编译时确定内存分配，灵活性较低。
 - 17 代码相对复杂，需要手动管理内存。

任务删除

删除任务时使用的函数如下：

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

参数说明：

参数	描述
pvTaskCode	任务句柄，使用 xTaskCreate 创建任务时可以得到一个句柄。 也可传入 NULL，这表示删除自己。

怎么删除任务？举个不好的例子：

- 自杀： `vTaskDelete(NULL)`
- 被杀：别的任务执行 `vTaskDelete(pvTaskCode)`, pvTaskCode 是自己的句柄
- 杀人：执行 `vTaskDelete(pvTaskCode)`, pvTaskCode 是别的任务的句柄

任务优先级与Tick

任务优先级

高优先级的任务先运行

优先级的取值范围是：0~(configMAX_PRIORITIES - 1)，数值越大优先级越高

FreeRTOS的调度器可以使用2种方法来快速找出优先级最高的、可以运行的任务。

使用不同的方法时， configMAX_PRIORITIES 的取值有所不同。

通用方法：

使用C函数实现，对所有的架构都是同样的代码。

对configMAX_PRIORITIES的取值没有限制。

但是configMAX_PRIORITIES的取值还是尽量小，因为取值越大越浪费内存，也浪费时间。

configUSE_PORT_OPTIMISED_TASK_SELECTION被定义为0、或者未定义时，使用此方法。

架构相关的优化的方法：

架构相关的汇编指令，可以从一个32位的数里快速地找出为1的最高位。

使用这些指令，可以快速找出优先级最高的、可以运行的任务。

使用这种方法时， configMAX_PRIORITIES 的取值不能超过32。

configUSE_PORT_OPTIMISED_TASK_SELECTION被定义为1时，使用此方法。

FreeRTOS会确保最高优先级的、可运行的任务，马上就能执行

对于相同优先级的、可运行的任务，轮流执行

tick

FreeRTOS中也有心跳，它使用定时器产生固定间隔的中断。

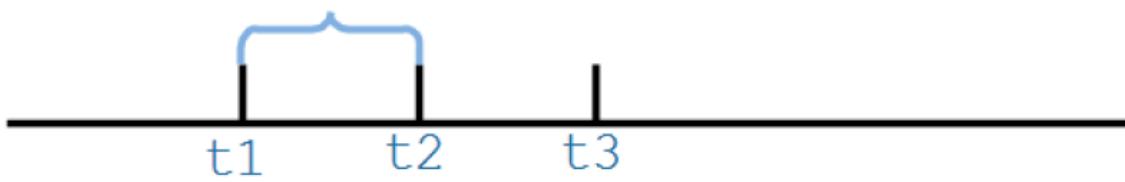
这叫Tick、滴答，比如每10ms发生一次时钟中断。

假设t1、t2、t3发生时钟中断

两次中断之间的时间被称为时间片(time slice、 tick period)

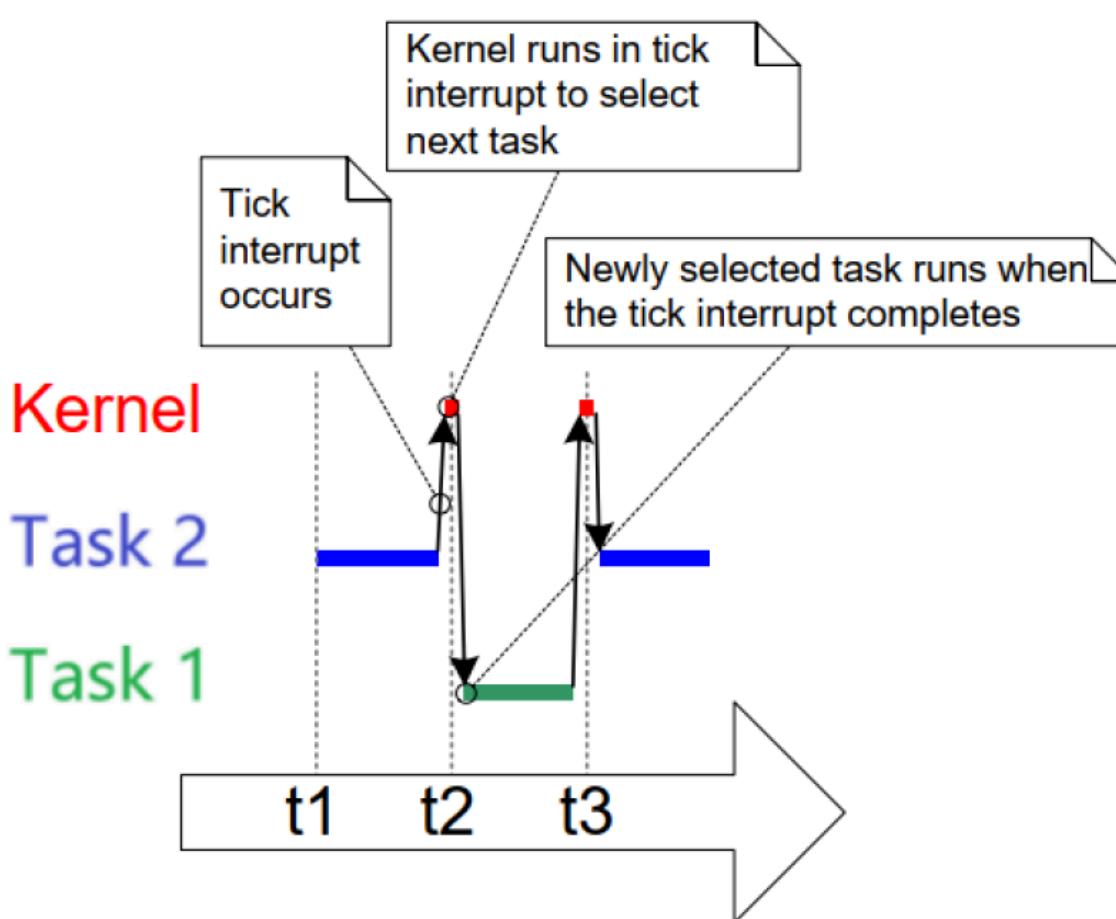
时间片的长度由configTICK_RATE_HZ 决定，假设configTICK_RATE_HZ为100，那么时间片长度就是10ms

时间片 time slice



相同优先级任务的切换：

1. 任务2从t1执行到t2
2. 在t2发生tick中断，进入tick中断处理函数：
选择下一个要运行的任务
执行完中断处理函数后，切换到新的任务：任务1
3. 任务1从t2执行到t3
4. 从图中可以看出，任务运行的时间并不是严格从t1,t2,t3哪里开始



有了Tick的概念后，我们就可以使用Tick来衡量时间了，比如：

```
1 vTaskDelay(2); // 等待2个Tick, 假设configTICK_RATE_HZ=100, Tick周期时10ms, 等待20ms
2
3 // 还可以使用pdMS_TO_TICKS宏把ms转换为tick
4 vTaskDelay(pdMS_TO_TICKS(100)); // 等待100ms
```

注意，基于Tick实现的延时并不精确，比如vTaskDelay(2) 的本意是延迟2个Tick周期，有可能经过1个Tick多一点就返回了。

使用vTaskDelay函数时，建议以ms为单位，使用pdMS_TO_TICKS把时间转换为Tick。
这样的代码就与configTICK_RATE_HZ无关，即使配置项configTICK_RATE_HZ改变了，我们也不用去修改代码。

优先级修改

使用uxTaskPriorityGet来获得任务的优先级：

```
1 | UBaseType_t uxTaskPriorityGet( const TaskHandle_t xTask );
```

使用参数xTask来指定任务，设置为NULL表示获取自己的优先级。

使用vTaskPrioritySet 来设置任务的优先级：

```
1 | void vTaskPrioritySet( TaskHandle_t xTask, UBaseType_t uxNewPriority );
```

使用参数xTask来指定任务，设置为NULL表示设置自己的优先级；

参数uxNewPriority表示新的优先级，取值范围是0~(configMAX_PRIORITIES - 1)。

任务状态

以前我们很简单地把任务的状态分为2中：运行(Running)、非运行(Not Running)。

但是非运行态也可以细分

Task3执行vTaskDelay后：处于非运行状态，要过3秒种才能再次运行

Task3运行期间，Task1、Task2也处于非运行状态，但是它们随时可以运行

这两种"非运行"状态就不一样，可以细分为：

阻塞状态(Blocked)

暂停状态(Suspended)

就绪状态(Ready)

阻塞状态

在实际产品中，我们不会让一个任务一直运行，而是使用"事件驱动"的方法让它运行：

任务要等待某个事件，事件发生后它才能运行

在等待事件过程中，它不消耗CPU资源

在等待事件的过程中，这个任务就处于阻塞状态(Blocked) 等待某个资源

在阻塞状态的任务，它可以等待两种类型的事件：

时间相关的事件

可以等待一段时间：我等2分钟

也可以一直等待，直到某个绝对时间：我等到下午3点

同步事件：这事件由别的任务，或者是中断程序产生

例子1：任务A等待任务B给它发送数据

例子2：任务A等待用户按下按键

同步事件的来源有很多(这些概念在后面会细讲)：

队列(queue)

二进制信号量(binary semaphores)

计数信号量(counting semaphores)

互斥量(mutexes)

递归互斥量、递归锁(recursive mutexes)

事件组(event groups)

任务通知(task notifications)

在等待一个同步事件时，可以加上超时时间。比如等待队列数据，超时时间设为10ms：

10ms之内有数据到来：成功返回

10ms到了，还是没有数据：超时返回

暂停状态

FreeRTOS中的任务也可以进入暂停状态，唯一的方法是通过vTaskSuspend函数。

```
1 | void vTaskSuspend( TaskHandle_t xTaskToSuspend );
```

参数xTaskToSuspend表示要暂停的任务，如果为NULL，表示暂停自己。

要退出暂停状态，只能由别人来操作：

别的任务调用：vTaskResume

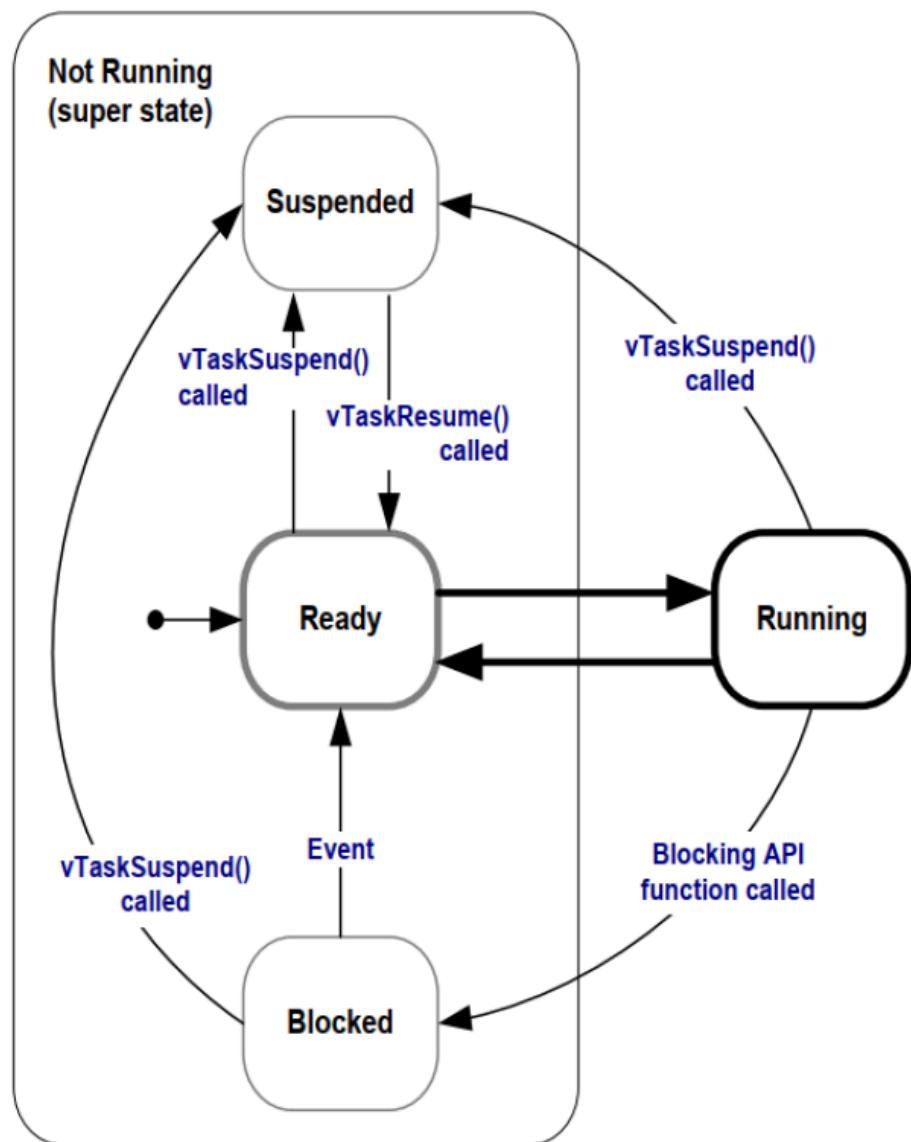
中断程序调用：xTaskResumeFromISR

实际开发中，暂停状态用得不多。

就绪状态

这个任务完全准备好了，随时可以运行：只是还轮不到它。这时，它就处于就绪态(Ready)。

状态转移图



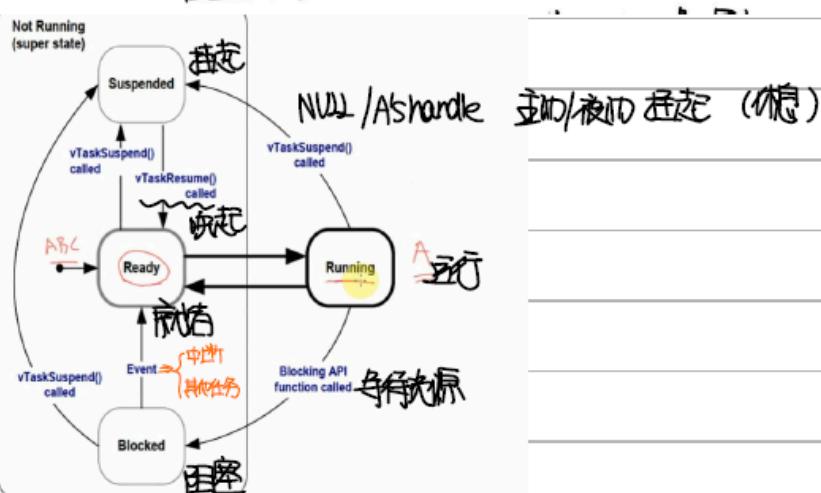
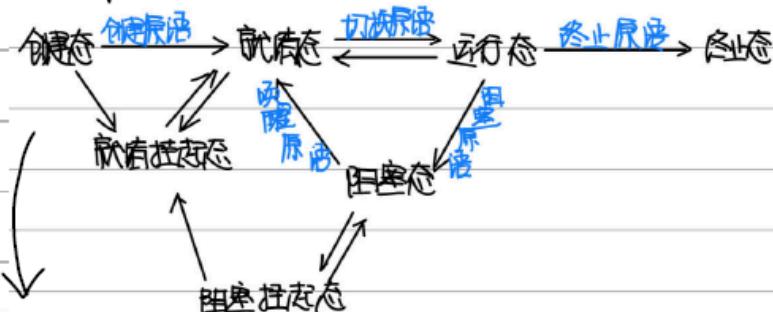
操作系统七状态模型：

Running 立行

Ready 就绪

Blocked (等待某事) 阻塞

Suspended 挂起



补充函数：获取TickCount函数 t=xTaskGetTockCount()

Delay函数

有两个Delay函数：

vTaskDelay：至少等待指定个数的Tick Interrupt才能变为就绪状态

vTaskDelayUntil：等待到指定的绝对时刻，才能变为就绪态。

函数原型：

```

1 void vTaskDelay( const TickType_t xTicksToDelay );
2 /* xTicksToDelay: 等待多少给Tick */
3
4 /* pxPreviousWakeTime: 上一次被唤醒的时间
5 * xTimeIncrement: 要阻塞到(pxPreviousWakeTime + xTimeIncrement)
6 * 单位都是Tick Count
7 */
8 BaseType_t xTaskDelayUntil( TickType_t * const pxPreviousWakeTime, const
    TickType_t xTimeIncrement );

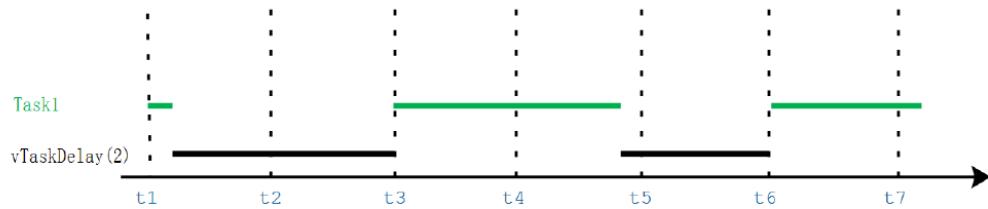
```

说明：

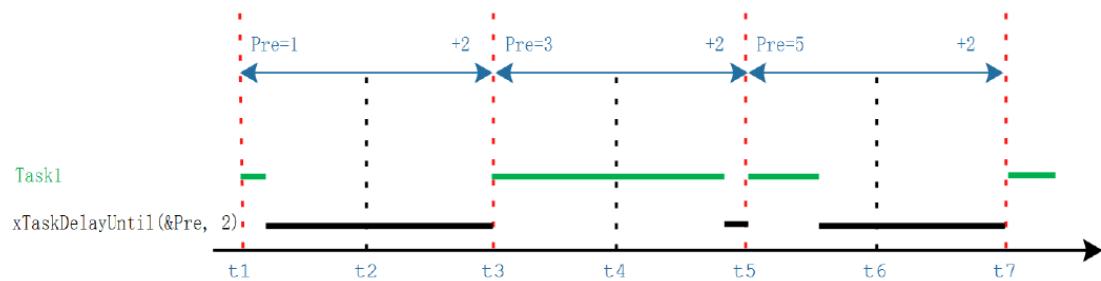
使用vTaskDelay(n)时，进入、退出vTaskDelay的时间间隔至少是n个Tick中断

使用xTaskDelayUntil(&Pre, n)时，前后两次退出xTaskDelayUntil的时间至少是n个Tick中断
退出xTaskDelayUntil时任务就进入的就绪状态，一般都能得到执行机会
所以可以使用xTaskDelayUntil来让任务周期性地运行

图解：



vTaskDelay的单位是tickcount，每次Delay的时间不定，需要转化为ms

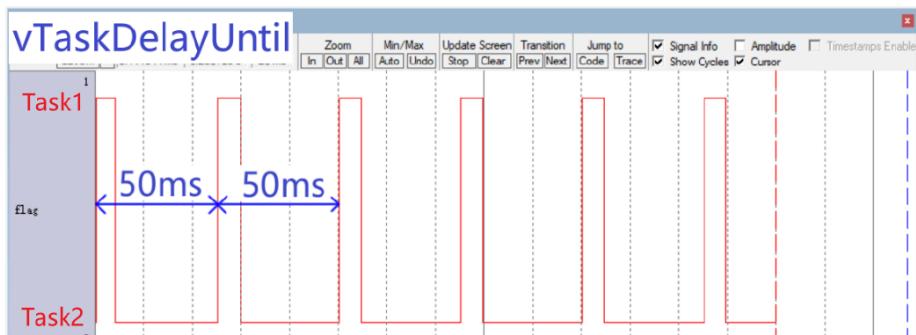
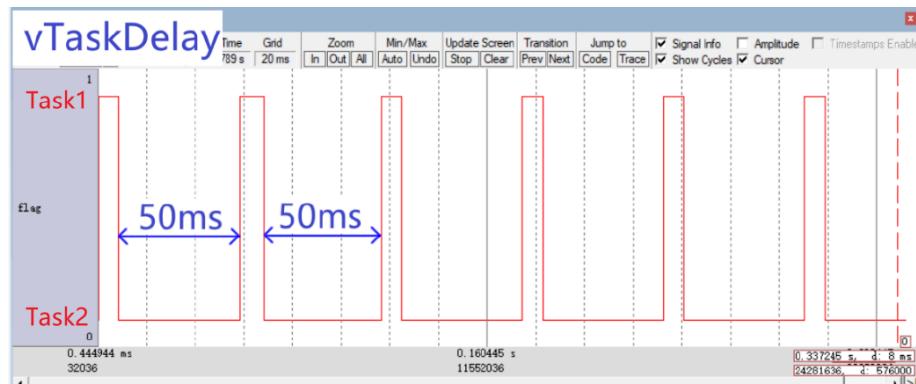


首个Pre需要自己设定，此后的可以自行更新

```
①直调pxPre + xTime 后退出  
②自动更新 pxPre = pxPre + xTime (自动更新传入拍针常量)
```

Demo示例：

```
1 void TaskFunc1(void *param)
2 {
3     TickType_t tStart = xTaskGetTickCount();
4     int i = 0;
5     int j = 0;
6     while (1)
7     {
8         task1flag = 1;
9         task2flag = 0;
10        task3flag = 0;
11
12        for (i = 0; i < rands[j]; i++)
13        {
14            printf("111");
15        }
16        j++;
17        j = j % 8;
18        // if (j == 8)
19        // j = 0; // j复位
20 #if swt
21         vTaskDelay(20);
22 #else
23         vTaskDelayUntil(&tStart, 20);
24 #endif
25    }
26 }
```



空闲任务及其钩子函数

介绍

空闲任务(Idle任务)的作用：释放被删除的任务的内存。

一个良好的程序，它的任务都是事件驱动的：平时大部分时间处于阻塞状态。

有可能我们自己创建的所有任务都无法执行，但是调度器必须能找到一个可以运行的任务：所以，我们要提供空闲任务。

在使用vTaskStartScheduler() 函数来创建、启动调度器时，这个函数内部会创建空闲任务。

空闲任务优先级为0：它不能阻碍用户任务运行

空闲任务要么处于就绪态，要么处于运行态，永远不会阻塞

空闲任务的优先级为0，这以为着一旦某个用户的任务变为就绪态，那么空闲任务马上被切换出去，让这个用户任务运行。

在这种情况下，我们说用户任务"抢占"(pre-empt)了空闲任务，这是由调度器实现的。

要注意的是：如果使用vTaskDelete() 来删除任务，那么你就要确保空闲任务有机会执行，否则就无法释放被删除任务的内存。

补充： vTaskDelete(句柄) 不需要IdleTask回收内存，NULL自杀时需要IdleTask回收内存。
(保证IdleTask有执行的机会)

钩子函数

FreeRTOS钩子函数 (Hook Functions) 是用户可以定义的函数，这些函数在特定事件发生时由**FreeRTOS内核**调用。

常见的钩子函数包括：

空闲钩子函数 (Idle Hook) : 当系统处于空闲状态时调用。

堆栈溢出钩子函数 (Stack Overflow Hook) : 当检测到任务堆栈溢出时调用。

内存分配失败钩子函数 (Malloc Failed Hook) : 当内存分配失败时调用。

任务切换钩子函数 (Task Switch Hook) : 在任务切换时调用。

这些钩子函数允许用户在特定事件发生时执行自定义代码，从而增强系统的可调试性和可靠性。

可以添加一个空闲任务的钩子函数(Idle Task Hook Functions)，空闲任务的循环没执行一次，就会调用一次钩子函数：

1. 执行一些低优先级的、后台的、需要连续执行的函数
2. 测量系统的空闲时间：空闲任务能被执行就意味着所有的高优先级任务都停止了，所以测量空闲任务占据的时间，就可以算出处理器占用率。
3. 让系统进入省电模式：空闲任务能被执行就意味着没有重要的事情要做，当然可以进入省电模式了。

钩子任务的限制：

不能导致空闲任务进入阻塞状态、暂停状态(idleTask只有就绪态和运行态)

如果你会使用vTaskDelete() 来删除任务，那么钩子函数要非常高效地执行。

如果空闲任务移植卡在钩子函数里的话，它就无法释放内存。

如果 vApplicationIdleHook 中的代码执行时间过长，Idle Task 就会长时间被占用，从而无法及时释放被删除任务的内存。

钩子函数使用的前提

在FreeRTOS\Source\tasks.c 中，可以看到如下代码，所以前提就是：

1. 把这个宏定义为1： configUSE_IDLE_HOOK
2. 实现vApplicationIdleHook 函数

```

#ifndef configUSE_IDLE_HOOK
{
    extern void vApplicationIdleHook( void );

    /* Call the user defined function from within the idle task. This
     * allows the application designer to add background functionality
     * without the overhead of a separate task.
     * NOTE: vApplicationIdleHook() MUST NOT, UNDER ANY CIRCUMSTANCES,
     * CALL A FUNCTION THAT MIGHT BLOCK. */
    vApplicationIdleHook();
}
#endif /* configUSE_IDLE_HOOK */

```

调度算法

基本概念

正在运行的任务，被称为"正在使用处理器"，它处于运行状态。

在单处理系统中，任何时间里只能有一个任务处于运行状态。

非运行状态的任务，它处于这3中状态之一：阻塞(Blocked)、暂停(Suspended)、就绪(Ready)。

就绪态的任务，可以被调度器挑选出来切换为运行状态，调度器永远都是挑选最高优先级的就绪态任务并让它进入运行状态。

阻塞状态的任务，它在等待"事件"，当事件发生时任务就会进入就绪状态。

事件分为两类：时间相关的事件、同步事件。

所谓时间相关的事件，就是设置超时时间：在指定时间内阻塞，时间到了就进入就绪状态。使用时间相关的事件，可以实现周期性的功能、可以实现超时功能。

同步事件就是：某个任务在等待某些信息，别的任务或者中断服务程序会给它发送信息。

怎么"发送信息"？方法很多，有：任务通知(task notification)、队列(queue)、事件组(event group)、信号量(semaphoe)、互斥量(mutex)等。

这些方法用来发送同步信息，比如表示某个外设得到了数据。

配置调度算法

所谓调度算法，就是怎么确定哪个就绪态的任务可以切换为运行状态。

通过配置文件FreeRTOSConfig.h的两个配置项来配置调度算法

configUSE_PREEMPTION、

configUSE_TIME_SLICING

还有第三个配置项：configUSE_TICKLESS_IDLE，它是一个高级选项，用于关闭Tick中断来实现省电，后续单独讲解。

现在我们假设configUSE_TICKLESS_IDLE被设为0，先不使用这个功能。

调度算法的行为主要体现在两方面：

高优先级的任务先运行、同优先级的就绪态任务如何被选中。

调度算法要确保同优先级的就绪态任务，能"轮流"运行，策略是"轮转调度"(Round Robin Scheduling)。

轮转调度并不保证任务的运行时间是公平分配的，我们还可以细化时间的分配方法。

三个角度来配置调度算法：

可否抢占？高优先级的任务能否优先执行(配置项: **configUSE_PREEMPTION**)

1. 可以：被称作"可抢占调度"(Pre-emptive)，高优先级的就绪任务马上执行，下面再细化。

2. 不可以：不能抢就只能协商了，被称作“合作调度模式”(Co-operative Scheduling)

当前任务执行时，更高优先级的任务就绪了也不能马上运行，只能等待当前任务主动让出CPU资源。

其他同优先级的任务也只能等待：更高优先级的任务都不能抢占，平级的更应该老实点

可抢占的前提下，同优先级的任务是否轮流执行(配置项：**configUSE_TIME_SLICING**)

1. 轮流执行：被称为“时间片轮转”(Time Slicing)，同优先级的任务轮流执行，你执行一个时间片、我再执行一个时间片

2. 不轮流执行：英文为“without Time Slicing”，当前任务会一直执行，直到主动放弃、或者被高优先级任务抢占

在“可抢占”+“时间片轮转”的前提下，进一步细化：

空闲任务是否让步于用户任务(配置项：**configIDLE_SHOULD_YIELD**)

1. 空闲任务低人一等，每执行一次循环，就看看是否主动让位给用户任务

2. 空闲任务跟用户任务一样，大家轮流执行，没有谁更特殊

实现逻辑：

底层：

IdleTask()

{ while(1)

 循环执行区

 {

 xxx;

主动触发调度

 #if : YIELD

 // 触发调度 (主动调度调度)

 #endif

}

 否则继续执行区，直至 tick 中断被调用开启调度

各种配置列表：

配置项	A	B	C	D	E
configUSE_PREEMPTION	1	1	1	1	0
configUSE_TIME_SLICING	1	1	0	0	x
configIDLE_SHOULD_YIELD	1	0	1	0	x
说明	常用	很少用	很少用	很少用	几乎不用

注：

- A：可抢占+时间片轮转+空闲任务让步
- B：可抢占+时间片轮转+空闲任务不让步
- C：可抢占+非时间片轮转+空闲任务让步
- D：可抢占+非时间片轮转+空闲任务不让步
- E：合作调度

效果对比：

抢占时：高优先级任务就绪时，就可以马上执行

不抢占时：优先级失去意义了，既然不能抢占就只能协商了，任务1(优先级为0)一直在运行(没有主动放弃CPU)，其他任务都无法执行。

即使任务3(优先级2)的vTaskDelay 已经超时、即使它的优先级更高，都没办法执行。

时间片轮转：在Tick中断中会引起任务切换

时间片不轮转：高优先级任务就绪时会引起任务切换，高优先级任务不再运行时也会引起任务切换。

可以看到任务3就绪后可以马上执行，它运行完毕后导致任务切换。

其他时间没有任务切换，可以看到任务1、任务2都运行了很长时间。

让步时：在空闲任务的每个循环中，会主动让出处理器，从图中可以看到flagIdleTaskrun的波形很小

不让步时：空闲任务跟任务1、任务2同等待遇，它们的波形宽度是差不多的

OS调度算法整理

评价指标：

1. CPU利用率
2. 周转时间
3. 等待时间
4. 响应时间
5. 系统吞吐量

先来先服务

角度：公平角度

规则：按照进程/作业的到达顺序进行服务

作用：

1. 作业调度
2. 进程调度

方式：非抢占式，只能主动放弃CPU

缺点：对排在长作业后的短作业，带权周转时间很大，对短作业体验非常不好，FCFS对长作业有利，对短作业不利
会导致饥饿问题

短作业优先

角度：

- 追求最少得平均等待时间
- 追求最少得平均周转时间
- 追求最少得平均带权周转时间

规则：最短的作业/进程先得到服务

方式：

- 默认非抢占式，只能等待主动放弃CPU
- 抢占式，按照最短剩余时间排序

缺点：对短作业有利，长作业不利，会产生饥饿

优点：最短的平均等待时间和平均周转时间

高响应比优先算法

角度：综合考虑作业/进程的等待时间和要求服务的时间

规则：调度时先计算各个作业的响应比，选择响应比最高的作业/进程进行服务

$$\text{响应比} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$$

作用：进程调度 作业调度

方式：非抢占式，只能等待主动放弃CPU

优点：

1. 综合考虑了等待时间和运行时间(要求服务时间)
2. 等待时间相同时，要求服务时间短的优先
3. 要求服务时间一样的话，等待时间短的优先
4. 避免了饥饿问题

以上三种方式，只适合早期的批处理系统

不适合于交互系统

时间片轮转

常用于分时操作系统，注重响应时间(不关注周转时间)

公平，轮流为各个进程服务

规则：按照进程到达就绪队列的顺序，轮流执行一个时间片。若没有执行完，剥夺处理机，重新放到就绪队列重新排队

只适合于进程调度(作业调度不合适)

抢占式调度，通过时钟中断

优缺点：

优点：公平，响应快，适合分时操作系统

缺点：高频率的切换有一定开销 不区分任务的紧急程度

不会造成饥饿问题

优先调度算法

适用于实时操作系统，按照任务的紧急程度决定处理顺序

规则：每个作业/任务有自己的优先级，调度时选择优先级最高的作业/进程(存在就绪队列)

作用：作业调度 进程调度 IO调度

方式：抢占式 非抢占式

优点：优先级区分紧急程度，重要程度。灵活调整对各种作业的偏好程度

可能造成饥饿问题

非抢占式：主动放弃CPU后，发生调度

抢占式：主动放弃CPU和就绪队列改变后发生调度

就绪队列顺序：先看优先级再看到达顺序

补充：

1. 就绪队列不一定只有一个，可以按照不同的优先级组织就绪队列，也可把优先级更高的放在对头位置

2. 优先级可以分分成

静态优先级：创建后确定，一直不变

动态优先级：创建后初始值，之后情况动态调整

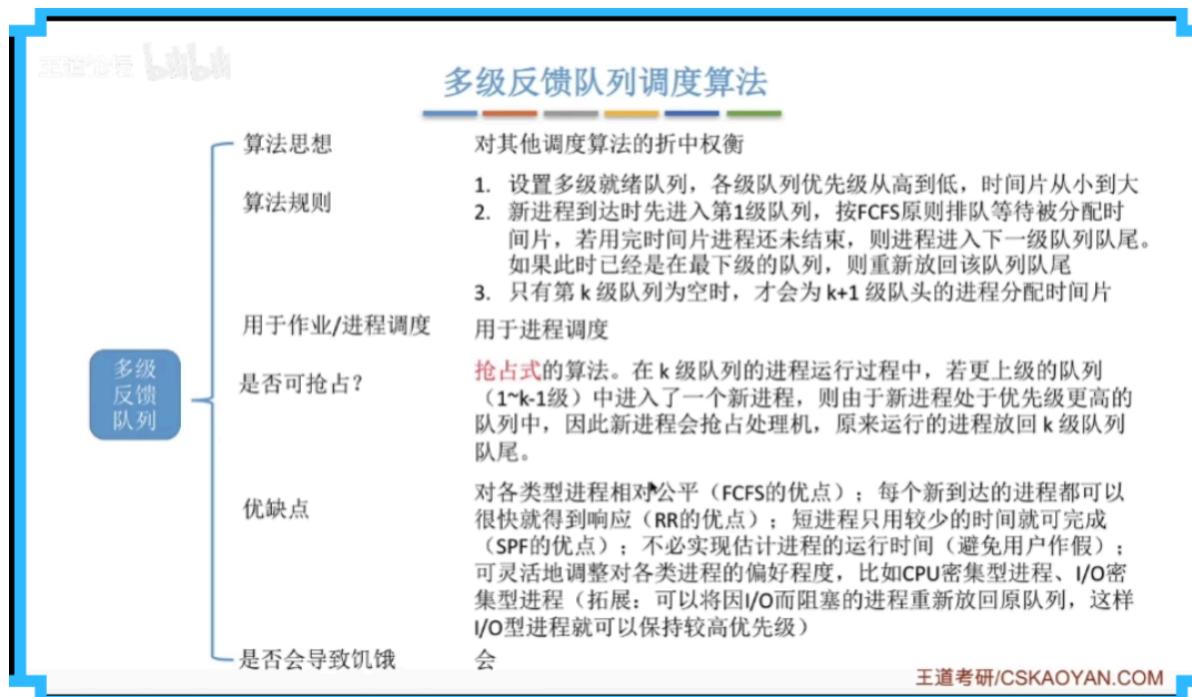
3. 优先级设置

1. 系统进程高于用户进程

2. 前台进程高于后台进程

多级反馈队列调度算法

会导致饥饿问题



综合了先来先服务，短作业优先，时间片轮转和优先调度算法的优点。

五、同步互斥通信

可以把多任务系统当做一个团队，里面的每一个任务就相当于团队里的一个人。
团队成员之间要协调工作进度(同步)、争用会议室(互斥)、沟通(通信)。
多任务系统中所涉及的概念，都可以在现实生活中找到例子。
各类RTOS都会涉及这些概念：任务通知(task notification)、队列(queue)、事件组(event group)、信号量(semaphoe)、互斥量(mutex)等。
我们先站在更高角度来讲解这些概念。

同步与互斥

假设有A、B两人早起抢厕所，A先行一步占用了；B慢了一步，于是就眯一会儿；
当A用完后叫醒B，B也就愉快地上厕所了。

在这个过程中，A、B是互斥地访问“厕所”，“厕所”被称为临界资源。

我们使用了“休眠-唤醒”的同步机制实现了“临界资源”的“互斥访问”。

同一时间只能有一个人使用的资源，被称为临界资源。

比如任务A、B都要使用串口来打印，串口就是临界资源。

如果A、B同时使用串口，那么打印出来的信息就是A、B混杂，无法分辨。

所以使用串口时，应该是这样：A用完，B再用；B用完，A再用。

同步示例：

同步示例 1：

```
100: void Task1Function(void * param)
101: {
102:     volatile int i = 0;
103:     while (1)
104:     {
105:         for (i = 0; i < 10000000; i++)
106:             sum++;
107:             //printf("1");
108:             flagCalcEnd = 1;
109:             vTaskDelete(NULL);
110:     }
111: }
112:
113: void Task2Function(void * param)
114: {
115:     while (1)
116:     {
117:         if (flagCalcEnd)
118:             printf("sum = %d\r\n", sum);
119:     }
120: }
```

》利用 flagCalcEnd 标志位进行同步

缺陷：Task2 会竞争资源(白白浪费 CPU 资源) 效率太低

互斥示例：

利用互斥：

```
23: void TaskGenericFunction(void * param)
24: {
25:     while (1)
26:     {
27:         if (!flagUARTused)
28:         {
29:             flagUARTused = 1; ←
30:             printf("%s\r\n", (char *)param);
31:             flagUARTused = 0; I
32:             vTaskDelay(1);
33:         }
34:     }
35: }
```

缺陷：

tick中断启动 Task3 执行但 flag未置1

task4 同理，又会打扰到 task3 (操作未完成)

程序往往出现问题

FreeRTOS中的线程通信：可以通过全局变量通信。

方法对比

能实现同步、互斥的内核方法有：

任务通知(task notification)、队列(queue)、事件组(event group)、信号量(semaphoe)、互斥量(mutex)。

它们都有类似的操作方法：获取/释放、阻塞/唤醒、超时。

比如：

- A获取资源，用完后A释放资源
- A获取不到资源则阻塞，B释放资源并把A唤醒

- A获取不到资源则阻塞，并定个闹钟；A要么超时返回，要么在这段时间内因为B释放资源而被唤醒。

内核对象	生产者	消费者	数据/状态	说明
队列	ALL	ALL	数据：若干个数据 谁都可以往队列里扔数据，谁都可以从队列里读数据	用来传递数据，发送者、接收者无限制，一个数据只能唤醒一个接收者
事件组	ALL	ALL	多个位：或、与 谁都可以设置(生产)多个位，谁都可以等待某个位、若干个位	用来传递事件，可以是N个事件，发送者、接受者无限制，可以唤醒多个接收者：像广播
信号量	ALL	ALL	数量：0~n 谁都可以增加一个数量，谁都可消耗一个数量	用来维持资源的个数，生产者、消费者无限制，1个资源只能唤醒1个接收者
任务通知	ALL	只有我	数据、状态都可以传输，使用任务通知时，必须指定接受者	N对1的关系：发送者无限制，接收者只能是这个任务
互斥量	只能A开锁	A上锁	位：0、1 我上锁：1变为0，只能由我开锁：0变为1	就像一个空厕所，谁使用谁上锁，也只能由他开锁

队列：

里面可以放任意数据，可以放多个数据

任务、ISR都可以放入数据；任务、ISR都可以从中读出数据

事件组：

一个事件用一bit表示，1表示事件发生了，0表示事件没发生

可以用来表示事件、事件的组合发生了，不能传递数据

有广播效果：事件或事件的组合发生了，等待它的多个任务都会被唤醒

信号量：

核心是"计数值"

任务、ISR释放信号量时让计数值加1

任务、ISR获得信号量时，让计数值减1

任务通知：

核心是任务的TCB里的数值

会被覆盖

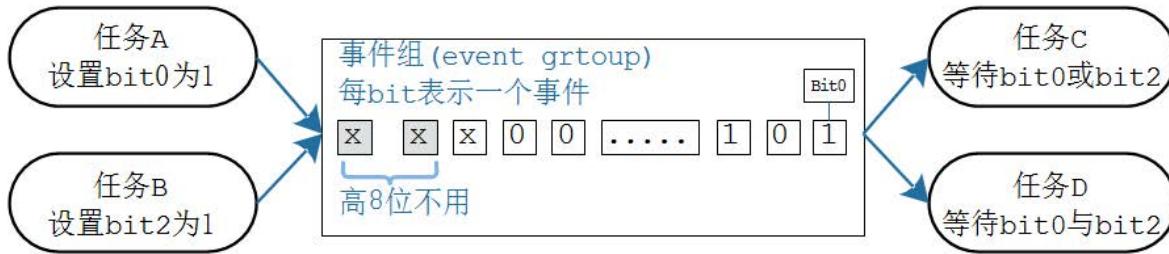
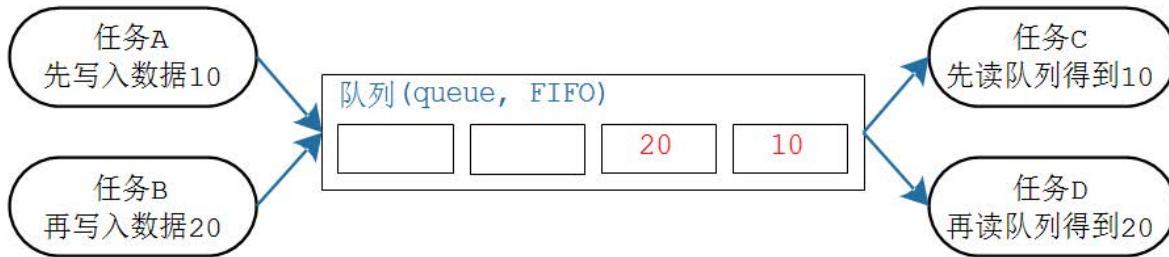
发通知给谁？必须指定接收任务

只能由接收任务本身获取该通知

互斥量：

数值只有0或1

谁获得互斥量，就必须由谁释放同一个互斥量



六、队列

队列(queue)可以用于"任务到任务"、"任务到中断"、"中断到任务"直接传输信息。

本章涉及如下内容：

- 怎么创建、清除、删除队列
- 队列中消息如何保存
- 怎么向队列发送数据、怎么从队列读取数据、怎么覆盖队列的数据
- 在队列上阻塞是什么意思
- 怎么在多个队列上阻塞
- 读写队列时如何影响任务的优先级

队列特性

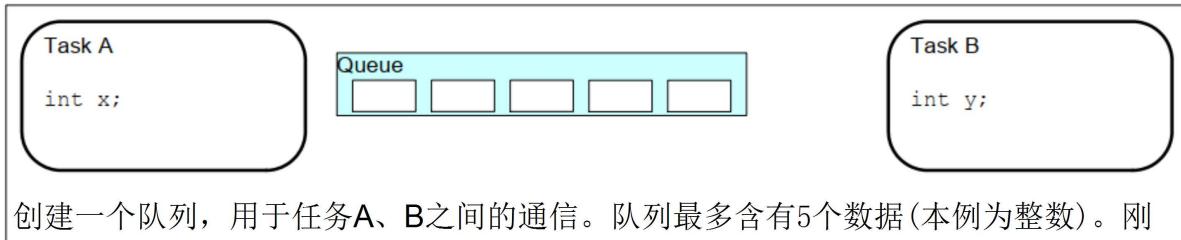
基本操作

队列的简化操如入下图所示，从此图可知：

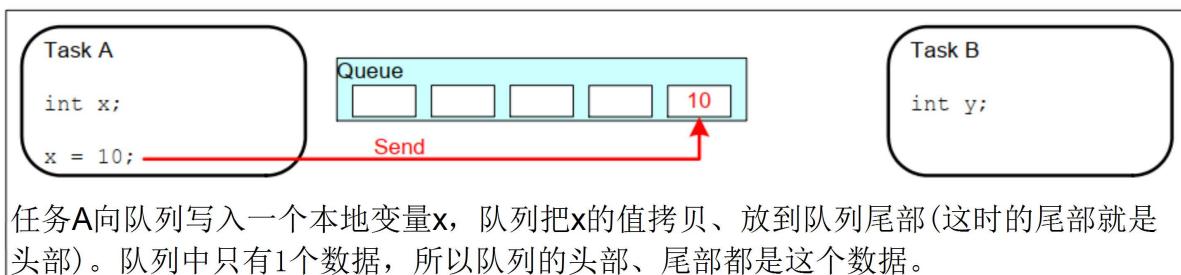
1. 队列可以包含若干个数据：队列中有若干项，这被称为"长度"(length)
2. 每个数据大小固定
3. 创建队列时就要指定长度、数据大小
4. 数据的操作采用先进先出的方法(FIFO， First In First Out)：写数据时放到尾部，读数据时从头部读
5. 也可以强制写队列头部：覆盖头部数据



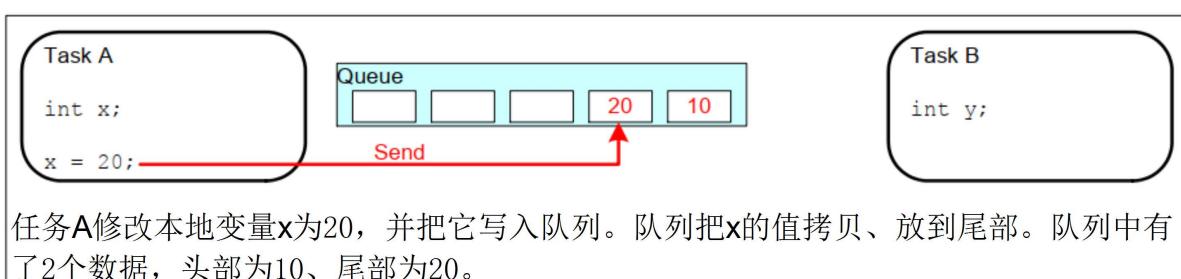
详细操作如下图：



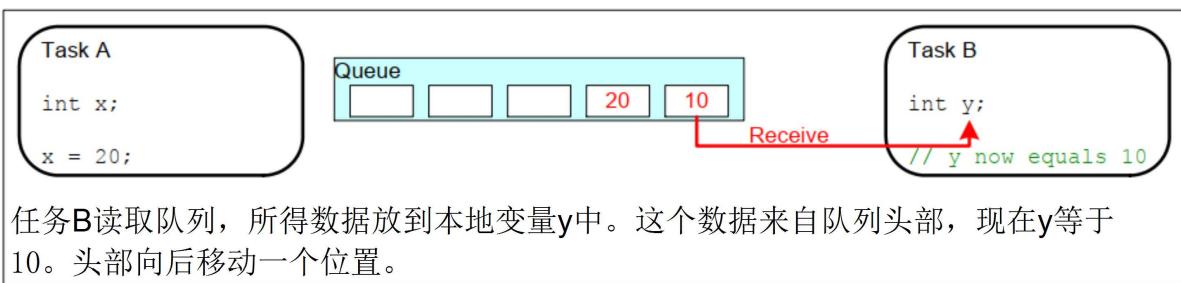
创建一个队列，用于任务A、B之间的通信。队列最多含有5个数据(本例为整数)。刚创建时队列为空。



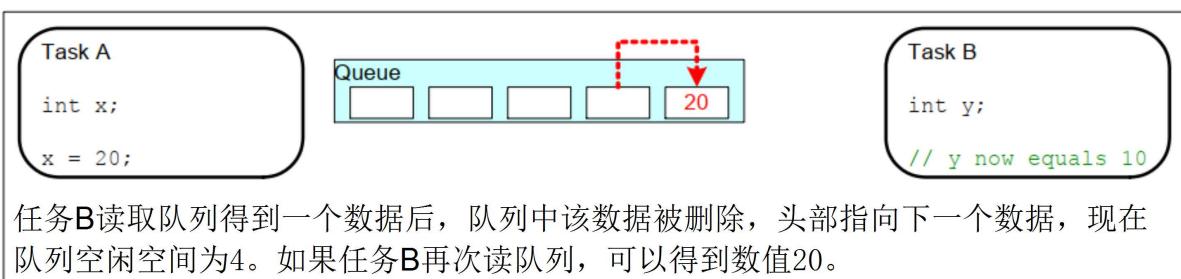
任务A向队列写入一个本地变量x，队列把x的值拷贝、放到队列尾部(这时的尾部就是头部)。队列中只有1个数据，所以队列的头部、尾部都是这个数据。



任务A修改本地变量x为20，并把它写入队列。队列把x的值拷贝、放到尾部。队列中有2个数据，头部为10、尾部为20。



任务B读取队列，所得数据放到本地变量y中。这个数据来自队列头部，现在y等于10。头部向后移动一个位置。



任务B读取队列得到一个数据后，队列中该数据被删除，头部指向下一个数据，现在队列空闲空间为4。如果任务B再次读队列，可以得到数值20。

传输数据两种方式

使用队列传输数据时有两种方法：

- 拷贝：把数据、把变量的值复制进队列里
- 引用：把数据、把变量的地址复制进队列里

FreeRTOS拷贝值的方式：

- 局部变量的值可以发送到队列中，后续即使函数退出、局部变量被回收，也不会影响队列中的数据(复制)
- 无需分配buffer来保存数据，队列中有buffer

- 局部变量可以马上再次使用
- 发送任务、接收任务解耦：接收任务不需要知道这数据是谁的、也不需要发送任务来释放数据
- 如果数据实在太大，你还是可以使用队列传输它的地址(**传输结构体数组等**)
- 队列的空间有FreeRTOS内核分配，无需任务操心
- 对于有内存保护功能的系统，如果队列使用引用方法，也就是使用地址，必须确保双方任务对这个地址都有访问权限。使用拷贝方法时，则无此限制：内核有足够的权限，把数据复制进队列、再把数据复制出队列。

队列的阻塞访问

只要知道队列的句柄，谁都可以读、写该队列。

任务、ISR都可读、写队列。可以多个任务读写队列。

任务读写队列时，简单地说：如果读写不成功，则阻塞；可以指定超时时间。

口语化地说，就是可以定个闹钟：如果能读写了就马上进入就绪态，否则就阻塞直到超时。

比如：某个任务读队列时，如果队列没有数据，则该任务可以进入阻塞状态：还可以指定阻塞的时间。

如果队列有数据了，则该阻塞的任务会变为就绪态。

如果一直都没有数据，则时间到之后它也会进入就绪态。

既然读取队列的任务个数没有限制，那么当多个任务读取空队列时，这些任务都会进入阻塞状态：有多个任务在等待同一个队列的数据。

当队列中有数据时，哪个任务会进入就绪态？

分配的两个原则：

- 优先级最高的任务
- 如果大家的优先级相同，那等待时间最久的任务会进入就绪态

跟读队列类似，一个任务要写队列时，如果队列满了，该任务也可以进入阻塞状态：还可以指定阻塞的时间。

如果队列有空间了，则该阻塞的任务会变为就绪态。

如果一直都没有空间，则时间到之后它也会进入就绪态。

既然写队列的任务个数没有限制，那么当多个任务写“满队列”时，这些任务都会进入阻塞状态：有多个任务在等待同一个队列的空间。

当队列中有空间时，哪个任务会进入就绪态？

- 优先级最高的任务
- 如果大家的优先级相同，那等待时间最久的任务会进入就绪态

队列函数

使用队列的流程：创建队列、写队列、读队列、删除队列。

创建队列

队列的创建有两种方法：动态分配内存、静态分配内存，

动态分配内存：xQueueCreate，队列的内存在函数内部动态分配

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

```

148 #if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
149     #define xQueueCreate( uxQueueLength, uxItemSize )      xQueueGenericCreate( ( uxQueueLength ), ( uxItemSize ), ( queueQUEUE_TYPE_BASE ) )
150 #endif
151

```

参数	说明
uxQueueLength	队列长度, 最多能存放多少个数据(item)
uxItemSize	每个数据(item)的大小: 以字节为单位
返回值	非0: 成功, 返回句柄, 以后使用句柄来操作队列 NULL: 失败, 因为内存不足

静态分配内存: xQueueCreateStatic, 队列的内存要事先分配好

```

1 QueueHandle_t xQueueCreateStatic( UBaseType_t uxQueueLength,
2     UBaseType_t uxItemSize,
3     uint8_t *pucQueueStorageBuffer,
4     StaticQueue_t *pxQueueBuffer
5 );

```

```

● ● ●
1 #if ( configSUPPORT_STATIC_ALLOCATION == 1 )
2     #define xQueueCreateStatic( uxQueueLength, uxItemSize, pucQueueStorage, pxQueueBuffer )  xQueueGenericCreateStatic( ( uxQueueLength ), ( uxItemSize ), ( pucQueueStorage ), ( pxQueueBuffer ), ( queueQUEUE_TYPE_BASE ) )
3 #endif /* configSUPPORT_STATIC_ALLOCATION */

```

参数	说明
uxQueueLength	队列长度, 最多能存放多少个数据(item)
uxItemSize	每个数据(item)的大小: 以字节为单位
pucQueueStorageBuffer	如果uxItemSize非0, pucQueueStorageBuffer必须指向一个uint8_t数组, 此数组大小至少为"uxQueueLength * uxItemSize"
pxQueueBuffer	必须执行一个StaticQueue_t结构体, 用来保存队列的数据结构
返回值	非0: 成功, 返回句柄, 以后使用句柄来操作队列 NULL: 失败, 因为pxQueueBuffer为NULL

```

// 示例代码
#define QUEUE_LENGTH 10
#define ITEM_SIZE sizeof( uint32_t )

// xQueueBuffer用来保存队列结构体
StaticQueue_t xQueueBuffer;

// ucQueueStorage 用来保存队列的数据
// 大小为: 队列长度 * 数据大小
uint8_t ucQueueStorage[ QUEUE_LENGTH * ITEM_SIZE ];

void vATask( void *pvParameters )
{
    QueueHandle_t xQueue1;

    // 创建队列: 可以容纳QUEUE_LENGTH个数据, 每个数据大小是ITEM_SIZE
    xQueue1 = xQueueCreateStatic( QUEUE_LENGTH,
                                 ITEM_SIZE,
                                 ucQueueStorage,
                                 &xQueueBuffer );
}

```

队列复位

队列刚被创建时，里面没有数据；使用过程中可以调用xQueueReset() 把队列恢复为初始状态，此函数原型为：

```

1 /* pxQueue : 复位哪个队列;
2  * 返回值: pdPASS(必定成功)
3  */
4 BaseType_t xQueueReset( QueueHandle_t pxQueue);

```

队列删除

删除队列的函数为vQueueDelete()，只能删除使用动态方法创建的队列，它会释放内存。原型如下：

```

1 void vQueueDelete( QueueHandle_t xQueue );

```

写队列

可以把数据写到队列头部，也可以写到尾部。

这些函数有两个版本：在任务中使用、在ISR中使用。

函数原型如下：

```

1 /* 等同于xQueueSendToBack
2  * 往队列尾部写入数据, 如果没有空间, 阻塞时间为xTicksToWait
3  */
4 BaseType_t xQueueSend(QueueHandle_t xQueue,
5  const void *pvItemToQueue,
6  TickType_t xTicksToWait
7 );
8 /*
9  * 往队列尾部写入数据, 如果没有空间, 阻塞时间为xTicksToWait
10 */
11

```

```

12 BaseType_t xQueueSendToBack(QueueHandle_t xQueue,
13 const void *pvItemToQueue,
14 TickType_t xTicksToWait
15 );
16
17 /*
18 * 往队列尾部写入数据，此函数可以在中断函数中使用，不可阻塞
19 */
20 BaseType_t xQueueSendToBackFromISR(QueueHandle_t xQueue,
21 const void *pvItemToQueue,
22 BaseType_t *pxHigherPriorityTaskWoken
23 );
24 /*
25 * 往队列头部写入数据，如果没有空间，阻塞时间为xTicksToWait
26 */
27 BaseType_t xQueueSendToFront(QueueHandle_t xQueue,
28 const void *pvItemToQueue,
29 TickType_t xTicksToWait
30 );
31 /*
32 * 往队列头部写入数据，此函数可以在中断函数中使用，不可阻塞
33 */
34 BaseType_t xQueueSendToFrontFromISR(QueueHandle_t xQueue,
35 const void *pvItemToQueue,
36 BaseType_t *pxHigherPriorityTaskWoken
37 );

```

参数	说明
xQueue	队列句柄，要写哪个队列
pvItemToQueue	数据指针，这个数据的值会被复制进队列，复制多大的数据？在创建队列时已经指定了数据大小
xTicksToWait	如果队列满则无法写入新数据，可以让任务进入阻塞状态，
xTicksToWait	表示阻塞的最大时间(Tick Count)。如果被设为0，无法写入数据时函数会立刻返回；如果被设为portMAX_DELAY，则会一直阻塞直到有空间可写
返回值	pdPASS：数据成功写入了队列 errQUEUE_FULL：写入失败，因为队列满了。

读队列

使用xQueueReceive() 函数读队列，读到一个数据后，队列中该数据会被移除。

这个函数有两个版本：在任务中使用、在ISR中使用。

函数原型如下：

```

1 BaseType_t xQueueReceive( QueueHandle_t xQueue,
2   void * const pvBuffer,
3   TickType_t xTicksToWait );
4
5 BaseType_t xQueueReceiveFromISR(QueueHandle_t xQueue,
6   void *pvBuffer,
7   BaseType_t *pxTaskWoken
8 );

```

参数	说明
xQueue	队列句柄，要读哪个队列
pvBuffer	bufer指针，队列的数据会被复制到这个buffer 复制多大的数据？在创建队列时已经指定了数据大小
xTicksToWait	队列空则无法读出数据，可以让任务进入阻塞状态，xTicksToWait表示阻塞的最大时间(Tick Count)。如果被设为0，无法读出数据时函数会立刻返回；如果被设为portMAX_DELAY，则会一直阻塞直到有数据可写
返回值	pdPASS：从队列读出数据入 errQUEUE_EMPTY：读取失败，因为队列空了。

查询

可以查询队列中有多少个数据、有多少空余空间。

函数原型如下：

```

1 /*
2  * 返回队列中可用数据的个数
3  */
4 UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );
5 /*
6  * 返回队列中可用空间的个数
7  */
8 UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );

```

覆盖/偷看

当队列长度为1时，可以使用xQueueOverwrite() 或xQueueOverwriteFromISR() 来覆盖数据。

注意，队列长度必须为1。如果队列长度大于1，xQueueOverwrite() 将覆盖队列的最后一个值。

当队列满时，这些函数会覆盖里面的数据，这也意味着这些函数不会被阻塞。

```
1 /* 覆盖队列
2 * xQueue: 写哪个队列
3 * pvItemToQueue: 数据地址
4 * 返回值: pdTRUE表示成功, pdFALSE表示失败
5 */
6 BaseType_t xQueueOverwrite(QueueHandle_t xQueue,
7 const void * pvItemToQueue
8 );
9
10 BaseType_t xQueueOverwriteFromISR(QueueHandle_t xQueue,
11 const void * pvItemToQueue,
12 BaseType_t *pxHigherPriorityTaskWoken
13 );
```

如果想让队列中的数据供多方读取，也就是说读取时不要移除数据，要留给后来人。那么可以使用"窥视"，也就是xQueuePeek() 或xQueuePeekFromISR()。
这些函数会从队列中复制出数据，但是不移除数据。
这也意味着，如果队列中没有数据，那么"偷看"时会导致阻塞；
一旦队列中有数据，以后每次"偷看"都会成功。

```
1 /* 偷看队列
2 * xQueue: 偷看哪个队列
3 * pvItemToQueue: 数据地址，用来保存复制出来的数据
4 * xTicksToWait: 没有数据的话阻塞一会
5 * 返回值: pdTRUE表示成功, pdFALSE表示失败
6 */
7 BaseType_t xQueuePeek(QueueHandle_t xQueue,
8 void * const pvBuffer,
9 TickType_t xTicksToWait
10 );
11
12 BaseType_t xQueuePeekFromISR(QueueHandle_t xQueue,
13 void *pvBuffer,
14 );
```

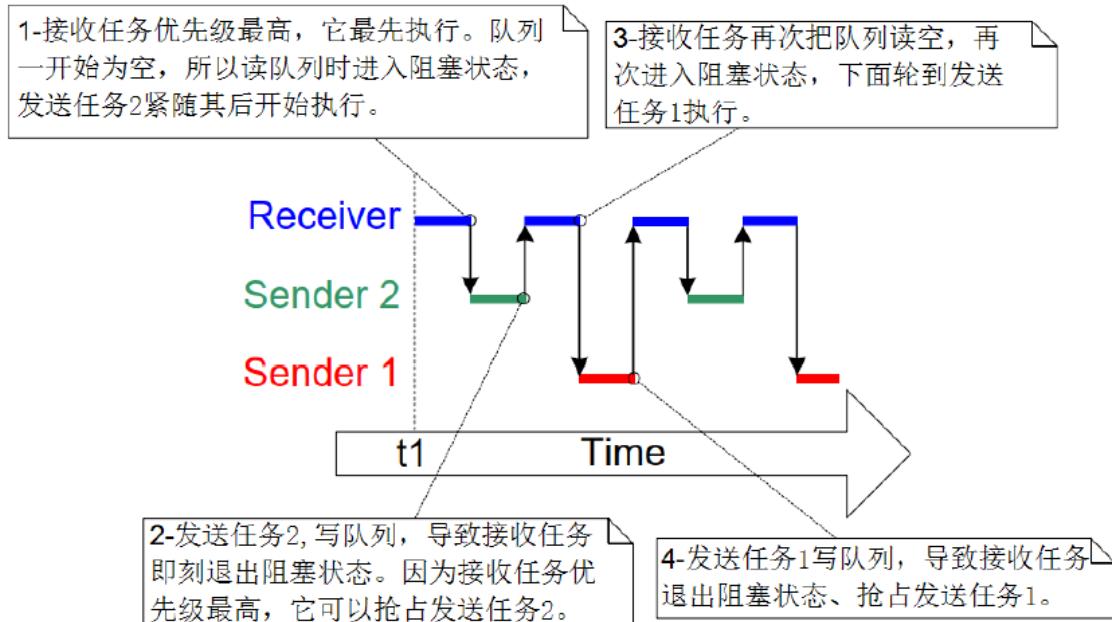
队列基本使用

本程序会创建一个队列，然后创建2个发送任务、1个接收任务：

发送任务优先级为1，分别往队列中写入100、200

接收任务优先级为2，读队列、打印数值

任务调度情况如下图所示：



```
1  /* 队列句柄，创建队列时会设置这个变量 */
2  QueueHandle_t xQueue;
3
4  int main( void )
5  {
6      prvSetupHardware();
7      /* 创建队列：长度为5，数据大小为4字节(存放一个整数) */
8      xQueue = xQueueCreate( 5, sizeof( int32_t ) );
9      if( xQueue != NULL )
10     {
11         /* 创建2个任务用于写队列，传入的参数分别是100、200
12          * 任务函数会连续执行，向队列发送数值100、200
13          * 优先级为1
14          */
15         xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL
16     );
17         xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL
18     );
19         /* 创建1个任务用于读队列
20          * 优先级为2，高于上面的两个任务
21          * 这意味着队列一有数据就会被读走
22          */
23         xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );
24     /* 启动调度器 */
25     vTaskStartScheduler();
26 }
27 }
```

```
28     /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
29     return 0;
30 }
```

发送任务

```
1 static void vsenderTask( void *pvParameters )
2 {
3     int32_t lValueToSend;
4     BaseType_t xStatus;
5     /* 我们会使用这个函数创建2个任务
6      * 这些任务的pvParameters不一样
7      */
8     lValueToSend = ( int32_t ) pvParameters;
9     /* 无限循环 */
10    for( ;; )
11    {
12        /* 写队列
13         * xQueue: 写哪个队列
14         * &lValueToSend: 写什么数据？传入数据的地址，会从这个地址把数据复制进队列
15         * 0: 不阻塞，如果队列满的话，写入失败，立刻返回
16         */
17        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
18        if( xStatus != pdPASS )
19        {
20            printf( "Could not send to the queue.\r\n" );
21        }
22    }
23 }
```

接收任务

```
1 static void vReceiverTask( void *pvParameters )
2 {
3     /* 读取队列时，用这个变量来存放数据 */
4     int32_t lReceivedValue;
5     BaseType_t xStatus;
6     const TickType_t xTicksToWait = pdMS_TO_TICKS( 100UL );
7     /* 无限循环 */
8     for( ;; )
9     {
10        /* 读队列
11         * xQueue: 读哪个队列
12         * &lReceivedValue: 读到的数据复制到这个地址
13         * xTicksToWait: 如果队列为空，阻塞一会
14         */
15        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
16        if( xStatus == pdPASS )
17        {
18            /* 读到了数据 */
19            printf( "Received = %d\r\n", lReceivedValue );
20        }
21        else
22        {
```

```

23     /* 没读到数据 */
24     printf( "Could not receive from the queue.\r\n" );
25 }
26 }
27 }
```

结果如下:

UART #1

```

Received = 100
Received = 200
```

分辨数据源

当有多个发送任务，通过同一个队列发出数据，接收任务如何分辨数据来源？

数据本身带有"来源"信息，比如写入队列的数据是一个结构体，结构体中的IDataSourceID用来表示数据来源：

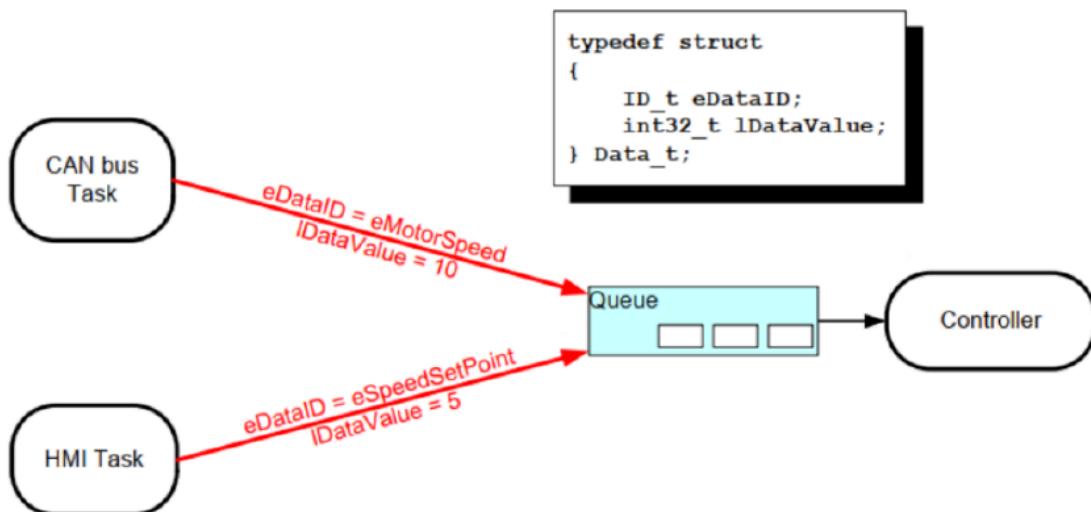
```

1 typedef struct {
2     ID_t eDataID;
3     int32_t lDataValue;
4 } Data_t;
```

不同的发送任务，先构造好结构体，填入自己的eDataID，再写队列；

接收任务读出数据后，根据eDataID 就可以知道数据来源了，如下图所示：

- CAN任务发送的数据: eDataID=eMotorSpeed
- HMI任务发送的数据: eDataID=eSpeedSetPoint



示例代码：

```

1  /* 定义2种数据来源(ID) */
2  typedef enum
3  {
4      eMotorSpeed,
5      eSpeedSetPoint
6 } ID_t;
7
8  /* 定义在队列中传输的数据的格式 */
9  typedef struct {
10     ID_t eDataID;
11     int32_t lDataValue;
12 } Data_t;
13
14 /* 定义2个结构体 */
15 static const Data_t xStructsToSend[ 2 ] =
16 {
17     { eMotorSpeed, 10 }, /* CAN任务发送的数据 */
18     { eSpeedSetPoint, 5 } /* HMI任务发送的数据 */
19 };
20
21 /* vSenderTask被用来创建2个任务，用于写队列
22 * vReceiverTask被用来创建1个任务，用于读队列
23 */
24 static void vSenderTask( void *pvParameters );
25 static void vReceiverTask( void *pvParameters );
26
27 /* 队列句柄， 创建队列时会设置这个变量 */
28 QueueHandle_t xQueue;
29
30 int main( void )
31 {
32     prvSetupHardware();
33     /* 创建队列： 长度为5， 数据大小为4字节(存放一个整数) */
34     xQueue = xQueueCreate( 5, sizeof( Data_t ) );
35     if( xQueue != NULL )
36     {
37         /* 创建2个任务用于写队列， 传入的参数是不同的结构体地址
38          * 任务函数会连续执行， 向队列发送结构体
39          * 优先级为2
40          */
41         xTaskCreate(vSenderTask, "CAN Task", 1000, (void *) &
(xStructsToSend[0]), 2, NULL);
42         xTaskCreate(vSenderTask, "HMI Task", 1000, (void *) &
(xStructsToSend[1]), 2, NULL);
43         /* 创建1个任务用于读队列
44          * 优先级为1， 低于上面的两个任务
45          * 这意味着发送任务优先写队列， 队列常常是满的状态
46          */
47         xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );
48         /* 启动调度器 */
49         vTaskStartScheduler();
50     }
51     else
52     {
53         /* 无法创建队列 */

```

```

54    }
55    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
56    return 0;
57 }

```

发送

```

1 static void vSenderTask( void *pvParameters )
2 {
3     BaseType_t xStatus;
4     const TickType_t xTicksToWait = pdMS_TO_TICKS( 100UL );
5     /* 无限循环 */
6     for( ;; )
7     {
8         /* 写队列
9         * xQueue: 写哪个队列
10        * pvParameters: 写什么数据？传入数据的地址，会从这个地址把数据复制进队列
11        * xTicksToWait: 如果队列满的话，阻塞一会
12        */
13        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait ); //结构
14        //体的地址传输
15        if( xStatus != pdPASS )
16        {
17            printf( "Could not send to the queue.\r\n" );
18        }
19    }

```

接收任务的函数：

```

1 static void vReceiverTask( void *pvParameters )
2 {
3     /* 读取队列时，用这个变量来存放数据 */
4     Data_t xReceivedStructure;
5     BaseType_t xStatus;
6     /* 无限循环 */
7     for( ;; )
8     {
9         /* 读队列
10        * xQueue: 读哪个队列
11        * &xReceivedStructure: 读到的数据复制到这个地址
12        * 0: 没有数据就立刻返回，不阻塞
13        */
14        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );
15        if( xStatus == pdPASS )
16        {
17            /* 读到了数据 */
18            if( xReceivedStructure.eDataID == eMotorSpeed )
19            {
20                printf( "From CAN, MotorSpeed =
21 %d\r\n", xReceivedStructure.lDataValue );
22            }
23            else if( xReceivedStructure.eDataID == eSpeedSetPoint )
24            {

```

```

24     printf( "From HMI, SpeedSetPoint =
25         %d\r\n", xReceivedStructure.lDataValue );
26     }
27     else
28     {
29         /* 没读到数据 */
30         printf( "Could not receive from the queue.\r\n" );
31     }
32 }
33 }
```

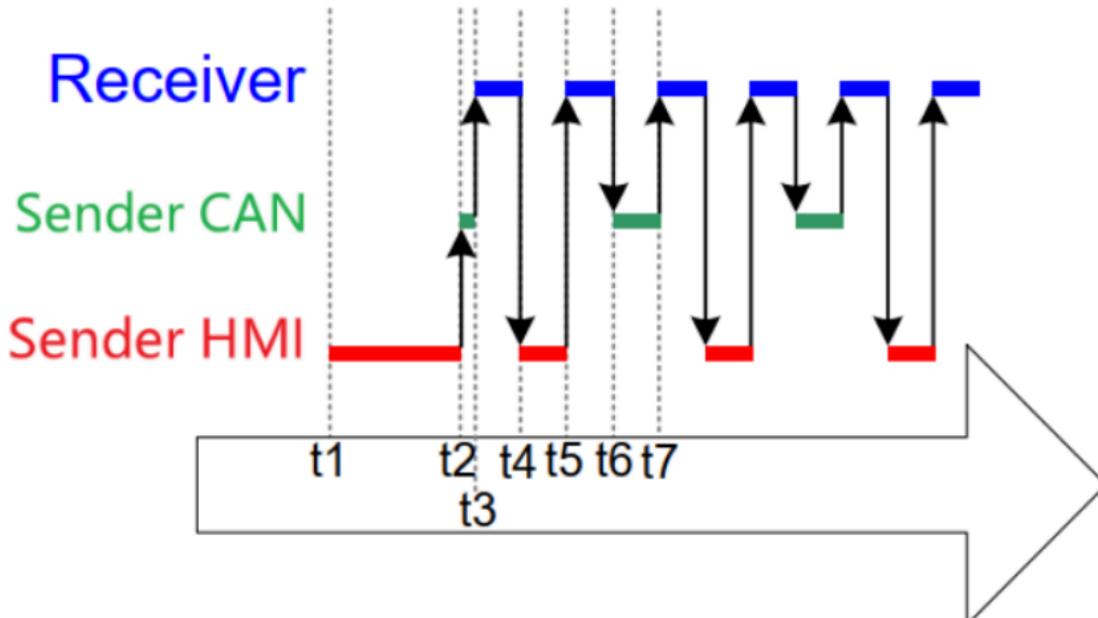
运行结果如下：

UART #1

```

From HMI, SpeedSetPoint = 5
From CAN, MotorSpeed = 10
From HMI, SpeedSetPoint = 5
From CAN, MotorSpeed = 10
From HMI, SpeedSetPoint = 5
From CAN, MotorSpeed = 10
From HMI, SpeedSetPoint = 5
From CAN, MotorSpeed = 10
From HMI, SpeedSetPoint = 5
From CAN, MotorSpeed = 10
From HMI, SpeedSetPoint = 5
```

结果分析：



- t1: HMI是最后创建的最高优先级任务，它先执行，一下子向队列写入5个数据，把队列都写满了
- t2: 队列已经满了，HMI任务再发起第6次写操作时，进入阻塞状态。这时CAN任务是最高优先级的就绪态任务，它开始执行
- t3: CAN任务发现队列已经满了，进入阻塞状态；接收任务变为最高优先级的就绪态任务，它开始运行
- t4: 现在，HMI任务、CAN任务的优先级都比接收任务高，它们都在等待队列有空闲的空间；一旦接收任务读出1个数据，会马上被抢占。被谁抢占？谁等待最久？HMI任务！所以在t4时刻，切换到HMI任

务。

t5: HMI任务向队列写入第6个数据，然后再次阻塞，这是CAN任务已经阻塞很久了。接收任务变为最高优先级的就绪态任务，开始执行。

t6: 现在，HMI任务、CAN任务的优先级都比接收任务高，它们都在等待队列有空闲的空间；一旦接收任务读出1个数据，会马上被抢占。被谁抢占？谁等待最久？CAN任务！所以在t6时刻，切换到CAN任务。

t7: CAN任务向队列写入数据，因为仅仅有一个空间供写入，所以它马上再次进入阻塞状态。这时HMI任务、CAN任务都在等待空闲空间，只有接收任务可以继续执行。

大数据块传输

FreeRTOS的队列使用拷贝传输，也就是要传输uint32_t时，把4字节的数据拷贝进队列；要传输一个8字节的结构体时，把8字节的数据拷贝进队列。

如果要传输1000字节的结构体呢？写队列时拷贝1000字节，读队列时再拷贝1000字节？不建议这么做，影响效率！

这时候，我们要传输的是这个巨大结构体的地址：把它的地址写入队列，对方从队列得到这个地址，使用地址去访问那1000字节的数据。

使用地址来间接传输数据时，这些数据放在RAM里，对于这块RAM，要保证这几点：

1. RAM的所有者、操作者，必须清晰明了

这块内存，就被称为“共享内存”。要确保不能同时修改RAM。比如，在写队列之前只有由发送者修改这块RAM，在读队列之后只能由接收者访问这块RAM。

2. RAM要保持可用

这块RAM应该是全局变量，或者是动态分配的内存。对于动态分配的内存，要确保它不能提前释放：要等到接收者用完后再释放。

另外，不能是局部变量。

代码示例：

```
1  /* 定义一个字符数组 */
2  static char pcBuffer[100];
3  /* vSenderTask被用来创建2个任务，用于写队列
4  * vReceiverTask被用来创建1个任务，用于读队列
5  */
6  static void vSenderTask( void *pvParameters );
7  static void vReceiverTask( void *pvParameters );
8
9  /* 队列句柄，创建队列时会设置这个变量 */
10 QueueHandle_t xQueue;
11
12 int main( void )
13 {
14     prvSetupHardware();
15     /* 创建队列：长度为1，数据大小为4字节(存放一个char指针) */
16     xQueue = xQueueCreate( 1, sizeof(char *) );
17     if( xQueue != NULL )
18     {
19         /* 创建1个任务用于写队列
20         * 任务函数会连续执行，构造buffer数据，把buffer地址写入队列
21         * 优先级为1
22         */
23         xTaskCreate( vSenderTask, "Sender", 1000, NULL, 1, NULL );
24         /* 创建1个任务用于读队列
```

```

25     * 优先级为2，高于上面的两个任务
26     * 这意味着读队列得到buffer地址后，本任务使用buffer时不会被打断
27     */
28     xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );
29     /* 启动调度器 */
30     vTaskStartScheduler();
31 }
32 else
33 {
34     /* 无法创建队列 */
35 }
36 /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
37 return 0;
38 }
```

发送任务的函数中，现在全局大数组pcBuffer中构造数据，然后把它的地址写入队列，代码如下：

```

1 static void vSenderTask( void *pvParameters )
2 {
3     BaseType_t xStatus;
4     static int cnt = 0;
5     char *buffer;
6     /* 无限循环 */
7     for( ;; )
8     {
9         sprintf(pcBuffer, "www.100ask.net Msg %d\r\n", cnt++);
10        buffer = pcBuffer; // buffer变量等于数组的地址，下面要把这个地址写入队列
11        /* 写队列
12         * xQueue：写哪个队列
13         * pvParameters：写什么数据？传入数据的地址，会从这个地址把数据复制进队列
14         * 0：如果队列满的话，即刻返回
15         */
16        xStatus = xQueueSendToBack( xQueue, &buffer, 0 ); /* 只需要写入4字节，
17         * 无需写入整个buffer */
18        if( xStatus != pdPASS )
19        {
20            printf( "Could not send to the queue.\r\n" );
21        }
22    }
}
```

接收任务的函数中，读取队列、得到buffer的地址、打印，代码如下：

```

1 static void vReceiverTask( void *pvParameters )
2 {
3     /* 读取队列时，用这个变量来存放数据 */
4     char *buffer;
5     const TickType_t xTicksToWait = pdMS_TO_TICKS( 100UL );
6     BaseType_t xStatus;
7     /* 无限循环 */
8     for( ;; )
9     {
10        /* 读队列
11         * xQueue：读哪个队列
```

```

12     * &xReceivedStructure: 读到的数据复制到这个地址
13     * xTicksToWait: 没有数据就阻塞一会儿
14     */
15     xStatus = xQueueReceive( xQueue, &buffer, xTicksToWait); /* 得到
buffer地址, 只是4字节 */
16     if( xStatus == pdPASS )
17     {
18         /* 读到了数据 */
19         printf("Get: %s", buffer);
20     }
21     else
22     {
23         /* 没读到数据 */
24         printf( "Could not receive from the queue.\r\n" );
25     }
26 }
27 }
```

运行结果如下图所示：

```

UART #1
Get: www.100ask.net Msg 0
Get: www.100ask.net Msg 1
Get: www.100ask.net Msg 2
Get: www.100ask.net Msg 3
Get: www.100ask.net Msg 4
Get: www.100ask.net Msg 5
Get: www.100ask.net Msg 6
Get: www.100ask.net Msg 7
Get: www.100ask.net Msg 8
Get: www.100ask.net Msg 9
Get: www.100ask.net Msg 10
Get: www.100ask.net Msg 11
Get: www.100ask.net Msg 12
```

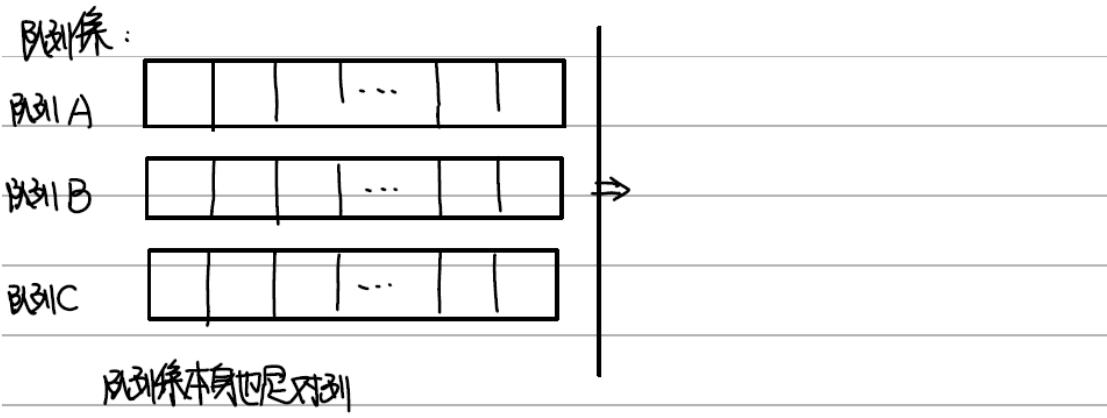
邮箱

FreeRTOS的邮箱概念跟别的RTOS不一样，这里的邮箱称为"橱窗"也许更恰当：

- 它是一个队列，队列长度只有1
- 写邮箱：新数据覆盖旧数据，在任务中使用xQueueOverwrite()，在中断中使用xQueueOverwriteFromISR()。既然是覆盖，那么无论邮箱中是否有数据，这些函数总能成功写入数据。
- 读邮箱：读数据时，数据不会被移除；在任务中使用xQueuePeek()，在中断中使用xQueuePeekFromISR()。

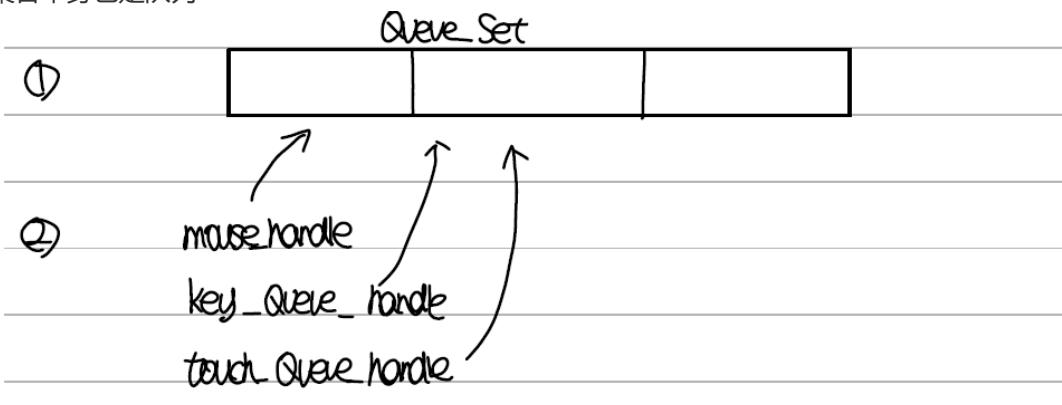
这意味着，第一次调用时会因为无数据而阻塞，一旦曾经写入数据，以后读邮箱时总能成功。

队列集



Queue Set：监测 A.B.C. 长度为 $A.length + B.length + C.length$

队列集合本身也是队列



③ touch 队列写入数据后，同时 touch handle 放入 Queue Set

写队列 1 次，就写队列 1 次，写队列 N 次，就写队列 N 次

反之读队列 1 次，读队列集 1 次

过程：

touch data touch queue $\xrightarrow{\text{handle}}$ Queue Set

④ Read QueueSet 返回 Queue (-1次)

再读 Queue (-1次)

注意点：

反思

假设队列A、B、C使用队列集，有哪些要注意的地方？

(多选题)

✓ A.写队列A N次，会导致写队列集N次，也就是队列集里有N个队列A的handle

✓ B.读一次队列集返回一个队列后，只能读这个队列一次

✓ C.创建队列集时，它要管理队列ABC，那么队列集的长度 = 队列A长度 + 队列B长度 + 队列C长度

相关函数

创建队列集合：xQueueCreateSet()

```
2919 #if ( ( configUSE_QUEUE_SETS == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
2920
2921     QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength )
2922     {
2923         QueueSetHandle_t pxQueue;
2924
2925         pxQueue = xQueueGenericCreate( uxEventQueueLength, ( UBaseType_t ) sizeof( Queue_t * ), queueQUEUE_TYPE_SET );
2926
2927         return pxQueue;
2928     }
2929
2930 #endif /* configUSE_QUEUE_SETS */
2931 /*-----*/
```

简历Queue与QueueSet联系：xQueueAddToSet()

```
#if ( configUSE_QUEUE_SETS == 1 )

    BaseType_t xQueueAddToSet( QueueSetMemberHandle_t xQueueOrSemaphore,
                                QueueSetHandle_t xQueueSet )
{
    BaseType_t xReturn;

    taskENTER_CRITICAL();
    {
        if( ( Queue_t * ) xQueueOrSemaphore )->pxQueueSetContainer != NULL )
        {
            /* Cannot add a queue/semaphore to more than one queue set. */
            xReturn = pdFAIL;
        }
        else if( ( Queue_t * ) xQueueOrSemaphore )->uxMessagesWaiting != ( UBaseType_t ) 0 )
        {
            /* Cannot add a queue/semaphore to a queue set if there are already
             * items in the queue/semaphore. */
            xReturn = pdFAIL;
        }
        else
        {
            ( Queue_t * ) xQueueOrSemaphore )->pxQueueSetContainer = xQueueSet;
            xReturn = pdPASS;
        }
    }
    taskEXIT_CRITICAL();

    return xReturn;
}

#endif /* configUSE_QUEUE_SETS */
```

读取队列集: xQueueSelectFromSet()

```
3003
3004 #if ( configUSE_QUEUE_SETS == 1 )
3005
3006     QueueSetMemberHandle_t xQueueSelectFromSet( QueueSetHandle_t xQueueSet,
3007                                                 TickType_t const xTicksToWait )
3008     [
3009         QueueSetMemberHandle_t xReturn = NULL;
3010
3011         ( void ) xQueueReceive( ( QueueHandle_t ) xQueueSet, &xReturn, xTicksToWait ); /*lint le961 Casting from one typedef to another is not redundant*/
3012         return xReturn;
3013     ]
3014
3015 #endif /* configUSE_QUEUE_SETS */
3016 /*-----*/
```

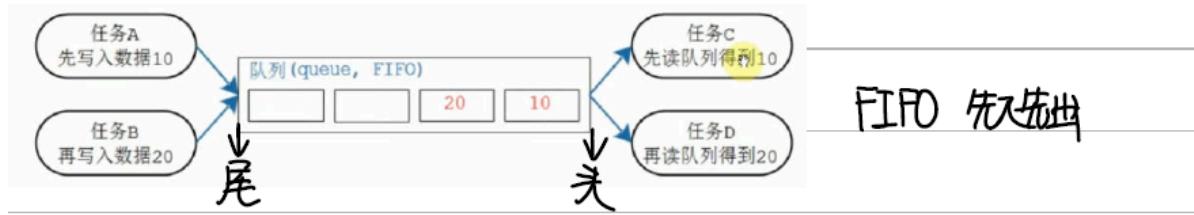
代码示例：

```
1      TaskHandle_t xHandleTask1;
2
3 #ifdef DEBUG
4     debug();
5 #endif
6
7     prvSetupHardware();
8 // 创建两个队列
9     xQueueHandle1 = xQueueCreate(2, sizeof(int)); // main函数中创建队列
10    if (xQueueHandle1 == NULL)
11    {
12        printf("can not create queue\r\n");
13    }
14    xQueueHandle2 = xQueueCreate(2, sizeof(int)); // main函数中创建队列
15    if (xQueueHandle2 == NULL)
16    {
17        printf("can not create queue\r\n");
18    }
19 // 创建队列集合
20    xQueueMySetHandle = xQueueCreateSet(4); // 两个队列的长度和
21 // 队列加入队列集合
22    xQueueAddToSet(xQueueHandle1,xQueueMySetHandle);
23    xQueueAddToSet(xQueueHandle2,xQueueMySetHandle);
24 // 创建三个任务
25    xTaskCreate(Task1Function, "Task1", 100, NULL, 1, &xHandleTask1);
26    xTaskCreate(Task2Function, "Task2", 100, NULL, 1, NULL);
27    xTaskCreate(Task3Function, "Task3", 100, NULL, 1, NULL);
28 /* Start the scheduler. */
29    vTaskStartScheduler();
30
31 /* Will only get here if there was not enough heap space to create the
32 idle task. */
33    return 0;
```

```
1 void Task1Function(void *param)
2 {
3     int i=0;
4     while (1)
5     {
6         xQueueSend(xQueueHandle1,&i,portMAX_DELAY);
7         i++;
8         vTaskDelay(10);
9     }
10 }
11
12 void Task2Function(void *param)
13 {
14     int i=-1;
15     while (1)
16     {
17         xQueueSend(xQueueHandle1,&i,portMAX_DELAY);
18         i--;
19         vTaskDelay(20);
20     }
21 }
22
23 void Task3Function(void *param)
24 {
25     QueueSetMemberHandle_t tempQueueHandle;
26     int i;
27     while (1)
28     {
29         //read Queue Set:which Queue has Data
30         tempQueueHandle = xQueueSelectFromSet(xQueueMySetHandle,portMAX_DELAY);
31         //read Queue : print Data
32         xQueueReceive(tempQueueHandle,&i,0); //Can read Queue must pass read Queue set : no wait
33         printf("get Date: %d\r\n",i);
34     }
35 }
36
```

队列实现同步互斥

队列结构体：



队列结构体：



→ buffer 指针

→ 两个链表(插入数据任务, 取出数据任务)

```
typedef struct QueueDefinition /* The old naming convention is used to prevent breaking kernel aware debuggers. */
{
    int8_t * pcHead;           /*< Points to the beginning of the queue storage area. */
    int8_t * pcWriteTo;        /*< Points to the free next place in the storage area. */

    union
    {
        QueuePointers_t xQueue;      /*< Data required exclusively when this structure is used as a queue. */
        SemaphoreData_t xSemaphore; /*< Data required exclusively when this structure is used as a semaphore. */
    } u;

    List_t xTasksWaitingToSend;   /*< List of tasks that are blocked waiting to post onto this queue. Stored in priority order. */
    List_t xTasksWaitingToReceive; /*< List of tasks that are blocked waiting to read from this queue. Stored in priority order. */

    volatile UBaseType_t uxMessagesWaiting; /*< The number of items currently in the queue. */
    UBaseType_t uxLength;             /*< The length of the queue defined as the number of items it will hold, not the number of bytes. */
    UBaseType_t uxItemSize;          /*< The size of each items that the queue will hold. */

    volatile int8_t cRxLock;         /*< Stores the number of items received from the queue (removed from the queue) while the queue was locked. */
    volatile int8_t cTxLock;         /*< Stores the number of items transmitted to the queue (added to the queue) while the queue was locked. */

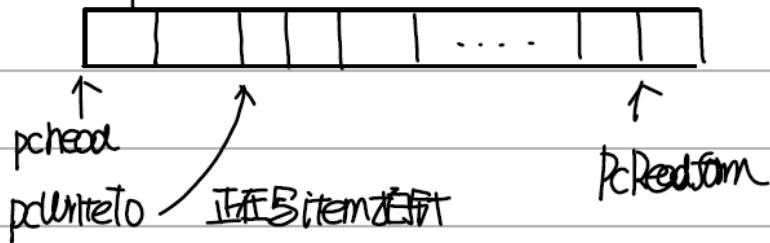
#if ( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
    uint8_t ucStaticallyAllocated; /*< Set to pdTRUE if the memory used by the queue was statically allocated to ensure no attempt is made to free it. */
#endif

#if ( configUSE_QUEUE_SETS == 1 )
    struct QueueDefinition * pxQueueSetContainer;
#endif

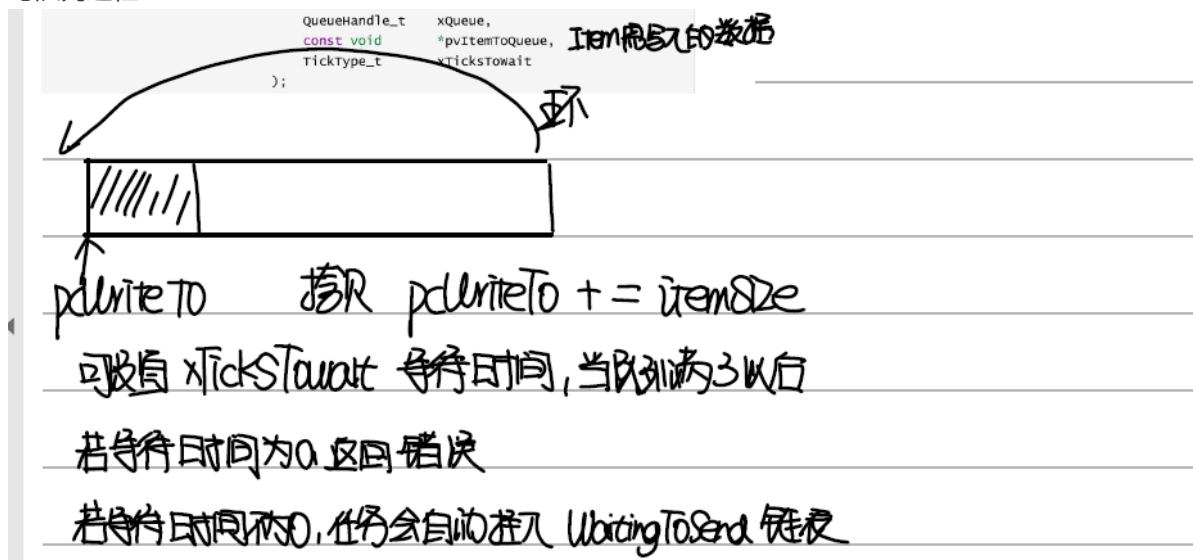
#if ( configUSE_TRACE_FACILITY == 1 )
    UBaseType_t uxQueueNumber;
    uint8_t ucQueueType;
#endif
} xQUEUE;
```

队列的本质是一个环型队列：

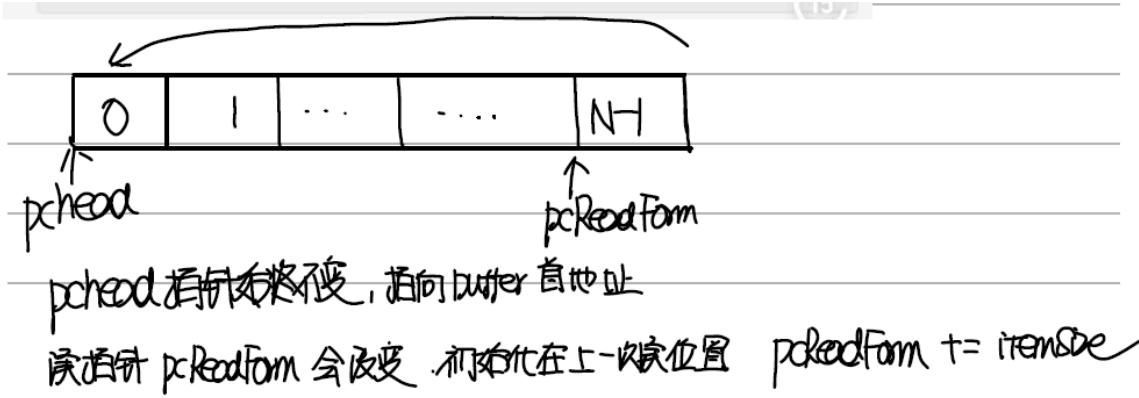
本质为一个环形缓冲区：



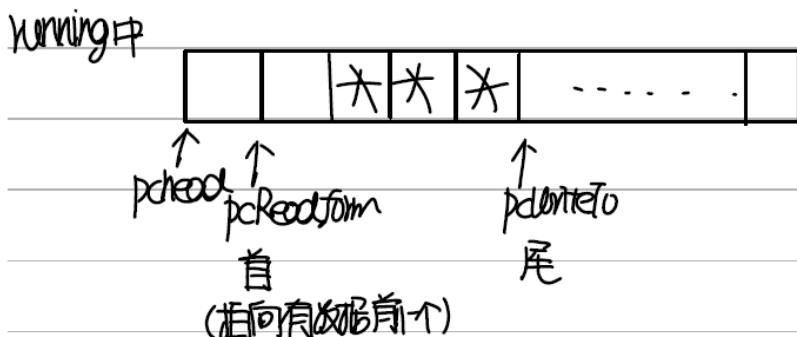
写队列过程：



读队列过程：



运行中指针的实际位置：



每次移动 itemSize

若头部写入：



$\text{pcReadFrom} -= \text{itemSize}$

队列实现同步：

```
1 void Task1Function(void * param)
2 {
3     volatile int i = 0;
4     while (1)
5     {
6         for (i = 0; i < 10000000; i++)
7             sum++;
8         xQueueSend(xQueueCalcHandle, &sum, portMAX_DELAY);
9         sum = 1;
10    }
11 }
12
13 void Task2Function(void * param)
14 {
15     int val;
16
17     while (1)
18     {
19         flagCalcEnd = 0;//模拟器中检测时间
20         xQueueReceive(xQueueCalcHandle, &val, portMAX_DELAY);//没有读到数据会一直阻塞，不占用CPU资源
21         flagCalcEnd = 1;//模拟器中检测时间
22         printf("sum = %d\r\n", val);
23     }
24 }
```

↑ 没有写入队列，任务阻塞
↓ 换任务

在队列中读取到数据后，再进行下一步操作，实现同步操作

队列实现互斥：

创建队列锁 `InitUARTLock()` 写入数据表明已被访问

获得锁 `GetUARTLock()` 队列中读数据，读到即有锁

释放锁： `PutUARTLock()` 队列中写入数据

用队列去实现锁

补充：主动放弃CPU方式

- 1 `vTaskDelay()`阻塞
- 2 `taskYIELD()` 主动发起任务调度

```
1 int InitUARTLock(void)
2 {
3     int val;
4     xQueueUARTcHandle = xQueueCreate(1, sizeof(int));
5     if (xQueueUARTcHandle == NULL)
6     {
7         printf("can not create queue\r\n");
8         return -1;
9     }
10    xQueueSend(xQueueUARTcHandle, &val, portMAX_DELAY);
11    return 0;
12 }
13
14 void GetUARTLock(void)
15 {
16     int val;
17     xQueueReceive(xQueueUARTcHandle, &val, portMAX_DELAY);
18 }
19
20 void PutUARTLock(void)
21 {
22     int val;
23     xQueueSend(xQueueUARTcHandle, &val, portMAX_DELAY);
24 }
```

```
1 void TaskGenericFunction(void * param)
2 {
3     while (1)
4     {
5         GetUARTLock();
6         printf("%s\r\n", (char *)param);
7         // task 3 is waiting
8         PutUARTLock(); /* task 3 ==> ready, task 4 is running */
9         taskYIELD(); //给出Task3的执行空间1主动放弃CPU
10    }
11 }
```

xQueueCreate内存申请在堆区，不会在函数结束后释放。

七、信号量的使用

前面介绍的队列(queue)可以用于传输数据：在任务之间、任务和中断之间。有时候我们只需要传递状态，并不需要传递具体的信息，比如：

- 我的事做完了，通知一下你
- 卖包子了、卖包子了，做好了1个包子！做好了2个包子！做好了3个包子！
- 这个停车位我占了，你们只能等着

在这种情况下我们可以使用信号量(semaphore)，它更节省内存。

本章涉及如下内容：

- 怎么创建、删除信号量
- 怎么发送、获得信号量
- 什么是计数型信号量？什么是二进制信号量？

信号量的特性

常规操作

信号量本质上也是一个队列，拥有队列结构体，但是没有存储数据的buffer

信号量这个名字很恰当：

- 信号：起通知作用
- 量：还可以用来表示资源的数量
 - 当“量”没有限制时，它就是“计数型信号量”(Counting Semaphores)
 - 当“量”只有0、1两个取值时，它就是“二进制信号量”(Binary Semaphores)
- 支持的动作：“give”给出资源，计数值加1；“take”获得资源，计数值减1

计数型信号量的典型场景是：

- 计数：事件产生时“give”信号量，让计数值加1；处理事件时要先“take”信号量，就是获得信号量，让计数值减1。
- 资源管理：要想访问资源需要先“take”信号量，让计数值减1；用完资源后“give”信号量，让计数值加1

信号量的“give”、“take”双方并不需要相同，可以用于生产者-消费者场合：

- 生产者为任务A、B，消费者为任务C、D
- 一开始信号量的计数值为0，如果任务C、D想获得信号量，会有两种结果：
 - 阻塞：买不到东西咱就等等吧，可以定个闹钟(超时时间)
 - 即刻返回失败：不等
- 任务A、B可以生产资源，就是让信号量的计数值增加1，并且把等待这个资源的顾客唤醒
- 唤醒谁？谁优先级高就唤醒谁，如果大家优先级一样就唤醒等待时间最长的人

二进制信号量跟计数型的唯一差别，就是计数值的最大值被限定为1。



信号量与队列

队列	信号量
可以容纳多个数据， 创建队列时有2部分内存: 队列结构体、存储数据的空间	只有计数值，无法容纳其他数据。 创建信号量时，只需要分配信号量结构体
生产者：没有空间存入数据时可以阻塞	生产者：用于不阻塞，计数值已经达到最大时返回失败
消费者：没有数据时可以阻塞	消费者：没有资源时可以阻塞

两种信号量

信号量的计数值都有限制：限定了最大值。

如果最大值被限定为1，那么它就是二进制信号量；如果最大值不是1，它就是计数型信号量。

二进制信号量	计数型信号量
被创建时初始值为0	被创建时初始值可以设定
其他操作是一样的	其他操作是一样的

信号量函数

使用信号量时，先创建、然后去添加资源、获得资源。

使用句柄来表示一个信号量。

创建信号量

使用信号量之前，要先创建，得到一个句柄；

使用信号量时，要使用句柄来表明使用哪个信号量。

	二进制信号量	计数型信号量
动态创建	xSemaphoreCreateBinary 计数值初始值为0	xSemaphoreCreateCounting
	vSemaphoreCreateBinary(过时了) 计数值初始值为1	
静态创建	xSemaphoreCreateBinaryStatic	xSemaphoreCreateCountingStatic

二进制信号量函数原型：

```

1  /* 创建一个二进制信号量，返回它的句柄。
2  * 此函数内部会分配信号量结构体
3  * 返回值：返回句柄，非NULL表示成功
4  */
5  SemaphoreHandle_t xSemaphoreCreateBinary( void );
6
7  /* 创建一个二进制信号量，返回它的句柄。
8  * 此函数无需动态分配内存，所以需要先有一个StaticSemaphore_t结构体，并传入它的指针
9  * 返回值：返回句柄，非NULL表示成功
10 */
11 SemaphoreHandle_t xSemaphoreCreateBinaryStatic( StaticSemaphore_t
*pxSemaphoreBuffer );

```

计数型信号量函数原型：

```

1  /* 创建一个计数型信号量，返回它的句柄。
2  * 此函数内部会分配信号量结构体
3  * uxMaxCount: 最大计数值
4  * uxInitialCount: 初始计数值
5  * 返回值：返回句柄，非NULL表示成功
6  */
7  SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount,
UBaseType_t uxInitialCount );
8
9  /* 创建一个计数型信号量，返回它的句柄。
10 * 此函数无需动态分配内存，所以需要先有一个StaticSemaphore_t结构体，并传入它的指针
11 * uxMaxCount: 最大计数值
12 * uxInitialCount: 初始计数值
13 * pxSemaphoreBuffer: StaticSemaphore_t结构体指针
14 * 返回值：返回句柄，非NULL表示成功
15 */
16 SemaphoreHandle_t xSemaphoreCreateCountingStatic( UBaseType_t uxMaxCount,
UBaseType_t uxInitialCount,
StaticSemaphore_t *pxSemaphoreBuffer );

```

结构体为：

```

1320 typedef struct xSTATIC_QUEUE
1321 {
1322     void * pvDummy1[ 3 ];
1323
1324     union
1325     {
1326         void * pvDummy2;
1327         UBaseType_t uxDummy2;
1328     } u;
1329
1330     StaticList_t xDummy3[ 2 ];
1331     UBaseType_t uxDummy4[ 3 ];
1332     uint8_t ucDummy5[ 2 ];
1333
1334 #if ( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
1335 |     uint8_t ucDummy6;
1336 #endif
1337
1338 #if ( configUSE_QUEUE_SETS == 1 )
1339 |     void * pvDummy7;
1340 #endif
1341
1342 #if ( configUSE_TRACE_FACILITY == 1 )
1343 |     UBaseType_t uxDummy8;
1344 |     uint8_t ucDummy9;
1345 #endif
1346 } staticQueue_t;
1347 typedef StaticQueue_t StaticSemaphore_t;
1348

```

删除信号量

对于动态创建的信号量，不再需要它们时，可以删除它们以回收内存。

vSemaphoreDelete可以用来删除二进制信号量、计数型信号量，函数原型如下：

```
1 /*  
2 * xSemaphore: 信号量句柄, 你要删除哪个信号量  
3 */  
4 void vSemaphoreDelete( SemaphoreHandle_t xSemaphore );
```

give/take

二进制信号量、计数型信号量的give、take操作函数是一样的。

这些函数也分为2个版本：给任务使用，给ISR使用。

列表如下：

	在任务中使用	在ISR中使用
give	xSemaphoreGive	xSemaphoreGiveFromISR
take	xSemaphoreTake	xSemaphoreTakeFromISR

xSemaphoreGive的函数原型如下：

```
1 BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

函数的参数与返回值列表如下：

参数	说明
xSemaphore	信号量句柄，释放哪个信号量
返回值	pdTRUE表示成功,如果二进制信号量的计数值已经是1，再次调用此函数则返回失败；如果计数型信号量的计数值已经是最大值，再次调用此函数则返回失败

xSemaphoreGiveFromISR的原型

```
1 BaseType_t xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore,  
2 BaseType_t *pxHigherPriorityTaskWoken  
3 );
```

参数	说明
xSemaphore	信号量句柄，释放哪个信号量
pxHigherPriorityTaskWoken	如果释放信号量导致更高优先级的任务变为了就绪态，则 *pxHigherPriorityTaskWoken = pdTRUE
返回值	pdTRUE表示成功,如果二进制信号量的计数值已经是1，再次调用此函数则返回失败；如果计数型信号量的计数值已经是最大值，再次调用此函数则返回失败

xSemaphoreTake的函数原型如下：

```
1 BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore,
2 TickType_t xTicksToWait
3 );
```

xSemaphoreTake函数的参数与返回值列表如下：

参数	说明
xSemaphore	信号量句柄，获取哪个信号量
xTicksToWait	如果无法马上获得信号量，阻塞一会： 0：不阻塞，马上返回 portMAX_DELAY：一直阻塞直到成功 其他值：阻塞的Tick个数，可以使用 pdMS_TO_TICKS() 来指定阻塞时间为若干ms
返回值	pdTRUE表示成功

xSemaphoreTakeFromISR的函数原型如下：

```
1 BaseType_t xSemaphoreTakeFromISR(SemaphoreHandle_t xSemaphore,
2 BaseType_t *pxHigherPriorityTaskWoken
3 );
```

参数列表：

参数	说明
xSemaphore	信号量句柄，获取哪个信号量
pxHigherPriorityTaskWoken	如果获取信号量导致更高优先级的任务变为了就绪态，则*pxHigherPriorityTaskWoken = pdTRUE
返回值	pdTRUE表示成功

二进制信号量实现同步/互斥

代码示例：

主逻辑：

```
SemaphoreHandle_t xBinarySemaphore;

int main( void )
{
    prvSetupHardware();

    /* 创建二进制信号量 */
    xBinarySemaphore = xSemaphoreCreateBinary();

    if( xBinarySemaphore != NULL )
    {
        /* 创建1个任务用于释放信号量
         * 优先级为2
         */
        xTaskCreate( vSenderTask, "Sender", 1000, NULL, 2, NULL );

        /* 创建1个任务用于获取信号量
         * 优先级为1
         */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* 启动调度器 */
        vTaskStartScheduler();
    }
    else
    {
        /* 无法创建二进制信号量 */
    }

    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}
```

发送, 接收任务：

- A: 发送任务优先级高, 先执行。连续3次释放二进制信号量, 只有第1次成功
- B: 发送任务进入阻塞态
- C: 接收任务得以执行, 得到信号量, 打印OK; 再次去获得信号量时, 进入阻塞状态
- 在发送任务的vTaskDelay退出之前, 运行的是空闲任务: 现在发送任务、接收任务都阻塞了
- D: 发送任务再次运行, 连续3次释放二进制信号量, 只有第1次成功
- E: 发送任务进入阻塞态
- F: 接收任务被唤醒, 得到信号量, 打印OK; 再次去获得信号量时, 进入阻塞状态

```

123 static void vSenderTask( void *pvParameters )
124 {
125     int i;
126     int cnt_ok = 0;
127     int cnt_err = 0;
128     const TickType_t xTicksToWait = pdMS_TO_TICKS( 10UL );
129
130     /* 无限循环 */
131     for( ; )
132     {
133         for (i = 0; i < 3; i++)
134         {
135             if (xSemaphoreGive(xBinarySemaphore) == pdTRUE)
136                 printf("Give BinarySemaphore %d time: OK\r\n", cnt_ok++);
137             else
138                 printf("Give BinarySemaphore %d time: ERR\r\n", cnt_err++);
139         }
140         vTaskDelay(xTicksToWait); B E
141     }
142 }
143 
```

```

146 static void vReceiverTask( void *pvParameters )
147 {
148     int cnt_ok = 0;
149     int cnt_err = 0;
150     /* 无限循环 */
151     for( ; )
152     {
153         if( xSemaphoreTake(xBinarySemaphore, portMAX_DELAY) == pdTRUE )
154         {
155             /* 得到了二进制信号量 */
156             printf("Get BinarySemaphore OK: %d\r\n", cnt_ok++);
157         }
158         else
159         {
160             /* 没有得到了二进制信号量 */
161             printf("Get BinarySemaphore ERR: %d\r\n", cnt_err++);
162         }
163     }
164 }
165 
```

结果：

UART #1

```

Give BinarySemaphore 0 time: OK
Give BinarySemaphore 0 time: ERR
Give BinarySemaphore 1 time: ERR
Get BinarySemaphore OK: 0
Give BinarySemaphore 1 time: OK
Give BinarySemaphore 2 time: ERR
Give BinarySemaphore 3 time: ERR
Get BinarySemaphore OK: 1
Give BinarySemaphore 2 time: OK
Give BinarySemaphore 4 time: ERR
Give BinarySemaphore 5 time: ERR
Get BinarySemaphore OK: 2
Give BinarySemaphore 3 time: OK
Give BinarySemaphore 6 time: ERR
Give BinarySemaphore 7 time: ERR
Get BinarySemaphore OK: 3
Give BinarySemaphore 4 time: OK
Give BinarySemaphore 8 time: ERR

```

同步：先V后P (前一个任务先释放，后一个任务再申请)

互斥：先P后V (先申请资源，再释放资源)

防止数据丢失

发送任务发出3次"提醒"，但是接收任务只接收到1次"提醒"，其中2次"提醒"丢失了。

这种情况很常见，比如每接收到一个串口字符，串口中断程序就给任务发一次"提醒"，假设收到多个字符、发出了多次"提醒"。

当任务来处理时，它只能得到1次"提醒"。

故而，需要一些方式来防止数据的丢失：

- 在串口中断中，把数据放入缓冲区
- 在任务中，一次性把缓冲区中的数据都读出
- 简单地说，就是：你提醒了我多次，我太忙只响应你一次，但是我一次性拿走所有数据

示例：

```
/* 在 main 函数中 */
SemaphoreHandle_t xBinarySemaphore;

int main( void )
{
    prvSetupHardware();

    /* 创建二进制信号量 */
    xBinarySemaphore = xSemaphoreCreateBinary();

    if( xBinarySemaphore != NULL )
    {
        /* 创建1个任务用于释放信号量
         * 优先级为2
         */
        xTaskCreate( vSenderTask, "Sender", 1000, NULL, 2, NULL );

        /* 创建1个任务用于获取信号量
         * 优先级为1
         */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* 启动调度器 */
        vTaskStartScheduler();
    }
    else
    {
        /* 无法创建二进制信号量 */
    }

    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}
```

- A: 发送任务优先级高，先执行。连续写入3个数据、释放3个信号量：只有1个信号量起作用
- B: 发送任务进入阻塞态
- C: 接收任务得以执行，得到信号量
- D: 接收任务一次性把所有数据取出
- E: 接收任务再次尝试获取信号量，进入阻塞状态
- 在发送任务的vTaskDelay退出之前，运行的是空闲任务：现在发送任务、接收任务都阻塞了
- F: 发送任务再次运行，连续写入3个数据、释放3个信号量：只有1个信号量起作用
- G: 发送任务进入阻塞态
- H: 接收任务被唤醒，得到信号量，一次性把所有数据取出

```

158 static void vSenderTask( void *pvParameters )
159 {
160     int i;
161     int cnt_tx = 0;
162     int cnt_ok = 0;
163     int cnt_err = 0;
164     const TickType_t xTicksToWait = pdMS_TO_TICKS( 10UL );
165
166     /* 无限循环 */
167     for( ; ) { A F
168         for ( i = 0; i < 3; i++ )
169         {
170             /* 放入数据 */
171             txbuf_put('a'+cnt_tx);
172             cnt_tx++;
173
174             /* 提醒对方 */
175             if (xSemaphoreGive(xBinarySemaphore) == pdTRUE)
176                 printf("Give BinarySemaphore %d time: OR\r\n", cnt_ok++);
177             else
178                 printf("Give BinarySemaphore %d time: ERR\r\n", cnt_err++);
179
180         }
181         vTaskDelay(xTicksToWait); B G
182     }
183 }
184 
```



```

187 static void vReceiverTask( void *pvParameters )
188 {
189     int cnt_ok = 0;
190     int cnt_err = 0;
191     uint8_t c;
192
193     /* 无限循环 */
194     for( ; ) { C E
195         if( xSemaphoreTake(xBinarySemaphore, portMAX_DELAY) == pdTRUE )
196         {
197             /* 得到了二进制信号量 */
198             printf("Get BinarySemaphore OK: %d, data: ", cnt_ok++);
199
200             /* 一次性把所有数据取出来 */
201             while (txbuf_get(&c) == 0)
202             {
203                 printf("%c", c);
204             }
205             printf("\r\n");
206
207         }
208         else
209         {
210             /* 没有得到了二进制信号量 */
211             printf("Get BinarySemaphore ERR: %d\r\n", cnt_err++);
212         }
213     }
214 }
215 
```

计数型信号量使用

使用计数型信号量时，可以多次释放信号量；当信号量的技术值达到最大时，再次释放信号量就会出错。

如果信号量计数值为n，就可以连续n次获取信号量，第(n+1)次获取信号量就会阻塞或失败。

示例：

主逻辑：

```

SemaphoreHandle_t xCountingSemaphore;

int main( void )
{
    prvSetupHardware();

    /* 创建计数型信号量 */
    xCountingSemaphore = xSemaphoreCreateCounting(3, 0);

    if( xCountingSemaphore != NULL )
    {
        /* 创建1个任务用于释放信号量
         * 优先级为2
         */
        xTaskCreate( vSenderTask, "Sender", 1000, NULL, 2, NULL );

        /* 创建1个任务用于获取信号量
         * 优先级为1
         */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* 启动调度器 */
        vTaskStartScheduler();
    }
    else
    {
        /* 无法创建信号量 */
    }

    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}

```

发送任务、接收任务的代码和执行流程如下：

- A：发送任务优先级高，先执行。连续释放4个信号量：只有前面3次成功，第4次失败

- B: 发送任务进入阻塞态
- CDE: 接收任务得以执行, 得到3个信号量
- F: 接收任务试图获得第4个信号量时进入阻塞状态
- 在发送任务的vTaskDelay退出之前, 运行的是空闲任务: 现在发送任务、接收任务都阻塞了
- G: 发送任务再次运行, 连续释放4个信号量: 只有前面3次成功, 第4次失败
- H: 发送任务进入阻塞态
- IJK: 接收任务得以执行, 得到3个信号量
- L: 接收任务再次获取信号量时进入阻塞状态

```

122 static void vSenderTask( void *pvParameters )
123 {
124     int i;
125     int cnt_ok = 0;
126     int cnt_err = 0;
127     const TickType_t xTicksToWait = pdMS_TO_TICKS( 20UL );
128
129     /* 无限循环 */
130     for( ;; )
131     {
132         for ( i = 0; i < 4; i++ )
133         {
134             /* 提醒对方 */
135             if ( xSemaphoreGive( xCountingSemaphore ) == pdTRUE )
136                 printf("Give CountingSemaphore #d time: OK\r\n", cnt_ok++);
137             else
138                 printf("Give CountingSemaphore #d time: ERR\r\n", cnt_err++);
139         }
140
141         vTaskDelay( xTicksToWait ); B H
142     }
143 }
```

```

146 static void vReceiverTask( void *pvParameters )
147 {
148     int cnt_ok = 0;
149     int cnt_err = 0;
150
151     /* 无限循环 */
152     for( ;; )
153     {
154         if( xSemaphoreTake( xCountingSemaphore, portMAX_DELAY ) == pdTRUE )
155         {
156             /* 得到了信号量 */
157             printf("Get CountingSemaphore OK: %d\r\n", cnt_ok++);
158         }
159         else
160         {
161             /* 没有得到了信号量 */
162             printf("Get CountingSemaphore ERR: %d\r\n", cnt_err++);
163         }
164     }
165 }
```

结果:

```

UART #1

Give CountingSemaphore 0 time: OK
Give CountingSemaphore 1 time: OK
Give CountingSemaphore 2 time: OK
Give CountingSemaphore 0 time: ERR
Get CountingSemaphore OK: 0
Get CountingSemaphore OK: 1
Get CountingSemaphore OK: 2
Give CountingSemaphore 3 time: OK
Give CountingSemaphore 4 time: OK
Give CountingSemaphore 5 time: OK
Give CountingSemaphore 1 time: ERR
Get CountingSemaphore OK: 3
Get CountingSemaphore OK: 4
Get CountingSemaphore OK: 5
Give CountingSemaphore 6 time: OK
Give CountingSemaphore 7 time: OK
Give CountingSemaphore 8 time: OK
Give CountingSemaphore 2 time: ERR

```

八、互斥量(Mutex)

使用队列、信号量, 都可以实现互斥访问, 以信号量为例:

- 信号量初始值为1
- 任务A想上厕所, "take"信号量成功, 它进入厕所
- 任务B也想上厕所, "take"信号量不成功, 等待
- 任务A用完厕所, "give"信号量; 轮到任务B使用

这需要有2个前提:

- 任务B很老实, 不撬门(一开始不"give"信号量)
- 没有坏人: 别的任务不会"give"信号量
(自己give, 自己take就很无语)

使用互斥量可以解决这个问题, 互斥量的名字取得很好:

- 量：值为0、1
- 互斥：用来实现互斥访问
它的核心在于：谁上锁，就只能由谁开锁。

很奇怪的是，FreeRTOS的互斥锁，并没有在代码上实现这点：

- 即使任务A获得了互斥锁，任务B竟然也可以释放互斥锁。
- 谁上锁、谁释放：只是约定。

互斥锁和二进制信号量的对比：

- 互斥锁解决了优先级反转的问题
- 解决了递归上锁解锁的问题

互斥量的使用

在多任务系统中，任务A正在使用某个资源，还没用完的情况下任务B也来使用的话，就可能导致问题。比如对于串口，任务A正使用它来打印，在打印过程中任务B也来打印，客户看到的结果就是A、B的信息混杂在一起。

这种现象很常见：

- 访问外设：刚举的串口例子
- 读、修改、写操作导致的问题

对于同一个变量，比如int a，如果有两个任务同时写它就有可能导致问题。

对于变量的修改，C代码只有一条语句，比如：a=a+8；，它的内部实现分为3步：读出原值、修改、写入。

```
int a = 1;
```

```
void add_a(void)
{
    a = a + 8; } 
```

①: R0 = [a地址]
②: R0 = R0+8
③: [a地址] = R0

我们想让任务A、B都执行add_a函数，a的最终结果是1+8+8=17。

假设任务A运行完代码①，在执行代码②之前被任务B抢占了：现在任务A的R0等于1。

任务B执行完add_a函数，a等于9。

任务A继续运行，在代码②处R0仍然是被抢占前的数值1，执行完②③的代码，a等于9，这跟预期的17不符合。

- 对变量的非原子化访问
修改变量、设置结构体、在16位的机器上写32位的变量，这些操作都是非原子的。也就是它们的操作过程都可能被打断，如果被打断的过程有其他任务来操作这些变量，就可能导致冲突。

- 函数重入

"可重入的函数"是指：多个任务同时调用它、任务和中断同时调用它，函数的运行也是安全的。

可重入的函数也被称为"线程安全"(thread safe)。

每个任务都维持自己的栈、自己的CPU寄存器，如果一个函数只使用局部变量，那么它就是线程安全的。

函数中一旦使用了全局变量、静态变量、其他外设，它就不是“可重入的”，如果改函数正在被调用，就必须阻止其他任务、中断再次调用它。

上述问题的解决方法是：

任务A访问这些全局变量、函数代码时，独占它，就是上个锁。

这些全局变量、函数代码必须被独占地使用，它们被称为临界资源。

互斥量也被称为互斥锁，使用过程如下：

- 互斥量初始值为1
- 任务A想访问临界资源，先获得并占有互斥量，然后开始访问
- 任务B也想访问临界资源，也要先获得互斥量：被别人占有了，于是阻塞
- 任务A使用完毕，释放互斥量；任务B被唤醒、得到并占有互斥量，然后开始访问临界资源
- 任务B使用完毕，释放互斥量

正常来说：在任务A占有互斥量的过程中，任务B、任务C等等，都无法释放互斥量。

但是FreeRTOS未实现这点：任务A占有互斥量的情况下，任务B也可释放互斥量。

互斥量函数

创建

互斥量是一种特殊的二进制信号量。

使用互斥量时，先创建、然后去获得、释放它。使用句柄来表示一个互斥量。

创建互斥量的函数有2种：动态分配内存，静态分配内存。

函数原型如下：

```
1  /* 创建一个互斥量，返回它的句柄。  
2   * 此函数内部会分配互斥量结构体  
3   * 返回值：返回句柄，非NULL表示成功  
4   */  
5   SemaphoreHandle_t xSemaphoreCreateMutex( void );  
6  /* 创建一个互斥量，返回它的句柄。  
7   * 此函数无需动态分配内存，所以需要先有一个StaticSemaphore_t结构体，并传入它的指针  
8   * 返回值：返回句柄，非NULL表示成功  
9   */  
10  SemaphoreHandle_t xSemaphoreCreateMutexStatic( StaticSemaphore_t  
*pxMutexBuffer);
```

要想使用互斥量，需要在配置文件FreeRTOSConfig.h中定义：

```
1 #define configUSE_MUTEXES 1
```

其他函数

要注意的是，在FreeRTOS中，互斥量（Mutex）不能在中断服务例程（ISR）中使用。

互斥量是设计用于任务之间的同步机制，而不是用于中断上下文。

如果需要在ISR中进行同步，可以使用FreeRTOS提供的其他机制，例如二值信号量（Binary Semaphore）或队列（Queue）。

FreeRTOS提供了专门的API函数用于在ISR中操作这些同步机制，例如xSemaphoreGiveFromISR和xQueueSendFromISR

各类操作函数，比如删除、give/take，跟一般是信号量是一样的。

```
1  /*
2  * xSemaphore: 信号量句柄，你要删除哪个信号量，互斥量也是一种信号量
3  */
4  void vSemaphoreDelete( SemaphoreHandle_t xSemaphore );
5
6  /* 释放 */
7  BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
8
9  /* 释放(ISR版本) */
10 BaseType_t xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore, BaseType_t
11 *pxHigherPriorityTaskWoken
12 );
13
14 /* 获得 */
15 BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t
16 xTicksToWait
17 );
18
19 /* 获得(ISR版本) */
20 xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore, BaseType_t
21 *pxHigherPriorityTaskWoken
22 );
```

互斥量基本使用

使用互斥量时有如下特点：

- 刚创建的互斥量可以被成功"take"
- "take"互斥量成功的任务，被称为"holder"，只能由它"give"互斥量；别的任务"give"不成功
- 在ISR中不能使用互斥量

示例:

main逻辑

```
/* 互斥量句柄 */
SemaphoreHandle_t xMutex;

int main( void )
{
    prvSetupHardware();

    /* 创建互斥量 */
    xMutex = xSemaphoreCreateMutex( );

    if( xMutex != NULL )
    {
        /* 创建2个任务：都是打印
         * 优先级相同
         */
        xTaskCreate( vSenderTask, "Sender1", 1000, (void *)1, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, (void *)2, 1, NULL );

        /* 启动调度器 */
        vTaskStartScheduler();
    }
    else
    {
        /* 无法创建互斥量 */
    }

    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}
```

发送

```

static void vSenderTask( void *pvParameters )
{
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 10UL );

```

```

int cnt = 0;
int task = (int)pvParameters;
int i;
char c;

/* 无限循环 */
for( ;; )
{
    /* 获得互斥量：上锁 */
    xSemaphoreTake(xMutex, portMAX_DELAY);

    printf("Task %d use UART count: %d, ", task, cnt++);
    c = (task == 1) ? 'a' : 'A';
    for (i = 0; i < 26; i++)
        printf("%c", c + i);
    printf("\r\n");

    /* 释放互斥量：开锁 */
    xSemaphoreGive(xMutex);

    vTaskDelay(xTicksToWait);
}
}

```

保留互斥量和去除互斥量的差别如下：

使用互斥量

UART #1
Task 2 use UART count: 0, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 0, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 1, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 1, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 2, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 2, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 3, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 3, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 4, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 4, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 5, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 5, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 6, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 6, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 7, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 7, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 8, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 8, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 9, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 9, abcdefghijklmnopqrstuvwxyz
Task 2 use UART count: 10, ABCDEFGHIJKLMNOPQRSTUVWXYZ

没使用互斥量

UART #1
Task 2 use UART count: 0, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 0, abcdefghijklmnopqrstuvwxyz
Task 2 Task 1 use UART count: 1, abcdefghijklmnopqrstuvwxyz
use UART count: 1, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 2, ABCDEFGHIJKLMNOPQRSTUVWXYZ
RT count: 2, abcdefghijklmnopqrstuvwxyz
Task 2 use UART countTask 1 use UART count: 3, abcdefghijklmnopqrstuvwxyz
Task 2 use UART countTask 1 use UART count: 3, abcdefghijklmnopqrstuvwxyz
Task 1 use UART count: 4, Task 2 use UART count: 4, ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefgijklmnopqrstuvwxyz
Task 2 use UART count: 5, ABCDEFGTask 1 use UART count: 5, abcdefghijklmnopqrstuvwxyz
HJKLMNOPQRSTUVWXYZ
Task 1 use UART count: 6, abcdefghijklmTask 2 use UART count: 6, ABCDEFGHIJKLMNOPQRSTUVWXYZ
mnopqrstuvwxyz
Task 2 use UART count: 7, ABCDEFGHIJKLMNOPQRSTUVWXYZTask 1 use UART count: 7, abcdefghijklmnopqrstuvwxyz
UVWXYZ
Task 1 use UART count: 8, abcdefghijklmnopqrstuvwxyzTask 2 use UART count: 8, ABCDEFGHIJKLMNOPQRSTUVWXYZ
mnopqrstuvwxyz
Task 2 use UART count: 9, ABCDEFGHIJKLMNOPQRSTUVWXYZ
Task Task 2 use UART count: 10, ABCDEFGHIJKLMNOPQRSTUVWXYZ
1 use UART count: 9, abcdefghijklmnopqrstuvwxyz

FreeRTOS中互斥量的缺陷

互斥量、互斥锁，本来的概念确实是：谁上锁就得由谁解锁。
但是FreeRTOS并没有实现这点，只是要求程序员按照这样的惯例写代码。

优先级反转与优先级继承

假设任务A、B都想使用串口，A优先级比较低：

- 任务A获得了串口的互斥量
- 任务B也想使用串口，它将会阻塞、等待A释放互斥量

- 高优先级的任务，被低优先级的任务延迟，这被称为“优先级反转”(priority inversion)

图解：

3个任务解决优先级反转：

A: 1

B: 2

C: 3 A lock

C lock失败

while(1)

C的优先级高于B，但C不能执行

- A先上锁，执行临界区代码
- B进入就绪队列，优先级高于A，B先执行
- C进入就绪队列，优先级高于B，C先执行，获取锁失败，阻塞
- B开始执行执行，C优先级大于B，但是需要等待B的执行

互斥量可以通过“优先级继承”，可以很大程度解决“优先级反转”的问题，这也是FreeRTOS中互斥量和二进制信号量的差别。

优先级继承图解：

解决：优先级继承

lock失败阻塞

C C优先级继承A

C work

A lock

B

A unlock (恢复优先级)

B work

A lost

C 优先级继承给A，使A高于B运行

- A上锁，执行临界区代码
- B进入就绪队列，优先级高于A，B先执行
- C进入就绪队列，优先级高于B，C先执行，获取锁失败，阻塞，并将自己的优先级继承给A。
- A获得了更高的优先级，先执行，解锁，退出临界区，恢复优先级
- C获得锁，运行
- C运行完毕B运行
- B运行完毕，A继续运行

优先级继承是二进制信号量所没有的

但是二进制信号量可以在ISR中断服务程序中进行。

递归锁

死锁

日常生活的死锁：我们只招有工作经验的人！我没有工作经验怎么办？那你就去找工作啊！

假设有2个互斥量M1、M2，2个任务A、B：

- A获得了互斥量M1
- B获得了互斥量M2
- A还要获得互斥量M2才能运行，结果A阻塞
- B还要获得互斥量M1才能运行，结果B阻塞
- A、B都阻塞，再无法释放它们持有的互斥量
- 死锁发生！

自我死锁(递归上锁)

假设这样的场景：

- 任务A获得了互斥锁M
- 它调用一个库函数
- 库函数要去获取同一个互斥锁M，于是它阻塞：任务A休眠，等待任务A来释放互斥锁！
- 死锁发生！

```
142: void XXXlib(void)
143: {
144:     xSemaphoreTake(xSemUART, portMAX_DELAY);
145:     printf("xxxx");
146:     xSemaphoreGive(xSemUART);
147: }
148:
149:
150: void TaskGenericFunction(void * param)
151: {
152:     while (1)
153:     {
154:         xSemaphoreTake(xSemUART, portMAX_DELAY);
155:         printf("%s\r\n", (char *)param);
156:         XXXlib();
157:         xSemaphoreGive(xSemUART);
158:         vTaskDelay(1);
159:     }
160: }
```

递归锁方案

可以使用递归锁(Recursive Mutexes)，去解决自我死锁的问题：

- 任务A获得递归锁M后，它还可以多次去获得这个锁
- "take"了N次，要"give"N次，这个锁才会被释放

递归锁的函数名一般互斥量的函数名不一样，参数类型一样，列表如下：

	递归锁	一般互斥量
创建	xSemaphoreCreateRecursiveMutex	xSemaphoreCreateMutex
获得	xSemaphoreTakeRecursive	xSemaphoreTake
释放	xSemaphoreGiveRecursive	xSemaphoreGive

函数原型如下：

```

1  /* 创建一个递归锁，返回它的句柄。
2  * 此函数内部会分配互斥量结构体
3  * 返回值：返回句柄，非NULL表示成功
4  */
5  SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( void );
6
7  /* 释放 */
8  BaseType_t xSemaphoreGiveRecursive( SemaphoreHandle_t xSemaphore );
9
10 /* 获得 */
11 BaseType_t xSemaphoreTakeRecursive(SemaphoreHandle_t xSemaphore,
12 TickType_t xTicksToWait
13 );
14

```

递归锁使用

递归锁实现了：谁上锁就由谁解锁。

主函数:

```
/* 递归锁句柄 */
SemaphoreHandle_t xMutex;

int main( void )
{
    prvSetupHardware();

    /* 创建递归锁 */
    xMutex = xSemaphoreCreateRecursiveMutex();

    if( xMutex != NULL )
    {
        /* 创建2个任务：一个上锁，另一个自己监守自盗(看看能否开别人的锁自己用)
         */
        xTaskCreate( vTakeTask, "Task1", 1000, NULL, 2, NULL );
        xTaskCreate( vGiveAndTakeTask, "Task2", 1000, NULL, 1, NULL );

        /* 启动调度器 */
        vTaskStartScheduler();
    }
    else
    {
        /* 无法创建递归锁 */
    }

    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}
```

设计两个任务：

- A: 任务1优先级最高，先运行，获得递归锁
- B: 任务1阻塞，让任务2得以运行
- C: 任务2运行，看看能否获得别人持有的递归锁：不能
- D: 任务2故意执行"give"操作，看看能否释放别人持有的递归锁：不能
- E: 任务2等待递归锁
- F: 任务1阻塞时间到后继续运行，使用循环多次获得、释放递归锁
- 递归锁在代码上实现了：谁持有递归锁，必须由谁释放。

```

116 static void vTakeTask( void *pvParameters )
117 {
118     const TickType_t xTicksToWait = pdMS_TO_TICKS( 100UL );
119     BaseType_t xStatus;
120     int i;
121
122     /* 无限循环 */
123     for(;;)
124     {
125         /* 获得递归锁: 上锁 */
126         xStatus = xSemaphoreTakeRecursive(xMutex, portMAX_DELAY);
127         printf("Task1 take the Mutex in main loop %s\r\n", \
128             (xStatus == pdTRUE) ? "Success" : "Failed");
129
130         /* 因塞很长时间, 让另一个任务执行,
131          看看它有无办法再次获得递归锁
132          */
133
134         vTaskDelay(xTicksToWait);
135
136         for( i = 0; i < 10; i++)
137         {
138             /* 获得递归锁: 上锁 */
139             xStatus = xSemaphoreTakeRecursive(xMutex, portMAX_DELAY);
140
141             printf("Task1 take the Mutex in sub loop %s, for time %d\r\n", \
142                 (xStatus == pdTRUE) ? "Success" : "Failed", i);
143
144             /* 释放递归锁 */
145             xSemaphoreGiveRecursive(xMutex);
146         }
147
148         /* 释放递归锁 */
149         xSemaphoreGiveRecursive(xMutex);
150     }
151 }

```

A

B

F

G

```

153 static void vGiveAndTakeTask( void *pvParameters )
154 {
155     const TickType_t xTicksToWait = pdMS_TO_TICKS( 100UL );
156     BaseType_t xStatus;
157
158     /* 尝试获得递归锁: 上锁 */
159     xStatus = xSemaphoreTakeRecursive(xMutex, 0);
160     printf("Task2: at first, take the Mutex %s\r\n", \
161         (xStatus == pdTRUE) ? "Success" : "Failed");
162
163     /* 如果失败则监守自盗: 开锁 */
164     if (xStatus != pdTRUE)
165     {
166         /* 无法释放别人持有的锁 */
167         xStatus = xSemaphoreGiveRecursive(xMutex);
168         printf("Task2: give Mutex %s\r\n", \
169             (xStatus == pdTRUE) ? "Success" : "Failed");
170     }
171
172     /* 如果无法获得, 一直等待 */
173     xStatus = xSemaphoreTakeRecursive(xMutex, portMAX_DELAY);
174     printf("Task2: and then, take the Mutex %s\r\n", \
175         (xStatus == pdTRUE) ? "Success" : "Failed");
176
177     /* 无限循环 */
178     for(;;)
179     {
180         /* 什么都不做 */
181         vTaskDelay(xTicksToWait);
182     }
183 }

```

C

D

E

结果:

UART #1

```

Task1 take the Mutex in main loop Success
Task2: at first, take the Mutex Failed
Task2: give Mutex Failed

Task1 take the Mutex in sub loop Success, for time 0
Task1 take the Mutex in sub loop Success, for time 1
Task1 take the Mutex in sub loop Success, for time 2
Task1 take the Mutex in sub loop Success, for time 3
Task1 take the Mutex in sub loop Success, for time 4
Task1 take the Mutex in sub loop Success, for time 5
Task1 take the Mutex in sub loop Success, for time 6
Task1 take the Mutex in sub loop Success, for time 7
Task1 take the Mutex in sub loop Success, for time 8
Task1 take the Mutex in sub loop Success, for time 9
Task1 take the Mutex in main loop Success
Task1 take the Mutex in sub loop Success, for time 0
Task1 take the Mutex in sub loop Success, for time 1
Task1 take the Mutex in sub loop Success, for time 2
Task1 take the Mutex in sub loop Success, for time 3
Task1 take the Mutex in sub loop Success, for time 4
Task1 take the Mutex in sub loop Success, for time 5
Task1 take the Mutex in sub loop Success, for time 6

```

总结:

互斥量与二进制信号量

- 互斥量创建后无需手动Give(初始值默认是1)
- 二进制信号量创建后需要手动Give(初始值默认是0)
- 互斥量实现了优先级继承, 解决了优先级反转的问题
- 互斥量和二进制信号量都没有解决递归上锁的问题
- 互斥量和二进制信号量的使用完全一致
- 互斥量无法在ISR中使用, 二进制信号量则可以

互斥量与递归锁

- 递归锁解决了递归上锁的问题
- 递归锁实现了谁上锁谁解锁

九、事件组

学校组织秋游，组长在等待：

- 张三：我到了
- 李四：我到了
- 王五：我到了
- 组长说：好，大家都到齐了，出发！

秋游回来第二天就要提交一篇心得报告，组长在焦急等待：张三、李四、王五谁先写好就交谁的。

在这个日常生活场景中：

- 出发：要等待这3个人都到齐，他们是“与”的关系
- 交报告：只需等待这3人中的任何一个，他们是“或”的关系

在FreeRTOS中，可以使用事件组(event group)来解决这些问题。

本章涉及如下内容：

- 事件组的概念与操作函数
- 事件组的优缺点
- 怎么设置、等待、清除事件组中的位
- 使用事件组来同步多个任务

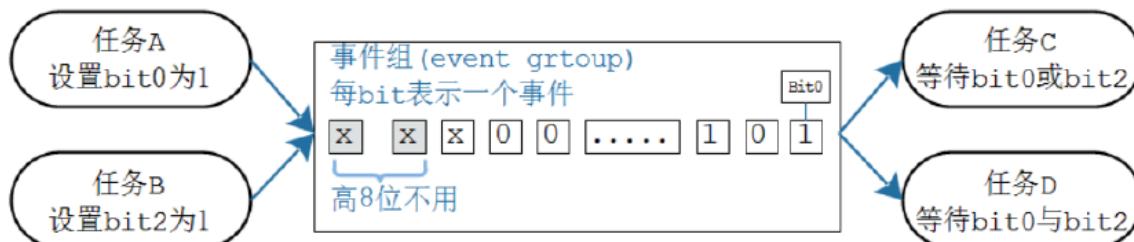
事件组概念与操作

概念

事件组可以和位图的概念对应：

- 每一位表示一个事件
- 每一位事件的含义由程序员决定，比如：Bit0表示用来串口是否就绪，Bit1表示按键是否被按下
- 这些位，值为1表示事件发生了，值为0表示事件没发生
- 一个或多个任务、ISR都可以去写这些位；一个或多个任务、ISR都可以去读这些位
- 可以等待某一位、某些位中的任意一个，也可以等待多位

图解：



事件组用一个整数来表示，其中的高8位留给内核使用，只能用其他的位来表示事件。**(高八位由内核使用)**

那么这个整数是多少位的？

- 如果configUSE_16_BIT_TICKS是1，那么这个整数就是16位的，低8位用来表示事件
- 如果configUSE_16_BIT_TICKS是0，那么这个整数就是32位的，低24位用来表示事件

configUSE_16_BIT_TICKS是用来表示Tick Count的，怎么会影响事件组？

这只是基于效率来考虑

- 如果configUSE_16_BIT_TICKS是1，就表示该处理器使用16位更高效，所以事件组也使用16位
- 如果configUSE_16_BIT_TICKS是0，就表示该处理器使用32位更高效，所以事件组也使用32位

事件组操作

事件组和队列、信号量等不太一样，主要集中在2个地方：

- 唤醒谁？
 - 队列、信号量：事件发生时，只会唤醒一个任务
 - 事件组：事件发生时，会唤醒所有符合条件的任务，简单地说它有“广播”的作用
- 是否清除事件？
 - 队列、信号量：是消耗型的资源，队列的数据被读走就没了；信号量被获取后就减少了
 - 事件组：被唤醒的任务有两个选择，可以让事件保留不动，也可以清除事件

事件组的常规操作：

- 先创建事件组
- 任务C、D等待事件：
 - 等待什么事件？可以等待某一位、某些位中的任意一个，也可以等待多位。简单地说就是“或”、“与”的关系。
 - 得到事件时，要不要清除？可选择清除、不清除。
- 任务A、B产生事件：设置事件组里的某一位、某些位

事件组函数

创建事件组

使用事件组之前，要先创建，得到一个句柄；

使用事件组时，要使用句柄来表明使用哪个事件组。

有两种创建方法：动态分配内存、静态分配内存。

函数原型如下：

```
1  /* 创建一个事件组，返回它的句柄。
2   * 此函数内部会分配事件组结构体
3   * 返回值：返回句柄，非NULL表示成功
4   */
5  EventHandle_t xEventGroupCreate( void );
6  /* 创建一个事件组，返回它的句柄。
7   * 此函数无需动态分配内存，所以需要先有一个StaticEventGroup_t结构体，并传入它的指针
8   * 返回值：返回句柄，非NULL表示成功
9   */
10 EventHandle_t xEventGroupCreateStatic( StaticEventGroup_t *
pxEventGroupBuffer );
```

删除事件组

对于动态创建的事件组，不再需要它们时，可以删除它们以回收内存。

vEventGroupDelete可以用来删除事件组，函数原型如下：

```
1  /*
2   * xEventGroup: 事件组句柄，你要删除哪个事件组
3   */
4  void vEventGroupDelete( EventGroupHandle_t xEventGroup )
```

设置事件组

可以设置事件组的某个位、某些位，使用的函数有2个：

- 在任务中使用xEventGroupSetBits()
- 在ISR中使用xEventGroupSetBitsFromISR()

有一个或多个任务在等待事件，如果这些事件符合这些任务的期望，那么任务还会被唤醒。

函数原型：

```
1  /* 设置事件组中的位
2   * xEventGroup: 哪个事件组
3   * uxBitsToSet: 设置哪些位？
4   * 如果uxBitsToSet的bitX, bitY为1, 那么事件组中的bitX, bitY被设置为1
5   * 可以用来设置多个位, 比如 0x15 就表示设置bit4, bit2, bit0
6   * 返回值: 返回原来的事件值(没什么意义, 因为很可能已经被其他任务修改了)
7   */
8  EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup,
9  const EventBits_t uxBitsToSet );
10
11 /* 设置事件组中的位
12 * xEventGroup: 哪个事件组
13 * uxBitsToSet: 设置哪些位？
14 * 如果uxBitsToSet的bitX, bitY为1, 那么事件组中的bitX, bitY被设置为1
15 * 可以用来设置多个位, 比如 0x15 就表示设置bit4, bit2, bit0
16 * pxHigherPriorityTaskWoken: 有没有导致更高优先级的任务进入就绪态? pdTRUE-
17 * 有, pdFALSE-没有
18 * 返回值: pdPASS-成功, pdFALSE-失败
19 */
20 BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup,
21 const EventBits_t uxBitsToSet,
22 BaseType_t * pxHigherPriorityTaskWoken );
```

值得注意的是，ISR中的函数，比如队列函数xQueueSendToBackFromISR、信号量函数xSemaphoreGiveFromISR，它们会唤醒某个任务，最多只会唤醒1个任务。

但是设置事件组时，有可能导致多个任务被唤醒，这会带来很大的不确定性。

所以xEventGroupSetBitsFromISR 函数不是直接去设置事件组，而是给一个FreeRTOS后台任务(daemon task)发送队列数据，由这个任务来设置事件组。

如果后台任务的优先级比当前被中断的任务优先级高，xEventGroupSetBitsFromISR 会设置*pxHigherPriorityTaskWoken 为pdTRUE。

如果daemon task成功地把队列数据发送给了后台任务，那么xEventGroupSetBitsFromISR 的返回值就是pdPASS。

等待事件

使用xEventGroupWaitBits 来等待事件，可以等待某一位、某些位中的任意一个，也可以等待多位；等到期望的事件后，还可以清除某些位。

函数原型如下：

```
1 EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup,
2     const EventBits_t uxBitsToWaitFor,
3     const BaseType_t xClearOnExit,
4     const BaseType_t xWaitForAllBits,
5     TickType_t xTicksToWait );
```

先引入一个概念：unblock condition。

一个任务在等待事件发生时，它处于阻塞状态；

当期望的时间发生时，这个状态就叫"unblock condition"，非阻塞条件，或称为"非阻塞条件成立"；

当"非阻塞条件成立"后，该任务就可以变为就绪态。

参数列表：

参数	说明
xEventGroup	等待哪个事件组？
uxBitsToWaitFor	等待哪些位？哪些位要被测试？
xWaitForAllBits	怎么测试？是"AND"还是"OR"？ pdTRUE: 等待的位，全部为1； pdFALSE: 等待的位，某一个为1即可
xClearOnExit	函数提出前是否要清除事件？ pdTRUE: 清除uxBitsToWaitFor指定的位 pdFALSE: 不清除
xTicksToWait	如果期待的事件未发生，阻塞多久。 可以设置为0：判断后即刻返回； 可设置为portMAX_DELAY：一定等到成功才返回； 可以设置为期望的Tick Count，一般用 pdMS_TO_TICKS() 把ms转换为Tick Count
返回值	返回的是事件值， 如果期待的事件发生了，返回的是"非阻塞条件成立"时的事件值； 如果是超时退出，返回的是超时时刻的事件值。

举例：

事件组的值	uxBitsToWaitFor	xWaitForAllBits	说明
0100	0101	pdTRUE	任务期望bit0,bit2都为1，当前值只有bit2满足，任务进入阻塞态；当事件组中bit0,bit2都为1时退出阻塞态
0100	0110	pdFALSE	任务期望bit0,bit2某一个为1，当前值满足，所以任务成功退出
0100	0110	pdTRUE	任务期望bit1,bit2都为1，当前值不满足，任务进入阻塞态；当事件组中bit1,bit2都为1时退出阻塞态

注意：

你可以使用xEventGroupWaitBits() 等待期望的事件

它发生之后再使用xEventGroupClearBits()来清除。

但是这两个函数之间，有可能被其他任务或中断抢占，它们可能会修改事件组。

可以使用设置xClearOnExit 为pdTRUE，使得对事件组的测试、清零都在xEventGroupWaitBits()函数内部完成，这是一个原子操作。

同步点

有一个事情需要多个任务协同，比如：

- 任务A：炒菜
- 任务B：买酒
- 任务C：摆台
- A、B、C做好自己的事后，还要等别人做完；大家一起做完，才可开饭

使用xEventGroupSync() 函数可以同步多个任务：

- 可以设置某位、某些位，表示自己做了什么事
- 可以等待某位、某些位，表示要等等其他任务
- 期望的时间发生后， xEventGroupSync() 才会成功返回。
- xEventGroupSync 成功返回后，会清除事件

函数原型如下：

```
1 EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup,
2     const EventBits_t uxBitsToSet,
3     const EventBits_t uxBitsToWaitFor,
4     TickType_t xTicksToWait );
```

参数列表如下：

参数列表如下：

参数	说明
xEventGroup	哪个事件组？
uxBitsToSet	要设置哪些事件？我完成了哪些事件？ 比如0x05(二进制为0101)会导致事件组的bit0,bit2被设置为1
uxBitsToWaitFor	等待那个位、哪些位？ 比如0x15(二级制10101)，表示要等待bit0,bit2,bit4都为1
xTicksToWait	如果期待的事件未发生，阻塞多久。 可以设置为0：判断后即刻返回； 可设置为portMAX_DELAY：一定等到成功才返回； 可以设置为期望的Tick Count，一般用 pdMS_TO_TICKS() 把ms转换为Tick Count
返回值	返回的是事件值， 如果期待的事件发生了，返回的是"非阻塞条件成立"时的事件值； 如果是超时退出，返回的是超时时刻的事件值。

事件组使用

等待多事件

添加事件组头文件

```
1 /* 1. 工程中添加event_groups.c */
2 /* 2. 源码中包含头文件 */
3 #include "event_groups.h"
```

main逻辑：

```
int main( void )
{
    prvSetupHardware();

    /* 创建递归锁 */
    xEventGroup = xEventGroupCreate( );

    if( xEventGroup != NULL )
    {
        /* 创建3个任务：洗菜/生火/炒菜
         */
        xTaskCreate( vWashingTask, "Task1", 1000, NULL, 1, NULL );
        xTaskCreate( vFiringTask, "Task2", 1000, NULL, 2, NULL );
        xTaskCreate( vCookingTask, "Task3", 1000, NULL, 3, NULL );

        /* 启动调度器 */
        vTaskStartScheduler();
    }
    else
    {
        /* 无法创建事件组 */
    }

    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}
```

制定三个任务：

- A: "炒菜任务"优先级最高，先执行。它要等待的2个事件未发生：洗菜、生火，进入阻塞状态
- B: "生火任务"接着执行，它要等待的1个事件未发生：洗菜，进入阻塞状态
- C: "洗菜任务"接着执行，它洗好菜，发出事件：洗菜，然后调用F等待"炒菜"事件
- D: "生火任务"等待的事件满足了，从B处继续执行，开始生火、发出"生火"事件
- E: "炒菜任务"等待的事件满足了，从A处继续执行，开始炒菜、发出"炒菜"事件
- F: "洗菜任务"等待的事件满足了，退出F、继续执行C

```
126 static void vWashingTask( void *pvParameters )
127 {
128     int i = 0;
129
130     /* 无限循环 */
131     for(;;)
132     {
133         printf("I am washing %d time...\r\n", i++);
134
135         /* 发出事件：我洗完菜了 */
136         xEventGroupSetBits(xEventGroup, WASHING);
137
138         /* 等待大厨炒完菜，再继续洗菜 */
139         xEventGroupWaitBits(xEventGroup, COOKING, pdTRUE, pdTRUE, portMAX_DELAY);
140     }
141 }
142
143 static void vFiringTask( void *pvParameters )
144 {
145     int i = 0;
146
147     /* 无限循环 */
148     for(;;)
149     {
150         /* 等待洗完菜，才生火 */
151         xEventGroupWaitBits(xEventGroup, WASHING, pdFALSE, pdTRUE, portMAX_DELAY);
152
153         printf("I am firing %d time...\r\n", i++);
154
155         /* 发出事件：我生好火了 */
156         xEventGroupSetBits(xEventGroup, FIRING);
157     }
158 }
159

160 static void vCookingTask( void *pvParameters )
161 {
162     int i = 0;
163
164     /* 无限循环 */
165     for(;;)
166     {
167         /* 等待2件事：洗菜、生火 */
168         xEventGroupWaitBits(xEventGroup, WASHING|FIRING, pdTRUE, pdTRUE, portMAX_DELAY);
169
170         printf("I am cooking %d time...\r\n", i++);
171
172         /* 发出事件：我炒好菜了 */
173         xEventGroupSetBits(xEventGroup, COOKING);
174     }
175 }
176 
```

结果：

```
UART #1
I am washing 0 time....
I am firing 0 time....
I am cooking 0 time....
I am washing 1 time....
I am firing 1 time....
I am cooking 1 time....
I am washing 2 time....
I am firing 2 time....
I am cooking 2 time....
I am washing 3 time....
I am firing 3 time....
I am cooking 3 time....
```

事件同步

主逻辑：

```
int main( void )
{
    prvSetupHardware();

    /* 创建递归锁 */
    xEventGroup = xEventGroupCreate( );

    if( xEventGroup != NULL )
    {
        /* 创建3个任务：洗菜/生火/炒菜
         */
        xTaskCreate( vCookingTask, "task1", 1000, "A", 1, NULL );
        xTaskCreate( vBuyingTask,   "task2", 1000, "B", 2, NULL );
        xTaskCreate( vTableTask,    "task3", 1000, "C", 3, NULL );

        /* 启动调度器 */
        vTaskStartScheduler();
    }
    else
    {
        /* 无法创建事件组 */
    }

    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}
```

创建三个任务，功能类似：

```
static void vCookingTask( void *pvParameters )
{
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100UL );
    int i = 0;

    /* 无限循环 */
    for( ;; )
    {
        /* 做自己的事 */

        printf("%s is cooking %d time....\r\n", (char *)pvParameters, i);

        /* 表示我做好了，还要等别人都做好 */
        xEventGroupSync(xEventGroup, COOKING, ALL, portMAX_DELAY);

        /* 别人也做好了，开饭 */
        printf("%s is eating %d time....\r\n", (char *)pvParameters, i++);
        vTaskDelay(xTicksToWait);
    }
}
```

要点在于xEventGroupSync 函数，它有3个功能：

- 设置事件：表示自己完成了某个、某些事件
- 等待事件：跟别的任务同步
- 成功返回后，清除“等待的事件”

运行结果：

```
UART #1
C is do the table 0 time....
B is buying 0 time....
A is cooking 0 time....
C is eating 0 time....
B is eating 0 time....
A is eating 0 time....
C is do the table 1 time....
B is buying 1 time....
A is cooking 1 time....
C is eating 1 time....
B is eating 1 time....
A is eating 1 time....
```

十、任务通知

所谓“任务通知”，你可以反过来读“通知任务”。

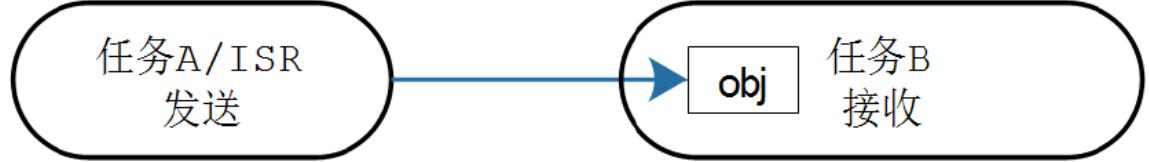
我们使用队列、信号量、事件组等等方法时，并不知道对方是谁。

使用任务通知时，可以明确指定：通知哪个任务。

使用队列、信号量、事件组时，我们都要事先创建对应的结构体，双方通过中间的结构体通信：



使用任务通知时，任务结构体TCB中就包含了内部对象，可以直接接收别人发过来的“通知”：



任务通知特性

优势和限制

任务通知的优势：

- 效率更高：
使用任务通知来发送事件、数据给某个任务时，效率更高。
比队列、信号量、事件组都有大的优势。
- 更节省内存：
使用其他方法时都要先创建对应的结构体，使用任务通知时无需额外创建结构体。

任务通知的限制：

- 不能发送数据给ISR：
ISR并没有任务结构体，所以无法使用任务通知的功能给ISR发送数据。
但是ISR可以使用任务通知的功能，发数据给任务。
- 数据只能给该任务独享
使用队列、信号量、事件组时，数据保存在这些结构体中，其他任务、ISR都可以访问这些数据。
使用任务通知时，数据存放入目标任务中，只有它可以访问这些数据。
在日常工作中，这个限制影响不大。
因为很多场合是从多个数据源把数据发给某个任务，而不是把一个数据源的数据发给多个任务。
- 无法缓冲数据
使用队列时，假设队列深度为N，那么它可以保持N个数据。
使用任务通知时，任务结构体中只有一个任务通知值，只能保持一个数据。
- 无法广播给多个任务
使用事件组可以同时给多个任务发送事件。
使用任务通知，只能发给一个任务。
- 如果发送受阻，发送方无法进入阻塞状态等待 (**发送无阻塞但是接受有**)
假设队列已经满了，使用xQueueSendToBack() 给队列发送数据时，任务可以进入阻塞状态等待发送完成。
使用任务通知时，即使对方无法接收数据，发送方也无法阻塞等待，只能即刻返回错误。

通知状态和通知值

每个任务都有一个结构体：TCB(Task Control Block)，里面有2个成员：

- 一个是uint8_t类型，用来表示通知状态
- 一个是uint32_t类型，用来表示通知值

```
typedef struct tskTaskControlBlock
{
    .....
    /* configTASK_NOTIFICATION_ARRAY_ENTRIES = 1 */
    volatile uint32_t ulNotifiedValue[ configTASK_NOTIFICATION_ARRAY_ENTRIES ];
    volatile uint8_t ucNotifyState[ configTASK_NOTIFICATION_ARRAY_ENTRIES ];
    .....
} tskTCB;
```

通知状态有3种取值：

- taskNOT_WAITING_NOTIFICATION：任务没有在等待通知
- taskWAITING_NOTIFICATION：任务在等待通知
- taskNOTIFICATION_RECEIVED：任务接收到通知，也被称为pending(有数据了，待处理)

##define taskNOT_WAITING_NOTIFICATION	((uint8_t) 0) /* 也是初始
状态 */	
##define taskWAITING_NOTIFICATION	((uint8_t) 1)
##define taskNOTIFICATION_RECEIVED	((uint8_t) 2)

通知值可以有很多种类型：

- 计数值
- 位(类似事件组)
- 任意数值

任务通知使用

使用任务通知，可以实现轻量级的队列(长度为1)、邮箱(覆盖的队列)、计数型信号量、二进制信号量、事件组。

两类函数

任务通知有2套函数，简化版、专业版，列表如下：

- 简化版函数的使用比较简单，它实际上也是使用专业版函数实现的
- 专业版函数支持很多参数，可以实现很多功能

	简化版	专业版
发出通知	xTaskNotifyGive vTaskNotifyGiveFromISR	xTaskNotify xTaskNotifyFromISR
取出通知	ulTaskNotifyTake	xTaskNotifyWait

xTaskNotifyGive/ulTaskNotifyTake

在任务中使用xTaskNotifyGive函数，在ISR中使用vTaskNotifyGiveFromISR函数，都是直接给其他任务发送通知：

- 使得通知值加一
- 并使得通知状态变为"pending"，也就是taskNOTIFICATION_RECEIVED，表示有数据了、待处理

可以使用ulTaskNotifyTake函数来取出通知值：

- 如果通知值等于0，则阻塞(可以指定超时时间)
- 当通知值大于0时，任务从阻塞态进入就绪态
- 在ulTaskNotifyTake返回之前，还可以做些清理工作：
把通知值减一，或者把通知值清零
使用ulTaskNotifyTake函数可以实现轻量级的、高效的二进制信号量、计数型信号量。

函数原型：

```

1 BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
2
3 void vTaskNotifyGiveFromISR( TaskHandle_t xTaskHandle, BaseType_t
4 *pxHigherPriorityTaskWoken );
5 uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t
6 xTicksToWait );

```

参数说明：

xTaskNotifyGive函数的参数说明如下：

参数	说明
xTaskToNotify	任务句柄(创建任务时得到)，给哪个任务发通知
返回值	必定返回pdPASS

vTaskNotifyGiveFromISR函数的参数说明如下：

参数	说明
xTaskHandle	任务句柄(创建任务时得到)，给哪个任务发通知
pxHigherPriorityTaskWoken	被通知的任务，可能正处于阻塞状态。 此函数发出通知后，会把它从阻塞状态切换为就绪态。 如果被唤醒的任务的优先级，高于当前任务的优先级， 则"pxHigherPriorityTaskWoken"被设置为pdTRUE， 这表示在中断返回之前要进行任务切换。

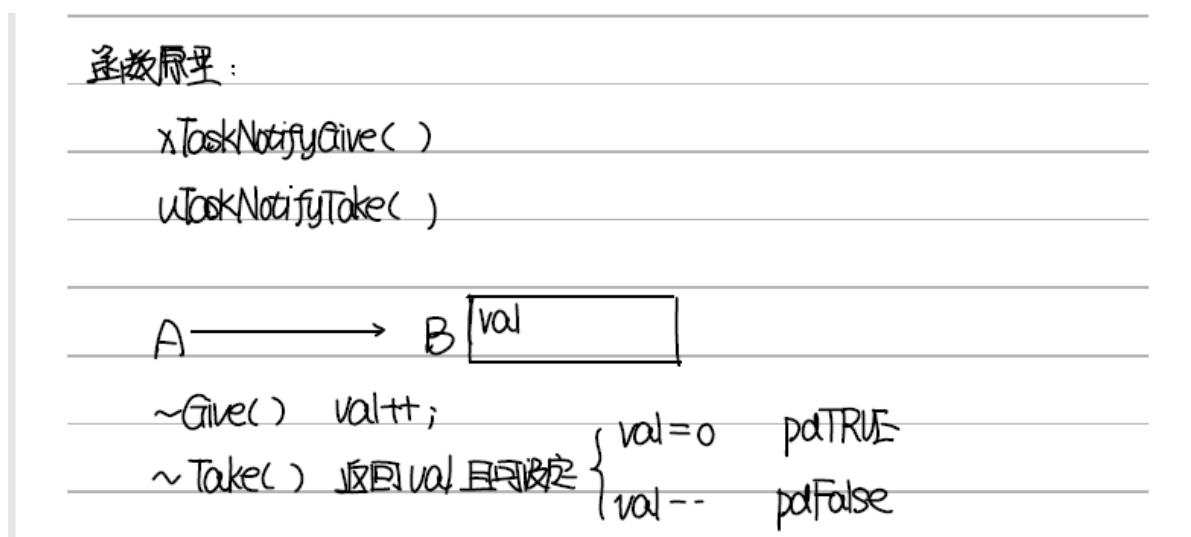
ulTaskNotifyTake函数的参数说明如下：

参数	说明
xClearCountOnExit	函数返回前是否清零： pdTRUE：把通知值清零 pdFALSE：如果通知值大于0，则把通知值减一
xTicksToWait	任务进入阻塞态的超时时间，它在等待通知值大于0。 0：不等待，即刻返回； portMAX_DELAY：一直等待，直到通知值大于0； 其他值：Tick Count，可以用 pdMS_TO_TICKS() 把ms转换为Tick Count
返回值	函数返回之前，在清零或减一之前的通知值。 如果xTicksToWait非0，则返回值有2种情况： 1. 大于0：在超时前，通知值被增加了 2. 等于0：一直没有其他任务增加通知值，最后超时返回0

详细解释：

- 等待通知：
ulTaskNotifyTake 会使调用任务进入阻塞状态，直到收到通知或超时。
xClearCountOnExit 参数决定了在函数返回时是否清除任务通知值。
- 超时时间：
xTicksToWait 参数指定了等待通知的最大时间。如果在指定时间内没有收到通知，函数会返回 0。
使用 pdMS_TO_TICKS 宏可以将毫秒转换为系统节拍数。
- 返回值：
返回值为任务通知值，表示在调用 ulTaskNotifyTake 时任务通知值的数量。
如果在超时时间内没有收到通知，返回值为 0。

图解：



xTaskNotify/xTaskNotifyWait

xTaskNotify 函数功能更强大，可以使用不同参数实现各类功能，比如：

- 让接收任务的通知值加一：这时xTaskNotify() 等同于xTaskNotifyGive()
- 设置接收任务的通知值的某一位、某些位，这就是一个轻量级的、更高效的事件组
- 把一个新值写入接收任务的通知值：上一次的通知值被读走后，写入才成功。这就是轻量级的、长度为1的队列

- 用一个新值覆盖接收任务的通知值：无论上一次的通知值是否被读走，覆盖都成功。类似 xQueueOverwrite() 函数，这就是轻量级的邮箱。

xTaskNotify() 比xTaskNotifyGive() 更灵活、强大，使用上也就更复杂。
xTaskNotifyFromISR() 是它对应的ISR版本。

这两个函数用来发出任务通知，使用哪个函数来取出任务通知呢？

使用xTaskNotifyWait() 函数！它比ulTaskNotifyTake() 更复杂：

- 可以让任务等待(可以加上超时时间)，等到任务状态为"pending"(也就是有数据)
- 还可以在函数进入、退出时，清除通知值的指定位

函数原型：

```

1 BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify,
2   uint32_t ulValue, eNotifyAction eAction );
3
4 BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify,
5   uint32_t ulValue,
6   eNotifyAction eAction,
7   BaseType_t *pxHigherPriorityTaskWoken );
8
9 BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,
10   uint32_t ulBitsToClearOnExit,
11   uint32_t *pulNotificationValue,
12   TickType_t xTicksToWait );

```

参数列表：

xTaskNotify函数的参数

参数	说明
xTaskToNotify	任务句柄(创建任务时得到)，给哪个任务发通知
ulValue	怎么使用ulValue，由eAction参数决定
eAction	见下表
返回值	pdPASS: 成功，大部分调用都会成功 pdFAIL: 只有一种情况会失败，当eAction为eSetValueWithoutOverwrite，并且通知状态为"pending"(表示有新数据未读)，这时就会失败。

eNotifyAction参数说明：

取值	说明
eNoAction	仅仅是更新通知状态为"pending"，未使用ulValue。这个选项相当于轻量级的、更高效的二进制信号量。
eSetBits	通知值 = 原来的通知值 ulValue，按位或。相当于轻量级的、更高效的事件组。
eIncrement	通知值 = 原来的通知值 + 1，未使用ulValue。相当于轻量级的、更高效的二进制信号量、计数型信号量。相当于 xTaskNotifyGive() 函数。

取值	说明
eSetValueWithoutOverwrite	不覆盖。如果通知状态为"pending"(表示有数据未读), 则此次调用xTaskNotify不做任何事, 返回pdFAIL。如果通知状态不是"pending"(表示没有新数据), 则: 通知值 = ulValue。
eSetValueWithOverwrite	覆盖。无论如何, 不管通知状态是否为"pendng", 通知值 = ulValue。

xTaskNotifyFromISR函数跟xTaskNotify很类似, 就多了最后一个参数pxHigherPriorityTaskWoken。在很多ISR函数中, 这个参数的作用都是类似的, 使用场景如下:

- 被通知的任务, 可能正处于阻塞状态
- xTaskNotifyFromISR 函数发出通知后, 会把接收任务从阻塞状态切换为就绪态
- 如果被唤醒的任务的优先级, 高于当前任务的优先级, "*pxHigherPriorityTaskWoken"被设置为 pdTRUE, 这表示在中断返回之前要进行任务切换。

参数图解:

~~xTaskNotify() ;~~

~~xTaskNotifyWait();~~



~~xTaskNotify(xTaskToNotify, ulValue, eAction);~~

~~对 eAction:~~

~~eNoAction: 仅通知不修改 value (设置 state, 不设置 value)~~

~~eSetBits : val | ulValue 改变 val 某些位为 1~~

~~eIncrement 改变 value +1~~

~~eSetValueWithoutOverwrite() 不覆盖 第一次发未响应但还未取又发布第二次~~

~~eSetValueWithOverwrite() 覆盖 第二次值覆盖/不覆盖第一次值~~

xTaskNotifyWait函数列表：

参数	说明
ulBitsToClearOnEntry	<p>在xTaskNotifyWait入口处，要清除通知值的哪些位？ 通知状态不是"pending"的情况下，才会清除。 它的本意是：我想等待某些事件发生，所以先把"旧数据"的某些位清零。 能清零的话：通知值 = 通知值 & ~ (ulBitsToClearOnEntry)。 比如传入0x01，表示清除通知值的bit0； 传入0xffffffff即ULONG_MAX，表示清除所有位，即把值设置为0</p>
ulBitsToClearOnExit	<p>在xTaskNotifyWait出口处，如果不是因为超时推出，而是因为得到了数据而退出时： 通知值 = 通知值 & ~ (ulBitsToClearOnExit)。 在清除某些位之前，通知值先被赋给"pulNotificationValue"。 比如入0x03，表示清除通知值的bit0、bit1； 传入0xffffffff即ULONG_MAX，表示清除所有位，即把值设置为0</p>
pulNotificationValue	<p>用来取出通知值。 在函数退出时，使用ulBitsToClearOnExit清除之前，把通知值赋给"pulNotificationValue"。 如果不需要取出通知值，可以设为NULL。</p>
xTicksToWait	<p>任务进入阻塞态的超时时间，它在等待通知状态变为"pending"。 0：不等待，即刻返回； portMAX_DELAY：一直等待，直到通知状态变为"pending"； 其他值：Tick Count，可以用pdMS_TO_TICKS() 把ms转换为Tick Count</p>
返回值	<p>1. pdPASS：成功 这表示xTaskNotifyWait成功获得了通知： 可能是调用函数之前，通知状态就是"pending"； 也可能是在阻塞期间，通知状态变为了"pending"。 2. pdFAIL：没有得到通知。</p>

图解：

`xTaskNotifyWait(ulBitsToClearOnEntry, ulBitsToClearOnExit, *pulNotificationValue, xTicksToWait)`

ulBitsToClearOnEntry : 入口处清除掉某些位 val &= ~ (bits) 清除位

ulBitsToClearOnExit : 出口处清除某些位 val &= ~ (bits) 清除位

pulNotificationValue : 指向 val 出口清除前的 value

xTicksToWait : 等待

返回 pdPass 表示成功

任务通知实例

轻量级信号量

可以使用任务通知实现轻量级的信号量，初始值为0，最小值也为0，不能指定初始值和最大值。

对比图：

对比：	信号量	使用任务通知实现信号量
创建	SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount, UBaseType_t uxInitialCount);	无
Give	xSemaphoreGive(SemaphoreHandle_t xSemaphore);	BaseType_t xTaskNotifyGive(TaskHandle_t xTaskToNotify);
Take	xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xBlockTime);	uint32_t ulTaskNotifyTake(BaseType_t xClearCountOnExit, TickType_t xTicksToWait);

↓

返回值

pdPASS 成功 其他 失败

↓

返回值

成功：返回新值 失败：返回0

Demo1：

```
1 void Task1Function(void *param)
2 {
3     volatile int i = 0;
4     while (1)
5     {
6         for (i = 0; i < 10000; i++)
7             sum++;
8         for (i = 0; i < 10; i++) // give notify in a 10 loop:give 10 times
9         {
10            xTaskNotifyGive(xHandleTask2);
11            // xSemaphoreGive(xSemCalc); // give the semaphore(free)
12        }
13        vTaskDelete(NULL);
14    }
15 }
16
17 void Task2Function(void *param)
18 {
19     int i = 0;
20     while (1)
21     {
22         flagCalcEnd = 0; // prove the running state
23         vale = ulTaskNotifyTake(pdFALSE, portMAX_DELAY); // blocked to wait
24         // xSemaphoreTake(xSemCalc, portMAX_DELAY); // get the semaphore if not: blocked(always)
25         flagCalcEnd = 1;
26         printf("sum = %d value = %d\r\n", sum, vale, i++);
27     }
28 }
```

Demo2:

main主逻辑:

```
int main( void )
{
    prvSetupHardware();

    /* 创建1个任务用于发送任务通知
     * 优先级为2
     */
    xTaskCreate( vSenderTask, "Sender", 1000, NULL, 2, NULL );

    /* 创建1个任务用于接收任务通知
     * 优先级为1
     */
    xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, &xRecvTask );

    /* 启动调度器 */
    vTaskStartScheduler();

    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}

static void vSenderTask( void *pvParameters )
{
    int i;
    int cnt_tx = 0;
    int cnt_ok = 0;
    int cnt_err = 0;
    char c;
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 20UL );

    /* 无限循环 */
    for( ;; )                                A
    {
        for ( i = 0; i < 3; i++)
        {
            /* 放入数据 */
            c = 'a'+cnt_tx;
            txbuf_put(c);
            cnt_tx++;

            /* 发出任务通知 */
            if (xTaskNotifyGive(xRecvTask) == pdPASS)
                printf("xTaskNotifyGive %d time: OK, val :%c\r\n", cnt_ok++, c);
            else
                printf("xTaskNotifyGive %d time: ERR\r\n", cnt_err++);
        }
        vTaskDelay(xTicksToWait); B
    }
}

static void vReceiverTask( void *pvParameters )
{
    int cnt_ok = 0;
    uint8_t c;
    int notify_val;

    /* 无限循环 */
    for( ;; )                                C E
    {
        /* 得到了任务通知，让通知值清零 */
        notify_val = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        /* 打印这几个字符 */
        printf("ulTaskNotifyTake OK: %d, data: ", cnt_ok++);
        /* 一次性把所有数据取出来 */
        while( (notify_val--) )
        {
            txbuf_get(&c);
            printf("%c", c);
        }
        printf("\r\n");
    }
}
```

发送任务、接收任务的代码和执行流程如下:

- A: 发送任务优先级最高, 先执行。连续存入3个字符、发出3次任务通知: 通知值累加为3
- B: 发送任务阻塞, 让接收任务能执行
- C: 接收任务读到通知值为3, 并把通知值清零
- D: 把3个字符依次读出、打印
- E: 再次读取任务通知, 阻塞

结果如下：

运行结果如下图所示：

```
UART #1
xTaskNotifyGive 0 time: OK, val :a
xTaskNotifyGive 1 time: OK, val :b
xTaskNotifyGive 2 time: OK, val :c
ulTaskNotifyTake OK: 0, data: abc
xTaskNotifyGive 3 time: OK, val :d
xTaskNotifyGive 4 time: OK, val :e
xTaskNotifyGive 5 time: OK, val :f
ulTaskNotifyTake OK: 1, data: def
xTaskNotifyGive 6 time: OK, val :g
xTaskNotifyGive 7 time: OK, val :h
xTaskNotifyGive 8 time: OK, val :i
ulTaskNotifyTake OK: 2, data: ghi
```

本程序使用xTaskNotifyGive/ulTaskNotifyTake 实现了轻量级的计数型信号量，代码更简单：

- 无需创建信号量
- 消耗内存更少
- 效率更高

信号量是个公开的资源，任何任务、ISR都可以使用它：可以释放、获取信号量。

而本节程序中，发送任务只能给指定的任务发送通知，目标明确；接收任务只能从自己的通知值中得到数据，来源明确。

轻量级队列

可以实现一个只能存放一个32位数据大小为1的队列。

可以设置是否覆盖val或者不覆盖val

send() < 覆盖 覆盖 val
 不覆盖 发送失败
 ⇒ 用 eAction 配置

函数对比：

队列		使用任务通知实现队列
创建	QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);	无
发送	BaseType_t xQueueSend(QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait);	BaseType_t xTaskNotify(TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction); 发 ⇒ 不会发送阻塞. 及时反馈
接收	BaseType_t xQueueReceive(QueueHandle_t xQueue, void * const pvBuffer, TickType_t xTicksToWait);	BaseType_t xTaskNotifyWait(uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t *pulNotificationValue, TickType_t xTicksToWait); 收 ⇒ 可以阻塞

Demo1:

```
1 void Task1Function(void *param)
2 {
3     volatile int i = 0;
4     while (1)
5     {
6         for (i = 0; i < 10000; i++)
7             sum++;
8         for (i = 0; i < 10; i++)
9         {
10             /* code */
11             //xQueueSend(xQueueCalcHandle, &sum, portMAX_DELAY); // give Data in 10 Times
12             xTaskNotify(xHandleTask2,sum,eSetValueWithOverwrite);
13             sum++;
14             vTaskDelay(1);
15         }
16         vTaskDelete(NULL);
17     }
18 }
19
20 void Task2Function(void *param)
21 {
22     uint32_t val;
23     int i = 0;
24     while (1)
25     {
26         flagCalcEnd = 0; // 模拟器中检测时间
27         //xQueueReceive(xQueueCalcHandle, &val, portMAX_DELAY); // 没有读到数据会一直阻塞，不占用CPU资源
28         xTaskNotifyWait(0,0xFFFF,&val,portMAX_DELAY);
29         flagCalcEnd = 1; // 模拟器中检测时间
30         printf("sum = %d i = %d\r\n", val, i++);
31     }
32 }
```

Demo2:

本节程序使用任务通知来传输任意数据，它创建2个任务：

- 发送任务：把数据通过xTaskNotify() 发送给其他任务
- 接收任务：使用xTaskNotifyWait 取出通知值，这表示字符，并打印出来

主逻辑：

```
int main( void )
{
    prvSetupHardware();

    /* 创建1个任务用于发送任务通知
     * 优先级为2
     */
    xTaskCreate( vSenderTask, "Sender", 1000, NULL, 2, NULL );

    /* 创建1个任务用于接收任务通知
     * 优先级为1
     */
    xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, &xRecvTask );

    /* 启动调度器 */
    vTaskStartScheduler();

    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}
```

```

111 static void vSenderTask( void *pvParameters )
112 {
113     int i;
114     int cnt_tx = 0;
115     int cnt_ok = 0;
116     int cnt_err = 0;
117     char c;
118     const TickType_t xTicksToWait = pdMS_TO_TICKS( 20UL );
119
120     /* 无限循环 */
121     for(;;)
122     {
123         for (i = 0; i < 3; i++)
124         {
125             /* 放入数据 */
126             c = 'a'+cnt_tx;
127             cnt_tx++;
128
129             /* 发出任务通知
130              * 发给谁? xRecvTask
131              * 发什么? c
132              * 能否覆盖? 不能, eSetValueWithoutOverwrite
133              */
134             if (xTaskNotify(xRecvTask,
135                             (uint32_t)c, /* 发送的数据 */
136                             eSetValueWithoutOverwrite) == pdPASS)
137                 printf("xTaskNotify %d time: OK, val :%c\r\n", cnt_ok++, c);
138             else
139                 printf("xTaskNotify %d time: ERR, val :%c\r\n", cnt_err++, c);
140
141         }
142         vTaskDelay(xTicksToWait); B
143     }
144 }

```

执行流程:

- A: 发送任务优先级最高, 先执行。连续给对方任务发送3个字符, 只成功了1次
- B: 发送任务阻塞, 让接收任务能执行
- C: 接收任务读取通知值
- D: 把读到的通知值作为字符打印出来
- E: 再次读取任务通知, 阻塞

结果:

```

UART #1
xTaskNotify 0 time: OK, val :a
xTaskNotify 0 time: ERR, val :b
xTaskNotify 1 time: ERR, val :c
xTaskNotifyWait OK: 0, data: a
xTaskNotify 1 time: OK, val :d
xTaskNotify 2 time: ERR, val :e
xTaskNotify 3 time: ERR, val :f
xTaskNotifyWait OK: 1, data: d

```

本程序使用xTaskNotify/xTaskNotifyWait 实现了轻量级的队列(该队列长度只有1), 代码更简单:

- 无需创建队列
- 消耗内存更少
- 效率更高

队列是个公开的资源, 任何任务、ISR都可以使用它: 可以存入数据、取出数据。

而本节程序中, 发送任务只能给指定的任务发送通知, 目标明确; 接收任务只能从自己的通知值中得到数据, 来源明确。

注意: 任务通知值只有一个, 数据可能丢失, 设计程序时要考虑这点。

轻量级事件组

事件中含有value可设置每一位对应每个事件：



用任务通知实现：设置 value 的一位 eSetBit 设置某些位

$B \rightarrow A$ [value, state]

任务B调用 xTaskNotify, state=1后，就会唤醒A，不能指定等待什么。

(任何任务进入A设置 value 会唤醒A)

函数对比：

	事件组	使用任务通知实现事件组
创建	EventGroupHandle_t xEventGroupCreate(void)	无
设置事件	EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet);	BaseType_t xTaskNotify(TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction);
等待事件	EventBits_t xEventGroupWaitBits(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait);	BaseType_t xTaskNotifyWait(uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t *pulNotificationValue, TickType_t xTicksToWait);

Demo:

Task1/Task2:

```
1 void Task1Function(void *param)
2 {
3     volatile int i = 0;
4     while (1)
5     {
6         for (i = 0; i < 100000; i++)
7             sum = sum + i;
8         xQueueSend(xQueueCalcHandle, &sum, portMAX_DELAY);
9         // set event 0(bit 0)
10        // xEventGroupSetBits(xEvent, 1 << 0);
11        xTaskNotify(xHandleTask3, (1 << 0), eSetBits);
12        printf("Task1 Set Bit 0\r\n");
13        vTaskDelete(NULL);
14    }
15 }
16
17 void Task2Function(void *param)
18 {
19     volatile int i = 0;
20     while (1)
21     {
22         for (i = 0; i < 1000000; i++)
23             dec = dec - i;
24         xQueueSend(xQueueCalcHandle, &dec, portMAX_DELAY);
25         // set event 0(bit 0)
26         // xEventGroupSetBits(xEvent, 1 << 1);
27         xTaskNotify(xHandleTask3, (1 << 1), eSetBits);
28         printf("Task2 Set Bit 1\r\n");
29         vTaskDelete(NULL);
30     }
31 }
```

Task3:

```
1 void Task3Function(void *param)
2 {
3     int val1, val2;
4     uint32_t bits=0;
5     while (1)
6     {
7         // xEventGroupWaitBits(xEvent, (1 << 0) | (1 << 1), pdTRUE, pdTRUE, portMAX_DELAY);
8         xTaskNotifyWait(0, 0, &bits, portMAX_DELAY);
9         if ((bits&0x02)==0x02) //自己去判断因为何时有的
10         { // 0x02 == 0x0000 0011
11             vTaskDelay(200);
12             xQueueSend(xQueueCalcHandle, &val1, portMAX_DELAY);
13             printf("val1= %d\r\n", val1);
14             xQueueReceive(xQueueCalcHandle, &val2, portMAX_DELAY);
15             printf("val2= %d\r\n", val2);
16         }else{
17             vTaskDelay(20);
18             printf("No Get All Bits:Get only 0x%r\n",bits);
19         }
20     }
21 }
```

→ 抓

⇒ 注意(bits & 0x0B)

结果：

```
UART #1
Task1 Set Bit 0
No Get All Bits:Get only 0x1
Task2 Set Bit 1
val1= 704982704
val2= -1783293664
```

十一、软件定时器

软件定时器就是"闹钟"，你可以设置闹钟，

- 在30分钟后让你起床工作
- 每隔1小时让你例行检查机器运行情况

软件定时器也可以完成两类事情：

- 在"未来"某个时间点，运行函数
- 周期性地运行函数

日常生活中我们可以定无数个"闹钟"，这无数的"闹钟"要基于一个真实的闹钟。

在FreeRTOS里，我们也可以设置无数个"软件定时器"，它们都是基于系统滴答中断(Tick Interrupt)。

本章涉及如下内容：

- 软件定时器的特性
- Daemon Task
- 定时器命令队列
- 一次性定时器、周期性定时器的差别
- 怎么操作定时器：创建、启动、复位、修改周期

软件定时器特性

使用定时器跟使用手机闹钟是类似的：

- 指定时间：启动定时器和运行回调函数，两者的间隔被称为定时器的周期(period)。
- 指定类型，定时器有两种类型：
 - 一次性(One-shot timers)：这类定时器启动后，它的回调函数只会被调用一次；可以手工再次启动它，但是不会自动启动它。
 - 自动加载定时器(Auto-reload timers)：这类定时器启动后，时间到之后它会自动启动它；这使得回调函数被周期性地调用。
- 指定要做什么事，就是指定回调函数

实际的闹钟分为：有效、无效两类。软件定时器也是类似的，它由两种状态：

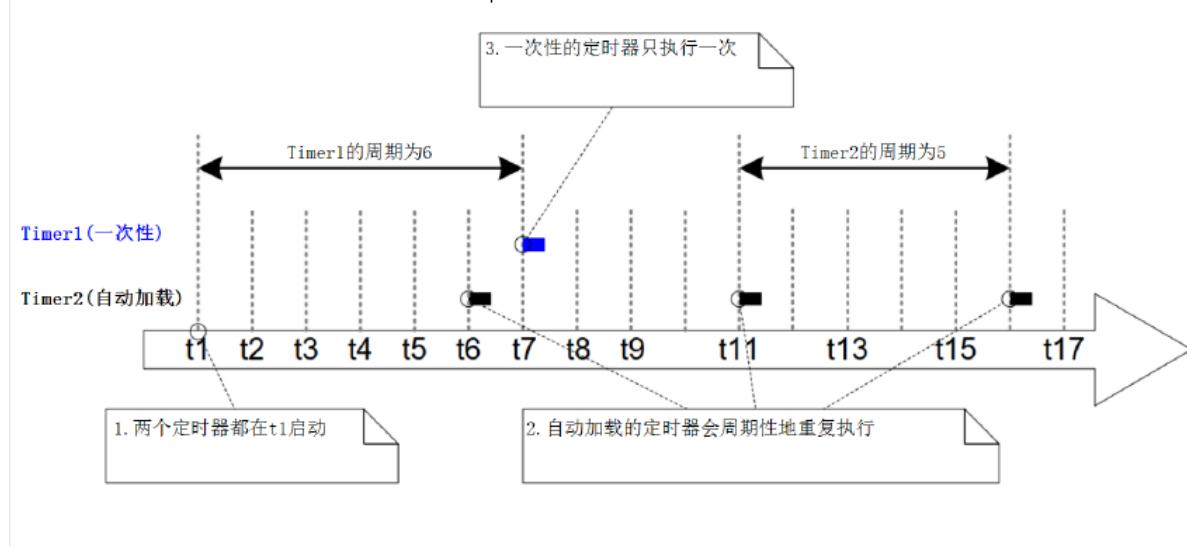
- 运行(Running、Active)：运行态的定时器，当指定时间到达之后，它的回调函数会被调用

- 冬眠(Dormant): 冬眠态的定时器还可以通过句柄来访问它，但是它不再运行，它的回调函数不会被调用

定时器运行情况示例如下：

- Timer1: 它是一次性的定时器，在t1启动，周期是6个Tick。经过6个tick后，在t7执行回调函数。它的回调函数只会被执行一次，然后该定时器进入冬眠状态。
- Timer2: 它是自动加载的定时器，在t1启动，周期是5个Tick。每经过5个tick它的回调函数都被执行，比如在t6、t11、t16都会执行。

图示：



软件定时器上下文

守护任务

FreeRTOS中有一个Tick中断，软件定时器基于Tick来运行。

在哪里执行定时器函数？第一印象就是在Tick中断里执行：

- 在Tick中断中判断定时器是否超时
- 如果超时了，调用它的回调函数

FreeRTOS是RTOS(实时操作系统)，它不允许在内核、在中断中执行不确定的代码：如果定时器函数很耗时，会影响整个系统。

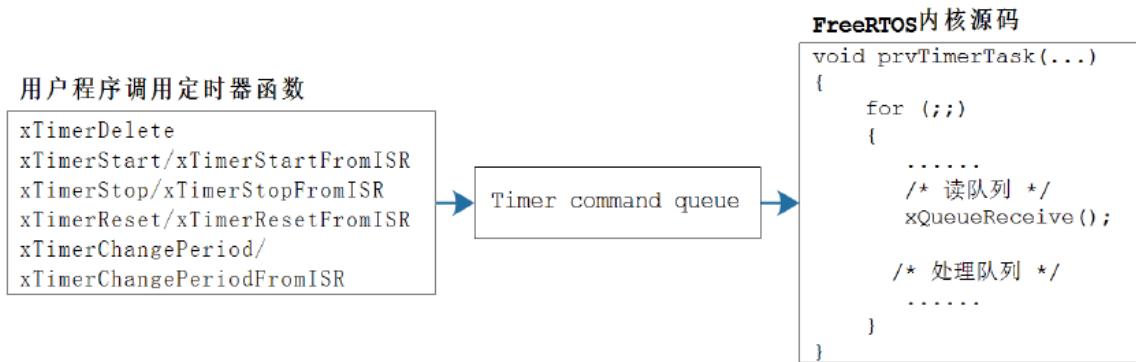
所以，FreeRTOS中，不在Tick中断中执行定时器函数。(Tick中断启用任务)

在哪里执行？在某个任务里执行，这个任务就是：RTOS Damemon Task，RTOS守护任务。

以前被称为"Timer server"，但是这个任务要做并不仅仅是定时器相关，所以改名为：RTOS Damemon Task。

当FreeRTOS的配置项configUSE_TIMERS 被设置为1时，在启动调度器时，会自动创建RTOS Damemon Task。

我们自己编写的任务函数要使用定时器时，是通过“定时器命令队列”(timer command queue)和守护任务交互，如下图：



守护任务的优先级为： configTIMER_TASK_PRIORITY；
定时器命令队列的长度为 configTIMER_QUEUE_LENGTH。

定时器触发后，回调函数被调用

优先级内核中进行
执行不确定的代码

→ { ① Tick中断中调用 (Linux) ↑
 ② FreeRTOS：某任务中执行 (实时)
 ↓
 (守护任务)

应注意：回调函数可以调用导致阻塞的函数，但建议应尽快完成，否则会阻碍

其它定时器函数

守护任务的调度

守护任务的调度，跟普通的任务并无差别。

当守护任务是当前优先级最高的就绪态任务时，它就可以运行。它的工作有两类：

- 处理命令：从命令队列里取出命令、处理
- 执行定时器的回调函数

能否及时处理定时器的命令、能否及时执行定时器的回调函数，严重依赖于守护任务的优先级。

示例：守护任务的优先性级较低

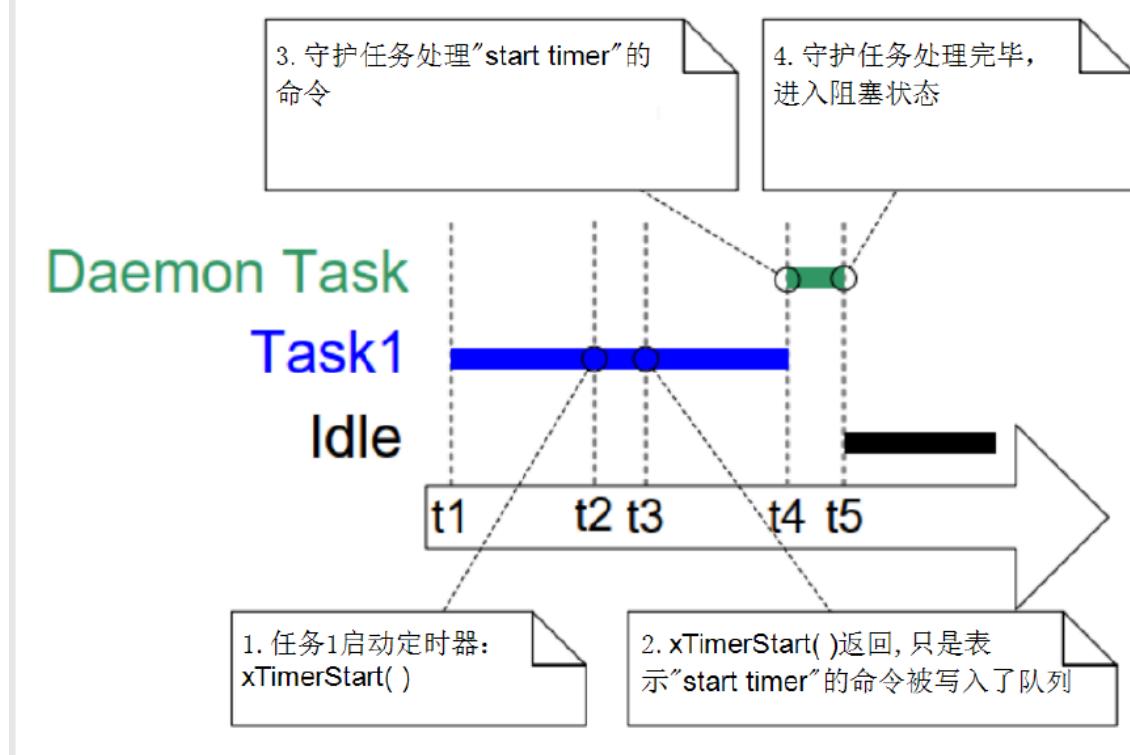
- t1：Task1处于运行态，守护任务处于阻塞态。
 守护任务在这两种情况下会退出阻塞态切换为就绪态：命令队列中有数据、某个定时器超时了。
 至于守护任务能否马上执行，取决于它的优先级。
- t2：Task1调用xTimerStart()
 要注意的是，xTimerStart() 只是把“start timer”的命令发给“定时器命令队列”，使得守护任务退出阻塞态。
 在本例中，Task1的优先级高于守护任务，所以守护任务无法抢占Task1。
- t3：Task1执行完xTimerStart()
 但是定时器的启动工作由守护任务来实现，所以xTimerStart() 返回并不表示定时器已经被启动了。
- t4：Task1由于某些原因进入阻塞态，现在轮到守护任务运行。
 守护任务从队列中取出“start timer”命令，启动定时器。

- t5：守护任务处理完队列中所有的命令，再次进入阻塞态。
Idle任务时优先级最高的就绪态任务，它执行。

注意：假设定时器在后续某个时刻tX超时了，超时时间是“tX-t2”，而非“tX-t4”，从xTimerStart() 函数被调用时算起。

定时器启动在t4，但是计时是从t2时刻。

图解：

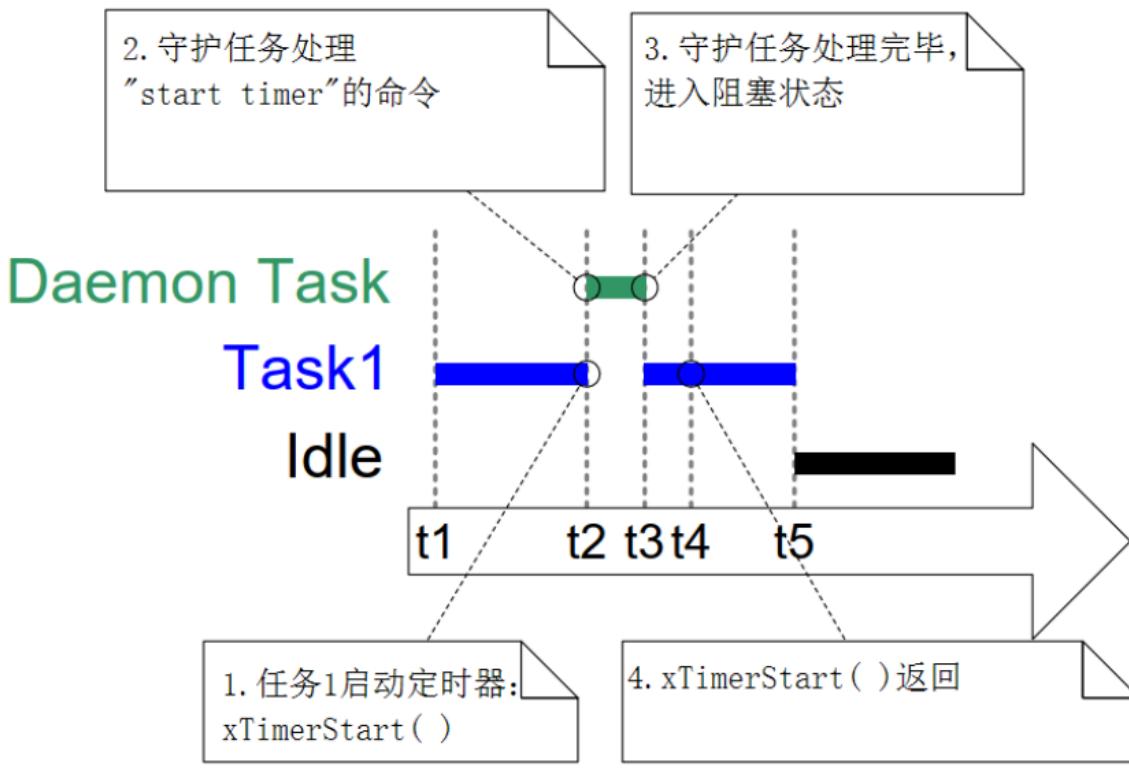


示例：守护任务的优先性级较高

- t1：Task1处于运行态，守护任务处于阻塞态。
守护任务在这两种情况下会退出阻塞态切换为就绪态：命令队列中有数据、某个定时器超时了。至于守护任务能否马上执行，取决于它的优先级。
- t2：Task1调用xTimerStart()
要注意的是，xTimerStart() 只是把“start timer”的命令发给“定时器命令队列”，使得守护任务退出阻塞态。
在本例中，守护任务的优先级高于Task1，所以守护任务抢占Task1，守护任务开始处理命令队列。
Task1在执行xTimerStart() 的过程中被抢占，这时它无法完成此函数。
- t3：守护任务处理完命令队列中所有的命令，再次进入阻塞态。
此时Task1是优先级最高的就绪态任务，它开始执行。
- t4：Task1之前被守护任务抢占，对xTimerStart() 的调用尚未返回
现在开始继续运行次函数、返回。
- t5：Task1由于某些原因进入阻塞态，进入阻塞态。
Idle任务时优先级最高的就绪态任务，它执行。

注意，定时器的超时时间是基于调用xTimerStart() 的时刻tX，而不是基于守护任务处理命令的时刻tY。
假设超时时间是10个Tick，超时时间是“tX+10”，而非“tY+10”。

启动定时器后会定时检查是否超时，超时调用回调函数。



综上所属：

守护任务：优先级应该设置的比较高，保证定时器函数及时执行
用于执行 Timer 事件

欲使用 timer，开启开关、定义相关回调，启动调度器时会自动建守护

任务

守护任务会同时管理多个定时器(创建的所有软件定时器，由守护任务管理)

回调函数

定时器的回调函数的原型如下：

```
1 | void ATimerCallback( TimerHandle_t xTimer );
```

定时器的回调函数是在守护任务中被调用的，守护任务不是专为某个定时器服务的，它还要处理其他定时器。

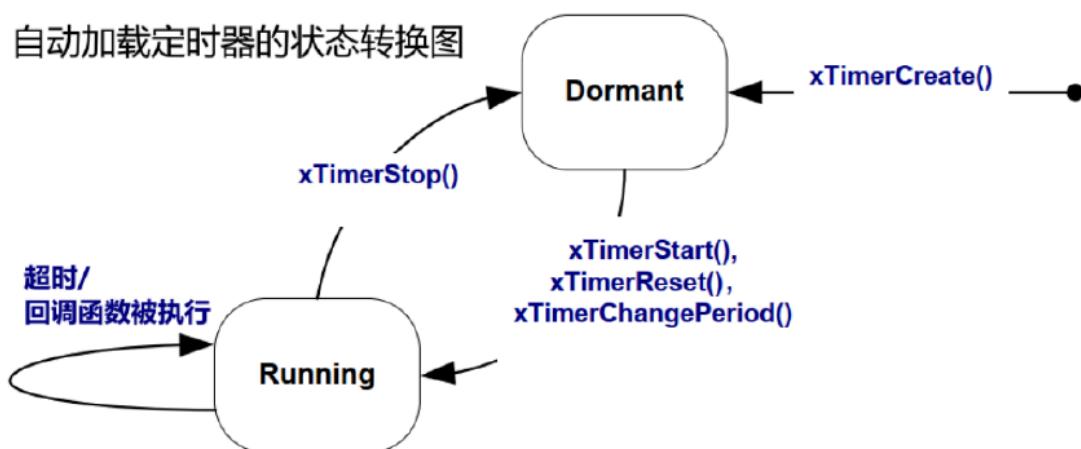
所以，定时器的回调函数不要影响其他人：

- 回调函数要尽快执行，不能进入阻塞状态(速度)
- 不要调用会导致阻塞的API函数，比如vTaskDelay() (**不阻塞**)
- 可以调用xQueueReceive() 之类的函数，但是超时时间要设为0：即刻返回，不可阻塞

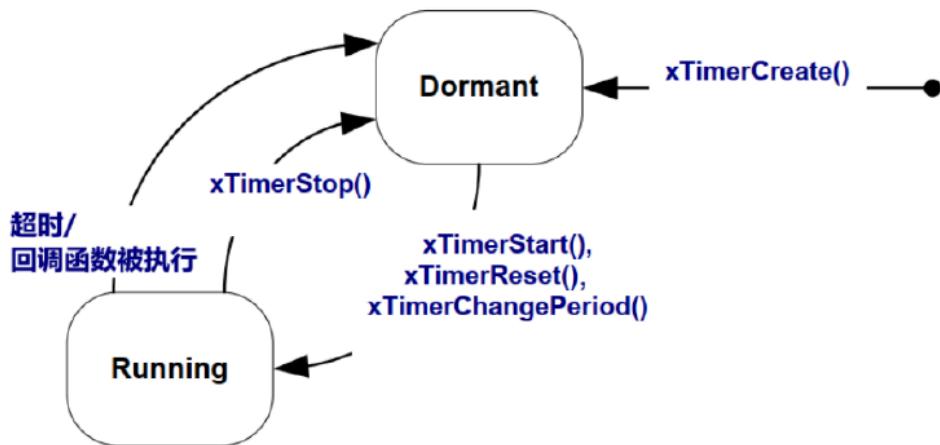
软件定时器函数

状态转化图：

自动加载定时器的状态转换图



一次性定时器的状态转换图



创建

要使用定时器，需要先创建它，得到它的句柄。

有两种方法创建定时器：动态分配内存、静态分配内存。

函数原型如下：

```
1  /* 使用动态分配内存的方法创建定时器
2  * pcTimerName: 定时器名字, 用处不大, 尽在调试时用到
3  * xTimerPeriodInTicks: 周期, 以Tick为单位
4  * uxAutoReload: 类型, pdTRUE表示自动加载, pdFALSE表示一次性
5  * pvTimerID: 回调函数可以使用此参数, 比如分辨是哪个定时器
6  * pxCallbackFunction: 回调函数
7  * 返回值: 成功则返回TimerHandle_t, 否则返回NULL
8  */
9  TimerHandle_t xTimerCreate( const char * const pcTimerName,
10  const TickType_t xTimerPeriodInTicks,
11  const BaseType_t uxAutoReload,
12  void * const pvTimerID,
13  TimerCallbackFunction_t pxCallbackFunction );
14
15  /* 使用静态分配内存的方法创建定时器
16  * pcTimerName: 定时器名字, 用处不大, 尽在调试时用到
17  * xTimerPeriodInTicks: 周期, 以Tick为单位
18  * uxAutoReload: 类型, pdTRUE表示自动加载, pdFALSE表示一次性
```

```

19 * pvTimerID: 回调函数可以使用此参数, 比如分辨是哪个定时器
20 * pxcallbackFunction: 回调函数
21 * pxTimerBuffer: 传入一个StaticTimer_t结构体, 将在上面构造定时器
22 * 返回值: 成功则返回TimerHandle_t, 否则返回NULL
23 */
24 TimerHandle_t xTimerCreateStatic(const char * const pcTimerName,
25 TickType_t xTimerPeriodInTicks,
26 UBaseType_t uxAutoReload,
27 void * pvTimerID,
28 TimerCallbackFunction_t pxCallbackFunction,
29 StaticTimer_t *pxTimerBuffer);

```

其中回调函数类型:

```

1 void ATimerCallback( TimerHandle_t xTimer );
2 typedef void (* TimerCallbackFunction_t)( TimerHandle_t xTimer );

```

删除

动态分配的定时器, 不再需要时可以删除掉以回收内存。

删除函数原型如下:

```

1 /* 删除定时器
2 * xTimer: 要删除哪个定时器
3 * xTicksToWait: 超时时间
4 * 返回值: pdFAIL表示"删除命令"在xTicksToWait个Tick内无法写入队列
5 * pdPASS表示成功
6 */
7  BaseType_t xTimerDelete( TimerHandle_t xTimer, TickType_t xTicksToWait );

```

定时器的很多API函数, 都是通过发送"命令"到命令队列, 由守护任务来实现。

如果队列满了, "命令"就无法即刻写入队列。我们可以指定一个超时时间**xTicksToWait**, 等待一会。

启动停止

启动定时器就是设置它的状态为运行态(Running、Active)。

停止定时器就是设置它的状态为冬眠(Dormant), 让它不能运行。

```

1 /* 启动定时器
2 * xTimer: 哪个定时器
3 * xTicksToWait: 超时时间
4 * 返回值: pdFAIL表示"启动命令"在xTicksToWait个Tick内无法写入队列
5 * pdPASS表示成功
6 */
7  BaseType_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
8
9 /* 启动定时器(ISR版本)
10 * xTimer: 哪个定时器
11 * pxHigherPriorityTaskWoken: 向队列发出命令使得守护任务被唤醒,
12 * 如果守护任务的优先级比当前任务的高,
13 * 则"*pxHigherPriorityTaskWoken = pdTRUE",
14 * 表示需要进行任务调度
15 * 返回值: pdFAIL表示"启动命令"无法写入队列
16 * pdPASS表示成功

```

```

17 */
18 BaseType_t xTimerStartFromISR( TimerHandle_t xTimer,
19 BaseType_t *pxHigherPriorityTaskWoken );
20
21 /* 停止定时器
22 * xTimer: 哪个定时器
23 * xTicksToWait: 超时时间
24 * 返回值: pdFAIL表示"停止命令"在xTicksToWait个Tick内无法写入队列
25 * pdPASS表示成功
26 */
27 BaseType_t xTimerStop( TimerHandle_t xTimer, TickType_t xTicksToWait );
28
29 /* 停止定时器(ISR版本)
30 * xTimer: 哪个定时器
31 * pxHigherPriorityTaskWoken: 向队列发出命令使得守护任务被唤醒,
32 * 如果守护任务的优先级比当前任务的高,
33 * 则"pxHigherPriorityTaskWoken = pdTRUE",
34 * 表示需要进行任务调度
35 * 返回值: pdFAIL表示"停止命令"无法写入队列
36 * pdPASS表示成功
37 */
38 BaseType_t xTimerStopFromISR( TimerHandle_t xTimer,
39 BaseType_t *pxHigherPriorityTaskWoken );

```

注意，这些函数的xTicksToWait 表示的是，把命令写入命令队列的超时时间。

命令队列可能已经满了，无法马上把命令写入队列里，可以等待一会。

xTicksToWait 不是定时器本身的超时时间，不是定时器本身的"周期"。

创建定时器时，设置了它的周期(period)。

xTimerStart() 函数是用来启动定时器。

假设调用xTimerStart() 的时刻是tX，定时器的周期是n，那么在tX+n 时刻定时器的回调函数被调用。

如果定时器已经被启动，但是它的函数尚未被执行，再次执行xTimerStart() 函数相当于执行

xTimerReset()，重新设定它的启动时间。

定时器复位

从定时器的状态转换图可以知道，使用xTimerReset() 函数可以让定时器的状态从冬眠态转换为运行态，相当于使用xTimerStart() 函数。

如果定时器已经处于运行态，使用xTimerReset() 函数就相当于重新确定超时时间。

假设调用xTimerReset() 的时刻是tX，定时器的周期是n，那么tX+n 就是重新确定的超时时间。

函数原型如下：

```

1 /* 复位定时器
2 * xTimer: 哪个定时器
3 * xTicksToWait: 超时时间
4 * 返回值: pdFAIL表示"复位命令"在xTicksToWait个Tick内无法写入队列
5 * pdPASS表示成功
6 */
7 BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
8
9 /* 复位定时器(ISR版本)
10 * xTimer: 哪个定时器
11 * pxHigherPriorityTaskWoken: 向队列发出命令使得守护任务被唤醒,
12 * 如果守护任务的优先级比当前任务的高,

```

```

13  /* 则"pxHigherPriorityTaskWoken = pdTRUE",
14  * 表示需要进行任务调度
15  * 返回值: pdFAIL表示"停止命令"无法写入队列
16  * pdPASS表示成功
17  */
18 BaseType_t xTimerResetFromISR( TimerHandle_t xTimer, BaseType_t
*pxHigherPriorityTaskWoken );

```

周期修改

从定时器的状态转换图可以知道，使用xTimerChangePeriod() 函数，处理能修改它的周期外，还可以让定时器的状态从冬眠态转换为运行态。

修改定时器的周期时，会使用新的周期重新计算它的超时时间。

假设调用xTimerChangePeriod() 函数的时间tX，新的周期是n，则tX+n 就是新的超时时间。

函数原型如下：

```

1  /* 修改定时器的周期
2  * xTimer: 哪个定时器
3  * xNewPeriod: 新周期
4  * xTicksToWait: 超时时间，命令写入队列的超时时间
5  * 返回值: pdFAIL表示"修改周期命令"在xTicksToWait个Tick内无法写入队列
6  * pdPASS表示成功
7  */
8 BaseType_t xTimerChangePeriod( TimerHandle_t xTimer,
9 TickType_t xNewPeriod,
10 TickType_t xTicksToWait );
11
12 /* 修改定时器的周期
13 * xTimer: 哪个定时器
14 * xNewPeriod: 新周期
15 * pxHigherPriorityTaskWoken: 向队列发出命令使得守护任务被唤醒，
16 * 如果守护任务的优先级比当前任务的高，
17 * 则"pxHigherPriorityTaskWoken = pdTRUE",
18 * 表示需要进行任务调度
19 * 返回值: pdFAIL表示"修改周期命令"在xTicksToWait个Tick内无法写入队列
20 * pdPASS表示成功
21 */
22 BaseType_t xTimerChangePeriodFromISR( TimerHandle_t xTimer,
23 TickType_t xNewPeriod,
24 BaseType_t *pxHigherPriorityTaskWoken );

```

定时器ID

定时器的结构体如下，里面有一项pvTimerID，它就是定时器ID：

```

typedef struct tmrTimerControl
{
    const char * pcTimerName;
    ListItem_t xTimerListItem;
    TickType_t xTimerPeriodInTicks;
    void * pvTimerID;
    TimerCallbackFunction_t pxCallbackFunction;
    #if ( configUSE_TRACE_FACILITY == 1 )
        UBaseType_t uxTimerNumber;
    #endif
    uint8_t ucStatus;
} xTIMER;

```

怎么使用定时器ID，完全由程序来决定：

- 可以用来标记定时器，表示自己是什么定时器
- 可以用来保存参数，给回调函数使用

它的初始值在创建定时器时由xTimerCreate() 这类函数传入，后续可以使用这些函数来操作：

- 更新ID：使用vTimerSetTimerID() 函数
- 查询ID：查询pvTimerGetTimerID() 函数

注意：这两个函数不涉及命令队列，它们是直接操作定时器结构体。

函数原型如下：

```

1  /* 获得定时器的ID
2  * xTimer: 哪个定时器
3  * 返回值: 定时器的ID
4  */
5  void *pvTimerGetTimerID( TimerHandle_t xTimer );
6  /* 设置定时器的ID
7  * xTimer: 哪个定时器
8  * pvNewID: 新ID
9  * 返回值: 无
10 */
11 void vTimerSetTimerID( TimerHandle_t xTimer, void *pvNewID );

```

软件定时器使用

常规使用

使用软件定时器的准备操作：

```

/* 1. 工程中 */
添加 timer.c

/* 2. 配置文件FreeRTOSConfig.h中 */
##define configUSE_TIMERS           1 /* 使能定时器 */
##define configTIMER_TASK_PRIORITY   31 /* 守护任务的优先级，尽可能高一些 */
##define configTIMER_QUEUE_LENGTH    5 /* 命令队列长度 */
##define configTIMER_TASK_STACK_DEPTH 32 /* 守护任务的栈大小 */

/* 3. 源码中 */
##include "timers.h"

```

创建定时器：

main逻辑：

```
static volatile uint8_t flagONESHOTimerRun = 0;
static volatile uint8_t flagAutoLoadTimerRun = 0;

static void vONESHOTimerFunc( TimerHandle_t xTimer );
static void vAutoLoadTimerFunc( TimerHandle_t xTimer );

/*-----*/
##define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS( 10 )
##define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 20 )

int main( void )
{
    TimerHandle_t xOneShotTimer;
    TimerHandle_t xAutoReloadTimer;

    prvSetupHardware();

    xOneShotTimer = xTimerCreate(
        "OneShot",                      /* 名字，不重要 */
        mainONE_SHOT_TIMER_PERIOD,      /* 周期 */
        pdFALSE,                        /* 一次性 */
        0,                             /* ID */
        vONESHOTimerFunc               /* 回调函数 */
    );

    xAutoReloadTimer = xTimerCreate(
        "AutoReload",                  /* 名字，不重要 */
        mainAUTO_RELOAD_TIMER_PERIOD,  /* 周期 */
        pdTRUE,                         /* 自动加载 */
        0,                             /* ID */
        vAutoLoadTimerFunc             /* 回调函数 */
    );

    if (xOneShotTimer && xAutoReloadTimer)
    {
        /* 启动定时器 */
        xTimerStart(xOneShotTimer, 0);
        xTimerStart(xAutoReloadTimer, 0);

        /* 启动调度器 */
        vTaskStartScheduler();
    }

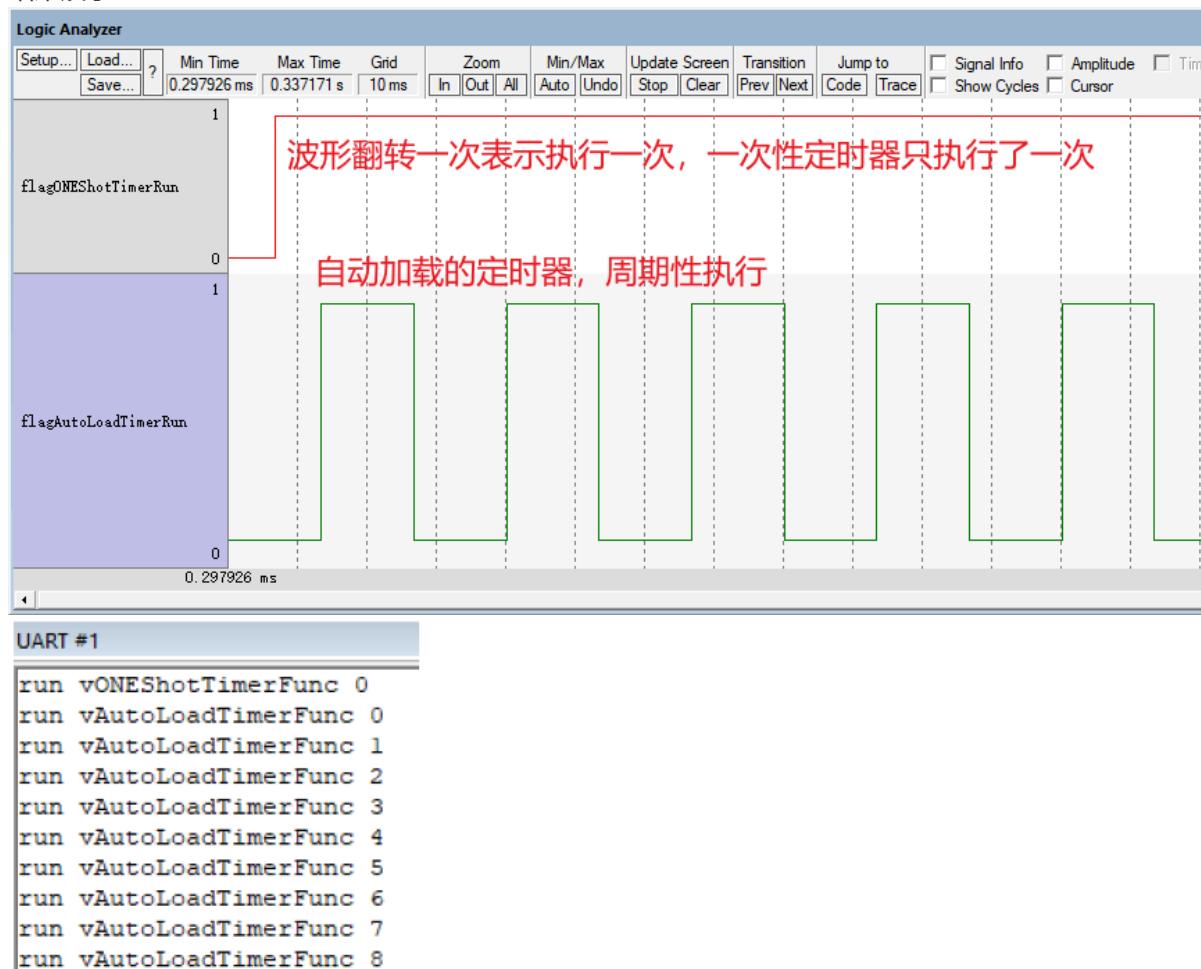
    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
    return 0;
}
```

回调函数：

```
static void vOneShotTimerFunc( TimerHandle_t xTimer )
{
    static int cnt = 0;
    flagOneShotTimerRun = !flagOneShotTimerRun;
    printf("run vOneShotTimerFunc %d\r\n", cnt++);
}

static void vAutoLoadTimerFunc( TimerHandle_t xTimer )
{
    static int cnt = 0;
    flagAutoLoadTimerRun = !flagAutoLoadTimerRun;
    printf("run vAutoLoadTimerFunc %d\r\n", cnt++);
}
```

结果展示：



软件定时器消抖

在嵌入式开发中，我们使用机械开关时经常碰到抖动问题：引脚电平在短时间内反复变化。怎么读到确定的按键状态？

- 连续读很多次，直到数值稳定：浪费CPU资源
- 使用定时器：要结合中断来使用

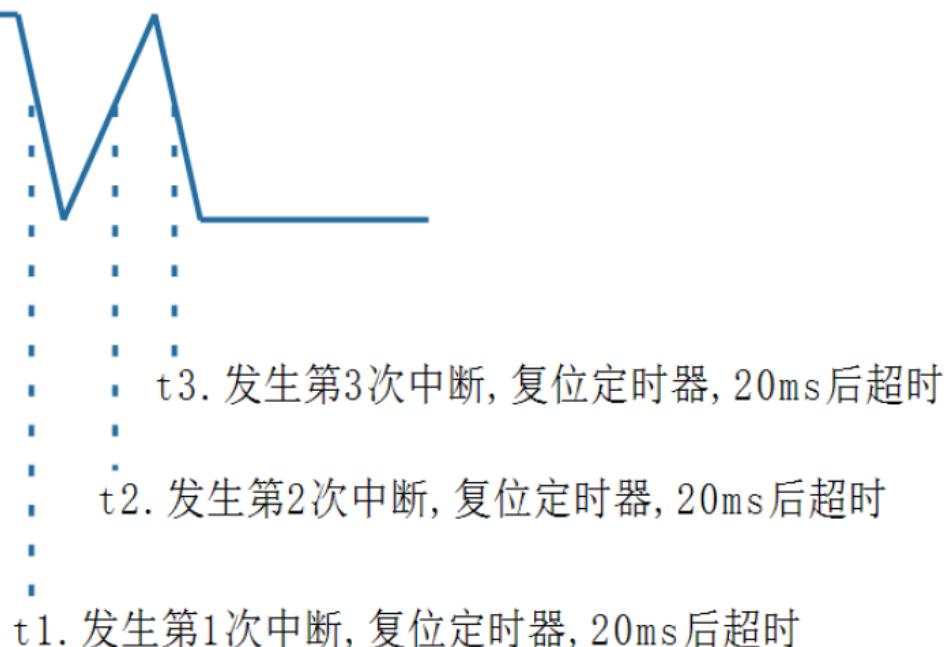
对于第2种方法，处理方法如下图所示，按下按键后：

- 在t1产生中断，这时不马上确定按键，而是复位定时器，假设周期为20ms，超时时间为“t1+20ms”
- 由于抖动，在t2再次产生中断，再次复位定时器，超时时间为“t2+20ms”

- 由于抖动，在t3再次产生中断，再次复位定时器，超时时间变为“t3+20ms”
- 在“t3+20ms”处，按键已经稳定，读取按键值

图解：

按键引脚电平



main函数中创建了一个一次性的定时器，从来处理抖动；

创建了一个任务，用来模拟产生抖动。

代码如下：

```

1 static void vKeyFilteringTimerFunc( TimerHandle_t xTimer ); //按键消抖
2 void vEmulateKeyTask( void *pvParameters ); //按键抖动模拟
3 static TimerHandle_t xKeyFilteringTimer; //定时器句柄
4
5 #define KEY_FILTERING_PERIOD pdMS_TO_TICKS( 20 ) //延迟20ms
6
7 int main( void ){
8     prvSetupHardware();
9     xKeyFilteringTimer = xTimerCreate(
10         "KeyFiltering", /* 名字，不重要 */
11         KEY_FILTERING_PERIOD, /* 周期 */
12         pdFALSE, /* 一次性 */
13         0, /* ID */
14         vKeyFilteringTimerFunc /* 回调函数 */
15     );
16     /* 在这个任务中多次调用xTimerReset来模拟按键抖动 */
17     xTaskCreate( vEmulateKeyTask, "EmulateKey", 1000, NULL, 1, NULL );
18     /* 启动调度器 */
19     vTaskStartScheduler();
20     /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
21     return 0;
22 }
```

模拟产生按键：

```
1 void vEmulateKeyTask( void *pvParameters )
```

```
2 {
3     int cnt = 0;
4     const TickType_t xDelayTicks = pdMS_TO_TICKS( 200UL );
5     for( ;; )
6     {
7         /* 模拟按键抖动，多次调用xTimerReset */
8         xTimerReset(xKeyFilteringTimer, 0); cnt++;
9         xTimerReset(xKeyFilteringTimer, 0); cnt++;
10        xTimerReset(xKeyFilteringTimer, 0); cnt++;
11        printf("Key jitters %d\r\n", cnt);
12        vTaskDelay(xDelayTicks);
13    }
14 }
```

回调函数：表示值稳定，确实摁下

```
1 static void vKeyFilteringTimerFunc( TimerHandle_t xTimer )
2 {
3     static int cnt = 0;
4     printf("vKeyFilteringTimerFunc %d\r\n", cnt++);
5 }
```

结果：

```
UART #1
-----
Key jitters 3
vKeyFilteringTimerFunc 0
Key jitters 6
vKeyFilteringTimerFunc 1
Key jitters 9
vKeyFilteringTimerFunc 2
Key jitters 12
vKeyFilteringTimerFunc 3
Key jitters 15
vKeyFilteringTimerFunc 4
Key jitters 18
vKeyFilteringTimerFunc 5
Key jitters 21
vKeyFilteringTimerFunc 6
Key jitters 24
vKeyFilteringTimerFunc 7
Key jitters 27
vKeyFilteringTimerFunc 8
```

实际上常常配合EXIT中断使用：

中断配置：

```
1 void KeyInit(void) // 配置GPIO引脚
2 {
3     GPIO_InitTypeDef GPIO_InitStructure; // 定义结构体
4     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // 使能时钟
5     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0; // 选择IO口 PA0
6     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; // 设置成上拉输入
7     GPIO_Init(GPIOA, &GPIO_InitStructure); // 使用结构体信息进行初始化IO口
8 }
9
10 // AFIO封装在GPIO中，主要有两个功能：中断引脚的选择以及艾用引脚映射
11 void KeyIntInit(void)
12 {
13     EXTI_InitTypeDef EXTI_InitStructure; // 定义EXTI初始化结构体 配置外部中断
14     NVIC_InitTypeDef NVIC_InitStructure; // 定义NVIC结构体 进行中断配置
15
16     RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE); /* 使能AFIO艾用时钟 */
17
18     GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0); /* 将GPIO口与中断线映射起来 */ // 选择中断引脚GPIOA的Pin0
19     /*
20     typedef struct
21     {
22         u32 EXTI_Line;
23         EXTIMode_TypeDef EXTI_Mode;
24         EXTITrigger_TypeDef EXTI_Trigger;
25         FunctionalState EXTI_LineCmd;
26     }EXTI_InitTypeDef;
27     */
28
29     EXTI_InitStructure.EXTI_Line = EXTI_Line0; // 中断线
30     EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; // 中断模式
31     EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising_Falling; // 双边沿触发
32     EXTI_InitStructure.EXTI_LineCmd = ENABLE; // 使能
33
34     EXTI_Init(&EXTI_InitStructure); // 初始化
35     /*
36     typedef struct
37     {
38         u8 NVIC_IrqChannel;
39         u8 NVIC_IrqChannelPreemptionPriority;
40         u8 NVIC_IrqChannelSubPriority;
41         FunctionalState NVIC_IrqChannelCmd;
42     } NVIC_InitTypeDef;
43     */
44     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); // 中断分组：芯片统一：执行一次
45     NVIC_InitStructure.NVIC_IrqChannel = EXTI0_IrqChannel; // 使能外部中断所在的通道
46     NVIC_InitStructure.NVIC_IrqChannelPreemptionPriority = 0; // 抢占优先级
47     NVIC_InitStructure.NVIC_IrqChannelSubPriority = 0; // 了优先级
48     NVIC_InitStructure.NVIC_IrqChannelCmd = ENABLE; // 使能外部中断通道
49     NVIC_Init(&NVIC_InitStructure); // 初始化
50 }
```

中断处理中：

```
1 void EXTI0_IRQHandler(void)
2 {
3     static int cnt = 0;
4     if (EXTI_GetITStatus(EXTI_Line0) != RESET)
5     {
6         printf("EXTI0_IRQHandler cnt = %d\r\n", cnt++);
7         /* 使用定时器消除抖动 */
8         xTimerReset(xMyTimerHandle, 0); /* Tcur + 2000 */ // 复位定时器 最终执行代码在定时器中，且完成延时后执行*/
9
10     EXTI_ClearITPendingBit(EXTI_Line0); // 清除中断
11 }
12 }
```

回调函数：

```
1 void MyTimerCallbackFunction(TimerHandle_t xTimer)
2 {
3     static int cnt = 0;
4     flagTimer = !flagTimer;
5     printf("Get GPIO Key cnt = %d\r\n", cnt++);
6 }
```

十二、中断管理

在RTOS中，需要应对各类事件。这些事件很多时候是通过硬件中断产生，怎么处理中断呢？

假设当前系统正在运行Task1时，用户按下了按键，触发了按键中断。这个中断的处理流程如下：

- CPU跳到固定地址去执行代码，这个固定地址通常被称为中断向量，这个跳转时硬件实现的
- 执行代码做什么？
 保存现场：Task1被打断，需要先保存Task1的运行环境，比如各类寄存器的值
 分辨中断：调用处理函数(这个函数就被称为ISR，interrupt service routine)
 恢复现场：继续运行Task1，或者运行其他优先级更高的任务

注意到，**ISR是在内核中被调用的**，ISR执行过程中，用户的任务无法执行。

ISR要尽量快，否则：

- 其他低优先级的中断无法被处理：实时性无法保证
- 用户任务无法被执行：系统显得很卡顿

如果这个硬件中断的处理，就是非常耗费时间呢？

对于这类中断的处理就要分为2部分：

- ISR：尽快做些清理、记录工作，然后触发某个任务
- 任务：更复杂的事情放在任务中处理
 所以，需要ISR和任务之间进行通信

要在FreeRTOS中熟练使用中断，有几个原则要先说明：

- FreeRTOS把任务认为是硬件无关的，任务的优先级由程序员决定，任务何时运行由调度器决定
- ISR虽然也是使用软件实现的，但是它被认为是硬件特性的一部分，因为它跟硬件密切相关
 - **何时执行？由硬件决定**
 - **哪个ISR被执行？由硬件决定**
- ISR的优先级高于任务：即使是优先级最低的中断，它的优先级也高于任务。任务只有在没有中断的情况下，才能执行。

本章涉及如下内容：

- FreeRTOS的哪些API函数能在ISR中使用

- 怎么把中断的处理分为两部分：ISR、任务
- ISR和任务之间的通信

两套API函数

原因

在任务函数中，我们可以调用各类API函数，比如队列操作函数：xQueueSendToBack。
但是在ISR中使用这个函数会导致问题，应该使用另一个函数：xQueueSendToBackFromISR，它的函数名含有后缀"FromISR"，表示"从ISR中给队列发送数据"。

FreeRTOS中很多API函数都有两套：一套在任务中使用，另一套在ISR中使用。
后者的函数名含有"FromISR"后缀。

为什么要引入两套API函数？

- 很多API函数会导致任务进入阻塞状态：
 - 运行这个函数的任务进入阻塞状态
 - 比如写队列时，如果队列已满，可以进入阻塞状态等待一会
- ISR调用API函数时，ISR不是"任务"，ISR不能进入阻塞状态(**中断执行尽可能快**)
- 在任务中、在ISR中，这些函数的功能是有差别的

为什么不使用同一套函数，比如在函数里面分辨当前调用者是任务还是ISR呢？

示例代码如下：

```
 BaseType_t xQueueSend( ... )
{
    if (is_in_isr())
    {
        /* 把数据放入队列 */

        /* 不管是否成功都直接返回 */
    }
    else /* 在任务中 */
    {
        /* 把数据放入队列 */
        /* 不成功就等待一会再重试 */
    }
}
```

FreeRTOS使用两套函数，而不是使用一套函数，是因为有如下好处：

- 使用同一套函数的话，需要增加额外的判断代码、增加额外的分支，是的函数更长、更复杂、难以测试。
- 在任务、ISR中调用时，需要的参数不一样，比如：
 - 在任务中调用：需要指定超时时间，表示如果不成功就阻塞一会
 - 在ISR中调用：不需要指定超时时间，无论是否成功都要即刻返回
 - 如果强行把两套函数揉在一起，会导致参数臃肿、无效
- 移植FreeRTOS时，还需要提供监测上下文的函数，比如is_in_isr()
- 有些处理器架构没有办法轻易分辨当前是处于任务中，还是处于ISR中，就需要额外添加更多、更复杂的代码

使用两套函数可以让程序更高效，但是也有一些缺点，比如你要使用第三方库函数时，即会在任务中调用它，也会在ISR总调用它。

这个第三方库函数用到了FreeRTOS的API函数，你无法修改库函数。

这个问题可以解决：

- 把中断的处理推迟到任务中进行(Defer interrupt processing)，在任务中调用库函数
- 尝试在库函数中使用"FromISR"函数：
 - 在任务中、在ISR中都可以调用"FromISR"函数
 - 反过来就不行，非FromISR函数无法在ISR中使用
- 第三方库函数也许会提供OS抽象层，自行判断当前环境是在任务还是在ISR中，分别调用不同的函数

两套API列表

类型	在任务中	在ISR中
队列(queue)	xQueueSendToBack	xQueueSendToBackFromISR
	xQueueSendToFront	xQueueSendToFrontFromISR
	xQueueReceive	xQueueReceiveFromISR
	xQueueOverwrite	xQueueOverwriteFromISR
	xQueuePeek	xQueuePeekFromISR
信号量(semaphore)	xSemaphoreGive	xSemaphoreGiveFromISR
	xSemaphoreTake	xSemaphoreTakeFromISR
类型	在任务中	在ISR中
事件组(event group)	xEventGroupSetBits	xEventGroupSetBitsFromISR
	xEventGroupGetBits	xEventGroupGetBitsFromISR
任务通知(task notification)	xTaskNotifyGive	vTaskNotifyGiveFromISR
	xTaskNotify	xTaskNotifyFromISR
软件定时器(software timer)	xTimerStart	xTimerStartFromISR
	xTimerStop	xTimerStopFromISR
	xTimerReset	xTimerResetFromISR
	xTimerChangePeriod	xTimerChangePeriodFromISR

在任务中	在ISR中
xQueueSendToBack	xQueueSendToBackFromISR
xQueueSendToFront	xQueueSendToFrontFromISR
xQueueReceive	xQueueReceiveFromISR
xQueueOverwrite	xQueueOverwriteFromISR
xQueuePeek	xQueuePeekFromISR
xSemaphoreGive	xSemaphoreGiveFromISR
xSemaphoreTake	xSemaphoreTakeFromISR

在任务中	在ISR中
xEventGroupSetBits	xEventGroupSetBitsFromISR blocked → ready
xEventGroupGetBits	xEventGroupGetBitsFromISR
xTaskNotifyGive	vTaskNotifyGiveFromISR
xTaskNotify	xTaskNotifyFromISR
xTimerStart	xTimerStartFromISR
xTimerStop	xTimerStopFromISR
xTimerReset	xTimerResetFromISR
xTimerChangePeriod	xTimerChangePeriodFromISR

中断比任何任务的优先级都高

ISR需要记录是否发生调度，并自行发起调度(可以由程序员设置)

xHigherPriorityTaskWoken参数

xHigherPriorityTaskWoken的含义是：是否有更高优先级的任务被唤醒了。

如果为pdTRUE，则意味着后面要进行任务切换。

以写队列为例：

任务A调用xQueueSendToBack() 写队列，有几种情况发生：

- 队列满了，任务A阻塞等待，另一个任务B运行
- 队列没满，任务A成功写入队列，但是它导致另一个任务B被唤醒，任务B的优先级更高：任务B先运行
- 队列没满，任务A成功写入队列，即刻返回

可以看到，在任务中调用API函数可能导致任务阻塞、任务切换，这叫做**context switch, 上下文切换**。

这个函数可能很长时间才返回，在函数的内部实现了任务切换。

xQueueSendToBackFromISR() 函数也可能导致任务切换，但是不会在函数内部进行切换，而是返回一个参数：表示是否需要切换。

函数原型与用法如下：

会尽快执行

不会发生阻塞(直接返回)

任务中函数：

可以快速跟守待队列中任务

可发起调度(任务优先级较高)

ISR中函数：

可以唤醒并记录是否调度

不会引发调度(先执行中断)

(将值作为参数传入)

```

/*
 * 往队列尾部写入数据，此函数可以在中断函数中使用，不可阻塞
 */
BaseType_t xQueueSendToBackFromISR(
    QueueHandle_t xQueue,
    const void *pvItemToQueue,
    BaseType_t *pxHigherPriorityTaskWoken
);

/* 用法示例 */

BaseType_t xHigherPriorityTaskWoken = pdFALSE;
xQueueSendToBackFromISR(xQueue, pvItemToQueue, &xHigherPriorityTaskWoken);

if (xHigherPriorityTaskWoken == pdTRUE)
{
    /* 任务切换 */
}

```

}

程序员自行调用任务切换的程序。

pxHigherPriorityTaskWoken参数，就是用来保存函数的结果：是否需要切换

- *pxHigherPriorityTaskWoken等于pdTRUE：函数的操作导致更高优先级的任务就绪了，ISR应该进行任务切换
- *pxHigherPriorityTaskWoken等于pdFALSE：没有进行任务切换的必要

为什么不在"FromISR"函数内部进行任务切换，而只是标记一下而已呢？为了效率！

```

void XXX_ISR()
{
    int i;
    for (i = 0; i < N; i++)
    {
        xQueueSendToBackFromISR(...); /* 被多次调用 */
    }
}

```

ISR中有可能多次调用"FromISR"函数，如果在"FromISR"内部进行任务切换，会浪费时间。

解决方法是：

- 在"FromISR"中标记是否需要切换
- 在ISR返回之前再进行任务切换

示例代码如下：

```
void XXX_ISR()
{
    int i;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    for (i = 0; i < N; i++)
    {
        xQueueSendToBackFromISR(..., &xHigherPriorityTaskWoken); /* 被多次调用 */
    }

    /* 最后再决定是否进行任务切换 */
    if (xHigherPriorityTaskWoken == pdTRUE)
    {
        /* 任务切换 */
    }
}
```

上述的例子很常见，比如UART中断：在UART的ISR中读取多个字符，发现收到回车符时才进行任务切换。

在ISR中调用API时不进行任务切换，而只是在"xHigherPriorityTaskWoken"中标记一下，除了效率，还有多种好处：

- 效率高：避免不必要的任务切换
- 让ISR更可控：中断随机产生，在API中进行任务切换的话，可能导致问题更复杂
- 可移植性
- 在Tick中断中，调用vApplicationTickHook()：它运行与ISR，只能使用"FromISR"的函数

使用"FromISR"函数时，如果不想使用xHigherPriorityTaskWoken参数，可以设置为NULL。

切换任务

FreeRTOS的ISR函数中，使用两个宏进行任务切换：

```
1 | portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
2 | portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

这两个宏做的事情是完全一样的，在老版本的FreeRTOS中，

- portEND_SWITCHING_ISR 使用汇编实现
 - portYIELD_FROM_ISR 使用C语言实现
- 新版本都统一使用portYIELD_FROM_ISR

```

void XXX_ISR()
{
    int i;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    for (i = 0; i < N; i++)
    {
        xQueueSendToBackFromISR(..., &xHigherPriorityTaskWoken); /* 被多次调用 */
    }

    /* 最后再决定是否进行任务切换
     * xHigherPriorityTaskWoken为pdTRUE时才切换
     */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}

```

注意：这是在中断中发起任务调度。

在任务中可以使用taskyield()

中断延迟处理

前面讲过，ISR要尽量快，否则：

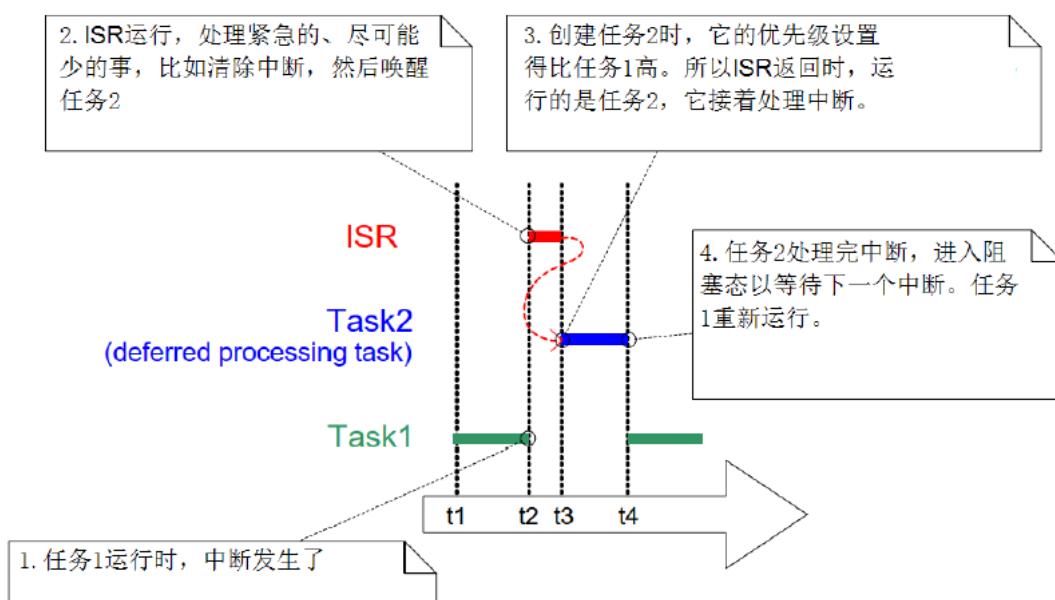
- 其他低优先级的中断无法被处理：实时性无法保证
- 用户任务无法被执行：系统显得很卡顿
- 如果运行中断嵌套，这会更复杂，ISR越快执行约有助于中断嵌套

如果这个硬件中断的处理，就是非常耗费时间呢？

对于这类中断的处理就要分为2部分：

- ISR：尽快做些清理、记录工作，然后触发某个任务
- 任务：更复杂的事情放在任务中处理

这种处理方式叫“中断的延迟处理”(Deferring interrupt processing)，处理流程如下图所示：



- t1：任务1运行，任务2阻塞
- t2：发生中断，

- 该中断的ISR函数被执行，任务1被打断
- ISR函数要尽快能快速地运行，它做一些必要的操作(比如清除中断)，然后唤醒任务2
- t3：在创建任务时设置任务2的优先级比任务1高(这取决于设计者)，所以ISR返回后，运行的是任务2，它要完成中断的处理。任务2就被称为"deferred processing task"，中断的延迟处理任务。
- t4：任务2处理完中断后，进入阻塞态以等待下一个中断，任务1重新运行

中断的延迟任务处理

中断任务通信

前面讲解过的队列、信号量、互斥量、事件组、任务通知等等方法，都可使用。
要注意的是，在ISR中使用的函数要有"FromISR"后缀。

十三、资源管理

在前面讲解互斥量时，引入过临界资源的概念。

在前面课程里，已经实现了临界资源的互斥访问。

本章节的内容比较少，只是引入两个功能：

- 屏蔽/使能中断
- 暂停/恢复调度器。

要独占式地访问临界资源，有3种方法：(内核临界资源常常需要原子访问)

- 公平竞争：比如使用互斥量，谁先获得互斥量谁就访问临界资源，这部分内容前面讲过。
- 谁要跟我抢，我就灭掉谁：
 - 中断要跟我抢？我屏蔽中断
 - 其他任务要跟我抢？我禁止调度器，不运行任务切换

屏蔽中断

屏蔽中断有两套宏：任务中使用、ISR中使用：

- 任务中使用：taskENTER_CRITICAL()/taskEXIT_CRITICAL()
- ISR中使用：taskENTER_CRITICAL_FROM_ISR()/taskEXIT_CRITICAL_FROM_ISR()

任务中屏蔽中断：

```

1 /* 在任务中，当前时刻中断是使能的
2  * 执行这句代码后，屏蔽中断
3 */
4 taskENTER_CRITICAL();
5 /* 访问临界资源 */
6
7 /* 重新使能中断 */
8 taskEXIT_CRITICAL();

```

在taskENTER_CRITICAL()/taskEXIT_CRITICAL() 之间：

- 低优先级的中断被屏蔽了：
优先级低于、等于configMAX_SYSCALL_INTERRUPT_PRIORITY

- 高优先级的中断可以产生：

优先级高于 configMAX_SYSCALL_INTERRUPT_PRIORITY

但是，这些中断ISR里，不允许使用FreeRTOS的API函数

- 任务调度依赖于中断、依赖于API函数，所以：这两段代码之间，不会有任务调度产生

configMAX_SYSCALL_INTERRUPT_PRIORITY 定义了可以调用 FreeRTOS API 函数的最高中断优先级。任何优先级高于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的中断都不能调用 FreeRTOS 的 API 函数。

这些高优先级中断通常用于处理时间关键的任务，因为它们不会被 FreeRTOS 的内核中断。

这套taskENTER_CRITICAL()/taskEXIT_CRITICAL() 宏，是可以递归使用的，它的内部会记录嵌套的深度，只有嵌套深度变为0时，调用taskEXIT_CRITICAL() 才会重新使能中断。

使用taskENTER_CRITICAL()/taskEXIT_CRITICAL() 来访问临界资源是很粗鲁的方法：

- 中断无法正常运行
 - 任务调度无法进行
- 所以，之间的代码要尽可能快速地执行

ISR中屏蔽中断

```
void vAnInterruptServiceRoutine( void )
{
    /* 用来记录当前中断是否使能 */
    UBaseType_t uxSavedInterruptStatus;

    /* 在ISR中，当前时刻中断可能是使能的，也可能是禁止的
     * 所以要记录当前状态，后面要恢复为原先的状态
    */
}
```

```
/* 执行这句代码后，屏蔽中断
*/
uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

/* 访问临界资源 */

/* 恢复中断状态 */
taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );
/* 现在，当前ISR可以被更高优先级的中断打断了 */
}
```

1. 屏蔽中断：

- 在 ISR 中，有时需要屏蔽其他中断，以确保当前中断处理过程不被打断。
- 屏蔽中断可以通过设置中断屏蔽寄存器或使用特定的函数来实现。

2. 记录返回值：

- 屏蔽中断时，通常会返回一个值，该值表示中断屏蔽寄存器的原始状态。
- 记录这个返回值是为了在 ISR 结束后能够恢复中断的原始状态。

3. 恢复中断：

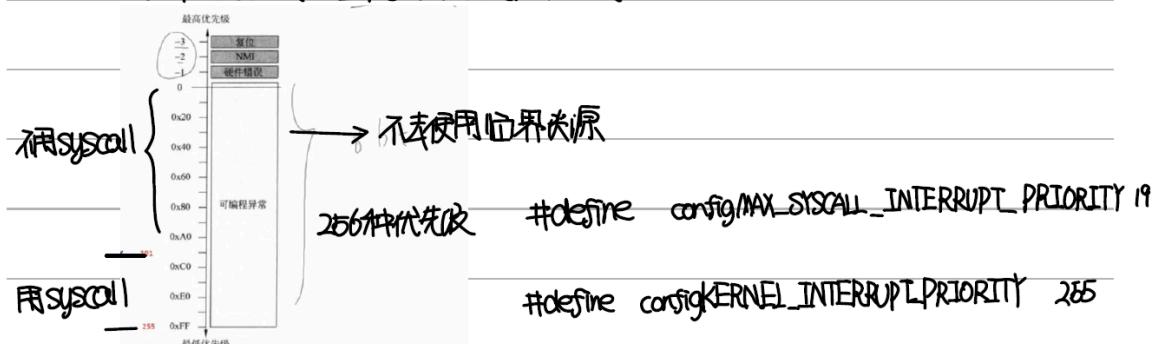
- 在 ISR 结束时，需要使用记录的返回值来恢复中断屏蔽寄存器的原始状态。
- 这样可以确保在 ISR 执行期间屏蔽的中断能够正确恢复，不会影响系统的正常运行。

在taskENTER_CRITICAL_FROM_ISR()/taskEXIT_CRITICAL_FROM_ISR() 之间:

- 低优先级的中断被屏蔽了:
优先级低于、等于configMAX_SYSCALL_INTERRUPT_PRIORITY
- 高优先级的中断可以产生:
优先级高于configMAX_SYSCALL_INTERRUPT_PRIORITY
但是，这些中断ISR里，不允许使用FreeRTOS的API函数
- 任务调度依赖于中断、依赖于API函数，所以：这两段代码之间，不会有任务调度产生

屏蔽中断：(屏蔽某些优先级较低的中断)

中断优先级：(数值越低，优先级越高)



屏蔽某类中断 (用用syscall的中断)

注意任务优先级是越大优先级越高，中断是越小优先级越高。

暂停调度器

如果有别的任务来跟你竞争临界资源，你可以把中断关掉：这当然可以禁止别的任务运行，但是这代价太大了。它会影响到中断的处理。

如果只是禁止别的任务来跟你竞争，不需要关中断，暂停调度器就可以了：在这期间，中断还是可以发生、处理。

暂停一个任务，唤醒一个任务：

vTaskSuspend()

```
#if ( INCLUDE_vTaskSuspend == 1 )

void vTaskSuspend( TaskHandle_t xTaskToSuspend )
{
    TCB_t * pxTCB;

    taskENTER_CRITICAL();
    {

        /* If null is passed in here then it is the running task that is
         * being suspended. */
        pxTCB = prvGetTCBFromHandle( xTaskToSuspend );

        traceTASK_SUSPEND( pxTCB );

        /* Remove task from the ready/delayed list and place in the
         * suspended list. */
        if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( uBaseType_t ) 0 )
        {
            taskRESET_READY_PRIORITY( pxTCB->uxPriority );
        }
        else
        {
            mTCOVERAGE_TEST_MARKER();
        }
    }
}
```

vTaskResume()

```
#if ( INCLUDE_vTaskSuspend == 1 )

void vTaskResume( TaskHandle_t xTaskToResume )
{
    TCB_t * const pxTCB = xTaskToResume;

    /* It does not make sense to resume the calling task. */
    configASSERT( xTaskToResume );

    /* The parameter cannot be NULL as it is impossible to resume the
     * currently executing task. */
    if( ( pxTCB != pxCurrentTCB ) && ( pxTCB != NULL ) )
    {
        taskENTER_CRITICAL();
        {

            if( prvTaskIsTaskSuspended( pxTCB ) != pdFALSE )
            {
                traceTASK_RESUME( pxTCB );

                /* The ready list can be accessed even if the scheduler is
                 * suspended because this is inside a critical section. */
                ( void ) uxListRemove( &( pxTCB->xStateListItem ) );
                prvAddTaskToReadyList( pxTCB );

                /* A higher priority task may have just been resumed. */
                if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )
                {
                    /* This yield may not cause the task just resumed to run,
                     * but will leave the lists in the correct state for the
                     * next yield. */
                    taskYIELD_IF_USING_PREEMPTION();
                }
            }
        }
    }
}
```

暂停所有任务调度，唤醒所有任务：

```
/* 暂停调度器 */
void vTaskSuspendAll( void );

/* 恢复调度器
 * 返回值: pdTRUE表示在暂定期间有更高优先级的任务就绪了
 *          可以不理会这个返回值
 */
BaseType_t xTaskResumeAll( void );
```

示例代码如下：

```
vTaskSuspendScheduler();

/* 访问临界资源 */

xTaskResumeScheduler();
```

这套vTaskSuspendScheduler()/xTaskResumeScheduler() 宏，是可以递归使用的，它的内部会记录嵌套的深度，只有嵌套深度变为0时，才能重新启用。
uxSchedulerSuspended可以记录嵌套的层数。

十四、调试

FreeRTOS提供了多种调试手段：

- 打印
- 断言
- Trace
- Hook回调

打印：

FreeRTOS工程中使用了microlib，里面实现了printf
我们只需实现一下函数即可使用printf：

```
1 | int fputc(int ch,FILE *f);
```

断言：

一般的C库里面，断言就是一个函数：

```
1 | void assert(scalar expression);
```

它的作用是：确认 expression 必须为真，如果 expression 为假的话就中止程序。

在FreeRTOS里，使用configASSERT()，比如：

```
1 | #define configASSERT(x) if (!x) while(1);
```

也可以让他提供更多信息：

```
1 #define configASSERT(x) \
2 if (!x) \
3 { \
4 printf("%s %s %d\r\n", __FILE__, __FUNCTION__, __LINE__); \
5 while(1); \
6 }
```

configASSERT(x)中，如果x为假，表示发生了很严重的错误，必须停止系统的运行。

它用在很多场合，比如：

- 队列操作

```
BaseType_t xQueueGenericSend( QueueHandle_t xQueue,
                               const void * const pvItemToQueue,
                               TickType_t xTicksToWait,
                               const BaseType_t xCopyPosition )
{
    BaseType_t xEntryTimeSet = pdFALSE, xYieldRequired;
    TimeOut_t xTimeOut;
    Queue_t * const pxQueue = xQueue;

    configASSERT( pxQueue );
    configASSERT(!((pvItemToQueue == NULL) && (pxQueue->uxItemSize != (UBaseType_t)OU)));
    configASSERT( !(xCopyPosition == queueOVERWRITE) && (pxQueue->uxLength != 1 ));
```

- 中断级别的判断

```
void vPortValidateInterruptPriority( void )  
{  
    uint32_t ulCurrentInterrupt;  
    uint8_t ucCurrentPriority;  
  
    /* Obtain the number of the currently executing interrupt. */  
    ulCurrentInterrupt = vPortGetIPSR();  
  
    /* Is the interrupt number a user defined interrupt? */  
    if( ulCurrentInterrupt >= portFIRST_USER_INTERRUPT_NUMBER )  
    {  
        /* Look up the interrupt's priority. */  
        ucCurrentPriority = pcInterruptPriorityRegisters[ ulCurrentInterrupt ];  
  
        configASSERT( ucCurrentPriority >= ucMaxSysCallPriority );  
    }  
}
```

260

百问网

```
/* Look up the interrupt's priority. */  
ucCurrentPriority = pcInterruptPriorityRegisters[ ulCurrentInterrupt ];  
  
configASSERT( ucCurrentPriority >= ucMaxSysCallPriority );  
}
```

Trace

FreeRTOS 中定义了很多 trace 开头的宏，这些宏被放在系统个关键位置。
它们一般都是空的宏，这不会影响代码：不影响编程处理的程序大小、不影响运行时间。
我们要调试某些功能时，可以修改宏：修改某些标记变量、打印信息等待。

trace 宏	描述
traceTASK_INCREMENT_TICK(xTickCount)	当 tick 计数自增之前此宏函数被调用。 参数 xTickCount 当前的 Tick 值，它还没有增加。
traceTASK_SWITCHED_OUT()	vTaskSwitchContext 中，把当前任务切换出去之前调用此宏函数。
traceTASK_SWITCHED_IN()	vTaskSwitchContext 中，新的任务已经被切换进来了，就调用此函数。
traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue)	当正在执行的当前任务因为试图去读取一个空的队列、信号或者互斥量而进入阻塞状态时，此函数会被立即调用。参数 pxQueue 保存的是试图读取的目标队列、信号或者互斥量的句柄，传递给此宏函数。
traceBLOCKING_ON_QUEUE_SEND(pxQueue)	当正在执行的当前任务因为试图往一个已经写满的队列或者信号或者互斥量而进入了阻塞状态时，此函数会被立即调用。参数 pxQueue 保存的是试图写入的目标队列、信号或者互斥量的句柄，传递给此宏函数。

traceQUEUE_SEND(pxQueue)	<p>当一个队列或者信号发送成功时，此宏函数会在内核函数 xQueueSend(),xQueueSendToFront(),xQueueSendToBack(), 以及所有的信号 give 函数中被调用，参数 pxQueue 是要发送的目标队列或信号的句柄，传递给此宏函数。</p>
traceQUEUE_SEND_FAILED(pxQueue)	<p>当一个队列或者信号发送失败时，此宏函数会在内核函数 xQueueSend(),xQueueSendToFront(),xQueueSendToBack(), 以及所有的信号 give 函数中被调用，参数 pxQueue 是要发送的目标队列或信号的句柄，传递给此宏函数。</p>

	当读取一个队列或者接收信号成功时，此宏函数会在内核函数 xQueueReceive() 以及所有的信号 take 函数中被调用，参数 pxQueue 是要接收的目标队列或信号的句柄，传递给此宏函数。
traceQUEUE_RECEIVE(pxQueue)	当读取一个队列或者接收信号失败时，此宏函数会在内核函数 xQueueReceive() 以及所有的信号 take 函数中被调用，参数 pxQueue 是要接收的目标队列或信号的句柄，传递给此宏函数。
traceQUEUE_RECEIVE_FAILED(pxQueue)	当在中断中发送一个队列成功时，此函数会在 xQueueSendFromISR() 中被调用。参数 pxQueue 是要发送的目标队列的句柄。
traceQUEUE_SEND_FROM_ISR(pxQueue)	当在中断中发送一个队列失败时，此函数会在 xQueueSendFromISR() 中被调用。参数 pxQueue 是要发送的目标队列的句柄。
traceQUEUE_SEND_FROM_ISR_FAIL(pxQueue)	当在中断中读取一个队列成功时，此函数会在 xQueueReceiveFromISR() 中被调

	用。参数 pxQueue 是要发送的目标队列的句柄。
traceQUEUE_RECEIVE_FROM_ISR_F AILED(pxQueue)	当在中断中读取一个队列失败时，此函数会在 xQueueReceiveFromISR() 中被调用。参数 pxQueue 是要发送的目标队列的句柄。
traceTASK_DELAY_UNTIL()	当一个任务因为调用了 vTaskDelayUntil() 进入了阻塞状态的前一刻此宏函数会在 vTaskDelayUntil() 中被立即调用。
traceTASK_DELAY()	当一个任务因为调用了 vTaskDelay() 进入了阻塞状态的前一刻此宏函数会在 vTaskDelay 中被立即调用。

Hook 函数

Malloc Hook 函数

编程时，一般的逻辑错误都容易解决。
难以处理的是内存越界、栈溢出等。
内存越界经常发生在堆的使用过程总：堆，就是使用 malloc 得到的内存。
并没有很好的方法检测内存越界，但是可以提供一些回调函数：

使用 pvPortMalloc 失败时，如果在 FreeRTOSConfig.h 里配置 configUSE_MALLOC_FAILED_HOOK 为 1，会调用：

```
1 | void vApplicationMallocFailedHook( void );
```

可以用这个回调函数实现打印错误信息

栈溢出Hook函数

在切换任务(vTaskSwitchContext)时调用 taskCHECK_FOR_STACK_OVERFLOW 来检测栈是否溢出，如果溢出会调用：

```
1 | void vApplicationStackOverflowHook( TaskHandle_t xTask, char * pcTaskName );
```

可以用这个回调函数实现打印错误信息

怎么判断栈溢出？有两种方法：

方法1：

- 当前任务被切换出去之前，它的整个运行现场都被保存在栈里，这时很可能就是它对栈的使用到达了峰值。
- 这方法很高效，但是并不精确
- 比如：任务在运行过程中调用了函数 A 大量地使用了栈，调用完函数 A 后才被调度。

方法2：

- 创建任务时，它的栈被填入固定的值，比如：0xa5
- 检测栈里最后 16 字节的数据，如果不是 0xa5 的话表示栈即将、或者已经被用完了
- 没有方法 1 快速，但是也足够快
- 能捕获几乎所有的栈溢出

为什么是几乎所有？可能有些函数使用栈时，非常凑巧地把栈设置为 0xa5：几乎不可能

十五、优化

在 Windows 中，当系统卡顿时我们可以查看任务管理器找到最消耗 CPU 资源的程序。在FreeRTOS中，我们也可以查看任务使用CPU的情况、使用栈的情况，然后针对性地进行优化。这就是查看"任务的统计"信息。

栈使用情况

在创建任务时分配了栈，可以填入固定的数值比如 0xa5，以后可以使用以下函数查看"栈的高水位"，也就是还有多少空余的栈空间：

```
1 | UBaseType_t uxTaskGetStackHighwaterMark( TaskHandle_t xTask );
```

原理是：从栈底往栈顶逐个字节地判断，它们的值持续是 0xa5 就表示它是空闲的。

参数说明：

参数/返回值	说明
xTask	哪个任务
返回值	<p>任务运行时、任务被切换时，都会用到栈。栈里原来值(0xa5)就会被覆盖。</p> <p>逐个函数从栈的尾部判断栈的值连续为 0xa5 的个数，它就是任务运行过程中空闲内存容量的最小值。</p> <p>注意：假设从栈尾开始连续为 0xa5 的栈空间是 N 字节，返回值是 $N/4$。</p>

运行时间统计

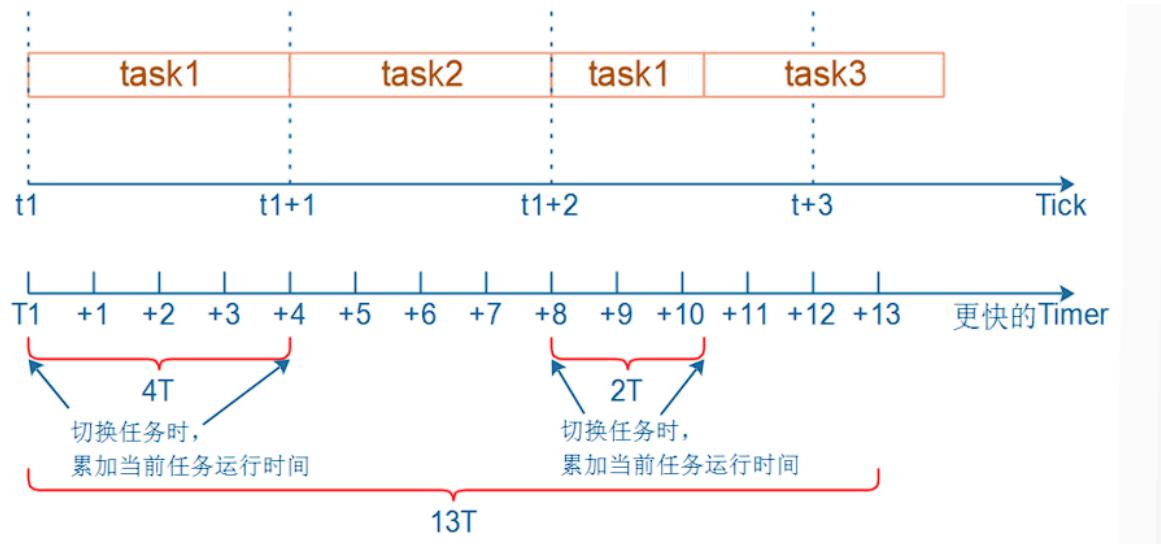
对于同优先级的任务，它们按照时间片轮流运行：你执行一个 Tick，我执行一个 Tick。

是否可以在Tick中断函数中，统计当前任务的累计运行时间？

不行！很不精确，因为有更高优先级的任务就绪时，当前任务还没运行一个完整的Tick就被抢占了。

我们需要比Tick更快的时钟，比如Tick周期时1ms，我们可以使用另一个定时器，让它发生中断的周期时0.1ms甚至更短。

使用这个定时器来衡量一个任务的运行时间，原理如下图所示：



在这段时间里: task1的CPU占用率是 $(4+2)/13=46\%$

- 切换到 Task1 时, 使用更快的定时器记录当前时间 T1
- Task1 被切换出去时, 使用更快的定时器记录当前时间 T4
- (T4-T1)就是它运行的时间, 累加起来
- 关键点: 在 `vTaskSwitchContext` 函数中, 使用**更快的定时器**统计运行时间

在上下文切换函数中, 使用更快的定时器统计运行时间。

代码:

配置:

```
1 #define configGENERATE_RUN_TIME_STATS 1
2 #define configUSE_TRACE_FACILITY 1
3 #define configUSE_STATS_FORMATTING_FUNCTIONS 1
```

实现宏 `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()`, 它用来
初始化更快的定时器

实现这两个宏之一, 它们用来返回当前时钟值(更快的定时器)

- `portGET_RUN_TIME_COUNTER_VALUE()`: 直接返回时钟值
- `portALT_GET_RUN_TIME_COUNTER_VALUE(Time)`: 设置 Time 变量等于时钟值

代码执行流程:

- 初始化更快的定时器: 启动调度器时

```
2071:    if( INCLUDE_vTaskSuspend == 1 )
2072:        prvTasksTaskSuspended
2073:    endif
2074:    if( INCLUDE_vTaskSuspend == 1 )
2075:        vTaskResume
2076:    endif
2077:    if( !INCLUDE_xTaskResumeFromISR )
2078:        xTaskResumeFromISR
2079:    endif
2080:    vTaskStartScheduler
2081:    vTaskEndScheduler
2082:    portCONFIGURE_TIMER_FOR_RUN_TIME_STATS();
2083:    traceTASK_SWITCHED_IN();
2084:
2085:
```

在任务切换时统计运行时间

```

3012: void vTaskSwitchContext( void )
3013: {
3014:     if( uxSchedulerSuspended != ( UBaseType_t ) pdFALSE )
3015:     {
3016:         /* The scheduler is currently suspended - do not allow a context
3017:          * switch. */
3018:         xYieldPending = pdTRUE;
3019:     }
3020:     else
3021:     {
3022:         xYieldPending = pdFALSE;
3023:         traceTASK_SWITCHED_OUT();
3024:
3025: #if ( configGENERATE_RUN_TIME_STATS == 1 )
3026: {
3027:     #ifdef portALT_GET_RUN_TIME_COUNTER_VALUE
3028:         portALT_GET_RUN_TIME_COUNTER_VALUE( ulTotalRunTime );
3029:     #else
3030:         ulTotalRunTime = portGET_RUN_TIME_COUNTER_VALUE();
3031:     #endif
3032:
3033:     /* Add the amount of time the task has been running to the
3034:      * accumulated time so far. The time the task started running was
3035:      * stored in ulTaskSwitchedInTime. Note that there is no overflow
3036:      * protection here so count values are only valid until the timer
3037:      * overflows. The guard against negative values is to protect
3038:      * against suspect run time stat counter implementations - which
3039:      * are provided by the application, not the kernel. */
3040:     if( ulTotalRunTime > ulTaskSwitchedInTime )
3041:     {
3042:         pxCurrentTCB->ulRunTimeCounter += ( ulTotalRunTime - ulTaskSwitchedInTime );
3043:     }
3044:     else
3045:     {
3046:         mtCOVERAGE_TEST_MARKER();
3047:     }
3048:
3049: }
3050:
3051: #endif /* configGENERATE_RUN_TIME_STATS */
3052:

```

1. 得到当前时间

2. 计算运行时间: 当前时间 - 上次运行时刻

3. 累加运行时间

4. 记录当前时间

● 获得统计信息，可以使用下列函数

- uxTaskGetSystemState: 对于每个任务它的统计信息都放在一个 TaskStatus_t 结构体里
- vTaskList: 得到的信息是可读的字符串，比如
- vTaskGetRunTimeStats: 得到的信息是可读的字符串

统计信息函数说明

uxTaskGetSystemState: 获得任务的统计信息

```

1  UBaseType_t uxTaskGetSystemState( TaskStatus_t * const pxTaskStatusArray,
2  const UBaseType_t uxArraySize,
3  uint32_t * const pulTotalRunTime );

```

参数	描述
pxTaskStatusArray	指向一个 TaskStatus_t 结构体数组，用来保存任务的统计信息。 有多少个任务？可以用 <code>uxTaskGetNumberOfTasks()</code> 来获得。
uxArraySize	数组大小、数组项个数，必须大于或等于 <code>uxTaskGetNumberOfTasks()</code>
pulTotalRunTime	用来保存当前总的运行时间(更快的定时器)，可以传入 NULL
返回值	传入的 pxTaskStatusArray 数组，被设置了几个数组项。 注意：如果传入的 uxArraySize 小于 <code>uxTaskGetNumberOfTasks()</code> ，返回值就是 0

vTaskList : 获得任务的统计信息，形式为可读的字符串。

注意，pcWriteBuffer 必须足够大。

```
1 | void vTaskList( signed char *pcWriteBuffer );
```

可读信息格式如下：

任务名	任务状态	优先级	空闲栈	任务号
tcpip	R	3	393	0
Tmr Svc	R	3	111	48
QConsB1	R	1	143	3
QProdB5	R	0	144	7
QConsB6	R	0	143	8
Po1SEM1	R	0	145	11
Po1SEM2	R	0	145	12
GenQ	R	0	155	17
MuLow	R	0	147	18
Rec3	R	0	141	30
SUSP_RX	R	0	148	36
Math1	R	0	167	38
Math2	R	0	167	39

vTaskGetRunTimeStats：获得任务的运行信息，形式为可读的字符串。

注意，pcWriteBuffer 必须足够大。

```
1 | void vTaskGetRunTimeStats( signed char *pcwriteBuffer );
```

可读信息如下：

任务名	任务运行时间	运行时间百分比
Po1SEM1	994	<1%
Po1SEM2	23248	1%
GenQ	194479	16%
MuLow	3690	<1%
Rec3	229450	18%
CNT1	242720	19%
PeekL	94	<1%
CNT_INC	165	<1%
CNT2	243166	20%
SUSP_RX	243192	20%
IDLE	55	<1%