

RTOS 韦东山

入门



JNOTES

CREATIVE NOTES



单片机程序设计模式

裸机程序设计模式

① 循环模式

② 前后台形式

③ 定时器方式

cnt cnt%2 cnt%5 延时中断

④ 基于状态机 static state 状态机 switch-case

→ 可解决不同函数之间相互影响

多任务系统

① 多任务模式：

交替执行 \Rightarrow RTOS 完成

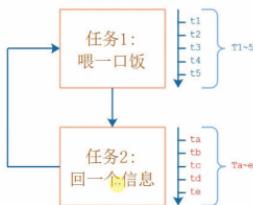
② 同步操作

③ 异步操作

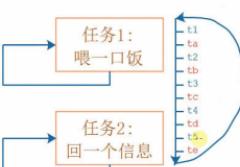
freertos入门

第一节 RTOS 架构的概念

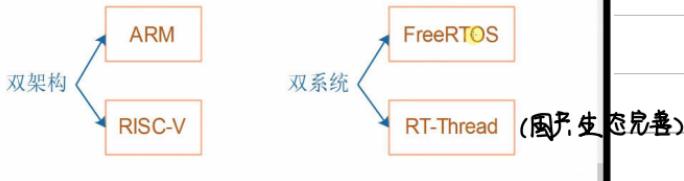
裸机开发



RTOS开发



CPU 架构



第二节 堆栈

堆的概念

空闲的内存

简单的实现

```

22 char heap_buf[1024]; 空闲内存
23 int pos = 0; 空闲地址
24
25 void *my_malloc(int size) 内存分配
26 {
27     int old_pos = pos;
28     pos += size;
29     return &heap_buf[old_pos];
30 }
31
32 void my_free(void *buf) 内存释放
33 {
34     /* err */
35 }
36
  
```

keil5 debug 方法

栈的作用

桌面英雄

测试 demo.

```

37 void c_fun(void)
38 {
39     ...
40 }
41 
42 void b_fun(void)
43 {
44     ...
45 }
46 
47 int a_fun(int val)
48 {
49     int a = 8;
50     a += val;
51 
52     b_fun(); // ①
53 
54     c_fun(); // ②
55 
56     return a;
57 }

```

```

60 int main(void)
61 {
62     char ch = 65; // char ch = 'A';
63     int i;
64     char *buf = my_malloc(100);
65 
66     unsigned char uch = 200;
67 
68     for (i = 0; i < 26; i++)
69         buf[i] = 'A' + i;
70 
71     a_fun(46); // ①
72 
73     return 0;
74 }

```

返回地址位置保存在郎

main → a.fun

LR(LinkRegister) = ①地址

调用 a.fun

a.fun → b.fun

LR = ②地址

调用 b.fun

LR会覆盖！

简述：郎中地址保存入栈

C函数执行：

① 堆栈 (郎...，局部变量)

② LR... 有几栈

③ 执行代码

过程

栈：从高地址向低地址扩充

main

a.fun

b.fun

SP=SP-N

内核

main end

① BL main

LR=返回地址

JR main

SP=SP-N

LR

... 其它寄存器

局部变量

郎中 stack

main end

SP=SP-M

LR...

OS调用(下一条指令地址)

郎中 stack

LR...

OS调用(下一条指令地址)

郎中 stack

局部变量

SP=SP-P

一个帧有即可

扩大郎中返回地址

② a.fun()

LR=a.fun

执行 a.fun

③ b.fun

LR=b.fun

执行 b.fun

每个任务都有自己的堆空间 (FreeRTOS中任务即线程)

相同核心

第三节 多个RTOS程序

启动源码中指向一个RTOS程序

FreeRTOS

Download FreeRTOS

Download the primary FreeRTOS and Long Term Support (LTS) releases below. Refer to FAQ for details on GitHub repository structure and versioning.

FreeRTOS 2021.07.00

Contains the latest FreeRTOS Kernel, FreeRTOS+ Libraries and AWS IoT libraries, along with example projects. Source code is also available on GitHub.

[Download](#)

带有工程示例

RTOS目录结构(横向)

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

```
|-- FreeRTOS
|   |-- Demo
|   |   |-- 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
|   |   |   |-- FreeRTOS源码
|   |   |-- Source
|   |       |-- 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
|   |           |-- RVD3
|   |               |-- 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
|   |           |-- 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
```

带部分移植

带移植相关

(可删)

更新到Keil

得到精简的RTOS工程

增加串口打印

①去无关代码

②增加串口打印

实现 fputc 方法

< 初始化串口

实现 fputc() 修改 fputc() 可实现 printf() 打印至串口

初始化串口源

```

64: /* 
65: void SerialPortInit(void)
66: {
67:     unsigned long ulWantedBaud = 115200;
68:     USART_InitTypeDef USART_InitStructure;
69:     GPIO_InitTypeDef GPIO_InitStructure;
70: 
71:     /* If the queue/semaphore was created correctly then setup the serial port
72: hardware. */
73:     /* Enable USART1 clock */
74:     RCC_APB2PeriphClockCmd( RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE );
75: 
76:     /* Configure USART1 Rx (PA10) as input floating */
77:     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
78:     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
79:     GPIO_Init( GPIOA, &GPIO_InitStructure );
80: 
81:     /* Configure USART1 Tx (PA9) as alternate function push-pull */
82:     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
83:     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
84:     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
85:     GPIO_Init( GPIOA, &GPIO_InitStructure );
86: 
87:     USART_InitStructure.USART_BaudRate = ulWantedBaud;
88:     USART_InitStructure.USART_WordLength = USART_WordLength_8b;
89:     USART_InitStructure.USART_StopBits = USART_StopBits_1;
90:     USART_InitStructure.USART_Parity = USART_Parity_No;
91:     USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
92:     USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
93:     USART_InitStructure.USART_Clock = USART_Clock_Disable;
94:     USART_InitStructure.USART_CPOL = USART_CPOL_Low;
95:     USART_InitStructure.USART_CPHA = USART_CPHA_2edge;
96:     USART_InitStructure.USART_LastBit = USART_LastBit_Disable;
97: 
98:     USART_Init( USART1, &USART_InitStructure );
99: 
100:    USART_ITConfig( USART1, USART_IT_RXNE, ENABLE );
101: 
102:    USART_Cmd( USART1, ENABLE );
103: } // end SerialPortInit */
104: */
```

fputc(char c, FILE *f): 将 c 写入 fp 文件中

改写把文件 c 写入串口

```

}; int fputc( int ch, FILE *f )
{
    USART_TypeDef* USARTx = USART1;      串口应为外设
    while ( ((USARTx->SR & (1<<7)) == 0);      等待空闲
    USARTx->DR = ch;      发送字符
    return ch;
}
```

第四节. 第一个RTOS程序

补充知识 `typedef` 定义函数指针类型

`typedef int (*FuncPtr) (int, int);`

定义 `FuncPtr` 类型函数指针 返回值为 `int`, 参数为 `(int, int)`

`typedef` 定义内存体结构类型

`typedef struct 结构名 *指针名`

创建任务 `xTaskCreate()`

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
                        const char * const pcName,
                        const configSTACK_DEPTH_TYPE usStackDepth,
                        void * const pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t * const pxCreatedTask )
```

任务执行函数

任务 标识 服务参数

优先级 任务控制句柄

demo

```
● ○ ●
1 void TaskFunc1(void *param)
2 {
3     while (1)
4         printf("Hello");
5 }
6 void TaskFunc2(void *param)
7 {
8     while (1)
9         printf("World");
10 }
11 int main(void)
12 {
13     TaskHandle_t xHandleTask1,xHandleTask2;
14
15 #ifdef DEBUG
16     debug();
17 #endif
18
19     prvSetupHardware();
20
21 // by yourself
22
23 //TaskHandle_t xHandleTask2;
24 // printf("Chang world!\r\n");
25 xTaskCreate(TaskFunc1, "Task1", 100, NULL, 1, &xHandleTask1);
26 xTaskCreate(TaskFunc2, "Task2", 100, NULL, 1, &xHandleTask2);
27
28
29 /* Start the scheduler. */
30 vTaskStartScheduler();
31
32 /* Will only get here if there was not enough heap space to create the
33 idle task. */
34 return 0;
35 }
```

移植规范与编程规范

两个数据类型

TickType_t

tick count 时钟中断次数的类型

BaseType_t

架构最高效的返回值类型

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:

- 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:

- 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

变量规范

前缀： p 指针 x 非标类型

c char pc char* 带指针

s short(16) l long(32)

u unsigned uc uint8_t

函数规范

前缀内部分 返回值类型 + 在哪个文件定义

prvSetupHardware():

prv 为前缀

demo.

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

第5节 例程分析

笔记

每个任务 \Rightarrow TCB + 任务控制块

xTaskCreate (Taskfunc, "Task1", 100, NULL, 1, &xHandleTask);
函数名称 名字 栈深 参数 优先级 指针(指向TCB)

xTaskCreateStatic(Taskfunc, "Task3", 100, NULL, 1, xTask3Stack,
(&xTask3));

xTask3Stack 任务分配的栈空间

```
1 #if ( configSUPPORT_STATIC_ALLOCATION == 1 )
2     TaskHandle_t xTaskCreateStatic( TaskFunction_t pxTaskCode,
3                                     const char * const pcName,
4                                     const uint32_t ulStackDepth,
5                                     void * const pvParameters,
6                                     UBaseType_t uxPriority,
7                                     StackType_t * const puxStackBuffer,
8                                     StaticTask_t * const pxTaskBuffer ) PRIVILEGED_FUNCTION;
9 #endif /* configSUPPORT_STATIC_ALLOCATION */
```

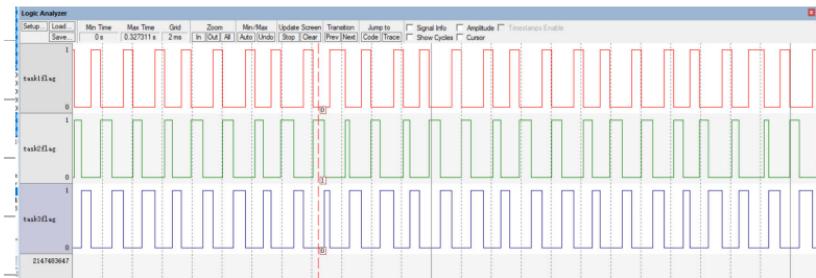
静态创建，TCB结构体与栈需要提前分配

并用静态创建需要添加宏定义 configSUPPORT_STATIC_ALLOCATION
还需要添加 vApplicationGetIdleTaskMemory() 函数

进一步实验

① freeRTOS 中高优先级的先执行，较高优先级的为高优先级

同优先级交错执行，可利用 keil 的逻辑分析仪去实现



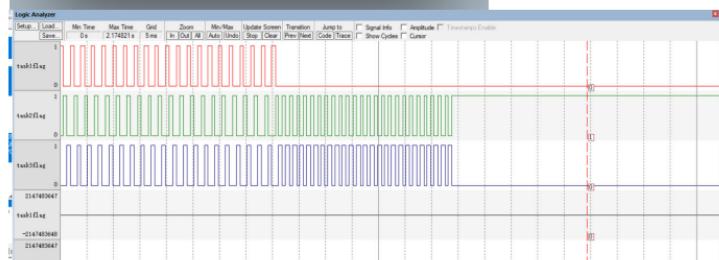
同优先级交错执行 同一时刻只有一个任务在执行

② 删除任务？ 利用任务ID TCB删除

```
1 void TaskFunc3(void *param)
2 {
3     int i = 0;
4     while (1)
5     {
6         task1flag = 0;
7         task2flag = 0;
8         task3flag = 1;
9         printf("333");
10
11         if ((i++) == 100)
12         {
13             vTaskDelete(xhandletask1);
14         }
15         if ((i) == 200)
16         {
17             vTaskDelete(NULL);
18         }
19     }
20 }
```

→ 杀掉任务

→ 自杀



⇒ 互斥划分

注意：

对于 xTaskCreateStatic() 静态创建任务

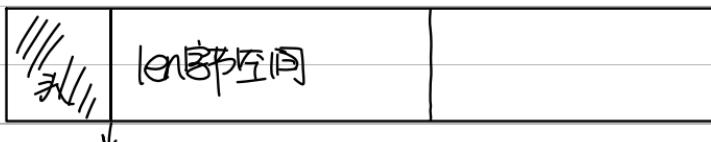
若令其回收必须记录其收回值

动态调度会将 TCB 的 handler 参数中

大两个值波动数及区间差异

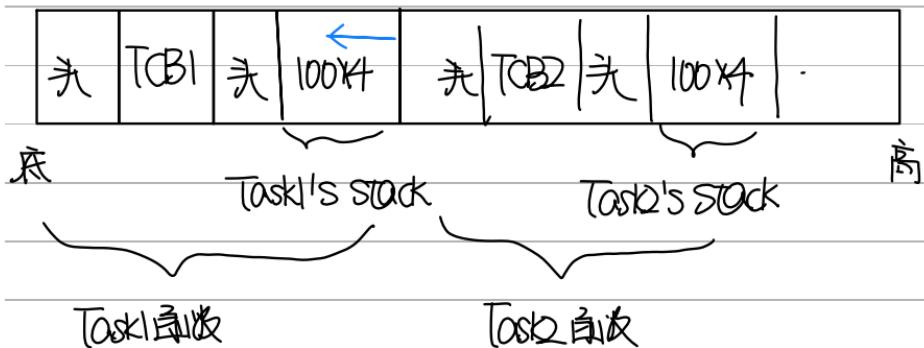
③ 每个任务都有自己的栈空间，运行时互不影响 → 不同任务可调用同一函数

④ malloc 分配内存角探究竟



buf 变量 buf = malloc(LEN)

头部中含有长波尾息，可用头部中 len 中 free 空间



精益生产会很头疼 TCBI …，导致系统崩溃

volatile 关键. 变量可变. 不利于编译器优化

```
1 void TaskFunc1(void *param)
2 {
3     volatile char buf[500];
4     while (1)
5     {
6         task1flag = 1;
7         task2flag = 0;
8         task3flag = 0;
9         printf("11");
10        for(int i=0;i<500;i++)
11            buf[i]=0;
12    }
13 }
14 }
```

分配木桶空间100字节

溢出就放为400B

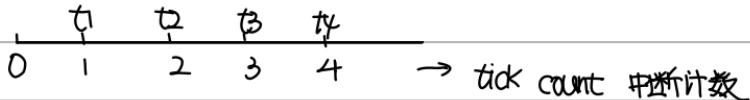
buf[500]并写入0, 溢出由

裁剪解决

第六节. 任务状态

任务状态理论.

① 任务切换基础 tick 中断



每隔1ms发生1次中断. 发生Tick中断后, 由TickIP0 判断是否
需要切换任务



freertosconfig 可配置任务时间

```
46: #define configCPU_CLOCK_HZ          ( ( unsigned long ) 72000000 )
47: #define configTICK_RATE_HZ |           ( ( TickType_t ) 1000 )
48: #define configMAX_PRIORITIES |        ( 3 )
49: #define configMINIMAL_STACK_SIZE |     ( ( unsigned short ) 176 )
```

基础 tick 1ms

同时也可指定每个任务执行多少个 tick

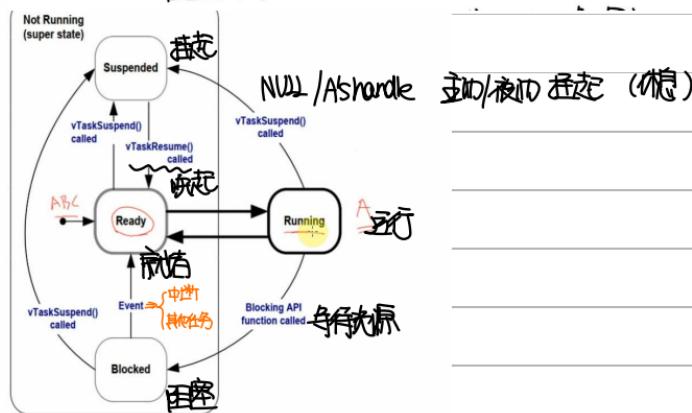
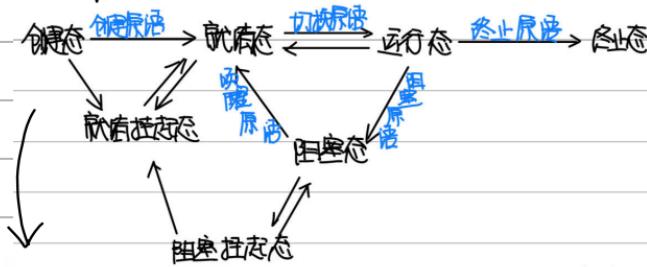
② 线程状态 Status

Running 执行

Ready 就绪

Blocked (等待某事) 阻塞

Suspended 挂起



任务管理 任务管理

就绪优先、挂起排队 ...

实验

挂起 vTaskSuspend()

唤醒 vTaskResume()

获取 tickcount xTaskGetTickCount()

测试时注意把 Keil 模拟的频率改为 8MHz

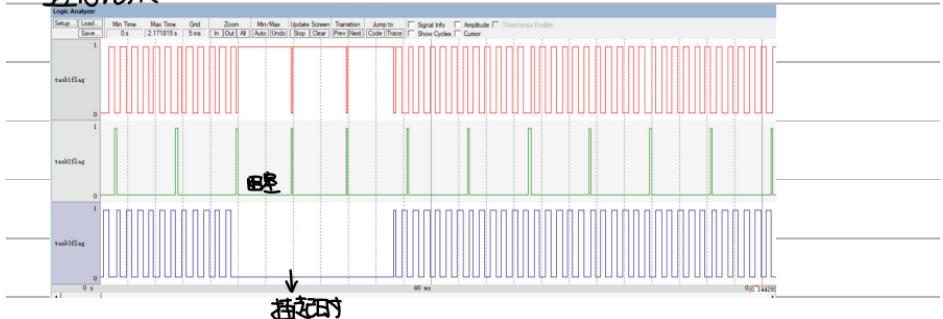
CPU DMH区 代码中 8MHz 9倍数得到 DMH区

代码 demo

```
1 void TaskFunc1(void *param)
2 {
3     TickType_t tStart = xTaskGetTickCount(); // 获取 tickcount
4     TickType_t t;
5     int flag = 0;
6     while (1)
7     {
8         t = xTaskGetTickCount(); // 实际记录
9         task1Flag = 1;
10        task2Flag = 0;
11        task3Flag = 0;
12        printf("111");
13        if ((t > tStart + 20) && (flag == 0)) // 20个 tick 后挂起
14        {
15            vTaskSuspend(xHandleTask3); // 任务3挂起
16            flag = 1;
17        }
18        if ((t > tStart + 50) && (flag == 1)) // 50个 tick 后唤醒
19        {
20            vTaskResume(xHandleTask3); // 任务3唤醒
21            flag = 2;
22        }
23    }
24 }
25 void TaskFunc3(void *param)
26 {
27     int i = 0;
28     while (1)
29     {
30         task1Flag = 0;
31         task2Flag = 0;
32         task3Flag = 1;
33         printf("333");
34     }
35 }
36 void TaskFunc2(void *param)
37 {
38     while (1)
39     {
40         task1Flag = 0;
41         task2Flag = 1;
42         task3Flag = 0;
43         printf("222");
44         vTaskDelay(10); // 这时 10 个 tick 会自动进入阻塞态等待被唤醒
45     }
46 }
47 }
```

附近一个

运行结果



vTaskDelay / vTaskDelayUntil

```
Task1func() {  
    while(1) {  
        do_something(); // 时间不定  
        vTaskDelay(N); // 延时 N tick  
    }  
}
```



任务执行的周期不同 Δt1≠Δt2

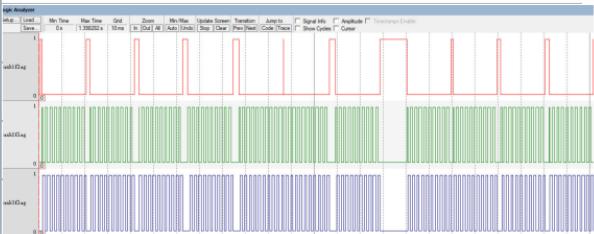
周期性执行? $\Delta t_1 = \Delta t_2 \Rightarrow vTaskDelay(n)()$ 亂數
 $vTaskDelayUntil(t_1, \Delta t)$; (开始返回) $t_1 + \Delta t$

任务

```

1 void TaskFunc1(void *param)
2 {
3     TickType_t tStart = xTaskGetTickCount();
4     int i = 0;
5     int j = 0;
6     while (1)
7     {
8         taskFlag = 1;
9         taskFlag = 0;
10        taskFlag = 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        vTaskDelay(20);
21    }
22 }
```

vTaskDelay() 测试



vTaskDelayUntil() 简介

```

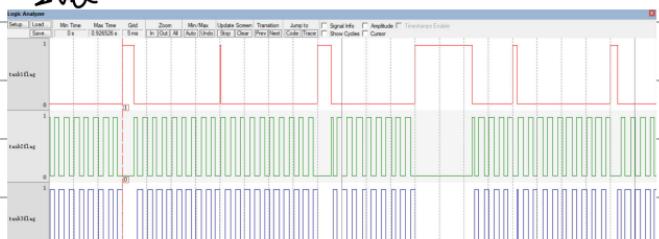
1 #define vTaskDelayUntil( pxPreviousWakeTime, xTimeIncrement )      \
2 {           ( void ) xTaskDelayUntil( pxPreviousWakeTime, xTimeIncrement ); \
3 }
```

① 直接用 $pxPre + xTime$ 后退出

② 自动更新 $pxPre \rightarrow pxPre + xTime$ (自动更新传入的针常量)

测试 Demo

现象：



代码

```

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 switch
21         vTaskDelay(20);
22 #else
23         vTaskDelayUntil(&tStart, 20);
24 #endif
25    }
26 }

```

→ 关 switch = 1 延迟 20ms

switch = 0 延迟 until(20)

空闲任务其钩子函数

Idle Task : 0 ⇒ 未执行，其会执行清理操作

Task1 : 1

Task2 : 2

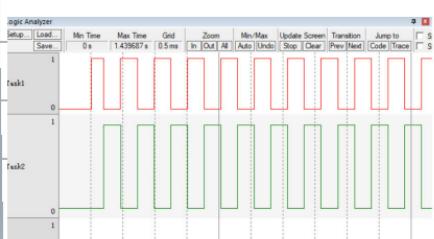
使用 vTaskDelete() 有三种参数 ↗ NULL 自然 需要 IdleTask 清理堆
示例：

```

1 void Taskfunc1(void *param)
2 {
3     TaskHandle_t xHandleTask2;
4     BaseType_t xReturn;
5     while (1)
6     {
7         Task1 = 1;
8         Task2 = 0;
9         Task3 = 0;
10        printf("111");
11        xReturn = xTaskCreate(Taskfunc2, "Task2", 100, NULL, 3, &xHandleTask2);
12        if (xReturn != pdPASS)
13        {
14            printf("Task1Create error");
15        }
16        vTaskDelete(xHandleTask2);
17    }
18 }
19 void Taskfunc2(void *param)
20 {
21     while (1)
22     {
23         Task1 = 0;
24         Task2 = 1;
25         Task3 = 0;
26         printf("222");
27         vTaskDelay(2);
28     }
29 }

```

正常运行但因为内容不写而创建失败

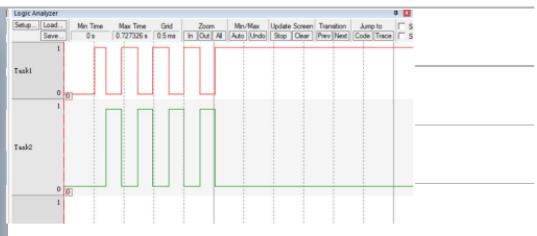


实验2

```

1 void TaskFunc1(void *param)
2 {
3     TaskHandle_t xHandleTask;
4     BaseType_t xreturn;
5     while (1)
6     {
7         Task1 = 1;
8         Task2 = 0;
9         Task3 = 0;
10        printf("111");
11        xreturn = xTaskCreate(TaskFunc2, "Task2", 1024, NULL, 3, &xHandleTask2);
12        if (xreturn != pdPASS)
13        {
14            printf("Task1Create Error");
15        }
16    }
17 }
18 void TaskFunc2(void *param)
19 {
20     while (1)
21     {
22         Task1 = 0;
23         Task2 = 1;
24         Task3 = 0;
25         printf("222");
26         vTaskDelete(NULL);
27     }
28 }

```



111222111222111222111222111Task1Create Error
111Task1Create Error111Task1Create Error111
111Task1Create Error111Task1Create Error111T
111Task1Create Error111Task1Create Error111Ta
111Task1Create Error111Task1Create Error111Ta

错误出错内存不足

IdleTask() 的创建在调度器中，vTaskStartScheduler()

钩子任务

我们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。钩子函数的作用有这些：

- 执行一些低优先级的、后台的、需要连续执行的函数
- 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。
- 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。

空闲任务的钩子函数的限制：

- 不能导致空闲任务进入阻塞状态、暂停状态
- 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。

修改空闲任务函数

taskc 中 portTASK_FUNCTION() 中配置 ApplicationIdleHook()

定义并实现函数

```

1 #if ( configUSE_IDLE_HOOK == 1 )
2 {
3     extern void vApplicationIdleHook( void );
4
5     /* Call the user defined function from within the idle task. This
6      * allows the application designer to add background functionality
7      * without the overhead of a separate task.
8      * NOTE: vApplicationIdleHook() MUST NOT, UNDER ANY CIRCUMSTANCES,
9      * CALL A FUNCTION THAT MIGHT BLOCK. */
10    vApplicationIdleHook();
11 }
12 #endif /* configUSE_IDLE_HOOK */

```

demo.

去 FreeRTOS config.h 定义
在 main 中增加钩子函数

```

1 void vApplicationIdleHook(void)
2 {
3     Task1 = 0;
4     Task2 = 0;
5     Task3 = 0;
6     ideoTask = 1;
7     printf("ideo Task");
8 }

```

同样自定义任务，但很显然没有调度

Task1 优先级为 0

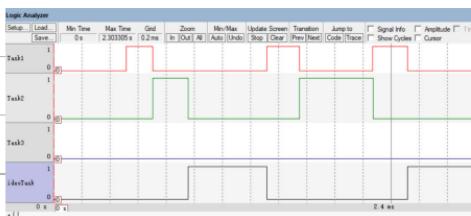
IdleTask 有机会执行 (回收内存)

同时添加钩子函数

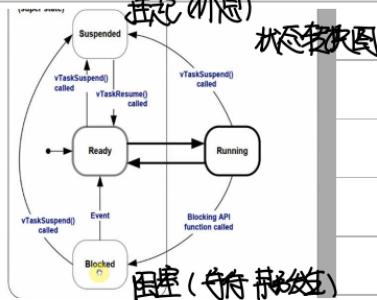
使 IdleTask 可执行别的内容

钩子函数没有进入多循环 (可跳出)

⇒ 放行结果



调度算法



挂起 (休息)

状态转换图

阻塞 (等待某操作)

事件 : {
 时间相关的事件
 同步事件
}

调度策略:

① 是否可抢占 configUSE_PREEMPTION 宏

为1： 可抢占 高优先级抢占低优先级

为0： 不可抢占 当前任务让出CPU后，高优先级才能使用(高优先级)
 (全局调度模式) 何插队, 会触发时间片轮转

② 可抢占条件下 configUSE_TIMESLICING 轮询同优先级是否轮流

为1： 时间片轮转

为0： 不轮流执行，一直执行直到主动放弃或被高优先级抢占

时间片轮转与主动放弃, 对事件

③ 可抢占且时间片轮转 configIDLE_SHOULD_YIELD

为1： 空闲任务低于用户任务，空闲任务轮询用户任务，只执行一次

为0： 空闲任务同用户任务，轮流执行

freertosconfig.h修改配置宏

底层

IdleTask()

{ while(1)

{ XXX

钩子);

#if: YIELD

调用 //角发调度 (主动调度)

#endif

}

否则会死锁，直至 tick 中打断形成调度

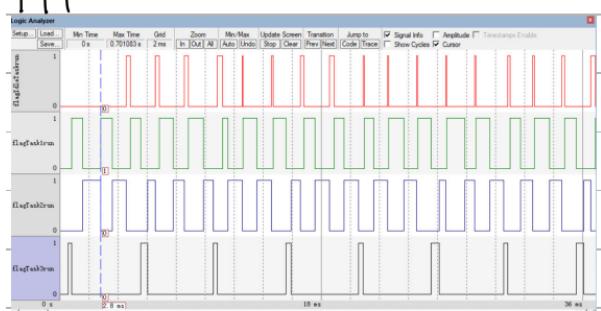
慎选：

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：

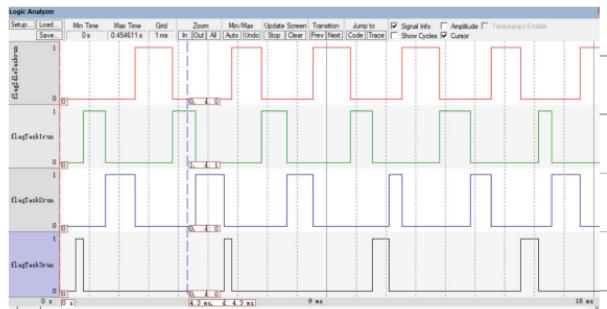
- A：可抢占+时间片轮转+空闲任务让步
- B：可抢占+时间片轮转+空闲任务不让步
- C：可抢占+非时间片轮转+空闲任务让步
- D：可抢占+非时间片轮转+空闲任务不让步
- E：合作调度

实现验证



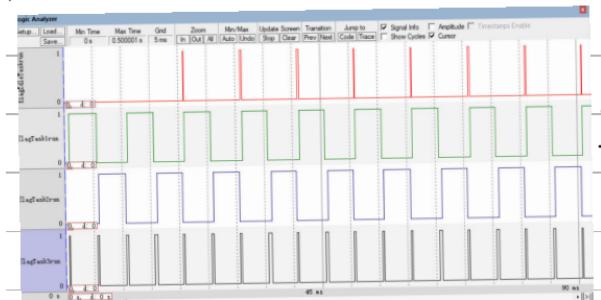
Task3 先执行，阻塞其他CPU
Task1 执行时间比短于 Task2
Task2 抢到了 Idle 很深
执行以用时长短

110



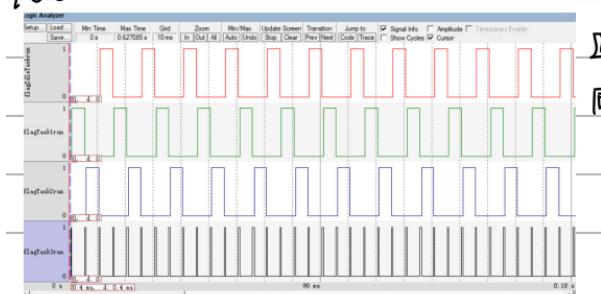
Task3 先执行 退出CPU
区别在于 Idle 不让步

101



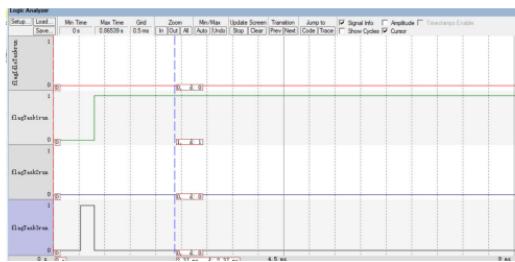
抢占全局锁只发生在
Task3 立即退出CPU后
Idle 让步但环不出门

100



区别在于 Idle 5 徒劳
同等待

0XX 合作 只让出口后才会发限流给抢占



Task3 江进后

Task1 一直霸占资源

测试 Demo.

```
● ● ●
1 void vTask3( void *pvParameters )
2 {
3     const TickType_t xDelay5ms = pdMS_TO_TICKS( 5UL );
4
5     /* 任务函数的主体一般都是无限循环 */
6     for( ;; )
7     {
8         flagIdleTaskrun = 0;
9         flagTask1run = 0;
10        flagTask2run = 0;
11        flagTask3run = 1;
12
13        /* 打印任务的信息 */
14        printf("T3\r\n");
15
16        // 如果不休眠的话，其他任务无法得到执行
17        vTaskDelay( xDelay5ms );
18    }
19 }
```

Task3 优先级为2

Task1, Task2, IdleTask

间隔50

第9节 同步与互斥

同步 进程工作的前后关系

互斥 对临界资源的访问，互进行

伪代码

```
01 void 抢厕所(void)
02 {
03     if (有人在用) 我睡一会儿; // blocked 阻塞
04     用厕所;
05     喂, 醒醒, 有人要用厕所吗;
06 }
```

同步示例 |

```
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: }
```

> 用 flagCalcEnd 标志位进行同步

缺陷 Task2 也会竞争资源(白白浪费CPU资源) 效率太低

什么叫异步?

(多选题)

- A. 你不知道它什么时候发生
- B. 中断是一种异步
- C. FreeRTOS的任务没有实现异步
- D. 异步, 就类似于中断

3. 任务间通信

```
41: prvSetupHardware();
42:
43: printf("Hello, world!\r\n");
44:
45: xTaskCreate(Task1Function, "Task1", 100, NULL, 1, &HandleTask1);
46: //xTaskCreate(Task2Function, "Task2", 100, NULL, 1, NULL);
47:
48: xTaskCreate(TaskGenericFunction, "Task3", 100, "Task 3 is running", 1, NULL);
49: xTaskCreate(TaskGenericFunction, "Task4", 100, "Task 4 is running", 1, NULL);
50:
51: /* Start the scheduler. */
52: vTaskStartScheduler();
53:
```

后果：Task3、Task4 的打印句不完整，task 中断后会很打断

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; 1
32:             vTaskDelay(1)| 互斥操作才可
33:         }
34:     }
35: }
```

缺陷：

task 中执行插入 Task3 执行但 flag 未置 |

Task4 同理，又会打扰到 Task3 (信号操作不可)

信号操作出现问题

总结

任务是线程，并且全局变量

可用全局变量共享

freeRTOS 实现同步机制

正确使用，共享

常用方式 例 利用 事件量，信号量，任务通知

第8节. 队列的使用

理论知识



队列结构体



创建队列 `XQueueGenericCreate()`, 参数：队列长度 & item大小

```
● ● ● 队列结构体
1 typedef struct QueueDefinition /* The old naming convention is used to prevent breaking kernel aware debuggers. */
2 {
3     int p_head;           /* Points to the beginning of the queue storage area. */
4     int p_writeTo;         /* Points to the free next place in the storage area. */
5 
6     union
7     {
8         QueueMessage_t q_items; /* Data required exclusively when this structure is used as a queue. */
9         SemaphoreData_t s_semaphore; /* Data required exclusively when this structure is used as a semaphore. */
10    } u;
11 
12     list_t t_tasksWaitingToSend; /* 写队列 */
13     list_t t_tasksWaitingToReceive; /* 读队列 */
14 
15     volatile QueueType_t uNumberOfItems; /* The number of items currently in the queue. */
16     volatile uint32_t uLength; /* The length of the queue defined as the number of items it will hold, not the number of bytes. */
17     volatile uint32_t uItemSize; /* The size of each item that the queue will hold. */
18 
19     volatile int32_t cReceived; /* Stores the number of items received from the queue (removed from the queue) while the queue was locked. Set to queueUNLOCKED when the queue is not locked. */
20     volatile int32_t cTransferred; /* Stores the number of items transmitted to the queue (added to the queue) while the queue was locked. Set to queueUNLOCKED when the queue is not locked. */
21 
22 #if ( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
23     uint8_t uStaticallyAllocated; /* Set to pdTRUE if the memory used by the queue was statically allocated to ensure no attempt is made to free the memory. */
24 #endif
25 
26 #if ( configQUEUE_SEMAPHORE == 1 )
27     struct QueueDefinition *pnSemaphoreContainer;
28 #endif
29 
30 #if ( configQUEUE_FACILITY == 1 )
31     QueueType_t uQueueNumber;
32     uint8_t ucQueueType;
33 #endif
34 } QueueDef;
```

本质上为一个环形缓冲区



队列(尾部)

```
/* 等同于xQueueSendToBack
 * 往队列尾部写入数据, 如果没有空间, 阻塞时间为xTicksToWait
 */
BaseType_t xQueueSend(
    QueueHandle_t xQueue,
    const void *pvItemToQueue,
```

```
        TickType_t xTicksToWait
    );
/* 往队列尾部写入数据, 如果没有空间, 阻塞时间为xTicksToWait
 */
BaseType_t xQueueSendToBack(
    QueueHandle_t xQueue,
    const void *pvItemToQueue,
    TickType_t xTicksToWait
);
```



pcWriteTo 指向 pcWriteTo + = itemSize

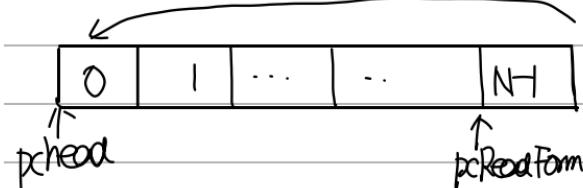
可设置 xTicksToWait 停待时间, 当队列满了以后

若停待时间为0, 返回错误

若停待时间不为0, 任务会自动进入 WaitingToSend 队列

队列(头部)

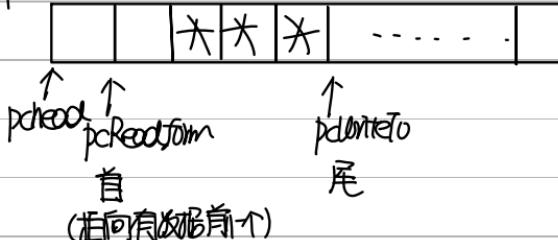
```
BaseType_t xQueueReceive( QueueHandle_t xQueue, → pBuf
    void * const pvBuffer, → buffer
    TickType_t xTicksToWait ); → 是阻塞等待, 阻塞在读缓冲区
BaseType_t xQueueReceiveFromISR(
    QueueHandle_t xQueue,
    void *pvBuffer,
    BaseType_t pxTaskWoken
);
```



phead 指向始读缓冲, 指向 buffer 首地址

读指针 pcReadForm 会改变, 初始化在上一次读位置 pcReadForm += itemSize

running中



每次增加 itemSize

写入队列

```
/*  
 * 往队列头部写入数据，如果没有空间，阻塞时间为xTicksTowait  
 */
```

```
BaseType_t xQueueSendToFront(  
    QueueHandle_t xQueue,  
    const void *pvItemToQueue,  
    TickType_t xTicksTowait  
);  
  
/*  
 * 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  
 */  
BaseType_t xQueueSendToFrontFromISR(  
    QueueHandle_t xQueue,  
    const void *pvItemToQueue,  
    BaseType_t *pxHigherPriorityTaskWoken  
);
```



$pReadFrom - = itemSize$

读写指针都指向同一个位置

多个任务有生产者唤醒权。①最高优先级 ②等待时间最长

队列的常规使用(队列实现与步骤)

① 同步的放进

xQueueCreate(队列长度, itemsize);

```
● ● ●
1 xQueueUARTHandle = xQueueCreate(1, sizeof(int));
2 if (xQueueUARTHandle == NULL)
3 {
4     printf("can not create queue\r\n");
5     return -1;
6 }
```

xQueueSend(xQueueHandle, &sum, portMAX_DELAY); //队列中写入数据

xQueueReceive(xQueueHandle, &val, portMAX_DELAY); //队列中读出数据

```
● ● ●
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     while (1)
17     {
18         flagCalcEnd = 0; //接收到任务时向
19         xQueueReceive(xQueueCalcHandle, &val, portMAX_DELAY); //没有读到数据会一直阻塞，不占用CPU资源
20         flagCalcEnd = 1; //接收到任务时向
21         printf("sum = %d\r\n", val);
22     }
23 }
```

↑ 读取队列数据
↓ 插入数据

② 异步的访问

创建队列锁 InitUARTLock() 写入数据表明可被访问 (寄里可执行)

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

主动放弃CPU

vTaskDelay() 阻塞

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 }
```

初识块状

(内存申请在堆区)

不存在函数依赖白板块

拆分块

并行块

```
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 }
```

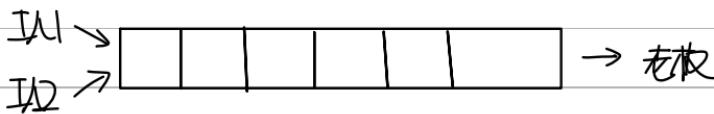
任务执行

在main函数中调用并行块 顺序为34 执行 TaskGenericFunction()

其它例应用 便入的为 &val 地址

故数据类型不定可直接构件

分解数据源



传播构件

```
typedef struct {  
    IDt eDataID;      // 利用ID分解不同源  
    int32_t DataValue;  
} Data
```

注：类似全局，只有读一次

传输大数据包

FreeRTOS 不用高复制，可用地址传递大数据



传输语句，接收语句，用语句访问 src2

利用指针 \Rightarrow {数据
地址 (另开辟一块新内存放入地址信息)}

注意

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：

- 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。
- 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。

队列与邮箱

邮箱 - 一个队列，长度只有1

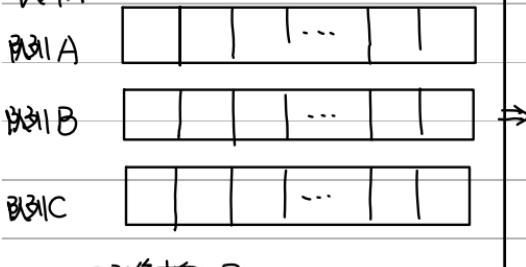
FreeRTOS的邮箱概念跟别的RTOS不一样，这里的邮箱称为“橱窗”也许更恰当：

- 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。

写：直接覆盖必成功

读：不会清除，除非从队列外必成功

队列：

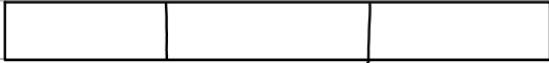


队列本身也是队列

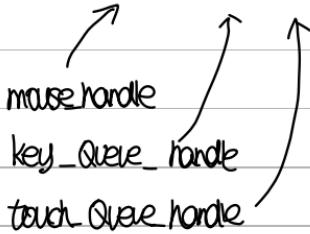
QueueSet . 队列ABC 长度为 A.length + B.length + C.length

Queue Set

①



②



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

写队列1次、就写队列条目1次、写队列N次、就写队列条目N次

反之读队列1次、读队列条目1次

过程：

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

④ Read Queue Set 返回 Queue (-N次)

用读 Queue (-N次)

注意

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

例1 Demo :

创建队列集合 xQueueCreateSet() 放入长度 返回 handle

建立 queue & queueSet 联系 xQueueAddToSet();

读 QueueSet() xQueueSelectFromSet()

main 程序:

```
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;
```

结果:

```
get Date: 0
get Date: -1
get Date: 1
get Date: -2
get Date: 2
get Date: 3
get Date: -3
get Date: 4
get Date: 5
get Date: -4
get Date: 6
get Date: 7
get Date: -5
get Date: 8
get Date: 9
get Date: -6
get Date: 10
get Date: 11
get Date: -7
get Date: 12
get Date: 13
get Date: -8
```

```
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 }
```

第九节：信号量的使用

简化学习



不使用信号量存储数据，只向表示某种资源的数目

二、

结构体

Value

① 创建结构体

② give/take value++ 或 --

Semaphore

三、

技术型信号量 创建时初始值可设

二进制信号量 创建时初始值为0

	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

对应的删除信号量 vSemaphoreDelete();

信号量本质上是个队列，有队列头/尾，无 buffer

value的保有 uxMessagesWaiting

value=0 时 { take() Blocked 直到可设定阻塞多久
give() 唤醒一个被阻塞的任务}

glue 5 take 函数：

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

take 可被阻塞时间

give 不被阻塞时间，失败返回错误信息 成功唤醒 (优先级高 同优先级待久)

串行使用

信号量优势

使用队列也可以实现同步，为什么还要使用信号量呢？

(多选题)

- A. 使用队列可以传递数据，数据的保存需要空间
- B. 使用信号量时不需要传递数据，更节省空间
- C. 使用信号量时不需要复制数据，效率更高

五、包含 "semphr.h" 头文件

注意西源宏 freeRTOSconfig.h 宏开关打开

向后兼容

```
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         xSemaphoreGive(xSemCalc); // give the semaphore if free
9         vTaskDelete(NULL); // TaskDelete() -> 若无此sum输出为100000 如何保证
10    }
11 }
12
13 void Task2Function(void * param) // 报错完壁挂 ??? 问题
14 {
15     while (1)
16     {
17         flagCalcEnd = 0; // prove the running state
18         xSemaphoreTake(xSemCalc, portMAX_DELAY); // get the semaphore if not: blocked(always)
19         flagCalcEnd = 1; // prove the running state
20         printf("sum = %d\n", sum);
21     }
22 }
```

驱动实现

```
1 void TaskGenericFunction(void * param)
2 {
3     while (1)
4     {
5         xSemaphoreTake(xSemUART, portMAX_DELAY); //先取锁是
6         printf("%s\r\n", (char *)param);
7         xSemaphoreGive(xSemUART); //使用完释放
8         vTaskDelay(1);
9     }
10 }
11
12
```

同步：先V后P

异步：先P后V

main()函数逻辑

```
1 prvSetupHardware();
2
3 xSemCalc = xSemaphoreCreateCounting(10, 0); //Create Semaphore
4 xSemUART = xSemaphoreCreateBinary(); //三重锁危险是
5 xSemaphoreGive(xSemUART); //表明串口可用
6
7 xTaskCreate(Task1Function, "Task1", 100, NULL, 1, &HandleTask1);
8 xTaskCreate(Task2Function, "Task2", 100, NULL, 1, NULL);
9
10 xTaskCreate(TaskGenericFunction, "Task3", 100, "Task 3 is running", 1, NULL);
11 xTaskCreate(TaskGenericFunction, "Task4", 100, "Task 4 is running", 1, NULL);
12
13 /* Start the scheduler. */
14 vTaskStartScheduler();
15
```

第十节 互斥量的使用

互斥量进阶讲解：

作用：保护临界资源

与二进制信号量对比：

解决优先级反转的问题

解决递归上锁/解锁的问题

```
1 //临界代码
2 void TaskGenericFunction(void * param)
3 {
4     while (1)
5     {
6         xSemaphoreTake(xSemaphoreUART, portMAX_DELAY); //Get semaphore
7         printf("%s\r\n", (char *)param);
8         xSemaphoreGive(xSemaphoreUART); //Give semaphore
9         vTaskDelay(1);
10    }
11 }
```

A → 执行 Get 信息

且在使用串口

此时 tick 到 B → Get 信息

在此阻塞

A 完成后实现唤醒 B

成功！

若 A → 执行 Get 信息

↑ tick 到 C → 解放信息
失败 同时使用串口
tick 到 D → Get 信息

关键问题：实现递归上锁/解锁（这个是并没有实现）

互斥量解决优先级反转

A: 1

B: 2

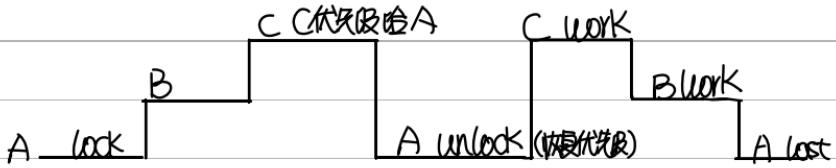
C: 3 A lock

C lock 失败

while(1)

C 的优先级高于 B，但 C 并不能执行

解决：优先级继承 Lock 先释放



C 优先级继承给A，使A高于B运行

递归上锁

```

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(); → 二次上锁，失败。递归锁
157:         xSemaphoreGive(xSemUART);
158:         vTaskDelay(1);
159:     }
160: }
```

解决：使用递归锁

Lock



A获得lock后，不可再取获得lock

unlock

lock与unlock要配对

普通：优先级继承

递归

递归锁：优先级继承 + 递归锁

普通 mutex 创建： xSemaphoreCreateMutex(void);

删除

vSemaphoreDelete() 递归也是神信号量

xSemaphoreGive() 放开

xSemaphoreTake() 解放

互斥锁创建：

xSemaphoreCreateRecursiveMutex()

xSemaphoreTakeRecursive() // Take

xSemaphoreGiveRecursive() // Give

互斥的使用：

线程创建后，无需手动 Give (缺省值为 1)

xSemaphoreCreateMutex() 需要手动开关

与信号量区别一致（常规使用）

优先级反转

```
1 int main(void)
2 {
3     prvSetupHardware();
4
5     /* 创建互斥量/二进制信号量 */
6     xLock = xSemaphoreCreateBinary(); // 创建的二进制信号量
7     xSemaphoreGive(xLock);
8
9     if (xLock != NULL)
10    {
11        /* 创建3个任务： LP,MP,HP(低/中/高优先级任务)
12        */
13        xTaskCreate(vLPTask, "LPTask", 1000, NULL, 1, NULL); // Last
14        xTaskCreate(vMPTask, "MPTask", 1000, NULL, 2, NULL); // second
15        xTaskCreate(vHPTask, "HPTask", 1000, NULL, 3, NULL); // first work!
16
17        /* 启动调度器 */
18        vTaskStartScheduler();
19    }
20    else
21    {
22        /* 无法创建互斥量/二进制信号量 */
23    }
24
25    /* 如果程序运行到了这里就表示出错了，一般是内存不足 */
26    return 0;
27 }
```

最优先级抢占调度

中级优先级调度

```

1 static void vLPTask(void *pvParameters) // make it to work firstly;
2 {
3     const TickType_t xTicksToWait = pdMS_TO_TICKS(10UL);
4     uint32_t i;
5     char c = 'A';
6     uint32_t number;
7     printf("LPTask start\r\n");
8
9     /* 无限循环 */
10    for (;;)
11    {
12        flagLPTaskRun = 1;
13        flagMPtaskRun = 0;
14        flagHPtaskRun = 0;
15
16        /* 获得互斥量/ 读制信号量 */
17        xSemaphoreTake(xLock, portMAX_DELAY); // get lock
18
19        /* 延时很久 */
20
21        printf("LPTask take the Lock for long time");
22        for (i = 0; i < 500; i++)
23        {
24            number = i;
25            number = number * 26;
26            flagLPTaskRun = 3;
27            flagMPtaskRun = 0;
28            flagHPtaskRun = 0;
29            printf("%c", c + number);
30        }
31        printf("\r\n");
32
33        /* 释放互斥量/ 读制信号量 */
34        xSemaphoreGive(xLock);
35
36        vTaskDelay(xTicksToWait);
37    }
38 }

```

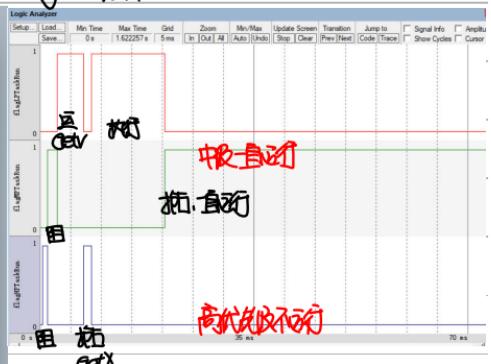
```

1 static void vMPTask(void *pvParameters)
2 {
3     const TickType_t xTicksToWait = pdMS_TO_TICKS(30UL); // define 30ms
4
5     flagLPTaskRun = 0;
6     flagMPtaskRun = 1;
7     flagHPtaskRun = 0;
8
9     printf("MPTask start\r\n");
10
11    /* |||LPTask, MPTask先运行 */
12    vTaskDelay(xTicksToWait); // delay 30ms
13
14    /* 无限循环 */
15    for (;;)
16    {
17        flagLPTaskRun = 0;
18        flagMPtaskRun = 1;
19        flagHPtaskRun = 0;
20    }
21 }

```

高优先级抢占

观察结果：



```

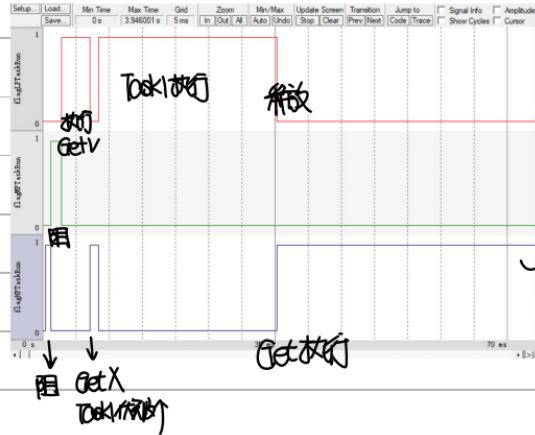
1 static void vHPTask(void *pvParameters)
2 {
3     const TickType_t xTicksToWait = pdMS_TO_TICKS(10UL); // define 10ms
4
5     flagLPTaskRun = 0;
6     flagMPtaskRun = 0;
7     flagHPtaskRun = 1;
8
9     printf("HPTask start\r\n");
10
11    /* 让LPTask先运行 */
12    vTaskDelay(xTicksToWait); // delay 10ms
13
14    /* 无限循环 */
15    for (;;)
16    {
17        flagLPTaskRun = 0;
18        flagMPtaskRun = 0;
19        flagHPtaskRun = 1;
20        printf("HPTask wait for Lock\r\n");
21
22        /* 获得互斥量/二进制信号量 */
23        xSemaphoreTake(xLock, portMAX_DELAY);
24
25        flagLPTaskRun = 0;
26        flagMPtaskRun = 0;
27        flagHPtaskRun = 1;
28
29        /* 释放互斥量/二进制信号量 */
30        xSemaphoreGive(xLock);
31    }
32 }

```

发生优先级反转 ⇒ 解决？ 优先级继承

只需将二进制信号量改为 mutex 即可

修改后界面



高优先级任务先于 Task2 执行，导致 Task2 反转

尽可能的实现更高优先级任务尽快处理 → freeRTOS 定时性

递归锁

互斥量同样没有实现，维持具体情况

```

1 void taskGenericFunction(void * param)
2 {
3     while (1)
4     {
5         xSemaphoreTake(xSemMFT, portMAX_DELAY);
6         printf("X\n");
7         xSemaphoreGive(xSemMFT);
8         vTaskDelay();
9     }
10 }
11
12 void TaskFunction(void * param)
13 {
14     vTaskDelay(10); // a injector to work
15     while (1)
16     {
17         while (1)
18             if(xSemaphoreTake(xSemMFT, 0)!=pdTRUE) // no wait take false: give by oneself and take again
19                 xSemaphoreGive(xSemMFT);
20             else
21                 break;
22     }
23     printf("X\n");
24     xSemaphoreGive(xSemMFT);
25     vTaskDelay(); // let other process can execute;
26 }
27 }
```

Task2 可以自己释放锁后再获得
一般的互斥没有实现能力有，由谁释放
→ 最终由谁摘出

递归锁实现了由谁持有，由谁释放（只能由拥有者释放）

同时递归锁也解决了很环创建及锁的问题

#define configUSE_RECURSIVE_MUTEX 1

```
1 void TaskGenericFunction(void * param)
2 {
3     while (1)
4     {
5         xSemaphoreTakeRecursive(xSemUART, portMAX_DELAY);
6         printf("RxData", (char *)param);
7         xSemaphoreGiveRecursive(xSemUART);
8         vTaskDelay(1);
9     }
10 }
11 void TaskFunction(void * param)
12 {
13     vTaskDelay(10); // let a intersector to work
14     while (1)
15     {
16         while(1);
17         if(xSemaphoreTakeRecursive(xSemUART, 0) == pdTRUE) // no wait take the value give by oneself and take again
18             xSemaphoreGiveRecursive(xSemUART);
19         else
20             break;
21     }
22 }
23
24 printf("RxData", (char *)param);
25 xSemaphoreGiveRecursive(xSemUART);
26 vTaskDelay(1); //let other process can execute;
27 }
28 }
```

混乱的互锁

可以确定 take 由谁释放

⇒ 最终正确输出

循环互锁

```
1 void TaskGenericFunction(void * param)
2 {
3     while (1)
4     {
5         int i=0;
6         xSemaphoreTakeRecursive(xSemUART, portMAX_DELAY);
7         printf("%s\n\n", (char *)param);
8         for (i = 0; i < 10; i++)
9         {
10             xSemaphoreTakeRecursive(xSemUART, portMAX_DELAY);
11             printf("%d in loop\n", i);
12             xSemaphoreGiveRecursive(xSemUART);
13         }
14         xSemaphoreGiveRecursive(xSemUART);
15         vTaskDelay(1);
16     }
17 }
```

} ⇒ 追踪上锁、并且解锁

第十一节 事件组

事件组理论

事件组 适用于 某事件 / 若干个事件中某个事件 / 若干个事件中所有事件



事件组结构体 EventGroupDef_t

```

1 typedef struct EventGroupDef_t
2 {
3     EventBits_t uxEventBits; // TickType 类型，每位对应一个事件，一位一个bit
4     List_t xTasksWaitingForBits; /* List of tasks waiting for a bit to be set. */
5     /* 内存分配在启动时 */
6     #if ( configUSE_TRACE_FACILITY == 1 )
7         UBaseType_t uxEventGroupNumber;
8     #endif
9
10    #if ( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
11        uint8_t ucStaticallyAllocated; /* Set to pdTRUE if the event group is statically allocated to ensure no attempt is made to free the memory. */
12    #endif
13 } EventGroup_t;

```

使用事件组：

① Create

xEventGroupCreate()

设置位：

② Set bit

xEventGroupSetBits()

$1 \ll 0 | 1 \ll 1 | 1 \ll 2$

③ Wait bit

xEventGroupWaitBits()

设置等待哪些事件
是否清零事件(1)
是否等待所有事件
是否阻塞

xEventGroupSync() 同步 该且子 bit(声明) 等待那些 bit(声明)

A, B, C 分别设置 Bit0, Bit1, Bit2

同时执行 Bit0 | Bit1 | Bit2



实现 A, B, C 同时退出 xEventGroupSync() 直数

事件组的使用：

事件组只起作用，数据保存用另想办法

为什么变量的定义，必须放在函数开头那里？不能放在代码之后？

(多选题)

- ✓ A. Keil默认支持的C
语言标准是
C89，必须这样做

- ✓ B. 如果是C99的话，变量的定义可以放在任何地方

- ✓ C. 可以在Keil中指定使用C99标准

注意：必须包含 eventGroup 且 将 C 项目由工程 直接添加

实验 demo.

main

```
1 prvSetupHardware();
2
3 xQueueCalcHandle = xQueueCreate(2, sizeof(int)); // main函数中创建队列
4 if (xQueueCalcHandle == NULL)
5 {
6     printf("can not create queue\r\n");
7 }
8 xEvent = xEventGroupCreate(); // Create a EventGroup
9
10 xTaskCreate(Task1Function, "Task1", 100, NULL, 1, &xHandleTask1);
11 xTaskCreate(Task2Function, "Task2", 100, NULL, 1, NULL);
12 xTaskCreate(Task3Function, "Task3", 100, NULL, 1, NULL);
13
14 /* Start the scheduler. */
15 vTaskStartScheduler();
```

⇒ 利用 Queue 保存数据

} 三个任务

Task3:

```
1 void Task3Function(void *param)
2 {
3     int val1, val2;
4
5     while (1)
6     {
7         xEventGroupWaitBits(xEvent, (1 << 0) | (1 << 1), pdTRUE, pdTRUE, portMAX_DELAY);
8         xQueueReceive(xQueueCalcHandle, &val1, portMAX_DELAY);
9         printf("val1 %d\r\n", val1);
10        xQueueReceive(xQueueCalcHandle, &val2, portMAX_DELAY);
11        printf("val2 %d\r\n", val2);
12    }
13 }
```

接收事件并打印

Task2

```

1 void Task2Function(void *param)
2 {
3     volatile int i = 0;
4     while (1)
5     {
6         for (i = 0; i < 100000; i++)
7             dec = dec - i;
8         xQueueSend(xQueueCalcHandle, &dec, portMAX_DELAY);
9         // set event 0(bit 0)
10        xEventGroupSetBits(xEvent, 1 << 0);
11    }
12 }

```

⇒累加，没有事件插入队列

Task1

```

1 void Task1Function(void *param)
2 {
3     volatile int i = 0;
4     while (1)
5     {
6         for (i = 0; i < 10000; i++)
7             sum = sum + i;
8         xQueueSend(xQueueCalcHandle, &sum, portMAX_DELAY);
9         // set event 0(bit 0)
10        xEventGroupSetBits(xEvent, 1 << 0);
11    }
12 }

```

⇒累加，设置事件插入队列

同步点：

xEventGroupSync() 指定等待哪些事件，待完成成功返回
main() 署名会清除对应的位

```

1 prvSetupHardware();
2
3 /* 创建任务组 */
4 xEventGroup = xEventGroupCreate( );
5
6 if( xEventGroup != NULL )
7 {
8     /* 创建3个任务：洗菜/生火/炒菜
9      */
10    xTaskCreate( vCookingTask, "task1", 1000, "A", 1, NULL );
11    xTaskCreate( vBuyingTask, "task2", 1000, "B", 2, NULL );
12    xTaskCreate( vTableTask, "task3", 1000, "C", 3, NULL );
13
14    /* 启动调度器 */
15    vTaskStartScheduler();
16 }
17 else
18 {
19     /* 无法创建事件组 */
20 }

```

} A.B.C 三个人完成三个不同的任务

Task1, Task2, Task3 三个任务

```
1 static void vCookingTask( void *pvParameters )
2 {
3     const TickType_t xTicksToWait = pdMS_TO_TICKS( 100UL );
4     int i = 0;
5
6     /* 无限循环 */
7     for( ;; )
8     {
9         /* 做自己的事 */
10        printf("%s is cooking %d time....\r\n", (char *)pvParameters, i);
11
12        /* 表示我做好了， 还要等别人都做好 */
13        xEventGroupSync(xEventGroup, COOKING, ALL, portMAX_DELAY); //实现同步
14
15        /* 别人也做好了， 开饭 */
16        printf("%s is eating %d time....\r\n", (char *)pvParameters, i++);
17        vTaskDelay(xTicksToWait);
18    }
19 }
20
21 static void vBuyingTask( void *pvParameters )
22 {
23     const TickType_t xTicksToWait = pdMS_TO_TICKS( 100UL );
24     int i = 0;
25
26     /* 无限循环 */
27     for( ;; )
28     {
29         /* 做自己的事 */
30         printf("%s is buying %d time....\r\n", (char *)pvParameters, i);
31
32         /* 表示我做好了， 还要等别人都做好 */
33         xEventGroupSync(xEventGroup, BUYING, ALL, portMAX_DELAY); //实现同步
34
35         /* 别人也做好了， 开饭 */
36         printf("%s is eating %d time....\r\n", (char *)pvParameters, i++);
37         vTaskDelay(xTicksToWait);
38    }
39 }
40
41 static void vTableTask( void *pvParameters )
42 {
43     const TickType_t xTicksToWait = pdMS_TO_TICKS( 100UL );
44     int i = 0;
45
46     /* 无限循环 */
47     for( ;; )
48     {
49         /* 做自己的事 */
50         printf("%s is do the table %d time....\r\n", (char *)pvParameters, i);
51
52         /* 表示我做好了， 还要等别人都做好 */
53         xEventGroupSync(xEventGroup, TABLE, ALL, portMAX_DELAY); //实现同步
54
55         /* 别人也做好了， 开饭 */
56         printf("%s is eating %d time....\r\n", (char *)pvParameters, i++);
57         vTaskDelay(xTicksToWait);
58    }
59 }
```

第十二讲 任务通知

通知介绍

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”：



多对一的关系

任务通知无需创建结构体

任务TCB结构体中合之内部

对象直接接收别人的通知

TCB

```
#if ( configUSE_TASK_NOTIFICATIONS == 1 )
    volatile uint32_t uNotifyValue; configTASK_NOTIFICATION_ARRAY_ENTRIES;
    volatile uint8_t uNotifyState; configTASK_NOTIFICATION_ARRAY_ENTRIES;
#endif
```

其他任务可向其写入值，且不会阻塞
要成功，要么失败（都不等待）
(没有等待链表)

本任务读状态可以被阻塞等待响应
要么立即返回

通知原生

xTaskNotifyGive()

uTaskNotifyTake()



~Give() val++;

~Take() 返回 val 且根据 { val=0 pdTRUE
val-- pdFalse }

xTaskNotify();

xTaskNotifyWait();



xTaskNotify(xTaskNotify, ulValue, eAction);

xt eAction

eNoAction 仅增加不修改 value (设置 state 不设置 value)

eSetBits : val | ulValue 泊置 val 某些位为 1

enhancement 提高 value +1

setValueWithoutOverWrite() 不覆盖 第一次发未响应但还未取又发送第二次

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

xTaskNotifyWait (ulBitsToClearOnEntry, ulBitsToClearOnExit, &puNotificationValue, xTime)

ulBitsToClearOnEntry . 入口处清除某些位 val &= ~ (bits) 清零位

ulBitsToClearOnExit . 出口处清除某些位 val &= ~ (bits) 清零位

puNotificationValue . 指向 val 出口清零后的 value

TxWait 等待

返回 polloss 失败成功

轻量级信号量

使用任务完成和实现轻量级信号量，初值为0，最大值为0且不能指定
初始值与最大值

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 uiTaskNotifyTake(BaseType_t xClearCountOnExit, TickType_t xTicksToWait);

区别

polloss 成功 其他失败

成功：返回成功 失败：返回0

demo

```

● ● ●
1 void TaskFunction(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(xSemaphore); // 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         val = ulTaskNotifyTake(pdFALSE, portMAX_DELAY); // blocked to wait
24         //xSemaphoreTake(xSemaphore, portMAX_DELAY); // get the semaphore if not: blocked(always)
25         FlagCalcEnd = 1;
26         printf("sum = %d value = %d i = %d\n", sum, val, i++);
27     }
28 }

```

无释放语句
轻量级
存于TCB内部

轻量级队列

实现的队列只能存放一个任务数据

九内存的队列

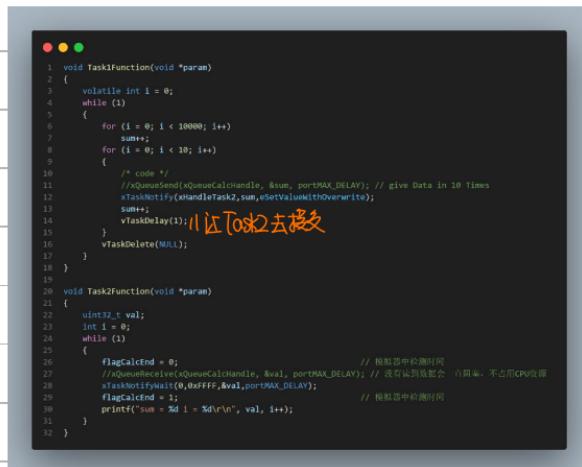
`send()` < 覆盖 覆盖 val
 不覆盖 发送失败

⇒ 用 eAction 处理

异步对化.

队列		使用任务通知实现队列
创建	<pre>QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);</pre>	无
发送	<pre>BaseType_t xQueueSend(QueueHandle_t xQueue, const void *pvItemToQueue, TickType_t xTicksToWait);</pre>	<pre>BaseType_t xTaskNotify(TaskHandle_t xTaskToNotify, uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t *puhNotificationValue, eNotifyAction eAction);</pre> <p>(发) ⇒ 不会发送阻塞 没有优先级</p>
接收	<pre>BaseType_t xQueueReceive(QueueHandle_t xQueue, void * const pvBuffer, TickType_t xTicksToWait);</pre>	<pre>BaseType_t xTaskNotifyWait(uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t *puhNotificationValue, TickType_t xTicksToWait);</pre> <p>(收) ⇒ 可以阻塞</p>

demo:



```

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,esrValueWithOverwrite);
13            sum++;
14            vTaskDelay(1); //让 Task2 去接收
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         flagGetCldnd = 0; // 检测器中检测时间
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         flagGetCldnd = 1; // 检测器中检测时间
30         printf("sum = %d \r\n", val, i++);
31     }
32 }
```

轻量级条件阻塞

事件中含 value 可设置具体值，对每个事件



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

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

任务多态的实现 设置 value 每一位 execBit 设置某些位

B → 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 *puNotificationValue, TickType_t xTicksToWait);

demo代码

后果

UART #1

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

Task1 / Task2

```
1 void Task1Function(void *param)
2 {
3     volatile int i = 0;
4     while (1)
5     {
6         for (i = 0; i < 1000000; i++)
7             sum = sum + i;
8         xQueueSend(xQueueCalcHandle, &sum, portMAX_DELAY);
9         // set event @(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 @(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         // xEventGroupSetBits(xEvent, (1 << 0) | (1 << 1), pdTRUE, pdTRUE, portMAX_DELAY);
8         xTaskNotify(xHandleTask1, 0, bits, portMAX_DELAY);
9         if ((bits&(0x0001))>=0x0001) →自己去判断是否为0或1的
10        // 0000 0001 & 0000 0001
11        vTaskDelay(20);
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        vTaskDelay(20);
17        if ((val1==val2)&&(bits>0))
18            vTaskDelay(20);
19        printf("No Get All Bits Get only 0x%04x\r\n",bits);
20    }
21 }
```



⇒ 應處(bits & 0x0B)

第十三章 定时器的使用

理论介绍

三要素

超时时间

函数

单次触发还是多次触发

XTimerCreate()

```

/* 使用动态分配内存的方法创建定时器
 * pcTimerName: 定时器名字, 用处不大, 尽在调试时用到
 * xTimerPeriodInTicks: 周期, 以Tick为单位
 * uxAutoReload: 类型, pdTRUE表示自动加载, pdFALSE表示一次性
 * pvTimerID: 回调函数可以使用此参数, 比如分辨是哪个定时器
 * pxCallbackFunction: 回调函数
 * 返回值: 成功则返回TimerHandle_t, 否则返回NULL
 */
TimerHandle_t xTimerCreate( const char * const pcTimerName, // 定时器名
                           const TickType_t xTimerPeriodInTicks, // 周期
                           const UBaseType_t uxAutoReload, // 自动重载
                           void * const pvTimerID, // 回调函数参数
                           TimerCallbackFunction_t pxCallbackFunction ); // 回调函数

```

回调函数类型

```

1  /*
2   * Defines the prototype to which timer callback functions must conform.
3   */
4  typedef void (* TimerCallbackFunction_t)( TimerHandle_t xTimer );

```

定义Timer结构体

```

1  /* The definition of the timers themselves. */
2  typedef struct tmrTimerControl {
3
4      const char * pcTimerName;
5      ListItem_t xTimerListitem;
6      TickType_t xTimerPeriodInTicks;
7      void * pvTimerID; // 可见定时器
8      TimerCallbackFunction_t pxCallbackFunction;
9      #if ( configUSE_TRACE_FACILITY == 1 )
10         UBaseType_t uxTimerNumber;
11     #endif
12     uint8_t ucStatus;
13 } xTIMER;

```

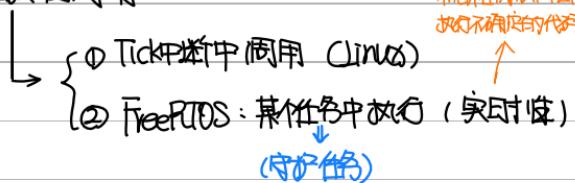
启动定时器 xTimerStart(xTimer, xTicksToWait);

t=TickO xTimerPeriodInTicks



定时器触发后，回调函数被调用

不能在内核中运行
或不确定的代码



注: 回调函数可以使用导致阻塞的函数, 但建议应尽快完成, 否则会阻碍

其它定时器函数

守护任务: 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 函数

欲使用 timer, 开启开关、定义相关回调, 启动调度器时会自动创建**任务**

```
0: #if configUSE_TIMERS == 1
1:
2:     #ifndef configTIMER_TASK_PRIORITY
3:         #error If configUSE_TIMERS is set to 1 then configTIMER_TASK_PRIORITY must a
4:     #endif /* configTIMER_TASK_PRIORITY */
5:
6:     #ifndef configTIMER_QUEUE_LENGTH
7:         #error If configUSE_TIMERS is set to 1 then configTIMER_QUEUE_LENGTH must al
8:     #endif /* configTIMER_QUEUE_LENGTH */    15
9:
10:    #ifndef configTIMER_TASK_STACK_DEPTH
11:        #error If configUSE_TIMERS is set to 1 then configTIMER_TASK_STACK_DEPTH mus
12:    #endif /* configTIMER_TASK_STACK_DEPTH */
13:
14: #endif /* configUSE_TIMERS */
```



自己编写的任务函数要使用定时器, 要通过定时器命令

队列与守护任务交互

并优先级 configTIMER_TASK_PRIORITY

并命令队列 configTIMER_QUEUE_LENGTH

用户程序调用定时器函数

xTimerDelete
xTimerStart/xTimerStartFromISR
xTimerStop/xTimerStopFromISR
xTimerReset/xTimerResetFromISR
xTimerChangePeriod/
xTimerChangePeriodFromISR

→ Timer command queue

FreeRTOS 内核源码
void prvTimerTask(...)
{
 for (;;) {

 /* 读队列 */
 xQueueReceive();
 /* 处理队列 */

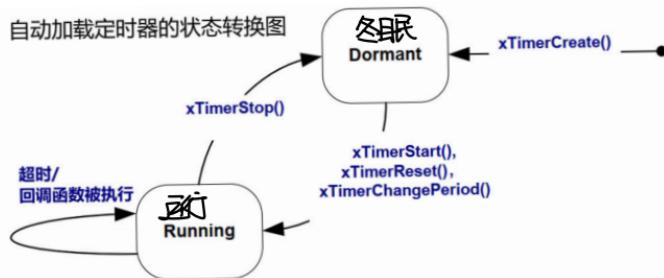
 }
}

xTickToWait 防止滴答就等待(阻塞)

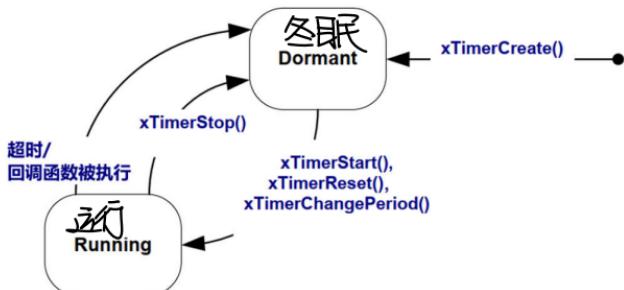
守护任务执行回调函数, 同时管理多个定时器 (创建的定时器由守护任务管理)

定时器状态转换图

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



定时器的一般使用

工程：

{ task1 . timer 每隔一段时间去打印
task2 . 优先级不同的守护任务

```
/* 1. 工程中 */
添加 timerSC

/* 2. 配置文件FreeRTOSConfig.h中 */
#ifndef 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"
```

→ FreeRTOSconfig

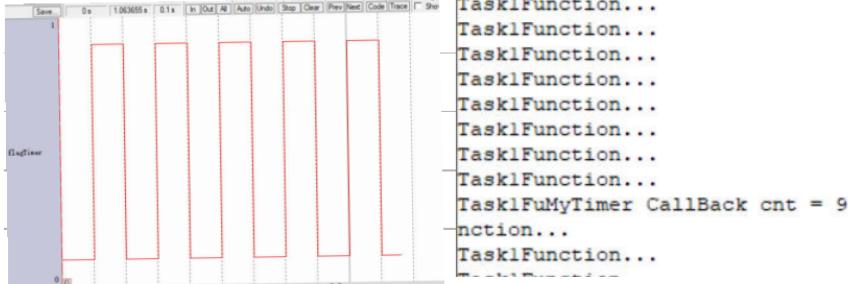
demo1

```
● ○ ●
1 prvSetupHardware();
2
3 xTimer1Handle = xTimerCreate("Timer1", 100, pdTRUE, NULL, TimerCallbackFunc); // 定时器
4
5 xTaskCreate(Task1Function, "Task1", 100, NULL, 1, &xHandleTask1); // 任务
6
7 /* Start the scheduler. */
8 vTaskStartScheduler();
9
10 /* Will only get here if there was not enough heap space to create the
11 idle task. */
12 return 0;
```

回调函数及任务 |

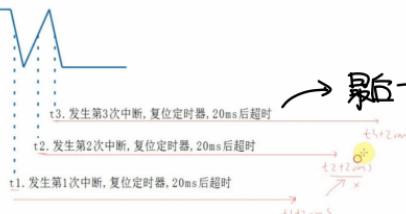
```
● ○ ●
1 void TimerCallbackFunc(TimerHandle_t xTimer)
2 {
3     static int cnt = 0;
4     flagTimer=!flagTimer;
5     printf("MyTimer CallBack cnt = %d \r\n", cnt++);
6 }
7 void TaskFunction(void *param)
8 {
9     xTimerStart(xTimerHandle,0); // start timer : timer will be managed by TimerServer
10    while (1)
11    {
12        /* code */
13        printf("Task1Function... \r\n");
14    }
15 }
```

结果



利用定时器撤销

机械波震动抖动： 定时器抖动



发生了三次中断, 但是最终只有一个会执行处理函数 —> 成为抖动

demo代码

main() 逻辑：

```
1 prvSetupHardware();
2
3 printf("Hello, world!\r\n");
4
5 KeyInit();
6 KeyInitInit(); // 初始化
7
8 xMyTimerHandle = xTimerCreate("mytimer", 2000, pdFALSE, NULL, MyTimerCallbackFunction); // 只触发一次
9
10 xTaskCreate(Task1Function, "Task1", 100, NULL, 1, &xHandleTask1);
11 // xTaskCreate(Task2Function, "Task2", 100, NULL, 1, NULL);
12
13 /* Start the scheduler. */
14 vTaskStartScheduler();
15
16 /* Will only get here if there was not enough heap space to create the
17 idle task. */
18 return 0;
```

结束语

KEY与EXTI初始化

```
● ● ●
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时钟在GPIO中，主要有两个功能：中断引脚的选择以及复用引脚映射
11 void KeyInitInit(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复用时钟 */
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         EXTI_Mode_TypeDef EXTI_Mode;
24         EXTI_Trigger_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_IROChannel;
39         u8 NVIC_IROChannelPreemptionPriority;
40         u8 NVIC_IROChannelSubPriority;
41         FunctionalState NVIC_IROChannelCmd;
42     } NVIC_InitTypeDef;
43     */
44     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //中断分组：芯片级：执行一次
45     NVIC_InitStructure.NVIC_IROChannel = EXTI_IROChannel; // 使能外部中断所在的通道
46     NVIC_InitStructure.NVIC_IROChannelPreemptionPriority = 0; // 抢占优先级
47     NVIC_InitStructure.NVIC_IROChannelSubPriority = 0; // 任务优先级
48     NVIC_InitStructure.NVIC_IROChannelCmd = ENABLE; // 使能外部中断通道
49     NVIC_Init(&NVIC_InitStructure); // 初始化
50 }
```

EXTI中断处理

```
● ● ●
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         /* 使用定时器清除中断 */
8         xTimerReset(xMyTimerHandle, 0); /* Tcur + 2000 //复位定时器，最终执行代码在定时器中，且完成时后执行*/
9
10     }
11     EXTI_ClearITPendingBit(EXTI_Line0); // 清除中断
12 }
```

定时器函数

```

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 }

```

第十四节 中断管理

内嵌API总结

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
	xTaskNotifyGive	vTaskNotifyGiveFromISR
	xTaskNotify	xTaskNotifyFromISR
	xTimerStart	xTimerStartFromISR
软件定时器(software timer)	xTimerStop	xTimerStopFromISR
	xTimerReset	xTimerResetFromISR
	xTimerChangePeriod	xTimerChangePeriodFromISR

会很快执行

不会发生阻塞(直接返回)

任务中函数:

可以快速退出并执行其他任务

可发起调度(调度级别最高)

ISR中函数:

可以唤醒并记录是否完成

不能引发调度(先执行)

(元组作为参数传入)

中断的处理流程

CPU → 固定地址执行代码 (中断向量) (硬件实现)

执行代码 / 保存现场

| 分辨中断, 调用处理函数 (ISR)

| 恢复现场

ISR在内核中周期，应尽快完成以确保实时性

而若处理很复杂 { ISR 做清理、记录工作，触发某任务
任务更复杂的处理 }

ISR的优先级高于任何任务

ISR中需要记录是否调度，并自行发起调度（设置嵌套置）

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

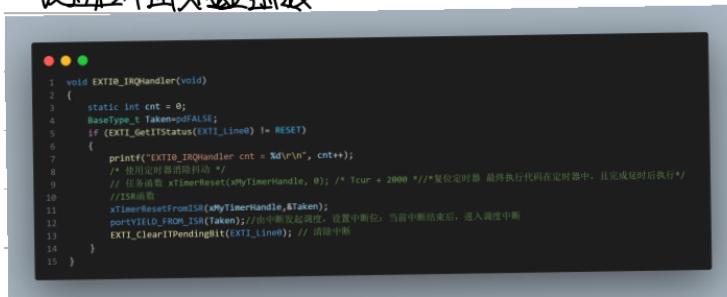
    for (i = 0; i < N; i++)
    {
        xQueueSendToBackFromISR(..., &xHigherPriorityTaskWoken); /* 被多次调用 */
    }

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

由中断发起调度（中断执行有任务优先级高）

发起调度也不足立即调度，当前中断仍在运行（设置一个优先级低的调度中断）

改进后中断处理函数



```
1 void EXTI9_5_IRQHandler(void)
2 {
3     static int cnt = 0;
4     BaseType_t xTaken = pdFALSE;
5     if (EXTI_GetITStatus(EXTI_Line0) != RESET)
6     {
7         printf("EXTI9_5_IRQHandler cnt = %d\n", cnt++);
8         /* 使用定时器清除启动 */
9         // 任务函数 xTimerReset(xMyTimerHandle, 0); /* Tcur + 2000 */ // 复位定时器 最终执行代码在定时器中，且完成时后执行*/
10        // ISRA函数
11        xTimerResetFromISR(xMyTimerHandle, xTaken);
12        portYIELD_FROM_ISR(xTaken); // 中断发起调度，设置中断位：当该中断结束后，进入调度中断
13        EXTI_ClearITPendingBit(EXTI_Line0); // 清除中断
14    }
15 }
```

* 中断处理完，需要自己发起调度

第十五节 资源管理

访问临界资源 (互斥的访问)

任务A, 任务B 可使用临界资源, 则访问前先禁止任务调度
(使用前禁止调度)

任务A, 中断C 可使用临界资源, 则访问前, 先关闭中断

注: 中断的优先级高于任务

暂停调度器

uxSchedulerSuspended ++ 为1时 禁止调度

uxSchedulerSuspended 为0时 可发起任务调度

通过 vTaskSuspendAll() 函数 \Rightarrow 停止任务调度

通过 vTaskResumeAll() 函数 \Rightarrow 恢复任务调度

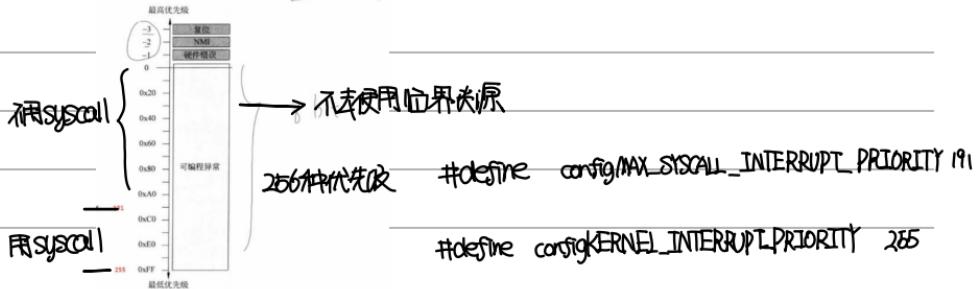
```
vTaskSuspendScheduler();
```

/* 访问临界资源 */

```
xTaskResumeScheduler();
```

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)

任务中屏蔽中断 : taskENTER_CRITICAL() / taskEXIT_CRITICAL()

ISR中屏蔽中断 : taskENTER_CRITICAL_FROM_ISR(),
taskEXIT_CRITICAL_FROM_ISR()

任务中屏蔽demo.

```
/* 在任务中，当前时刻中断是使能的  
 * 执行这句代码后，屏蔽中断  
 */  
taskENTER_CRITICAL(); //屏蔽  
  
/* 访问临界资源 */  
  
/* 重新使能中断 */  
taskEXIT_CRITICAL(); //用完就关
```

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  
     */  
    //记录当前状态  
  
    /* 执行这句代码后，屏蔽中断  
     */  
    uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR(); //屏蔽返回值  
  
    /* 访问临界资源 */  
  
    /* 恢复中断状态 */  
    taskEXIT_CRITICAL_FROM_ISR(uxSavedInterruptStatus); //恢复中断状态  
    /* 现在，当前ISR可以被更高优先级的中断打断了 */  
}
```

之前 中断状态
① 禁止
② 开启

有困惑？

之后 恢复
① 禁止
② 开启

第十六章 调试优化

调试

① 打印

FreeRTOS 使用了 microlib 实现了 printf()

需自己实现 int fputc (int ch, FILE *f);

② 断言

void assert (expression); expression 为假，中断

FreeRTOS 用 configASSERT() 这个宏（默认为空）

可自己定义

#define configASSERT(x) if (!x) while (1); //自己设置

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

③ Trace 宏（跟踪）

④ Hook 函数（自定义钩子）

Malloc Hook 函数（堆溢出）

Configure_MALLOC_FAILED_HOOK 为 1

调用 vApplicationMallocFailedHook() 函数

自己提供实现

捕获出 Hook 函数

方法 1. 任务切换出去前，对现场进行保存，在此时可

判断是否达到峰值（高效但不精确）

```

48: #if ( configCHECK_FOR_STACK_OVERFLOW == 1 ) && ( portSTACK_GROWTH < 0 )
49: /* Only the current stack state is to be checked. */
50: #define taskCHECK_FOR_STACK_OVERFLOW()
51:
52: {
53: /* Is the currently saved stack pointer within the stack limit? */
54: if( pxCurrentTCB->pxTopOfStack <= pxCurrentTCB->pxStack )
55: {
56: vApplicationStackOverflowHook( TaskHandle_t pxCurrentTCB, pxCurrentTCB->pcTaskName );
57: }
58: }
59:
60: #endif /* configCHECK_FOR_STACK_OVERFLOW == 1 */

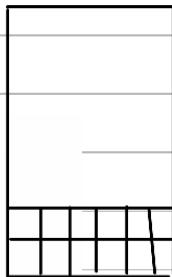
```

判断溢出

方法2：（金属为0xA5）

从向上判断是否有连续的A5

(16个Byte不足金属0xA5 认为溢出)



→ 溢出

切换任务时 检查是否溢出

优化：

查看优化信息 → 用于优化

冗余校空闲.

```

1 static configSTACK_DEPTH_TYPE prvTaskCheckFreeStackSpace() const uint8_t * pucStackByte )
2 {
3     uint32_t ulCount = 0U;
4
5     while( *pucStackByte == ( uint8_t ) taskSTACK_FILL_BYTE ) 0xA5
6     {
7         pucStackByte += portSTACK_IS_ALIGNMENT();
8         ulCount++; → 找到头部的零填充
9     }
10
11     ulCount /= ( uint32_t ) sizeof( StackType_t ); /*lint le901 Casting is not redundant on smaller architectures. */
12
13     return ( configSTACK_DEPTH_TYPE ) ulCount;
14 }

```

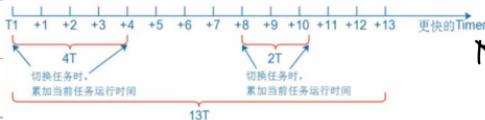
获取空闲栈大小

查找空间 vTaskGetStackHighWaterMark()

统计任务CPU运行时间



不精确



每隔时间t发生中断

count++

切换任务时记录 count 值

并读出 Timer 值 Count + Timer

更精确

在这段时间里：task1的CPU占用率是 $(4+2)/13=46\%$

累加 tick 时间即可计算 CPU 占有率

问题在于 tick 不一定精确 \Rightarrow 更快的定时器

任务调度策略与统计信息

获取任务统计信息

vTaskList():

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
PoISEM1	R	0	145	11
PoISEM2	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

获取任务运行信息

vTaskGetRunTimeStats()

void vTaskGetRunTimeStats(signed char *pcWriteBuffer);

可读信息格式如下：

任务名	任务运行时间	运行时间百分比
PoISEM1	994	<1%
PoISEM2	23246	1%
GenQ	194479	16%
MuLow	3690	<1%
Rec3	229450	18%
CNT1	242720	19%
PeakL	94	<1%
CNT_INC	165	<1%
CNT2	243166	20%
SUSP_RX	243192	20%
IDLE	55	<1%

\Rightarrow 模块去提供更好的

底层接口 API GetCount()

自底向上

硬件定时器 TIME

$$\begin{aligned}
 \text{频率计算: } CK_{CNT_OV} &= CK_{CNT} / (ARR+1) \\
 &= CK_{PSB} / (PSB+1) / (ARR+1)
 \end{aligned}$$

定时器宏配置

```

1 #include "stm32f10x_lib.h"
2
3 // 定时器配置
4 #define SYS_FREQ          (72000000) // 系统频率
5 #define TIMx              TIM1      // 目标定时器
6 #define TIMx_Period        (100-1) // 定时器的周期，单位: us
7 #define TIMx_Prescaler    (1024-1) // 定时器的分频系数，值范围: 0~65535
8 #define TIMx_PreLoadValue (SYS_FREQ/1000000*TIMx_Period/TIMx_Prescaler) // 定时器预分频系数，范围: 0~65535
9 #define TIMx_Div           (TIMx_Prescaler)
10 #define TIMx_Clk_Enable() RCC_APB1PeriphClockCmd( RCC_APB1Periph_TIM3, ENABLE ) // 目标定时器的时钟使能
11 // 定时器使能
12 #define TIMx_IRQ          TIM3_IRQn // 目标中断
13 #define TIMx_NVIC_PrePriority (0) // 中断的抢占优先级等级
14 #define TIMx_NVIC_SubPriority (1) // 中断的子优先级等级
15
16 #define TIMx_IRQHandler     TIM3_IRQHandler
17

```

定时器初始化:

```

1 void TimerInit(void)
2 {
3     /* 1. 使能时钟 */
4     TIMx_Clk_Enable(); // RCC_APB1PeriphClockCmd( RCC_APB1Periph_TIM3, ENABLE )
5
6     // CK_CNT_OV = CK_PSB / (PSB+1) / (ARR+1) 时钟频率计算
7     /*
8     CK_PSB=DIV为1, CK_PSB=72MHz(72 000 000Hz)
9     若多周期为CK_Period=10us，则频率将变为(1/CK_Period)*10^-6
10    CK_PSB = (PSB+1)*(ARR+1)*(1/(CK_Period)*10^-6)
11    CK_PSB*CK_Period=(PSB+1)*(ARR+1)*10^6
12    故而PSB+1 = CK_PSB*(CK_Period/(ARR+1))/1 000 000(us)
13    */
14
15     /* 2. 配置时钟时基结构体的成员，设置定时器参数 */
16     gTIMx_TimeBase_Init.TIM_Period      = TIMx_PRE_LOAD_VALUE; // 设置重载装载值 ARR 1024-1
17     gTIMx_TimeBase_Init.TIM_Prescaler   = TIMx_Prescaler; // 设置分频系数
18     gTIMx_TimeBase_Init.TIM_ClockDivision = TIMx_Div; // 设置分频系数为1:不分频
19     gTIMx_TimeBase_Init.TIM_CounterMode = TIM_CounterMode_Up; // 设置计数方式: Up
20
21     /* 3. 调用库函数初始化定时器 */
22     TIM_TimeBaseInit( TIMx, &gTIMx_TimeBase_Init );
23
24     /* 4. 配置中断的属性 */
25     gTIMx_NVIC_Init.NVIC IRQChannel      = TIMx_IRQn; // 选择中断
26     gTIMx_NVIC_Init.NVIC IRQChannelSubPriority = TIMx_NVIC_SubPriority; // 设置中断的子优先级
27     gTIMx_NVIC_Init.NVIC IRQChannelPreemptionPriority = TIMx_NVIC_PrePriority; // 设置中断的抢占优先级
28     gTIMx_NVIC_Init.NVIC IRQChannelCmd    = ENABLE; // 使能中断
29
30     /* 5. 初始化中断 */
31     NVIC_Init( &gTIMx_NVIC_Init );
32
33     /* 6. 选择定时器的中断触发方式 */
34     TIM_ITConfig( TIMx, TIM_IT_Update, ENABLE ); // 选择为更新触发中断，即向上计数到溢装载值或者向下计数到0触发中断
35
36     /* 7. 启动定时器 */
37     TIM_Cmd( TIMx, ENABLE );
38 }

```

定时器中断触发

```
1 void TIMx_IRQHandler(void)
2 {
3     if( TIM_GetITStatus(TIMx, TIM_IT_Update) == SET ) // 判断是否是更新触发中断
4     {
5         test_cnt = !test_cnt; //每次中断:反转test_cnt
6         g_timer_cnt++;
7         TIM_ClearITPendingBit( TIMx, TIM_IT_Update ); // 清除中断
8     }
9 }
```

再看

```
1 #define configGENERATE_RUN_TIME_STATS 1 //宏开关
2 #define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS TimerInit //启动延时值
3 #define portGET_RUN_TIME_COUNTER_VALUE TimerGetCount //读取计数值 > 自实现
```

开启开关后，实现此内自实现即可

统计函数使用

```
1 void Task1Function(void * param)
2 {
3     volatile int i = 0;
4
5     //xTimerStart(xMyTimerHandle, 0);
6
7     while (1)
8     {
9         //printf("Task1Function ...\\r\\n");
10        //vTaskList(pcWriteBuffer);
11        vTaskGetRunTimeStats(pcWriteBuffer);
12        printf(pcWriteBuffer);
13        vTaskDelay(5000);
14    }
15 }
```

第十七章 总结

main() {

 while(1);

}

 func1()

{

随机挂 PTOS

 → while(1),

}

 func2()

{

 while(2);

}

① 创建任务 TCB结构体，执行函数，栈

② A —— B

通信、同步、共享

队列、队列集、信号量、互斥量、新旧任务切换

↓
(环形缓冲区)

③ 线程间(内部) 展示会 关中断、先往多线程

freertos实践