FreeRTOS学习笔记

开发板：正点原子STM32F103战舰

**目录**

[1 基础知识 2](#_Toc161818365)

[2 同步和互斥 5](#_Toc161818366)

[3 队列 6](#_Toc161818367)

[4 信号量 8](#_Toc161818368)

[5 互斥量 8](#_Toc161818369)

[6 事件组 9](#_Toc161818370)

[7 任务通知 10](#_Toc161818371)

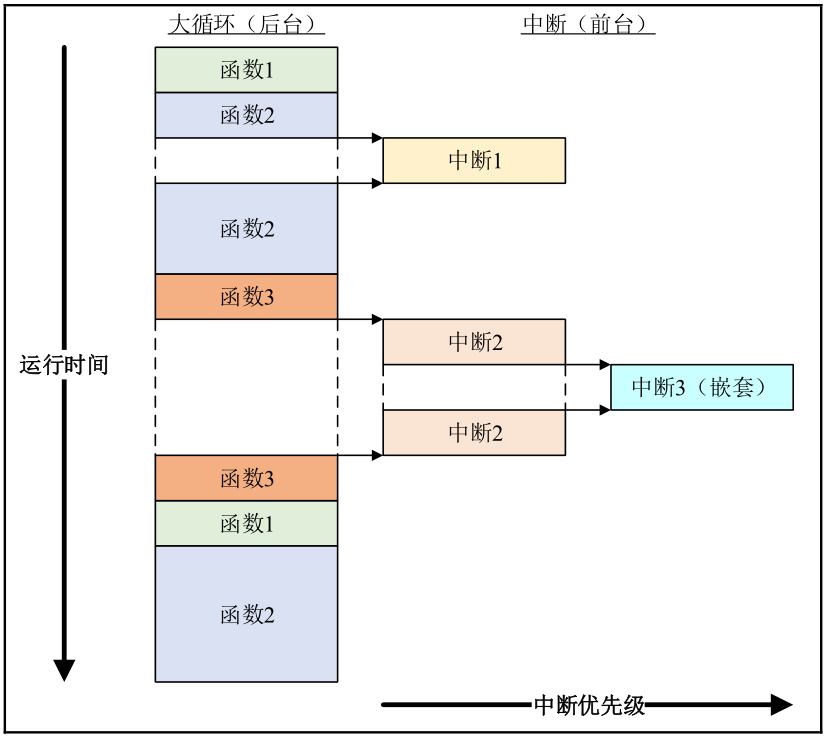
[8 FreeRTOS中断管理 11](#_Toc161818372)

[9 列表和列表项 14](#_Toc161818373)

[10 系统启动流程 14](#_Toc161818374)

# 1 基础知识

FreeRTOS为免费的实时操作系统，任务和任务管理是RTOS的核心，使用RTOS的目的就是为了使用RTOS的多任务管理能力。FreeRTOS使用的是多任务系统，与裸机的单任务系统相比，实时性得到了提高。

 图示

描述已自动生成

**单任务（前后台）系统 多任务系统**

1. **单任务系统**

一般是在main函数中使用一个大循环，在循环中调用相应的函数以处理相应的事务，这个大循环可以视为程序的前后台，前台为各种中断的中断服务函数，后台为大循环中的函数。因为大循环中的函数没有优先级，只能顺序执行，所以不论事务多紧急，都只能顺序执行各个函数。虽然可使用中断服务函数处理紧急事务，但在大工程项目中，这种单任务系统就会显得力不从心。

1. **多任务系统**

FreeRTOS中任务就是一个函数，原型为：void ATaskFunction( void \*pvParameters )。这个函数不能有返回值，函数内部尽量使用局部变量，每个任务都有自己的栈，任务的局部变量会存放在自己的栈里。从宏观上看可以同时执行多个任务，其实CPU在同一时刻只能处理一个任务，但是多任务系统的任务调度器会根据相应算法进行分配CPU的使用权，优先级相同的任务可以轮流运行一段极短的时间，即时间片调度，所以宏观角度就是同时执行多个任务。执行任务时，高优先级的任务可以抢占低优先级的任务，即抢占式调度。

1. **任务优先级**

通过配置FreeRTOSConfig.h中的宏configMAX\_PRIORITIES，可将每个任务分配一个0~(configMAX\_PRIORITIES-1)的任务优先级，优先级数值越大，优先级越高。configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION定义为1时，STM32则使用硬件方法计算下一个要运行的任务，STM32是32位CPU，所以configMAX\_PRIORITIES不能超过32，且为了节省硬件资源，应合理地将宏 configMAX\_PRIORITIES定义为满足应用需求的最小值。

1. **任务状态**

FreeRTOS 中任务存在四种任务状态，分别为运行态、就绪态、阻塞态和挂起态。

1. **运行态**：正在被运行的任务就处于运行态；
2. **就绪态**：已经能够被执行的任务就处于就绪态；
3. **阻塞态**：一个任务因延时一段时间或等待外部事件发生时就处于阻塞态，处于阻塞态的任务通常都有一个阻塞的超时

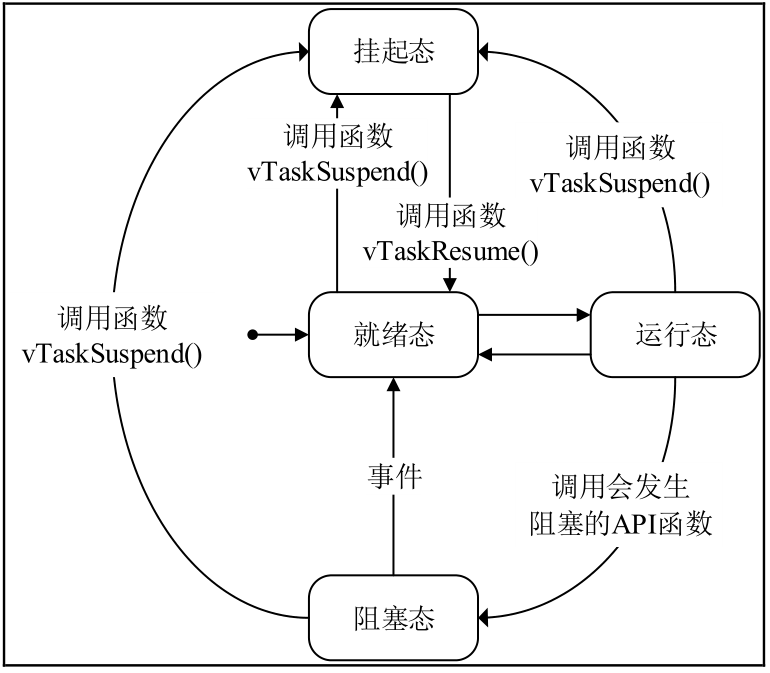
时间，在任务阻塞达到或超过这个超时时间后，即使任务等待的外部事件还没有发生，任务的阻塞态也会被解除；在等待事件的过程中不消耗CPU资源。

1. **挂起态**：与阻塞态一样，处于挂起态的任务也无法被运行，通过函数 vTaskSuspend()和vTaskResums()进入和退出挂

起态。vTaskSuspend()传入NULL时表示将自己挂起，vTaskResums()函数只能由别的任务操作。实际开发中，挂起状态用得不多。

1. **任务控制块**

FreeRTOS中的每一个已创建任务都包含一个任务控制块，任务控制块是一个结构体变量，用于存储任务的属性。结构体包含很多成员变量，可通过FreeRTOSConfig.h 配置文件中的配置项宏定义进行裁剪。当使用静态方式创建任务时，需要用户自行分配一块内存，作为任务的栈空间。使用动态方式创建任务时，系统则会自动从系统堆中分配一块内存，作为任务的栈空间。



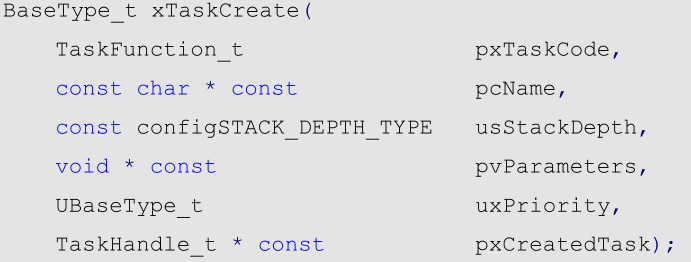
**任务状态转换图**

1. **常用API函数汇总**

动态方式创建任务：xTaskCreate() 静态方式创建任务：xTaskCreateStatic() 删除任务：vTaskDelete()

使用动态方式创建任务时需要将宏configSUPPORT\_DYNAMIC\_ALLOCATION配置为1，此函数创建的任务会立刻进入就绪态，函数原型如下。xTaskCreateStatic()函数使用静态方式创建任务，任务的任务控制块以及任务的栈空间所需的内存

需要由用户分配提供，首先需要将宏configSUPPORT\_STATIC\_ALLOCATION 配置为1，任务创建完成后直接进入就绪态。

 图形用户界面, 文本, 应用程序

描述已自动生成

参数介绍如下

|  |  |
| --- | --- |
| **参数** | **描述** |
| pxTaskCode | 指向任务函数的指针 |
| pcName | 任务名，最大长度为 configMAX\_TASK\_NAME\_LEN |
| uxPriority | 任务优先级，最大值为(configMAX\_PRIORITIES-1) |
| **动态创建** | |
| usStackDepth | 任务堆栈大小，单位：字（注意，单位不是字节） |
| pvParameters | 传递给任务函数的参数 |
| pxCreatedTask | 任务句柄，任务成功创建后，会返回任务句柄。任务句柄就是任务的任务控制块 |
| **静态创建** | |
| ulStackDepth | 任务堆栈大小，单位：字（注意，单位不是字节） |
| pvParameters | 传递给任务函数的参数 |
| puxStackBuffer | 任务栈指针，内存由用户分配提供 |
| pxTaskBuffer | 任务控制块指针，内存由用户分配提供 |

示例：（通常使用动态方式创建任务）

1. **静态创建任务**

#define START\_TASK\_PRIO 1 /\* 任务优先级 \*/

#define START\_STK\_SIZE 128 /\* 任务堆栈大小 \*/

StackType\_t StartTaskStack[START\_STK\_SIZE]; /\* 任务堆栈 \*/

StaticTask\_t StartTaskTCB; /\* 任务控制块 \*/

TaskHandle\_t StartTask\_Handler; /\* 任务句柄 \*/

void start\_task(void \*pvParameters); /\* 任务函数 \*/

StartTask\_Handler = xTaskCreateStatic((TaskFunction\_t )start\_task, /\* 任务函数 \*/

(const char\* )"start\_task", /\* 任务名称 \*/

(uint32\_t )START\_STK\_SIZE, /\* 任务堆栈大小 \*/

(void\* )NULL, /\* 传递给任务函数的参数 \*/

(UBaseType\_t )START\_TASK\_PRIO, /\* 任务优先级 \*/

(StackType\_t\* )StartTaskStack, /\* 任务堆栈 \*/

(StaticTask\_t\* )&StartTaskTCB); /\* 任务控制块 \*/

1. **动态创建任务**

#define START\_TASK\_PRIO 1 /\* 任务优先级 \*/

#define START\_STK\_SIZE 128 /\* 任务堆栈大小 \*/

TaskHandle\_t StartTask\_Handler; /\* 任务句柄 \*/

void start\_task(void \*pvParameters); /\* 任务函数 \*/

xTaskCreate((TaskFunction\_t )start\_task, /\* 任务函数 \*/

(const char\* )"start\_task", /\* 任务名称 \*/

(uint16\_t )START\_STK\_SIZE, /\* 任务堆栈大小 \*/

(void\* )NULL, /\* 传入给任务函数的参数 \*/

(UBaseType\_t )START\_TASK\_PRIO, /\* 任务优先级 \*/

(TaskHandle\_t\* )&StartTask\_Handler); /\* 任务句柄 \*/

注意，静态创建任务时，还需要用户提供用于提供空闲任务和软件定时器服务任务（如果启用了软件定时器）内存的两个回调函数，这两个回调函数分别为函数 vApplicationGetIdleTaskMemory()和函数vApplicationGetTimerTaskMemory()。具体代码参考正点原子例程。创建空闲任务回调函数示例如下：

static StackType\_t IdleTaskStack[configMINIMAL\_STACK\_SIZE]; /\* 空闲任务任务堆栈 \*/

static StaticTask\_t IdleTaskTCB; /\* 空闲任务控制块 \*/

void vApplicationGetIdleTaskMemory(StaticTask\_t \*\*ppxIdleTaskTCBBuffer, /\* 任务控制块内存 \*/

StackType\_t \*\*ppxIdleTaskStackBuffer, /\* 任务堆栈内存 \*/

uint32\_t \*pulIdleTaskStackSize) /\* 任务堆栈大小 \*/

{

\*ppxIdleTaskTCBBuffer = &IdleTaskTCB;

\*ppxIdleTaskStackBuffer = IdleTaskStack;

\*pulIdleTaskStackSize = configMINIMAL\_STACK\_SIZE;

}

1. **删除任务**

函数 vTaskDelete()用于删除任务，被删除的任务将被从就绪态任务列表、阻塞态任务列表、挂起态任务列表和事件列表中移除，首先要将宏INCLUDE\_vTaskDelete配置为 1。此函数只有一个参数xTaskToDelete，为待删除任务的任务句柄，为NULL时表示将自己删除。空闲任务会负责释放被删除任务中由系统分配的内存，但是由用户在任务删除前申请的内存，则需要由用户在任务被删除前提前释放，否则将导致内存泄露，空闲任务的优先级为0。

1. **挂起任务**

函数vTaskSuspend()用于挂起任务，只有一个参数xTaskToSuspend，为待挂起任务的任务句柄；恢复被挂起的任务使用函数 vTaskResume()，此函数页只有一个参数xTaskToResume，为待恢复任务的任务句柄。任务挂起不支持嵌套，即无论任务被挂起多少次，只要恢复一次，此任务就会恢复到就绪态。

1. **栈的作用**

ARM架构的CPU中存在15个寄存器：r0~r15，其中r13（SP栈），r14（LR返回地址，即下一条指令的地址），r15（PC当前指针）。常用的5个汇编指令：

1. LDR r0, [addr] /\* 将地址addr处的值读取到寄存器r0中，LDR是读4个字节 \*/
2. STR r0, [addr] /\* 将r0寄存器中的值写入到地址addr处 \*/
3. ADD r0,r1,r2 /\* r0 = r1 + r2 \*/
4. PUSH {r3, LR} /\* 将r3和LR两寄存器的值写入到内存中sp指针指向的位置 LR为r14，写入内存时按寄存器标

号大小排序后写入，先写LR，再写r3\*/

1. POP {r3,pc} /\* 从内存中sp指针指向的位置读数据到r3和pc寄存器中，同样根据寄存器标号大小进行赋值，

低地址取出的数据赋值给低标号的寄存器，高地址取出的数据给高标号的寄存器 \*/

CPU运行的时候就是从内存中读取并如上指令，执行函数时会进行进栈和出栈操作，当函数执行过程中被中断打断后，应该保存现场，将被打断瞬间所有CPU寄存器的值保存到内存中的栈里，所以任务需要有独立的栈。任务切换时，所有的寄存器都要保留；假如函数A调用函数B时需要使用r0，r1寄存器进行参数传递，此时就不需要保存r0，r1寄存器；当函数A运行过程中被硬件中断打断，硬件会帮助保存一部分寄存器的值，剩下的用到的寄存器要手动保存。

1. **时间片**

FreeRTOS使用定时器产生固定间隔的中断，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，相同优先级的任务就是根据时间片轮流执行的。

1. **两个Delay函数**

任务中使用延时函数后就会进入阻塞态，FreeRTOS中有两个Delay函数：vTaskDelay()和vTaskDelayUntil()。

vTaskDelay()：等待指定个数的时间片才能变为就绪状态。

vTaskDelayUntil()：等待到指定的绝对时刻，才能变为就绪态。

1. **空闲任务**

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。空闲任务的优先级为0，不能阻碍用户任务的运行。空闲任务要么处于就绪态，要么处于运行态，永远不会阻塞。空闲任务会负责释放被删除任务中由系统分配的内存，但要确保空闲任务有机会执行，否则无法释放被删除任务的内存。

1. **空闲任务的钩子函数**

实现该钩子函数前需要将宏configUSE\_IDLE\_HOOK配置为1，再实现vApplicationIdleHook 函数即可。空闲任务每执行

1. 就会调用一次钩子函数。钩子函数的作用为：
2. 执行一些低优先级的、后台的、需要连续执行的函数。
3. 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。
4. 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。

钩子函数的限制：

1. 不能导致空闲任务进入阻塞状态、暂停状态。
2. 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。
3. **任务相关函数汇总**

|  |  |  |  |
| --- | --- | --- | --- |
| 1. **函数** | **描述** | **函数** | **描述** |
| uxTaskPriorityGet() | 获取任务优先级 | vTaskPrioritySet() | 设置任务优先级 |
| uxTaskGetSystemState() | 获取所有任务的状态信息 | vTaskGetInfo() | 获取单个任务的状态信息 |
| xTaskGetApplicationTaskTag() | 获取任务Tag | xTaskGetCurrentTaskHandle() | 获取当前任务的任务句柄 |
| xTaskGetHandle() | 获取指定任务的任务句柄 | xTaskGetIdleTaskHandle() | 获取空闲任务的任务句柄 |
| uxTaskGetStackHighWaterMark() | 获取任务的任务栈历史剩余最小值 | eTaskGetState() | 获取任务状态 |
| pcTaskGetName | 获取任务名 | xTaskGetTickCount() | 获取系统时钟节拍计数器的值 |
| xTaskGetTickCountFromISR() | 中断中获取系统使用节拍计数器的值 | xTaskGetSchedulerState() | 获取任务调度器状态 |
| uxTaskGetNumberOfTasks() | 获取系统中任务的数量 | vTaskList() | 以表格形式获取所有任务的信息 |
| vTaskGetRunTimeStats() | 获取任务的运行时间等信息 | vTaskSetApplicationTaskTag() | 设置任务 |
| TagSetThreadLocalStoragePointer() | 设置任务的独有数据记录数组指针 | GetThreadLocalStoragePointer() | 获取任务的独有数据记录数组指针 |

要使用如上函数，需要先将相关的宏配置为1，如将INCLUDE\_uxTaskPriorityGet配置为1，则使能uxTaskPriorityGet()函数。

注：

# 2 同步和互斥

举例说明：A和B准备上厕所，A上完厕所后B才能上厕所，这就是同步；A上厕所的时候B不能进去上厕所，这就是互斥；A上完厕所后告诉B，B再去上厕所，这就是通过同步实现互斥。再举例：团队里，A写完报表后，B才能拿着报表去汇报，这就是同步；A使用会议室时，B也想用，但必须等待A用完才能使用，这就是互斥；A使用完会议室后告诉B，B就可以用会议室了，这就是通过同步实现互斥。同步和互斥通常放在一起讲。能实现同步和互斥的内核方法有很多，如：任务通知、队列、事件组、信号量、互斥量等。

1. **队列**：队列里面可以放任意数据，可以放多个数据。任务、ISR都可以向队列中放入数据和读取数据。

图形用户界面, 文本

描述已自动生成

1. 事件组：事件组用于广播事件或事件的组合发生了，不能用于传递数据。一个事件用1bit表示，1表示发生了，0表示

没发生。当事件发生后，等待这个事件或事件组合的任务就会被唤醒。

图形用户界面, 文本, 应用程序

描述已自动生成

1. **信号量：**信号量的核心是计数值，任务或ISR释放信号量时，计数值加1；任务或ISR获得信号量时，计数值减1。

图形用户界面, 文本, 应用程序

描述已自动生成

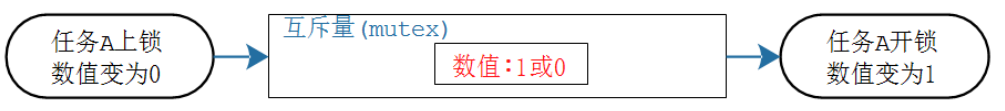
1. **任务通知：**任务通知的核心是任务的TCB中的数值，但是任务通知是通过更新任务通知值来发送事件或数据的，一个

任务通知值只能保存一次。必须指定接收的任务，且只能由接收任务本身获取该通知。

文本

低可信度描述已自动生成

1. **互斥量**：互斥量的数值只有0和1，谁获得互斥量，就必须由谁释放同一个互斥量。



# 3 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。项目开发中经常会遇到任务与任务之间、任务与中断之间进行信息传递。不使用操作系统时，一般使用全局变量进行信息传递，但在操作系统中使用这种方式会涉及资源管理的问题，比如读写冲突等，FreeRTOS为此提供了队列机制。队列是一种任务到任务、任务到中断、中断到任务数据交流的一种机制。队列不属于某个特定的任务，可以在任何的任务或中断中往队列中写入消息，或者从队列中读取消息。

1. **特性**

队列可以包含若干个数据，每个数据大小固定，在创建队列时就要指定长度和数据的大小。操作队列时按照先进先出的

方法，写数据时放到尾部，读数据时从头部读。也可以强制写队列头部，覆盖头部数据。只要知道队列的句柄，所有的任务和ISR都可读、写队列。

读取队列时，若队列中没有数据，则进入阻塞状态，等待数据，当队列中有数据时，该任务就冲阻塞态进入就绪态，也可设置阻塞超时时间，如果队列中移植没有数据，则时间到了它也会进入就绪态。当多个任务因为读取一个队列进入阻塞态时，当队列中有数据了，优先级最高的任务会进入就绪态，如果任务优先级都相同，则等待时间最长的任务进入就绪态。

写队列同理，也有阻塞超时时间，队列满了任务就进入阻塞态，等待队列有空闲空间。多个任务因写队列进入阻塞态时，当队列有空闲空间，优先级最高的任务会进入就绪态，如果任务优先级都相同，则等待时间最长的任务进入就绪态。

1. **传输数据方法**

使用队列数据时有两种方法：拷贝、引用。

1. **拷贝**：把数据、把变量的值复制进队列里。将局部变量的值发送到队列中，后续即使函数退出，也不会影响队列中

的数据。

1. **引用**：把数据、把变量的地址复制进队列里。如果数据太大就可以使用队列传输他的地址，队列的空间由FreeRTOS

内核分配，无需任务操心。注意，对于有内存保护功能的系统，引用地址时必须确保双方任务对这个地址都有访问权限。

1. **队列相关函数**

使用队列流程为：创建队列、写队列、读队列、复位队列、删除队列。

1. **创建队列**：

创建队列有两种方法：动态分配内存、静态分配内存。

动态分配内存：QueueHandle\_t xQueueCreate( UBaseType\_t uxQueueLength, //队列长度

UBaseType\_t uxItemSize); //每个数据的大小

如果创建成功，返回队列句柄；创建失败，则说明内存不足，返回NULL。

静态分配内存：xQueueCreateStatic( uxQueueLength, //队列长度

uxItemSize, //每个数据的大小

pucQueueStorage, //存储区域的起始地址，区域大小至少为“长度\*大小”

pxQueueBuffer) //创建成功则为句柄，否则为NULL

1. **复位队列**

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。

BaseType\_t xQueueReset( QueueHandle\_t pxQueue); // pxQueue：要复位的队列

1. **删除队列**

只能删除通过动态方法创建的队列，它会释放内存，函数如下。

void vQueueDelete( QueueHandle\_t xQueue ); // xQueue：要删除的队列

1. **写队列**

写队列时，可以将数据写到头部，也可以写到尾部，可以在任务中和中断中使用。



1. **读队列**

读队列时，读到一个数据后，该数据会从队列中移除，同样可以在任务中和中断中使用。



注意，xQueuePeek()只是偷看一下队列中的数据，最后并不会删除数据。

1. **查询**

可以查询队列中有多少数据、还剩余多少空间。

UBaseType\_t uxQueueMessagesWaiting( const QueueHandle\_t xQueue ); //返回队列可用数据的个数

UBaseType\_t uxQueueSpacesAvailable( const QueueHandle\_t xQueue ); //返回队列可用空间的个数

1. **队列锁**

在队列被上锁后，可以往队列中写入消息和读取消息，但是队列消息的读取和写入不会影响到队列读取和写入阻塞任务列表中的任务阻塞，队列的写入和读取阻塞任务列表会在队列解锁后，统一处理。

队列上锁的函数：prvLockQueue(pxQueue)

队列解锁的函数：prvUnlockQueue(Queue\_t \* const pxQueue)

# 4 信号量

信号量是一种解决同步问题的机制，可以实现对共享资源的有序访问。信号量分为两种：计数型信号量、二值信号量。

二者的区别在于对计数值的限制，当计数值被限制为1时，则是二值信号量，否则为计数型信号量。二值信号量在创建时初始值为0，计数型信号量在创建时可以设定初始值。除此之外，计数型信号量和二值型信号量的操作是一样的。

* 1. **信号量的操作**

对信号量支持give和take操作，give给出资源，计数值加1，give时不存在阻塞时间，当计数值达到最大值时直接返回错误；take获得资源，计数值减1，当take时计数值为0则进入阻塞，当有信号量时任务优先级高的任务可take获得到资源，若任务优先级相同，则阻塞时间最长的获得资源。give、take双方并不需要相同，可以用于生产者-消费者场合。

* 1. **信号量和队列的区别**

|  |  |
| --- | --- |
| **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：不需要阻塞，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 |

* 1. **相关函数**



表格

描述已自动生成

# 5 互斥量

互斥信号量其实就是拥有优先级继承的二值信号量，用来解决优先级反转的问题。在同步的应用中使用二值信号量更合适，在需要互斥访问的应用中使用互斥信号量更合适。互斥信号量就像一把钥匙，当访问临界资源时就可使用互斥信号量进行上锁，进行互斥访问，当临界资源使用完成后再解锁，别的任务就可以接着使用临界资源了。FreeRTOS中互斥信号量没有实现由谁上锁就由谁释放，比如任务一对临界资源上锁后，其他任务也可进行解锁。互斥信号量和二值信号量的操作函数相同，也可进行give和take操作，所以互斥信号量也可设置阻塞时间。

* 1. **优先级继承**

当一个互斥信号量被一个低优先级的任务持有时，高优先级的任务想要获取该互斥量时就会进入阻塞状态，这时若有

中等优先级在执行，低优先级任务就无法解锁该互斥量，导致高优先级的任务移植无法执行，这就是优先级翻转。优先级继承就是当高优先级想要获取低优先级持有的互斥信号量时，进入阻塞的同时，将优先级任务的优先级设置为和该高优先级任务相同，进而尽可能的减少了高优先级任务进入阻塞状态的时间。

注意，中断服务函数不是任务，没有优先级，且中断服务函数也不能因为等待互斥量而进入阻塞状态，所以中断中无法使用互斥信号量。

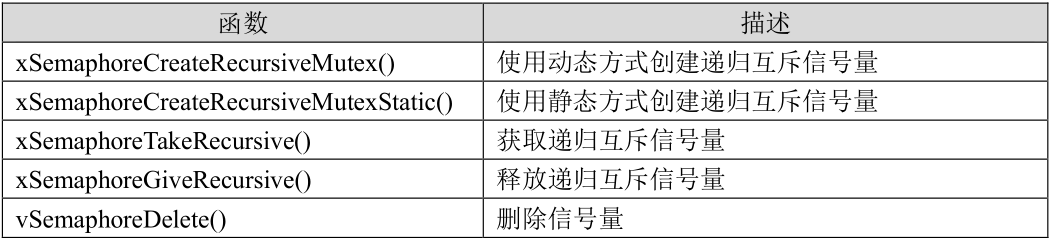
* 1. **相关函数**



* 1. **递归互斥信号量**

递归互斥信号量时特殊的互斥信号量，不同的是递归互斥信号量被获取后，可由持有者重复获取，释放的次数和获取的

次数应该相同。递归互斥信号量用来解决死锁的问题，比如任务一获得了互斥信号量，任务一由调用了一个函数，这个函数中需要获取这个互斥量，从而导致该任务阻塞，即自我死锁。再如：任务1获取互斥量L1，任务2获取互斥量L2，任务1要获取L2才能运行，进入阻塞等待任务2释放L2；任务2需要L1才能运行，也进入阻塞，这就导致死锁。递归锁实现了由谁上锁就要由谁来解锁。递归互斥信号量也有优先级继承机制，所以也不能在中断服务函数中使用。相关函数如下：



**小结**：队列可以在任务与任务间、任务与中断服务函数间传递数据，但信号量只能传递计数值，不能传递数据。根据计数值

的大小可将信号量分为计数型信号量和二值信号量。二者都可进行give和take操作，计数型信号量在创建时可设置初始值和计数的最大值，当达到最大计数值时，执行give操作没有阻塞，会直接返回错误，当计数值为0时，执行take操作会进入阻塞，可设置阻塞时间。二值信号量创建时默认初始值为0。计数型信号量通常用于事件计数和实现对资源的有序访问；二值信号量通常用于互斥访问或同步，但二值信号量会导致优先级翻转的问题，所以引入了互斥信号量。互斥信号量有优先级继承机制，尽可能的减少高优先级任务因等待信号量而阻塞的时间。但FreeRTOS中的互斥信号量没有实现由谁获取就由谁释放，所以可能会导致死锁。递归互斥信号量解决了这个问题，它除了有优先级继承机制外，还实现了由谁获取就由谁释放，且可由持有者重复获取，释放的次数也应该和获取的次数相等。注意，这几种信号量都不能在中断服务函数中使用，因为中断服务函数不是任务，没有任务优先级，且中断服务函数也不能因为等待互斥量而进入阻塞状态，所以信号量只能在任务间使用。

# 6 事件组

事件组和任务量类似，也属于任务间同步的机制。信号量用于单事件已同步，多事件的同步就需要用到事件组。事件组

是一组事件标志的集合，事件标志用于指示一个事件是否发生，为0表示没发生，为1表示发生了。事件组用EventBits\_t定义，EventBits\_t实际上是一个无符号的16位或32位整数。宏configUSE\_16\_BIT\_TICKS配置为0时EventBits\_t为32位，宏configUSE\_16\_BIT\_TICKS配置为1时EventBits\_t为16位。高8位留给系统使用，其余位才能设置为事件标志。

* 1. 事件组的用法

1. **任务同步**

如A、B、C三人相约郊游，A到车站了将标志位1置1；B到车站了将标志位2置1；C到车站了将标志位3置

1；只有三个人都到车站了，才能坐车出发。

1. **等待多个事件**

A、B、C三人做菜，A负责炒菜、B负责买菜、C负责洗菜，只有B和C的工作做完后A才能炒菜。所以A可检

测相应标志位，等待B、C将相应标志位置1后再炒菜。

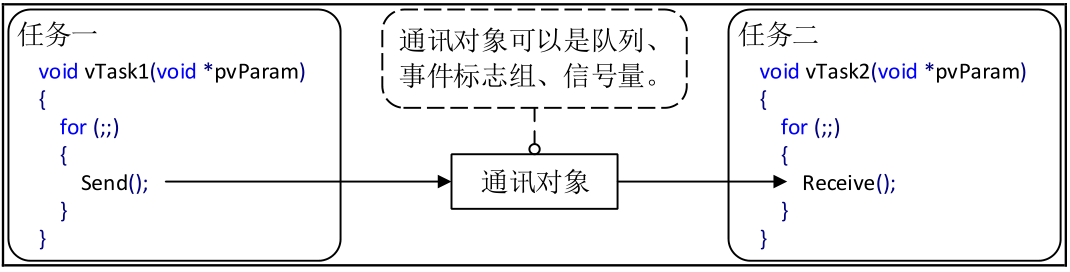
再如公司需要交一篇汇报，A、B、C三人任何一个交一篇都可以，三个标志位任何一个置1都可进行下一步操作。

* 1. **相关函数**



# 7 任务通知

任务通知也是用于任务间进行同步和通讯的一种机制，与前面介绍的队列、信号量等不同的是，任务通知不需要创建结构体，在创建任务时，任务结构体中就包含了任务通知对象。在FreeRTOS中，每个任务都有两个用于任务通知功能的数组，分别为任务通知数组、任务通知状态数组。任务通知数组中的每个元素都是1个32位无符号类型的通知值，任务通知状态数组中的元素则表示与之对应的任务通知的状态。



**间接通讯示意图**

图示

描述已自动生成

**直接通讯示意图**

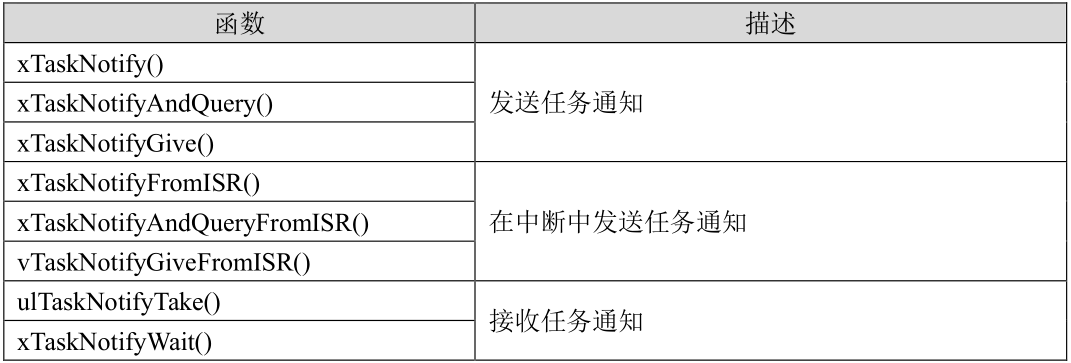
* + 1. **任务通知的优势**

使用任务通知向任务发送事件或数据比使用队列、信号量、事件组的能通讯对象要快得多，且可以节省大量的内存。任

务通知只需要在每个任务中占用固定5个字节。

* + 1. **任务通知的缺点**
       1. 不能向中断服务函数传递数据和信息，因为中断服务函数不是任务，没有对应的通知状态数组。
       2. 不能同时向多个任务传递数据，任务通知只能直接发送事件或数据到指定任务，且数据只能接收任务独享。
       3. 不能缓存数据，任务通知通过更新任务通知值来发送事件和数据，一个任务的通知值只能保存一次。
       4. 不能进入阻塞去等待任务，如果接受任务无法接受数据，任务通知就会直接返回错误，不会等待。
    2. **任务通知的相关函数**

**表1**



**表2**

表格

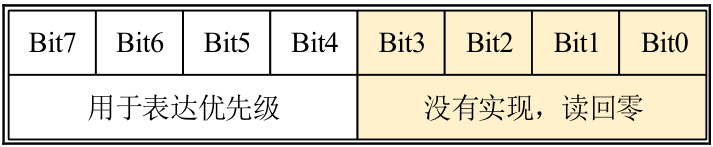
描述已自动生成

一个任务可以有多个任务通知，多个通知就通过数组的下标进行索引，表1中的函数都是对相关数组中下标为0的元素进行操作，表2中函数可以对指定下标的元素进行操作。

# 8 FreeRTOS中断管理

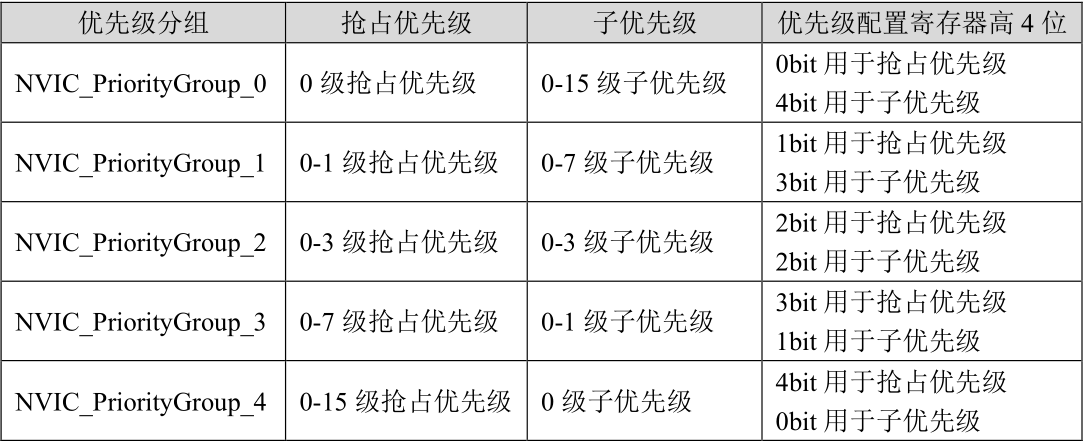
中断是CPU的一种常见特性，一般由硬件产生，ARM Cortex-M内核的MCU具有一个用于中断管理的嵌套向量中断控制器NVIC，NVIC最多支持256个中断源，包括16个系统中断和240个外部中断。芯片厂商通常用不完这些资源，如正点原子战舰开发板使用的STM32F103芯片只用到了10个系统中断和60个外部中断。

ARM Cortex-M使用NVIC对不同优先级的中断进行管理，NVIC使用8位宽的寄存器来配置中断的优先等级，即等级范围为0~255，STM32只用到了高4位，即等级范围为0~15，如下所示。



**中断优先级配置寄存器**

STM32中断优先级分为抢占优先级和子优先级，抢占优先级高的中断可以打断正在执行的抢占优先级低的中断，即中断嵌套；抢占优先级相同时，子优先级高的中断不能打断子优先级低的中断，即子优先级不支持中断嵌套。HAL库中对优先级进行了分组，如下所示。



**优先级分组**

FreeRTOS官方强烈建议STM32在使用时使用分组4，即NVIC\_PriorityGroup\_4，即优先级配置寄存器的高 4 位全部用于抢占优先级，不使用子优先级，这样用户只需要设置抢占优先级即可。

* 1. **FreeRTOS中断配置项**

FreeRTOSConfig.h 文件中有 6 个与中断相关的 FreeRTOS 配置项，如下表所示。

|  |  |
| --- | --- |
| **配置项** | **功能** |
| configPRIO\_BITS | MCU的8位优先级配置寄存器实际使用的位数 |
| configLIBRARY\_LOWEST\_INTERRUPT\_PRIORITY | MCU的最低中断优先等级，建议使用组4，即最低为15级 |
| configLIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY | 设置FreeRTOS可管理中断的最高优先级 |
| configKERNEL\_INTERRUPT\_PRIORITY | MCU的最低中断优先等级在中断优先级配置寄存器中的值 |
| configMAX\_SYSCALL\_INTERRUPT\_PRIORITY | FreeRTOS可管理中断的最高优先等级在中断优先级配置寄存器中的值 |
| configMAX\_API\_CALL\_INTERRUPT\_PRIORITY | 与宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY等价 |

STM32只用到了中断优先级寄存器的高4位，所以设置configPRIO\_BITS为4；我们使用优先级分组4，即不使用子优先级，所以优先级范围为0~15，最低优先级为15，所以设置configLIBRARY\_LOWEST\_INTERRUPT\_PRIORITY为15；宏configLIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY用来设置FreeRTOS可管理的最高优先级中断，正点原子教程中将其设置为了5，即中断优先级高于5（即优先级0~4）的中断不受FreeRTOS的影响；在 FreeRTOS 的源码中，使用宏configKERNEL\_INTERRUPT\_PRIORITY用来将SysTick 和 PenSV 的中断优先级设置为最低优先级，最低优先级为15，且STM32只使用优先级寄存器的高4位，所以将该宏设置为15<<4，即(configLIBRARY\_LOWEST\_INTERRUPT\_PRIORITY<<(8-configPRIO\_BITS))；同理，最高优先级为5时，将宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY设置为5<<4，即(configLIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY<<(8-configPRIO\_BITS))。

* 1. **进出临界区**

临界区是指那些必须完整运行的区域，临界区中的代码必须被完整执行，不能被打断，如一些使用软件模拟的通讯时序，

必须严格按照通讯协议的时序执进行，FreeRTOS通过关闭和打开受FreeRTOS管理的中断设置临界区。与进出临界区相关的宏定义分别为：

进入临界区： taskENTER\_CRITICAL() /\* 屏蔽中断 \*/

退出临界区： taskEXIT\_CRITICAL() /\* 使能中断 \*/

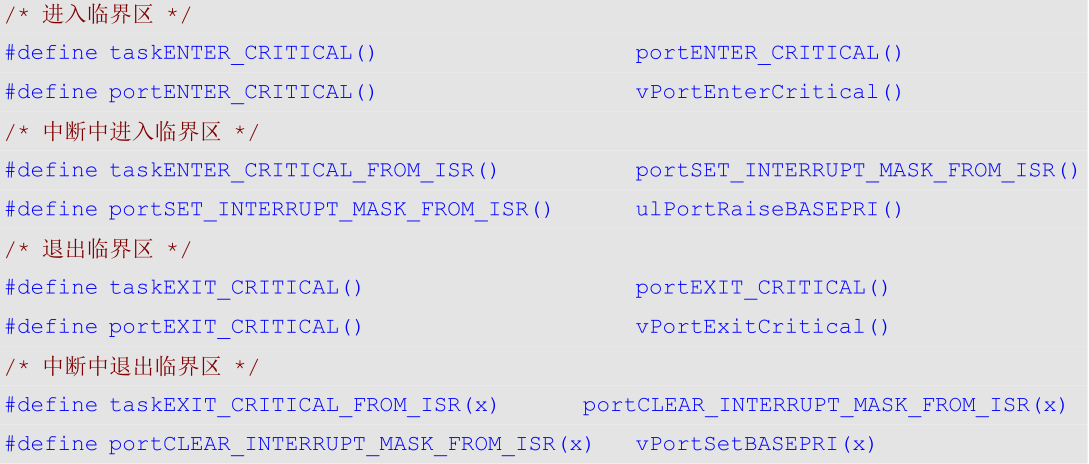
中断中进入临界区： taskENTER\_CRITICAL\_FROM\_ISR() /\* 屏蔽中断 \*/

中断中退出临界区： taskEXIT\_CRITICAL\_FROM\_ISR(x) /\* 恢复中断 \*/

暂停任务调度： vTaskSuspendAll() /\* 暂停任务调度器 \*/

恢复任务调度： xTaskResumeAll() /\* 恢复任务调度器 \*/

定义代码如下，



其中参数 x 就是宏 taskENTER\_CRITICAL\_FROM\_ISR()的返回值，用于在从中断中对出临界区时，恢复 BASEPRI 寄存器。

* 1. **两套API函数**

在任务函数中有两类函数：在任务中使用的函数、在中断中使用的函数。比如操作队列的函数：xQueueSendToBack()可用

在任务中向队列尾部写入数据，但在ISR中使用该函数会出现问题，需要使用xQueueSendToBackFromISR()函数，该函数用于从ISR中向队列尾部发送数据。FreeRTOS中很多函数都有在ISR中使用的版本，设计ISR版本的函数是因为很多API函数会导致任务进入阻塞状态，但ISR不是任务，所以不能进入阻塞状态，那为什么不在函数中判断当前是在任务中使用还是中断中呢，那是因为使用两套函数会使程序更高效。注意，在任务中这两套函数都可以使用，但中断中只能使用ISR版本的。

使用两套函数也会出现问题，比如使用第三方库函数时，在任务中和中断中都可能会调用第三方库函数，当第三方库函数调用FreeRTOS的API函数时可能会导致在中断中使用任务版本的函数，解决方法如下；

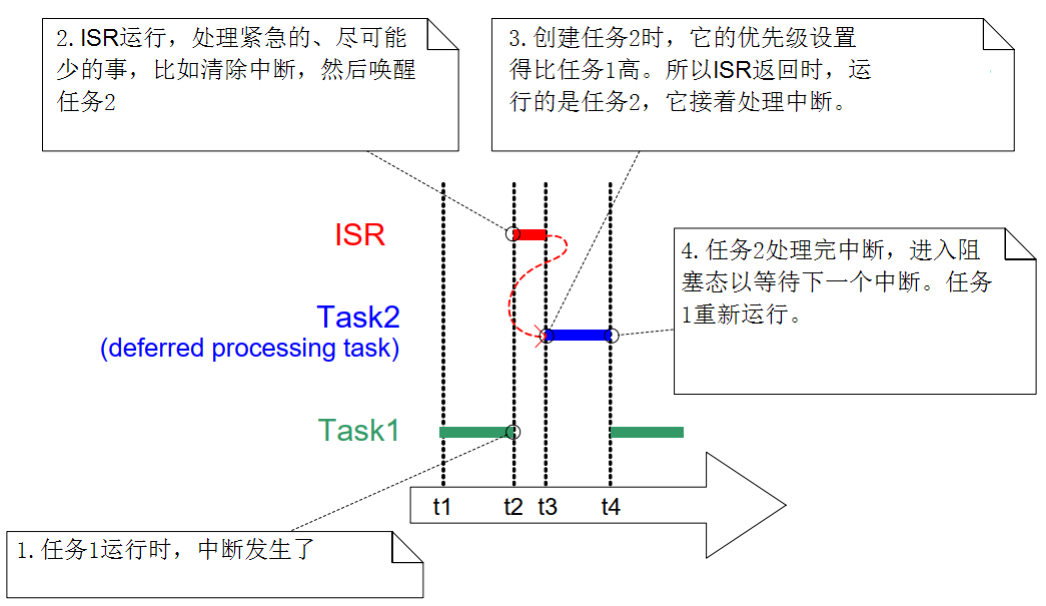
1. 把中断的处理推迟到任务中进行，在任务中再调用第三方库函数。
2. 尝试在库函数中使用ISR版本的函数，这样在任务和中断中就都可以使用了。
3. 第三方库函数也会提供OS抽象层，所以可以自行判断当前是在任务中还是中断中，然后使用对应版本的函数。

两套API函数汇总如下：



这两套函数的具体区别如下：

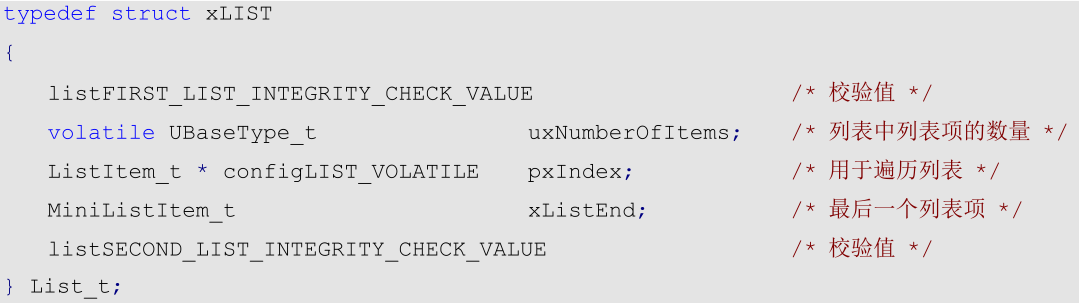
1. 参数不同，任务API函数可设置阻塞时间，而ISR函数没有阻塞时间，比如写队列失败就直接返回，不等待。
2. 任务调度时间不同，任务API函数中有一个死循环，比如队列为空时执行写队列操作，当把数据放到队列中后会判断等待队列是否为空，如有任务因等待队列数据而进入阻塞状态，会立刻将最高优先级或等待时间最长的任务从等待列表移动到就绪列表中，若该任务的任务优先级高于正在运行的任务，会马上执行任务调度。在ISR函数中如果出现上述情况，也会把任务从等待列表移动到就绪列表中，但不会马上执行任务调度，而是先用一个参数pxHigherPriorityTaskWoken记录一下需要调度，之后就可根据这个参数来手动执行任务调度。为什么不直接进行任务调度呢，因为任何任务的优先级都没有中断服务函数的优先级高，即使执行任务调度也要等中断服务函数执行完才会执行任务，所以在ISR函数中无论执行几次任务调度都没用，在执行ISR函数后使用portYIELD\_FROM\_ISR()函数执行一次任务调度就可以了，这样效率更高。将参数pxHigherPriorityTaskWoken传进去即可判断是否需要执行任务调度，这个函数只是触发任务调度，不是立刻进行任务调度，此函数就是设置一个优先级最低的中断，等当前中断服务函数执行完成后会立刻执行这个最低优先级的中断执行任务调度。综上，在ISR函数中不进行任务调度会提高效率；让ISR更可控，因为中断随机产生，在ISR中进行任务调度可能会使问题更复杂；可移植性更好。
   1. 中断的延迟处理



如上图所示，在运行任务1时发生了中断，为了尽快执行中断，ISR中只处理了极少的事情，然后创建了一个任务优先级更高的任务2，在任务2中继续处理应该在中断中处理的事情，处理完成后任务进入阻塞态，任务1继续执行。

# 9 列表和列表项

列表是 FreeRTOS 中最基本的一种数据结构，其在物理存储单元上是非连续、非顺序的。FreeRTOS 中的列表是一个双向链表，在list.h 文件中有列表的相关定义。



**列表结构体**

其中，校验值我们暂不研究，uxNumberOfItems 用于记录列表中列表项的个数（不包含 xListEnd）；pxIndex 用于指向列表中的某个列表项，一般用于遍历列表中的所有列表项；xListEnd 是一个迷你列表项，一般被设置为最大值，用于将列表中的所有列表项按升序排序时，排在最末尾，同时 xListEnd 也用于挂载其他插入到列表中的列表项。

图形用户界面, 文本, 应用程序

描述已自动生成

**列表项结构体**

列表项是列表中用于存放数据的地方，xItemValue 为列表项的值，对列表进行升序或降序排列时会用到；pxNext和 pxPrevious 分别用于指向列表中列表项的下一个列表项和上一个列表项；pxOwner用于指向包含列表项的对象（通常是任务控制块），因此，列表项和包含列表项的对象之间存在双向链接；pxContainer 用于指向列表项所在列表。

迷你列表项只用于标记列表的末尾和挂载其他插入列表中的列表项，因此不需要成员变量 pxOwner 和 pxContainer，以节省内存开销。

表格

低可信度描述已自动生成

**迷你列表项结构体**

下面介绍常用的列表与列表项相关函数，使用示例见正点原子代码例程。

1）初始化列表：vListInitialise()用于初始化列表，只有一个参数pxList，即待初始化的列表。

2）初始化列表项：vListInitialiseItem()用于初始化列表项，只有一个参数pxItem，即待初始化的列表项。

3）无序插入列表项：vListInsertEnd()插入列表项时为无序插入，直接列表后面插入。

4）有序插入列表项： vListInsert()插入列表项时为有序插入，根据列表项的xItemValue值升序排序后插入到相应位置。

5）移除列表项：uxListRemove()用于移除列表项，只有一个参数pxItemToRemove，为待移除的列表项。

# 10 系统启动流程

创建好任务后，调用函数 vTaskStartScheduler()即可启动任务调度器，FreeRTOS 便会开始进行任务调度，除非调用函数 xTaskEndScheduler()停止任务调度器，否则不会再返回。

vTaskStartScheduler()主要做了六件事情，介绍如下，具体代码在工程中查看，这里不列出。

1）创建空闲任务，根据是否支持静态内存管理，使用静态方式或动态方式创建空闲任务。

2）创建定时器服务任务，同样根据是否配置支持静态内存管理，使用静态或动态方式创建定时器服务任务。

3）关闭中断，使用 portDISABLE\_INTERRUPT()关闭FreeRTOS管理的中断，主要是为了防止 SysTick中断在任务调

度器开启之前或过程中产生中断，在开始运行第一个任务时会重新打开中断。

4）初始化一些全局变量，并将任务调度器的运行标志设置为已运行。

5）初始化时基定时器，此定时器用于统计任务运行时间，此功能需要一个硬件定时器提供高精度的计数，这个硬件定时

器就在这里进行配置，如果配置不启用任务运行时间统计功能的，就无需进行这项硬件定时器的配置。

6）最后就是调用函数 xPortStartScheduler()，下面介绍此函数。

函数 xPortStartScheduler()完成启动任务调度器中与硬件架构相关的配置部分，以及启动第一个任务。主要工作如下。

1）如果启用断言，则检测用户在 FreeRTOSConfig.h 文件中对中断的相关配置是否有误。

2）配置 PendSV 和 SysTick 的中断优先级为最低优先级。

3）调用函数 vPortSetupTimerInterrupt()配置SysTick，首先会将SysTick当前计数值清空，并根据FreeRTOSConfig.h 文

件中配置的configSYSTICK\_CLOCK\_HZ（SysTick 时钟源频率）和 configTICK\_RATE\_HZ（系统时钟节拍频率）计算并设置 SysTick的重装载值，然后启动SysTick计数和中断。

4）初始化临界区嵌套计数器为0。

5）调用函数 prvEnableVFP()使能FPU，因为ARM Cortex-M3内核MCU无FPU，此函数仅在ARM Cortex-M4/M7内核

MCU平台上被调用，执行改函数后FPU被开启。

6）将 FPCCR寄存器的[31:30]置1，这样在进出异常时，FPU 的相关寄存器就会自动地保存和恢复，同样地，因为 ARM

Cortex-M3 内核 MCU 无 FPU，此当代码仅在 ARM Cortex-M4/M7 内核 MCU 平台上被调用。

7）调用函数 prvStartFirstTask()启动第一个任务，下面介绍此函数。

函数 prvStartFirstTask()用于初始化启动第一个任务前的环境，主要是重新设置 MSP 指针，并使能全局中断。

MSP指针：

程序在运行过程中需要一定的栈空间来保存局部变量等一些信息。当有信息保存到栈中时，MCU会自动更新SP指针，使SP指针指向最后一个入栈的元素，那么程序就可以根据SP指针来从栈中存取信息。对于正点原子的 STM32F1、STM32F4、STM32F7 和 STM32H7 开发板上使用的 ARM Cortex-M 的 MCU 内核来说，ARM Cortex-M 提供了两个栈空间，这两个栈空间的堆栈指针分别是MSP（主堆栈指针）和PSP（进程堆栈指针）。在FreeRTOS中MSP是给系统栈空间使用的，而PSP是给任务栈使用的，也就是说，FreeRTOS 任务的栈空间是通过 PSP 指向的，而在进入中断服务函数时，则是使用MSP指针。当使用不同的堆栈指针时，SP会等于当前使用的堆栈指针。

我们需要将MSP指向中断向量表的第一个成员，即栈底指针。这个操作相当于丢弃了程序之前保存在栈中的数据，因为FreeRTOS从开启任务调度器到启动第一个任务都是不会返回的，是一条不归路，因此将栈中的数据丢弃，也不会有影响。

图形用户界面

低可信度描述已自动生成

重新赋值 MSP后，接下来就重新使能全局中断，因为之前在函数 vTaskStartScheduler()中关闭了FreeRTOS管理的中断，最后使用 SVC 指令，并传入系统调用号0，触发SVC中断。SVC中断函数vPortSVCHandler()就是用来跳转到第一个任务函数中去的。首先会获取优先级最高的任务（也就是即将运行的任务）的任务栈地址，将栈中内容出栈到CPU寄存器中，任务栈中的内容在调用任务创建函数的时候，已经初始化了。然后再设置 PSP 指针，那么，这么一来，任务的运行环境就准备好了。注意，SVC中断服务函数只在这里使用了一次，之后均不调用。