# 配置

## 模块

* [**Customisation**](https://www.freertos.org/a00110.html)
* [**Memory Management**](https://www.freertos.org/a00111.html)
* [**Stack Overflow Detection**](https://www.freertos.org/Stacks-and-stack-overflow-checking.html)

# 配置文件

FreeRTOS是使用名为FreeRTOSConfig.h的配置文件定制的。每个FreeRTOS应用程序的预处理器包含路径中都必须具有FreeRTOSConfig.h头文件。FreeRTOSConfig.h使RTOS内核适合正在构建的应用程序。因此，它特定于应用程序而不是RTOS，并且应位于应用程序目录中，而不是RTOS内核源代码目录之一中。

RTOS源代码下载中包含的每个演示应用程序都有其自己的FreeRTOSConfig.h文件。其中的一些演示很老，并不包含所有可用的配置选项。在RTOS源文件中，将省略的配置选项设置为默认值。

这是典型的FreeRTOSConfig.h定义，后面是每个参数的说明：

**/\* Here is a good place to include header files that are required across**

**your application. \*/**

**#include "something.h"**

**#define** [**configUSE\_PREEMPTION**](https://www.freertos.org/a00110.html#configUSE_PREEMPTION) **1**

**#define** [**configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION**](https://www.freertos.org/a00110.html#configUSE_PORT_OPTIMISED_TASK_SELECTION) **0**

**#define** [**configUSE\_TICKLESS\_IDLE**](https://www.freertos.org/a00110.html#configUSE_TICKLESS_IDLE) **0**

**#define** [**configCPU\_CLOCK\_HZ**](https://www.freertos.org/a00110.html#configCPU_CLOCK_HZ) **60000000**

**#define** [**configSYSTICK\_CLOCK\_HZ**](https://www.freertos.org/a00110.html#configSYSTICK_CLOCK_HZ) **1000000**

**#define** [**configTICK\_RATE\_HZ**](https://www.freertos.org/a00110.html#configTICK_RATE_HZ) **250**

**#define** [**configMAX\_PRIORITIES**](https://www.freertos.org/a00110.html#configMAX_PRIORITIES) **5**

**#define** [**configMINIMAL\_STACK\_SIZE**](https://www.freertos.org/a00110.html#configMINIMAL_STACK_SIZE) **128**

**#define** [**configMAX\_TASK\_NAME\_LEN**](https://www.freertos.org/a00110.html#configMAX_TASK_NAME_LEN) **16**

**#define** [**configUSE\_16\_BIT\_TICKS**](https://www.freertos.org/a00110.html#configUSE_16_BIT_TICKS) **0**

**#define** [**configIDLE\_SHOULD\_YIELD**](https://www.freertos.org/a00110.html#configIDLE_SHOULD_YIELD) **1**

**#define** [**configUSE\_TASK\_NOTIFICATIONS**](https://www.freertos.org/a00110.html#configUSE_TASK_NOTIFICATIONS) **1**

**#define** [**configTASK\_NOTIFICATION\_ARRAY\_ENTRIES**](https://www.freertos.org/a00110.html#configTASK_NOTIFICATION_ARRAY_ENTRIES) **3**

**#define** [**configUSE\_MUTEXES**](https://www.freertos.org/a00110.html#configUSE_MUTEXES) **0**

**#define** [**configUSE\_RECURSIVE\_MUTEXES**](https://www.freertos.org/a00110.html#configUSE_RECURSIVE_MUTEXES) **0**

**#define** [**configUSE\_COUNTING\_SEMAPHORES**](https://www.freertos.org/a00110.html#configUSE_COUNTING_SEMAPHORES) **0**

**#define** [**configUSE\_ALTERNATIVE\_API**](https://www.freertos.org/a00110.html#configUSE_ALTERNATIVE_API) **0 /\* Deprecated! \*/**

**#define** [**configQUEUE\_REGISTRY\_SIZE**](https://www.freertos.org/a00110.html#configQUEUE_REGISTRY_SIZE) **10**

**#define** [**configUSE\_QUEUE\_SETS**](https://www.freertos.org/a00110.html#configUSE_QUEUE_SETS) **0**

**#define** [**configUSE\_TIME\_SLICING**](https://www.freertos.org/a00110.html#configUSE_TIME_SLICING) **0**

**#define** [**configUSE\_NEWLIB\_REENTRANT**](https://www.freertos.org/a00110.html#configUSE_NEWLIB_REENTRANT) **0**

**#define** [**configENABLE\_BACKWARD\_COMPATIBILITY**](https://www.freertos.org/a00110.html#configENABLE_BACKWARD_COMPATIBILITY) **0**

**#define** [**configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS**](https://www.freertos.org/a00110.html#configNUM_THREAD_LOCAL_STORAGE_POINTERS) **5**

**#define** [**configSTACK\_DEPTH\_TYPE**](https://www.freertos.org/a00110.html#configSTACK_DEPTH_TYPE) **uint16\_t**

**#define** [**configMESSAGE\_BUFFER\_LENGTH\_TYPE**](https://www.freertos.org/a00110.html#configMESSAGE_BUFFER_LENGTH_TYPE) **size\_t**

**/\* Memory allocation related definitions. \*/**

**#define** [**configSUPPORT\_STATIC\_ALLOCATION**](https://www.freertos.org/a00110.html#configSUPPORT_STATIC_ALLOCATION) **1**

**#define** [**configSUPPORT\_DYNAMIC\_ALLOCATION**](https://www.freertos.org/a00110.html#configSUPPORT_DYNAMIC_ALLOCATION) **1**

**#define** [**configTOTAL\_HEAP\_SIZE**](https://www.freertos.org/a00110.html#configTOTAL_HEAP_SIZE) **10240**

**#define** [**configAPPLICATION\_ALLOCATED\_HEAP**](https://www.freertos.org/a00110.html#configAPPLICATION_ALLOCATED_HEAP) **1**

**/\* Hook function related definitions. \*/**

**#define** [**configUSE\_IDLE\_HOOK**](https://www.freertos.org/a00110.html#configUSE_IDLE_HOOK) **0**

**#define** [**configUSE\_TICK\_HOOK**](https://www.freertos.org/a00110.html#configUSE_TICK_HOOK) **0**

**#define** [**configCHECK\_FOR\_STACK\_OVERFLOW**](https://www.freertos.org/a00110.html#configCHECK_FOR_STACK_OVERFLOW) **0**

**#define** [**configUSE\_MALLOC\_FAILED\_HOOK**](https://www.freertos.org/a00110.html#configUSE_MALLOC_FAILED_HOOK) **0**

**#define** [**configUSE\_DAEMON\_TASK\_STARTUP\_HOOK**](https://www.freertos.org/a00110.html#configUSE_DAEMON_TASK_STARTUP_HOOK) **0**

**/\* Run time and task stats gathering related definitions. \*/**

**#define** [**configGENERATE\_RUN\_TIME\_STATS**](https://www.freertos.org/a00110.html#configGENERATE_RUN_TIME_STATS) **0**

**#define** [**configUSE\_TRACE\_FACILITY**](https://www.freertos.org/a00110.html#configUSE_TRACE_FACILITY) **0**

**#define** [**configUSE\_STATS\_FORMATTING\_FUNCTIONS**](https://www.freertos.org/a00110.html#configUSE_STATS_FORMATTING_FUNCTIONS) **0**

**/\* Co-routine related definitions. \*/**

**#define** [**configUSE\_CO\_ROUTINES**](https://www.freertos.org/a00110.html#configUSE_CO_ROUTINES) **0**

**#define** [**configMAX\_CO\_ROUTINE\_PRIORITIES**](https://www.freertos.org/a00110.html#configMAX_CO_ROUTINE_PRIORITIES) **1**

**/\* Software timer related definitions. \*/**

**#define** [**configUSE\_TIMERS**](https://www.freertos.org/a00110.html#configUSE_TIMERS) **1**

**#define** [**configTIMER\_TASK\_PRIORITY**](https://www.freertos.org/a00110.html#configTIMER_TASK_PRIORITY) **3**

**#define** [**configTIMER\_QUEUE\_LENGTH**](https://www.freertos.org/a00110.html#configTIMER_QUEUE_LENGTH) **10**

**#define** [**configTIMER\_TASK\_STACK\_DEPTH**](https://www.freertos.org/a00110.html#configTIMER_TASK_STACK_DEPTH) **configMINIMAL\_STACK\_SIZE**

**/\* Interrupt nesting behaviour configuration. \*/**

**#define** [**configKERNEL\_INTERRUPT\_PRIORITY**](https://www.freertos.org/a00110.html#kernel_priority) **[dependent of processor]**

**#define** [**configMAX\_SYSCALL\_INTERRUPT\_PRIORITY**](https://www.freertos.org/a00110.html#kernel_priority) **[dependent on processor and application]**

**#define** [**configMAX\_API\_CALL\_INTERRUPT\_PRIORITY**](https://www.freertos.org/a00110.html#kernel_priority) **[dependent on processor and application]**

**/\* Define to trap errors during development. \*/**

**#define** [**configASSERT**](https://www.freertos.org/a00110.html#configASSERT)**( ( x ) ) if( ( x ) == 0 )** **vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )**

**/\* FreeRTOS MPU specific definitions. \*/**

**#define** [**configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS**](https://www.freertos.org/a00110.html#configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS) **0**

**#define** [**configTOTAL\_MPU\_REGIONS**](https://www.freertos.org/a00110.html#configTOTAL_MPU_REGIONS) **8 /\* Default value. \*/**

**#define** [**configTEX\_S\_C\_B\_FLASH**](https://www.freertos.org/a00110.html#configTEX_S_C_B_FLASH) **0x07UL /\* Default value. \*/**

**#define** [**configTEX\_S\_C\_B\_SRAM**](https://www.freertos.org/a00110.html#configTEX_S_C_B_SRAM) **0x07UL /\* Default value. \*/**

**#define** [**configENFORCE\_SYSTEM\_CALLS\_FROM\_KERNEL\_ONLY**](https://www.freertos.org/a00110.html#configENFORCE_SYSTEM_CALLS_FROM_KERNEL_ONLY) **1**

**/\* Optional functions - most linkers will remove unused functions anyway. \*/**

**#define** [**INCLUDE\_vTaskPrioritySet**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_uxTaskPriorityGet**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_vTaskDelete**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_vTaskSuspend**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_xResumeFromISR**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_vTaskDelayUntil**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_vTaskDelay**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_xTaskGetSchedulerState**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_xTaskGetCurrentTaskHandle**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_uxTaskGetStackHighWaterMark**](https://www.freertos.org/a00110.html#include_parameters) **0**

**#define** [**INCLUDE\_xTaskGetIdleTaskHandle**](https://www.freertos.org/a00110.html#include_parameters) **0**

**#define** [**INCLUDE\_eTaskGetState**](https://www.freertos.org/a00110.html#include_parameters) **0**

**#define** [**INCLUDE\_xEventGroupSetBitFromISR**](https://www.freertos.org/a00110.html#include_parameters) **1**

**#define** [**INCLUDE\_xTimerPendFunctionCall**](https://www.freertos.org/a00110.html#include_parameters) **0**

**#define** [**INCLUDE\_xTaskAbortDelay**](https://www.freertos.org/a00110.html#include_parameters) **0**

**#define** [**INCLUDE\_xTaskGetHandle**](https://www.freertos.org/a00110.html#include_parameters) **0**

**#define** [**INCLUDE\_xTaskResumeFromISR**](https://www.freertos.org/a00110.html#include_parameters) **1**

**/\* A header file that defines trace macro can be included here. \*/**

**#endif /\* FREERTOS\_CONFIG\_H \*/**

## 'config'参数

### configUSE\_PREEMPTION

设置为1使用抢占式RTOS调度程序，或设置为0使用协作式RTOS调度程序。

### configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION

一些FreeRTOS接口有两种选择下一个要执行的任务的方法-通用接口和特定接口。

通用接口：

* 在configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION设置为0或未实现特定接口时使用。
* 可以与所有FreeRTOS接口一起使用。
* 完全用C编写，使其效率低于特定接口。
* 不限制可用优先级的最大数量。

特定接口：

* 不适用于所有接口。
* 在configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION设置为1时使用。
* 依赖于一个或多个特定于体系结构的汇编指令（通常为计数前导零[CLZ]或等效指令），因此只能与专门为其编写的体系结构一起使用。
* 比通用方法更有效率。
* 通常，最大可用优先级数限制为32。

### configUSE\_TICKLESS\_IDLE

将configUSE\_TICKLESS\_IDLE设置为1以使用[低功耗tickless模式](https://www.freertos.org/low-power-tickless-rtos.html)，或者将0设置为始终保持滴答中断运行。没有为所有FreeRTOS端口提供低功耗的tickless实现。

### configUSE\_IDLE\_HOOK

如果要使用[空闲钩子](https://www.freertos.org/RTOS-idle-task.html)，则设置为1；或者，如果忽略空闲钩子，则设置为0。

### configUSE\_MALLOC\_FAILED\_HOOK

每当创建任务，队列或信号时，内核都会使用对pvPortMalloc（）的调用从堆中分配内存。FreeRTOS的官方下载包括用于此目的的四个样本内存分配方案。该方案分别在heap\_1.c，heap\_2.c，heap\_3.c，heap\_4.c和heap\_5.c源文件中实现。configUSE\_MALLOC\_FAILED\_HOOK仅在使用这几个示例方案之一时才有意义。

malloc（）失败的钩子函数是一个钩子（或回调）函数，如果pvPortMalloc（）返回NULL，则该钩子函数（如果定义和配置了）将被调用。仅当FreeRTOS堆内存不足以使请求的分配成功时，才会返回NULL。

如果configUSE\_MALLOC\_FAILED\_HOOK设置为1，则应用程序必须定义一个失败的malloc（）挂钩函数。如果configUSE\_MALLOC\_FAILED\_HOOK设置为0，则即使定义了malloc（）失败的挂钩函数，也不会调用。malloc（）失败的挂钩函数必须具有如下所示的名称和原型。

**void vApplicationMallocFailedHook（void）;**

### configUSE\_DAEMON\_TASK\_STARTUP\_HOOK

如果configUSE\_TIMERS和configUSE\_DAEMON\_TASK\_STARTUP\_HOOK都设置为1，则应用程序必须定义一个具有确切名称和原型的钩子函数，如下所示。第一次执行RTOS守护程序任务（也称为[计时器服务任务](https://www.freertos.org/RTOS-software-timer-service-daemon-task.html)）时，hook函数将被完全调用一次 。任何需要运行RTOS的应用程序初始化代码都可以放入挂钩函数中。

**void vApplicationDaemonTaskStartupHook（void）;**

### configUSE\_TICK\_HOOK

如果要使用[嘀嗒钩](https://www.freertos.org/a00016.html" \l "TickHook)子函数，则设置为1；如果忽略[嘀嗒钩](https://www.freertos.org/a00016.html#TickHook)子函数，则设置为0。

### configCPU\_CLOCK\_HZ

输入以Hz为单位的频率，该频率将驱动用于驱动滴答中断的外设驱动的*内部*时钟–通常与驱动内部CPU时钟的时钟相同。为了正确配置计时器外设，此值是必需的。

### configSYSTICK\_CLOCK\_HZ

仅用于ARM Cortex-M端口的可选参数。

默认情况下，ARM Cortex-M端口从Cortex-M SysTick计时器生成RTOS滴答中断。大多数Cortex-M MCU都以与MCU本身相同的频率运行SysTick计时器-在这种情况下，不需要configSYSTICK\_CLOCK\_HZ且应将其保留为未定义状态。如果SysTick定时器的时钟频率与MCU内核的时钟频率不同，则照常将configCPU\_CLOCK\_HZ设置为MCU时钟频率，将configSYSTICK\_CLOCK\_HZ设置为SysTick时钟频率。

### configTICK\_RATE\_HZ

RTOS滴答中断的频率。

滴答中断用于测量时间。因此，较高的滴答频率意味着可以将时间测量为较高的分辨率。但是，较高的滴答频率也意味着RTOS内核将使用更多的CPU时间，因此效率较低。RTOS演示应用程序均使用1000Hz的滴答频率。它用于测试RTOS内核，并且比通常所需的更高。

一个以上的任务可以共享相同的优先级。通过在每个RTOS滴答之间切换任务，RTOS调度程序将让优先级相同的任务之间共享处理器时间。因此，较高的滴答频率会减少分配给每个任务的“时间片”。

### configMAX\_PRIORITIES

应用程序任务可用 的[优先](https://www.freertos.org/RTOS-task-priority.html)级数。任意数量的任务可以共享相同的优先级。协程分别进行优先级排序–请参见configMAX\_CO\_ROUTINE\_PRIORITIES。

每个可用的优先级都会占用RTOS内核中的一点RAM，因此该值不应设置为高于应用程序实际需要的值。

如果[configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION](https://www.freertos.org/a00110.html" \l "configUSE_PORT_OPTIMISED_TASK_SELECTION)设置为1，则最大允许值将受到限制。

### configMINIMAL\_STACK\_SIZE

空闲任务使用的堆栈大小。通常，不应将此值从演示应用程序随附的FreeRTOSConfig.h文件中为您使用的接口设置的值中减去。

类似于[xTaskCreate（）](https://www.freertos.org/a00125.html)和 [xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)函数的堆栈大小参数，堆栈大小以字而不是字节指定。如果放在堆栈上的每个项目都是32位，则堆栈大小为100表示​​400字节（每个32位堆栈项目消耗4个字节）。

### configMAX\_TASK\_NAME\_LEN

创建任务时赋予任务的描述性名称的最大允许长度。长度以*包括*NULL终止字节的字符数指定 。

### configUSE\_TRACE\_FACILITY

如果您希望包括其他数据结构和功能以帮助执行可视化和跟踪，请设置为1。

### configUSE\_STATS\_FORMATTING\_FUNCTIONS

将configUSE\_TRACE\_FACILITY和configUSE\_STATS\_FORMATTING\_FUNCTIONS设置为1，以在[构建中](https://www.freertos.org/a00021.html" \l "vTaskList)包括[vTaskList（）](https://www.freertos.org/a00021.html#vTaskList)和[vTaskGetRunTimeStats（）](https://www.freertos.org/a00021.html" \l "vTaskGetRunTimeStats) 函数。设置为0会从构建中忽略vTaskList（）和vTaskGetRunTimeStates（）。

### configUSE\_16\_BIT\_TICKS

时间以“滴答”表示，这是自RTOS内核启动以来滴答中断执行的次数。滴答计数保存在TickType\_t类型的变量中。

将configUSE\_16\_BIT\_TICKS定义为1会导致TickType\_t被定义（类型定义）为无符号的16位类型。将configUSE\_16\_BIT\_TICKS定义为0会导致TickType\_t被定义（类型定义）为无符号的32位类型。

使用16位类型将大大提高8位和16位体系结构的性能，但将最大可指定时间段限制为65535个“滴答声”。因此，假设滴答频率为250Hz，则使用16位计数器时任务可以延迟或阻止的最长时间为262秒，而使用32位计数器时为17179869秒。

### configIDLE\_SHOULD\_YIELD

此参数控制处于空闲优先级的任务的行为。它仅在以下情况下有效：

1. 使用抢占式调度程序。
2. 程序内创建了与空闲优先级相同优先级运行的任务。

如果configUSE\_TIME\_SLICING设置为1（或未定义），则相同优先级的任务将使用时间片。如果没有一个任务被抢占，则可以假定将给定优先级的每个任务分配相等的处理时间-如果优先级高于空闲优先级，则确实如此。

当任务共享空闲优先级时，行为可能会略有不同。如果将configIDLE\_SHOULD\_YIELD设置为1，则如果其他任何处于空闲优先级的任务准备运行，则空闲任务将立即产生。当应用程序中的任务可用于计划时，这可确保在空闲任务中花费最少的时间。但是，此行为可能会产生不良影响（取决于您的应用程序的需求），如下所示：



上图显示了全部以空闲优先级运行的四个任务的执行模式。任务A，B和C是应用程序任务。任务I是空闲任务。在时间T0，T1，...，T6定期进行上下文切换。当闲置任务产生时，任务A开始执行-但闲置任务已经消耗了一些当前时间片。这导致任务I和任务A有效地共享同一时间片。因此，应用程序任务B和C比应用程序任务A获得更多的处理时间。

可以通过以下方式避免这种情况：

* 如果合适，请使用[空闲钩](https://www.freertos.org/RTOS-idle-task.html)子代替空闲优先级的单独任务。
* 以高于空闲优先级的优先级创建所有应用程序任务。
* 将configIDLE\_SHOULD\_YIELD设置为0。

将configIDLE\_SHOULD\_YIELD设置为0可以防止空闲任务产生处理时间，直到其时间片结束为止。这样可以确保为所有处于空闲优先级的任务分配等量的处理时间（如果没有一个任务被抢占），但要付出总处理时间中更大一部分分配给空闲任务的代价。

### configUSE\_TASK\_NOTIFICATIONS

将configUSE\_TASK\_NOTIFICATIONS设置为1（或未定义configUSE\_TASK\_NOTIFICATIONS）将在构建中包括 [直接任务通知](https://www.freertos.org/RTOS-task-notifications.html)功能及其关联的API。

将configUSE\_TASK\_NOTIFICATIONS设置为0将从构建中移除直接任务通知功能及其关联的API。

当直接任务通知包含在构建中时，每个任务会占用8个字节的额外RAM。

### configTASK\_NOTIFICATION\_ARRAY\_ENTRIES

每个RTOS任务都有[一系列任务通知](https://www.freertos.org/RTOS-task-notifications.html)。configTASK\_NOTIFICATION\_ARRAY\_ENTRIES设置数组中的索引数。

在FreeRTOS V10.4.0之前的版本中，任务只有一个通知值，而不是值数组，因此为了向后兼容，如果未定义configTASK\_NOTIFICATION\_ARRAY\_ENTRIES，则默认为1。

### configUSE\_MUTEXES

设置为1可在构建中包含互斥信号量，或设置为0可从构建中忽略互斥信号量。读者应该熟悉互斥信号量和二值信号量的功能在FreeRTOS中之间的区别。

### configUSE\_RECURSIVE\_MUTEXES

设置为1可在构建中包括递归互斥信号量，或设置为0可从构建中省略递归互斥信号量。

### configUSE\_COUNTING\_SEMAPHORES

设置为1可在构建中包括计数信号量，或设置为0可在构建中忽略计数信号量。

### configUSE\_ALTERNATIVE\_API

设置为1可以在构建中包括“替代”队列函数，设置为0可以从构建中省略“替代”队列函数。在queue.h头文件中描述了备用API。 **替代API已弃用，不应在新设计中使用**。

### configCHECK\_FOR\_STACK\_OVERFLOW

该[堆栈溢出检测](https://www.freertos.org/Stacks-and-stack-overflow-checking.html)页描述了使用这个参数。

### configQUEUE\_REGISTRY\_SIZE

队列注册表具有两个目的，这两个目的都与RTOS内核感知的调试相关联：

1. 它允许将文本名称与队列关联，以便在调试GUI中轻松识别队列。
2. 它包含调试器查找每个已注册队列和信号量所需的信息。

除非使用的是RTOS内核感知调试器，否则队列注册表没有任何用途。

configQUEUE\_REGISTRY\_SIZE定义可以注册的最大队列和信号量。仅需要注册要使用RTOS内核感知调试器查看的队列和信号量。有关更多信息，请参阅[vQueueAddToRegistry（）](https://www.freertos.org/vQueueAddToRegistry.html) 和[vQueueUnregisterQueue（）](https://www.freertos.org/vQueueUnregisterQueue.html)的API参考文档。

### configUSE\_QUEUE\_SETS

设置为1以包括[队列集](https://www.freertos.org/Pend-on-multiple-rtos-objects.html)功能（阻塞或插入多个队列和信号量的功能），设置为0则忽略队列集功能。

### configUSE\_TIME\_SLICING

缺省情况下（如果未定义configUSE\_TIME\_SLICING或configUSE\_TIME\_SLICING定义为1），FreeRTOS将优先抢占式调度与时间分片一起使用。这意味着RTOS调度程序将始终运行处于“就绪”状态的最高优先级任务，并会在每个RTOS滴答中断时在优先级相同的任务之间切换。如果configUSE\_TIME\_SLICING设置为0，则RTOS调度程序仍将运行处于“就绪”状态的最高优先级任务，但不会仅由于发生滴答中断而在优先级相同的任务之间切换。

### configUSE\_NEWLIB\_REENTRANT

如果configUSE\_NEWLIB\_REENTRANT设置为1，则将为每个创建的任务分配一个[newlib](http://sourceware.org/newlib/" \t "_blank)进入结构。

注意Newlib支持已包含在普遍需求中，但FreeRTOS维护者本身并未使用。FreeRTOS不负责由此产生的newlib操作。用户必须熟悉newlib，并且必须提供必要存根的系统范围的实现。警告，（在撰写本文时）当前的newlib设计实现了系统范围内的malloc（），必须提供锁。

### configENABLE\_BACKWARD\_COMPATIBILITY

FreeRTOS.h头文件包含一组#define宏，这些宏将8.0.0之前的FreeRTOS版本中使用的数据类型的名称映射到8.0.0之前的版本中。这些宏允许应用程序代码将其构建的FreeRTOS版本从8.0.0之前的版本更新为8.0.0以后的版本，而无需进行修改。在FreeRTOSConfig.h中将configENABLE\_BACKWARD\_COMPATIBILITY设置为0会从构建中排除这些宏，从而允许进行验证，以确保未使用8.0.0之前的版本。

### configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS

设置每个任务的[线程本地存储数组中](https://www.freertos.org/thread-local-storage-pointers.html)的索引数 。

### configSTACK\_DEPTH\_TYPE

设置用于在对[xTaskCreate（）的](https://www.freertos.org/a00125.html)调用中指定堆栈深度的类型 ，并使用其他各种堆栈大小（例如，在返回 [堆栈高水位标记时](https://www.freertos.org/uxTaskGetStackHighWaterMark.html)）。

FreeRTOS的较旧版本使用UBaseType\_t类型的变量指定了堆栈大小，但是发现这对8位微控制器的限制太大。configSTACK\_DEPTH\_TYPE通过允许应用程序开发人员指定要使用的类型来消除该限制。

### configMESSAGE\_BUFFER\_LENGTH\_TYPE

[FreeRTOS消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)使用configMESSAGE\_BUFFER\_LENGTH\_TYPE类型的变量来存储每个消息的长度。如果未定义configMESSAGE\_BUFFER\_LENGTH\_TYPE，则默认为size\_t。如果存储在消息缓冲区中的消息永远不会大于255个字节，那么在32位微控制器上将configMESSAGE\_BUFFER\_LENGTH\_TYPE定义为uint8\_t将为每个消息节省3个字节。同样，如果存储在消息缓冲区中的消息永远不会大于65535字节，则在32位微控制器上将configMESSAGE\_BUFFER\_LENGTH\_TYPE定义为uint16\_t将为每条消息节省2个字节。

### configSUPPORT\_STATIC\_ALLOCATION

如果将configSUPPORT\_STATIC\_ALLOCATION设置为1，则可以使用应用程序编写器提供的RAM创建RTOS对象。

如果configSUPPORT\_STATIC\_ALLOCATION设置为0，则只能使用从FreeRTOS堆分配的RAM创建RTOS对象。

如果未定义configSUPPORT\_STATIC\_ALLOCATION，则默认为0。

如果configSUPPORT\_STATIC\_ALLOCATION设置为1，则应用程序编写器还必须提供两个回调函数：vApplicationGetIdleTaskMemory（）以提供供RTOS空闲任务使用的内存，以及（如果configUSE\_TIMERS设置为1）vApplicationGetTimerTaskMemory（）提供以供使用的内存RTOS守护程序/计时器服务任务。下面提供示例。

**/\* configSUPPORT\_STATIC\_ALLOCATION is set to 1, so the application must provide an**

**implementation of vApplicationGetIdleTaskMemory() to provide the memory that is**

**used by the Idle task. \*/**

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

**StackType\_t \*\*ppxIdleTaskStackBuffer,**

**uint32\_t \*pulIdleTaskStackSize )**

**{**

**/\* If the buffers to be provided to the Idle task are declared inside this**

**function then they must be declared static – otherwise they will be allocated on**

**the stack and so not exists after this function exits. \*/**

**static StaticTask\_t xIdleTaskTCB;**

**static StackType\_t uxIdleTaskStack[ configMINIMAL\_STACK\_SIZE ];**

**/\* Pass out a pointer to the StaticTask\_t structure in which the Idle task’s**

**state will be stored. \*/**

**\*ppxIdleTaskTCBBuffer = &xIdleTaskTCB;**

**/\* Pass out the array that will be used as the Idle task’s stack. \*/**

**\*ppxIdleTaskStackBuffer = uxIdleTaskStack;**

**/\* Pass out the size of the array pointed to by \*ppxIdleTaskStackBuffer.**

**Note that, as the array is necessarily of type StackType\_t,**

**configMINIMAL\_STACK\_SIZE is specified in words, not bytes. \*/**

**\*pulIdleTaskStackSize = configMINIMAL\_STACK\_SIZE;**

**}**

**/\*———————————————————–\*/**

**/\* configSUPPORT\_STATIC\_ALLOCATION and configUSE\_TIMERS are both set to 1, so the**

**application must provide an implementation of vApplicationGetTimerTaskMemory()**

**to provide the memory that is used by the Timer service task. \*/**

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

**StackType\_t \*\*ppxTimerTaskStackBuffer,**

**uint32\_t \*pulTimerTaskStackSize )**

**{**

**/\* If the buffers to be provided to the Timer task are declared inside this**

**function then they must be declared static – otherwise they will be allocated on**

**the stack and so not exists after this function exits. \*/**

**static StaticTask\_t xTimerTaskTCB;**

**static StackType\_t uxTimerTaskStack[ configTIMER\_TASK\_STACK\_DEPTH ];**

**/\* Pass out a pointer to the StaticTask\_t structure in which the Timer**

**task’s state will be stored. \*/**

**\*ppxTimerTaskTCBBuffer = &xTimerTaskTCB;**

**/\* Pass out the array that will be used as the Timer task’s stack. \*/**

**\*ppxTimerTaskStackBuffer = uxTimerTaskStack;**

**/\* Pass out the size of the array pointed to by \*ppxTimerTaskStackBuffer.**

**Note that, as the array is necessarily of type StackType\_t,**

**configTIMER\_TASK\_STACK\_DEPTH is specified in words, not bytes. \*/**

**\*pulTimerTaskStackSize = configTIMER\_TASK\_STACK\_DEPTH;**

**}**

**Examples of the callback functions that must be provided by the application to**

**supply the RAM used by the Idle and Timer Service tasks if configSUPPORT\_STATIC\_ALLOCATION**

**is set to 1.**

有关 更多信息，请参见“[静态Vs动态内存分配”](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

### configSUPPORT\_DYNAMIC\_ALLOCATION

如果configSUPPORT\_DYNAMIC\_ALLOCATION设置为1，则可以使用从FreeRTOS堆自动分配的RAM创建RTOS对象。

如果configSUPPORT\_DYNAMIC\_ALLOCATION设置为0，则只能使用应用程序编写器提供的RAM创建RTOS对象。

如果未定义configSUPPORT\_DYNAMIC\_ALLOCATION，则默认为1。

有关 更多信息，请参见“[静态Vs动态内存分配”](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

### configTOTAL\_HEAP\_SIZE

FreeRTOS堆中可用的RAM总量。

仅当[configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)设置为1并且应用程序使用FreeRTOS源代码下载中提供的示例内存分配方案之一时，才使用此值。有关更多详细信息，请参见[内存配置](https://www.freertos.org/a00111.html)部分。

### configAPPLICATION\_ALLOCATED\_HEAP

默认情况下，[FreeRTOS堆](https://www.freertos.org/a00111.html)由FreeRTOS声明，并由链接器放置在内存中。将configAPPLICATION\_ALLOCATED\_HEAP设置为1可以使堆改为由应用程序编写器声明，这允许应用程序编写器将堆放在内存中的任意位置。

如果使用heap\_1.c，heap\_2.c或heap\_4.c，并且configAPPLICATION\_ALLOCATED\_HEAP设置为1，则应用程序编写器必须提供具有确切名称和维的uint8\_t数组，如下所示。该数组将用作FreeRTOS堆。如何将数组放置在特定的内存位置取决于所使用的编译器–请参阅编译器的文档。

**uint8\_t ucHeap [configTOTAL\_HEAP\_SIZE];**

### configGENERATE\_RUN\_TIME\_STATS

“[运行时状态”](https://www.freertos.org/rtos-run-time-stats.html)页面描述了此参数的用法。

### configUSE\_CO\_ROUTINES

设置为1可在构建中包含协同例程功能，设置为0可从构建中省略协同例程功能。要包含协例程，必须在项目中包含croutine.c。

### configMAX\_CO\_ROUTINE\_PRIORITIES

可用于应用程序协同程序 的[优先](https://www.freertos.org/co-routine-priorities.html)级数。任意数量的协同例程可以共享相同的优先级。任务分别进行优先级排序-请参阅configMAX\_PRIORITIES。

### configUSE\_TIMERS

设置为1以包括软件计时器功能，或设置为0以省略软件计时器功能。有关完整说明，请参见[FreeRTOS软件计时器](https://www.freertos.org/RTOS-software-timer.html)页面。

### configTIMER\_TASK\_PRIORITY

设置软件计时器服务/守护程序任务的优先级。有关完整说明，请参见[FreeRTOS软件计时器](https://www.freertos.org/RTOS-software-timer.html)页面。

### configTIMER\_QUEUE\_LENGTH

设置软件计时器命令队列的长度。有关完整说明，请参见[FreeRTOS软件计时器](https://www.freertos.org/RTOS-software-timer.html)页面。

### configTIMER\_TASK\_STACK\_DEPTH

设置分配给软件计时器服务/守护程序任务的堆栈深度。有关完整说明，请参见[FreeRTOS软件计时器](https://www.freertos.org/RTOS-software-timer.html)页面。

### configKERNEL\_INTERRUPT\_PRIORITY configMAX\_SYSCALL\_INTERRUPT\_PRIORITY和 configMAX\_API\_CALL\_INTERRUPT\_PRIORITY

包含configKERNEL\_INTERRUPT\_PRIORITY设置的端口包括ARM Cortex-M3，PIC24，dsPIC，PIC32，SuperH和RX600。包含configMAX\_SYSCALL\_INTERRUPT\_PRIORITY设置的端口包括PIC32，RX600，ARM Cortex-A和ARM Cortex-M端口。

ARM Cortex-M3和ARM Cortex-M4用户请注意本节末尾的特别说明！

configMAX\_API\_CALL\_INTERRUPT\_PRIORITY是configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的新名称，仅由较新的端口使用。两者是等效的。

configKERNEL\_INTERRUPT\_PRIORITY应该设置为最低优先级。

请注意，在以下讨论中，只能从中断服务例程中调用以“ FromISR”结尾的API函数。

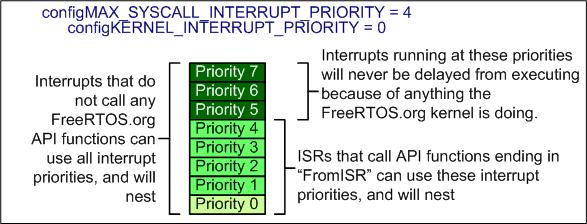
对于仅实现configKERNEL\_INTERRUPT\_PRIORITY的端口，  
configKERNEL\_INTERRUPT\_PRIORITY设置RTOS内核本身使用的中断优先级。调用API函数的中断也必须以此优先级执行。不调用API函数的中断可以以更高的优先级执行，因此不会因RTOS内核活动（在硬件本身的限制内）而延迟执行。

对于同时实现configKERNEL\_INTERRUPT\_PRIORITY和configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的端口：  
configKERNEL\_INTERRUPT\_PRIORITY设置RTOS内核本身使用的中断优先级。configMAX\_SYSCALL\_INTERRUPT\_PRIORITY设置最高的中断优先级，从中可以调用中断安全的FreeRTOS API函数。

通过将configMAX\_SYSCALL\_INTERRUPT\_PRIORITY设置为高于configKERNEL\_INTERRUPT\_PRIORITY（即，以更高的优先级），可以实现完整的中断嵌套模型。 **这意味着FreeRTOS内核甚至在关键部分内部也不会完全禁用中断。** 此外，这没有分段内核体系结构的缺点。但是请注意，某些微控制器体系结构将在接受新中断时（在硬件中）禁用中断，这意味着在硬件接受中断与FreeRTOS代码重新启用中断之间的短时间内不可避免地禁用了中断。

不调用API函数的中断可以以高于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的优先级执行，因此不会因RTOS内核执行而延迟。

例如，假设一个假设的微控制器具有8个中断优先级-0是最低的，7是最高的（请参阅本节末尾针对ARM Cortex-M3用户的特别说明）。下图描述了如果将两个配置常量设置为4和0，则在每个优先级上可以做什么和不能做什么，如下所示：

  
中断优先级配置示例

这些配置参数允许非常灵活的中断处理：

* 可以按照系统中的其他任何任务编写并优先处理中断处理“任务”。这些是被中断唤醒的任务。中断服务程序（ISR）本身应写得尽可能短–它只获取数据，然后唤醒高优先级处理程序任务。然后，ISR直接返回到唤醒的处理程序任务中，因此中断处理在时间上是连续的，就像所有中断都在ISR本身中完成一样。这样做的好处是，在执行处理程序任务时，所有中断都保持启用状态。
* 实现configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的端口将进一步扩展–允许使用完全嵌套的模型，其中RTOS内核中断优先级和configMAX\_SYSCALL\_INTERRUPT\_PRIORITY之间的中断可以嵌套并进行适用的API调用。优先级高于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的中断不会因RTOS内核活动而延迟。
* 在最高系统调用优先级之上运行的ISR永远不会被RTOS内核本身掩盖，因此它们的响应能力不受RTOS内核功能的影响。对于需要非常高的时间精度的中断（例如执行电动机换向的中断）而言，这是理想的选择。但是，这样的ISR无法使用FreeRTOS API函数。

要使用此方案，您的应用程序设计必须遵守以下规则：**使用FreeRTOS API的任何中断必须设置为与RTOS内核相同的优先级（由configKERNEL\_INTERRUPT\_PRIORITY宏配置），或者对于以下端口，configMAX\_SYSCALL\_INTERRUPT\_PRIORITY或以下：此功能。**

针对ARM Cortex-M3和ARM Cortex-M4用户的特别说明：请阅读 [专门介绍ARM Cortex-M设备上的中断优先级设置的页面](https://www.freertos.org/RTOS-Cortex-M3-M4.html)。至少要记住，ARM Cortex-M3内核使用数字低优先级数字来表示高优先级中断，这看起来似乎违反直觉，而且很容易忘记！如果您希望为中断分配一个低优先级，请不要为其分配优先级0（或其他低数值），因为这可能导致中断实际上在系统中具有最高优先级–因此，如果这会导致系统崩溃优先级高于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY。

实际上，ARM Cortex-M3内核的最低优先级为255 –但是，不同的ARM Cortex-M3供应商实现了不同数量的优先级位，并且提供了期望以不同方式指定优先级的库函数。例如，在STM32上，您可以在ST驱动程序库调用中指定的最低优先级实际上为15 –您可以指定的最高优先级为0。

### configASSERT

configASSERT（）宏的语义与标准C assert（）宏相同。如果传递给configASSERT（）的参数为零，则触发断言。

在整个FreeRTOS源文件中都会调用configASSERT（）来检查应用程序如何使用FreeRTOS。强烈建议使用定义的configASSERT（）开发FreeRTOS应用程序。

示例定义（显示在文件顶部，并在下面复制）调用vAssertCalled（），并传入触发configASSERT（）调用的文件名和行号（\_\_FILE\_\_和\_\_LINE\_\_是大多数编译器提供的标准宏）。这只是为了演示，因为vAssertCalled（）不是FreeRTOS函数，可以将configASSERT（）定义为采取应用程序编写者认为适当的任何操作。

定义configASSERT（）的方式是正常的，这样可以防止应用程序进一步执行。这有两个原因；在断言时停止应用程序可以调试断言的原因，执行经过触发的断言可能仍会导致崩溃。

请注意，定义configASSERT（）会增加应用程序代码的大小和执行时间。当应用程序稳定后，只需注释掉FreeRTOSConfig.h中的configASSERT（）定义，就可以消除额外的开销。

**/\* Define configASSERT() to call vAssertCalled() if the assertion fails. The assertion**

**has failed if the value of the parameter passed into configASSERT() equals zero. \*/**

**#define configASSERT ( x ) if( ( x ) == 0 ) vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )**

If running FreeRTOS under the control of a debugger, then configASSERT() can be defined to just disable interrupts and sit in a loop, as demonstrated below. That will have the effect of stopping the code on the line that failed the assert test – pausing the debugger will then immediately take you to the offending line so you can see why it failed.

**/\* Define configASSERT() to disable interrupts and sit in a loop. \*/**

**#define configASSERT ( x ) if( ( x ) == 0 ) { taskDISABLE\_INTERRUPTS(); for( ;; ); }**

### configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS

configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS仅由FreeRTOS MPU使用。

如果configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS设置为1，则应用程序编写器必须提供一个名为“ application\_defined\_privileged\_functions.h”的头文件，可以在其中实现应用程序编写器需要以特权模式执行的功能。请注意，尽管扩展名为.h，但头文件应包含C函数的实现，而不仅仅是函数的原型。

在“ application\_defined\_privileged\_functions.h”中实现的函数必须分别使用prvRaisePrivilege（）函数和portRESET\_PRIVILEGE（）宏来保存和恢复处理器的特权状态。例如，如果提供了打印功能的库访问了应用程序编写器无法控制的RAM，因此无法将其分配给受存储器保护的用户模式任务，则可以使用以下代码将打印功能封装在特权功能中：

**void MPU\_debug\_printf( const char \*pcMessage )**

**{**

**/\* State the privilege level of the processor when the function was called. \*/**

**BaseType\_t xRunningPrivileged = prvRaisePrivilege();**

**/\* Call the library function, which now has access to all RAM. \*/**

**debug\_printf( pcMessage );**

**/\* Reset the processor privilege level to its original value. \*/**

**portRESET\_PRIVILEGE( xRunningPrivileged );**

**}**

该技术只能在开发过程中使用，而不能在部署过程中使用，因为它会绕过内存保护。

### configTOTAL\_MPU\_REGIONS

用于ARM Cortex-M4微控制器的FreeRTOS MPU（内存保护单元）端口支持具有16个MPU区域的设备。对于具有16个MPU区域的设备，将configTOTAL\_MPU\_REGIONS设置为16。如果未定义，则默认为8。

### configTEX\_S\_C\_B\_FLASH

TEX，可共享（S），可缓存（C）和可缓冲（B）位定义内存类型，并在必要时定义MPU区域的可缓存和可共享属性。configTEX\_S\_C\_B\_FLASH 允许应用程序编写者覆盖TEX的默认值，覆盖Flash的MPU区的Shareable（S），Cacheable（C）和Bufferable（B）位。如果未定义，则默认为0x07UL，这意味着TEX = 000，S = 1，C = 1，B = 1。

### configTEX\_S\_C\_B\_SRAM

TEX，可共享（S），可缓存（C）和可缓冲（B）位定义内存类型，并在必要时定义MPU区域的可缓存和可共享属性。configTEX\_S\_C\_B\_SRAM 允许应用程序编写者覆盖TEX的默认值，覆盖RAM的MPU区域的Shareable（S），Cacheable（C）和Bufferable（B）位。如果未定义，则默认为0x07UL，这意味着TEX = 000，S = 1，C = 1，B = 1。

### configENFORCE\_SYSTEM\_CALLS\_FROM\_KERNEL\_ONLY

可以将configENFORCE\_SYSTEM\_CALLS\_FROM\_KERNEL\_ONLY定义为1，以防止源自内核代码外部的任何特权升级（输入中断后由硬件本身执行的升级除外）。当configENFORCE\_SYSTEM\_CALLS\_FROM\_KERNEL\_ONLY 在设置为1 FreeRTOSConfig.h中，变量\_\_syscalls\_flash\_start\_\_和 \_\_syscalls\_flash\_end\_\_需要导出表单链接脚本分别表示该系统调用内存的起始和结束地址。为了获得最大的安全性，建议将其定义为1。

## 包含参数

以“ INCLUDE”开头的宏允许将实时内核中那些未被应用程序使用的组件从构建中排除。这样可以确保RTOS使用的ROM或RAM不会超出特定嵌入式应用程序所需的数量。

每个宏采用以下形式：

**INCLUDE\_FunctionName**

…其中FunctionName表示可以选择排除的API函数（或函数集）。要包含API函数，请将宏设置为1，以排除函数将宏设置为0。例如，包含vTaskDelete（）API函数，请使用：

**#define INCLUDE\_vTaskDelete 1**

要从构建中排除vTaskDelete（），请使用：

**#define INCLUDE\_vTaskDelete 0**

# 内存管理

[另请参见[静态与动态内存分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面，该页面描述了静态（不使用FreeRTOS堆）或动态分配RTOS对象的[优缺点](https://www.freertos.org/a00110.html" \l "configAPPLICATION_ALLOCATED_HEAP) ，以及可在FreeRTOSConfig.h中定义的[configAPPLICATION\_ALLOCATED\_HEAP](https://www.freertos.org/a00110.html" \l "configAPPLICATION_ALLOCATED_HEAP)常数 的说明。[参考项目，演示如何在没有堆实现的情况下使用FreeRTOS](https://www.freertos.org/freertos-static-allocation-demo.html)。]

每次创建任务，队列，互斥锁，软件计时器，信号灯或事件组时，RTOS内核都需要RAM。可以从RTOS API对象创建函数中的RTOS堆自动动态分配RAM，也可以[由应用程序编写者提供](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)RAM 。

如果RTOS对象是动态创建的，则有时可以将标准C库malloc（）和free（）函数用于此目的，但是**…**

1. 它们并不总是在嵌入式系统上可用，
2. 他们占用了宝贵的代码空间，
3. 它们不是线程安全的，并且
4. 它们不是确定性的（执行函数所需的时间因调用而异）

**…**经常需要替代的内存分配实现。

一个嵌入式/实时系统可能具有与另一个系统不同的RAM和时序要求-因此，单一的RAM分配算法仅适用于部分应用程序。

为了解决这个问题，FreeRTOS将内存分配API保留在其可移植层中。可移植层位于实现核心RTOS功能的源文件之外，从而允许提供适合于正在开发的实时系统的特定于应用程序的实现。当RTOS内核需要RAM时，它不调用malloc（），而是调用pvPortMalloc（）。当释放RAM时，RTOS内核将调用vPortFree（）而不是调用free（）。

FreeRTOS提供了几种堆管理方案，这些方案的复杂性和功能范围广泛。也可以提供自己的堆实现，甚至可以同时使用两个堆实现。同时使用两个堆实现允许将任务堆栈和其他RTOS对象放置在快速内部RAM中，并将应用程序数据放置在较慢的外部RAM中。

## RTOS源代码下载中包含的内存分配实现

FreeRTOS下载包括五个样本内存分配实现，以下各小节中介绍了每个实现。这些小节还包含有关何时提供的每个实现可能最适合选择的信息。

每个提供的实现都包含在单独的源文件中（分别为heap\_1.c，heap\_2.c，heap\_3.c，heap\_4.c和heap\_5.c），这些文件位于主RTOS源代码下载的 Source / Portable / MemMang目录中。 。可以根据需要添加其他实现。这些源文件中的一个文件应该一次包含在项目中[由这些可移植层功能定义的堆将由RTOS内核使用，即使使用RTOS的应用程序选择使用其自己的堆实现]。

以下内容：

* [heap\_1](https://www.freertos.org/a00111.html#heap_1) –最简单，不允许释放内存。
* [heap\_2](https://www.freertos.org/a00111.html#heap_2) –允许释放内存，但不合并相邻的空闲块。
* [heap\_3](https://www.freertos.org/a00111.html#heap_3) –简单地包装标准malloc（）和free（）以确保线程安全。
* [heap\_4](https://www.freertos.org/a00111.html#heap_4) –合并相邻的空闲块以避免碎片。包括绝对地址放置选项。
* [heap\_5](https://www.freertos.org/a00111.html#heap_5) –按照heap\_4，能够跨多个不相邻的内存区域分布堆。

笔记：

* 因为FreeRTOS添加了[对静态分配的支持，](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)所以heap\_1的用处不大。
* 现在，heap\_2被认为是遗留的，因为首选更新的heap\_4实现。

### heap\_1.c

 因为FreeRTOS添加了[对静态分配的支持，](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)所以heap\_1的用处不大。

heap\_1是所有方法中最简单的实现。它*不*容许一旦被分配到被释放的内存。尽管如此，heap\_1.c适用于大量嵌入式应用程序。这是因为许多小型且深度嵌入的应用程序会在系统启动时创建所有必需的任务，队列，信号量等，然后在程序生命周期内使用所有这些对象（直到再次关闭该应用程序或重新启动该应用程序）。什么都不会被删除。

该实现仅在请求RAM时将单个阵列细分为较小的块。阵列的总大小（堆的总大小）由configTOTAL\_HEAP\_SIZE设置-在FreeRTOSConfig.h中定义。提供了[configAPPLICATION\_ALLOCATED\_HEAP](https://www.freertos.org/a00110.html" \l "configAPPLICATION_ALLOCATED_HEAP) FreeRTOSConfig.h配置常量，以允许将堆放置在内存中的特定地址处。

xPortGetFreeHeapSize（）API函数返回仍未分配的堆空间总量，从而可以优化configTOTAL\_HEAP\_SIZE设置。

heap\_1实现：

* 如果您的应用程序从不删除任务，队列，信号灯，互斥锁等（实际上涵盖了使用FreeRTOS的大多数应用程序），则可以使用。
* 始终是确定性的（总是花费相同的时间来执行），并且不会导致内存碎片。
* 它是从静态分配的数组中分配的非常简单的内存，这意味着它通常适用于不允许真正的动态内存分配的应用程序。

### 堆\_2.c

 现在首选使用heap\_2，因为首选使用heap\_4。

heap\_2使用最佳拟合算法，并且与方案1不同，它允许释放先前分配的块。它并*不*相邻的空闲块合并成一个单一的大的块。有关执行无结合块的实现，请参见heap\_4.c。

可用堆空间的总量由configTOTAL\_HEAP\_SIZE设置-在FreeRTOSConfig.h中定义。提供了[configAPPLICATION\_ALLOCATED\_HEAP](https://www.freertos.org/a00110.html" \l "configAPPLICATION_ALLOCATED_HEAP) FreeRTOSConfig.h配置常量，以允许将堆放置在内存中的特定地址处。

xPortGetFreeHeapSize（）API函数返回仍未分配的堆空间总量（允许优化configTOTAL\_HEAP\_SIZE设置），但未提供有关如何将未分配的内存分成较小的块的信息。

此实现：

* 即使应用程序反复删除任务，队列，信号量，互斥对象等，也可以使用，以下有关内存碎片的警告。
* 如果*不是*，如果被分配和释放内存使用是随机的大小。例如：
  + 如果应用程序动态创建和删除任务，并且分配给正在创建的任务的堆栈大小始终相同，则在大多数情况下可以使用heap2.c。但是，如果分配给正在创建的任务的堆栈大小并不总是相同，则可用的空闲内存可能会分成许多小块，最终导致分配失败。在这种情况下，heap\_4.c会更好。
  + 如果应用程序动态创建和删除队列，并且每种情况下队列存储区域都相同（队列存储区域是队列项目大小乘以队列长度），则在大多数情况下都可以使用heap\_2.c。但是，如果每种情况下队列存储区域都不相同，则可用的空闲内存可能会分成许多小块，最终导致分配失败。在这种情况下，heap\_4.c会更好。
  + 该应用程序直接调用pvPortMalloc（）和vPortFree（），而不仅仅是通过其他FreeRTOS API函数间接调用。
* 如果您的应用程序以不可预测的顺序排队，任务，信号量，互斥锁等，可能会导致内存碎片问题。对于几乎所有应用程序来说这都是不可能的，但应牢记。
* 不确定，但比大多数标准C库malloc实现要高效得多。

heap\_2.c适用于许多必须动态创建对象的小型实时系统。有关将空闲内存块组合成单个较大块的类似实现，请参见heap\_4。

### heap\_3.c

 这为标准C库malloc（）和free（）函数实现了一个简单的包装，在大多数情况下，这些包装将随您选择的编译器一起提供。包装器只是使malloc（）和free（）函数线程安全。

此实现：

* 需要链接器设置堆，并且需要编译器库提供malloc（）和free（）实现。
* 是不确定的。
* 可能会大大增加RTOS内核代码的大小。

注意，当使用heap\_3时，FreeRTOSConfig.h中的configTOTAL\_HEAP\_SIZE设置无效。

### heap\_4.c

 该方案使用了第一拟合算法，并且与方案2不同，它确实将相邻的空闲内存块组合为一个大块（它确实包括合并算法）。

可用堆空间的总量由configTOTAL\_HEAP\_SIZE设置-在FreeRTOSConfig.h中定义。提供了[configAPPLICATION\_ALLOCATED\_HEAP](https://www.freertos.org/a00110.html" \l "configAPPLICATION_ALLOCATED_HEAP) FreeRTOSConfig.h配置常量，以允许将堆放置在内存中的特定地址处。

xPortGetFreeHeapSize（）API函数返回调用该函数时仍未分配的堆空间总量，而xPortGetMinimumEverFreeHeapSize（）API函数返回FreeRTOS应用程序已启动的系统中已存在的最小可用堆空间。这两个函数均未提供有关如何将未分配的内存分成较小的块的信息。

vPortGetHeapStats（）API函数提供了其他信息。它将填充heap\_t结构的成员，如下所示。

**/\* Prototype of the vPortGetHeapStats() function. \*/**

**void vPortGetHeapStats( HeapStats\_t \*xHeapStats );**

**/\* Definition of the Heap\_stats\_t structure. \*/**

**typedef struct xHeapStats**

**{**

**size\_t xAvailableHeapSpaceInBytes; /\* The total heap size currently available – this is the sum of all the free blocks, not the largest block that can be allocated. \*/**

**size\_t xSizeOfLargestFreeBlockInBytes; /\* The maximum size, in bytes, of all the free blocks within the heap at the time vPortGetHeapStats() is called. \*/**

**size\_t xSizeOfSmallestFreeBlockInBytes; /\* The minimum size, in bytes, of all the free blocks within the heap at the time vPortGetHeapStats() is called. \*/**

**size\_t xNumberOfFreeBlocks; /\* The number of free memory blocks within the heap at the time vPortGetHeapStats() is called. \*/**

**size\_t xMinimumEverFreeBytesRemaining; /\* The minimum amount of total free memory (sum of all free blocks) there has been in the heap since the system booted. \*/**

**size\_t xNumberOfSuccessfulAllocations; /\* The number of calls to pvPortMalloc() that have returned a valid memory block. \*/**

**size\_t xNumberOfSuccessfulFrees; /\* The number of calls to vPortFree() that has successfully freed a block of memory. \*/**

**} HeapStats\_t;**

heap\_4：

* 即使应用程序反复删除任务，队列，信号量，互斥量等，也可以使用。
* 与堆\_2实现相比，导致堆空间严重碎片化成多个小块的可能性要小得多，即使分配和释放的内存大小是随机的。
* 不确定，但比大多数标准C库malloc实现要高效得多。

对于希望直接在应用程序代码中使用便携式层内存分配方案的应用程序，heap\_4.c尤其有用（而不是仅通过调用本身调用pvPortMalloc（）和vPortFree（）的API函数来间接使用）。

### heap\_5.c

 该方案使用与heap\_4相同的首次拟合和内存合并算法，并允许堆跨越多个不相邻（不连续）的内存区域。

Heap\_5通过调用vPortDefineHeapRegions（）进行初始化，只有 在执行vPortDefineHeapRegions（）之后**才能使用**。创建RTOS对象（任务，队列，信号量等）将隐式调用pvPortMalloc（），因此在使用heap\_5时，必须在创建任何此类对象之前先调用vPortDefineHeapRegions（）。

vPortDefineHeapRegions（）使用单个参数。该参数是HeapRegion\_t结构的数组。HeapRegion\_t在Portable.h中定义为

**typedef struct HeapRegion**

**{**

**/\* Start address of a block of memory that will be part of the heap.\*/**

**uint8\_t \*pucStartAddress;**

**/\* Size of the block of memory. \*/**

**size\_t xSizeInBytes;**

**} HeapRegion\_t;**

**The HeapRegion\_t type definition**

数组使用零大小的NULL区域终止，并且数组中定义的存储区**必须**按地址顺序出现，从低地址到高地址。以下源代码片段提供了一个示例。的 [MSVC的Win32模拟器演示](https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html) 也用途heap\_5所以可以被用作一个参考。

**/\* Allocate two blocks of RAM for use by the heap. The first is a block of**

**0x10000 bytes starting from address 0x80000000, and the second a block of**

**0xa0000 bytes starting from address 0x90000000. The block starting at**

**0x80000000 has the lower start address so appears in the array fist. \*/**

**const HeapRegion\_t xHeapRegions[] =**

**{**

**{ ( uint8\_t \* ) 0x80000000UL, 0x10000 },**

**{ ( uint8\_t \* ) 0x90000000UL, 0xa0000 },**

**{ NULL, 0 } /\* Terminates the array. \*/**

**};**

**/\* Pass the array into vPortDefineHeapRegions(). \*/**

**vPortDefineHeapRegions( xHeapRegions );**

**Initialising heap\_5 after defining the memory blocks to be used by the heap**

xPortGetFreeHeapSize（）API函数返回调用该函数时仍未分配的堆空间总量，而xPortGetMinimumEverFreeHeapSize（）API函数返回FreeRTOS应用程序已启动的系统中已存在的最小可用堆空间。这两个函数均未提供有关如何将未分配的内存分成较小的块的信息。

所述[vPortGetHeapStats（）](https://www.freertos.org/a00111.html#xPortGetFreeHeapSize) API函数提供在堆上的状态的附加信息。

# 堆栈使用情况和堆栈溢出检查

### 堆栈使用

[另请参见[uxTaskGetStackHighWaterMark（）](https://www.freertos.org/uxTaskGetStackHighWaterMark.html) API函数]

每个任务维护自己的堆栈。如果使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务，则将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配用作任务堆栈的内存 ，并通过传递给xTaskCreate（）API函数的参数确定其大小。如果使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建 任务，则由应用程序编写者预先分配用作任务堆栈的内存。堆栈溢出是应用程序不稳定的非常常见的原因。因此，FreeRTOS提供了两种可选机制，可用于协助检测和纠正此类事件。使用的选项是使用 [configCHECK\_FOR\_STACK\_OVERFLOW配置常量配置的](https://www.freertos.org/a00110.html)。

请注意，这些选项仅在未对内存映射进行分段的体系结构上可用。此外，在发生RTOS内核溢出检查之前，某些处理器可能会响应堆栈损坏而生成错误或异常。如果configCHECK\_FOR\_STACK\_OVERFLOW未设置为0，则应用程序必须提供堆栈溢出挂钩函数。该挂钩函数必须名为vApplicationStackOverflowHook（），并具有以下原型：

**void vApplicationStackOverflowHook( TaskHandle\_t xTask,**

**signed char \*pcTaskName );**

xTask和pcTaskName参数分别将有问题的任务的句柄和名称传递给hook函数。但是请注意，根据溢出的严重程度，这些参数本身可能会损坏，在这种情况下，可以直接检查pxCurrentTCB变量。

堆栈溢出检查会带来上下文切换开销，因此仅在开发或测试阶段才建议使用它。

### 堆栈溢出检测-方法1

在RTOS内核将任务从“运行”状态换出后，堆栈很可能会达到其最大（最深）值，因为此时堆栈将包含任务上下文。此时，RTOS内核可以检查处理器堆栈指针是否保留在有效堆栈空间内。如果堆栈指针包含的值超出有效堆栈范围，则调用堆栈溢出挂钩函数。

此方法很快，但不能保证捕获所有堆栈溢出。将configCHECK\_FOR\_STACK\_OVERFLOW设置为1即可使用此方法。

### 堆栈溢出检测–方法2

首次创建任务时，其堆栈会填充一个已知值。当将任务从运行状态换出时，RTOS内核可以检查有效堆栈范围内的最后16个字节，以确保任务或中断活动未覆盖这些已知值。如果这16个字节中的任何一个不保持其初始值，则调用堆栈溢出钩子函数。

此方法的效率比方法一低，但仍然相当快。它很可能捕获堆栈溢出，但仍不能保证捕获所有溢出。

将configCHECK\_FOR\_STACK\_OVERFLOW设置为2以使用此方法。

# Task Creation [[API](https://www.freertos.org/a00106.html)]

## Modules

* [**xTaskCreate**](https://www.freertos.org/a00125.html)
* [**xTaskCreateStatic**](https://www.freertos.org/xTaskCreateStatic.html)
* [**vTaskDelete**](https://www.freertos.org/a00126.html)

# 详细说明

### TaskHandle\_t task. h

引用任务的类型。例如，对xTaskCreate的调用（通过指针参数）返回TaskHandle\_t变量，然后可以将该变量用作vTaskDelete的参数以删除任务。

# xTaskCreate [[Task Creation](https://www.freertos.org/a00019.html)]

task. h

**BaseType\_t xTaskCreate( TaskFunction\_t pvTaskCode,**

**const char \* const pcName,**

[**configSTACK\_DEPTH\_TYPE**](https://www.freertos.org/a00110.html#configSTACK_DEPTH_TYPE) **usStackDepth,**

**void \*pvParameters,**

**UBaseType\_t uxPriority,**

**TaskHandle\_t \*pxCreatedTask**

**);**

创建一个新[任务](https://www.freertos.org/a00015.html)，并将其添加到准备运行的任务列表中。 [要使](https://www.freertos.org/a00110.html#configSUPPORT_DYNAMIC_ALLOCATION) 此RTOS API功能可用，必须在FreeRTOSConfig.h[中将configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)设置为1，或将其保留为未定义状态（在这种情况下，它将默认为1）。

每个任务都需要用于保持任务状态并由任务用作其堆栈的RAM。如果使用xTaskCreate（）创建任务，则从[FreeRTOS堆](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务，则RAM由应用程序编写器提供，因此可以在编译时静态分配。有关 更多信息，请参见 [静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

如果您使用的是[FreeRTOS-MPU](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html)，则建议使用[xTaskCreateRestricted（）](https://www.freertos.org/xTaskCreateRestricted.html) 代替xTaskCreate（）。

**参数：**

|  |  |
| --- | --- |
| *pvTaskCode* | 指向任务输入功能的指针（仅是实现任务的功能的名称，请参见下面的示例）。  任务通常 [实现为无限循环](https://www.freertos.org/implementing-a-FreeRTOS-task.html)，并且绝不能尝试从其实现函数中返回或退出。但是，任务可以 [自行删除](https://www.freertos.org/a00126.html)。 |
| *pcName* | 任务的描述性名称。这主要用于方便调试，但也可以用于 [获取任务句柄](https://www.freertos.org/a00021.html#xTaskGetHandle)。  使用[FreeRTOSConfig.h中](https://www.freertos.org/a00110.html)的configMAX\_TASK\_NAME\_LEN参数设置任务名称的最大长度。 |
| *usStackDepth* | [分配](https://www.freertos.org/a00111.html) 用作任务堆栈 的字数（不是字节！）。例如，如果堆栈为16位宽，而usStackDepth为100，则将分配200个字节用作任务的堆栈。再举一个例子，如果堆栈为32位宽，而usStackDepth为400，则将分配1600字节用作任务的堆栈。  堆栈深度乘以堆栈宽度不得超过size\_t类型变量中可以包含的最大值。  请参阅常见问题解答[堆栈应该有多大？](https://www.freertos.org/FAQMem.html#StackSize) |
| *pvParameters* | 一个值，它将作为任务的参数传递到创建的任务中。  如果将pvParameters设置为变量的地址，则在创建的任务执行时该变量必须仍然存在–因此传递堆栈变量的地址无效。 |
| *uxPriority* | 创建的任务执行 的[优先级](https://www.freertos.org/RTOS-task-priority.html)。  包含MPU支持的系统可以选择通过在uxPrriority中设置portPRIVILEGE\_BIT位在特权（系统）模式下创建任务。例如，要创建优先级为2的特权任务，请将uxPriority设置为（2 | portPRIVILEGE\_BIT）。 |
| *pxCreatedTask* | 用于通过xTaskCreate（）函数将句柄传递给创建的任务。pxCreatedTask是可选的，可以设置为NULL。 |

**返回值：**

如果任务创建成功，则返回pdPASS。否则，返回errCOULD\_NOT\_ALLOCATE\_REQUIRED\_MEMORY。

**Example usage:**

**/\* Task to be created. \*/**

**void vTaskCode( void \* pvParameters )**

**{**

**/\* The parameter value is expected to be 1 as 1 is passed in the**

**pvParameters value in the call to xTaskCreate() below.**

[**configASSERT**](https://www.freertos.org/a00110.html#configASSERT)**( ( ( uint32\_t ) pvParameters ) == 1 );**

**for( ;; )**

**{**

**/\* Task code goes here. \*/**

**}**

**}**

**/\* Function that creates a task. \*/**

**void vOtherFunction( void )**

**{**

**BaseType\_t xReturned;**

**TaskHandle\_t xHandle = NULL;**

**/\* Create the task, storing the handle. \*/**

**xReturned = xTaskCreate(**

**vTaskCode, /\* Function that implements the task. \*/**

**"NAME", /\* Text name for the task. \*/**

**STACK\_SIZE, /\* Stack size in words, not bytes. \*/**

**( void \* ) 1, /\* Parameter passed into the task. \*/**

**tskIDLE\_PRIORITY,/\* Priority at which the task is created. \*/**

**&xHandle ); /\* Used to pass out the created task's handle. \*/**

**if( xReturned == pdPASS )**

**{**

**/\* The task was created. Use the task's handle to delete the task. \*/**

[**vTaskDelete**](https://www.freertos.org/a00126.html)**( xHandle );**

**}**

**}**

# xTaskCreateStatic [[Task Creation](https://www.freertos.org/a00019.html)]

**task. h**

**TaskHandle\_t xTaskCreateStatic( TaskFunction\_t pxTaskCode,**

**const char \* const pcName,**

**const uint32\_t ulStackDepth,**

**void \* const pvParameters,**

**UBaseType\_t uxPriority,**

**StackType\_t \* const puxStackBuffer,**

**StaticTask\_t \* const pxTaskBuffer );**

创建一个新[任务](https://www.freertos.org/a00015.html)，并将其添加到准备运行的任务列表中。必须在FreeRTOSConfig.h[中将configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_STATIC_ALLOCATION)设置为1，此RTOS API函数才可用。

每个任务都需要用于保持任务状态并由任务用作其堆栈的RAM。如果使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务，则从[FreeRTOS堆](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用xTaskCreateStatic（）创建任务，则应用程序编写器将提供RAM，这会导致大量参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

如果您使用的是[FreeRTOS-MPU](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html)，则建议使用[xTaskCreateRestricted（）](https://www.freertos.org/xTaskCreateRestricted.html)代替xTaskCreateStatic（）。

**参数：**

|  |  |
| --- | --- |
| *pxTaskCode* | 指向任务输入功能的指针（仅是实现任务的功能的名称，请参见下面的示例）。  任务通常[实现为无限循环](https://www.freertos.org/implementing-a-FreeRTOS-task.html)，并且绝不能尝试从其实现函数中返回或退出。但是，任务可以[自行删除](https://www.freertos.org/a00126.html)。 |
| *pcName* | 任务的描述性名称。这主要用于方便调试，但也可以用于[获取任务句柄](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)。  使用[FreeRTOSConfig.h中](https://www.freertos.org/a00110.html)的configMAX\_TASK\_NAME\_LEN参数设置任务名称的最大长度。 |
| *ulStackDepth* | puxStackBuffer参数用于将StackType\_t变量数组传递到xTaskCreateStatic（）。ulStackDepth必须设置为数组中索引的数量。  请参阅常见问题解答[堆栈应该有多大？](https://www.freertos.org/FAQMem.html#StackSize) |
| *pvParameters* | 一个值，它将作为任务的参数传递到创建的任务中。  如果将pvParameters设置为变量的地址，则在创建的任务执行时该变量必须仍然存在–因此传递堆栈变量的地址无效。 |
| *uxPriority* | 创建的任务执行的[优先级](https://www.freertos.org/RTOS-task-priority.html)。  包含MPU支持的系统可以选择通过在uxPrriority中设置portPRIVILEGE\_BIT位在特权（系统）模式下创建任务。例如，要创建优先级为2的特权任务，请将uxPriority设置为（2 | portPRIVILEGE\_BIT）。 |
| *puxStackBuffer* | 必须指向至少具有ulStackDepth索引的StackType\_t数组（请参见上面的ulStackDepth参数）–该数组将用作任务的堆栈，因此必须是持久的（未在函数的堆栈上声明）。 |
| *pxTaskBuffer* | 必须指向StaticTask\_t类型的变量。该变量将用于保存新任务的数据结构（TCB），因此它必须是持久性的（未在函数堆栈上声明）。 |

**返回值：**

如果puxStackBuffer或pxTaskBuffer都不为NULL，则将创建任务，并返回任务的句柄。如果puxStackBuffer或pxTaskBuffer为NULL，则不会创建任务，并且将返回NULL。

**用法示例：**

**/\* Dimensions the buffer that the task being created will use as its stack.**

**NOTE: This is the number of words the stack will hold, not the number of**

**bytes. For example, if each stack item is 32-bits, and this is set to 100,**

**then 400 bytes (100 \* 32-bits) will be allocated. \*/**

**#define STACK\_SIZE 200**

**/\* Structure that will hold the TCB of the task being created. \*/**

**StaticTask\_t xTaskBuffer;**

**/\* Buffer that the task being created will use as its stack. Note this is**

**an array of StackType\_t variables. The size of StackType\_t is dependent on**

**the RTOS port. \*/**

**StackType\_t xStack[ STACK\_SIZE ];**

**/\* Function that implements the task being created. \*/**

**void vTaskCode( void \* pvParameters )**

**{**

**/\* The parameter value is expected to be 1 as 1 is passed in the**

**pvParameters value in the call to xTaskCreateStatic(). \*/**

**configASSERT( ( uint32\_t ) pvParameters == 1UL );**

**for( ;; )**

**{**

**/\* Task code goes here. \*/**

**}**

**}**

**/\* Function that creates a task. \*/**

**void vOtherFunction( void )**

**{**

**TaskHandle\_t xHandle = NULL;**

**/\* Create the task without using any dynamic memory allocation. \*/**

**xHandle = xTaskCreateStatic(**

**vTaskCode, /\* Function that implements the task. \*/**

**"NAME", /\* Text name for the task. \*/**

**STACK\_SIZE, /\* Number of indexes in the xStack array. \*/**

**( void \* ) 1, /\* Parameter passed into the task. \*/**

**tskIDLE\_PRIORITY,/\* Priority at which the task is created. \*/**

**xStack, /\* Array to use as the task's stack. \*/**

**&xTaskBuffer ); /\* Variable to hold the task's data structure. \*/**

**/\* puxStackBuffer and pxTaskBuffer were not NULL, so the task will have**

**been created, and xHandle will be the task's handle. Use the handle**

**to suspend the task. \*/**

[**vTaskSuspend**](https://www.freertos.org/a00130.html)**( xHandle );**

**}**

# vTaskDelete [[Task Creation](https://www.freertos.org/a00019.html)]

task. h

**void vTaskDelete( TaskHandle\_t xTask );**

必须将INCLUDE\_vTaskDelete定义为1才能使用此功能。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

从RTOS内核管理中删除任务。要删除的任务将从所有准备就绪，阻止，暂停和事件列表中删除。

注意：空闲任务负责从已删除的任务中释放RTOS内核分配的内存。因此，重要的是，如果您的应用程序对vTaskDelete（）进行了任何调用，请不要使空闲任务耗尽微控制器的处理时间。任务代码分配的内存不会自动释放，应该在删除任务之前释放它。

请参阅演示应用程序文件死亡。c表示使用vTaskDelete（）的示例代码。

**参数：**

|  |  |
| --- | --- |
| *xTask* | 要删除的任务的句柄。传递NULL将导致调用任务被删除。 |

**用法示例：**

**void vOtherFunction( void )**

**{**

**TaskHandle\_t xHandle = NULL;**

**// Create the task, storing the handle.**

**xTaskCreate( vTaskCode, "NAME", STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );**

**// Use the handle to delete the task.**

**if( xHandle != NULL )**

**{**

**vTaskDelete( xHandle );**

**}**

**}**

# 任务控制

## 模块

* [**vTaskDelay**](https://www.freertos.org/a00127.html)
* [**vTaskDelayUntil**](https://www.freertos.org/vtaskdelayuntil.html)
* [**uxTaskPriorityGet**](https://www.freertos.org/a00128.html)
* [**vTaskPrioritySet**](https://www.freertos.org/a00129.html)
* [**vTaskSuspend**](https://www.freertos.org/a00130.html)
* [**vTaskResume**](https://www.freertos.org/a00131.html)
* [**xTaskResumeFromISR**](https://www.freertos.org/taskresumefromisr.html)
* [**xTaskAbortDelay**](https://www.freertos.org/xTaskAbortDelay.html)

# vTaskDelay [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**void vTaskDelay( const TickType\_t xTicksToDelay );**

必须将INCLUDE\_vTaskDelay定义为1才能使用此功能。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

将任务延迟给定的滴答数。任务保持阻塞的实际时间取决于滴答率。常量portTICK\_PERIOD\_MS可用于根据滴答速率（分辨率为一个滴答周期）计算实时时间。

vTaskDelay（）指定**相对于**调用vTaskDelay（）的时间，任务希望解除阻塞的时间。例如，将阻止时间段指定为100个滴答声将使任务在调用vTaskDelay（）之后取消阻止100个滴答声。因此，vTaskDelay（）不能提供一种控制周期性任务频率的好方法，因为通过代码的路径以及其他任务和中断活动将影响vTaskDelay（）的调用频率，从而影响时间接下来执行任务的位置。请参阅vTaskDelayUntil（）以了解旨在简化固定频率执行的替代API函数。它通过指定调用任务应解除阻止的绝对时间（而不是相对时间）来实现。

**参数：**

|  |  |
| --- | --- |
| *xTicksToDelay* | 调用任务应阻塞的时间（以滴答周期为单位）。 |

**用法示例：**

**void vTaskFunction( void \* pvParameters )**

**{**

**/\* Block for 500ms. \*/**

**const TickType\_t xDelay = 500 / portTICK\_PERIOD\_MS;**

**for( ;; )**

**{**

**/\* Simply toggle the LED every 500ms, blocking between each toggle. \*/**

**vToggleLED();**

**vTaskDelay( xDelay );**

**}**

**}**

# vTaskDelayUntil [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**void vTaskDelayUntil( TickType\_t \*pxPreviousWakeTime,**

**const TickType\_t xTimeIncrement );**

必须将INCLUDE\_vTaskDelayUntil定义为1，此功能才可用。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

将任务延迟到指定时间。定期任务可以使用此功能以确保执行频率恒定。

此函数在一个重要方面与vTaskDelay（）不同：vTaskDelay（）指定*相*对于调用vTaskDelay（）的时间，任务希望解除阻塞的时间，而vTaskDelayUntil（）指定的是任务希望的*绝对*时间解除封锁。

从调用vTaskDelay（）开始，vTaskDelay（）将导致任务在指定的滴答数内阻塞。因此，很难单独使用vTaskDelay（）来生成固定的执行频率，因为在调用vTaskDelay（）之后任务解除阻塞与下一次调用vTaskDelay（）的任务之间的时间可能不固定[该任务可能采用其他方法调用之间的代码路径，或者每次执行时都可能被中断或抢占不同的次数]。

vTaskDelay（）指定相对于调用函数的时间的唤醒时间，而vTaskDelayUntil（）指定希望解除阻止的绝对（精确）时间。

应当注意，如果vTaskDelayUntil（）用于指定过去的唤醒时间，它将立即返回（无阻塞）。因此，使用vTaskDelayUntil（）定期执行的任务必须重新计算其所需的唤醒时间，如果由于某种原因（例如，该任务被暂时置于Suspended状态）而导致该周期性执行被暂停，从而导致该任务丢失一个或更多定期处决。可以通过将引用作为pxPreviousWakeTime参数传递的变量与当前滴答计数进行比较来检测。但是，在大多数使用情况下，这不是必需的。

常量portTICK\_PERIOD\_MS可用于根据滴答速率（分辨率为一个滴答周期）计算实时时间。

在通过调用vTaskSuspendAll（）挂起RTOS调度程序后，不得调用此函数。

**参数：**

|  |  |
| --- | --- |
| *pxPreviousWakeTime* | 指向变量的指针，该变量保存上一次取消阻止任务的时间。变量必须在首次使用之前用当前时间进行初始化（请参见下面的示例）。此后，变量将在vTaskDelayUntil（）中自动更新。 |
| *xTimeIncrement* | 周期时间段。该任务将在时间（\* pxPreviousWakeTime + xTimeIncrement）解除阻止。使用相同的xTimeIncrement参数值调用vTaskDelayUntil将使任务以固定的间隔时间执行。 |

**用法示例：**

**// Perform an action every 10 ticks.**

**void vTaskFunction( void \* pvParameters )**

**{**

**TickType\_t xLastWakeTime;**

**const TickType\_t xFrequency = 10;**

**// Initialise the xLastWakeTime variable with the current time.**

**xLastWakeTime = xTaskGetTickCount();**

**for( ;; )**

**{**

**// Wait for the next cycle.**

**vTaskDelayUntil( &xLastWakeTime, xFrequency );**

**// Perform action here.**

**}**

**}**

# uxTaskPriorityGet [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**UBaseType\_t uxTaskPriorityGet( TaskHandle\_t xTask );**

**UBaseType\_t uxTaskPriorityGet（TaskHandle\_t xTask）;**

必须将INCLUDE\_uxTaskPriorityGet定义为1才能使用此功能。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

获得任何任务的优先级。

**参数：**

|  |  |
| --- | --- |
| *任务* | 处理要查询的任务。传递NULL句柄会导致返回调用任务的优先级。 |

**返回值：**

xTask的优先级。

**用法示例：**

**void vAFunction( void )**

**{**

**TaskHandle\_t xHandle;**

**// Create a task, storing the handle.**

**xTaskCreate( vTaskCode, "NAME", STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );**

**// ...**

**// Use the handle to obtain the priority of the created task.**

**// It was created with tskIDLE\_PRIORITY, but may have changed**

**// it itself.**

**if( uxTaskPriorityGet( xHandle ) != tskIDLE\_PRIORITY )**

**{**

**// The task has changed its priority.**

**}**

**// ...**

**// Is our priority higher than the created task?**

**if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )**

**{**

**// Our priority (obtained using NULL handle) is higher.**

**}**

**}**

# vTaskPrioritySet [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**void vTaskPrioritySet( TaskHandle\_t xTask,**

**UBaseType\_t uxNewPriority );**

必须将INCLUDE\_vTaskPrioritySet定义为1才能使用此功能。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

设置任何任务的优先级。

如果设置的优先级高于当前执行的任务，则在函数返回之前将进行上下文切换。

**参数：**

|  |  |
| --- | --- |
| *xTask* | 处理要为其设置优先级的任务。传递NULL句柄会导致设置调用任务的优先级。 |
| *uxNewPriority* | 任务将被设置的优先级。 |

**用法示例：**

**void vAFunction( void )**

**{**

**TaskHandle\_t xHandle;**

**// Create a task, storing the handle.**

**xTaskCreate( vTaskCode, "NAME", STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );**

**// ...**

**// Use the handle to raise the priority of the created task.**

**vTaskPrioritySet( xHandle, tskIDLE\_PRIORITY + 1 );**

**// ...**

**// Use a NULL handle to raise our priority to the same value.**

**vTaskPrioritySet( NULL, tskIDLE\_PRIORITY + 1 );**

**}**

# vTaskSuspend [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**void vTaskSuspend( TaskHandle\_t xTaskToSuspend );**

必须将INCLUDE\_vTaskSuspend定义为1，此功能才可用。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

暂停任何任务。暂停任务后，无论其优先级如何，都将永远不会获得任何微控制器处理时间。

对vTaskSuspend的调用不是累积性的–即，在同一任务上两次调用vTaskSuspend（）仍然只需要对vTaskResume（）进行一次调用即可准备挂起的任务。

**参数：**

|  |  |
| --- | --- |
| *xTaskToSuspend* | 处理被暂停的任务。传递NULL句柄将导致调用任务被挂起。 |

**用法示例：**

**void vAFunction( void )**

**{**

**TaskHandle\_t xHandle;**

**// Create a task, storing the handle.**

**xTaskCreate( vTaskCode, "NAME", STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );**

**// ...**

**// Use the handle to suspend the created task.**

**vTaskSuspend( xHandle );**

**// ...**

**// The created task will not run during this period, unless**

**// another task calls vTaskResume( xHandle ).**

**//...**

**// Suspend ourselves.**

**vTaskSuspend( NULL );**

**// We cannot get here unless another task calls vTaskResume**

**// with our handle as the parameter.**

**}**

# vTaskResume [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**void vTaskResume( TaskHandle\_t xTaskToResume );**

必须将INCLUDE\_vTaskSuspend定义为1，此功能才可用。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

恢复暂停的任务。

通过一次调用vTaskResume（），已被一个或多个vTaskSuspend（）调用暂停的任务将可以再次运行。

**参数：**

|  |  |
| --- | --- |
| *xTaskToResume* | 处理正在准备的任务。 |

**用法示例：**

**void vAFunction( void )**

**{**

**TaskHandle\_t xHandle;**

**// Create a task, storing the handle.**

**xTaskCreate( vTaskCode, "NAME", STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );**

**// ...**

**// Use the handle to suspend the created task.**

**vTaskSuspend( xHandle );**

**// ...**

**// The created task will not run during this period, unless**

**// another task calls vTaskResume( xHandle ).**

**//...**

**// Resume the suspended task ourselves.**

**vTaskResume( xHandle );**

**// The created task will once again get microcontroller processing**

**// time in accordance with its priority within the system.**

**}**

# xTaskResumeFromISR [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**BaseType\_t xTaskResumeFromISR( TaskHandle\_t xTaskToResume );**

要使此功能可用，必须将INCLUDE\_vTaskSuspend和INCLUDE\_xTaskResumeFromISR定义为1。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

恢复可从ISR内部调用的挂起任务的功能。

通过对xTaskResumeFromISR（）的一次调用，已被多个vTaskSuspend（）调用之一暂停的任务将可再次运行。

xTaskResumeFromISR（）通常被认为是危险函数，因为它的操作未锁存。因此，如果中断可能在挂起任务之前到达，因此中断丢失，则绝对不应将其与中断同步使用。使用信号量，或者最好直接使用任务通知，可以避免这种情况的发生。提供了一个[使用直接任务通知的工作示例](https://www.freertos.org/RTOS_Task_Notification_As_Counting_Semaphore.html) 。

**参数：**

|  |  |
| --- | --- |
| *xTaskToResume* | 处理正在准备的任务。 |

**返回值：**

|  |
| --- |
| 如果恢复任务，则为pdTRUE，这将导致上下文切换，否则为pdFALSE。ISR使用它来确定在ISR之后是否可能需要上下文切换。 |

**用法示例：**

**TaskHandle\_t xHandle;**

**void vAFunction( void )**

**{**

**// Create a task, storing the handle.**

**xTaskCreate( vTaskCode, "NAME", STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );**

**// ... Rest of code.**

**}**

**void vTaskCode( void \*pvParameters )**

**{**

**// The task being suspended and resumed.**

**for( ;; )**

**{**

**// ... Perform some function here.**

**// The task suspends itself.**

**vTaskSuspend( NULL );**

**// The task is now suspended, so will not reach here until the ISR resumes it.**

**}**

**}**

**void vAnExampleISR( void )**

**{**

**BaseType\_t xYieldRequired;**

**// Resume the suspended task.**

**xYieldRequired = xTaskResumeFromISR( xHandle );**

**if( xYieldRequired == pdTRUE )**

**{**

**// We should switch context so the ISR returns to a different task.**

**// NOTE: How this is done depends on the port you are using. Check**

**// the documentation and examples for your port.**

**portYIELD\_FROM\_ISR();**

**}**

**}**

# xTaskAbortDelay [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**BaseType\_t xTaskAbortDelay( TaskHandle\_t xTask );**

即使没有发生等待任务处于阻塞状态的事件，并且指定的超时没有到期，也 强制任务离开“[阻塞”状态](https://www.freertos.org/RTOS-task-states.html)并进入“就绪”状态。

必须将INCLUDE\_xTaskAbortDelay定义为1，此功能才可用。有关更多信息，请参见 [RTOS配置](https://www.freertos.org/a00110.html)文档。

**参数：**

|  |  |
| --- | --- |
| *xTask* | 将被强制退出“阻止”状态的任务的句柄。  要获取任务的句柄，请使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务 并使用pxCreatedTask参数，或者使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务 并存储返回的值，或者在调用[xTaskGetHandle（）时](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)使用任务名称。 |

**返回值：**

如果xTask引用的任务未处于“阻止”状态，则返回pdFAIL。否则返回pdPASS。

# 内核控制 [ [API](https://www.freertos.org/a00106.html) ]

## 模组

* [**taskYIELD**](https://www.freertos.org/a00020.html#taskYIELD)
* [**taskENTER\_CRITICAL**](https://www.freertos.org/taskENTER_CRITICAL_taskEXIT_CRITICAL.html)
* [**taskEXIT\_CRITICAL**](https://www.freertos.org/taskENTER_CRITICAL_taskEXIT_CRITICAL.html)
* [**任务ENTER\_CRITICAL\_FROM\_ISR**](https://www.freertos.org/taskENTER_CRITICAL_FROM_ISR_taskEXIT_CRITICAL_FROM_ISR.html)
* [**taskEXIT\_CRITICAL\_FROM\_ISR**](https://www.freertos.org/taskENTER_CRITICAL_FROM_ISR_taskEXIT_CRITICAL_FROM_ISR.html)
* [**taskDISABLE\_INTERRUPTS**](https://www.freertos.org/a00020.html#taskDISABLE_INTERRUPTS)
* [**taskENABLE\_INTERRUPTS**](https://www.freertos.org/a00020.html#taskENABLE_INTERRUPTS)
* [**vTaskStartScheduler**](https://www.freertos.org/a00132.html)
* [**vTaskEndScheduler**](https://www.freertos.org/a00133.html)
* [**vTaskSuspendAll**](https://www.freertos.org/a00134.html)
* [**xTaskResumeAll**](https://www.freertos.org/a00135.html)
* [**vTaskStepTick**](https://www.freertos.org/vTaskStepTick.html)

## 详细说明

### taskYIELD

taks.h

taskYIELD（）用于请求上下文切换到另一个任务。但是，如果没有其他任务比调用taskYIELD（）的任务具有更高或更高的优先级，则RTOS调度程序将简单地选择名为taskYIELD（）的任务以再次运行。

如果[configUSE\_PREEMPTION](https://www.freertos.org/a00110.html#configUSE_PREEMPTION) 设置为1，则RTOS调度程序将始终运行能够运行的最高优先级任务，因此调用taskYIELD（）将永远不会导致切换到更高优先级的任务。

### taskDISABLE\_INTERRUPTS（）

taks.h

如果所采用的端口支持[configMAX\_SYSCALL\_INTERRUPT\_PRIORITY](https://www.freertos.org/a00110.html" \l "kernel_priority)（或configMAX\_API\_CALL\_INTERRUPT\_PRIORITY）恒定，那么taskDISABLE\_INTERRUPTS要么禁止所有中断，或掩蔽（禁用）中断到configMAX\_SYSCALL\_INTERRUPT\_PRIORITY设置。检查正在使用的端口的taskDISABLE\_INTERRUPTS的实现。

如果使用的端口不支持configMAX\_SYSCALL\_INTERRUPT\_PRIORITY常量，则taskDISABLE\_INTERRUPTS（）将全局禁用所有可屏蔽中断。

通常，不会直接调用此宏，而应在其位置使用[taskENTER\_CRITICAL（）和taskEXIT\_CRITICAL（）](https://www.freertos.org/taskENTER_CRITICAL_taskEXIT_CRITICAL.html)。

### taskENABLE\_INTERRUPTS（）

taks.h

用于启用微控制器中断的宏。

通常，不会直接调用此宏，而应在其位置使用[taskENTER\_CRITICAL（）和taskEXIT\_CRITICAL（）](https://www.freertos.org/taskENTER_CRITICAL_taskEXIT_CRITICAL.html)。

# taskENTER\_CRITICAL() taskEXIT\_CRITICAL() [[RTOS Kernel Control](https://www.freertos.org/a00020.html)]

**task. h**

**void taskENTER\_CRITICAL( void );**

**void taskEXIT\_CRITICAL( void );**

通过调用taskENTER\_CRITICAL（）输入关键部分，然后通过调用taskEXIT\_CRITICAL（）退出关键部分。

taskENTER\_CRITICAL（）和taskEXIT\_CRITICAL（）宏提供了一个基本的临界区实现，可以通过简单地全局禁用中断或最高特定中断优先级来禁用中断。有关在 不禁用中断的情况下创建关键部分的信息，请参见[vTaskSuspendAll（）](https://www.freertos.org/a00134.html) RTOS API函数。

如果正在使用的FreeRTOS端口未使用 [configMAX\_SYSCALL\_INTERRUPT\_PRIORITY](https://www.freertos.org/a00110.html" \l "kernel_priority) 内核配置常量（也称为configMAX\_API\_CALL\_INTERRUPT\_PRIORITY），则调用taskENTER\_CRITICAL（）将使中断全局禁用。如果使用的FreeRTOS端口确实使用了configMAX\_SYSCALL\_INTERRUPT\_PRIORITY内核配置常量，则调用taskENTER\_CRITICAL（）将使中断处于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY设置的中断优先级及以下，并禁用所有优先级较高的中断。

抢占式上下文切换仅在中断内部发生，因此在禁用中断时不会发生。因此，除非任务显式尝试阻止或屈服（它不应在关键部分内部执行此操作），否则确保称为taskENTER\_CRITICAL（）的任务将保持在“运行”状态，直到退出关键部分为止。

对taskENTER\_CRITICAL（）和taskEXIT\_CRITICAL（）的调用被设计为嵌套。因此，仅当对taskENTER\_CRITICAL（）的每个先前调用都执行了对taskEXIT\_CRITICAL（）的调用时，才会退出关键部分。

关键部分必须保持很短，否则将对中断响应时间产生不利影响。每次对taskENTER\_CRITICAL（）的调用都必须与对taskEXIT\_CRITICAL（）的调用紧密配对。

不得从关键部分调用FreeRTOS API函数。

不得从中断服务程序（ISR）中[调用taskENTER\_CRITICAL（）](https://www.freertos.org/taskENTER_CRITICAL_FROM_ISR_taskEXIT_CRITICAL_FROM_ISR.html) 和taskEXIT\_CRITICAL（）–有关等效的中断安全信息，请参阅 [taskENTER\_CRITICAL\_FROM\_ISR（）](https://www.freertos.org/taskENTER_CRITICAL_FROM_ISR_taskEXIT_CRITICAL_FROM_ISR.html)和taskEXIT\_CRITICAL\_FROM\_ISR（）。

**参数：**

|  |  |
| --- | --- |
| *没有* |  |

**返回值：**

没有

**用法示例：**

**/\* A function that makes use of a critical section. \*/**

**void vDemoFunction( void )**

**{**

**/\* Enter the critical section. In this example, this function is itself called**

**from within a critical section, so entering this critical section will result**

**in a nesting depth of 2. \*/**

**taskENTER\_CRITICAL();**

**/\* Perform the action that is being protected by the critical section here. \*/**

**/\* Exit the critical section. In this example, this function is itself called**

**from a critical section, so this call to taskEXIT\_CRITICAL() will decrement the**

**nesting count by one, but not result in interrupts becoming enabled. \*/**

**taskEXIT\_CRITICAL();**

**}**

**/\* A task that calls vDemoFunction() from within a critical section. \*/**

**void vTask1( void \* pvParameters )**

**{**

**for( ;; )**

**{**

**/\* Perform some functionality here. \*/**

**/\* Call taskENTER\_CRITICAL() to create a critical section. \*/**

**taskENTER\_CRITICAL();**

**/\* Execute the code that requires the critical section here. \*/**

**/\* Calls to taskENTER\_CRITICAL() can be nested so it is safe to call a**

**function that includes its own calls to taskENTER\_CRITICAL() and**

**taskEXIT\_CRITICAL(). \*/**

**vDemoFunction();**

**/\* The operation that required the critical section is complete so exit the**

**critical section. After this call to taskEXIT\_CRITICAL(), the nesting depth**

**will be zero, so interrupts will have been re-enabled. \*/**

**taskEXIT\_CRITICAL();**

**}**

**}**

# taskENTER\_CRITICAL\_FROM\_ISR() taskEXIT\_CRITICAL\_FROM\_ISR() [[RTOS Kernel Control](https://www.freertos.org/a00020.html)]

**task. h**

**UBaseType\_t taskENTER\_CRITICAL\_FROM\_ISR( void );**

**void taskEXIT\_CRITICAL\_FROM\_ISR( UBaseType\_t uxSavedInterruptStatus );**

可以在中断服务程序（ISR）中使用 的[taskENTER\_CRITICAL（）和taskEXIT\_CRITICAL（）的](https://www.freertos.org/taskENTER_CRITICAL_taskEXIT_CRITICAL.html)版本。

在ISR中，关键部分通过调用taskENTER\_CRITICAL\_FROM\_ISR（）进入，然后通过调用taskEXIT\_CRITICAL\_FROM\_ISR（）退出。

taskENTER\_CRITICAL\_FROM\_ISR（）和taskEXIT\_CRITICAL\_FROM\_ISR（）宏提供了基本的临界区实现，可以通过简单地全局（最高）禁用中断或在特定中断优先级上禁用中断来工作。

如果FreeRTOS的端口被用于支持中断，并在下面被中断优先级组嵌套然后调用taskENTER\_CRITICAL\_FROM\_ISR（）将禁止中断[configMAX\_SYSCALL\_INTERRUPT\_PRIORITY](https://www.freertos.org/a00110.html" \l "kernel_priority) （或configMAX\_API\_CALL\_INTERRUPT\_PRIORITY）内核配置不变，并启用所有其他中断优先级。如果使用的FreeRTOS端口不支持中断嵌套，则taskENTER\_CRITICAL\_FROM\_ISR（）和taskEXIT\_CRITICAL\_FROM\_ISR（）无效。

对taskENTER\_CRITICAL\_FROM\_ISR（）和taskEXIT\_CRITICAL\_FROM\_ISR（）的调用被设计为嵌套，但是如何使用宏的语义与taskENTER\_CRITICAL（）和taskEXIT\_CRITICAL（）等效项不同。

关键部分必须保持很短，否则它们将对优先级较高的中断的响应时间产生不利影响，否则这些中断将嵌套。每次对taskENTER\_CRITICAL\_FROM\_ISR（）的调用都必须与对taskEXIT\_CRITICAL\_FROM\_ISR（）的调用紧密配对。

不得从关键部分调用FreeRTOS API函数。

**参数：**

|  |  |
| --- | --- |
| *uxSavedInterruptStatus* | taskEXIT\_CRITICAL\_FROM\_ISR（）将uxSavedInterruptStatus作为其唯一参数。用作uxSavedInterruptStatus参数的值必须是从对taskENTER\_CRITICAL\_FROM\_ISR（）的匹配调用返回的值。  taskENTER\_CRITICAL\_FROM\_ISR（）不接受任何参数。 |

**返回值：**

taskENTER\_CRITICAL\_FROM\_ISR（）返回中断屏蔽状态，与调用宏之前一样。在对taskEXIT\_CRITICAL\_FROM\_ISR（）的匹配调用中，必须将taskENTER\_CRITICAL\_FROM\_ISR（）返回的值用作uxSavedInterruptStatus参数。

taskEXIT\_CRITICAL\_FROM\_ISR（）不返回值。

**用法示例：**

**/\* A function called from an ISR. \*/**

**void vDemoFunction( void )**

**{**

**UBaseType\_t uxSavedInterruptStatus;**

**/\* Enter the critical section. In this example, this function is itself called from**

**within a critical section, so entering this critical section will result in a nesting**

**depth of 2. Save the value returned by taskENTER\_CRITICAL\_FROM\_ISR() into a local**

**stack variable so it can be passed into taskEXIT\_CRITICAL\_FROM\_ISR(). \*/**

**uxSavedInterruptStatus = taskENTER\_CRITICAL\_FROM\_ISR();**

**/\* Perform the action that is being protected by the critical section here. \*/**

**/\* Exit the critical section. In this example, this function is itself called from a**

**critical section, so interrupts will have already been disabled before a value was**

**stored in uxSavedInterruptStatus, and therefore passing uxSavedInterruptStatus into**

**taskEXIT\_CRITICAL\_FROM\_ISR() will not result in interrupts being re-enabled. \*/**

**taskEXIT\_CRITICAL\_FROM\_ISR( uxSavedInterruptStatus );**

**}**

**/\* A task that calls vDemoFunction() from within an interrupt service routine. \*/**

**void vDemoISR( void )**

**{**

**UBaseType\_t uxSavedInterruptStatus;**

**/\* Call taskENTER\_CRITICAL\_FROM\_ISR() to create a critical section, saving the**

**returned value into a local stack variable. \*/**

**uxSavedInterruptStatus = taskENTER\_CRITICAL\_FROM\_ISR();**

**/\* Execute the code that requires the critical section here. \*/**

**/\* Calls to taskENTER\_CRITICAL\_FROM\_ISR() can be nested so it is safe to call a**

**function that includes its own calls to taskENTER\_CRITICAL\_FROM\_ISR() and**

**taskEXIT\_CRITICAL\_FROM\_ISR(). \*/**

**vDemoFunction();**

**/\* The operation that required the critical section is complete so exit the**

**critical section. Assuming interrupts were enabled on entry to this ISR, the value**

**saved in uxSavedInterruptStatus will result in interrupts being re-enabled.\*/**

**taskEXIT\_CRITICAL\_FROM\_ISR( uxSavedInterruptStatus );**

**}**

# vTaskStartScheduler [[RTOS Kernel Control](https://www.freertos.org/a00020.html)]

task. h

**void vTaskStartScheduler( void );**

启动RTOS调度程序。调用RTOS内核后，可以控制执行哪些任务以及何时执行。

在[空闲任务](https://www.freertos.org/RTOS-idle-task.html)和可选的 [定时器守护任务](https://www.freertos.org/RTOS-software-timer-service-daemon-task.html)的RTOS调度程序启动时自动创建。

仅当没有足够的[RTOS堆](https://www.freertos.org/a00111.html)来创建空闲或计时器守护程序任务时，vTaskStartScheduler（）才会返回。

所有RTOS演示应用程序项目都包含使用vTaskStartScheduler（）的示例，通常在main.c的main（）函数中。

**用法示例：**

**void vAFunction( void )**

**{**

**// Tasks can be created before or after starting the RTOS**

**scheduler**

[**xTaskCreate**](https://www.freertos.org/a00125.html)**( vTaskCode,**

**"NAME",**

**STACK\_SIZE,**

**NULL,**

**tskIDLE\_PRIORITY,**

**NULL );**

**// Start the real time scheduler.**

**vTaskStartScheduler();**

**// Will not get here unless there is insufficient RAM.**

**}**

# vTaskEndScheduler [[RTOS Kernel Control](https://www.freertos.org/a00020.html)]

task. h

**void vTaskEndScheduler( void );**

注意：这仅适用于x86实模式PC端口。

停止RTOS内核滴答。所有创建的任务将被自动删除，并且多任务处理（抢占式或协作式）将停止。然后，从调用vTaskStartScheduler（）的点继续执行，就好像vTaskStartScheduler（）刚刚返回一样。

请参阅演示应用程序文件主文件。有关使用vTaskEndScheduler（）的示例，请参见demo / PC目录中的c。

vTaskEndScheduler（）要求在可移植层中定义退出函数（有关PC端口，请参见端口c中的vPortEndScheduler（））。这将执行特定于硬件的操作，例如停止RTOS内核滴答。

vTaskEndScheduler（）将导致释放RTOS内核分配的所有资源-但不会释放应用程序任务分配的资源。

**用法示例：**

**void vTaskCode( void \* pvParameters )**

**{**

**for( ;; )**

**{**

**// Task code goes here.**

**// At some point we want to end the real time kernel processing**

**// so call ...**

**vTaskEndScheduler ();**

**}**

**}**

**void vAFunction( void )**

**{**

**// Create at least one task before starting the RTOS kernel.**

**xTaskCreate( vTaskCode, "NAME", STACK\_SIZE, NULL, tskIDLE\_PRIORITY, NULL );**

**// Start the real time kernel with preemption.**

**vTaskStartScheduler();**

**// Will only get here when the vTaskCode () task has called**

**// vTaskEndScheduler (). When we get here we are back to single task**

**// execution.**

**}**

# vTaskSuspendAll [[RTOS Kernel Control](https://www.freertos.org/a00020.html)]

task. h

**void vTaskSuspendAll( void );**

挂起调度程序。挂起调度程序可以防止发生上下文切换，但可以使中断保持启用状态。如果在挂起调度程序时中断请求上下文切换，则该请求将保持挂起状态，并且仅在重新启动调度程序（未挂起）时才执行该请求。

调用[xTaskResumeAll（）](https://www.freertos.org/a00135.html)过渡调度出挂起状态下一次调用vTaskSuspendAll（）。

可以嵌套对vTaskSuspendAll（）的调用。在调度程序将退出“挂起”状态并重新进入“活动”状态之前，必须对xTaskResumeAll（）进行与先前对vTaskSuspendAll（）相同的调用。

xTaskResumeAll（）只能从正在执行的任务中调用，因此，在调度程序处于初始化状态（在启动调度程序之前）时，不得调用xTaskResumeAll（）。

调度程序挂起时，不得调用其他FreeRTOS API函数。

挂起调度程序时，**不得**调用可能导致上下文切换的API函数（例如vTaskDelayUntil（），xQueueSend（）等）。

**用法示例：**

**/\* A function that suspends then resumes the scheduler. \*/**

**void vDemoFunction( void )**

**{**

**/\* This function suspends the scheduler. When it is called from vTask1 the**

**scheduler is already suspended, so this call creates a nesting depth of 2. \*/**

**vTaskSuspendAll();**

**/\* Perform an action here. \*/**

**/\* As calls to vTaskSuspendAll() are nested, resuming the scheduler here will**

**not cause the scheduler to re-enter the active state. \*/**

**xTaskResumeAll();**

**}**

**void vTask1( void \* pvParameters )**

**{**

**for( ;; )**

**{**

**/\* Perform some actions here. \*/**

**/\* At some point the task wants to perform an operation during which it does**

**not want to get swapped out, or it wants to access data which is also**

**accessed from another task (but not from an interrupt). It cannot use**

**taskENTER\_CRITICAL()/taskEXIT\_CRITICAL() as the length of the operation may**

**cause interrupts to be missed. \*/**

**/\* Prevent the scheduler from performing a context switch. \*/**

**vTaskSuspendAll();**

**/\* Perform the operation here. There is no need to use critical sections as**

**the task has all the processing time other than that utilized by interrupt**

**service routines.\*/**

**/\* Calls to vTaskSuspendAll() can be nested so it is safe to call a (non API)**

**function which also contains calls to vTaskSuspendAll(). API functions**

**should not be called while the scheduler is suspended. \*/**

**vDemoFunction();**

**/\* The operation is complete. Set the scheduler back into the Active**

**state. \*/**

**if( xTaskResumeAll() == pdTRUE )**

**{**

**/\* A context switch occurred within xTaskResumeAll(). \*/**

**}**

**else**

**{**

**/\* A context switch did not occur within xTaskResumeAll(). \*/**

**}**

**}**

**}**

# xTaskResumeAll [[RTOS Kernel Control](https://www.freertos.org/a00020.html)]

task. h

**BaseType\_t xTaskResumeAll( void );**

使用对vTaskSuspendAll（）的调用在挂起调度程序后继续。

xTaskResumeAll（）仅恢复调度程序。它不会取消暂停先前通过调用vTaskSuspend（）而暂停的任务。

**返回值：**

如果恢复调度程序导致上下文切换，则返回pdTRUE，否则返回pdFALSE。

**用法示例：**

**void vTask1( void \* pvParameters )**

**{**

**for( ;; )**

**{**

**/\* Task code goes here. \*/**

**/\* ... \*/**

**/\* At some point the task wants to perform a long operation**

**during which it does not want to get swapped out. It cannot**

**use taskENTER\_CRITICAL ()/taskEXIT\_CRITICAL () as the length**

**of the operation may cause interrupts to be missed -**

**including the ticks.**

**Prevent the RTOS kernel swapping out the task. \*/**

**vTaskSuspendAll();**

**/\* Perform the operation here. There is no need to use critical**

**sections as we have all the microcontroller processing time.**

**During this time interrupts will still operate and the real**

**time RTOS kernel tick count will be maintained. \*/**

**/\* ... \*/**

**/\* The operation is complete. Restart the RTOS kernel. We want to force**

**a context switch - but there is no point if resuming the scheduler**

**caused a context switch already. \*/**

**if( !xTaskResumeAll () )**

**{**

**taskYIELD ();**

**}**

**}**

**}**

# vTaskStepTick [[RTOS Kernel Control](https://www.freertos.org/a00020.html)]

task.h

**void vTaskStepTick( TickType\_t xTicksToJump );**

如果将RTOS配置为使用[无滴答空闲功能，](https://www.freertos.org/low-power-tickless-rtos.html) 则只要空闲任务是唯一能够执行的任务，滴答中断就会停止，微控制器进入低功耗状态。退出低功耗状态后，必须校正滴答计数值，以说明停止时经过的时间。

如果FreeRTOS端口包括默认的[portSUPPRESS\_TICKS\_AND\_SLEEP（）](https://www.freertos.org/low-power-tickless-rtos.html) 实现，则内部将使用vTaskStepTick（）来确保维护正确的滴答计数值。vTaskStepTick（）是公共API函数，它允许覆盖默认的portSUPPRESS\_TICKS\_AND\_SLEEP（）实现，如果使用的端口不提供默认值，则可以提供portSUPPRESS\_TICKS\_AND\_SLEEP（）。

必须将configUSE\_TICKLESS\_IDLE配置常量设置为1，vTaskStepTick（）才可用。

**参数：**

|  |  |
| --- | --- |
| *xTicksToJump* | 自滴答中断停止以来已经过去的RTOS滴答次数。为了正确操作，该参数必须小于或等于portSUPPRESS\_TICKS\_AND\_SLEEP（）参数。 |

**返回值：**

没有。

**用法示例：**

该示例显示了对多个函数的调用。只有vTaskStepTick（）是FreeRTOS API的一部分。其他功能特定于正在使用的硬件上可用的时钟和省电模式，因此必须由应用程序编写器提供。

**/\* First define the portSUPPRESS\_TICKS\_AND\_SLEEP(). The parameter is the time,**

**in ticks, until the kernel next needs to execute. \*/**

**#define portSUPPRESS\_TICKS\_AND\_SLEEP( xIdleTime ) vApplicationSleep( xIdleTime )**

**/\* Define the function that is called by portSUPPRESS\_TICKS\_AND\_SLEEP(). \*/**

**void vApplicationSleep( TickType\_t xExpectedIdleTime )**

**{**

**unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;**

**/\* Read the current time from a time source that will remain operational**

**while the microcontroller is in a low power state. \*/**

**ulLowPowerTimeBeforeSleep = ulGetExternalTime();**

**/\* Stop the timer that is generating the tick interrupt. \*/**

**prvStopTickInterruptTimer();**

**/\* Configure an interrupt to bring the microcontroller out of its low power**

**state at the time the kernel next needs to execute. The interrupt must be**

**generated from a source that is remains operational when the microcontroller**

**is in a low power state. \*/**

**vSetWakeTimeInterrupt( xExpectedIdleTime );**

**/\* Enter the low power state. \*/**

**prvSleep();**

**/\* Determine how long the microcontroller was actually in a low power state**

**for, which will be less than xExpectedIdleTime if the microcontroller was**

**brought out of low power mode by an interrupt other than that configured by**

**the vSetWakeTimeInterrupt() call. Note that the scheduler is suspended**

**before portSUPPRESS\_TICKS\_AND\_SLEEP() is called, and resumed when**

**portSUPPRESS\_TICKS\_AND\_SLEEP() returns. Therefore no other tasks will**

**execute until this function completes. \*/**

**ulLowPowerTimeAfterSleep = ulGetExternalTime();**

**/\* Correct the kernels tick count to account for the time the microcontroller**

**spent in its low power state. \*/**

**vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );**

**/\* Restart the timer that is generating the tick interrupt. \*/**

**prvStartTickInterruptTimer();**

**}**

# xTaskCatchUpTicks [[RTOS Kernel Control](https://www.freertos.org/a00020.html)]

task.h

**BaseType\_t xTaskCatchUpTicks( TickType\_t xTicksToCatchUp );**

在应用程序代码将中断长时间禁用后，更正滴答计数值。此函数与[vTaskStepTick（）](https://www.freertos.org/vTaskStepTick.html)类似，但是，与[vTaskStepTick（）](https://www.freertos.org/vTaskStepTick.html)不同 ，此函数可以将滴答计数向前移动超过应从阻止状态中删除任务的时间。这意味着xTaskCatchUpTicks（）可以从阻止状态中删除任务。

**参数：**

|  |  |
| --- | --- |
| *xTicksToCatchUp* | 由于禁用了中断而错过的滴答中断的数量。它的值不会自动计算，因此必须由应用程序编写器计算。 |

**返回值：**

如果向前移动滴答计数导致任务离开阻塞状态并执行上下文切换，则为pdTRUE。否则为pdFALSE。

**用法示例：**

**void vExampleFunction( void )**

**{**

**unsigned long ulTimeBefore, ulTimeAfter;**

**/\* Read the current time before arbitrary processing takes place. \*/**

**ulTimeBefore = ulGetExternalTime();**

**/\* Stop the timer that is generating the tick interrupt. \*/**

**prvStopTickInterruptTimer();**

**/\* Perform some arbitrary processing. \*/**

**arbitrary\_processing();**

**/\* Read the current time for computing elapsed time since ticks**

**were disabled. \*/**

**ulTimeAfter = ulGetExternalTime();**

**if ( xTaskCatchUpTicks( ulTimeAfter - ulTimeBefore ) == pdTRUE )**

**{**

**/\* Moving the tick count forward resulted in a context switch. \*/**

**}**

**/\* Restart the timer that is generating the tick interrupt. \*/**

**prvStartTickInterruptTimer();**

**}**

# RTOS Task Notifications [[API](https://www.freertos.org/a00106.html)]

## [RTOS task notification](https://www.freertos.org/RTOS-task-notifications.html) API functions:

* [**xTaskNotifyGive() / xTaskNotifyGiveIndexed()**](https://www.freertos.org/xTaskNotifyGive.html)
* [**vTaskNotifyGiveFromISR() / vTaskNotifyGiveIndexedFromISR()**](https://www.freertos.org/vTaskNotifyGiveFromISR.html)
* [**ulTaskNotifyTake() / ulTaskNotifyTakeIndexed()**](https://www.freertos.org/ulTaskNotifyTake.html)
* [**xTaskNotify() / xTaskNotifyIndexed()**](https://www.freertos.org/xTaskNotify.html)
* [**xTaskNotifyAndQuery() / xTaskNotifyAndQueryIndexed()**](https://www.freertos.org/xTaskNotifyAndQuery.html)
* [**xTaskNotifyAndQueryFromISR / xTaskNotifyAndQueryFromISRIndexed()/**](https://www.freertos.org/xTaskNotifyAndQueryFromISR.html)
* [**xTaskNotifyFromISR() / xTaskNotifyFromISRIndexed()**](https://www.freertos.org/xTaskNotifyFromISR.html)
* [**xTaskNotifyWait() / xTaskNotifyWaitIndexed()**](https://www.freertos.org/xTaskNotifyWait.html)
* [**xTaskNotifyStateClear() / xTaskNotifyStateClearIndexed()**](https://www.freertos.org/xTaskNotifyStateClear.html)
* [**ulTasknotifyValueClear() / ulTasknotifyValueClearIndexed()**](https://www.freertos.org/ulTasknotifyValueClear.html)

# xTaskNotifyGive / xTaskNotifyGiveIndexed [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**BaseType\_t xTaskNotifyGive( TaskHandle\_t xTaskToNotify );**

**BaseType\_t xTaskNotifyGiveIndexed( TaskHandle\_t xTaskToNotify,**

**UBaseType\_t uxIndexToNotify );**

每个任务都有一个“任务通知”（或仅是“通知”）数组，每个通知都有一个状态和一个32位值。甲[直接到任务通知](https://www.freertos.org/RTOS-task-notifications.html)是一个事件直接发送到可以解锁接收任务，和任选地在许多不同的方式更新接收任务的通知值中的一个任务。例如，通知可能会覆盖接收任务的通知值之一，或者仅在接收任务的通知值之一中设置一个或多个位。

xTaskNotifyGive（）是一个宏，旨在在将任务通知[用作轻量级且更快的二进制或计数信号量](https://www.freertos.org/RTOS-task-notifications.html" \l "uses)替代方法时使用。FreeRTOS信号量是使用xSemaphoreGive（）API函数给出的，xTaskNotifyGive（）是等效的，它使用接收RTOS任务的通知值之一代替信号量。

xTaskNotifyGive（）和xTaskNotifyGiveIndexed（）是等效的宏–唯一的区别是xTaskNotifyGiveIndexed（）可以对数组内的任何任务通知进行操作，而xTaskNotifyGive（）始终对数组索引0处的任务通知进行操作。

当任务通知值用作二进制或计数信号量等效项时，被通知的任务应使用[ulTask​​NotifyTake（）](https://www.freertos.org/ulTaskNotifyTake.html) API函数而不是[xTaskNotifyWait（）](https://www.freertos.org/xTaskNotifyWait.html) API函数来等待通知。

**注意：**阵列中的每个通知都是独立运行的–一项任务一次只能阻塞一个阵列中的一个通知，而不会被发送给任何其他阵列索引的通知所阻塞。

不得从中断服务程序中调用xTaskNotifyGive（）。请改用 [vTaskNotifyGiveFromISR（）](https://www.freertos.org/vTaskNotifyGiveFromISR.html)。

[为了](https://www.freertos.org/a00110.html#configUSE_TASK_NOTIFICATIONS) 使这些宏可用，必须在FreeRTOSConfig.h[中将configUSE\_TASK\_NOTIFICATIONS](https://www.freertos.org/a00110.html" \l "configUSE_TASK_NOTIFICATIONS)设置为1（或保持未定义状态）。常量 [configTASK\_NOTIFICATION\_ARRAY\_ENTRIES](https://www.freertos.org/a00110.html" \l "configTASK_NOTIFICATION_ARRAY_ENTRIES)设置每个任务的任务通知数组中的索引数。

**向后兼容性信息：**  
在FreeRTOS V10.4.0之前，每个任务都具有一个“通知值”，并且所有任务通知API函数都在该值上进行操作。用一组通知值代替单个通知值需要一组新的API函数，这些函数可以处理数组中的特定通知。xTaskNotifyGive（）是原始API函数，并且始终通过对数组索引0处的通知值进行操作来保持向后兼容。调用xTaskNotifyGive（）等效于将uxIndexToNotify参数设置为0的xTaskNotifyGiveIndexed（）。

**参数：**

|  |  |
| --- | --- |
| *xTaskToNotify* | 通知RTOS任务的句柄，并使其通知值增加。  要获取任务的句柄，请使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务 并使用pxCreatedTask参数，或者使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务 并存储返回的值，或者在调用[xTaskGetHandle（）时](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)使用任务名称。  [xTaskGetCurrentTaskHandle（）](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle) API函数 返回当前执行的RTOS任务的[句柄](https://www.freertos.org/a00021.html" \l "xTaskGetCurrentTaskHandle)。 |
| *uxIndexToNotify* | 目标任务的通知值数组中的索引，通知将发送到该索引。  uxIndexToNotify必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。xTaskNotifyGive（）没有此参数，并且始终将通知发送到索引0。 |

**返回值：**

xTaskNotifyGiveIndexed（）是一个宏，它在eAction参数设置为eIncrement的情况下调用[xTaskNotifyIndexed（）](https://www.freertos.org/xTaskNotify.html)，因此所有调用均返回pdPASS。

**用法示例：**

[更多示例参见[RTOS任务通知主页面](https://www.freertos.org/RTOS-task-notifications.html)]

**/\* Prototypes of the two tasks created by main(). \*/**

**static void prvTask1( void \*pvParameters );**

**static void prvTask2( void \*pvParameters );**

**/\* Handles for the tasks create by main(). \*/**

**static TaskHandle\_t xTask1 = NULL, xTask2 = NULL;**

**/\* Create two tasks that send notifications back and forth to each other,**

**then start the RTOS scheduler. \*/**

**void main( void )**

**{**

**xTaskCreate( prvTask1, “Task1”, 200, NULL, tskIDLE\_PRIORITY, &xTask1 );**

**xTaskCreate( prvTask2, “Task2”, 200, NULL, tskIDLE\_PRIORITY, &xTask2 );**

**vTaskStartScheduler();**

**}**

**/\*———————————————————–\*/**

**/\* prvTask1() uses the ‘indexed’ version of the API. \*/**

**static void prvTask1( void \*pvParameters )**

**{**

**for( ;; )**

**{**

**/\* Send notification to prvTask2(), bringing it out of the**

**Blocked state. \*/**

**xTaskNotifyGiveIndexed( xTask2, 0 );**

**/\* Block to wait for prvTask2() to notify this task. \*/**

**ulTaskNotifyTakeIndexed( 0, pdTRUE, portMAX\_DELAY );**

**}**

**}**

**/\*———————————————————–\*/**

**/\* prvTask2() uses the original version of the API (without the**

**‘Indexed’). \*/**

**static void prvTask2( void \*pvParameters )**

**{**

**for( ;; )**

**{**

**/\* Block to wait for prvTask1() to notify this task. \*/**

**ulTaskNotifyTake( pdTRUE, portMAX\_DELAY );**

**/\* Send a notification to prvTask1(), bringing it out of the**

**Blocked state. \*/**

**xTaskNotifyGive( xTask1 );**

**}**

**}**

# vTaskNotifyGiveFromISR / vTaskNotifyGiveIndexedFromISR [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**void vTaskNotifyGiveFromISR( TaskHandle\_t xTaskToNotify,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

**void vTaskNotifyGiveIndexedFromISR( TaskHandle\_t xTaskHandle,**

**UBaseType\_t uxIndexToNotify,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

可以从中断服务程序（ISR）使用的xTaskNotifyGive（）和xTaskNotifyGiveIndexed（）版本。请参阅[xTaskNotifyGive（）](https://www.freertos.org/xTaskNotifyGive.html) API函数的文档页面，以[了解](https://www.freertos.org/xTaskNotifyGive.html)其操作和必要的配置参数以及向后兼容性信息。

**参数：**

|  |  |
| --- | --- |
| *xTaskToNotify* | 通知RTOS任务的句柄，并使其通知值增加。  要获取任务的句柄，请使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务 并使用pxCreatedTask参数，或者使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务 并存储返回的值，或者在调用[xTaskGetHandle（）时](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)使用任务名称。  [xTaskGetCurrentTaskHandle（）](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle) API函数 返回当前执行的RTOS任务的[句柄](https://www.freertos.org/a00021.html" \l "xTaskGetCurrentTaskHandle)。 |
| *uxIndexToNotify* | 目标任务的通知值数组中的索引，通知将发送到该索引。  uxIndexToNotify必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。  xTaskNotifyGiveFromISR（）没有此参数，并且始终将通知发送到索引0。 |
| *pxHigherPriorityTaskWoken* | \* pxHigherPriorityTaskWoken必须初始化为0。  如果发送通知导致任务取消阻止，并且取消阻止的任务的优先级高于当前运行的任务，则vTaskNotifyGiveFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。  如果vTaskNotifyGiveFromISR（）将此值设置为pdTRUE，则应在退出中断之前请求上下文切换。请参见下面的示例。  pxHigherPriorityTaskWoken是一个可选参数，可以设置为NULL。 |

**用法示例：**

[更多示例参见[RTOS任务通知主页面](https://www.freertos.org/RTOS-task-notifications.html)]

**/\* This is an example of a transmit function in a generic peripheral driver. An**

**RTOS task calls the transmit function, then waits in the Blocked state (so not**

**using an CPU time) until it is notified that the transmission is complete. The**

**transmission is performed by a DMA, and the DMA end interrupt is used to notify**

**the task. \*/**

**static TaskHandle\_t xTaskToNotify = NULL;**

**/\* The peripheral driver’s transmit function. \*/**

**void StartTransmission( uint8\_t \*pcData, size\_t xDataLength )**

**{**

**/\* At this point xTaskToNotify should be NULL as no transmission is in**

**progress. A mutex can be used to guard access to the peripheral if**

**necessary. \*/**

**configASSERT( xTaskToNotify == NULL );**

**/\* Store the handle of the calling task. \*/**

**xTaskToNotify =** [**xTaskGetCurrentTaskHandle()**](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle)**;**

**/\* Start the transmission – an interrupt is generated when the transmission**

**is complete. \*/**

**vStartTransmit( pcData, xDatalength );**

**}**

**/\*———————————————————–\*/**

**/\* The transmit end interrupt. \*/**

**void vTransmitEndISR( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**/\* At this point xTaskToNotify should not be NULL as a transmission was**

**in progress. \*/**

**configASSERT( xTaskToNotify != NULL );**

**/\* Notify the task that the transmission is complete. \*/**

**vTaskNotifyGiveIndexedFromISR( xTaskToNotify, 0, &xHigherPriorityTaskWoken );**

**/\* There are no transmissions in progress, so no tasks to notify. \*/**

**xTaskToNotify = NULL;**

**/\* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch**

**should be performed to ensure the interrupt returns directly to the highest**

**priority task. The macro used for this purpose is dependent on the port in**

**use and may be called portEND\_SWITCHING\_ISR(). \*/**

**portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

**/\*———————————————————–\*/**

**/\* The task that initiates the transmission, then enters the Blocked state (so**

**not consuming any CPU time) to wait for it to complete. \*/**

**void vAFunctionCalledFromATask( uint8\_t ucDataToTransmit, size\_t xDataLength )**

**{**

**uint32\_t ulNotificationValue;**

**const TickType\_t xMaxBlockTime = pdMS\_TO\_TICKS( 200 );**

**/\* Start the transmission by calling the function shown above. \*/**

**StartTransmission( ucDataToTransmit, xDataLength );**

**/\* Wait for the transmission to complete. \*/**

**ulNotificationValue =** [**ulTaskNotifyTakeIndexed**](https://www.freertos.org/ulTaskNotifyTake.html)**( 0, pdFALSE, xMaxBlockTime );**

**if( ulNotificationValue == 1 )**

**{**

**/\* The transmission ended as expected. \*/**

**}**

**else**

**{**

**/\* The call to ulTaskNotifyTake() timed out. \*/**

**}**

**}**

# ulTaskNotifyTake / ulTaskNotifyTakeIndexed [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**uint32\_t ulTaskNotifyTake( BaseType\_t xClearCountOnExit,**

**TickType\_t xTicksToWait );**

**uint32\_t ulTaskNotifyTakeIndexed( UBaseType\_t uxIndexToWaitOn,**

**BaseType\_t xClearCountOnExit,**

**TickType\_t xTicksToWait );**

[直接到任务通知](https://www.freertos.org/RTOS-task-notifications.html)是一个事件直接发送到可以解锁接收任务，和任选地在许多不同的方式更新接收任务的通知值中的一个任务。例如，通知可能会覆盖接收任务的通知值之一，或者只是在接收任务的通知值之一中设置一个或多个位。

ulTask​​NotifyTake（）是一个宏，旨在用于将任务通知 [用作更快，更轻量的二进制或计数信号量](https://www.freertos.org/RTOS-task-notifications.html#uses)替代方案时使用。FreeRTOS信号量是使用xSemaphoreTake（）API函数获取的，ulTask​​NotifyTake（）是等效的，它使用通知值代替信号量。

ulTask​​NotifyTake（）和ulTask​​NotifyTakeIndexed（）是等效的宏–唯一的区别是ulTask​​NotifyTakeIndexed（）可以对数组内的任何任务通知进行操作，而ulTask​​NotifyTake（）始终对数组索引0处的任务通知进行操作。

当任务使用通知值作为二进制值或计算信号量时，其他任务和中断应使用[xTaskNotifyGive（）](https://www.freertos.org/xTaskNotifyGive.html)宏或 [xTaskNotify（）](https://www.freertos.org/xTaskNotify.html)函数（其函数的eAction参数设置为eIncrement [）](https://www.freertos.org/xTaskNotifyGive.html)向其发送通知。 ）。

ulTask​​NotifyTake（）可以在退出时将任务的通知值清除为零（在这种情况下，通知值的作用类似于二进制信号量），或者在退出时减小任务的通知值，在这种情况下，通知值的作用更像是计数信号量。

RTOS任务可以使用ulTask​​NotifyTake（）来[可选地]阻塞以等待任务的通知值不为零。处于“阻塞”状态时，该任务不会消耗任何CPU时间。

**注意：**阵列中的每个通知都是独立运行的–一项任务一次只能阻塞一个阵列中的一个通知，而不会被发送给任何其他阵列索引的通知所阻塞。

就像[xTaskNotifyWait（）](https://www.freertos.org/xTaskNotifyWait.html)在通知待处理时返回，而ulTask​​NotifyTake（）在任务的通知值不为零时返回，则在返回之前减小任务的通知值。

[为了](https://www.freertos.org/a00110.html#configUSE_TASK_NOTIFICATIONS) 使这些宏可用，必须在FreeRTOSConfig.h[中将configUSE\_TASK\_NOTIFICATIONS](https://www.freertos.org/a00110.html" \l "configUSE_TASK_NOTIFICATIONS)设置为1（或保持未定义状态）。常量 [configTASK\_NOTIFICATION\_ARRAY\_ENTRIES](https://www.freertos.org/a00110.html" \l "configTASK_NOTIFICATION_ARRAY_ENTRIES)设置每个任务的任务通知数组中的索引数。

**向后兼容性信息：**  
在FreeRTOS V10.4.0之前，每个任务都具有一个“通知值”，并且所有任务通知API函数都在该值上进行操作。用一组通知值代替单个通知值需要一组新的API函数，这些函数可以处理数组中的特定通知。ulTask​​NotifyTake（）是原始API函数，并且始终通过对数组索引0处的通知值进行操作来保持向后兼容。调用ulTask​​NotifyTake（）等效于将uxIndexToWaitOn参数设置为0的ulTask​​NotifyTakeIndexed（）。

**参数：**

|  |  |
| --- | --- |
| *uxIndexToWaitOn* | 调用任务将等待通知为非零的调用任务的通知值数组中的索引。  uxIndexToWaitOn必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。  xTaskNotifyTake（）没有此参数，并且始终等待索引0上的通知。 |
| *xClearCountOnExit* | 如果收到了RTOS任务通知，并且xClearCountOnExit设置为pdFALSE，则在ulTask​​NotifyTake（）退出之前，RTOS任务的通知值将减小。这等效于成功调用xSemaphoreTake（）会减少计数信号量的值。  如果收到RTOS任务通知并将xClearCountOnExit设置为pdTRUE，则在ulTask​​NotifyTake（）退出之前，RTOS任务的通知值将重置为0。这等效于成功调用xSemaphoreTake（）后，二进制信号量的值保留为零（或为空，或“不可用”）。 |
| *xTicksToWait* | 如果在调用ulTask​​NotifyTake（）时通知尚未挂起，则在“阻塞”状态下等待接收通知的最长时间。  处于阻塞状态时，RTOS任务不会消耗任何CPU时间。  时间以RTOS滴答周期指定。pdMS\_TO\_TICKS（）宏可用于将以毫秒为单位的时间转换为以刻度为单位的时间。 |

**返回值：**

任务的通知值的值在递减或清除之前（请参阅xClearCountOnExit的说明）。

**用法示例：**

[更多示例参见[RTOS任务通知主页面](https://www.freertos.org/RTOS-task-notifications.html)]

**/\* An interrupt handler. The interrupt handler does not perform any processing,**

**instead it unblocks a high priority task in which the event that generated the**

**interrupt is processed. If the priority of the task is high enough then the**

**interrupt will return directly to the task (so it will interrupt one task but**

**return to a different task), so the processing will occur contiguously in time –**

**just as if all the processing had been done in the interrupt handler itself. \*/**

**void vANInterruptHandler( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken;**

**/\* Clear the interrupt. \*/**

**prvClearInterruptSource();**

**/\* xHigherPriorityTaskWoken must be initialised to pdFALSE. If calling**

**vTaskNotifyGiveFromISR() unblocks the handling task, and the priority of**

**the handling task is higher than the priority of the currently running task,**

**then xHigherPriorityTaskWoken will automatically get set to pdTRUE. \*/**

**xHigherPriorityTaskWoken = pdFALSE;**

**/\* Unblock the handling task so the task can perform any processing necessitated**

**by the interrupt. xHandlingTask is the task’s handle, which was obtained**

[**when the task was created**](https://www.freertos.org/a00125.html)**. \*/**

**vTaskNotifyGiveIndexedFromISR( xHandlingTask, 0, &xHigherPriorityTaskWoken );**

**/\* Force a context switch if xHigherPriorityTaskWoken is now set to pdTRUE.**

**The macro used to do this is dependent on the port and may be called**

**portEND\_SWITCHING\_ISR. \*/**

**portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

**/\*———————————————————–\*/**

**/\* A task that blocks waiting to be notified that the peripheral needs servicing,**

**processing all the events pending in the peripheral each time it is notified to**

**do so. \*/**

**void vHandlingTask( void \*pvParameters )**

**{**

**BaseType\_t xEvent;**

**for( ;; )**

**{**

**/\* Block indefinitely (without a timeout, so no need to check the function’s**

**return value) to wait for a notification. Here the RTOS task notification**

**is being used as a binary semaphore, so the notification value is cleared**

**to zero on exit. NOTE! Real applications should not block indefinitely,**

**but instead time out occasionally in order to handle error conditions**

**that may prevent the interrupt from sending any more notifications. \*/**

**ulTaskNotifyTakeIndexed( 0, /\* Use the 0th notification \*/**

**pdTRUE, /\* Clear the notification value**

**before exiting. \*/**

**portMAX\_DELAY ); /\* Block indefinitely. \*/**

**/\* The RTOS task notification is used as a binary (as opposed to a**

**counting) semaphore, so only go back to wait for further notifications**

**when all events pending in the peripheral have been processed. \*/**

**do**

**{**

**xEvent = xQueryPeripheral();**

**if( xEvent != NO\_MORE\_EVENTS )**

**{**

**vProcessPeripheralEvent( xEvent );**

**}**

**} while( xEvent != NO\_MORE\_EVENTS );**

**}**

**}**

# xTaskNotify / xTaskNotifyIndexed [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**BaseType\_t xTaskNotify( TaskHandle\_t xTaskToNotify,**

**uint32\_t ulValue,**

**eNotifyAction eAction );**

**BaseType\_t xTaskNotifyIndexed( TaskHandle\_t xTaskToNotify,**

**UBaseType\_t uxIndexToNotify,**

**uint32\_t ulValue,**

**eNotifyAction eAction );**

[如果您使用RTOS任务通知来实现二进制或计数信号量类型行为，请使用更简单的 [xTaskNotifyGive（）](https://www.freertos.org/xTaskNotifyGive.html) API函数而不是xTaskNotify（）。

每个任务都有一个“任务通知”（或仅是“通知”）数组，每个通知都有一个状态和一个32位值。甲[直接到任务通知](https://www.freertos.org/RTOS-task-notifications.html)是一个事件直接发送到可以解锁接收任务，和任选地在许多不同的方式更新接收任务的通知值中的一个任务。例如，通知可能会覆盖接收任务的通知值之一，或者仅在接收任务的通知值之一中设置一个或多个位。

xTaskNotify（）用于直接向RTOS任务发送事件，并有可能解除对它的阻止，并可以选择通过以下其中一种方式更新接收任务的通知值之一：

* 将32位数字写入通知值
* 通知值加（递增）一
* 在通知值中设置一位或多位
* 保持通知值不变

xTaskNotify（）和xTaskNotifyIndexed（）是等效函数–唯一的区别是xTaskNotifyIndexed（）可以对数组内的任何任务通知进行操作，而xTaskNotify（）始终对数组索引0处的任务通知进行操作。

不得从中断服务程序（ISR）调用此函数。请改用 [xTaskNotifyFromISR（）](https://www.freertos.org/xTaskNotifyFromISR.html)。

[要使](https://www.freertos.org/a00110.html#configUSE_TASK_NOTIFICATIONS) 这些功能可用，必须在FreeRTOSConfig.h[中将configUSE\_TASK\_NOTIFICATIONS](https://www.freertos.org/a00110.html" \l "configUSE_TASK_NOTIFICATIONS)设置为1（或保持未定义状态）。常量 [configTASK\_NOTIFICATION\_ARRAY\_ENTRIES](https://www.freertos.org/a00110.html" \l "configTASK_NOTIFICATION_ARRAY_ENTRIES)设置每个任务的任务通知数组中的索引数。

**向后兼容性信息：**  
在FreeRTOS V10.4.0之前，每个任务都具有一个“通知值”，并且所有任务通知API函数都在该值上进行操作。用一组通知值代替单个通知值需要一组新的API函数，这些函数可以处理数组中的特定通知。xTaskNotify（）是原始API函数，并且始终通过对数组索引0处的通知值进行操作来保持向后兼容。调用xTaskNotify（）等效于将uxIndexToNotify参数设置为0的xTaskNotifyIndexed（）。

**参数：**

|  |  |
| --- | --- |
| *xTaskToNotify* | 通知RTOS任务的句柄。这是*目标*任务。  要获取任务的句柄，请使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务 并使用pxCreatedTask参数，或者使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务 并存储返回的值，或者在调用[xTaskGetHandle（）时](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)使用任务名称。  [xTaskGetCurrentTaskHandle（）](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle) API函数 返回当前执行的RTOS任务的[句柄](https://www.freertos.org/a00021.html" \l "xTaskGetCurrentTaskHandle)。 |
| *uxIndexToNotify* | 目标任务的通知值数组中的索引，通知将发送到该索引。  uxIndexToNotify必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。  xTaskNotify（）没有此参数，并且始终将通知发送到索引0。 |
| *ulValue* | 用于更新目标任务的通知值。请参阅下面的eAction参数说明。 |
| *电子行动* | 一个枚举类型，可以采用下表中记录的值之一来执行关联的操作。 |

|  |  |
| --- | --- |
| **eAction设置** | **动作已执行** |
| eNoAction | 目标任务接收到该事件，但是其通知值未更新。在这种情况下，不使用ulValue。 |
| eSetBits | 目标任务的通知值将与ulValue按位或。例如，如果ulValue设置为0x01，则将在目标任务的通知值内设置位0。同样，如果ulValue为0x04，则将在目标任务的通知值中设置位2。这样，RTOS任务通知机制可以用作[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)的轻量级替代方案 。 |
| 电子增量 | 目标任务的通知值将增加一，从而使对xTaskNotify（）的调用等效于对[xTaskNotifyGive（）](https://www.freertos.org/xTaskNotifyGive.html)的调用。在这种情况下，不使用ulValue。 |
| eSetValueWithOverwrite | 目标任务的通知值无条件设置为ulValue。通过这种方式，RTOS任务通知机制被用作[xQueueOverwrite（）](https://www.freertos.org/xQueueOverwrite.html)的轻量替代方案。 |
| eSetValueWithoutOverwrite | 如果目标任务尚未有待处理的通知，则其通知值将设置为ulValue。  如果目标任务已经有一个待处理的通知，则其通知值不会更新，因为这样做会在使用前覆盖先前的值。在这种情况下，对xTaskNotify（）的调用将失败，并返回pdFALSE。  这样，在长度为1的队列中，RTOS任务通知机制被用作[xQueueSend（）](https://www.freertos.org/a00117.html)的轻量替代方案 。 |

**返回值：**

除将eAction设置为eSetValueWithoutOverwrite且目标任务的通知值无法更新（因为目标任务已经有待处理的通知）以外，在所有情况下都返回pdPASS。

**用法示例：**

[更多示例参见[RTOS任务通知主页面](https://www.freertos.org/RTOS-task-notifications.html)]

**/\* Set bit 8 in the 0th notification value of the task referenced by**

**xTask1Handle. \*/**

**xTaskNotifyIndexed( xTask1Handle, 0, ( 1UL << 8UL ), eSetBits );**

**/\* Send a notification to the task referenced by xTask2Handle, potentially**

**removing the task from the Blocked state, but without updating the task’s**

**notification value. \*/**

**xTaskNotify( xTask2Handle, 0, eNoAction );**

**/\* Set the notification value of the task referenced by xTask3Handle to 0x50,**

**even if the task had not read its previous notification value. \*/**

**xTaskNotify( xTask3Handle, 0x50, eSetValueWithOverwrite );**

**/\* Set the notification value of the task referenced by xTask4Handle to 0xfff,**

**but only if to do so would not overwrite the task’s existing notification**

**value before the task had obtained it (by a call to** [**xTaskNotifyWait()**](https://www.freertos.org/xTaskNotifyWait.html)

**or** [**ulTaskNotifyTake()**](https://www.freertos.org/ulTaskNotifyTake.html)**). \*/**

**if( xTaskNotify( xTask4Handle, 0xfff, eSetValueWithoutOverwrite ) == pdPASS )**

**{**

**/\* The task’s notification value was updated. \*/**

**}**

**else**

**{**

**/\* The task’s notification value was not updated. \*/**

**}**

# xTaskNotifyAndQuery / xTaskNotifyAndQueryIndexed [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**BaseType\_t xTaskNotifyAndQuery( TaskHandle\_t xTaskToNotify,**

**uint32\_t ulValue,**

**eNotifyAction eAction,**

**uint32\_t \*pulPreviousNotifyValue );**

**BaseType\_t xTaskNotifyAndQueryIndexed( TaskHandle\_t xTaskToNotify,**

**UBaseType\_t uxIndexToNotify,**

**uint32\_t ulValue,**

**eNotifyAction eAction,**

**uint32\_t \*pulPreviousNotifyValue );**

有关更多详细信息，请参见 [RTOS任务通知](https://www.freertos.org/RTOS-task-notifications.html)。

xTaskNotifyAndQueryIndexed（）执行相同的操作， [xTaskNotifyIndexed（）](https://www.freertos.org/xTaskNotify.html)与所述另外，它也返回目标任务的预先通知的值（在被调用的函数，而不是当时间通知值在函数返回）在附加pulPreviousNotifyValue参数。

xTaskNotifyAndQuery（）执行与xTaskNotify（）相同的操作， 另外它还会在附加的[pulPreviousNotifyValue中](https://www.freertos.org/xTaskNotify.html)返回目标任务的先前通知值（调用函数时（而不是函数返回时的）通知值）。参数。

不得从中断服务程序（ISR）调用此函数。请改用 [xTaskNotifyAndQueryFromISR（）](https://www.freertos.org/xTaskNotifyAndQueryFromISR.html) 。

**参数：**

|  |  |
| --- | --- |
| *xTaskToNotify* | 通知RTOS任务的句柄。这是*目标*任务。  要获取任务的句柄，请使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务 并使用pxCreatedTask参数，或者使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务 并存储返回的值，或者在调用[xTaskGetHandle（）时](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)使用任务名称。  [xTaskGetCurrentTaskHandle（）](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle) API函数 返回当前执行的RTOS任务的[句柄](https://www.freertos.org/a00021.html" \l "xTaskGetCurrentTaskHandle)。 |
| *uxIndexToNotify* | 目标任务的通知值数组中的索引，通知将发送到该索引。  uxIndexToNotify必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。 |
| *ulValue* | 用于更新目标任务的通知值。请参阅下面的eAction参数说明。 |
| *eAction* | 一个枚举类型，可以采用下表中记录的值之一来执行关联的操作。 |
| *pulPreviousNotifyValue* | 可用于在xTaskNotifyAndQuery（）的操作修改任何位之前传递目标任务的通知值。  pulPreviousNotifyValue是一个可选参数，如果不需要，可以将其设置为NULL。如果未使用[pulPreviousNotifyValue，](https://www.freertos.org/xTaskNotify.html)则考虑使用[xTaskNotify（）](https://www.freertos.org/xTaskNotify.html) 代替xTaskNotifyAndQuery（）。 |

|  |  |
| --- | --- |
| **eAction设置** | **动作已执行** |
| eNoAction | 目标任务接收到该事件，但是其通知值未更新。在这种情况下，不使用ulValue。 |
| eSetBits | 目标任务的通知值将与ulValue按位或。例如，如果ulValue设置为0x01，则将在目标任务的通知值内设置位0。同样，如果ulValue为0x04，则将在目标任务的通知值中设置位2。这样，RTOS任务通知机制可以用作[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)的轻量级替代方案 。 |
| eIncrement | 目标任务的通知值将增加一，从而使对xTaskNotify（）的调用等效于对[xTaskNotifyGive（）](https://www.freertos.org/xTaskNotifyGive.html)的调用。在这种情况下，不使用ulValue。 |
| eSetValueWithOverwrite | 目标任务的通知值无条件设置为ulValue。通过这种方式，RTOS任务通知机制被用作[xQueueOverwrite（）](https://www.freertos.org/xQueueOverwrite.html)的轻量替代方案。 |
| eSetValueWithoutOverwrite | 如果目标任务尚未有待处理的通知，则其通知值将设置为ulValue。  如果目标任务已经有一个待处理的通知，则其通知值不会更新，因为这样做会在使用前覆盖先前的值。在这种情况下，对xTaskNotify（）的调用将失败，并返回pdFALSE。  这样，在长度为1的队列中，RTOS任务通知机制被用作[xQueueSend（）](https://www.freertos.org/a00117.html)的轻量替代方案 。 |

**返回值：**

除将eAction设置为eSetValueWithoutOverwrite且目标任务的通知值无法更新（因为目标任务已经有待处理的通知）以外，在所有情况下都返回pdPASS。

**用法示例：**

**uint32\_t ulPreviousValue;**

**/\* Set bit 8 in the 0th notification value of the task referenced**

**by xTask1Handle. Store the task’s previous 0th notification**

**value (before bit 8 is set) in ulPreviousValue. \*/**

**xTaskNotifyAndQueryIndexed( xTask1Handle,**

**0,**

**( 1UL << 8UL ),**

**eSetBits,**

**&ulPreviousValue );**

**/\* Send a notification to the task referenced by xTask2Handle,**

**potentially removing the task from the Blocked state, but without**

**updating the task’s notification value. Store the tasks notification**

**value in ulPreviousValue. \*/**

**xTaskNotifyAndQuery( xTask2Handle, 0, eNoAction, &ulPreviousValue );**

**/\* Set the notification value of the task referenced by xTask3Handle**

**to 0x50, even if the task had not read its previous notification value.**

**The task’s previous notification value is of no interest so the last**

**parameter is set to NULL. \*/**

**xTaskNotifyAndQuery( xTask3Handle, 0x50, eSetValueWithOverwrite, NULL );**

**/\* Set the notification value of the task referenced by xTask4Handle**

**to 0xfff,**

**but only if to do so would not overwrite the task’s existing notification**

**value before the task had obtained it (by a call to** [**xTaskNotifyWait()**](https://www.freertos.org/xTaskNotifyWait.html)

**or** [**ulTaskNotifyTake()**](https://www.freertos.org/ulTaskNotifyTake.html)**). The task’s previous notification value is saved**

**in ulPreviousValue. \*/**

**if( xTaskNotifyAndQuery( xTask4Handle,**

**0xfff,**

**eSetValueWithoutOverwrite,**

**&ulPreviousValue ) == pdPASS )**

**{**

**/\* The task’s notification value was updated. \*/**

**}**

**else**

**{**

**/\* The task’s notification value was not updated. \*/**

**}**

# xTaskNotifyAndQueryFromISR / xTaskNotifyAndQueryIndexedFromISR [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**BaseType\_t xTaskNotifyAndQueryFromISR(**

**TaskHandle\_t xTaskToNotify,**

**uint32\_t ulValue,**

**eNotifyAction eAction,**

**uint32\_t \*pulPreviousNotifyValue,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

**BaseType\_t xTaskNotifyAndQueryIndexedFromISR(**

**TaskHandle\_t xTaskToNotify,**

**UBaseType\_t uxIndexToNotify**

**uint32\_t ulValue,**

**eNotifyAction eAction,**

**uint32\_t \*pulPreviousNotifyValue,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

有关更多详细信息，请参见 [RTOS任务通知](https://www.freertos.org/RTOS-task-notifications.html)。

xTaskNotifyAndQueryIndexedFromISR（）执行相同的操作， [xTaskNotifyIndexedFromISR（）](https://www.freertos.org/xTaskNotifyFromISR.html)与所述另外，它也返回目标任务的预先通知的值（在时间通知值的函数被调用，而不是在时间函数返回）在附加pulPreviousNotifyValue参数。

xTaskNotifyAndQueryFromISR（）执行相同的操作， [xTaskNotifyFromISR（）](https://www.freertos.org/xTaskNotifyFromISR.html)与所述另外，它也返回目标任务的预先通知的值（在时间通知值的函数被调用，而不是在时间函数返回）在附加pulPreviousNotifyValue参数。

**参数：**

|  |  |
| --- | --- |
| *xTaskToNotify* | 通知RTOS任务的句柄。这是*目标*任务。  要获取任务的句柄，请使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务 并使用pxCreatedTask参数，或者使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务 并存储返回的值，或者在调用[xTaskGetHandle（）时](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)使用任务名称。  [xTaskGetCurrentTaskHandle（）](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle) API函数 返回当前执行的RTOS任务的[句柄](https://www.freertos.org/a00021.html" \l "xTaskGetCurrentTaskHandle)。 |
| *uxIndexToNotify* | 目标任务的通知值数组中的索引，通知将发送到该索引。  uxIndexToNotify必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。 |
| *ulValue* | 用于更新目标任务的通知值。请参阅下面的eAction参数说明。 |
| *eAction* | 一个枚举类型，可以采用下表中记录的值之一来执行关联的操作。 |
| *pulPreviousNotifyValue* | 可用于在xTaskNotifyAndQueryFromISR（）的操作修改任何位之前传递目标任务的通知值。  pulPreviousNotifyValue是一个可选参数，如果不需要，可以将其设置为NULL。如果未使用[pulPreviousNotifyValue，](https://www.freertos.org/xTaskNotify.html)则考虑使用[xTaskNotify（）](https://www.freertos.org/xTaskNotify.html) 代替xTaskNotifyAndQueryFromISR（）。 |
| *pxHigherPriorityTaskWoken* | \* pxHigherPriorityTaskWoken必须初始化为pdFALSE（0）。  如果发送通知导致任务取消阻止，xTaskNotifyAndQueryFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE，并且取消阻止的任务的优先级高于当前运行的任务。  如果xTaskNotifyAndQueryFromISR（）将此值设置为pdTRUE，则应在退出中断之前请求上下文切换。请参见下面的示例。  pxHigherPriorityTaskWoken是一个可选参数，可以设置为NULL。 |

|  |  |
| --- | --- |
| **eAction设置** | **动作已执行** |
| eNoAction | 目标任务接收到该事件，但是其通知值未更新。在这种情况下，不使用ulValue。 |
| eSetBits | 目标任务的通知值将与ulValue按位或。例如，如果ulValue设置为0x01，则将在目标任务的通知值内设置位0。同样，如果ulValue为0x04，则将在目标任务的通知值中设置位2。这样，RTOS任务通知机制可以用作[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)的轻量级替代方案 。 |
| eIncrement | 目标任务的通知值将增加一，从而使对xTaskNotify（）的调用等效于对[xTaskNotifyGive（）](https://www.freertos.org/xTaskNotifyGive.html)的调用。在这种情况下，不使用ulValue。 |
| eSetValueWithOverwrite | 目标任务的通知值无条件设置为ulValue。通过这种方式，RTOS任务通知机制被用作[xQueueOverwrite（）](https://www.freertos.org/xQueueOverwrite.html)的轻量替代方案。 |
| eSetValueWithoutOverwrite | 如果目标任务尚未有待处理的通知，则其通知值将设置为ulValue。  如果目标任务已经有一个待处理的通知，则其通知值不会更新，因为这样做会在使用前覆盖先前的值。在这种情况下，对xTaskNotify（）的调用将失败，并返回pdFALSE。  这样，在长度为1的队列中，RTOS任务通知机制被用作[xQueueSend（）](https://www.freertos.org/a00117.html)的轻量替代方案 。 |

**返回值：**

除将eAction设置为eSetValueWithoutOverwrite且目标任务的通知值无法更新（因为目标任务已经有待处理的通知）以外，在所有情况下都返回pdPASS。

**用法示例：**

**void vAnISR( void )**

**{**

**/\* Must be Initialised to pdFALSE! \*/**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE.**

**uint32\_t ulPreviousValue;**

**/\* Set bit 8 in the 0th notification value of the task referenced**

**by xTask1Handle. Store the task’s previous 0th notification value**

**(before bit 8 is set) in ulPreviousValue. \*/**

**xTaskNotifyAndQueryIndexedFromISR( xTask1Handle,**

**0,**

**( 1UL << 8UL ),**

**eSetBits,**

**&ulPreviousValue,**

**&xHigherPriorityTaskWoken );**

**/\* The task’s previous notification value is saved in**

**ulPreviousValue. \*/**

**/\* If the task referenced by xTask1Handle was in the Blocked**

**state, waiting for the notification, then it will now have been**

**moved from the Blocked state to the Ready state. If its priority**

**is higher than the priority of the currently executing task (the**

**task this interrupt interrupted) then xHigherPriorityTaskWoken will**

**have been set to pdTRUE, and passing the variable into a call to**

**portYIELD\_FROM\_ISR() will result in the interrupt returning directly**

**to the unblocked task. If xHigherPriorityTaskWoken is still pdFALSE**

**then passing it into portYIELD\_FROM\_ISR() will have no effect. \*/**

**portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

# xTaskNotifyFromISR / xTaskNotifyIndexedFromISR [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**BaseType\_t xTaskNotifyFromISR( TaskHandle\_t xTaskToNotify,**

**uint32\_t ulValue,**

**eNotifyAction eAction,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

**BaseType\_t xTaskNotifyIndexedFromISR( TaskHandle\_t xTaskToNotify,**

**UBaseType\_t uxIndexToNotify,**

**uint32\_t ulValue,**

**eNotifyAction eAction,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

可以从中断服务程序（ISR）使用的xTaskNotify（）和xTaskNotifyIndexed（）版本。请参见[xTaskNotify（）](https://www.freertos.org/xTaskNotify.html) API函数的文档页面，以 [了解](https://www.freertos.org/xTaskNotify.html)其操作和必要的配置参数以及向后兼容的信息。

**参数：**

|  |  |
| --- | --- |
| *xTaskToNotify* | 通知RTOS任务的句柄。这是*目标*任务。  要获取任务的句柄，请使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务 并使用pxCreatedTask参数，或者使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务 并存储返回的值，或者在调用[xTaskGetHandle（）时](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)使用任务名称。  [xTaskGetCurrentTaskHandle（）](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle) API函数 返回当前执行的RTOS任务的[句柄](https://www.freertos.org/a00021.html" \l "xTaskGetCurrentTaskHandle)。 |
| *uxIndexToNotify* | 目标任务的通知值数组中的索引，通知将发送到该索引。  uxIndexToNotify必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。  xTaskNotifyFromISR（）没有此参数，并且始终将通知发送到索引0。 |
| *ulValue* | 用于更新目标任务的通知值。请参阅下面的eAction参数说明。 |
| *eAction* | 一个枚举类型，可以采用下表中记录的值之一来执行关联的操作。 |
| *pxHigherPriorityTaskWoken* | \* pxHigherPriorityTaskWoken必须初始化为0。  如果发送通知导致任务取消阻止，xTaskNotifyFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE，并且取消阻止的任务的优先级高于当前运行的任务。  如果xTaskNotifyFromISR（）将此值设置为pdTRUE，则应在退出中断之前请求上下文切换。请参见下面的示例。  pxHigherPriorityTaskWoken是一个可选参数，可以设置为NULL。 |

|  |  |
| --- | --- |
| **eAction设置** | **动作已执行** |
| eNoAction | 目标任务接收到该事件，但是其通知值未更新。在这种情况下，不使用ulValue。 |
| eSetBits | 目标任务的通知值将与ulValue按位或。例如，如果ulValue设置为0x01，则将在目标任务的通知值内设置位0。同样，如果ulValue为0x04，则将在目标任务的通知值中设置位2。这样，RTOS任务通知机制可以用作[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)的轻量级替代方案 。 |
| eIncrement | 目标任务的通知值将增加1，从而使对xTaskNotifyFromISR（）的调用等效于对[vTaskNotifyGiveFromISR（）](https://www.freertos.org/vTaskNotifyGiveFromISR.html)的调用。在这种情况下，不使用ulValue。 |
| eSetValueWithOverwrite | 目标任务的通知值无条件设置为ulValue。通过这种方式，RTOS任务通知机制被用作[xQueueOverwrite（）](https://www.freertos.org/xQueueOverwrite.html)的轻量替代方案。 |
| eSetValueWithoutOverwrite | 如果目标任务尚未有待处理的通知，则其通知值将设置为ulValue。  如果目标任务已经有一个待处理的通知，则其通知值不会更新，因为这样做会在使用前覆盖先前的值。在这种情况下，对xTaskNotify（）的调用将失败。  这样，在长度为1的队列中，RTOS任务通知机制被用作[xQueueSend（）](https://www.freertos.org/a00117.html)的轻量替代方案 。 |

**返回值：**

除将eAction设置为eSetValueWithoutOverwrite且目标任务的通知值无法更新（因为目标任务已经有待处理的通知）以外，在所有情况下都返回pdPASS。

**用法示例：**

[更多示例参见[RTOS任务通知主页面](https://www.freertos.org/RTOS-task-notifications.html)]

本示例演示如何将xTaskNotifyFromISR（）与eSetBits操作一起使用。有关显示如何使用eNoAction，eSetValueWithOverwrite和eSetValueWithoutOverwrite操作的示例，请参见[xTaskNotify（）](https://www.freertos.org/xTaskNotify.html) API文档页面。

**/\* The interrupt handler does not perform any processing itself. Instead it**

**it unblocks a high priority task in which the events that generated the**

**interrupt are processed. If the priority of the task is high enough then the**

**interrupt will return directly to the task (so it will interrupt one task but**

**return to a different task), so the processing will occur contiguously in time –**

**just as if all the processing had been done in the interrupt handler itself.**

**The status of the interrupting peripheral is sent to the task using an RTOS task**

**notification. \*/**

**void vANInterruptHandler( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken;**

**uint32\_t ulStatusRegister;**

**/\* Read the interrupt status register which has a bit for each interrupt**

**source (for example, maybe an Rx bit, a Tx bit, a buffer overrun bit, etc. \*/**

**ulStatusRegister = ulReadPeripheralInterruptStatus();**

**/\* Clear the interrupts. \*/**

**vClearPeripheralInterruptStatus( ulStatusRegister );**

**/\* xHigherPriorityTaskWoken must be initialised to pdFALSE. If calling**

**xTaskNotifyFromISR() unblocks the handling task, and the priority of**

**the handling task is higher than the priority of the currently running task,**

**then xHigherPriorityTaskWoken will automatically get set to pdTRUE. \*/**

**xHigherPriorityTaskWoken = pdFALSE;**

**/\* Unblock the handling task so the task can perform any processing necessitated**

**by the interrupt. xHandlingTask is the task’s handle, which was obtained**

[**when the task was created**](https://www.freertos.org/a00125.html)**. The handling task’s 0th notification value**

**is bitwise ORed with the interrupt status – ensuring bits that are already**

**set are not overwritten. \*/**

**xTaskNotifyIndexedFromISR( xHandlingTask,**

**0,**

**ulStatusRegister,**

**eSetBits,**

**&xHigherPriorityTaskWoken );**

**/\* Force a context switch if xHigherPriorityTaskWoken is now set to pdTRUE.**

**The macro used to do this is dependent on the port and may be called**

**portEND\_SWITCHING\_ISR. \*/**

**portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

**/\* ———————————————————– \*/**

**/\* A task that blocks waiting to be notified that the peripheral needs servicing,**

**processing all the events pending in the peripheral each time it is notified to**

**do so. \*/**

**void vHandlingTask( void \*pvParameters )**

**{**

**uint32\_t ulInterruptStatus;**

**for( ;; )**

**{**

**/\* Block indefinitely (without a timeout, so no need to check the function’s**

**return value) to wait for a notification. NOTE! Real applications**

**should not block indefinitely, but instead time out occasionally in order**

**to handle error conditions that may prevent the interrupt from sending**

**any more notifications. \*/**

[**xTaskNotifyWaitIndexed**](https://www.freertos.org/xTaskNotifyWait.html)**( 0, /\* Wait for 0th Notificaition \*/**

**0x00, /\* Don’t clear any bits on entry. \*/**

**ULONG\_MAX, /\* Clear all bits on exit. \*/**

**&ulInterruptStatus, /\* Receives the notification value. \*/**

**portMAX\_DELAY ); /\* Block indefinitely. \*/**

**/\* Process any bits set in the received notification value. This assumes**

**the peripheral sets bit 1 for an Rx interrupt, bit 2 for a Tx interrupt,**

**and bit 3 for a buffer overrun interrupt. \*/**

**if( ( ulInterruptStatus & 0x01 ) != 0x00 )**

**{**

**prvProcessRxInterrupt();**

**}**

**if( ( ulInterruptStatus & 0x02 ) != 0x00 )**

**{**

**prvProcessTxInterrupt();**

**}**

**if( ( ulInterruptStatus & 0x04 ) != 0x00 )**

**{**

**prvClearBufferOverrun();**

**}**

**}**

**}**

# xTaskNotifyWait / xTaskNotifyWaitIndexed [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**BaseType\_t xTaskNotifyWait( uint32\_t ulBitsToClearOnEntry,**

**uint32\_t ulBitsToClearOnExit,**

**uint32\_t \*pulNotificationValue,**

**TickType\_t xTicksToWait );**

**BaseType\_t xTaskNotifyWaitIndexed( UBaseType\_t uxIndexToWaitOn,**

**uint32\_t ulBitsToClearOnEntry,**

**uint32\_t ulBitsToClearOnExit,**

**uint32\_t \*pulNotificationValue,**

**TickType\_t xTicksToWait );**

[如果使用RTOS任务通知来实现二进制或计数信号量类型行为，则使用更简单的 [ulTask​​NotifyTake（）](https://www.freertos.org/ulTaskNotifyTake.html) API函数代替xTaskNotifyWait（）。

每个任务都有一个“任务通知”（或仅是“通知”）数组，每个通知都有一个状态和一个32位值。甲[直接到任务通知](https://www.freertos.org/RTOS-task-notifications.html)是一个事件直接发送到可以解锁接收任务，和任选地在许多不同的方式更新接收任务的通知值中的一个任务。例如，通知可能会覆盖接收任务的通知值之一，或者仅在接收任务的通知值之一中设置一个或多个位。

xTaskNotifyWait（）以可选的超时等待调用任务接收通知。如果正在等待通知的接收RTOS任务已被阻止，则在等待通知到达时，将从已阻止状态中删除接收RTOS任务，并清除通知。

**注意：**阵列中的每个通知都是独立运行的–一项任务一次只能阻塞一个阵列中的一个通知，而不会被发送给任何其他阵列索引的通知所阻塞。

xTaskNotifyWait（）和xTaskNotifyWaitIndexed（）是等效的宏–唯一的区别是xTaskNotifyWaitIndexed（）可以对数组内的任何任务通知进行操作，而xTaskNotifyWait（）始终对数组索引0处的任务通知进行操作。

不得从中断服务程序中调用xTaskNotifyGive（）。请改用 [vTaskNotifyGiveFromISR（）](https://www.freertos.org/vTaskNotifyGiveFromISR.html)。

[为了](https://www.freertos.org/a00110.html#configUSE_TASK_NOTIFICATIONS) 使这些宏可用，必须在FreeRTOSConfig.h[中将configUSE\_TASK\_NOTIFICATIONS](https://www.freertos.org/a00110.html" \l "configUSE_TASK_NOTIFICATIONS)设置为1（或保持未定义状态）。常量 [configTASK\_NOTIFICATION\_ARRAY\_ENTRIES](https://www.freertos.org/a00110.html" \l "configTASK_NOTIFICATION_ARRAY_ENTRIES)设置每个任务的任务通知数组中的索引数。

**向后兼容性信息：**  
在FreeRTOS V10.4.0之前，每个任务都具有一个“通知值”，并且所有任务通知API函数都在该值上进行操作。用一组通知值代替单个通知值需要一组新的API函数，这些函数可以处理数组中的特定通知。xTaskNotifyWait（）是原始的API函数，并且通过始终对数组中索引0处的通知值进行操作来保持向后兼容。调用xTaskNotifyWait（）等效于将uxIndexToWaitOn参数设置为0的情况下调用xTaskNotifyWaitIndexed（）。

**参数：**

|  |  |
| --- | --- |
| *uxIndexToWaitOn* | 调用任务的通知值数组中的索引，调用任务将在该索引上等待接收到通知。  uxIndexToWaitOn必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。  xTaskNotifyWait（）没有此参数，并且始终等待索引0上的通知。 |
| *ulBitsToClearOnEntry* | 如果调用xTaskNotifyWait（）时通知尚未挂起，则在调用xTaskNotifyWait（）函数时（在任务等待新通知之前），将在调用RTOS任务的通知值中清除ulBitsToClearOnEntry中设置的任何位。  例如，如果ulBitsToClearOnEntry为0x01，则在进入函数时将清除任务通知值的位0。  将ulBitsToClearOnEntry设置为0xffffffff（ULONG\_MAX）将清除任务通知值中的所有位，从而有效地将值清除为0。 |
| *ulBitsToClearOnExit* | 如果收到通知，则xTaskNotifyWait（）函数退出之前，将在调用RTOS任务的通知值中清除ulBitsToClearOnExit中设置的任何位。  将RTOS任务的通知值保存在\* pulNotificationValue中后，将清除这些位（请参阅下面的pulNotificationValue描述）。  例如，如果ulBitsToClearOnExit为0x03，则在函数退出之前，将清除任务通知值的位0和位1。  将ulBitsToClearOnExit设置为0xffffffff（ULONG\_MAX）将清除任务通知值中的所有位，从而有效地将值清除为0。 |
| *pulNotificationValue* | 用于传递RTOS任务的通知值。复制到\* pulNotificationValue的值是由于ulBitsToClearOnExit设置而清除任何位之前的RTOS任务的通知值。  如果不需要通知值，则将pulNotificationValue设置为NULL。 |
| *xTicksToWait* | 如果在调用xTaskNotifyWait（）时通知尚未挂起，则在“阻塞”状态下等待接收通知的最长时间。  处于阻塞状态时，RTOS任务不会消耗任何CPU时间。  时间以RTOS滴答周期指定。pdMS\_TO\_TICKS（）宏可用于将以毫秒为单位的时间转换为以刻度为单位的时间。 |

**返回值：**

如果已收到通知，或者在调用xTaskNotifyWait（）时通知已待处理，则为pdTRUE。

如果在收到通知之前对xTaskNotifyWait（）的调用超时，则为pdFALSE。

**用法示例：**

**/\* This task shows bits within the RTOS task notification value being used to pass**

**different events to the task in the same way that flags in an** [**event group**](https://www.freertos.org/FreeRTOS-Event-Groups.html) **might**

**be used for the same purpose. \*/**

**void vAnEventProcessingTask( void \*pvParameters )**

**{**

**uint32\_t ulNotifiedValue;**

**for( ;; )**

**{**

**/\* Block indefinitely (without a timeout, so no need to check the function’s**

**return value) to wait for a notification.**

**Bits in this RTOS task’s notification value are set by the notifying**

**tasks and interrupts to indicate which events have occurred. \*/**

**xTaskNotifyWaitIndexed( 0, /\* Wait for 0th notification. \*/**

**0x00, /\* Don’t clear any notification bits on entry. \*/**

**ULONG\_MAX, /\* Reset the notification value to 0 on exit. \*/**

**&ulNotifiedValue, /\* Notified value pass out in**

**ulNotifiedValue. \*/**

**portMAX\_DELAY ); /\* Block indefinitely. \*/**

**/\* Process any events that have been latched in the notified value. \*/**

**if( ( ulNotifiedValue & 0x01 ) != 0 )**

**{**

**/\* Bit 0 was set – process whichever event is represented by bit 0. \*/**

**prvProcessBit0Event();**

**}**

**if( ( ulNotifiedValue & 0x02 ) != 0 )**

**{**

**/\* Bit 1 was set – process whichever event is represented by bit 1. \*/**

**prvProcessBit1Event();**

**}**

**if( ( ulNotifiedValue & 0x04 ) != 0 )**

**{**

**/\* Bit 2 was set – process whichever event is represented by bit 2. \*/**

**prvProcessBit2Event();**

**}**

**/\* Etc. \*/**

**}**

**}**

# xTaskNotifyStateClear / xTaskNotifyStateClearIndexed [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**BaseType\_t xTaskNotifyStateClear( TaskHandle\_t xTask );**

**BaseType\_t xTaskNotifyStateClearIndexed( TaskHandle\_t xTask,**

**UBaseType\_t uxIndexToClear );**

每个RTOS任务都有一系列*[任务通知](https://www.freertos.org/RTOS-task-notifications.html)*。每个任务通知都有一个可以为“待处理”或“不待处理”的*通知状态*，以及一个32位*通知值*。

如果将通知发送到通知数组中的索引，则该索引处的通知被称为“挂起”，直到任务读取其通知值或通过调用xTaskNotifyStateClear（）将通知状态明确清除为“未挂起”状态为止。

xTaskNotifyStateClear（）和xTaskNotifyStateClearIndexed（）是等效的宏–唯一的区别是xTaskNotifyStateClearIndexed（）可以对数组内的任何任务通知进行操作，而xTaskNotifyStateClear（）始终对数组索引0处的任务通知进行操作。

[为了](https://www.freertos.org/a00110.html#configUSE_TASK_NOTIFICATIONS) 使这些宏可用，必须在FreeRTOSConfig.h[中将configUSE\_TASK\_NOTIFICATIONS](https://www.freertos.org/a00110.html" \l "configUSE_TASK_NOTIFICATIONS)设置为1（或保持未定义状态）。常量 [configTASK\_NOTIFICATION\_ARRAY\_ENTRIES](https://www.freertos.org/a00110.html" \l "configTASK_NOTIFICATION_ARRAY_ENTRIES)设置每个任务的任务通知数组中的索引数。

**向后兼容性信息：**  
在FreeRTOS V10.4.0之前，每个任务都具有一个“通知值”，并且所有任务通知API函数都在该值上进行操作。用一组通知值代替单个通知值需要一组新的API函数，这些函数可以处理数组中的特定通知。xTaskNotifyStateClear（）是原始的API函数，并且通过始终对数组中索引0处的通知值进行操作来保持向后兼容。调用xTaskNotifyStateClear（）等效于将uxIndexToNotify参数设置为0的xTaskNotifyStateClearIndexed（）。

**参数：**

|  |  |
| --- | --- |
| *xTask* | 将清除其通知状态的RTOS任务的句柄。将xTask设置为NULL可清除调用任务的通知状态。  要获取任务的句柄，请使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务 并使用pxCreatedTask参数，或者使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务 并存储返回的值，或者在调用[xTaskGetHandle（）时](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)使用任务名称。  [xTaskGetCurrentTaskHandle（）](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle) API函数 返回当前执行的RTOS任务的[句柄](https://www.freertos.org/a00021.html" \l "xTaskGetCurrentTaskHandle)。 |
| *uxIndexToClear* | 目标任务的通知值数组中要执行的索引。例如，将uxIndexToClear设置为1将清除数组中索引1处的通知状态。  uxIndexToClear必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。  ulTask​​NotifyStateClear（）没有此参数，并且始终对索引为0的通知起作用。 |

**返回值：**

如果xTask引用的任务有待处理的通知，并且该通知已清除，则返回pdTRUE。如果xTask引用的任务没有通知待处理，则返回pdFALSE。

**用法示例：**

[更多示例参见[RTOS任务通知主页面](https://www.freertos.org/RTOS-task-notifications.html)]

**/\* An example UART send function. The function starts a UART transmission then**

**waits to be notified that the transmission is complete. The transmission**

**complete notification is sent from the UART interrupt. The calling task’s**

**notification state is cleared before the transmission is started to ensure it is**

**not co-incidentally already pending before the task attempts to block on its**

**notification state. \*/**

**void vSerialPutString( const signed char \* const pcStringToSend,**

**unsigned short usStringLength )**

**{**

**const TickType\_t xMaxBlockTime = pdMS\_TO\_TICKS( 5000 );**

**/\* xSendingTask holds the handle of the task waiting for the transmission to**

**complete. If xSendingTask is NULL then a transmission is not in progress.**

**Don’t start to send a new string unless transmission of the previous string**

**is complete. \*/**

**if( ( xSendingTask == NULL ) && ( usStringLength > 0 ) )**

**{**

**/\* Ensure the calling task’s 0th notification state is not already**

**pending. \*/**

**xTaskNotifyStateClearIndexed( NULL, 0 );**

**/\* Store the handle of the transmitting task. This is used to unblock**

**the task when the transmission has completed. \*/**

**xSendingTask = xTaskGetCurrentTaskHandle();**

**/\* Start sending the string – the transmission is then controlled by an**

**interrupt. \*/**

**UARTSendString( pcStringToSend, usStringLength );**

**/\* Wait in the Blocked state (so not using any CPU time) until the UART**

**ISR sends the 0th notification to xSendingTask to notify (and unblock) the**

**task when the transmission is complete. \*/**

[**ulTaskNotifyTake**](https://www.freertos.org/ulTaskNotifyTake.html)**( 0, pdTRUE, xMaxBlockTime );**

**}**

**}**

# ulTaskNotifyValueClear / ulTaskNotifyValueClearIndexed [[RTOS Task Notification API](https://www.freertos.org/RTOS-task-notification-API.html)]

task.h

**uint32\_t ulTaskNotifyValueClear( TaskHandle\_t xTask,**

**uint32\_t ulBitsToClear );**

**uint32\_t ulTaskNotifyValueClearIndexed( TaskHandle\_t xTask,**

**UBaseType\_t uxIndexToClear,**

**uint32\_t ulBitsToClear );**

每个RTOS任务都有一系列*[任务通知](https://www.freertos.org/RTOS-task-notifications.html)*。每个任务通知都有一个可以为“待处理”或“不待处理”的*通知状态*，以及一个32位*通知值*。

ulTask​​NotifyValueClearIndexed（）清除xTask引用的任务的数组索引uxIndexToClear的通知值中ulBitsToClear位掩码所指定的位。

ulTask​​NotifyValueClear（）和ulTask​​NotifyValueClearIndexed（）是等效的宏–唯一的区别是ulTask​​NotifyValueClearIndexed（）可以对数组内的任何任务通知进行操作，而ulTask​​NotifyValueClear（）始终对数组索引0处的任务通知进行操作。

[为了](https://www.freertos.org/a00110.html#configUSE_TASK_NOTIFICATIONS) 使这些宏可用，必须在FreeRTOSConfig.h[中将configUSE\_TASK\_NOTIFICATIONS](https://www.freertos.org/a00110.html" \l "configUSE_TASK_NOTIFICATIONS)设置为1（或保持未定义状态）。常量 [configTASK\_NOTIFICATION\_ARRAY\_ENTRIES](https://www.freertos.org/a00110.html" \l "configTASK_NOTIFICATION_ARRAY_ENTRIES)设置每个任务的任务通知数组中的索引数。

**参数：**

|  |  |
| --- | --- |
| *xTask* | RTOS任务的句柄，将清除其通知值中的位。将xTask设置为NULL可清除调用任务的通知值中的位。  要获取任务的句柄，请使用[xTaskCreate（）](https://www.freertos.org/a00125.html)创建任务并使用pxCreatedTask参数，或者使用[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务并存储返回的值，或者在调用[xTaskGetHandle（）时](https://www.freertos.org/a00021.html" \l "xTaskGetHandle)使用任务名称。  [xTaskGetCurrentTaskHandle（）](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle) API函数 返回当前执行的RTOS任务的[句柄](https://www.freertos.org/a00021.html" \l "xTaskGetCurrentTaskHandle)。 |
| *uxIndexToClear* | 目标任务的通知值数组中的索引，用于在其中清除位。uxIndexToClear必须小于configTASK\_NOTIFICATION\_ARRAY\_ENTRIES。ulTask​​NotifyValueClear（）没有此参数，并且始终清除索引0处的通知值中的位。 |
| *ulBitsToClear* | xTask的通知值中要清除的位的位掩码。将位设置为1可以清除任务的通知值中的相应位。将ulBitsToClear设置为0xffffffff（在32位体系结构上为UINT\_MAX）以将通知值清除为0。将ulBitsToClear设置为0以查询任务的通知值而不清除任何位。 |

**返回值：**

在清除由ulBitsToClear指定的位之前，目标任务的通知值的值。

**用法示例：**

**#define MESSAGE\_RECEIVED\_BIT 8**

**#define TICKS\_UNTIL\_TIMEOUT 100**

**unsigned long ulNotification, ulMessageReceivedMask;**

**/\* Clear any message received events. \*/**

**ulMessageReceivedMask = 1u << MESSAGE\_RECEIVED\_BIT;**

**ulTaskNotifyValueClear( ulMessageReceivedMask );**

**/\* Send a message that expects a response. \*/**

**send\_message();**

**/\* Block this task until it has another pending notification. In this example,**

**the task only ever uses the MESSAGE\_RECEIVED\_BIT of its notification value,**

**so the next event can only ever be on message received. \*/**

[**xTaskNotifyWait**](https://www.freertos.org/xTaskNotifyWait.html)**( 0u, /\* Don’t clear any notification bits on entry. \*/**

**0u, /\* Don’t clear any notification bits on exit. \*/**

**&ulNotification,**

**TICKS\_UNTIL\_TIMEOUT );**

**/\* If there wasn’t a timeout, then the only possible event was received.**

**In this example, that is the MESSAGE\_RECEIVED\_EVENT. \*/**

**if( ulNotification == 0u )**

**{**

**/\* Handle the response timeout. \*/**

**process\_response\_timeout();**

**}**

**else if( ulNotification == ulMessageReceivedMask )**

**{**

**/\* Process the response event. \*/**

**process\_response();**

**ulTaskNotifyValueClear( ulMessageReceivedMask );**

**}**

**else**

**{**

**/\* The example task should only ever receive MESSAGE\_RECEIVED\_EVENTS. \*/**

**process\_error();**

**}**

# 任务实用程序

## 模组

* [**uxTaskGetSystemState**](https://www.freertos.org/uxTaskGetSystemState.html)
* [**vTaskGetInfo**](https://www.freertos.org/vTaskGetInfo.html)
* [**xTaskGetCurrentTaskHandle**](https://www.freertos.org/a00021.html#xTaskGetCurrentTaskHandle)
* [**xTaskGetIdleTaskHandle**](https://www.freertos.org/a00021.html#xTaskGetIdleTaskHandle)
* [**uxTaskGetStackHighWaterMark**](https://www.freertos.org/uxTaskGetStackHighWaterMark.html)
* [**eTaskGetState**](https://www.freertos.org/a00021.html#eTaskGetState)
* [**pcTaskGetName**](https://www.freertos.org/a00021.html#pcTaskGetName)
* [**xTaskGetHandle**](https://www.freertos.org/a00021.html#xTaskGetHandle)
* [**xTaskGetTickCount**](https://www.freertos.org/a00021.html#xTaskGetTickCount)
* [**xTaskGetTickCountFromISR**](https://www.freertos.org/a00021.html#xTaskGetTickCountFromISR)
* [**xTaskGetSchedulerState**](https://www.freertos.org/a00021.html#xTaskGetSchedulerState)
* [**uxTaskGetNumberOfTasks**](https://www.freertos.org/a00021.html#usTaskGetNumberOfTasks)
* [**vTaskList**](https://www.freertos.org/a00021.html#vTaskList)
* [**vTaskStartTrace**](https://www.freertos.org/a00021.html#vTaskStartTrace)
* [**ulTask​​EndTrace**](https://www.freertos.org/a00021.html#usTaskEndTrace)
* [**vTaskGetRunTimeStats**](https://www.freertos.org/a00021.html#vTaskGetRunTimeStats)
* [**vTaskGetIdleRunTimeCounter**](https://www.freertos.org/a00021.html#vTaskGetIdleRunTimeCounter)
* [**vTaskSetApplicationTaskTag**](https://www.freertos.org/vTaskSetApplicationTag.html)
* [**xTaskGetApplicationTaskTag**](https://www.freertos.org/xTaskGetApplicationTaskTag.html)
* [**xTaskCallApplicationTaskHook**](https://www.freertos.org/xTaskCallApplicationTaskHook.html)
* [**pvTaskGetThreadLocalStoragePointer**](https://www.freertos.org/pvTaskGetThreadLocalStoragePointer.html)
* [**vTaskSetThreadLocalStoragePointer**](https://www.freertos.org/vTaskSetThreadLocalStoragePointer.html)
* [**vTaskSetTimeOutState**](https://www.freertos.org/vTaskSetTimeOutState.html)
* [**xTaskCheckForTimeOut**](https://www.freertos.org/xTaskCheckForTimeOut.html)

### xTaskGetCurrentTaskHandle

task.h

**TaskHandle\_t xTaskGetCurrentTaskHandle（void）;**

必须将INCLUDE\_xTaskGetCurrentTaskHandle设置为1才能使用此功能。

**返回值：**

当前正在运行（调用）任务的句柄。

### xTaskGetIdleTaskHandle

task.h

**TaskHandle\_t xTaskGetIdleTaskHandle（void）;**

必须将INCLUDE\_xTaskGetIdleTaskHandle设置为1才能使用此功能。

**返回值：**

与空闲任务关联的任务句柄。启动RTOS计划程序时，将自动创建空闲任务。

### eTaskGetState

task.h

**eTaskState eTaskGetState（TaskHandle\_t xTask）;**

以枚举类型返回执行eTaskGetState（）时任务所处的状态。

必须将FreeRTOSConfig.h中的INCLUDE\_eTaskGetState设置为1，eTaskGetState（）才可用。

另请参见[vTaskGetInfo（）](https://www.freertos.org/vTaskGetInfo.html)。

**参数：**

|  |  |
| --- | --- |
| 任务 | 主题任务（正在查询的任务）的句柄。 |

**返回值：**

下表列出了xTask参数所引用的任务可以存在的每个可能状态，eTaskGetState（）将返回的值。

|  |  |
| --- | --- |
| **状态** | **返回值** |
| Ready | eReady |
| Running | eRunning（调用任务正在查询自己的优先级） |
| Blocked | eBlocked |
| Suspended | eSuspended |
| Deleted | eDeleted（任务TCB正在等待清理） |

### pcTaskGetName

task.h

**char \* pcTaskGetName（TaskHandle\_t xTaskToQuery）;**

从任务的句柄中查找任务的名称。

**参数：**

|  |  |
| --- | --- |
| xTaskToQuery | 正在查询任务的句柄。可以将xTaskToQuery设置为NULL来查询调用任务的名称。 |

**返回值：**

指向主题任务名称的指针，该名称是标准的以NULL结尾的C字符串。

### xTaskGetHandle

task.h

**TaskHandle\_t xTaskGetHandle（const char \* pcNameToQuery）;**

从任务名称中查找任务的句柄。

注意：此功能需要相对较长的时间才能完成，并且每个任务只能调用一次。一旦获得任务的句柄，就可以将其存储在本地以供重复使用。

必须将FreeRTOSConfig.h中的INCLUDE\_xTaskGetHandle设置为1，才能使xTaskGetHandle（）可用。

**参数：**

|  |  |
| --- | --- |
| pcNameToQuery | 将为其返回句柄的任务的文本名称（作为标准的C NULL终止字符串）。  有关 设置任务文本名称的信息，请参见[xTaskCreate（）](https://www.freertos.org/a00125.html) 和[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html) API函数的pcName参数。 |

**返回值：**

如果可以找到具有在pcNameToQuery中传递的名称的任务，则返回任务的句柄，否则返回NULL。

### xTaskGetTickCount

task.h

**volatile TickType\_t xTaskGetTickCount（void）;**

无法从ISR调用此函数。请改用xTaskGetTickCountFromFromISR（）。

**返回值：**

自调用vTaskStartScheduler以来的滴答计数。

### xTaskGetTickCountFromISR

task.h

**volatile TickType\_t xTaskGetTickCountFromISR（void）;**

可以从ISR调用的xTaskGetTickCount（）版本。

**返回值：**

自调用vTaskStartScheduler以来的滴答计数。

### xTaskGetSchedulerState

task.h

**BaseType\_t xTaskGetSchedulerState（void）;**

**返回值：**

以下常量之一（在task.h中定义）：taskSCHEDULER\_NOT\_STARTED，taskSCHEDULER\_RUNNING，taskSCHEDULER\_SUSPENDED。

要使此功能可用，必须在FreeRTOSConfig.h中将INCLUDE\_xTaskGetSchedulerState或configUSE\_TIMERS设置为1。

### uxTaskGetNumberOfTasks

task.h

**UBaseType\_t uxTaskGetNumberOfTasks（void）;**

**返回值：**

RTOS内核当前正在管理的任务数。这包括所有准备就绪，已阻止和已暂停的任务。空闲任务已删除但尚未释放的任务也将包括在计数中。

### vTaskList

task.h

**void vTaskList（char \* pcWriteBuffer）;**

要使此功能可用，必须在FreeRTOSConfig.h中将configUSE\_TRACE\_FACILITY和configUSE\_STATS\_FORMATTING\_FUNCTIONS定义为1。有关更多信息，请参见[配置部分](https://www.freertos.org/a00110.html)。

注意：此功能将在其持续时间内禁用中断。它不适合正常的应用程序运行时使用，而是作为调试辅助。

vTaskList（）调用[uxTaskGetSystemState（）](https://www.freertos.org/uxTaskGetSystemState.html)，然后将[uxTaskGetSystemState（）](https://www.freertos.org/uxTaskGetSystemState.html)生成的原始数据格式化为人类可读（ASCII）表，该表显示每个任务的状态，包括任务堆栈的高水位标记（高水位标记越小，任务越接近堆栈溢出）。 [单击此处查看生成的输出示例](https://www.freertos.org/fr-content-src/uploads/2018/07/log.gif)。

在ASCII表中，以下字母用于表示任务的状态：

* “ B” –已阻塞
* “ R” –准备就绪
* 'D'–删除（等待清理）
* 'S'–暂停或被阻塞，没有超时

vTaskList（）是仅出于方便目的而提供的实用程序函数。它不被视为内核的一部分。有关 实用程序函数的信息，请参见[vTaskGetRunTimeStats（）](https://www.freertos.org/a00021.html" \l "vTaskGetRunTimeStats)，该实用程序函数会生成类似的运行时任务利用率信息表。

**参数：**

|  |  |
| --- | --- |
| pcWriteBuffer | 将上述详细信息写入ASCII格式的缓冲区。假定此缓冲区足够大以包含生成的报告。每个任务大约40个字节就足够了。 |

### vTaskStartTrace

task.h

**void vTaskStartTrace（char \* pcBuffer，unsigned long ulBufferSize）;**

[此功能与FreeRTOS V7.1.0中已删除的旧版跟踪实用程序有关，用户可能会发现更易于使用且功能更强大的更新的[跟踪挂钩宏](https://www.freertos.org/rtos-trace-macros.html)。

启动RTOS内核活动跟踪。跟踪记录何时运行哪个任务的标识。

跟踪文件以二进制格式存储。使用一个单独的DOS实用程序convtrce.exe将其转换为制表符分隔的文本文件，该文件可以在电子表格中查看和绘制。

**参数：**

|  |  |
| --- | --- |
| pcBuffer | 跟踪将写入其中的缓冲区。 |
| ulBufferSize | pcBuffer的大小（以字节为单位）。跟踪将一直持续到缓冲区已满或调用ulTask​​EndTrace（）为止。 |

### ulTask​​EndTrace

task.h

**unsigned long ulTask​​EndTrace（void）;**

[此功能与FreeRTOS V7.1.0中已删除的旧版跟踪实用程序有关，用户可能会发现更易于使用且功能更强大的更新的[跟踪挂钩宏](https://www.freertos.org/rtos-trace-macros.html)。

停止RTOS内核活动跟踪。请参见vTaskStartTrace（）。

**返回值：**

已写入跟踪缓冲区的字节数。

### vTaskGetRunTimeStats

task.h

**void vTaskGetRunTimeStats（char \* pcWriteBuffer）;**

有关此功能的完整说明，请参见“[运行时统计”](https://www.freertos.org/rtos-run-time-stats.html)页面。

configGENERATE\_RUN\_TIME\_STATS，configUSE\_STATS\_FORMATTING\_FUNCTIONS和configSUPPORT\_DYNAMIC\_ALLOCATION必须都定义为1，此功能才可用。然后，应用程序还必须提供portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS（）和portGET\_RUN\_TIME\_COUNTER\_VALUE的定义，以配置外设定时器/计数器并分别返回定时器的当前计数值。计数器应至少是滴答计数频率的10倍。

注意：此功能将在其持续时间内禁用中断。它不适合正常的应用程序运行时使用，而是作为调试辅助。

vTaskGetRunTimeStats（）调用[uxTaskGetSystemState（）](https://www.freertos.org/uxTaskGetSystemState.html)，然后将[uxTaskGetSystemState（）](https://www.freertos.org/uxTaskGetSystemState.html)生成的原始数据格式化为人类可读（ASCII）表，该表显示每个任务在运行状态下所花费的时间（每个任务所消耗的CPU时间） 。数据以绝对值和百分比值形式提供。绝对值的分辨率取决于应用程序提供的运行时统计时钟的频率。

vTaskGetRunTimeStats（）是仅出于方便目的而提供的实用程序函数。它不被视为内核的一部分。有关 生成每个任务状态信息的实用程序功能，请参见[vTaskList（）](https://www.freertos.org/a00021.html#vTaskList)。

**参数：**

|  |  |
| --- | --- |
| pcWriteBuffer | 执行时间将以ASCII形式写入的缓冲区。假定此缓冲区足够大以包含生成的报告。每个任务大约40个字节就足够了。 |

### vTaskGetIdleRunTimeCounter

task.h

**TickType\_t xTaskGetIdleRunTimeCounter（void）;**

返回空闲任务的运行时计数器。此功能可用于确定空闲任务接收了多少CPU时间。有关[运行时状态](https://www.freertos.org/rtos-run-time-stats.html)功能的完整说明，请参见“ [运行时状态”](https://www.freertos.org/rtos-run-time-stats.html)页面。

configGENERATE\_RUN\_TIME\_STATS和INCLUDE\_xTaskGetIdleTaskHandle必须都定义为1，此功能才可用。然后，应用程序还必须提供portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS（）和portGET\_RUN\_TIME\_COUNTER\_VALUE的定义，以配置外设定时器/计数器并分别返回定时器的当前计数值。建议使计时器至少是滴答计数频率的10倍。

# uxTaskGetSystemState() [[Task Utilities](https://www.freertos.org/a00021.html)]

task.h

**UBaseType\_t uxTaskGetSystemState(**

[**TaskStatus\_t**](https://www.freertos.org/uxTaskGetSystemState.html#TaskStatus_t) **\* const pxTaskStatusArray,**

**const UBaseType\_t uxArraySize,**

**unsigned long \* const pulTotalRunTime );**

必须在FreeRTOSConfig.h中将configUSE\_TRACE\_FACILITY定义为1，才能使uxTaskGetSystemState（）可用。

uxTaskGetSystemState（）为系统中的每个任务填充[TaskStatus\_t](https://www.freertos.org/uxTaskGetSystemState.html" \l "TaskStatus_t)结构。TaskStatus\_t结构包含任务句柄的成员，任务名称，任务优先级，任务状态以及任务消耗的运行时间总量。

有关为单个任务而不是每个任务填充TaskStatus\_t结构的版本，请参见[vTaskGetInfo（）](https://www.freertos.org/vTaskGetInfo.html)。

注意：此功能仅用于调试使用，因为其使用会导致调度程序长时间处于挂起状态。

**参数：**

|  |  |
| --- | --- |
| pxTaskStatusArray | 指向[TaskStatus\_t](https://www.freertos.org/uxTaskGetSystemState.html" \l "TaskStatus_t)结构数组的指针。对于受RTOS控制的每个任务，该数组必须至少包含一个TaskStatus\_t结构。可以使用[uxTaskGetNumberOfTasks（）](https://www.freertos.org/a00021.html" \l "usTaskGetNumberOfTasks) API函数来确定RTOS控制下的任务数。 |
| uxArraySize | pxTaskStatusArray参数指向的数组的大小。该大小指定为数组中索引的数量（数组中包含的[TaskStatus\_t](https://www.freertos.org/uxTaskGetSystemState.html#TaskStatus_t)结构的数量），而不是数组中的字节数。 |
| pulTotalRunTime | 如果在FreeRTOSConfig.h中将configGENERATE\_RUN\_TIME\_STATS设置为1，则uxTaskGetSystemState（）将\* pulTotalRunTime设置为自目标启动以来的总运行时间（由[运行时间统计时钟](https://www.freertos.org/rtos-run-time-stats.html)定义）。可以将pulTotalRunTime设置为NULL以忽略总运行时间值。 |

**返回值：**

uxTaskGetSystemState（）填充的[TaskStatus\_t](https://www.freertos.org/uxTaskGetSystemState.html" \l "TaskStatus_t)结构的数量。该值应等于uxTaskGetNumberOfTasks（）API函数返回的数字，但是如果uxArraySize参数中传递的值太小，则该数字将为零。

**用法示例：**

**/\* This example demonstrates how a human readable table of run time stats**

**information is generated from raw data provided by uxTaskGetSystemState().**

**The human readable table is written to pcWriteBuffer. (see the vTaskList()**

**API function which actually does just this). \*/**

**void vTaskGetRunTimeStats( signed char \*pcWriteBuffer )**

**{**

[**TaskStatus\_t**](https://www.freertos.org/uxTaskGetSystemState.html#TaskStatus_t) **\*pxTaskStatusArray;**

**volatile UBaseType\_t uxArraySize, x;**

**unsigned long ulTotalRunTime, ulStatsAsPercentage;**

**/\* Make sure the write buffer does not contain a string. \*/**

**\*pcWriteBuffer = 0x00;**

**/\* Take a snapshot of the number of tasks in case it changes while this**

**function is executing. \*/**

**uxArraySize = uxTaskGetNumberOfTasks();**

**/\* Allocate a TaskStatus\_t structure for each task. An array could be**

**allocated statically at compile time. \*/**

**pxTaskStatusArray = pvPortMalloc( uxArraySize \* sizeof( TaskStatus\_t ) );**

**if( pxTaskStatusArray != NULL )**

**{**

**/\* Generate raw status information about each task. \*/**

**uxArraySize = uxTaskGetSystemState( pxTaskStatusArray,**

**uxArraySize,**

**&ulTotalRunTime );**

**/\* For percentage calculations. \*/**

**ulTotalRunTime /= 100UL;**

**/\* Avoid divide by zero errors. \*/**

**if( ulTotalRunTime > 0 )**

**{**

**/\* For each populated position in the pxTaskStatusArray array,**

**format the raw data as human readable ASCII data. \*/**

**for( x = 0; x < uxArraySize; x++ )**

**{**

**/\* What percentage of the total run time has the task used?**

**This will always be rounded down to the nearest integer.**

**ulTotalRunTimeDiv100 has already been divided by 100. \*/**

**ulStatsAsPercentage =**

**pxTaskStatusArray[ x ].ulRunTimeCounter / ulTotalRunTime;**

**if( ulStatsAsPercentage > 0UL )**

**{**

**sprintf( pcWriteBuffer, "%stt%lutt%lu%%rn",**

**pxTaskStatusArray[ x ].pcTaskName,**

**pxTaskStatusArray[ x ].ulRunTimeCounter,**

**ulStatsAsPercentage );**

**}**

**else**

**{**

**/\* If the percentage is zero here then the task has**

**consumed less than 1% of the total run time. \*/**

**sprintf( pcWriteBuffer, "%stt%lutt<1%%rn",**

**pxTaskStatusArray[ x ].pcTaskName,**

**pxTaskStatusArray[ x ].ulRunTimeCounter );**

**}**

**pcWriteBuffer += strlen( ( char \* ) pcWriteBuffer );**

**}**

**}**

**/\* The array is no longer needed, free the memory it consumes. \*/**

**vPortFree( pxTaskStatusArray );**

**}**

**}**

### The TaskStatus\_t definition

**typedef struct xTASK\_STATUS**

**{**

**/\* The handle of the task to which the rest of the information in the**

**structure relates. \*/**

**TaskHandle\_t xHandle;**

**/\* A pointer to the task's name. This value will be invalid if the task was**

**deleted since the structure was populated! \*/**

**const signed char \*pcTaskName;**

**/\* A number unique to the task. \*/**

**UBaseType\_t xTaskNumber;**

**/\* The state in which the task existed when the structure was populated. \*/**

[**eTaskState**](https://www.freertos.org/a00021.html#eTaskGetState) **eCurrentState;**

**/\* The priority at which the task was running (may be inherited) when the**

**structure was populated. \*/**

**UBaseType\_t uxCurrentPriority;**

**/\* The priority to which the task will return if the task's current priority**

**has been inherited to avoid unbounded priority inversion when obtaining a**

**mutex. Only valid if configUSE\_MUTEXES is defined as 1 in**

**FreeRTOSConfig.h. \*/**

**UBaseType\_t uxBasePriority;**

**/\* The total run time allocated to the task so far, as defined by the [run](https://www.freertos.org/rtos-run-time-stats.html)**

**[time stats clock](https://www.freertos.org/rtos-run-time-stats.html). Only valid when configGENERATE\_RUN\_TIME\_STATS is**

**defined as 1 in FreeRTOSConfig.h. \*/**

**unsigned long ulRunTimeCounter;**

**/\* Points to the lowest address of the task's stack area. \*/**

**StackType\_t \*pxStackBase;**

**/\* The minimum amount of stack space that has remained for the task since**

**the task was created. The closer this value is to zero the closer the task**

**has come to overflowing its stack. \*/**

[**configSTACK\_DEPTH\_TYPE**](https://www.freertos.org/a00110.html#configSTACK_DEPTH_TYPE) **usStackHighWaterMark;**

**} TaskStatus\_t;**

# vTaskGetInfo() [[Task Utilities](https://www.freertos.org/a00021.html)]

task.h

**void vTaskGetInfo( TaskHandle\_t xTask,**

**TaskStatus\_t \*pxTaskStatus,**

**BaseType\_t xGetFreeStackSpace,**

**eTaskState eState );**

必须在FreeRTOSConfig.h中将configUSE\_TRACE\_FACILITY定义为1，vTaskGetInfo（）才可用。

[uxTaskGetSystemState](https://www.freertos.org/uxTaskGetSystemState.html)（） 为系统中的每个任务填充一个[TaskStatus\_t](https://www.freertos.org/vTaskGetInfo.html" \l "TaskStatus_t)结构， 而[vTaskGetInfo](https://www.freertos.org/uxTaskGetSystemState.html)（）为一个任务填充一个TaskStatus\_t结构。TaskStatus\_t结构除其他外，包含任务句柄的成员，任务名称，任务优先级，任务状态以及任务消耗的运行时间总量。

注意：此功能仅用于调试使用，因为其使用会导致调度程序长时间处于挂起状态。

**参数：**

|  |  |
| --- | --- |
| 任务 | 正在查询任务的句柄。将xTask设置为NULL将返回有关调用任务的信息。 |
| pxTaskStatus | pxTaskStatus指向的TaskStatus\_t结构将填充有关xTask参数中传递的句柄引用的任务的信息。 |
| xGetFreeStackSpace | TaskStatus\_t结构包含一个成员，用于报告要查询的任务的堆栈高水位线。堆栈高水位标记是已存在的最小堆栈空间量，因此数字越接近零，任务就越接近其堆栈溢出。计算堆栈高水位标记需要相对较长的时间，并且可以使系统暂时无响应-因此提供了xGetFreeStackSpace参数，以跳过高水位线检查。如果xGetFreeStackSpace未设置为pdFALSE，则仅将高水印值写入TaskStatus\_t结构。 |
| 房地产 | TaskStatus\_t结构包含一个成员，用于报告要查询的任务的状态。获取任务状态并不像简单的分配那样快-因此提供了eState参数以允许从TaskStatus\_t结构中省略状态信息。要获取状态信息，然后将eState设置为eInvalid-否则，在eState中传递的值将在TaskStatus\_t结构中报告为任务状态。 |

**用法示例：**

**void vAFunction( void )**

**{**

**TaskHandle\_t xHandle;**

**TaskStatus\_t xTaskDetails;**

**/\* Obtain the handle of a task from its name. \*/**

**xHandle =** [**xTaskGetHandle**](https://www.freertos.org/a00021.html#xTaskGetHandle)**( "Task\_Name" );**

**/\* Check the handle is not NULL. \*/**

**configASSERT( xHandle );**

**/\* Use the handle to obtain further information about the task. \*/**

**vTaskGetInfo( /\* The handle of the task being queried. \*/**

**xHandle,**

**/\* The TaskStatus\_t structure to complete with information**

**on xTask. \*/**

**&xTaskDetails,**

**/\* Include the stack high water mark value in the**

**TaskStatus\_t structure. \*/**

**pdTRUE,**

**/\* Include the task state in the TaskStatus\_t structure. \*/**

**eInvalid );**

**}**

### The TaskStatus\_t definition

**typedef struct xTASK\_STATUS**

**{**

**/\* The handle of the task to which the rest of the information in the**

**structure relates. \*/**

**TaskHandle\_t xHandle;**

**/\* A pointer to the task's name. This value will be invalid if the task was**

**deleted since the structure was populated! \*/**

**const signed char \*pcTaskName;**

**/\* A number unique to the task. \*/**

**UBaseType\_t xTaskNumber;**

**/\* The state in which the task existed when the structure was populated. \*/**

[**eTaskState**](https://www.freertos.org/a00021.html#eTaskGetState) **eCurrentState;**

**/\* The priority at which the task was running (may be inherited) when the**

**structure was populated. \*/**

**UBaseType\_t uxCurrentPriority;**

**/\* The priority to which the task will return if the task's current priority**

**has been inherited to avoid unbounded priority inversion when obtaining a**

**mutex. Only valid if configUSE\_MUTEXES is defined as 1 in**

**FreeRTOSConfig.h. \*/**

**UBaseType\_t uxBasePriority;**

**/\* The total run time allocated to the task so far, as defined by the [run](https://www.freertos.org/rtos-run-time-stats.html)**

**[time stats clock](https://www.freertos.org/rtos-run-time-stats.html). Only valid when configGENERATE\_RUN\_TIME\_STATS is**

**defined as 1 in FreeRTOSConfig.h. \*/**

**unsigned long ulRunTimeCounter;**

**/\* Points to the lowest address of the task's stack area. \*/**

**StackType\_t \*pxStackBase;**

**/\* The minimum amount of stack space that has remained for the task since**

**the task was created. The closer this value is to zero the closer the task**

**has come to overflowing its stack. \*/**

[**configSTACK\_DEPTH\_TYPE**](https://www.freertos.org/a00110.html#configSTACK_DEPTH_TYPE) **usStackHighWaterMark;**

**} TaskStatus\_t;**

# xTaskGetApplicationTaskTag xTaskGetApplicationTaskTagFromISR [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**TaskHookFunction\_t xTaskGetApplicationTaskTag( TaskHandle\_t xTask );**

**TaskHookFunction\_t xTaskGetApplicationTaskTagFromISR( TaskHandle\_t xTask );**

为了使这些功能可用，必须将configUSE\_APPLICATION\_TASK\_TAG定义为1。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

xTaskGetApplicationTaskTagTagFromISR（）是xTaskGetApplicationTaskTagTag（）的一个版本，可以从中断服务例程（ISR）进行调用。

返回与任务关联的“标签”值。标签值的含义和使用由应用程序编写者定义。RTOS内核本身通常不会访问标签值。

此功能仅适用于高级用户。

**参数：**

|  |  |
| --- | --- |
| 任务 | 正在查询任务的句柄。任务可以使用NULL作为参数值来查询自己的标记值。 |

**返回值：**

被查询任务的“标签”值。

**用法示例：**

**/\* In this example, an integer is set as the task tag value. \*/**

**void vATask( void \*pvParameters )**

**{**

**/\* Assign a tag value of 1 to the currently executing task.**

**The (void \*) cast is used to prevent compiler warnings. \*/**

**vTaskSetApplicationTaskTag( NULL, ( void \* ) 1 );**

**for( ;; )**

**{**

**/\* Rest of task code goes here. \*/**

**}**

**}**

**void vAFunction( void )**

**{**

**TaskHandle\_t xHandle;**

**int iReturnedTaskHandle;**

**/\* Create a task from the vATask() function, storing the handle to the**

**created task in the xTask variable. \*/**

**/\* Create the task. \*/**

**if( xTaskCreate(**

**vATask, /\* Pointer to the function that implements**

**the task. \*/**

**"Demo task", /\* Text name given to the task. \*/**

**STACK\_SIZE, /\* The size of the stack that should be created**

**for the task. This is defined in words, not**

**bytes. \*/**

**NULL, /\* The task does not use the**

**parameter. \*/**

**TASK\_PRIORITY, /\* The priority to assign to the newly created**

**task. \*/**

**&xHandle /\* The handle to the task being created will be**

**placed in xHandle. \*/**

**) == pdPASS )**

**{**

**/\* The task was created successfully. Delay for a short period to allow**

**the task to run. \*/**

**vTaskDelay( 100 );**

**/\* What tag value is assigned to the task? The returned tag value is**

**stored in an integer, so cast to an integer to prevent compiler**

**warnings. \*/**

**iReturnedTaskHandle = ( int ) xTaskGetApplicationTaskTag( xHandle );**

**}**

**}**

# uxTaskGetStackHighWaterMark uxTaskGetStackHighWaterMark2 [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**UBaseType\_t uxTaskGetStackHighWaterMark**

**( TaskHandle\_t xTask );**

[**configSTACK\_DEPTH\_TYPE**](https://www.freertos.org/a00110.html#configSTACK_DEPTH_TYPE) **uxTaskGetStackHighWaterMark2**

**( TaskHandle\_t xTask );**

要使这些功能可用，必须将INCLUDE\_uxTaskGetStackHighWaterMark定义为1。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

uxTaskGetStackHighWaterMark2（）是uxTaskGetStackHighWaterMark（）的版本，该版本返回用户可定义的类型以消除8位架构上UBaseType\_t类型的数据类型宽度限制。

任务使用的堆栈将随着任务执行和中断处理而增长和收缩。uxTaskGetStackHighWaterMark（）返回自任务开始执行以来任务可使用的最小剩余堆栈空间量，即任务堆栈处于最大（最深）值时未使用的堆栈量。这就是所谓的堆栈“高水位标记”。

**参数：**

|  |  |
| --- | --- |
| 任务 | 正在查询任务的句柄。任务可以通过传递NULL作为xTask参数来查询自己的高水位线。 |

**返回值：**

|  |
| --- |
| 返回的值是以字为单位的高水位标记（例如，在32位计算机上，返回值为1表示未使用4个字节的堆栈）。如果返回值为零，则任务可能已溢出其堆栈。如果返回值接近零，则任务已接近溢出其堆栈。 |

**用法示例：**

**void vTask1( void \* pvParameters )**

**{**

**UBaseType\_t uxHighWaterMark;**

**/\* Inspect our own high water mark on entering the task. \*/**

**uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );**

**for( ;; )**

**{**

**/\* Call any function. \*/**

**vTaskDelay( 1000 );**

**/\* Calling the function will have used some stack space, we would**

**therefore now expect uxTaskGetStackHighWaterMark() to return a**

**value lower than when it was called on entering the task. \*/**

**uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );**

**}**

**}**

# vTaskSetApplicationTaskTag [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**void vTaskSetApplicationTaskTag(**

**TaskHandle\_t xTask,**

**TaskHookFunction\_t pxTagValue );**

必须将configUSE\_APPLICATION\_TASK\_TAG定义为1，此功能才能使用。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

可以为每个任务分配一个“标签”值。该值仅用于应用程序– RTOS内核本身不以任何方式使用它。该[RTOS跟踪宏](https://www.freertos.org/rtos-trace-macros.html)文档页面提供了一个应用程序可能如何利用此功能的一个很好的例子。

**参数：**

|  |  |
| --- | --- |
| 任务 | 为其分配了标签值的任务的句柄。将xTask传递为NULL会将标记分配给调用任务。 |
| pxTagValue | 分配给任务标签的值。这是TaskHookFunction\_t类型，允许将功能指针分配为标签，尽管实际上可以分配任何值。请参见下面的示例。 |

**用法示例：**

**/\* In this example an integer is set as the task tag value.**

**See the RTOS trace hook macros documentation page for an**

**example how such an assignment can be used. \*/**

**void vATask( void \*pvParameters )**

**{**

**/\* Assign a tag value of 1 to myself. \*/**

**vTaskSetApplicationTaskTag( NULL, ( void \* ) 1 );**

**for( ;; )**

**{**

**/\* Rest of task code goes here. \*/**

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\* In this example a callback function is being assigned as the task tag.**

**First define the callback function - this must have type TaskHookFunction\_t**

**as per this example. \*/**

**static BaseType\_t prvExampleTaskHook( void \* pvParameter )**

**{**

**/\* Perform some action - this could be anything from logging a value,**

**updating the task state, outputting a value, etc. \*/**

**return 0;**

**}**

**/\* Now define the task that sets prvExampleTaskHook as its hook/tag value.**

**This is in fact registering the task callback, as described on the**

**xTaskCallApplicationTaskHook() documentation page. \*/**

**void vAnotherTask( void \*pvParameters )**

**{**

**/\* Register our callback function. \*/**

**vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );**

**for( ;; )**

**{**

**/\* Rest of task code goes here. \*/**

**}**

**}**

**/\* As an example use of the hook (callback) we can get the RTOS kernel to call the**

**hook function of each task that is being switched out during a reschedule. \*/**

**#define traceTASK\_SWITCHED\_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB,**

**0 )**

# xTaskCallApplicationTaskHook [[Task Control](https://www.freertos.org/a00112.html)]

task. h

**BaseType\_t xTaskCallApplicationTaskHook(**

**TaskHandle\_t xTask,**

**void \*pvParameter );**

必须将configUSE\_APPLICATION\_TASK\_TAG定义为1，此功能才能使用。有关更多信息，请参见[RTOS配置](https://www.freertos.org/a00110.html)文档。

可以为每个任务分配一个“标签”值。通常，此值仅供应用程序使用，RTOS内核不会访问它。但是，可以使用标记将钩子（或回调）函数分配给任务-通过调用xTaskCallApplicationTaskHook（）执行钩子函数。每个任务可以定义自己的回调，也可以根本不定义回调。

尽管可以使用第一个函数参数来调用任何任务的钩子函数，但是任务钩子函数最常见的用法是使用跟踪钩子宏，如以下示例所示。

任务挂钩函数必须具有TaskHookFunction\_t类型，即带有void \*参数，并返回BaseType\_t类型的值。void \*参数可用于将任何信息传递给hook函数。

**参数：**

|  |  |
| --- | --- |
| 任务 | 正在调用其挂钩函数的任务的句柄。将NULL作为xTask传递将调用与当前正在执行的任务相关联的hook函数。 |
| pvParameter | 传递给挂钩函数的值。这可以是指向结构的指针，也可以只是数字值。 |

**用法示例：**

**/\* In this example a callback function is being assigned as the task tag.**

**First define the callback function - this must have type TaskHookFunction\_t**

**as per this example. \*/**

**static BaseType\_t prvExampleTaskHook( void \* pvParameter )**

**{**

**/\* Perform some action - this could be anything from logging a value,**

**updating the task state, outputting a value, etc. \*/**

**return 0;**

**}**

**/\* Now define the task that sets prvExampleTaskHook as its hook/tag value.**

**This is in fact registering the task callback, as described on the**

**xTaskCallApplicationTaskHook() documentation page. \*/**

**void vAnotherTask( void \*pvParameters )**

**{**

**/\* Register our callback function. \*/**

**vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );**

**for( ;; )**

**{**

**/\* Rest of task code goes here. \*/**

**}**

**}**

**/\* As an example use of the hook (callback) we can get the RTOS kernel to**

**call the hook function of each task that is being switched out during a**

**reschedule. \*/**

**#define traceTASK\_SWITCHED\_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )**

## vTaskSetThreadLocalStoragePointer [[Task Utilities](https://www.freertos.org/a00021.html)]

task. h

**void vTaskSetThreadLocalStoragePointer( TaskHandle\_t xTaskToSet,**

**BaseType\_t xIndex,**

**void \*pvValue )**

在任务的[线程本地存储数组中](https://www.freertos.org/thread-local-storage-pointers.html)设置一个值 。

此功能仅适用于高级用户。

**参数：**

|  |  |
| --- | --- |
| xTaskToSet | 线程本地数据正在写入的任务的句柄。通过使用NULL作为参数值，任务可以写入其自己的线程本地数据。 |
| 索引 | 写入数据的线程本地存储阵列的索引。  可用数组索引的数量由[FreeRTOSConfig.h中](https://www.freertos.org/a00110.html)的[configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS](https://www.freertos.org/a00110.html#configNUM_THREAD_LOCAL_STORAGE_POINTERS) 编译时配置常量 设置 。 |
| pv值 | 要写入由xIndex参数指定的索引的值。 |

**用法示例：**

请参阅[线程本地存储阵列](https://www.freertos.org/thread-local-storage-pointers.html) 文档页面上提供的示例。

## pvTaskGetThreadLocalStoragePointer [[Task Utilities](https://www.freertos.org/a00021.html)]

task. h

**void \*pvTaskGetThreadLocalStoragePointer(**

**TaskHandle\_t xTaskToQuery,**

**BaseType\_t xIndex );**

从任务的[线程本地存储数组中](https://www.freertos.org/thread-local-storage-pointers.html)检索一个值 。

此功能仅适用于高级用户。

**参数：**

|  |  |
| --- | --- |
| xTaskToQuery | 从中读取线程本地数据的任务的句柄。通过使用NULL作为参数值，任务可以读取自己的线程本地数据。 |
| 索引 | 到线程本地存储阵列中的索引，将从中读取数据。  可用数组索引的数量由[FreeRTOSConfig.h中](https://www.freertos.org/a00110.html)的[configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS](https://www.freertos.org/a00110.html#configNUM_THREAD_LOCAL_STORAGE_POINTERS) 编译时配置常量 设置 。 |

**返回值：**

存储在任务xTaskToQuery的线程本地存储数组的索引位置xIndex中的值。

**用法示例：**

请参阅[线程本地存储阵列](https://www.freertos.org/thread-local-storage-pointers.html) 文档页面上提供的示例。

# vTaskSetTimeOutState() [[Task Utilities](https://www.freertos.org/a00021.html)]

task.h

**void vTaskSetTimeOutState( TimeOut\_t \* const pxTimeOut );**

此功能仅适用于高级用户。

任务可以进入阻止状态以等待事件。通常，任务不会无限期地处于“阻塞”状态，而是将指定超时期限。如果超时时间在任务等待事件发生之前到期，则该任务将从“阻塞”状态中删除。

如果任务在等待事件发生时多次进入和退出“阻止”状态，则必须调整每次任务进入“阻止”状态所用的超时时间，以确保在“阻止”状态花费的所有时间总计不超过最初指定的超时期限。xTaskCheckForTimeOut（）进行调整时会考虑偶发计数溢出之类的偶发事件，否则会导致手动调整容易出错。

vTaskSetTimeOutState（）与xTaskCheckForTimeOut（）一起使用。调用vTaskSetTimeOutState（）设置初始条件，然后可以调用xTaskCheckForTimeOut（）来检查超时条件，并在未发生超时的情况下调整剩余的块时间。

**参数：**

|  |  |
| --- | --- |
| pxTimeOut | 指向结构的指针，该结构将被初始化以保存确定是否已发生超时所必需的信息。 |

**用法示例：**

[xTaskCheckForTimeOut（）](https://www.freertos.org/xTaskCheckForTimeOut.html) 文档页面 上提供了一个示例。

# xTaskCheckForTimeOut() [[Task Utilities](https://www.freertos.org/a00021.html)]

task.h

**BaseType\_t xTaskCheckForTimeOut( TimeOut\_t \* const pxTimeOut,**

**TickType\_t \* const pxTicksToWait );**

此功能仅适用于高级用户。

任务可以进入阻止状态以等待事件。通常，任务不会无限期地处于“阻塞”状态，而是将指定超时期限。如果超时时间在任务等待事件发生之前到期，则该任务将从“阻塞”状态中删除。

如果任务在等待事件发生时多次进入和退出“阻止”状态，则必须调整每次任务进入“阻止”状态所用的超时时间，以确保在“阻止”状态花费的所有时间总计不超过最初指定的超时期限。xTaskCheckForTimeOut（）进行调整时会考虑偶发计数溢出之类的偶发事件，否则会导致手动调整容易出错。

xTaskCheckForTimeOut（）与vTaskSetTimeOutState（）一起使用。调用vTaskSetTimeOutState（）设置初始条件，然后可以调用xTaskCheckForTimeOut（）来检查超时条件，并在未发生超时的情况下调整剩余的块时间。

**参数：**

|  |  |
| --- | --- |
| *pxTimeOut* | 指向结构的指针，该结构保存确定是否发生超时所必需的信息。使用vTaskSetTimeOutState（）初始化pxTimeOut。 |
| *pxTicksToWait* | 用于传递调整后的阻止时间，这是考虑到已处于“阻止”状态的时间后剩余的阻止时间。 |

**返回值：**

如果返回pdTRUE，则没有剩余的块时间，并且发生了超时。

如果返回pdFALSE，则剩余一些块时间，因此不会发生超时。

**用法示例：**

**/\* Driver library function used to receive uxWantedBytes from an Rx buffer that**

**is filled by a UART interrupt. If there are not enough bytes in the Rx buffer**

**then the task enters the Blocked state until it is notified that more data has**

**been placed into the buffer. If there is still not enough data then the task**

**re-enters the Blocked state, and xTaskCheckForTimeOut() is used to re-calculate**

**the Block time to ensure the total amount of time spent in the Blocked state does**

**not exceed MAX\_TIME\_TO\_WAIT. This continues until either the buffer contains at**

**least uxWantedBytes bytes, or the total amount of time spent in the Blocked state**

**reaches MAX\_TIME\_TO\_WAIT – at which point the task reads however many bytes are**

**available up to a maximum of uxWantedBytes. \*/**

**size\_t xUART\_Receive( uint8\_t \*pucBuffer, size\_t uxWantedBytes )**

**{**

**size\_t uxReceived = 0;**

**TickType\_t xTicksToWait = MAX\_TIME\_TO\_WAIT;**

**TimeOut\_t xTimeOut;**

**/\* Initialize xTimeOut. This records the time at which this function was**

**entered. \*/**

**vTaskSetTimeOutState( &xTimeOut );**

**/\* Loop until the buffer contains the wanted number of bytes, or a timeout**

**occurs. \*/**

**while( UART\_bytes\_in\_rx\_buffer( pxUARTInstance ) < uxWantedBytes )**

**{**

**/\* The buffer didn't contain enough data so this task is going to**

**enter the Blocked state. Adjusting xTicksToWait to account for any time**

**that has been spent in the Blocked state within this function so far to**

**ensure the total amount of time spent in the Blocked state does not exceed**

**MAX\_TIME\_TO\_WAIT. \*/**

**if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )**

**{**

**/\* Timed out before the wanted number of bytes were available, exit the**

**loop. \*/**

**break;**

**}**

**/\* Wait for a maximum of xTicksToWait ticks to be notified that the receive**

**interrupt has placed more data into the buffer. \*/**

**ulTaskNotifyTake( pdTRUE, xTicksToWait );**

**}**

**/\* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The**

**actual number of bytes read (which might be less than uxWantedBytes) is**

**returned. \*/**

**uxReceived = UART\_read\_from\_receive\_buffer( pxUARTInstance,**

**pucBuffer,**

**uxWantedBytes );**

**return uxReceived;**

**}**

# FreeRTOS-MPU 特定功能 [[API](https://www.freertos.org/a00106.html)]

## 模组

* [**xTaskCreateRestricted()**](https://www.freertos.org/xTaskCreateRestricted.html)
* [**vTaskAllocateMPURegions()**](https://www.freertos.org/vTaskAllocateMPURegions.html)
* [**portSWITCH\_TO\_USER\_MODE()**](https://www.freertos.org/portSWITCH_TO_USER_MODE.html)

# xTaskCreateRestricted [[FreeRTOS-MPU Specific](https://www.freertos.org/FreeRTOS-MPU-specific.html)]

task. h

**BaseType\_t xTaskCreateRestricted(**

**TaskParameters\_t \*pxTaskDefinition,**

**TaskHandle\_t \*pxCreatedTask );**

创建一个新的内存保护单元（MPU）受限制的任务，并将其添加到准备运行的任务列表中。

xTaskCreateRestricted（）旨在与 [FreeRTOS-MPU一起使用](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html)，其 [演示应用程序](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#FreeRTOS-MPU-Demos)包含正在使用的xTaskCreateRestricted（）的全面且已记录的示例。

**参数：**

|  |  |
| --- | --- |
| pxTaskDefinition | 指向定义任务的结构的指针。该结构在本页上进行描述。 |
| pxCreatedTask | 用于传回可用来引用创建的任务的句柄。 |

**返回值：**

如果成功创建任务并将其添加到就绪列表，则为pdPASS，否则，在文件projdefs.h中定义的错误代码

包含MPU支持的任务比不包含的任务需要更多的参数来创建。将每个参数分别传递给xTaskCreateRestricted（）会很麻烦，因此使用TaskParameters\_t结构允许在编译时静态配置参数。该结构在task.h中定义为：

**typedef struct xTASK\_PARAMETERS**

**{**

**TaskFunction\_t pvTaskCode;**

**const signed char \* const pcName;**

**unsigned short usStackDepth;**

**void \*pvParameters;**

**UBaseType\_t uxPriority;**

**portSTACK\_TYPE \*puxStackBuffer;**

**MemoryRegion\_t xRegions[ portNUM\_CONFIGURABLE\_REGIONS ];**

**} TaskParameters\_t;**

**....where MemoryRegion\_t is defined as:**

**typedef struct xMEMORY\_REGION**

**{**

**void \*pvBaseAddress;**

**unsigned long ulLengthInBytes;**

**unsigned long ulParameters;**

**} MemoryRegion\_t;**

以下是每个结构成员的描述：

* pvTaskCode到uxPriority

这些成员与具有相同名称的[xTaskCreate（）](https://www.freertos.org/a00125.html)的参数完全相同。特别是，uxPriority用于设置任务的优先级和任务执行的模式。例如，要创建优先级2的用户模式任务，只需将uxPriority设置为2，创建优先级2的特权模式任务，将uxPriority设置为（2 | portPRIVILEGE\_BIT）。

* puxStackBuffer

每次在MPU中切换任务时，都会动态地重新配置它以定义一个区域，该区域为任务提供对其自身堆栈的读写访问。MPU区域必须满足许多约束条件-特别是每个区域的大小和对齐方式都必须等于两个值的相同幂。

每次创建任务时，标准FreeRTOS端口都使用pvPortMalloc（）分配新的堆栈。提供一个可以满足MPU数据对齐要求的pvPortMalloc（）实现是可能的，但它的RAM使用率也很复杂且效率很低。为了消除这种复杂性，FreeRTOS-MPU允许在编译时静态声明堆栈。这允许使用编译器扩展来管理对齐方式，并由链接程序来管理RAM使用效率。例如，如果使用GCC，则可以使用以下代码声明堆栈并正确对齐：

**char cTaskStack [1024] \_\_attribute \_\_（（align（1024））;**

通常将puxStackBuffer设置为静态声明的堆栈的地址。作为替代方案，可以将puxStackBuffer设置为NULL –在这种情况下，将调用pvPortMallocAligned（）分配任务堆栈，应用程序编写者有责任提供满足MPU对齐要求的pvPortMallocAligned（）实现。

* xMemoryRegions

xRegions是MemoryRegion\_t结构的数组，每个结构都定义单个用户可定义的内存区域，供正在创建的任务使用。ARM Cortex-M3 FreeRTOS-MPU端口将端口NUM\_CONFIGURABLE\_REGIONS定义为3。

pvBaseAddress和ulLengthInBytes成员分别作为存储区的开始和存储区的长度是不言自明的。ulParameters定义如何允许任务访问内存区域，并可以采用以下值的按位或：

**portMPU\_REGION\_READ\_WRITE**

**portMPU\_REGION\_PRIVILEGED\_READ\_ONLY**

**portMPU\_REGION\_READ\_ONLY**

**portMPU\_REGION\_PRIVILEGED\_READ\_WRITE**

**portMPU\_REGION\_CACHEABLE\_BUFFERABLE**

**portMPU\_REGION\_EXECUTE\_NEVER**

用法示例（请参考FreeRTOS-MPU[演示应用程序](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#FreeRTOS-MPU-Demos) 以获得更完整和全面的示例）：

**/\* Declare the stack that will be used by the task. The stack alignment must**

**match its size and be a power of 2, so if 128 words are reserved for the stack**

**then it must be aligned to ( 128 \* 4 ) bytes. This example used GCC syntax. \*/**

**static portSTACK\_TYPE xTaskStack[ 128 ] \_\_attribute\_\_((aligned(128\*4)));**

**/\* Declare an array that will be accessed by the task. The task should only**

**be able to read from the array, and not write to it. \*/**

**char cReadOnlyArray[ 512 ] \_\_attribute\_\_((aligned(512)));**

**/\* Fill in a TaskParameters\_t structure to define the task - this is the**

**structure passed to the xTaskCreateRestricted() function. \*/**

**static const TaskParameters\_t xTaskDefinition =**

**{**

**vTaskFunction, /\* pvTaskCode \*/**

**"A task", /\* pcName \*/**

**128, /\* usStackDepth - defined in words, not bytes. \*/**

**NULL, /\* pvParameters \*/**

**1, /\* uxPriority - priority 1, start in User mode. \*/**

**xTaskStack, /\* puxStackBuffer - the array to use as the task stack. \*/**

**/\* xRegions - In this case only one of the three user definable regions is**

**actually used. The parameters are used to set the region to read only. \*/**

**{**

**/\* Base address Length Parameters \*/**

**{ cReadOnlyArray, mainREAD\_ONLY\_ALIGN\_SIZE, portMPU\_REGION\_READ\_ONLY },**

**{ 0, 0, 0 },**

**{ 0, 0, 0 },**

**}**

**};**

**void main( void )**

**{**

**/\* Create the task defined by xTaskDefinition. NULL is used as the second**

**parameter as a task handle is not required. \*/**

**xTaskCreateRestricted( &xTaskDefinition, NULL );**

**/\* Start the RTOS scheduler. \*/**

**vTaskStartScheduler();**

**/\* Should not reach here! \*/**

**}**

# vTaskAllocateMPURegions [[FreeRTOS-MPU Specific](https://www.freertos.org/FreeRTOS-MPU-specific.html)]

task. h

**void vTaskAllocateMPURegions(**

**TaskHandle\_t xTaskToModify,**

**const MemoryRegion\_t \* const xRegions );**

使用对xTaskCreateRestricted（）的调用创建任务时，将内存区域分配给受限任务。然后可以在运行时使用vTaskAllocateMPURegions（）修改或重新定义区域。

vTaskAllocateMPURegions（）旨在与 [FreeRTOS-MPU一起使用](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html)，其 [演示应用程序](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#FreeRTOS-MPU-Demos)包含正在使用的vTaskAllocateMPURegions（）的示例。

**参数：**

|  |  |
| --- | --- |
| xTask | 正在更新的任务的句柄。 |
| x*Regions* | 指向MemoryRegion\_t结构数组的指针，每个结构都包含一个新的内存区域定义。应该使用常数portNUM\_CONFIGURABLE\_REGIONS来确定阵列的尺寸，在ARM Cortex-M3上将其设置为3。 |

MemoryRegion\_t在task.h中定义为：

**typedef struct xMEMORY\_REGION**

**{**

**void \*pvBaseAddress;**

**unsigned long ulLengthInBytes;**

**unsigned long ulParameters;**

**} MemoryRegion\_t;**

pvBaseAddress和ulLengthInBytes成员分别作为存储区的开始和存储区的长度是不言自明的。重要的是要注意，MPU区域必须满足许多约束条件-特别是每个区域的大小和对齐方式都必须等于两个值的相同幂。

ulParameters定义如何允许任务访问内存区域，并可以采用以下值的按位或：

**portMPU\_REGION\_READ\_WRITE**

**portMPU\_REGION\_PRIVILEGED\_READ\_ONLY**

**portMPU\_REGION\_READ\_ONLY**

**portMPU\_REGION\_PRIVILEGED\_READ\_WRITE**

**portMPU\_REGION\_CACHEABLE\_BUFFERABLE**

**portMPU\_REGION\_EXECUTE\_NEVER**

用法示例（请参考FreeRTOS-MPU[演示应用程序](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#FreeRTOS-MPU-Demos) 以获得更完整和全面的示例）：

**/\* Define an array that the task will both read from and write to. Make sure**

**the size and alignment are appropriate for an MPU region (note this uses GCC**

**syntax). \*/**

**static unsigned char ucOneKByte[ 1024 ] \_\_attribute\_\_((align( 1024 )));**

**/\* Define an array of MemoryRegion\_t structures that configures an MPU region**

**allowing read/write access for 1024 bytes starting at the beginning of the**

**ucOneKByte array. The other two of the maximum 3 definable regions are**

**unused so set to zero. \*/**

**static const MemoryRegion\_t xAltRegions[ portNUM\_CONFIGURABLE\_REGIONS ] =**

**{**

**/\* Base address Length Parameters \*/**

**{ ucOneKByte, 1024, portMPU\_REGION\_READ\_WRITE },**

**{ 0, 0, 0 },**

**{ 0, 0, 0 }**

**};**

**void vATask( void \*pvParameters )**

**{**

**/\* This task was created such that it has access to certain regions of**

**memory as defined by the MPU configuration. At some point it is**

**desired that these MPU regions are replaced with that defined in the**

**xAltRegions const struct above. Use a call to vTaskAllocateMPURegions()**

**for this purpose. NULL is used as the task handle to indicate that this**

**function should modify the MPU regions of the calling task. \*/**

**vTaskAllocateMPURegions( NULL, xAltRegions );**

**/\* Now the task can continue its function, but from this point on can only**

**access its stack and the ucOneKByte array (unless any other statically**

**defined or shared regions have been declared elsewhere). \*/**

**}**

# portSWITCH\_TO\_USER\_MODE [[FreeRTOS-MPU Specific](https://www.freertos.org/FreeRTOS-MPU-specific.html)]

task. h

**void portSWITCH\_TO\_USER\_MODE( void );**

将调用任务设置为用户模式。在用户模式下，任务无法返回到特权模式。

portSWITCH\_TO\_USER\_MODE（）旨在与 [FreeRTOS-MPU一起使用](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html)，其 [演示应用程序](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#FreeRTOS-MPU-Demos)包含正在使用的portSWITCH\_TO\_USER\_MODE（）的示例。

# 队列管理 [ [API](https://www.freertos.org/a00106.html) ]

## 模组

* [**xQueueCreate**](https://www.freertos.org/a00116.html)
* [**xQueueCreateStatic**](https://www.freertos.org/xQueueCreateStatic.html)
* [**vQueueDelete**](https://www.freertos.org/a00018.html#vQueueDelete)
* [**xQueueSend**](https://www.freertos.org/a00117.html)
* [**xQueueSendFromISR**](https://www.freertos.org/a00119.html)
* [**xQueueSendToBack**](https://www.freertos.org/xQueueSendToBack.html)
* [**xQueueSendToBackFromISR**](https://www.freertos.org/xQueueSendToBackFromISR.html)
* [**xQueueSendToFront**](https://www.freertos.org/xQueueSendToFront.html)
* [**xQueueSendToFrontFromISR**](https://www.freertos.org/xQueueSendToFrontFromISR.html)
* [**xQueueReceive**](https://www.freertos.org/a00118.html)
* [**xQueueReceiveFromISR**](https://www.freertos.org/a00120.html)
* [**uxQueueMessagesWaiting**](https://www.freertos.org/a00018.html#ucQueueMessagesWaiting)
* [**uxQueueMessagesWaitingFromISR**](https://www.freertos.org/a00018.html#ucQueueMessagesWaitingFromISR)
* [**uxQueueSpaces可用**](https://www.freertos.org/a00018.html#uxQueueSpacesAvailable)
* [**xQueueReset**](https://www.freertos.org/a00018.html#xQueueReset)
* [**xQueuePeek**](https://www.freertos.org/xQueuePeek.html)
* [**xQueuePeekFromISR**](https://www.freertos.org/xQueuePeekFromISR.html)
* [**vQueueAddToRegistry**](https://www.freertos.org/vQueueAddToRegistry.html)
* [**pcQueueGetName**](https://www.freertos.org/pcQueueGetName.html)
* [**vQueueUnregisterQueue**](https://www.freertos.org/vQueueUnregisterQueue.html)
* [**xQueueIsQueueEmptyFromISR**](https://www.freertos.org/a00018.html#xQueueIsQueueEmptyFromISR)
* [**xQueueIsQueueFullFromISR**](https://www.freertos.org/a00018.html#xQueueIsQueueFullFromISR)
* [**xQueueOverwrite**](https://www.freertos.org/xQueueOverwrite.html)
* [**xQueueOverwriteFromISR**](https://www.freertos.org/xQueueOverwriteFromISR.html)

## 详细说明

### uxQueueMessagesWaiting

队列

**UBaseType\_t uxQueueMessagesWaiting（QueueHandle\_t xQueue）;**

返回存储在队列中的消息数。

**参数：**

|  |  |
| --- | --- |
| xQueue | 正在查询的队列的句柄。 |

**返回值：**

队列中可用的消息数。

### uxQueueMessagesWaitingFromISR

队列

**UBaseType\_t uxQueueMessagesWaiting（QueueHandle\_t xQueue）;**

可以从ISR调用的uxQueueMessagesWaiting（）版本。返回存储在队列中的消息数。

**参数：**

|  |  |
| --- | --- |
| xQueue | 正在查询的队列的句柄。 |

**返回值：**

队列中可用的消息数。

### uxQueueSpacesAvailable

队列

**UBaseType\_t uxQueueSpacesAvailable（QueueHandle\_t xQueue）;**

返回队列中的可用空间数。

**参数：**

|  |  |
| --- | --- |
| xQueue | 正在查询的队列的句柄。 |

**返回值：**

队列中可用的可用空间数。

### vQueueDelete

队列

**void vQueueDelete（QueueHandle\_t xQueue）;**

删除队列–释放分配用于存储放置在队列中的项目的所有内存。

**参数：**

|  |  |
| --- | --- |
| xQueue | 要删除的队列的句柄。 |

### xQueueReset

队列

**BaseType\_t xQueueReset（QueueHandle\_t xQueue）;**

将队列重置为其原始的空状态。

**参数：**

|  |  |
| --- | --- |
| xQueue | 重置队列的句柄 |

**返回值：**

因为FreeRTOS V7.2.0 xQueueReset（）总是返回pdPASS。

### xQueueIsQueueEmptyFromISR

队列

**BaseType\_t xQueueIsQueueEmptyFromISR（const QueueHandle\_t xQueue）;**

查询队列以确定该队列是否为空。此功能只能在ISR中使用。

**参数：**

|  |  |
| --- | --- |
| xQueue | 正在查询的队列的句柄 |

**返回值：**

如果队列不为空，则为pdFALSE；如果队列为空，则为pdTRUE。

### xQueueIsQueueFullFromISR

队列

**BaseType\_t xQueueIsQueueFullFromISR（const QueueHandle\_t xQueue）;**

查询队列以确定队列是否已满。此功能只能在ISR中使用。

**参数：**

|  |  |
| --- | --- |
| xQueue | 正在查询的队列的句柄 |

**返回值：**

如果队列未满，则为pdFALSE；如果队列已满，则为pdTRUE。

# xQueueCreate

queue. h

**QueueHandle\_t xQueueCreate( UBaseType\_t uxQueueLength,**

**UBaseType\_t uxItemSize );**

创建一个新[队列](https://www.freertos.org/Embedded-RTOS-Queues.html)并返回一个句柄，通过该句柄可以引用该队列。 [要使](https://www.freertos.org/a00110.html#configSUPPORT_DYNAMIC_ALLOCATION) 此RTOS API功能可用，必须在FreeRTOSConfig.h[中将configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)设置为1，或将其保留为未定义状态（在这种情况下，它将默认为1）。

每个队列都需要RAM，该RAM用于保存队列状态以及保存队列（队列存储区域）中包含的项目。如果使用xQueueCreate（）创建队列，则从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用[xQueueCreateStatic（）](https://www.freertos.org/xQueueCreateStatic.html)创建队列 ，则应用程序[编写器](https://www.freertos.org/xQueueCreateStatic.html)将提供RAM，这会导致大量参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

**参数：**

|  |  |
| --- | --- |
| uxQueueLength | 队列一次可以容纳的最大项目数。 |
| uxItemSize | 存放队列中每个项目所需的大小（以字节为单位）。  项目按副本而不是引用排队，因此这是将为每个排队的项目复制的字节数。队列中的每个项目都必须具有相同的大小。 |

**返回值：**

如果队列创建成功，则返回创建队列的句柄。如果[无法分配](https://www.freertos.org/a00111.html)创建队列所需的内存，则返回NULL。

**用法示例：**

**struct AMessage**

**{**

**char ucMessageID;**

**char ucData[ 20 ];**

**};**

**void vATask( void \*pvParameters )**

**{**

**QueueHandle\_t xQueue1, xQueue2;**

**/\* Create a queue capable of containing 10 unsigned long values. \*/**

**xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );**

**if( xQueue1 == NULL )**

**{**

**/\* Queue was not created and must not be used. \*/**

**}**

**/\* Create a queue capable of containing 10 pointers to AMessage**

**structures. These are to be queued by pointers as they are**

**relatively large structures. \*/**

**xQueue2 = xQueueCreate( 10, sizeof( struct AMessage \* ) );**

**if( xQueue2 == NULL )**

**{**

**/\* Queue was not created and must not be used. \*/**

**}**

**/\* ... Rest of task code. \*/**

**}**

# xQueueCreateStatic

queue. h

**QueueHandle\_t xQueueCreateStatic(**

**UBaseType\_t uxQueueLength,**

**UBaseType\_t uxItemSize,**

**uint8\_t \*pucQueueStorageBuffer,**

**StaticQueue\_t \*pxQueueBuffer );**

创建一个新[队列](https://www.freertos.org/Embedded-RTOS-Queues.html)并返回一个句柄，通过该句柄可以引用该队列。 必须在FreeRTOSConfig.h[中将configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_STATIC_ALLOCATION)设置为1，此RTOS API函数才可用。

每个队列都需要RAM，该RAM用于保存队列状态以及保存队列（队列存储区域）中包含的项目。如果使用[xQueueCreate（）](https://www.freertos.org/a00116.html)创建队列，则会从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配此RAM 。如果使用xQueueCreateStatic（）创建队列，则应用程序编写器将提供RAM，这会导致大量参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

**参数：**

|  |  |
| --- | --- |
| uxQueueLength | 队列一次可以容纳的最大项目数。 |
| uxItemSize | 存放队列中每个项目所需的大小（以字节为单位）。  项目按副本而不是引用排队，因此这是将为每个排队的项目复制的字节数。队列中的每个项目都必须具有相同的大小。 |
| pucQueueStorageBuffer | 如果uxItemSize不为零，则pucQueueStorageBuffer必须指向一个uint8\_t数组，该数组至少足够大以容纳一次可以存在于队列中的最大项目数，即（uxQueueLength \* uxItemSize）字节。如果uxItemSize为零，则pucQueueStorageBuffer可以为NULL。 |
| pxQueueBuffer | 必须指向类型为StaticQueue\_t的变量，该变量将用于保存队列的数据结构。 |

**返回值：**

如果队列创建成功，则返回创建队列的句柄。如果pxQueueBuffer为NULL，则返回NULL。

**用法示例：**

**/\* The queue is to be created to hold a maximum of 10 uint64\_t**

**variables. \*/**

**#define QUEUE\_LENGTH 10**

**#define ITEM\_SIZE sizeof( uint64\_t )**

**/\* The variable used to hold the queue's data structure. \*/**

**static StaticQueue\_t xStaticQueue;**

**/\* The array to use as the queue's storage area. This must be at least**

**uxQueueLength \* uxItemSize bytes. \*/**

**uint8\_t ucQueueStorageArea[ QUEUE\_LENGTH \* ITEM\_SIZE ];**

**void vATask( void \*pvParameters )**

**{**

**QueueHandle\_t xQueue;**

**/\* Create a queue capable of containing 10 uint64\_t values. \*/**

**xQueue = xQueueCreateStatic( QUEUE\_LENGTH,**

**ITEM\_SIZE,**

**ucQueueStorageArea,**

**&xStaticQueue );**

**/\* pxQueueBuffer was not NULL so xQueue should not be NULL. \*/**

[**configASSERT**](https://www.freertos.org/a00110.html#configASSERT)**( xQueue );**

**}**

# xQueueSend [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueueSend(**

**QueueHandle\_t xQueue,**

**const void \* pvItemToQueue,**

**TickType\_t xTicksToWait**

**);**

这是一个调用xQueueGenericSend（）的宏。包含它是为了与不包含xQueueSendToFront（）和xQueueSendToBack（）宏的FreeRTOS版本向后兼容。它等效于xQueueSendToBack（）。

将项目发布到队列中。该项目按副本而不是参考排队。不得从中断服务程序中调用此函数。有关可以在ISR中使用的替代方法，请参见xQueueSendFromISR（）。

**参数：**

|  |  |
| --- | --- |
| xQueue | 要在其上发布项目的队列的句柄。 |
| pvItemToQueue | 指向要放在队列中的项目的指针。创建队列时已定义了队列将要容纳的项目的大小，因此，这许多字节将从pvItemToQueue复制到队列存储区域。 |
| xTicksToWait | 如果任务已满，则该任务应阻止等待队列上的可用空间的最长时间。如果队列已满并且xTicksToWait设置为0，则调用将立即返回。时间以滴答周期定义，因此如果需要，应使用常量portTICK\_PERIOD\_MS转换为实时。  如果将[INCLUDE\_vTaskSuspend](https://www.freertos.org/a00110.html)设置为“ 1”，则将阻止时间指定为portMAX\_DELAY将导致任务无限期地阻止（无超时）。 |

**返回：**如果项目已成功发布，则**返回**pdTRUE，否则返回errQUEUE\_FULL。

**用法示例：**

**struct AMessage**

**{**

**char ucMessageID;**

**char ucData[ 20 ];**

**} xMessage;**

**unsigned long ulVar = 10UL;**

**void vATask( void \*pvParameters )**

**{**

**QueueHandle\_t xQueue1, xQueue2;**

**struct AMessage \*pxMessage;**

**/\* Create a queue capable of containing 10 unsigned long values. \*/**

**xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );**

**/\* Create a queue capable of containing 10 pointers to AMessage structures.**

**These should be passed by pointer as they contain a lot of data. \*/**

**xQueue2 = xQueueCreate( 10, sizeof( struct AMessage \* ) );**

**/\* ... \*/**

**if( xQueue1 != 0 )**

**{**

**/\* Send an unsigned long. Wait for 10 ticks for space to become**

**available if necessary. \*/**

**if( xQueueSend( xQueue1,**

**( void \* ) &ulVar,**

**( TickType\_t ) 10 ) != pdPASS )**

**{**

**/\* Failed to post the message, even after 10 ticks. \*/**

**}**

**}**

**if( xQueue2 != 0 )**

**{**

**/\* Send a pointer to a struct AMessage object. Don't block if the**

**queue is already full. \*/**

**pxMessage = & xMessage;**

**xQueueSend( xQueue2, ( void \* ) &pxMessage, ( TickType\_t ) 0 );**

**}**

**/\* ... Rest of task code. \*/**

**}**

# xQueueSendFromISR [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueueSendFromISR**

**(**

**QueueHandle\_t xQueue,**

**const void \*pvItemToQueue,**

**BaseType\_t \*pxHigherPriorityTaskWoken**

**);**

这是一个调用xQueueGenericSendFromISR（）的宏。包含它是为了与不包含xQueueSendToBackFromISR（）和xQueueSendToFrontFromISR（）宏的FreeRTOS版本向后兼容。

将项目发布到队列的后面。在中断服务程序中可以安全地使用此功能。

项目通过复制而不是引用进行排队，因此最好仅将小项目排队，尤其是从ISR调用时。在大多数情况下，最好存储一个指向正在排队的项目的指针。

**参数：**

|  |  |
| --- | --- |
| *xQueue* | 要在其上发布项目的队列的句柄。 |
| *pvItemToQueue* | 指向要放在队列中的项目的指针。创建队列时已定义了队列将要容纳的项目的大小，因此，这许多字节将从pvItemToQueue复制到队列存储区域。 |
| *pxHigherPriorityTaskWoken* | 如果发送到队列导致任务取消阻止，并且未阻止的任务的优先级高于当前运行的任务，则xQueueSendFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。如果xQueueSendFromISR（）将此值设置为pdTRUE，则应在退出中断之前请求上下文切换。  从FreeRTOS V7.3.0开始，pxHigherPriorityTaskWoken是可选参数，可以设置为NULL。 |

**返回值：**

如果数据已成功发送到队列，则为pdTRUE，否则为errQUEUE\_FULL。

缓冲IO的示例用法（ISR每次调用可以获得多个值）：

**void vBufferISR( void )**

**{**

**char cIn;**

**BaseType\_t xHigherPriorityTaskWoken;**

**/\* We have not woken a task at the start of the ISR. \*/**

**xHigherPriorityTaskWoken = pdFALSE;**

**/\* Loop until the buffer is empty. \*/**

**do**

**{**

**/\* Obtain a byte from the buffer. \*/**

**cIn = portINPUT\_BYTE( RX\_REGISTER\_ADDRESS );**

**/\* Post the byte. \*/**

**xQueueSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );**

**} while( portINPUT\_BYTE( BUFFER\_COUNT ) );**

**/\* Now the buffer is empty we can switch context if necessary. \*/**

**if( xHigherPriorityTaskWoken )**

**{**

**/\* Actual macro used here is port specific. \*/**

**taskYIELD\_FROM\_ISR ();**

**}**

**}**

# xQueueSendToBack [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueueSendToBack(**

**QueueHandle\_t xQueue,**

**const void \* pvItemToQueue,**

**TickType\_t xTicksToWait**

**);**

这是一个调用xQueueGenericSend（）的宏。它等效于xQueueSend（）。

将项目发布到队列的后面。该项目按副本而不是参考排队。不得从中断服务程序中调用此函数。有关在ISR中可以使用的替代方法，请参见xQueueSendToBackFromISR（）。

**参数：**

|  |  |
| --- | --- |
| xQueue | 要在其上发布项目的队列的句柄。 |
| pvItemToQueue | 指向要放在队列中的项目的指针。创建队列时已定义了队列将要容纳的项目的大小，因此，这许多字节将从pvItemToQueue复制到队列存储区域。 |
| xTicksToWait | 如果任务已满，则该任务应阻止等待队列上的可用空间的最长时间。如果将其设置为0，则调用将立即返回。时间以滴答周期定义，因此如果需要，应使用常数portTICK\_PERIOD\_MS转换为实时。  如果将[INCLUDE\_vTaskSuspend](https://www.freertos.org/a00110.html)设置为“ 1”，则将阻止时间指定为portMAX\_DELAY将导致任务无限期地阻止（无超时）。 |

**返回值：**

如果项目已成功发布，则为pdTRUE，否则为errQUEUE\_FULL。

**用法示例：**

**struct AMessage**

**{**

**char ucMessageID;**

**char ucData[ 20 ];**

**} xMessage;**

**unsigned long ulVar = 10UL;**

**void vATask( void \*pvParameters )**

**{**

**QueueHandle\_t xQueue1, xQueue2;**

**struct AMessage \*pxMessage;**

**/\* Create a queue capable of containing 10 unsigned long values. \*/**

**xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );**

**/\* Create a queue capable of containing 10 pointers to AMessage**

**structures. These should be passed by pointer as they contain a lot of**

**data. \*/**

**xQueue2 = xQueueCreate( 10, sizeof( struct AMessage \* ) );**

**/\* ... \*/**

**if( xQueue1 != 0 )**

**{**

**/\* Send an unsigned long. Wait for 10 ticks for space to become**

**available if necessary. \*/**

**if( xQueueSendToBack( xQueue1,**

**( void \* ) &ulVar,**

**( TickType\_t ) 10 ) != pdPASS )**

**{**

**/\* Failed to post the message, even after 10 ticks. \*/**

**}**

**}**

**if( xQueue2 != 0 )**

**{**

**/\* Send a pointer to a struct AMessage object. Don't block if the**

**queue is already full. \*/**

**pxMessage = & xMessage;**

**xQueueSendToBack( xQueue2, ( void \* ) &pxMessage, ( TickType\_t ) 0 );**

**}**

**/\* ... Rest of task code. \*/**

**}**

# xQueueSendToBackFromISR [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueueSendToBackFromISR**

**(**

**QueueHandle\_t xQueue,**

**const void \*pvItemToQueue,**

**BaseType\_t \*pxHigherPriorityTaskWoken**

**);**

这是一个调用xQueueGenericSendFromISR（）的宏。

将项目发布到队列的后面。在中断服务程序中可以安全地使用此功能。

项目通过复制而不是引用进行排队，因此最好仅将小项目排队，尤其是从ISR调用时。

**参数：**

|  |  |
| --- | --- |
| *xQueue* | 要在其上发布项目的队列的句柄。 |
| *pvItemToQueue* | 指向要放在队列中的项目的指针。创建队列时已定义了队列将要容纳的项目的大小，因此，这许多字节将从pvItemToQueue复制到队列存储区域。 |
| *pxHigherPriorityTaskWoken* | 如果发送到队列导致任务取消阻止，并且未阻止的任务的优先级高于当前运行的任务，则xQueueSendTobackFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。如果xQueueSendToBackFromISR（）将此值设置为pdTRUE，则应在退出中断之前请求上下文切换。  从FreeRTOS V7.3.0开始，pxHigherPriorityTaskWoken是可选参数，可以设置为NULL。 |

**返回值：**

如果发送到队列成功，则为pdPASS，否则为errQUEUE\_FULL。

缓冲IO的示例用法（ISR每次调用可以获得多个值）：

**void vBufferISR( void )**

**{**

**char cIn;**

**BaseType\_t xHigherPriorityTaskWoken;**

**/\* We have not woken a task at the start of the ISR. \*/**

**xHigherPriorityTaskWoken = pdFALSE;**

**/\* Loop until the buffer is empty. \*/**

**do**

**{**

**/\* Obtain a byte from the buffer. \*/**

**cIn = portINPUT\_BYTE( RX\_REGISTER\_ADDRESS );**

**/\* Post the byte. \*/**

**xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );**

**} while( portINPUT\_BYTE( BUFFER\_COUNT ) );**

**/\* Now the buffer is empty we can switch context if necessary. \*/**

**if( xHigherPriorityTaskWoken )**

**{**

**/\* Actual macro used here is port specific. \*/**

**taskYIELD\_FROM\_ISR ();**

**}**

**}**

# xQueueSendToFront [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueueSendToFront( QueueHandle\_t xQueue,**

**const void \* pvItemToQueue,**

**TickType\_t xTicksToWait );**

这是一个调用xQueueGenericSend（）的宏。

将项目发布到队列的最前面。该项目按副本而不是参考排队。不得从中断服务程序中调用此函数。有关可以在ISR中使用的替代方法，请参见xQueueSendToFrontFromISR（）。

**参数：**

|  |  |
| --- | --- |
| xQueue | 要在其上发布项目的队列的句柄。 |
| pvItemToQueue | 指向要放在队列中的项目的指针。创建队列时已定义了队列将要容纳的项目的大小，因此，这许多字节将从pvItemToQueue复制到队列存储区域。 |
| xTicksToWait | 如果任务已满，则该任务应阻止等待队列上的可用空间的最长时间。如果将其设置为0，则调用将立即返回。时间以滴答周期定义，因此如果需要，应使用常数portTICK\_PERIOD\_MS转换为实时。  如果将[INCLUDE\_vTaskSuspend](https://www.freertos.org/a00110.html)设置为“ 1”，则将阻止时间指定为portMAX\_DELAY将导致任务无限期地阻止（无超时）。 |

**返回值：**

如果项目已成功发布，则为pdTRUE，否则为errQUEUE\_FULL。

**用法示例：**

**struct AMessage**

**{**

**char ucMessageID;**

**char ucData[ 20 ];**

**} xMessage;**

**unsigned long ulVar = 10UL;**

**void vATask( void \*pvParameters )**

**{**

**QueueHandle\_t xQueue1, xQueue2;**

**struct AMessage \*pxMessage;**

**/\* Create a queue capable of containing 10 unsigned long values. \*/**

**xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );**

**/\* Create a queue capable of containing 10 pointers to AMessage**

**structures. These should be passed by pointer as they contain a lot of**

**data. \*/**

**xQueue2 = xQueueCreate( 10, sizeof( struct AMessage \* ) );**

**/\* ... \*/**

**if( xQueue1 != 0 )**

**{**

**/\* Send an unsigned long. Wait for 10 ticks for space to become**

**available if necessary. \*/**

**if( xQueueSendToFront( xQueue1,**

**( void \* ) &ulVar,**

**( TickType\_t ) 10 ) != pdPASS )**

**{**

**/\* Failed to post the message, even after 10 ticks. \*/**

**}**

**}**

**if( xQueue2 != 0 )**

**{**

**/\* Send a pointer to a struct AMessage object. Don't block if the**

**queue is already full. \*/**

**pxMessage = & xMessage;**

**xQueueSendToFront( xQueue2, ( void \* ) &pxMessage, ( TickType\_t ) 0 );**

**}**

**/\* ... Rest of task code. \*/**

**}**

# xQueueSendToFrontFromISR [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueueSendToFrontFromISR**

**(**

**QueueHandle\_t xQueue,**

**const void \*pvItemToQueue,**

**BaseType\_t \*pxHigherPriorityTaskWoken**

**);**

这是一个调用xQueueGenericSendFromISR（）的宏。

将项目发布到队列的最前面。在中断服务程序中可以安全地使用此功能。

通过复制而不是引用将项目排队，因此最好只发送小项目，或者发送指向该项目的指针。

**参数：**

|  |  |
| --- | --- |
| *xQueue* | 要在其上发布项目的队列的句柄。 |
| *pvItemToQueue* | 指向要放在队列中的项目的指针。创建队列时已定义了队列将要容纳的项目的大小，因此，这许多字节将从pvItemToQueue复制到队列存储区域。 |
| *pxHigherPriorityTaskWoken* | 如果发送到队列导致任务取消阻止，并且未阻止的任务的优先级高于当前运行的任务，则xQueueSendToFrontFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。如果xQueueSendToFrontFromISR（）将此值设置为pdTRUE，则应在退出中断之前请求上下文切换。  从FreeRTOS V7.3.0开始，pxHigherPriorityTaskWoken是可选参数，可以设置为NULL。 |

**返回值：**

如果数据已成功发送到队列，则为pdPass，否则为errQUEUE\_FULL。

**用法示例：**

**void vBufferISR( void )**

**{**

**char cIn;**

**BaseType\_t xHigherPriorityTaskWoken;**

**/\* We have not woken a task at the start of the ISR. \*/**

**xHigherPriorityTaskWoken = pdFALSE;**

**/\* Obtain a byte from the buffer. \*/**

**cIn = portINPUT\_BYTE( RX\_REGISTER\_ADDRESS );**

**if( cIn == EMERGENCY\_MESSAGE )**

**{**

**/\* Post the byte to the front of the queue. \*/**

**xQueueSendToFrontFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );**

**}**

**else**

**{**

**/\* Post the byte to the back of the queue. \*/**

**xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );**

**}**

**/\* Did sending to the queue unblock a higher priority task? \*/**

**if( xHigherPriorityTaskWoken )**

**{**

**/\* Actual macro used here is port specific. \*/**

**taskYIELD\_FROM\_ISR ();**

**}**

**}**

# xQueueReceive [[Queue Management](https://www.freertos.org/a00018.html)]

queue. h

**BaseType\_t xQueueReceive(**

**QueueHandle\_t xQueue,**

**void \*pvBuffer,**

**TickType\_t xTicksToWait**

**);**

这是一个调用xQueueGenericReceive（）函数的宏。

从队列中接收项目。该项目以副本形式接收，因此必须提供足够大小的缓冲区。创建队列时定义了复制到缓冲区中的字节数。

不得在中断服务程序中使用此功能。另请参见xQueueReceiveFromISR。

**参数：**

|  |  |
| --- | --- |
| xQueue | 要从中接收项目的队列的句柄。 |
| pvBuffer | 指向将接收的项目复制到其中的缓冲区的指针。 |
| xTicksToWait | 如果在调用时队列为空，任务应等待等待接收项目的最长时间。如果队列为空，则将xTicksToWait设置为0将导致该函数立即返回。时间以滴答周期定义，因此如果需要，应使用常数portTICK\_PERIOD\_MS转换为实时。  如果将[INCLUDE\_vTaskSuspend](https://www.freertos.org/a00110.html)设置为“ 1”，则将阻止时间指定为portMAX\_DELAY将导致任务无限期地阻止（无超时）。 |

**返回值：**

如果从队列成功接收到项目，则为pdTRUE，否则为pdFALSE。

**用法示例：**

|  |
| --- |
| **/\* Define a variable of type struct AMMessage. The examples below demonstrate**  **how to pass the whole variable through the queue, and as the structure is**  **moderately large, also how to pass a reference to the variable through a queue. \*/**  **struct AMessage**  **{**  **char ucMessageID;**  **char ucData[ 20 ];**  **} xMessage;**  **/\* Queue used to send and receive complete struct AMessage structures. \*/**  **QueueHandle\_t xStructQueue = NULL;**  **/\* Queue used to send and receive pointers to struct AMessage structures. \*/**  **QueueHandle\_t xPointerQueue = NULL;**  **void vCreateQueues( void )**  **{**  **xMessage.ucMessageID = 0xab;**  **memset( &( xMessage.ucData ), 0x12, 20 );**  **/\* Create the queue used to send complete struct AMessage structures. This can**  **also be created after the schedule starts, but care must be task to ensure**  **nothing uses the queue until after it has been created. \*/**  **xStructQueue =** [**xQueueCreate**](https://www.freertos.org/a00116.html)**(**  **/\* The number of items the queue can hold. \*/**  **10,**  **/\* Size of each item is big enough to hold the**  **whole structure. \*/**  **sizeof( xMessage ) );**  **/\* Create the queue used to send pointers to struct AMessage structures. \*/**  **xPointerQueue = xQueueCreate(**  **/\* The number of items the queue can hold. \*/**  **10,**  **/\* Size of each item is big enough to hold only a**  **pointer. \*/**  **sizeof( &xMessage ) );**  **if( ( xStructQueue == NULL ) || ( xPointerQueue == NULL ) )**  **{**  **/\* One or more queues were not created successfully as there was not enough**  **heap memory available. Handle the error here. Queues can also be created**  **statically. \*/**  **}**  **}**  **/\* Task that writes to the queues. \*/**  **void vATask( void \*pvParameters )**  **{**  **struct AMessage \*pxPointerToxMessage;**  **/\* Send the entire structure to the queue created to hold 10 structures. \*/**  [**xQueueSend**](https://www.freertos.org/a00117.html)**( /\* The handle of the queue. \*/**  **xStructQueue,**  **/\* The address of the xMessage variable. sizeof( struct AMessage )**  **bytes are copied from here into the queue. \*/**  **( void \* ) &xMessage,**  **/\* Block time of 0 says don't block if the queue is already full.**  **Check the value returned by xQueueSend() to know if the message**  **was sent to the queue successfully. \*/**  **( TickType\_t ) 0 );**  **/\* Store the address of the xMessage variable in a pointer variable. \*/**  **pxPointerToxMessage = &xMessage;**  **/\* Send the address of xMessage to the queue created to hold 10 pointers. \*/**  **xQueueSend( /\* The handle of the queue. \*/**  **xPointerQueue,**  **/\* The address of the variable that holds the address of xMessage.**  **sizeof( &xMessage ) bytes are copied from here into the queue. As the**  **variable holds the address of xMessage it is the address of xMessage**  **that is copied into the queue. \*/**  **( void \* ) &pxPointerToxMessage,**  **( TickType\_t ) 0 );**  **/\* ... Rest of task code goes here. \*/**  **}**  **/\* Task that reads from the queues. \*/**  **void vADifferentTask( void \*pvParameters )**  **{**  **struct AMessage xRxedStructure, \*pxRxedPointer;**  **if( xStructQueue != NULL )**  **{**  **/\* Receive a message from the created queue to hold complex struct AMessage**  **structure. Block for 10 ticks if a message is not immediately available.**  **The value is read into a struct AMessage variable, so after calling**  **xQueueReceive() xRxedStructure will hold a copy of xMessage. \*/**  **if( xQueueReceive( xStructQueue,**  **&( xRxedStructure ),**  **( TickType\_t ) 10 ) == pdPASS )**  **{**  **/\* xRxedStructure now contains a copy of xMessage. \*/**  **}**  **}**  **if( xPointerQueue != NULL )**  **{**  **/\* Receive a message from the created queue to hold pointers. Block for 10**  **ticks if a message is not immediately available. The value is read into a**  **pointer variable, and as the value received is the address of the xMessage**  **variable, after this call pxRxedPointer will point to xMessage. \*/**  **if( xQueueReceive( xPointerQueue,**  **&( pxRxedPointer ),**  **( TickType\_t ) 10 ) == pdPASS )**  **{**  **/\* \*pxRxedPointer now points to xMessage. \*/**  **}**  **}**  **/\* ... Rest of task code goes here. \*/**  **}**    **A demonstration of how to send and receive structures and pointers to structures** |

# xQueueReceiveFromISR [[Queue Management](https://www.freertos.org/a00018.html)]

queue. h

**BaseType\_t xQueueReceiveFromISR**

**(**

**QueueHandle\_t xQueue,**

**void \*pvBuffer,**

**BaseType\_t \*pxHigherPriorityTaskWoken**

**);**

从队列中接收项目。在中断服务程序中可以安全地使用此功能。

**参数：**

|  |  |
| --- | --- |
| *xQueue* | 要从中接收项目的队列的句柄。 |
| *pvBuffer* | 指向将接收的项目复制到其中的缓冲区的指针。 |
| *pxHigherPriorityTaskWoken* | 可能会阻塞任务，等待队列上的空间可用。如果xQueueReceiveFromISR使此类任务解除阻止，则\* pxHigherPriorityTaskWoken将被设置为pdTRUE，否则\* pxHigherPriorityTaskWoken将保持不变。  从FreeRTOS V7.3.0开始，pxHigherPriorityTaskWoken是可选参数，可以设置为NULL。 |

**返回值：**

如果从队列成功接收到项目，则为pdTRUE，否则为pdFALSE。

**用法示例：**

**QueueHandle\_t xQueue;**

**/\* Function to create a queue and post some values. \*/**

**void vAFunction( void \*pvParameters )**

**{**

**char cValueToPost;**

**const TickType\_t xTicksToWait = ( TickType\_t )0xff;**

**/\* Create a queue capable of containing 10 characters. \*/**

**xQueue = xQueueCreate( 10, sizeof( char ) );**

**if( xQueue == 0 )**

**{**

**/\* Failed to create the queue. \*/**

**}**

**/\* ... \*/**

**/\* Post some characters that will be used within an ISR. If the queue**

**is full then this task will block for xTicksToWait ticks. \*/**

**cValueToPost = 'a';**

**xQueueSend( xQueue, ( void \* ) &cValueToPost, xTicksToWait );**

**cValueToPost = 'b';**

**xQueueSend( xQueue, ( void \* ) &cValueToPost, xTicksToWait );**

**/\* ... keep posting characters ... this task may block when the queue**

**becomes full. \*/**

**cValueToPost = 'c';**

**xQueueSend( xQueue, ( void \* ) &cValueToPost, xTicksToWait );**

**}**

**/\* ISR that outputs all the characters received on the queue. \*/**

**void vISR\_Routine( void )**

**{**

**BaseType\_t xTaskWokenByReceive = pdFALSE;**

**char cRxedChar;**

**while( xQueueReceiveFromISR( xQueue,**

**( void \* ) &cRxedChar,**

**&xTaskWokenByReceive) )**

**{**

**/\* A character was received. Output the character now. \*/**

**vOutputCharacter( cRxedChar );**

**/\* If removing the character from the queue woke the task that was**

**posting onto the queue xTaskWokenByReceive will have been set to**

**pdTRUE. No matter how many times this loop iterates only one**

**task will be woken. \*/**

**}**

**if( xTaskWokenByReceive != pdFALSE )**

**{**

**/\* We should switch context so the ISR returns to a different task.**

**NOTE: How this is done depends on the port you are using. Check**

**the documentation and examples for your port. \*/**

**taskYIELD ();**

**}**

**}**

# xQueueOverwrite [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueueOverwrite(**

**QueueHandle\_t xQueue,**

**const void \* pvItemToQueue**

**);**

这是一个调用xQueueGenericSend（）函数的宏。

[xQueueSendToBack（）的](https://www.freertos.org/xQueueSendToBack.html) 版本，即使队列已满，该版本也会写入队列，从而覆盖队列中已保存的数据。

xQueueOverwrite（）用于长度为1的队列，这意味着队列为空或已满。

不得从中断服务程序（ISR）调用此函数。有关可以在ISR中使用的替代方法，请参见xQueueOverwriteFromISR（）。

**参数：**

|  |  |
| --- | --- |
| xQueue | 要将数据发送到的队列的句柄。 |
| pvItemToQueue | 指向要放在队列中的项目的指针。[创建队列](https://www.freertos.org/a00116.html)时已定义了队列将容纳的项目大小 ，并且会将许多字节从pvItemToQueue复制到队列存储区域。 |

**返回值：**

xQueueOverwrite（）是调用xQueueGenericSend（）的宏，因此具有与xQueueSendToFront（）相同的返回值。但是，pdPASS是唯一可以返回的值，因为即使队列已满，xQueueOverwrite（）也会写入队列。

**用法示例：**

**void vFunction( void \*pvParameters )**

**{**

**QueueHandle\_t xQueue;**

**unsigned long ulVarToSend, ulValReceived;**

**/\* Create a queue to hold one unsigned long value. It is strongly**

**recommended \*not\* to use xQueueOverwrite() on queues that can**

**contain more than one value, and doing so will trigger an assertion**

**if configASSERT() is defined. \*/**

**xQueue = xQueueCreate( 1, sizeof( unsigned long ) );**

**/\* Write the value 10 to the queue using xQueueOverwrite(). \*/**

**ulVarToSend = 10;**

**xQueueOverwrite( xQueue, &ulVarToSend );**

**/\* Peeking the queue should now return 10, but leave the value 10 in**

**the queue. A block time of zero is used as it is known that the**

**queue holds a value. \*/**

**ulValReceived = 0;**

**xQueuePeek( xQueue, &ulValReceived, 0 );**

**if( ulValReceived != 10 )**

**{**

**/\* Error, unless another task removed the value. \*/**

**}**

**/\* The queue is still full. Use xQueueOverwrite() to overwrite the**

**value held in the queue with 100. \*/**

**ulVarToSend = 100;**

**xQueueOverwrite( xQueue, &ulVarToSend );**

**/\* This time read from the queue, leaving the queue empty once more.**

**A block time of 0 is used again. \*/**

**xQueueReceive( xQueue, &ulValReceived, 0 );**

**/\* The value read should be the last value written, even though the**

**queue was already full when the value was written. \*/**

**if( ulValReceived != 100 )**

**{**

**/\* Error unless another task is using the same queue. \*/**

**}**

**/\* ... \*/**

**}**

# xQueueOverwriteFromISR [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueueOverwrite**

**(**

**QueueHandle\_t xQueue,**

**const void \* pvItemToQueue**

**BaseType\_t \*pxHigherPriorityTaskWoken**

**);**

这是一个调用xQueueGenericSendFromISR（）函数的宏。

可以在ISR中使用 的[xQueueOverwrite（）](https://www.freertos.org/xQueueOverwrite.html)版本。xQueueOverwriteFromISR（）与xQueueSendToBackFromISR（）类似，但是即使队列已满，它也会写入队列，从而覆盖队列中已保存的数据。

xQueueOverwriteFromISR（）适用于长度为1的队列，这意味着该队列为空或已满。

**参数：**

|  |  |
| --- | --- |
| xQueue | 要将数据发送到的队列的句柄。 |
| pvItemToQueue | 指向要放在队列中的项目的指针。[创建队列](https://www.freertos.org/a00116.html)时已定义了队列将容纳的项目大小 ，并且会将许多字节从pvItemToQueue复制到队列存储区域。 |
| pxHigherPriorityTaskWoken | 如果发送到队列导致任务取消阻止，并且未阻止的任务的优先级高于当前运行的任务，则xQueueOverwriteFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。如果xQueueOverwriteFromISR（）将此值设置为pdTRUE，则应在退出中断之前请求上下文切换。有关正在使用的端口，请参考文档的“中断服务例程”部分，以了解如何完成此操作。 |

**返回值：**

xQueueOverwriteFromISR（）是一个调用xQueueGenericSendFromISR（）的宏，因此具有与xQueueSendToFrontFromISR（）相同的返回值。但是，pdPASS是唯一可以返回的值，因为即使队列已满，xQueueOverwriteFromISR（）也会写入队列。

**用法示例：**

**QueueHandle\_t xQueue;**

**void vFunction( void \*pvParameters )**

**{**

**/\* Create a queue to hold one unsigned long value. It is strongly**

**recommended not to use xQueueOverwriteFromISR() on queues that can**

**contain more than one value, and doing so will trigger an assertion**

**if configASSERT() is defined. \*/**

**xQueue = xQueueCreate( 1, sizeof( unsigned long ) );**

**}**

**void vAnInterruptHandler( void )**

**{**

**/\* xHigherPriorityTaskWoken must be set to pdFALSE before it is used. \*/**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**unsigned long ulVarToSend, ulValReceived;**

**/\* Write the value 10 to the queue using xQueueOverwriteFromISR(). \*/**

**ulVarToSend = 10;**

**xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );**

**/\* The queue is full, but calling xQueueOverwriteFromISR() again will still**

**pass because the value held in the queue will be overwritten with the**

**new value. \*/**

**ulVarToSend = 100;**

**xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );**

**/\* Reading from the queue will now return 100. \*/**

**/\* ... \*/**

**if( xHigherPrioritytaskWoken == pdTRUE )**

**{**

**/\* Writing to the queue caused a task to unblock and the unblocked task**

**has a priority higher than or equal to the priority of the currently**

**executing task (the task this interrupt interrupted). Perform a**

**context switch so this interrupt returns directly to the unblocked**

**task. \*/**

**portYIELD\_FROM\_ISR(); /\* or portEND\_SWITCHING\_ISR() depending on the**

**port.\*/**

**}**

**}**

# xQueuePeek [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueuePeek(**

**QueueHandle\_t xQueue,**

**void \*pvBuffer,**

**TickType\_t xTicksToWait**

**);**

这是一个调用xQueueGenericReceive（）函数的宏。

从队列中接收项目而不从队列中删除该项目。该项目以副本形式接收，因此必须提供足够大小的缓冲区。创建队列时定义了复制到缓冲区中的字节数。

成功接收的项目保留在队列中，因此将在下一次调用或对xQueueReceive（）的调用中再次返回。

不得在中断服务程序中使用此宏。

**参数：**

|  |  |
| --- | --- |
| xQueue | 要从中接收项目的队列的句柄。 |
| pvBuffer | 指向将接收的项目复制到其中的缓冲区的指针。它必须至少足够大以容纳创建队列时定义的队列项目的大小。 |
| xTicksToWait | 如果在调用时队列为空，任务应等待等待接收项目的最长时间。时间以滴答周期定义，因此如果需要，应使用常数portTICK\_PERIOD\_MS转换为实时。  如果将[INCLUDE\_vTaskSuspend](https://www.freertos.org/a00110.html)设置为“ 1”，则将阻止时间指定为portMAX\_DELAY将导致任务无限期地阻止（无超时）。 |

**返回值：**

如果已成功从队列中接收（偷看）某项，则为pdTRUE，否则为pdFALSE。

**用法示例：**

**struct AMessage**

**{**

**char ucMessageID;**

**char ucData[ 20 ];**

**} xMessage;**

**QueueHandle\_t xQueue;**

**// Task to create a queue and post a value.**

**void vATask( void \*pvParameters )**

**{**

**struct AMessage \*pxMessage;**

**// Create a queue capable of containing 10 pointers to AMessage structures.**

**// These should be passed by pointer as they contain a lot of data.**

**xQueue = xQueueCreate( 10, sizeof( struct AMessage \* ) );**

**if( xQueue == 0 )**

**{**

**// Failed to create the queue.**

**}**

**// ...**

**// Send a pointer to a struct AMessage object. Don't block if the**

**// queue is already full.**

**pxMessage = & xMessage;**

**xQueueSend( xQueue, ( void \* ) &pxMessage, ( TickType\_t ) 0 );**

**// ... Rest of task code.**

**}**

**// Task to peek the data from the queue.**

**void vADifferentTask( void \*pvParameters )**

**{**

**struct AMessage \*pxRxedMessage;**

**if( xQueue != 0 )**

**{**

**// Peek a message on the created queue. Block for 10 ticks if a**

**// message is not immediately available.**

**if( xQueuePeek( xQueue, &( pxRxedMessage ), ( TickType\_t ) 10 ) )**

**{**

**// pcRxedMessage now points to the struct AMessage variable posted**

**// by vATask, but the item still remains on the queue.**

**}**

**}**

**// ... Rest of task code.**

**}**

# xQueuePeekFromISR [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**BaseType\_t xQueuePeekFromISR(**

**QueueHandle\_t xQueue,**

**void \*pvBuffer,**

**);**

可以从中断服务例程（ISR）使用 的[xQueuePeek（）](https://www.freertos.org/xQueuePeek.html)版本。

从队列中接收项目而不从队列中删除该项目。该项目以副本形式接收，因此必须提供足够大小的缓冲区。创建队列时定义了复制到缓冲区中的字节数。

成功接收的项目保留在队列中，因此将在下一次调用或对任何队列接收功能的调用中再次返回。

**参数：**

|  |  |
| --- | --- |
| xQueue | 要从中接收项目的队列的句柄。 |
| pvBuffer | 指向将接收的项目复制到其中的缓冲区的指针。它必须至少足够大以容纳创建队列时定义的队列项目的大小。 |

**返回值：**

如果已成功从队列中接收（偷看）某项，则为pdTRUE，否则为pdFALSE。

# vQueueAddToRegistry [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**void vQueueAddToRegistry(**

**QueueHandle\_t xQueue,**

**char \*pcQueueName,**

**);**

为队列分配名称，并将队列添加到注册表。

**参数：**

|  |  |
| --- | --- |
| *xQueue* | 队列的句柄被添加到注册表中。 |
| *pcQueueName* | 要分配给队列的名称。这只是用于方便调试的文本字符串。队列注册表仅存储指向该字符串的指针，因此该字符串必须是持久性的（全局的，最好是在ROM / Flash中），而不是在堆栈上定义的。 |

队列注册表具有两个目的，这两个目的都与RTOS内核感知的调试相关联：

1. 它允许将文本名称与队列关联，以便在调试GUI中轻松识别队列。
2. 它包含调试器查找每个已注册队列和信号量所需的信息。

除非使用的是RTOS内核感知调试器，否则队列注册表没有任何用途。

configQUEUE\_REGISTRY\_SIZE定义可以注册的最大队列和信号量。仅需要注册要使用RTOS内核感知调试器查看的队列和信号量。

例：

**void vAFunction( void )**

**{**

**QueueHandle\_t xQueue;**

**/\* Create a queue big enough to hold 10 chars. \*/**

**xQueue = xQueueCreate( 10, sizeof( char ) );**

**/\* We want this queue to be viewable in a RTOS kernel aware debugger,**

**so register it. \*/**

**vQueueAddToRegistry( xQueue, "AMeaningfulName" );**

**}**

# vQueueUnregisterQueue [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**void vQueueUnregisterQueue(**

**QueueHandle\_t xQueue,**

**);**

从队列注册表中删除队列。

**参数：**

|  |  |
| --- | --- |
| *xQueue* | 从注册表中删除队列的句柄。 |

队列注册表具有两个目的，这两个目的都与RTOS内核感知的调试相关联：

1. 它允许将文本名称与队列关联，以便在调试GUI中轻松识别队列。
2. 它包含调试器查找每个已注册队列和信号量所需的信息。

除非使用的是RTOS内核感知调试器，否则队列注册表没有任何用途。

configQUEUE\_REGISTRY\_SIZE定义可以注册的最大队列和信号量。仅需要注册要使用RTOS内核感知调试器查看的队列和信号量。

例：

**void vAFunction( void )**

**{**

**QueueHandle\_t xQueue;**

**/\* Create a queue big enough to hold 10 chars. \*/**

**xQueue = xQueueCreate( 10, sizeof( char ) );**

**/\* We want this queue to be viewable in a RTOS kernel aware debugger,**

**so register it. \*/**

**vQueueAddToRegistry( xQueue, "AMeaningfulName" );**

**/\* The queue gets used here. \*/**

**/\* At some later time, the queue is going to be deleted, first**

**remove it from the registry. \*/**

**vQueueUnregisterQueue( xQueue );**

**vQueueDelete( xQueue );**

**}**

# pcQueueGetName [[Queue Management](https://www.freertos.org/a00018.html)]

queue.h

**const char \*pcQueueGetName( QueueHandle\_t xQueue )**

从队列的句柄中查找队列名称。

如果队列已添加到[队列注册表中，](https://www.freertos.org/vQueueAddToRegistry.html)则仅具有名称。

**参数：**

|  |  |
| --- | --- |
| xQueue | 正在查询的队列的句柄。 |

**返回值：**

如果xQueue引用的队列在队列注册表中，则返回队列的文本名称，否则返回NULL。

# 队列集 [ [API](https://www.freertos.org/a00106.html) ]

## FreeRTOS队列集API函数

* [**xQueueCreateSet**](https://www.freertos.org/xQueueCreateSet.html)
* [**xQueueAddToSet**](https://www.freertos.org/xQueueAddToSet.html)
* [**xQueueRemoveFromSet**](https://www.freertos.org/xQueueRemoveFromSet.html)
* [**xQueueSelectFromSet**](https://www.freertos.org/xQueueSelectFromSet.html)
* [**xQueueSelectFromSetFromISR**](https://www.freertos.org/xQueueSelectFromSetFromISR.html)

# xQueueCreateSet() [[Queue Set API](https://www.freertos.org/RTOS-queue-sets.html)]

queue.h

**QueueSetHandle\_t xQueueCreateSet**

**(**

**const UBaseType\_t uxEventQueueLength**

**);**

必须将FreeRTOSConfig.h中的configUSE\_QUEUE\_SETS设置为1，才能使用xQueueCreateSet（）API函数。

队列集提供了一种机制，允许RTOS任务同时阻止（挂起）来自多个RTOS队列或信号量的读取操作。请注意，还有使用队列集的更简单的选择。有关更多信息，请参见“ [阻止多个对象”](https://www.freertos.org/Pend-on-multiple-rtos-objects.html)页面。

必须先通过调用xQueueCreateSet（）显式创建队列集，然后才能使用它。创建之后，可以使用对[xQueueAddToSet（）的](https://www.freertos.org/xQueueAddToSet.html)调用将标准FreeRTOS队列和信号量添加到集合中。 然后，使用[xQueueSelectFromSet（）](https://www.freertos.org/xQueueSelectFromSet.html)确定集合中包含的哪些队列或信号量（如果有）处于队列读取或信号量获取操作成功的状态。

笔记：

* 队列和信号量添加到队列集中时，**必须为空**。添加对象时，例如使用已经可用的信号量创建的二进制信号量时，请格外小心[如果使用vSemaphoreCreateBinary（）宏创建信号量，则为这种情况，如果使用首选的xSemaphoreCreateBinary（）创建信号量，则为这种情况。功能]。
* 在包含互斥量的队列集上进行阻止将不会导致互斥体持有者继承被阻止任务的优先级。
* 添加到队列集的每个队列中的每个空间都需要额外的4个字节的RAM。因此，不应将具有最大最大计数值的计数信号量添加到队列集。
* 除非对xQueueSelectFromSet（）的调用先返回了该集合成员的句柄，否则不得对队列集合的成员执行接收（对于队列而言）或获取（对于信号量而言）操作。

**参数：**

|  |  |
| --- | --- |
| uxEventQueueLength | 队列集存储发生在队列中的事件和集合中包含的信号量。uxEventQueueLength指定可以立即排队的最大事件数。  要绝对确定事件不会丢失，必须将uxEventQueueLength设置为添加到集合中的队列长度的总和，其中二进制信号量和互斥量的长度为1，计数信号量的长度由其最大计数值设置。例如：   * 如果队列集将保留长度为5的队列，另一个长度为12的队列以及二进制信号量，则uxEventQueueLength应该设置为（5 + 12 +1）或18。 * 如果队列集包含三个二进制信号量，则应将uxEventQueueLength设置为（1 +1 + 1）或3。 * 如果队列集要保存最大计数为5的计数信号量和最大计数为3的计数信号量，则uxEventQueueLength应该设置为（5 + 3）或8。 |

**返回值：**

如果成功创建了队列集，那么将返回创建的队列集的句柄。否则返回NULL。

**用法示例：**

**/\* Define the lengths of the queues that will be added to the queue set. \*/**

**#define QUEUE\_LENGTH\_1 10**

**#define QUEUE\_LENGTH\_2 10**

**/\* Binary semaphores have an effective length of 1. \*/**

**#define BINARY\_SEMAPHORE\_LENGTH 1**

**/\* Define the size of the item to be held by queue 1 and queue 2 respectively.**

**The values used here are just for demonstration purposes. \*/**

**#define ITEM\_SIZE\_QUEUE\_1 sizeof( uint32\_t )**

**#define ITEM\_SIZE\_QUEUE\_2 sizeof( something\_else\_t )**

**/\* The combined length of the two queues and binary semaphore that will be**

**added to the queue set. \*/**

**#define COMBINED\_LENGTH ( QUEUE\_LENGTH\_1 +**

**QUEUE\_LENGTH\_2 +**

**BINARY\_SEMAPHORE\_LENGTH )**

**void vAFunction( void )**

**{**

**static QueueSetHandle\_t xQueueSet;**

**QueueHandle\_t xQueue1, xQueue2, xSemaphore;**

**QueueSetMemberHandle\_t xActivatedMember;**

**uint32\_t xReceivedFromQueue1;**

**something\_else\_t xReceivedFromQueue2;**

**/\* Create the queue set large enough to hold an event for every space in**

**every queue and semaphore that is to be added to the set. \*/**

**xQueueSet = xQueueCreateSet( COMBINED\_LENGTH );**

**/\* Create the queues and semaphores that will be contained in the set. \*/**

**xQueue1 = xQueueCreate( QUEUE\_LENGTH\_1, ITEM\_SIZE\_QUEUE\_1 );**

**xQueue2 = xQueueCreate( QUEUE\_LENGTH\_2, ITEM\_SIZE\_QUEUE\_2 );**

**/\* Create the semaphore that is being added to the set. \*/**

**xSemaphore = xSemaphoreCreateBinary();**

**/\* Check everything was created. \*/**

**configASSERT( xQueueSet );**

**configASSERT( xQueue1 );**

**configASSERT( xQueue2 );**

**configASSERT( xSemaphore );**

**/\* Add the queues and semaphores to the set. Reading from these queues and**

**semaphore can only be performed after a call to xQueueSelectFromSet() has**

**returned the queue or semaphore handle from this point on. \*/**

**xQueueAddToSet( xQueue1, xQueueSet );**

**xQueueAddToSet( xQueue2, xQueueSet );**

**xQueueAddToSet( xSemaphore, xQueueSet );**

**for( ;; )**

**{**

**/\* Block to wait for something to be available from the queues or**

**semaphore that have been added to the set. Don't block longer than**

**200ms. \*/**

**xActivatedMember = xQueueSelectFromSet( xQueueSet,**

**200 / portTICK\_PERIOD\_MS );**

**/\* Which set member was selected? Receives/takes can use a block time**

**of zero as they are guaranteed to pass because xQueueSelectFromSet()**

**would not have returned the handle unless something was available. \*/**

**if( xActivatedMember == xQueue1 )**

**{**

**xQueueReceive( xActivatedMember, &xReceivedFromQueue1, 0 );**

**vProcessValueFromQueue1( xReceivedFromQueue1 );**

**}**

**else if( xActivatedMember == xQueue2 )**

**{**

**xQueueReceive( xActivatedMember, &xReceivedFromQueue2, 0 );**

**vProcessValueFromQueue2( &xReceivedFromQueue2 );**

**}**

**else if( xActivatedMember == xSemaphore )**

**{**

**/\* Take the semaphore to make sure it can be "given" again. \*/**

**xSemaphoreTake( xActivatedMember, 0 );**

**vProcessEventNotifiedBySemaphore();**

**break;**

**}**

**else**

**{**

**/\* The 200ms block time expired without an RTOS queue or semaphore**

**being ready to process. \*/**

**}**

**}**

**}**

# xQueueAddToSet() [[Queue Set API](https://www.freertos.org/RTOS-queue-sets.html)]

queue.h

**BaseType\_t xQueueAddToSet**

**(**

**QueueSetMemberHandle\_t xQueueOrSemaphore,**

**QueueSetHandle\_t xQueueSet**

**);**

必须将FreeRTOSConfig.h中的configUSE\_QUEUE\_SETS设置为1，才能使用xQueueAddToSet（）API函数。

将RTOS队列或信号量添加到先前通过调用[xQueueCreateSet（）](https://www.freertos.org/xQueueCreateSet.html)创建的队列集中。

除非对xQueueSelectFromSet（）的调用先返回了该集合成员的句柄，否则不得对队列集合的成员执行接收（对于队列而言）或获取（对于信号量而言）操作。

**参数：**

|  |  |
| --- | --- |
| xQueueOrSemaphore | 将队列或信号量的句柄添加到队列集中（广播到QueueSetMemberHandle\_t类型）。 |
| xQueueSet | 要添加队列或信号量的队列集的句柄。 |

**返回值：**

如果队列或信号量已成功添加到队列集中，则返回pdPASS。如果由于队列已经是其他队列集的成员而无法将其成功添加到队列集，则返回pdFAIL。

**用法示例：**

请参阅[xQueueCreateSet（）](https://www.freertos.org/xQueueCreateSet.html)文档页面上的[示例](https://www.freertos.org/xQueueCreateSet.html)。

# xQueueRemoveFromSet() [[Queue Set API](https://www.freertos.org/RTOS-queue-sets.html)]

queue.h

**BaseType\_t xQueueRemoveFromSet**

**(**

**QueueSetMemberHandle\_t xQueueOrSemaphore,**

**QueueSetHandle\_t xQueueSet**

**);**

必须在FreeRTOSConfig.h中将configUSE\_QUEUE\_SETS设置为1，才能使用xQueueRemoveFromSet（）API函数。

从队列集中删除RTOS队列或信号量。

仅当队列或信号为空时，才可以从队列集中删除RTOS队列或信号。

**参数：**

|  |  |
| --- | --- |
| *xQueueOrSemaphore* | 从队列集中删除队列或信号量的句柄（广播到QueueSetMemberHandle\_t类型）。 |
| *xQueueSet* | 队列集的句柄，其中包含队列或信号量。 |

**返回值：**

如果已成功从队列集中删除队列或信号量，则返回pdPASS。如果队列不在队列集中，或者队列（或信号量）不为空，则返回pdFAIL。

**用法示例：**

本示例假定xQueueSet是已经创建的队列集，并且xQueue是已经创建并添加到xQueueSet的队列。

**if( xQueueRemoveFromSet( xQueue, xQueueSet ) != pdPASS )**

**{**

**/\* Either xQueue was not a member of the xQueueSet set, or xQueue is**

**not empty and therefore cannot be removed from the set. \*/**

**}**

**else**

**{**

**/\* The queue was successfully removed from the set. \*/**

**}**

# xQueueSelectFromSet() [[Queue Set API](https://www.freertos.org/RTOS-queue-sets.html)]

queue.h

**QueueSetMemberHandle\_t xQueueSelectFromSet**

**(**

**QueueSetHandle\_t xQueueSet,**

**const TickType\_t xTicksToWait**

**);**

必须将FreeRTOSConfig.h中的configUSE\_QUEUE\_SETS设置为1，才能使用xQueueSelectFromSet（）API函数。

xQueueSelectFromSet（）从队列集合的成员中选择一个队列或信号量，该队列或信号量包含数据（对于队列而言）或可用于数据（对于信号量而言）。xQueueSelectFromSet（）有效地允许任务阻塞（挂起）对队列中所有队列和信号量的同时读取操作。

笔记：

* 有使用队列集的更简单的选择。有关更多信息，请参见“ [阻止多个对象”](https://www.freertos.org/Pend-on-multiple-rtos-objects.html)页面。
* 在包含互斥量的队列集上进行阻止将不会导致互斥体持有者继承被阻止任务的优先级。
* 除非对xQueueSelectFromSet（）的调用先返回了该集合成员的句柄，否则不得对队列集合的成员执行接收（对于队列而言）或获取（对于信号量而言）操作。

**参数：**

|  |  |
| --- | --- |
| xQueueSet | 任务将（可能）阻塞的队列集。 |
| xTicksToWait | 调用任务将保持在阻塞状态（正在执行其他任务）等待队列集中的成员准备好成功进行队列读取或信号接收操作的最长时间（以滴答为单位）。 |

**返回值：**

xQueueSelectFromSet（）将返回包含数据的队列集中包含的队列的句柄（广播到QueueSetMemberHandle\_t类型），或返回可用队列集合中包含的信号量的句柄（广播到QueueSetMemberHandle\_t类型），或者为NULL如果在指定的块时间到期之前不存在这样的队列或信号量。

**用法示例：**

请参阅[xQueueCreateSet（）](https://www.freertos.org/xQueueCreateSet.html)文档页面上的[示例](https://www.freertos.org/xQueueCreateSet.html)。

# xQueueSelectFromSetFromISR()

queue.h

**QueueSetMemberHandle\_t xQueueSelectFromSetFromISR**

**(**

**QueueSetHandle\_t xQueueSet**

**);**

必须在FreeRTOSConfig.h中将configUSE\_QUEUE\_SETS设置为1，xQueueSelectFromSetFromISR（）API函数才可用。

可以从中断服务例程（ISR）使用 的[xQueueSelectFromSet（）](https://www.freertos.org/xQueueSelectFromSet.html)版本。

**参数：**

|  |  |
| --- | --- |
| xQueueSet | 正在查询的队列集。由于此功能旨在用于中断，因此无法阻止读取。 |

**返回值：**

xQueueSelectFromSetFromISR（）将返回包含数据的队列集中包含的队列的句柄（广播到QueueSetMemberHandle\_t类型），或者返回可用队列集合中包含的信号量的句柄（广播到QueueSetMemberHandle\_t类型），或者为NULL如果不存在这样的队列或信号量。

# 信号量 [ [API](https://www.freertos.org/a00106.html) ]

|  |
| --- |
| [提示：“任务通知”可以在许多情况下提供轻量级的信号量替代方案](https://www.freertos.org/RTOS-task-notifications.html) |

## 模组

* [**xSemaphoreCreateBinary**](https://www.freertos.org/xSemaphoreCreateBinary.html)
* [**xSemaphoreCreateBinaryStatic**](https://www.freertos.org/xSemaphoreCreateBinaryStatic.html)
* [**vSemaphoreCreateBinary**](https://www.freertos.org/a00121.html) [use [xSemaphoreCreateBinary()](https://www.freertos.org/xSemaphoreCreateBinary.html) for new designs]
* [**xSemaphoreCreateCounting**](https://www.freertos.org/CreateCounting.html)
* [**xSemaphoreCreateCountingStatic**](https://www.freertos.org/xSemaphoreCreateCountingStatic.html)
* [**xSemaphoreCreateMutex**](https://www.freertos.org/CreateMutex.html)
* [**xSemaphoreCreateMutexStatic**](https://www.freertos.org/xSemaphoreCreateMutexStatic.html)
* [**xSemaphoreCreateRecursiveMutex**](https://www.freertos.org/xSemaphoreCreateRecursiveMutex.html)
* [**xSemaphoreCreateRecursiveMutexStatic**](https://www.freertos.org/xSemaphoreCreateRecursiveMutexStatic.html)
* [**vSemaphoreDelete**](https://www.freertos.org/a00113.html#vSemaphoreDelete)
* [**xSemaphoreGetMutexHolder**](https://www.freertos.org/xSemaphoreGetMutexHolder.html)
* [**xSemaphoreTake**](https://www.freertos.org/a00122.html)
* [**xSemaphoreTakeFromISR**](https://www.freertos.org/xSemaphoreTakeFromISR.html)
* [**xSemaphoreTakeRecursive**](https://www.freertos.org/xSemaphoreTakeRecursive.html)
* [**xSemaphoreGive**](https://www.freertos.org/a00123.html)
* [**xSemaphoreGiveRecursive**](https://www.freertos.org/xSemaphoreGiveRecursive.html)
* [**xSemaphoreGiveFromISR**](https://www.freertos.org/a00124.html)
* [**uxSemaphoreGetCount**](https://www.freertos.org/uxSemaphoreGetCount.html)

### vSemaphoreDelete

semphr.h

**void vSemaphoreDelete( SemaphoreHandle\_t xSemaphore );**

删除一个信号量，包括互斥锁类型的信号量和递归信号量。

请勿删除已阻塞任务的信号量（处于等待状态的任务处于阻塞状态的任务）。

**参数：**

|  |  |
| --- | --- |
| *xSemaphore* | 信号量的句柄被删除。 |

# xSemaphoreCreateBinary [[信号量](https://www.freertos.org/a00113.html)]

|  |
| --- |
| [**提示：在许多使用情况下，使用直接任务通知而不是二进制信号量会更快，内存效率更高**](https://www.freertos.org/RTOS_Task_Notification_As_Binary_Semaphore.html) |

semphr. h

**SemaphoreHandle\_t xSemaphoreCreateBinary( void );**

创建一个[二进制信号量](https://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html)，并返回一个可以引用该信号量的句柄。 [要使](https://www.freertos.org/a00110.html#configSUPPORT_DYNAMIC_ALLOCATION) 此RTOS API功能可用，必须在FreeRTOSConfig.h[中将configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)设置为1，或将其保留为未定义状态（在这种情况下，它将默认为1）。

每个二进制信号量都需要少量的RAM，用于保持信号量的状态。如果使用xSemaphoreCreateBinary（）创建了二进制信号量，则所需的RAM将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配。如果使用[xSemaphoreCreateBinaryStatic（）](https://www.freertos.org/xSemaphoreCreateBinaryStatic.html)创建了二进制信号量，则 RAM由应用程序编写器提供，这需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

信号量在“空”的状态创建的，这意味着必须首先使用被给予旗语[xSemaphoreGive（）](https://www.freertos.org/a00123.html) API函数可随后采取之前（获得）使用[xSemaphoreTake](https://www.freertos.org/a00122.html)（）函数。

二进制信号量和[互斥](https://www.freertos.org/CreateMutex.html)量非常相似，但有一些细微的区别：互斥体包括优先级继承机制，二进制信号量则没有。这使得二进制信号量是实现同步（任务之间或任务之间和中断之间）的更好选择，而互斥体则是实现简单互斥的更好选择。

一旦获得了二进制信号量，就无需将其返回，因此可以通过一个任务/中断连续地“给予”该信号量而另一个任务持续地“获取”该信号量来实现任务同步。[xSemaphoreGiveFromISR（）](https://www.freertos.org/a00124.html)文档页面上的示例代码对此进行了演示。请注意，通常可以使用[直接任务通知](https://www.freertos.org/RTOS_Task_Notification_As_Binary_Semaphore.html)以更有效的方式实现相同的功能。

如果另一个具有较高优先级的任务尝试获取相同的互斥锁，则可能会提高“采用”互斥锁的任务的优先级。拥有互斥锁的任务“继承”尝试“采用”相同互斥锁的任务的优先级。这意味着必须始终“放弃”互斥锁–否则，优先级较高的任务将永远无法获得互斥锁，而优先级较低的任务将永远不会“取消继承”优先级。[xSemaphoreTake（）](https://www.freertos.org/a00122.html)文档页面上提供了用于实现互斥的互斥锁的示例。

互斥量和二进制信号量均由SemaphoreHandle\_t类型的变量引用，并且可以在采用该类型参数的任何任务级API函数中使用。与互斥锁不同，二进制信号量可以在中断服务程序中使用。

**返回值：**

|  |  |
| --- | --- |
| NULL | 无法创建信号灯，因为 可用[FreeRTOS堆](https://www.freertos.org/a00111.html)不足。 |
| 任何其他值 | 信号灯已成功创建。返回的值是可用来引用信号量的句柄。 |

**用法示例：**

**SemaphoreHandle\_t xSemaphore;**

**void vATask( void \* pvParameters )**

**{**

**/\* Attempt to create a semaphore. \*/**

**xSemaphore = xSemaphoreCreateBinary();**

**if( xSemaphore == NULL )**

**{**

**/\* There was insufficient FreeRTOS heap available for the semaphore to**

**be created successfully. \*/**

**}**

**else**

**{**

**/\* The semaphore can now be used. Its handle is stored in the**

**xSemahore variable. Calling xSemaphoreTake() on the semaphore here**

**will fail until the semaphore has first been given. \*/**

**}**

**}**

# xSemaphoreCreateBinaryStatic [[信号量](https://www.freertos.org/a00113.html)]

|  |
| --- |
| [**提示：在许多使用情况下，使用直接任务通知而不是二进制信号量会更快，内存效率更高**](https://www.freertos.org/RTOS_Task_Notification_As_Binary_Semaphore.html) |

semphr. h

**SemaphoreHandle\_t xSemaphoreCreateBinaryStatic(**

**StaticSemaphore\_t \*pxSemaphoreBuffer );**

创建一个[二进制信号量](https://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html)，并返回一个可以引用该信号量的句柄。 必须在FreeRTOSConfig.h[中将configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_STATIC_ALLOCATION)设置为1，此RTOS API函数才可用。

每个二进制信号量都需要少量的RAM，用于保持信号量的状态。如果使用[xSemaphoreCreateBinary（）](https://www.freertos.org/xSemaphoreCreateBinary.html)创建了二进制信号量，则 所需的RAM将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配。如果使用xSemaphoreCreateBinaryStatic（）创建了二进制信号量，则RAM由应用程序编写器提供，这需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

信号量在“空”的状态创建的，这意味着必须首先使用被给予旗语[xSemaphoreGive（）](https://www.freertos.org/a00123.html) API函数可随后采取之前（获得）使用[xSemaphoreTake](https://www.freertos.org/a00122.html)（）函数。

二进制信号量和[互斥](https://www.freertos.org/CreateMutex.html)量非常相似，但有一些细微的区别：互斥体包括优先级继承机制，二进制信号量则没有。这使得二进制信号量是实现同步（任务之间或任务之间和中断之间）的更好选择，而互斥体则是实现简单互斥的更好选择。

一旦获得了二进制信号量，就无需将其返回，因此可以通过一个任务/中断连续地“给予”该信号量而另一个任务持续地“获取”该信号量来实现任务同步。[xSemaphoreGiveFromISR（）](https://www.freertos.org/a00124.html)文档页面上的示例代码对此进行了演示。请注意，通常可以使用[直接任务通知](https://www.freertos.org/RTOS_Task_Notification_As_Binary_Semaphore.html)以更有效的方式实现相同的功能。

如果另一个具有较高优先级的任务尝试获取相同的互斥锁，则可能会提高“采用”互斥锁的任务的优先级。拥有互斥锁的任务“继承”尝试“采用”相同互斥锁的任务的优先级。这意味着必须始终“放弃”互斥锁–否则，优先级较高的任务将永远无法获得互斥锁，而优先级较低的任务将永远不会“取消继承”优先级。[xSemaphoreTake（）](https://www.freertos.org/a00122.html)文档页面上提供了用于实现互斥的互斥锁的示例。

互斥量和二进制信号量均由SemaphoreHandle\_t类型的变量引用，并且可以在采用该类型参数的任何任务级API函数中使用。与互斥锁不同，二进制信号量可以在中断服务程序中使用。

**参数：**

|  |  |
| --- | --- |
| pxSemaphoreBuffer | 必须指向类型为StaticSemaphore\_t的变量，该变量将用于保留信号量的状态。 |

**返回值：**

|  |  |
| --- | --- |
| 空值 | 由于pxSemaphoreBuffer为NULL，因此无法创建信号灯。 |
| 任何其他值 | 信号灯已成功创建。返回的值是可用来引用信号量的句柄。 |

**用法示例：**

**SemaphoreHandle\_t xSemaphore = NULL;**

**StaticSemaphore\_t xSemaphoreBuffer;**

**void vATask( void \* pvParameters )**

**{**

**/\* Create a binary semaphore without using any dynamic memory**

**allocation. The semaphore's data structures will be saved into**

**the xSemaphoreBuffer variable. \*/**

**xSemaphore = xSemaphoreCreateBinaryStatic( &xSemaphoreBuffer );**

**/\* The pxSemaphoreBuffer was not NULL, so it is expected that the**

**handle will not be NULL. \*/**

[**configASSERT**](https://www.freertos.org/a00110.html#configASSERT)**( xSemaphore );**

**/\* Rest of the task code goes here. \*/**

**}**

# vSemaphoreCreateBinary [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**vSemaphoreCreateBinary( SemaphoreHandle\_t xSemaphore )**

**注意：** vSemaphoreCreateBinary（）宏保留在源代码中，以确保向后兼容，但不应在新设计中使用它。请改用 [xSemaphoreCreateBinary（）](https://www.freertos.org/xSemaphoreCreateBinary.html)函数。

同样，在许多情况下，使用[直接任务通知](https://www.freertos.org/RTOS-task-notifications.html) 来代替二进制信号量更快，更节省内存 。

使用现有队列机制创建信号灯的宏。队列长度为1，因为这是二进制信号量。数据大小为0，因为我们不想实际存储任何数据–我们只想知道队列是空还是已满。

二进制信号量和[互斥](https://www.freertos.org/CreateMutex.html)量非常相似，但有一些细微的区别：互斥体包括优先级继承机制，二进制信号量则没有。这使得二进制信号量是实现同步（任务之间或任务之间和中断之间）的更好选择，而互斥体则是实现简单互斥的更好选择。

一旦获得了二进制信号量，就无需将其返回，因此可以通过一个任务/中断连续地“给予”该信号量而另一个任务持续地“获取”该信号量来实现任务同步。[xSemaphoreGiveFromISR（）](https://www.freertos.org/a00124.html)文档页面上的示例代码对此进行了演示。

如果另一个具有较高优先级的任务尝试获取相同的互斥锁，则可能会提高“采用”互斥锁的任务的优先级。拥有互斥锁的任务“继承”尝试“采用”相同互斥锁的任务的优先级。这意味着必须始终“放弃”互斥锁–否则，优先级较高的任务将永远无法获得互斥锁，而优先级较低的任务将永远不会“取消继承”优先级。[xSemaphoreTake（）](https://www.freertos.org/a00122.html)文档页面上提供了用于实现互斥的互斥锁的示例。

互斥量和二进制信号量都被分配给SemaphoreHandle\_t类型的变量，并且可以在任何采用此类型参数的API函数中使用。

**参数：**

|  |  |
| --- | --- |
| x信号量 | 处理创建的信号灯。应为SemaphoreHandle\_t类型。 |

**用法示例：**

**SemaphoreHandle\_t xSemaphore;**

**void vATask( void \* pvParameters )**

**{**

**// Semaphore cannot be used before a call to vSemaphoreCreateBinary ().**

**// This is a macro so pass the variable in directly.**

**vSemaphoreCreateBinary( xSemaphore );**

**if( xSemaphore != NULL )**

**{**

**// The semaphore was created successfully.**

**// The semaphore can now be used.**

**}**

**}**

# xSemaphoreCreateCounting [[信号量](https://www.freertos.org/a00113.html)]

|  |
| --- |
| [**提示：在许多情况下，“任务通知”可以为计数信号量提供轻巧的选择**](https://www.freertos.org/RTOS_Task_Notification_As_Counting_Semaphore.html) |

semphr.h

**SemaphoreHandle\_t xSemaphoreCreateCounting( UBaseType\_t uxMaxCount,**

**UBaseType\_t uxInitialCount);**

创建一个[计数信号量](https://www.freertos.org/Real-time-embedded-RTOS-Counting-Semaphores.html) 并返回一个句柄，通过它可以引用新创建的信号量。 [要使](https://www.freertos.org/a00110.html#configSUPPORT_DYNAMIC_ALLOCATION) 此RTOS API功能可用，必须在FreeRTOSConfig.h[中将configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)设置为1，或将其保留为未定义状态（在这种情况下，它将默认为1）。

每个计数信号量都需要少量的RAM，用于保持信号量的状态。如果使用xSemaphoreCreateCounting（）创建了计数信号量，则将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用[xSemaphoreCreateCountingStatic（）](https://www.freertos.org/xSemaphoreCreateCountingStatic.html)创建计数信号量，则 RAM由应用程序编写器提供，这需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

计数信号量通常用于两件事：

1. 计数事件。

在这种使用情况下，事件处理程序将在每次事件发生时“给出”信号量（增加信号量计数值），而处理程序任务将在每次处理事件时“获得”信号量（减小信号量计数值）。因此，计数值是已发生的事件数与已处理的事件数之间的差。在这种情况下，期望初始计数值为零。

请注意，通常可以使用[直接任务通知](https://www.freertos.org/RTOS-task-notifications.html)以更有效的方式实现相同的功能。

1. 资源管理。

在这种使用情况下，计数值指示可用资源的数量。要获得对资源的控制，任务必须首先获得一个信号量-减少信号量计数值。当计数值达到零时，将没有可用资源。当任务使用资源完成时，它将“给予”信号量–增加信号量计数值。在这种情况下，期望初始计数值等于最大计数值，指示所有资源都是空闲的。

**参数：**

|  |  |
| --- | --- |
| uxMaxCount | 可以达到的最大计数值。当信号量达到该值时，它就不能再被“给予”了。 |
| uxInitialCount | 创建信号量时分配给它的计数值。 |

**返回值：**

如果成功创建了信号灯，则返回该信号灯的句柄。如果由于[无法分配](https://www.freertos.org/a00111.html)保存信号量所需的RAM而无法创建信号量， 则返回NULL。

**用法示例：**

**void vATask( void \* pvParameters )**

**{**

**SemaphoreHandle\_t xSemaphore;**

**/\* Create a counting semaphore that has a maximum count of 10 and an**

**initial count of 0. \*/**

**xSemaphore = xSemaphoreCreateCounting( 10, 0 );**

**if( xSemaphore != NULL )**

**{**

**/\* The semaphore was created successfully. \*/**

**}**

**}**

# xSemaphoreCreateCountingStatic [[信号量](https://www.freertos.org/a00113.html)]

|  |
| --- |
| [**提示：在许多情况下，“任务通知”可以为计数信号量提供轻巧的选择**](https://www.freertos.org/RTOS_Task_Notification_As_Counting_Semaphore.html) |

semphr. h

**SemaphoreHandle\_t xSemaphoreCreateCountingStatic(**

**UBaseType\_t uxMaxCount,**

**UBaseType\_t uxInitialCount**

**StaticSemaphore\_t \*pxSemaphoreBuffer ); );**

创建一个[计数信号量](https://www.freertos.org/Real-time-embedded-RTOS-Counting-Semaphores.html) 并返回一个句柄，通过它可以引用新创建的信号量。 必须在FreeRTOSConfig.h[中将configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_STATIC_ALLOCATION)设置为1，此RTOS API函数才可用。

每个计数信号量都需要少量的RAM，用于保持信号量的状态。如果使用[xSemaphoreCreateCounting（）](https://www.freertos.org/CreateCounting.html)创建了计数信号量， 则将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用xSemaphoreCreateCountingStatic（）创建计数信号量，则RAM由应用程序编写器提供，这需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

计数信号量通常用于两件事：

1. 计数事件。

在这种使用情况下，事件处理程序将在每次事件发生时“给出”信号量（增加信号量计数值），而处理程序任务将在每次处理事件时“获得”信号量（减小信号量计数值）。因此，计数值是已发生的事件数与已处理的事件数之间的差。在这种情况下，期望初始计数值为零。

请注意，通常可以使用[直接任务通知](https://www.freertos.org/RTOS-task-notifications.html)以更有效的方式实现相同的功能。

1. 资源管理。

在这种使用情况下，计数值指示可用资源的数量。要获得对资源的控制，任务必须首先获得一个信号量-减少信号量计数值。当计数值达到零时，将没有可用资源。当任务使用资源完成时，它将“给予”信号量–增加信号量计数值。在这种情况下，期望初始计数值等于最大计数值，指示所有资源都是空闲的。

**参数：**

|  |  |
| --- | --- |
| uxMaxCount | 可以达到的最大计数值。当信号量达到该值时，它就不能再被“给予”了。 |
| uxInitialCount | 创建信号量时分配给它的计数值。 |
| pxSemaphoreBuffer | 必须指向类型为StaticSemaphore\_t的变量，然后该变量用于保存信号量的数据结构。 |

**返回值：**

如果成功创建了信号灯，则返回该信号灯的句柄。如果pxSemaphoreBuffer为NULL，则返回NULL。

**用法示例：**

**static StaticSemaphore\_t xSemaphoreBuffer;**

**void vATask( void \* pvParameters )**

**{**

**SemaphoreHandle\_t xSemaphore;**

**/\* Create a counting semaphore that has a maximum count of 10 and an**

**initial count of 0. The semaphore's data structures are stored in the**

**xSemaphoreBuffer variable - no dynamic memory allocation is performed. \*/**

**xSemaphore = xSemaphoreCreateCountingStatic( 10, 0, &xSemaphoreBuffer );**

**/\* pxSemaphoreBuffer was not NULL so it is expected that the semaphore**

**will be created. \*/**

[**configASSERT**](https://www.freertos.org/a00110.html#configASSERT)**( xSemaphore );**

**}**

# xSemaphoreCreateMutex [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**SemaphoreHandle\_t xSemaphoreCreateMutex( void )**

创建一个[互斥锁](https://www.freertos.org/Real-time-embedded-RTOS-mutexes.html)，并返回一个句柄，通过该句柄可以引用所创建的互斥锁。互斥体不能在中断服务程序中使用。

[要使](https://www.freertos.org/a00110.html#configSUPPORT_DYNAMIC_ALLOCATION) 此RTOS API功能可用，必须在FreeRTOSConfig.h[中将configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)设置为1，或将其保留为未定义状态（在这种情况下，它将默认为1）。

每个互斥锁都需要少量的RAM，用于保持互斥锁的状态。如果使用xSemaphoreCreateMutex（）创建了互斥锁，则将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用[xSemaphoreCreateMutexStatic（）](https://www.freertos.org/xSemaphoreCreateMutexStatic.html)创建了互斥锁，则 RAM由应用程序编写器提供，这需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

使用[xSemaphoreTake](https://www.freertos.org/a00122.html)（）获取互斥体，并使用[xSemaphoreGive（）](https://www.freertos.org/a00123.html)给出互斥体。xSemaphoreTakeRecursive（）和xSemaphoreGiveRecursive（）仅可用于使用[xSemaphoreCreateRecursiveMutex](https://www.freertos.org/xSemaphoreCreateRecursiveMutex.html)（）创建的互斥锁。

互斥[量](https://www.freertos.org/a00121.html)和[二进制信号量](https://www.freertos.org/a00121.html)非常相似，但有一些细微的区别：互斥量包括优先级继承机制，二进制信号量没有。这使得二进制信号量是实现同步（任务之间或任务之间和中断之间）的更好选择，而互斥体则是实现简单互斥的更好选择。

如果另一个具有较高优先级的任务尝试获取相同的互斥锁，则“采用”互斥锁的任务的优先级将暂时提高。拥有互斥锁的任务“继承”尝试“采用”相同互斥锁的任务的优先级。这意味着必须始终“放弃”互斥锁–否则，优先级较高的任务将永远无法获得互斥锁，而优先级较低的任务将永远不会“取消继承”优先级。

[xSemaphoreTake（）](https://www.freertos.org/a00122.html)文档页面 上提供了用于实现互斥的互斥锁的示例。

一旦获得了二进制信号量，就无需将其返回，因此可以通过一个任务/中断连续地“给予”该信号量而另一个任务持续地“获取”该信号量来实现任务同步。[xSemaphoreGiveFromISR（）](https://www.freertos.org/a00124.html)文档页面上的示例代码对此进行了演示。注意，使用直接任务通知可以以更有效的方式实现相同的功能。

互斥量和二进制信号量的句柄都分配给SemaphoreHandle\_t类型的变量，并且可以在采用该类型参数的任何任务级别（与中断安全相反）的API函数中使用。

**返回：**

|  |  |
| --- | --- |
|  | 如果成功创建了互斥锁类型信号量，则返回创建的互斥锁的句柄。如果由于[无法分配](https://www.freertos.org/a00111.html)保存互斥锁所需的内存而未创建互斥锁 [，](https://www.freertos.org/a00111.html) 则返回NULL。 |

**用法示例：**

**SemaphoreHandle\_t xSemaphore;**

**void vATask( void \* pvParameters )**

**{**

**/\* Create a mutex type semaphore. \*/**

**xSemaphore = xSemaphoreCreateMutex();**

**if( xSemaphore != NULL )**

**{**

**/\* The semaphore was created successfully and**

**can be used. \*/**

**}**

**}**

# xSemaphoreCreateMutexStatic [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**SemaphoreHandle\_t xSemaphoreCreateMutexStatic(**

**StaticSemaphore\_t \*pxMutexBuffer );**

创建一个[互斥锁](https://www.freertos.org/Real-time-embedded-RTOS-mutexes.html)，并返回一个句柄，通过该句柄可以引用所创建的互斥锁。互斥体不能在中断服务程序中使用。

必须在FreeRTOSConfig.h[中将configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_STATIC_ALLOCATION)设置为1，此RTOS API函数才可用。

每个互斥锁都需要少量的RAM，用于保持互斥锁的状态。如果使用[xSemaphoreCreateMutex](https://www.freertos.org/CreateMutex.html)（）创建了互斥锁，则将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用xSemaphoreCreateMutexStatic（）创建了互斥锁，则RAM由应用程序编写器提供，这需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

使用[xSemaphoreTake](https://www.freertos.org/a00122.html)（）获取互斥体，并使用[xSemaphoreGive（）](https://www.freertos.org/a00123.html)给出互斥体。xSemaphoreTakeRecursive（）和xSemaphoreGiveRecursive（）仅可用于使用[xSemaphoreCreateResursiveMutex](https://www.freertos.org/xSemaphoreCreateRecursiveMutex.html)（）创建的互斥锁。

互斥[量](https://www.freertos.org/a00121.html)和[二进制信号量](https://www.freertos.org/a00121.html)非常相似，但有一些细微的区别：互斥量包括优先级继承机制，二进制信号量没有。这使得二进制信号量是实现同步（任务之间或任务之间和中断之间）的更好选择，而互斥体则是实现简单互斥的更好选择。

如果另一个具有较高优先级的任务尝试获取相同的互斥锁，则“采用”互斥锁的任务的优先级将暂时提高。拥有互斥锁的任务“继承”尝试“采用”相同互斥锁的任务的优先级。这意味着必须始终“放弃”互斥锁–否则，优先级较高的任务将永远无法获得互斥锁，而优先级较低的任务将永远不会“取消继承”优先级。

[xSemaphoreTake（）](https://www.freertos.org/a00122.html)文档页面 上提供了用于实现互斥的互斥锁的示例。

一旦获得了二进制信号量，就无需将其返回，因此可以通过一个任务/中断连续地“给予”该信号量而另一个任务持续地“获取”该信号量来实现任务同步。[xSemaphoreGiveFromISR（）](https://www.freertos.org/a00124.html)文档页面上的示例代码对此进行了演示。注意，使用直接任务通知可以以更有效的方式实现相同的功能。

互斥量和二进制信号量的句柄都分配给SemaphoreHandle\_t类型的变量，并且可以在采用该类型参数的任何任务级别（与中断安全相反）的API函数中使用。

**参数：**

|  |  |
| --- | --- |
| pxMutexBuffer | 必须指向类型为StaticSemaphore\_t的变量，该变量将用于保持互斥类型信号量的状态。 |

**返回：**

|  |  |
| --- | --- |
|  | 如果成功创建了互斥锁类型信号量，则返回创建的互斥锁的句柄。如果由于pxMutexBuffer为NULL而未创建互斥锁，则返回NULL。 |

**用法示例：**

**SemaphoreHandle\_t xSemaphore = NULL;**

**StaticSemaphore\_t xMutexBuffer;**

**void vATask( void \* pvParameters )**

**{**

**/\* Create a mutex semaphore without using any dynamic memory**

**allocation. The mutex's data structures will be saved into**

**the xMutexBuffer variable. \*/**

**xSemaphore = xSemaphoreCreateMutexStatic( &xMutexBuffer );**

**/\* The pxMutexBuffer was not NULL, so it is expected that the**

**handle will not be NULL. \*/**

[**configASSERT**](https://www.freertos.org/a00110.html#configASSERT)**( xSemaphore );**

**/\* Rest of the task code goes here. \*/**

**}**

# xSemaphoreCreateRecursiveMutex [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**SemaphoreHandle\_t xSemaphoreCreateRecursiveMutex( void )**

创建一个[递归互斥对象](https://www.freertos.org/RTOS-Recursive-Mutexes.html)，并返回一个可以引用该互斥对象的句柄。递归互斥锁不能在中断服务程序中使用。 必须在FreeRTOSConfig.h[中将configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)和configUSE\_RECURSIVE\_MUTEXES都设置为1，以使xSemaphoreCreateRecursiveMutex（）可用（configSUPPORT\_DYNAMIC\_ALLOCATION也可以保持未定义状态，在这种情况下将失败，直到1）。

每个递归互斥体都需要少量的RAM，用于保持递归互斥体的状态。如果使用[xSemaphoreCreateRecursiveMutex（）](https://www.freertos.org/xSemaphoreCreateRecursiveMutex.html)创建了互斥锁， 则将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用xSemaphoreCreateRecursiveMutexStatic（）创建了递归互斥锁，则RAM由应用程序编写器提供，它需要一个附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

分别使用[xSemaphoreTakeRecursive（）](https://www.freertos.org/xSemaphoreTakeRecursive.html) “获取”（获取）递归互斥锁，并使用[xSemaphoreGiveRecursive（）](https://www.freertos.org/xSemaphoreGiveRecursive.html) API函数“释放 ”递归互斥锁 。不得使用xSemaphoreTake（）和xSemaphoreGive（）。

使用xSemaphoreCreateMutex（）和xSemaphoreCreateMutexStatic（）创建非递归互斥体。一个非递归互斥锁只能被一个任务“占用”一次。任务尝试使用它已经拥有的非递归互斥锁的任何尝试都将失败–并且该互斥锁将始终在任务首次“提供”互斥锁时给予。

与非递归互斥锁相反，一个任务可以“采用”多次递归互斥锁，并且递归互斥锁仅在保持任务“赋予”互斥锁相同次数后才返回。

像非递归互斥锁一样，递归互斥锁实现优先级继承算法。如果另一个具有较高优先级的任务尝试获取相同的互斥锁，则“采用”互斥锁的任务的优先级将暂时提高。拥有互斥锁的任务“继承”尝试“采用”相同互斥锁的任务的优先级。这意味着必须始终“放弃”互斥锁–否则，优先级较高的任务将永远无法获得互斥锁，而优先级较低的任务将永远不会“取消继承”优先级。

**返回值：**

如果成功创建了递归互斥锁，则返回创建的互斥锁的句柄。如果由于[无法分配](https://www.freertos.org/a00111.html)保持互斥锁所需的内存而未创建递归互斥锁， 则返回NULL。

**用法示例：**

**SemaphoreHandle\_t xMutex;**

**void vATask( void \* pvParameters )**

**{**

**Create a recursive mutex.**

**xMutex = xSemaphoreCreateRecursiveMutex();**

**if( xMutex != NULL )**

**{**

**/\* The recursive mutex was created successfully and**

**can now be used. \*/**

**}**

**}**

# xSemaphoreCreateRecursiveMutexStatic [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**SemaphoreHandle\_t xSemaphoreCreateRecursiveMutexStatic(**

**StaticSemaphore\_t \*pxMutexBuffer )**

创建一个[递归互斥对象](https://www.freertos.org/RTOS-Recursive-Mutexes.html)，并返回一个可以引用该互斥对象的句柄。递归互斥锁不能在中断服务程序中使用。 必须将FreeRTOSConfig.h中的configUSE\_RECURSIVE\_MUTEXES和[configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_STATIC_ALLOCATION)都设置为1，才能使xSemaphoreCreateRecursiveMutexStatic（）可用。

每个递归互斥体都需要少量的RAM，用于保持递归互斥体的状态。如果使用[xSemaphoreCreateRecursiveMutex（）](https://www.freertos.org/xSemaphoreCreateRecursiveMutex.html)创建了互斥锁， 则将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用xSemaphoreCreateRecursiveMutexStatic（）创建了递归互斥锁，则RAM由应用程序编写器提供，它需要一个附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

分别使用[xSemaphoreTakeRecursive（）](https://www.freertos.org/xSemaphoreTakeRecursive.html) “获取”（获取）递归互斥锁，并使用[xSemaphoreGiveRecursive（）](https://www.freertos.org/xSemaphoreGiveRecursive.html) API函数“释放 ”递归互斥锁 。不得使用xSemaphoreTake（）和xSemaphoreGive（）。

使用xSemaphoreCreateMutex（）和xSemaphoreCreateMutexStatic（）创建非递归互斥体。一个非递归互斥锁只能被一个任务“占用”一次。任务尝试使用它已经拥有的非递归互斥锁的任何尝试都将失败–并且该互斥锁将始终在任务首次“提供”互斥锁时给予。

与非递归互斥锁相反，一个任务可以“采用”多次递归互斥锁，并且递归互斥锁仅在保持任务“赋予”互斥锁相同次数后才返回。

像非递归互斥锁一样，递归互斥锁实现优先级继承算法。如果另一个具有较高优先级的任务尝试获取相同的互斥锁，则“采用”互斥锁的任务的优先级将暂时提高。拥有互斥锁的任务“继承”尝试“采用”相同互斥锁的任务的优先级。这意味着必须始终“放弃”互斥锁–否则，优先级较高的任务将永远无法获得互斥锁，而优先级较低的任务将永远不会“取消继承”优先级。

**参数：**

|  |  |
| --- | --- |
| pxMutexBuffer | 必须指向类型为StaticSemaphore\_t的变量，该变量将用于保持互斥类型信号量的状态。 |

**返回：**

|  |  |
| --- | --- |
|  | 如果成功创建了递归互斥锁，则返回创建的互斥锁的句柄。如果由于pxMutexBuffer为NULL而未创建递归互斥锁，则返回NULL。 |

**用法示例：**

**SemaphoreHandle\_t xSemaphore = NULL;**

**StaticSemaphore\_t xMutexBuffer;**

**void vATask( void \* pvParameters )**

**{**

**/\* Create a recursivemutex semaphore without using any dynamic**

**memory allocation. The mutex's data structures will be saved into**

**the xMutexBuffer variable. \*/**

**xSemaphore = xSemaphoreCreateRecursiveMutexStatic( &xMutexBuffer );**

**/\* The pxMutexBuffer was not NULL, so it is expected that the**

**handle will not be NULL. \*/**

[**configASSERT**](https://www.freertos.org/a00110.html#configASSERT)**( xSemaphore );**

**/\* Rest of the task code goes here. \*/**

**}**

# xSemaphoreGetMutexHolder [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**TaskHandle\_t xSemaphoreGetMutexHolder( SemaphoreHandle\_t xMutex );**

要使此功能可用，必须在FreeRTOSConfig.h中将INCLUDE\_xSemaphoreGetMutexHolder设置为1。

返回保存函数参数指定的互斥量的任务的句柄（如果有）。

xSemaphoreGetMutexHolder（）可以可靠地用于确定调用任务是否为互斥体持有者，但是如果互斥体是由调用任务以外的任何任务持有的，则不能可靠地使用xSemaphoreGetMutexHolder（）。这是因为互斥体持有者可能会在调用函数的调用任务与测试函数返回值的调用任务之间发生变化。

必须将FreeRTOSConfig.h中的configUSE\_MUTEXES设置为1，才能使xSemaphoreGetMutexHolder（）可用。

**参数：**

|  |  |
| --- | --- |
| *xMutex* | 被查询的互斥锁的句柄。 |

**返回值：**

持有xMutex参数指定的互斥锁的任务的句柄。如果在xMutex参数中传递的信号量不是互斥锁类型信号量，或者该互斥锁可用并且因此不被任何任务保留，则返回NULL。

# uxSemaphoreGetCount [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**UBaseType\_t uxSemaphoreGetCount( SemaphoreHandle\_t xSemaphore );**

返回信号量的计数。

**参数：**

|  |  |
| --- | --- |
| *xSemaphore* | 正在查询信号量的句柄。 |

**返回值：**

如果信号量是计数信号量，则返回信号量当前计数值。如果该信号量是二进制信号量，则如果该信号量可用，则返回1，如果该信号量不可用，则返回0。

# xSemaphoreTake [[信号量](https://www.freertos.org/a00113.html)]

|  |
| --- |
| [**提示：在许多使用情况下，使用直接任务通知而不是信号量更快，更高效地使用内存**](https://www.freertos.org/RTOS-task-notifications.html) |

semphr. h

**xSemaphoreTake( SemaphoreHandle\_t xSemaphore,**

**TickType\_t xTicksToWait );**

宏获取信号量。信号量必须事先通过调用xSemaphoreCreateBinary（），xSemaphoreCreateMutex（）或xSemaphoreCreateCounting（）来创建。

不得从ISR调用此宏。如果需要，可以使用xQueueReceiveFromISR（）从中断中获取信号，尽管这不是正常操作。信号量使用队列作为其底层机制，因此功能在某种程度上可以互操作。

**参数：**

|  |  |
| --- | --- |
| xSemaphore | 信号灯的句柄–在创建信号灯时获得。 |
| xTicksToWait | 等待信号量可用的时间（以秒为单位）。宏端口TICK\_PERIOD\_MS可用于将其转换为实时。零的阻塞时间可用于轮询信号量。  如果将[INCLUDE\_vTaskSuspend](https://www.freertos.org/a00110.html)设置为“ 1”，则将阻止时间指定为portMAX\_DELAY将导致任务无限期地阻止（无超时）。 |

**返回值：**

如果已获得信号量，则为pdTRUE。如果xTicksToWait过期而没有信号量，则为pdFALSE。

**用法示例：**

**SemaphoreHandle\_t xSemaphore = NULL;**

**/\* A task that creates a semaphore. \*/**

**void vATask( void \* pvParameters )**

**{**

**/\* Create the semaphore to guard a shared resource. As we are using**

**the semaphore for mutual exclusion we create a mutex semaphore**

**rather than a binary semaphore. \*/**

**xSemaphore = xSemaphoreCreateMutex();**

**}**

**/\* A task that uses the semaphore. \*/**

**void vAnotherTask( void \* pvParameters )**

**{**

**/\* ... Do other things. \*/**

**if( xSemaphore != NULL )**

**{**

**/\* See if we can obtain the semaphore. If the semaphore is not**

**available wait 10 ticks to see if it becomes free. \*/**

**if( xSemaphoreTake( xSemaphore, ( TickType\_t ) 10 ) == pdTRUE )**

**{**

**/\* We were able to obtain the semaphore and can now access the**

**shared resource. \*/**

**/\* ... \*/**

**/\* We have finished accessing the shared resource. Release the**

**semaphore. \*/**

**xSemaphoreGive( xSemaphore );**

**}**

**else**

**{**

**/\* We could not obtain the semaphore and can therefore not access**

**the shared resource safely. \*/**

**}**

**}**

**}**

# xSemaphoreTakeFromISR [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**xSemaphoreTakeFromISR**

**(**

**SemaphoreHandle\_t xSemaphore,**

**signed BaseType\_t \*pxHigherPriorityTaskWoken**

**)**

可以从ISR调用 的[xSemaphoreTake（）](https://www.freertos.org/a00122.html)版本。与xSemaphoreTake（）不同，xSemaphoreTakeFromISR（）不允许指定块时间。

**参数：**

|  |  |
| --- | --- |
| xSemaphore | 信号量被“占用”。信号量由SemaphoreHandle\_t类型的变量引用，并且必须在使用前明确创建。 |
| pxHigherPriorityTaskWoken | 信号量有可能（尽管不太可能，并且取决于信号量的类型）在其上阻塞了一个或多个任务，等待给出信号量。调用xSemaphoreTakeFromISR（）将使任务被阻塞，等待该信号离开阻塞状态。如果调用API函数导致任务退出“已阻止”状态，并且未阻止任务的优先级等于或高于当前正在执行的任务（被中断的任务），则内部，API函数会将\* pxHigherPriorityTaskWoken设置为pdTRUE。  如果xSemaphoreTakeFromISR（）将\* pxHigherPriorityTaskWoken设置为pdTRUE，则应在退出中断之前执行上下文切换。这将确保中断直接返回到最高优先级的就绪状态任务。该机制与xQueueReceiveFromISR（）函数中使用的机制相同，读者可以参考[xQueueReceiveFromISR（）](https://www.freertos.org/a00120.html)文档进行进一步说明。  从FreeRTOS V7.3.0开始，pxHigherPriorityTaskWoken是可选参数，可以设置为NULL。 |

**返回值：**

如果成功获取了信号量，则为pdTRUE。pdFALSE，如果由于不可用而未能成功获取信号量。

he semaphore was successfully taken. pdFALSE if the semaphore was not successfully taken because it was not available.

# xSemaphoreTakeRecursive [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**xSemaphoreTakeRecursive( SemaphoreHandle\_t xMutex,**

**TickType\_t xTicksToWait );**

以递归方式获取或“获取”互斥体类型的信号量。互斥锁必须先前已通过调用xSemaphoreCreateRecursiveMutex（）创建。

必须在FreeRTOSConfig.h中将configUSE\_RECURSIVE\_MUTEXES设置为1，此宏才可用。

不得在使用xSemaphoreCreateMutex（）创建的互斥锁上使用此宏。

所有者可以反复“递归”使用递归使用的互斥锁。在所有者为每个成功的“获取”请求调用xSemaphoreGiveRecursive（）之前，互斥量不再可用。例如，如果一个任务成功“获取”了相同的互斥体5次，则该互斥体将对其他任何任务不可用，直到它也已将互斥体恰好“给予”了5次。

**参数：**

|  |  |
| --- | --- |
| *xMutex* | 获取互斥锁的句柄。这是xSemaphoreCreateRecursiveMutex（）返回的句柄。 |
| *xTicksToWait* | 等待信号量可用的时间（以秒为单位）。宏端口TICK\_PERIOD\_MS可用于将其转换为实时。零的阻塞时间可用于轮询信号量。如果任务已经拥有该信号量，则无论xTicksToWait的值如何，xSemaphoreTakeRecursive（）都会立即返回。 |

**返回值：**

如果已获得信号量，则为pdTRUE。如果xTicksToWait过期而没有信号量，则为pdFALSE。

**用法示例：**

**SemaphoreHandle\_t xMutex = NULL;**

**// A task that creates a mutex.**

**void vATask( void \* pvParameters )**

**{**

**// Create the mutex to guard a shared resource.**

**xMutex = xSemaphoreCreateRecursiveMutex();**

**}**

**// A task that uses the mutex.**

**void vAnotherTask( void \* pvParameters )**

**{**

**// ... Do other things.**

**if( xMutex != NULL )**

**{**

**// See if we can obtain the mutex. If the mutex is not available**

**// wait 10 ticks to see if it becomes free.**

**if( xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 ) == pdTRUE )**

**{**

**// We were able to obtain the mutex and can now access the**

**// shared resource.**

**// ...**

**// For some reason due to the nature of the code further calls to**

**// xSemaphoreTakeRecursive() are made on the same mutex. In real**

**// code these would not be just sequential calls as this would make**

**// no sense. Instead the calls are likely to be buried inside**

**// a more complex call structure.**

**xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 );**

**xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 );**

**// The mutex has now been 'taken' three times, so will not be**

**// available to another task until it has also been given back**

**// three times. Again it is unlikely that real code would have**

**// these calls sequentially, but instead buried in a more complex**

**// call structure. This is just for illustrative purposes.**

**xSemaphoreGiveRecursive( xMutex );**

**xSemaphoreGiveRecursive( xMutex );**

**xSemaphoreGiveRecursive( xMutex );**

**// Now the mutex can be taken by other tasks.**

**}**

**else**

**{**

**// We could not obtain the mutex and can therefore not access**

**// the shared resource safely.**

**}**

**}**

**}**

# xSemaphoreGive [[信号量](https://www.freertos.org/a00113.html)]

|  |
| --- |
| [**提示：在许多使用情况下，使用直接任务通知而不是信号量更快，更高效地使用内存**](https://www.freertos.org/RTOS-task-notifications.html) |

semphr. h

**xSemaphoreGive( SemaphoreHandle\_t xSemaphore );**

释放信号量的*宏*。信号量必须事先通过调用xSemaphoreCreateBinary（），xSemaphoreCreateMutex（）或xSemaphoreCreateCounting（）来创建。

不得从ISR使用此功能。有关可以从ISR使用的替代方法，请参见xSemaphoreGiveFromISR（）。

也不得在使用xSemaphoreCreateRecursiveMutex（）创建的信号量上使用此宏。

**参数：**

|  |  |
| --- | --- |
| *x信号量* | 释放信号量的句柄。这是创建信号量时返回的句柄。 |

**返回值：**

如果已释放信号，则为pdTRUE。如果发生错误，则为pdFALSE。信号量是使用队列实现的。如果队列上没有空间可发送消息，则可能会发生错误-指示未正确正确获取信号量。

**用法示例：**

**SemaphoreHandle\_t xSemaphore = NULL;**

**void vATask( void \* pvParameters )**

**{**

**// Create the semaphore to guard a shared resource. As we are using**

**// the semaphore for mutual exclusion we create a mutex semaphore**

**// rather than a binary semaphore.**

**xSemaphore = xSemaphoreCreateMutex();**

**if( xSemaphore != NULL )**

**{**

**if( xSemaphoreGive( xSemaphore ) != pdTRUE )**

**{**

**// We would expect this call to fail because we cannot give**

**// a semaphore without first "taking" it!**

**}**

**// Obtain the semaphore - don't block if the semaphore is not**

**// immediately available.**

**if( xSemaphoreTake( xSemaphore, ( TickType\_t ) 0 ) )**

**{**

**// We now have the semaphore and can access the shared resource.**

**// ...**

**// We have finished accessing the shared resource so can free the**

**// semaphore.**

**if( xSemaphoreGive( xSemaphore ) != pdTRUE )**

**{**

**// We would not expect this call to fail because we must have**

**// obtained the semaphore to get here.**

**}**

**}**

**}**

**}**

# xSemaphoreGiveRecursive [[Semaphores](https://www.freertos.org/a00113.html)]

semphr. h

**xSemaphoreGiveRecursive( SemaphoreHandle\_t xMutex )**

*宏*以递归方式释放或“赋予”互斥体类型的信号量。互斥锁必须先前已通过调用xSemaphoreCreateRecursiveMutex（）创建。

必须在FreeRTOSConfig.h中将configUSE\_RECURSIVE\_MUTEXES设置为1，此宏才可用。

不得在使用xSemaphoreCreateMutex（）创建的互斥锁上使用此宏。

所有者可以反复“递归”使用递归使用的互斥锁。在所有者为每个成功的“获取”请求调用xSemaphoreGiveRecursive（）之前，互斥量不再可用。例如，如果一个任务成功“获取”了相同的互斥体5次，则该互斥体将对其他任何任务不可用，直到它也已将互斥体恰好“给予”了5次。

**参数：**

|  |  |
| --- | --- |
| *xMutex* | 互斥锁被释放或“给定”的句柄。这是xSemaphoreCreateRecursiveMutex（）返回的句柄。 |

**返回值：**

如果成功提供了信号量，则为pdTRUE。

**用法示例：**

**SemaphoreHandle\_t xMutex = NULL;**

**// A task that creates a mutex.**

**void vATask( void \* pvParameters )**

**{**

**// Create the mutex to guard a shared resource.**

**xMutex = xSemaphoreCreateRecursiveMutex();**

**}**

**// A task that uses the mutex.**

**void vAnotherTask( void \* pvParameters )**

**{**

**// ... Do other things.**

**if( xMutex != NULL )**

**{**

**// See if we can obtain the mutex. If the mutex is not available**

**// wait 10 ticks to see if it becomes free.**

**if( xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 ) == pdTRUE )**

**{**

**// We were able to obtain the mutex and can now access the**

**// shared resource.**

**// ...**

**// For some reason due to the nature of the code further calls to**

**// xSemaphoreTakeRecursive() are made on the same mutex. In real**

**// code these would not be just sequential calls as this would make**

**// no sense. Instead the calls are likely to be buried inside**

**// a more complex call structure.**

**xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 );**

**xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 );**

**// The mutex has now been 'taken' three times, so will not be**

**// available to another task until it has also been given back**

**// three times. Again it is unlikely that real code would have**

**// these calls sequentially, it would be more likely that the calls**

**// to xSemaphoreGiveRecursive() would be called as a call stack**

**// unwound. This is just for demonstrative purposes.**

**xSemaphoreGiveRecursive( xMutex );**

**xSemaphoreGiveRecursive( xMutex );**

**xSemaphoreGiveRecursive( xMutex );**

**// Now the mutex can be taken by other tasks.**

**}**

**else**

**{**

**// We could not obtain the mutex and can therefore not access**

**// the shared resource safely.**

**}**

**}**

**}**

# xSemaphoreGiveFromISR [[信号量](https://www.freertos.org/a00113.html)]

|  |
| --- |
| [**提示：在许多使用情况下，使用直接任务通知而不是信号量更快，更高效地使用内存**](https://www.freertos.org/RTOS-task-notifications.html) |

semphr. h

**xSemaphoreGiveFromISR**

**(**

**SemaphoreHandle\_t xSemaphore,**

**signed BaseType\_t \*pxHigherPriorityTaskWoken**

**)**

释放信号量的宏。信号量必须事先通过调用xSemaphoreCreateBinary（）或xSemaphoreCreateCounting（）来创建。

互斥类型信号量（通过调用xSemaphoreCreateMutex（）创建的信号量）不得与此宏一起使用。

可以从ISR使用此宏。

**参数：**

|  |  |
| --- | --- |
| x信号量 | 释放信号量的句柄。这是创建信号量时返回的句柄。 |
| pxHigherPriorityTaskWoken | 如果提供信号量导致任务取消阻止，并且未阻止任务的优先级高于当前运行的任务，则xSemaphoreGiveFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。如果xSemaphoreGiveFromISR（）将此值设置为pdTRUE，则应在退出中断之前请求上下文切换。  从FreeRTOS V7.3.0开始，pxHigherPriorityTaskWoken是可选参数，可以设置为NULL。 |

**返回值：**

如果成功提供了信号量，则为pdTRUE，否则为errQUEUE\_FULL。

**用法示例：**

请注意，以下显示的功能通常可以通过使用[直接任务通知](https://www.freertos.org/RTOS-task-notifications.html) 来代替信号量[来](https://www.freertos.org/RTOS-task-notifications.html)以更有效的方式实现。

**#define LONG\_TIME 0xffff**

**#define TICKS\_TO\_WAIT 10**

**SemaphoreHandle\_t xSemaphore = NULL;**

**/\* Repetitive task. \*/**

**void vATask( void \* pvParameters )**

**{**

**/\* We are using the semaphore for synchronisation so we create a binary**

**semaphore rather than a mutex. We must make sure that the interrupt**

**does not attempt to use the semaphore before it is created! \*/**

**xSemaphore = xSemaphoreCreateBinary();**

**for( ;; )**

**{**

**/\* We want this task to run every 10 ticks of a timer. The semaphore**

**was created before this task was started.**

**Block waiting for the semaphore to become available. \*/**

**if( xSemaphoreTake( xSemaphore, LONG\_TIME ) == pdTRUE )**

**{**

**/\* It is time to execute. \*/**

**...**

**/\* We have finished our task. Return to the top of the loop where**

**we will block on the semaphore until it is time to execute**

**again. Note when using the semaphore for synchronisation with an**

**ISR in this manner there is no need to 'give' the semaphore**

**back. \*/**

**}**

**}**

**}**

**/\* Timer ISR \*/**

**void vTimerISR( void \* pvParameters )**

**{**

**static unsigned char ucLocalTickCount = 0;**

**static signed BaseType\_t xHigherPriorityTaskWoken;**

**/\* A timer tick has occurred. \*/**

**... Do other time functions.**

**/\* Is it time for vATask() to run? \*/**

**xHigherPriorityTaskWoken = pdFALSE;**

**ucLocalTickCount++;**

**if( ucLocalTickCount >= TICKS\_TO\_WAIT )**

**{**

**/\* Unblock the task by releasing the semaphore. \*/**

**xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );**

**/\* Reset the count so we release the semaphore again in 10 ticks**

**time. \*/**

**ucLocalTickCount = 0;**

**}**

**/\* If xHigherPriorityTaskWoken was set to true you**

**we should yield. The actual macro used here is**

**port specific. \*/**

**portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

# 软件计时器 [ [API](https://www.freertos.org/a00106.html) ]

## FreeRTOS软件计时器API函数

* [**xTimerCreate**](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)
* [**xTimerCreateStatic**](https://www.freertos.org/xTimerCreateStatic.html)
* [**xTimerIsTimerActive**](https://www.freertos.org/FreeRTOS-timers-xTimerIsTimerActive.html)
* [**pvTimerGetTimerID**](https://www.freertos.org/FreeRTOS-timers-pvTimerGetTimerID.html)
* [**pcTimerGetName**](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html#pcTimerGetName)
* [**vTimerSetReloadMode**](https://www.freertos.org/FreeRTOS-Timers-vTimerSetReloadMode.html)
* [**xTimerStart**](https://www.freertos.org/FreeRTOS-timers-xTimerStart.html)
* [**xTimerStop**](https://www.freertos.org/FreeRTOS-timers-xTimerStop.html)
* [**xTimerChangePeriod**](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriod.html)
* [**xTimerDelete**](https://www.freertos.org/FreeRTOS-timers-xTimerDelete.html)
* [**xTimerReset**](https://www.freertos.org/FreeRTOS-timers-xTimerReset.html)
* [**xTimerStartFromISR**](https://www.freertos.org/FreeRTOS-timers-xTimerStartFromISR.html)
* [**xTimerStopFromISR**](https://www.freertos.org/FreeRTOS-timers-xTimerStopFromISR.html)
* [**xTimerChangePeriodFromISR**](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriodFromISR.html)
* [**xTimerResetFromISR**](https://www.freertos.org/FreeRTOS-timers-xTimerResetFromISR.html)
* [**pvTimerGetTimerID**](https://www.freertos.org/FreeRTOS-timers-pvTimerGetTimerID.html)
* [**vTimerSetTimerID**](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html)
* [**xTimerGetTimerDaemonTaskHandle**](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html#xTimerGetTimerDaemonTaskHandle)
* [**xTimerPendFunctionCall**](https://www.freertos.org/xTimerPendFunctionCall.html)
* [**xTimerPendFunctionCallFromISR**](https://www.freertos.org/xTimerPendFunctionCallFromISR.html)
* [**pcTimerGetName**](https://www.freertos.org/FreeRTOS-timers-pcTimerGetName.html)
* [**xTimerGetPeriod**](https://www.freertos.org/FreeRTOS-timers-xTimerGetPeriod.html)
* [**xTimerGetExpiryTime**](https://www.freertos.org/FreeRTOS-timers-xTimerGetExpiryTime.html)

### xTimerGetTimerDaemonTaskHandle

计时器

**TaskHandle\_t xTimerGetTimerDaemonTaskHandle（void）;**

**返回值：**

返回与软件计时器守护程序（或服务）任务关联的任务句柄。如果在FreeRTOSConfig.h中将configUSE\_TIMERS设置为1，则在启动RTOS调度程序时将自动创建计时器守护程序任务。

### pcTimerGetName

计时器

**const char \* pcTimerGetName（TimerHandle\_t xTimer）**

**参数：**

返回创建计时器时分配给计时器的可读名称。

|  |  |
| --- | --- |
| 计时器 | 查询计时器的句柄。 |

**返回值：**

指向计时器名称的指针，该名称是标准的以NULL结尾的C字符串。

# xTimerCreate [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**TimerHandle\_t xTimerCreate**

**( const char \* const pcTimerName,**

**const TickType\_t xTimerPeriod,**

**const UBaseType\_t uxAutoReload,**

**void \* const pvTimerID,**

**TimerCallbackFunction\_t pxCallbackFunction );**

创建一个新的[软件计时器](https://www.freertos.org/RTOS-software-timer.html)实例，并返回一个可以引用计时器的句柄。

要使此RTOS API函数可用：

1. 在FreeRTOSConfig.h中 configUSE\_TIMERS和[configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)必须都设置为1（configSUPPORT\_DYNAMIC\_ALLOCATION也可以保持未定义状态，在这种情况下它将默认为1）。
2. FreeRTOS / Source / timers.c C源文件必须包含在构建中。

每个软件计时器都需要少量的RAM，用于保持计时器的状态。如果使用xTimerCreate（）创建了计时器，则此RAM是从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配的。如果使用[xTimerCreateStatic（）](https://www.freertos.org/xTimerCreateStatic.html)创建了软件计时器 ，则应用程序[编写器](https://www.freertos.org/xTimerCreateStatic.html)将提供RAM，这需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

计时器处于休眠状态。所述[xTimerStart（）](https://www.freertos.org/FreeRTOS-timers-xTimerStart.html)， [xTimerReset（）](https://www.freertos.org/FreeRTOS-timers-xTimerReset.html)， [xTimerStartFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerStartFromISR.html)， [xTimerResetFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerResetFromISR.html)， [xTimerChangePeriod（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriod.html)和 [xTimerChangePeriodFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriodFromISR.html) API函数都可以被用于转换的定时器成为激活状态。

**参数：**

|  |  |
| --- | --- |
| pcTimerName | 分配给计时器的可读文本名称。这样做纯粹是为了协助调试。RTOS内核本身仅通过其句柄引用计时器，而从未通过其名称引用计时器。 |
| xTimerPeriod | 计时器的时间段。周期以刻度为单位，宏pdMS\_TO\_TICKS（）可用于将以毫秒为单位的时间转换为以刻度为单位的时间。例如，如果计时器必须在100个滴答之后过期，则只需将xTimerPeriod设置为100。或者，如果计时器必须在500ms之后过期，则将xTimerPeriod设置为pdMS\_TO\_TICKS（500）。仅当[configTICK\_RATE\_HZ](https://www.freertos.org/a00110.html" \l "configTICK_RATE_HZ)小于或等于1000时，才能使用pdMS\_TO\_TICKS（） 。计时器周期必须大于0。 |
| uxAutoReload | 如果将uxAutoReload设置为pdTRUE，则计时器将以xTimerPeriod参数设置的频率重复终止。如果将uxAutoReload设置为pdFALSE，则计时器将是一次触发，并在其到期后进入休眠状态。 |
| pvTimerID | 分配给正在创建的计时器的标识符。通常，在将同一回调函数分配给多个计时器时，将在计时器回调函数中使用它来识别哪个计时器到期，或者与[vTimerSetTimerID（）](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html) 和[pvTimerGetTimerID（）](https://www.freertos.org/FreeRTOS-timers-pvTimerGetTimerID.html) API函数一起使用，以在 [两次](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html)调用计时器之间保存值回调函数。 |
| pxCallbackFunction | 计时器到期时要调用的函数。回调函数必须具有TimerCallbackFunction\_t定义的原型，即：  **void vCallbackFunction（TimerHandle\_t xTimer）;**  . |

**返回值：**

如果成功创建了计时器，则返回新创建的计时器的句柄。如果由于剩余的FreeRTOS堆不足而无法分配计时器结构而无法创建计时器，则返回NULL。

**用法示例：**

**#define NUM\_TIMERS 5**

**/\* An array to hold handles to the created timers. \*/**

**TimerHandle\_t xTimers[ NUM\_TIMERS ];**

**/\* Define a callback function that will be used by multiple timer**

**instances. The callback function does nothing but count the number**

**of times the associated timer expires, and stop the timer once the**

**timer has expired 10 times. The count is saved as the ID of the**

**timer. \*/**

**void vTimerCallback( TimerHandle\_t xTimer )**

**{**

**const uint32\_t ulMaxExpiryCountBeforeStopping = 10;**

**uint32\_t ulCount;**

**/\* Optionally do something if the pxTimer parameter is NULL. \*/**

**configASSERT( xTimer );**

**/\* The number of times this timer has expired is saved as the**

**timer's ID. Obtain the count. \*/**

**ulCount = ( uint32\_t )** [**pvTimerGetTimerID**](https://www.freertos.org/FreeRTOS-timers-pvTimerGetTimerID.html)**( xTimer );**

**/\* Increment the count, then test to see if the timer has expired**

**ulMaxExpiryCountBeforeStopping yet. \*/**

**ulCount++;**

**/\* If the timer has expired 10 times then stop it from running. \*/**

**if( ulCount >= ulMaxExpiryCountBeforeStopping )**

**{**

**/\* Do not use a block time if calling a timer API function**

**from a timer callback function, as doing so could cause a**

**deadlock! \*/**

[**xTimerStop**](https://www.freertos.org/FreeRTOS-timers-xTimerStop.html)**( pxTimer, 0 );**

**}**

**else**

**{**

**/\* Store the incremented count back into the timer's ID field**

**so it can be read back again the next time this software timer**

**expires. \*/**

[**vTimerSetTimerID**](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html)**( xTimer, ( void \* ) ulCount );**

**}**

**}**

**void main( void )**

**{**

**long x;**

**/\* Create then start some timers. Starting the timers before**

**the RTOS scheduler has been started means the timers will start**

**running immediately that the RTOS scheduler starts. \*/**

**for( x = 0; x < NUM\_TIMERS; x++ )**

**{**

**xTimers[ x ] = xTimerCreate**

**( /\* Just a text name, not used by the RTOS**

**kernel. \*/**

**"Timer",**

**/\* The timer period in ticks, must be**

**greater than 0. \*/**

**( 100 \* x ) + 100,**

**/\* The timers will auto-reload themselves**

**when they expire. \*/**

**pdTRUE,**

**/\* The ID is used to store a count of the**

**number of times the timer has expired, which**

**is initialised to 0. \*/**

**( void \* ) 0,**

**/\* Each timer calls the same callback when**

**it expires. \*/**

**vTimerCallback**

**);**

**if( xTimers[ x ] == NULL )**

**{**

**/\* The timer was not created. \*/**

**}**

**else**

**{**

**/\* Start the timer. No block time is specified, and**

**even if one was it would be ignored because the RTOS**

**scheduler has not yet been started. \*/**

**if(** [**xTimerStart**](https://www.freertos.org/FreeRTOS-timers-xTimerStart.html)**( xTimers[ x ], 0 ) != pdPASS )**

**{**

**/\* The timer could not be set into the Active**

**state. \*/**

**}**

**}**

**}**

**/\* ...**

**Create tasks here.**

**... \*/**

**/\* Starting the RTOS scheduler will start the timers running**

**as they have already been set into the active state. \*/**

**vTaskStartScheduler();**

**/\* Should not reach here. \*/**

**for( ;; );**

**}**

# xTimerCreateStatic [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**TimerHandle\_t xTimerCreateStatic**

**( const char \* const pcTimerName,**

**const TickType\_t xTimerPeriod,**

**const UBaseType\_t uxAutoReload,**

**void \* const pvTimerID,**

**TimerCallbackFunction\_t pxCallbackFunction**

**StaticTimer\_t \*pxTimerBuffer );**

创建一个新的[软件计时器](https://www.freertos.org/RTOS-software-timer.html)实例，并返回一个可以引用计时器的句柄。

要使此RTOS API函数可用：

1. 在FreeRTOSConfig.h中，必须将 configUSE\_TIMERS和[configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_STATIC_ALLOCATION)都设置为1。
2. FreeRTOS / Source / timer.c C源文件必须包含在构建中。

每个软件计时器都需要少量的RAM，用于保持计时器的状态。如果使用[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)创建计时器，则 所需的RAM将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配。如果使用xTimerCreateStatic（）创建了软件计时器，则应用程序编写器将提供RAM，这需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

计时器处于休眠状态。所述[xTimerStart（）](https://www.freertos.org/FreeRTOS-timers-xTimerStart.html)， [xTimerReset（）](https://www.freertos.org/FreeRTOS-timers-xTimerReset.html)， [xTimerStartFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerStartFromISR.html)， [xTimerResetFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerResetFromISR.html)， [xTimerChangePeriod（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriod.html)和 [xTimerChangePeriodFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriodFromISR.html) API函数都可以被用于转换的定时器成为激活状态。

**参数：**

|  |  |
| --- | --- |
| pcTimerName | 分配给计时器的可读文本名称。这样做纯粹是为了协助调试。RTOS内核本身仅通过其句柄引用计时器，而从未通过其名称引用计时器。 |
| xTimerPeriod | 计时器的时间段。周期以刻度为单位指定，宏pdMS\_TO\_TICKS（）可用于将以毫秒为单位的时间转换为以刻度为单位的时间。例如，如果计时器必须在100个滴答之后过期，则只需将xTimerPeriod设置为100。或者，如果计时器必须在500ms之后过期，则将xTimerPeriod设置为pdMS\_TO\_TICKS（500）。仅当[configTICK\_RATE\_HZ](https://www.freertos.org/a00110.html" \l "configTICK_RATE_HZ)小于或等于1000时，才能使用pdMS\_TO\_TICKS（） 。计时器周期必须大于0。 |
| uxAutoReload | 如果将uxAutoReload设置为pdTRUE，则计时器将以xTimerPeriod参数设置的频率重复终止。如果将uxAutoReload设置为pdFALSE，则计时器将是一次触发，并在其到期后进入休眠状态。 |
| pvTimerID | 分配给正在创建的计时器的标识符。通常，在将同一回调函数分配给多个计时器时，将在计时器回调函数中使用它来识别哪个计时器到期，或者与[vTimerSetTimerID（）](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html) 和[pvTimerGetTimerID（）](https://www.freertos.org/FreeRTOS-timers-pvTimerGetTimerID.html) API函数一起使用，以在 [两次](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html)调用计时器之间保存值回调函数。 |
| pxCallbackFunction | 计时器到期时要调用的函数。回调函数必须具有TimerCallbackFunction\_t定义的原型，即：  **void vCallbackFunction（TimerHandle\_t xTimer）;** |
| pxTimerBuffer | 必须指向StaticTimer\_t类型的变量，然后将其用于保存计时器的状态。 |

**返回值：**

如果成功创建了计时器，则返回新创建的计时器的句柄。如果pxTimerBuffer为NULL，则不会创建计时器，并且返回NULL。

**用法示例：**

**pxTimerBuffer #define NUM\_TIMERS 5**

**/\* An array to hold handles to the created timers. \*/**

**TimerHandle\_t xTimers[ NUM\_TIMERS ];**

**/\* An array of StaticTimer\_t structures, which are used to store**

**the state of each created timer. \*/**

**StaticTimer\_t xTimerBuffers[ NUM\_TIMERS ];**

**/\* Define a callback function that will be used by multiple timer**

**instances. The callback function does nothing but count the number**

**of times the associated timer expires, and stop the timer once the**

**timer has expired 10 times. The count is saved as the ID of the**

**timer. \*/**

**void vTimerCallback( TimerHandle\_t xTimer )**

**{**

**const uint32\_t ulMaxExpiryCountBeforeStopping = 10;**

**uint32\_t ulCount;**

**/\* Optionally do something if the pxTimer parameter is NULL. \*/**

**configASSERT( pxTimer );**

**/\* The number of times this timer has expired is saved as the**

**timer's ID. Obtain the count. \*/**

**ulCount = ( uint32\_t )** [**pvTimerGetTimerID**](https://www.freertos.org/FreeRTOS-timers-pvTimerGetTimerID.html)**( xTimer );**

**/\* Increment the count, then test to see if the timer has expired**

**ulMaxExpiryCountBeforeStopping yet. \*/**

**ulCount++;**

**/\* If the timer has expired 10 times then stop it from running. \*/**

**if( ulCount >= ulMaxExpiryCountBeforeStopping )**

**{**

**/\* Do not use a block time if calling a timer API function**

**from a timer callback function, as doing so could cause a**

**deadlock! \*/**

[**xTimerStop**](https://www.freertos.org/FreeRTOS-timers-xTimerStop.html)**( pxTimer, 0 );**

**}**

**else**

**{**

**/\* Store the incremented count back into the timer's ID field**

**so it can be read back again the next time this software timer**

**expires. \*/**

[**vTimerSetTimerID**](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html)**( xTimer, ( void \* ) ulCount );**

**}**

**}**

**void main( void )**

**{**

**long x;**

**/\* Create then start some timers. Starting the timers before**

**the RTOS scheduler has been started means the timers will start**

**running immediately that the RTOS scheduler starts. \*/**

**for( x = 0; x < NUM\_TIMERS; x++ )**

**{**

**xTimers[ x ] = xTimerCreateStatic**

**( /\* Just a text name, not used by the RTOS**

**kernel. \*/**

**"Timer",**

**/\* The timer period in ticks, must be**

**greater than 0. \*/**

**( 100 \* x ) + 100,**

**/\* The timers will auto-reload themselves**

**when they expire. \*/**

**pdTRUE,**

**/\* The ID is used to store a count of the**

**number of times the timer has expired, which**

**is initialised to 0. \*/**

**( void \* ) 0,**

**/\* Each timer calls the same callback when**

**it expires. \*/**

**vTimerCallback,**

**/\* Pass in the address of a StaticTimer\_t**

**variable, which will hold the data associated with**

**the timer being created. \*/**

**&( xTimerBuffers[ x ] );**

**);**

**if( xTimers[ x ] == NULL )**

**{**

**/\* The timer was not created. \*/**

**}**

**else**

**{**

**/\* Start the timer. No block time is specified, and**

**even if one was it would be ignored because the RTOS**

**scheduler has not yet been started. \*/**

**if(** [**xTimerStart**](https://www.freertos.org/FreeRTOS-timers-xTimerStart.html)**( xTimers[ x ], 0 ) != pdPASS )**

**{**

**/\* The timer could not be set into the Active**

**state. \*/**

**}**

**}**

**}**

**/\* ...**

**Create tasks here.**

**... \*/**

**/\* Starting the RTOS scheduler will start the timers running**

**as they have already been set into the active state. \*/**

**vTaskStartScheduler();**

**/\* Should not reach here. \*/**

**for( ;; );**

**}**

# xTimerIsTimerActive [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerIsTimerActive( TimerHandle\_t xTimer );**

查询[软件计时器](https://www.freertos.org/RTOS-software-timer.html)以查看其是否处于活动状态或休眠状态。

在以下情况下，计时器将处于休眠状态：

1. 它已创建但尚未启动，或者
2. 这是一个过期的单发计时器，尚未重启。

计时器处于休眠状态。所述[xTimerStart（）](https://www.freertos.org/FreeRTOS-timers-xTimerStart.html)， [xTimerReset（）](https://www.freertos.org/FreeRTOS-timers-xTimerReset.html)， [xTimerStartFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerStartFromISR.html)， [xTimerResetFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerResetFromISR.html)， [xTimerChangePeriod（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriod.html)和 [xTimerChangePeriodFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriodFromISR.html) API函数都可以被用于转换的定时器成为激活状态。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 正在查询计时器。 |

**返回值：**

如果计时器处于休眠状态，则将返回pdFALSE。如果计时器处于活动状态，则将返回pdFALSE以外的值。

**用法示例：**

**/\* This function assumes xTimer has already**

**been created. \*/**

**void vAFunction( TimerHandle\_t xTimer )**

**{**

**/\* or more simply and equivalently**

**"if( xTimerIsTimerActive( xTimer ) )" \*/**

**if( xTimerIsTimerActive( xTimer ) != pdFALSE )**

**{**

**/\* xTimer is active, do something. \*/**

**}**

**else**

**{**

**/\* xTimer is not active, do something else. \*/**

**}**

**}**

# xTimerStart [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerStart( TimerHandle\_t xTimer,**

**TickType\_t xBlockTime );**

计时器服务/守护程序任务提供了[软件计时器](https://www.freertos.org/RTOS-software-timer.html)功能。许多公共的FreeRTOS计时器API函数通过称为计时器命令队列的队列将命令发送到计时器服务任务。定时器命令队列是RTOS内核本身专用的，应用程序代码不能直接访问它。计时器命令队列的长度由configTIMER\_QUEUE\_LENGTH配置常量设置。

xTimerStart（）启动一个计时器，该计时器先前是使用[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html) API函数创建的 。如果计时器已经启动并且已经处于活动状态，则xTimerStart（）具有与[xTimerReset（）](https://www.freertos.org/FreeRTOS-timers-xTimerReset.html) API函数等效的功能。

启动计时器可确保计时器处于活动状态。如果同时没有停止，删除或重置计时器，则与x计时器关联的回调函数将在调用xTimerStart（）之后被称为“ n”滴答，其中“ n”是计时器定义的时间段。

在启动RTOS调度程序之前调用xTimerStart（）是有效的，但是完成此操作后，计时器将不会真正启动，直到启动RTOS调度程序为止，并且计时器的到期时间将与RTOS调度程序的启动时间有关，与调用xTimerStart（）的时间无关。

必须将configUSE\_TIMERS配置常量设置为1，xTimerStart（）才可用。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 计时器的句柄正在启动/重新启动。 |
| xBlockTime | 指定在调用xTimerStart（）时，如果队列已满，则调用任务应保持在“阻塞”状态以等待开始命令成功发送到计时器命令队列的时间（以秒为单位）。如果在启动RTOS调度程序之前调用xTimerStart（），则xBlockTime将被忽略。 |

**返回值：**

如果即使经过xBlockTime刻度后仍无法将启动命令发送到计时器命令队列，则将返回pdFAIL。如果命令已成功发送到计时器命令队列，则将返回pdPASS。实际处理命令的时间将取决于计时器服务/守护程序任务相对于系统中其他任务的优先级，尽管计时器的到期时间与实际调用xTimerStart（）有关。计时器服务/守护程序任务优先级由configTIMER\_TASK\_PRIORITY配置常量设置。

**用法示例：**

请参阅[xTimerCreate（）文档页面](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)上的[示例](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)。

# xTimerStop [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerStop( TimerHandle\_t xTimer,**

**TickType\_t xBlockTime );**

计时器服务/守护程序任务提供了[软件计时器](https://www.freertos.org/RTOS-software-timer.html)功能。许多公共的FreeRTOS计时器API函数通过称为计时器命令队列的队列将命令发送到计时器服务任务。定时器命令队列是RTOS内核本身专用的，应用程序代码不能直接访问它。计时器命令队列的长度由configTIMER\_QUEUE\_LENGTH配置常量设置。

xTimerStop（）停止使用 [xTimerStart（）](https://www.freertos.org/FreeRTOS-timers-xTimerStart.html)， [xTimerReset（）](https://www.freertos.org/FreeRTOS-timers-xTimerReset.html)， [xTimerStartFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerStartFromISR.html)， [xTimerResetFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerResetFromISR.html)， [xTimerChangePeriod（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriod.html)和 [xTimerChangePeriodFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriodFromISR.html) API函数之一[启动的计时器](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriodFromISR.html)。

停止计时器可确保计时器未处于活动状态。

必须将configUSE\_TIMERS配置常量设置为1，xTimerStop（）才可用。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 计时器的句柄被停止。 |
| xBlockTime | 指定在调用xTimerStop（）时，如果队列已满，则调用任务应保持在Blocked状态的时间，以等待将stop命令成功发送到计时器命令队列。如果在启动RTOS调度程序之前调用了xTimerStop（），则将忽略xBlockTime。 |

**返回值：**

如果即使经过xBlockTime刻度后仍无法将stop命令发送到计时器命令队列，则将返回pdFAIL。如果命令已成功发送到计时器命令队列，则将返回pdPASS。实际处理命令的时间将取决于计时器服务/守护程序任务相对于系统中其他任务的优先级。计时器服务/守护程序任务优先级由configTIMER\_TASK\_PRIORITY配置常量设置。

**用法示例：**

请参阅[xTimerCreate（）文档页面](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)上的[示例](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)。

# xTimerChangePeriod [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerChangePeriod( TimerHandle\_t xTimer,**

**TickType\_t xNewPeriod,**

**TickType\_t xBlockTime );**

计时器服务/守护程序任务提供了[软件计时器](https://www.freertos.org/RTOS-software-timer.html)功能。许多公共的FreeRTOS计时器API函数通过称为计时器命令队列的队列将命令发送到计时器服务任务。定时器命令队列是RTOS内核本身专用的，应用程序代码不能直接访问它。计时器命令队列的长度由configTIMER\_QUEUE\_LENGTH配置常量设置。

xTimerChangePeriod（）更改以前使用[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html) API函数创建的计时器的时间段。

可以调用xTimerChangePeriod（）来更改活动或休眠状态计时器的周期。更改休眠计时器的时间也会启动计时器。

必须将configUSE\_TIMERS配置常数设置为1，xTimerChangePeriod（）才可用。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 周期已更改的计时器的句柄。 |
| xNewPeriod | xTimer的新时期。计时器周期以滴答周期指定，因此常数portTICK\_PERIOD\_MS可用于转换以毫秒为单位指定的时间。例如，如果计时器必须在100个滴答之后到期，则xNewPeriod应该设置为100。或者，如果计时器必须在500ms之后过期，则可以将xNewPeriod设置为（500 / portTICK\_PERIOD\_MS），前提是configTICK\_RATE\_HZ小于或等于1000。 。 |
| xBlockTime | 指定在调用xTimerChangePeriod（）时，如果队列已满，则调用任务应保持在Blocked状态的时间，以等待更改周期命令成功发送到计时器命令队列。如果在RTOS调度程序启动之前调用了xTimerChangePeriod（），则xBlockTime将被忽略。 |

**返回值：**

如果即使经过xBlockTime刻度后仍无法将更改周期命令发送到计时器命令队列，则将返回pdFAIL。如果命令已成功发送到计时器命令队列，则将返回pdPASS。实际处理命令的时间将取决于计时器服务/守护程序任务相对于系统中其他任务的优先级。计时器服务/守护程序任务优先级由configTIMER\_TASK\_PRIORITY配置常量设置。

**用法示例：**

**/\* This function assumes xTimer has already been created. If the timer**

**referenced by xTimer is already active when it is called, then the timer**

**is deleted. If the timer referenced by xTimer is not active when it is**

**called, then the period of the timer is set to 500ms and the timer is**

**started. \*/**

**void vAFunction( TimerHandle\_t xTimer )**

**{**

**/\* or more simply and equivalently**

**"if( xTimerIsTimerActive( xTimer ) )" \*/**

**if( xTimerIsTimerActive( xTimer ) != pdFALSE )**

**{**

**/\* xTimer is already active - delete it. \*/**

**xTimerDelete( xTimer );**

**}**

**else**

**{**

**/\* xTimer is not active, change its period to 500ms. This will also**

**cause the timer to start. Block for a maximum of 100 ticks if the**

**change period command cannot immediately be sent to the timer**

**command queue. \*/**

**if( xTimerChangePeriod( xTimer, 500 / portTICK\_PERIOD\_MS, 100 )**

**== pdPASS )**

**{**

**/\* The command was successfully sent. \*/**

**}**

**else**

**{**

**/\* The command could not be sent, even after waiting for 100 ticks**

**to pass. Take appropriate action here. \*/**

**}**

**}**

**}**

# xTimerDelete [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerDelete( TimerHandle\_t xTimer,**

**TickType\_t xBlockTime );**

计时器服务/守护程序任务提供了[软件计时器](https://www.freertos.org/RTOS-software-timer.html)功能。许多公共的FreeRTOS计时器API函数通过称为计时器命令队列的队列将命令发送到计时器服务任务。定时器命令队列是RTOS内核本身专用的，应用程序代码不能直接访问它。计时器命令队列的长度由configTIMER\_QUEUE\_LENGTH配置常量设置。

xTimerDelete（）删除以前使用[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html) API函数创建的计时器 。

必须将configUSE\_TIMERS配置常量设置为1，xTimerDelete（）才可用。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 计时器的句柄被删除。 |
| xBlockTime | 指定在调用xTimerDelete（）时，如果队列已满，则调用任务应保持在Blocked状态的时间，以等待将delete命令成功发送到计时器命令队列。如果在RTOS调度程序启动之前调用了xTimerDelete（），则xBlockTime将被忽略。 |

**返回值：**

如果即使经过xBlockTime刻度后仍无法将删除命令发送到计时器命令队列，则将返回pdFAIL。如果命令已成功发送到计时器命令队列，则将返回pdPASS。实际处理命令的时间将取决于计时器服务/守护程序任务相对于系统中其他任务的优先级。计时器服务/守护程序任务优先级由configTIMER\_TASK\_PRIORITY配置常量设置。

**用法示例：**

请参阅[xTimerChangePeriod（）文档页面](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriod.html)上的[示例](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriod.html)。

# xTimerReset [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerReset( TimerHandle\_t xTimer,**

**TickType\_t xBlockTime );**

计时器服务/守护程序任务提供了[软件计时器](https://www.freertos.org/RTOS-software-timer.html)功能。许多公共的FreeRTOS计时器API函数通过称为计时器命令队列的队列将命令发送到计时器服务任务。定时器命令队列是RTOS内核本身专用的，应用程序代码不能直接访问它。计时器命令队列的长度由configTIMER\_QUEUE\_LENGTH配置常量设置。

xTimerReset（）重新启动以前使用[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html) API函数创建的计时器 。如果计时器已经启动并且已经处于活动状态，则xTimerReset（）将使计时器重新评估其到期时间，以使其与调用xTimerReset（）的时间有关。如果计时器处于休眠状态，则xTimerReset（）具有与[xTimerStart（）](https://www.freertos.org/FreeRTOS-timers-xTimerStart.html) API函数等效的功能。

重置计时器可确保计时器处于活动状态。如果同时没有停止，删除或重置计时器，则在调用xTimerReset（）之后，与计时器关联的回调函数将被称为“ n”滴答，其中“ n”是计时器定义的时间段。

在启动RTOS调度程序之前调用xTimerReset（）是有效的，但是完成此操作后，计时器将不会真正启动，直到启动RTOS调度程序为止，并且计时器的到期时间将与启动RTOS调度程序的时间有关，与调用xTimerReset（）的时间无关。

必须将configUSE\_TIMERS配置常量设置为1，xTimerReset（）才可用。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 重置/启动/重新启动计时器的句柄。 |
| xBlockTime | 指定在调用xTimerReset（）时，如果队列已满，则调用任务应保持在“阻塞”状态以等待重置命令成功发送到计时器命令队列的时间（以滴答为单位）。如果在启动RTOS调度程序之前调用了xTimerReset（），则将忽略xBlockTime。 |

**返回值：**

如果即使经过xBlockTime刻度后仍无法将重置命令发送到计时器命令队列，则将返回pdFAIL。如果命令已成功发送到计时器命令队列，则将返回pdPASS。实际处理命令的时间将取决于计时器服务/守护程序任务相对于系统中其他任务的优先级，尽管计时器的到期时间与实际调用xTimerReset（）有关。计时器服务/守护程序任务优先级由configTIMER\_TASK\_PRIORITY配置常量设置。

**用法示例：**

**/\* When a key is pressed, an LCD back-light is switched on. If 5 seconds pass**

**without a key being pressed, then the LCD back-light is switched off. In**

**this case, the timer is a one-shot timer. \*/**

**TimerHandle\_t xBacklightTimer = NULL;**

**/\* The callback function assigned to the one-shot timer. In this case the**

**parameter is not used. \*/**

**void vBacklightTimerCallback( TimerHandle\_t pxTimer )**

**{**

**/\* The timer expired, therefore 5 seconds must have passed since a key**

**was pressed. Switch off the LCD back-light. \*/**

**vSetBacklightState( BACKLIGHT\_OFF );**

**}**

**/\* The key press event handler. \*/**

**void vKeyPressEventHandler( char cKey )**

**{**

**/\* Ensure the LCD back-light is on, then reset the timer that is**

**responsible for turning the back-light off after 5 seconds of**

**key inactivity. Wait 10 ticks for the command to be successfully sent**

**if it cannot be sent immediately. \*/**

**vSetBacklightState( BACKLIGHT\_ON );**

**if( xTimerReset( xBacklightTimer, 10 ) != pdPASS )**

**{**

**/\* The reset command was not executed successfully. Take appropriate**

**action here. \*/**

**}**

**/\* Perform the rest of the key processing here. \*/**

**}**

**void main( void )**

**{**

**long x;**

**/\* Create then start the one-shot timer that is responsible for turning**

**the back-light off if no keys are pressed within a 5 second period. \*/**

**xBacklightTimer = xTimerCreate**

**(**

**/\* Just a text name, not used by the RTOS kernel. \*/**

**"BacklightTimer",**

**/\* The timer period in ticks. \*/**

**( 5000 / portTICK\_PERIOD\_MS),**

**/\* The timer is a one-shot timer. \*/**

**pdFALSE,**

**/\* The id is not used by the callback so can take any**

**value. \*/**

**0,**

**/\* The callback function that switches the LCD back-light**

**off. \*/**

**vBacklightTimerCallback**

**);**

**if( xBacklightTimer == NULL )**

**{**

**/\* The timer was not created. \*/**

**}**

**else**

**{**

**/\* Start the timer. No block time is specified, and even if one was**

**it would be ignored because the RTOS scheduler has not yet been**

**started. \*/**

**if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )**

**{**

**/\* The timer could not be set into the Active state. \*/**

**}**

**}**

**/\* ...**

**Create tasks here.**

**... \*/**

**/\* Starting the RTOS scheduler will start the timer running as it has**

**already been set into the active state. \*/**

**vTaskStartScheduler();**

**/\* Should not reach here. \*/**

**for( ;; );**

**}**

# xTimerStartFromISR [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerStartFromISR**

**(**

**TimerHandle\_t xTimer,**

**BaseType\_t \*pxHigherPriorityTaskWoken**

**);**

可以从中断服务程序中调用 的[xTimerStart（）](https://www.freertos.org/FreeRTOS-timers-xTimerStart.html)版本。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 计时器的句柄正在启动/重新启动。 |
| pxHigherPriorityTaskWoken | 计时器服务/守护程序任务的大部分时间都处于“阻塞”状态，等待消息到达计时器命令队列。调用xTimerStartFromISR（）会将消息写入计时器命令队列，因此有可能将计时器服务/守护程序任务转换为“已阻止”状态。如果调用xTimerStartFromISR（）导致计时器服务/守护程序任务退出“阻塞”状态，并且计时器服务/守护程序任务的优先级等于或大于当前正在执行的任务（被中断的任务），则\* pxHigherPriorityTaskWoken将获得在xTimerStartFromISR（）函数内部将其设置为pdTRUE。如果xTimerStartFromISR（）将此值设置为pdTRUE，则应在中断退出之前执行上下文切换。 |

**返回值：**

如果无法将启动命令发送到计时器命令队列，则将返回pdFAIL。如果命令已成功发送到计时器命令队列，则将返回pdPASS。实际处理命令的时间将取决于计时器服务/守护程序任务相对于系统中其他任务的优先级，尽管计时器的到期时间与实际调用xTimerStartFromISR（）的时间有关。计时器服务/守护程序任务优先级由configTIMER\_TASK\_PRIORITY配置常量设置。

**用法示例：**

**/\* This scenario assumes xBacklightTimer has already been created. When a**

**key is pressed, an LCD back-light is switched on. If 5 seconds pass**

**without a key being pressed, then the LCD back-light is switched off. In**

**this case, the timer is a one-shot timer, and unlike the example given for**

**the xTimerReset() function, the key press event handler is an interrupt**

**service routine. \*/**

**/\* The callback function assigned to the one-shot timer. In this case the**

**parameter is not used. \*/**

**void vBacklightTimerCallback( TimerHandle\_t pxTimer )**

**{**

**/\* The timer expired, therefore 5 seconds must have passed since a key**

**was pressed. Switch off the LCD back-light. \*/**

**vSetBacklightState( BACKLIGHT\_OFF );**

**}**

**/\* The key press interrupt service routine. \*/**

**void vKeyPressEventInterruptHandler( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**/\* Ensure the LCD back-light is on, then restart the timer that is**

**responsible for turning the back-light off after 5 seconds of**

**key inactivity. This is an interrupt service routine so can only**

**call FreeRTOS API functions that end in "FromISR". \*/**

**vSetBacklightState( BACKLIGHT\_ON );**

**/\* xTimerStartFromISR() or xTimerResetFromISR() could be called here**

**as both cause the timer to re-calculate its expiry time.**

**xHigherPriorityTaskWoken was initialised to pdFALSE when it was**

**declared (in this function). \*/**

**if( xTimerStartFromISR( xBacklightTimer,**

**&xHigherPriorityTaskWoken ) != pdPASS )**

**{**

**/\* The start command was not executed successfully. Take appropriate**

**action here. \*/**

**}**

**/\* Perform the rest of the key processing here. \*/**

**/\* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch**

**should be performed. The syntax required to perform a context switch**

**from inside an ISR varies from port to port, and from compiler to**

**compiler. Inspect the demos for the port you are using to find the**

**actual syntax required. \*/**

**if( xHigherPriorityTaskWoken != pdFALSE )**

**{**

**/\* Call the interrupt safe yield function here (actual function**

**depends on the FreeRTOS port being used). \*/**

**}**

**}**

# xTimerStopFromISR [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerStopFromISR**

**(**

**TimerHandle\_t xTimer,**

**BaseType\_t \*pxHigherPriorityTaskWoken**

**);**

可以从中断服务程序中调用 的[xTimerStop（）](https://www.freertos.org/FreeRTOS-timers-xTimerStop.html)版本。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 计时器的句柄被停止。 |
| pxHigherPriorityTaskWoken | 计时器服务/守护程序任务的大部分时间都处于“阻塞”状态，等待消息到达计时器命令队列。调用xTimerStopFromISR（）会将消息写入计时器命令队列，因此有可能将计时器服务/守护程序任务从“阻塞”状态中转换出来。如果调用xTimerStopFromISR（）导致计时器服务/守护程序任务退出“阻塞”状态，并且计时器服务/守护程序任务的优先级等于或大于当前正在执行的任务（被中断的任务），则\* pxHigherPriorityTaskWoken将获得在xTimerStopFromISR（）函数内部将其设置为pdTRUE。如果xTimerStopFromISR（）将此值设置为pdTRUE，则应在中断退出之前执行上下文切换。 |

**返回值：**

如果停止命令无法发送到计时器命令队列，则将返回pdFAIL。如果命令已成功发送到计时器命令队列，则将返回pdPASS。实际处理命令的时间将取决于计时器服务/守护程序任务相对于系统中其他任务的优先级。计时器服务/守护程序任务优先级由configTIMER\_TASK\_PRIORITY配置常量设置。

**用法示例：**

**/\* This scenario assumes xTimer has already been created and started. When**

**an interrupt occurs, the timer should be simply stopped. \*/**

**/\* The interrupt service routine that stops the timer. \*/**

**void vAnExampleInterruptServiceRoutine( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**/\* The interrupt has occurred - simply stop the timer.**

**xHigherPriorityTaskWoken was set to pdFALSE where it was defined**

**(within this function). As this is an interrupt service routine, only**

**FreeRTOS API functions that end in "FromISR" can be used. \*/**

**if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )**

**{**

**/\* The stop command was not executed successfully. Take appropriate**

**action here. \*/**

**}**

**/\* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch**

**should be performed. The syntax required to perform a context switch**

**from inside an ISR varies from port to port, and from compiler to**

**compiler. Inspect the demos for the port you are using to find the**

**actual syntax required. \*/**

**if( xHigherPriorityTaskWoken != pdFALSE )**

**{**

**/\* Call the interrupt safe yield function here (actual function**

**depends on the FreeRTOS port being used). \*/**

**}**

**}**

# xTimerChangePeriodFromISR [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerChangePeriodFromISR**

**(**

**TimerHandle\_t xTimer,**

**TickType\_t xNewPeriod,**

**BaseType\_t \*pxHigherPriorityTaskWoken**

**);**

可以从中断服务程序中调用 的[xTimerChangePeriod（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriod.html)版本。

**参数：**

|  |  |
| --- | --- |
| xTimer | 周期已更改 的[软件计时器](https://www.freertos.org/RTOS-software-timer.html)的句柄。 |
| xNewPeriod | xTimer的新时期。计时器周期以滴答周期指定，因此常数portTICK\_PERIOD\_MS可用于转换以毫秒为单位指定的时间。例如，如果计时器必须在100个滴答之后到期，则xNewPeriod应该设置为100。或者，如果计时器必须在500ms之后过期，则可以将xNewPeriod设置为（500 / portTICK\_PERIOD\_MS），前提是configTICK\_RATE\_HZ小于或等于1000。 。 |
| pxHigherPriorityTaskWoken | 计时器服务/守护程序任务的大部分时间都处于“阻塞”状态，等待消息到达计时器命令队列。调用xTimerChangePeriodFromISR（）会将消息写入计时器命令队列，因此有可能将计时器服务/守护程序任务从“阻塞”状态转换出来。如果调用xTimerChangePeriodFromISR（）导致计时器服务/守护程序任务退出“阻塞”状态，并且计时器服务/守护程序任务的优先级等于或大于当前正在执行的任务（被中断的任务），则\* pxHigherPriorityTaskWoken将获得在xTimerChangePeriodFromISR（）函数内部将其设置为pdTRUE。如果xTimerChangePeriodFromISR（）将此值设置为pdTRUE，则应在中断退出之前执行上下文切换。 |

**返回值：**

如果无法将更改计时器周期的命令发送到计时器命令队列，则将返回pdFAIL。如果命令已成功发送到计时器命令队列，则将返回pdPASS。实际处理命令的时间将取决于计时器服务/守护程序任务相对于系统中其他任务的优先级。计时器服务/守护程序任务优先级由configTIMER\_TASK\_PRIORITY配置常量设置。

**用法示例：**

**/\* This scenario assumes xTimer has already been created and started. When**

**an interrupt occurs, the period of xTimer should be changed to 500ms. \*/**

**/\* The interrupt service routine that changes the period of xTimer. \*/**

**void vAnExampleInterruptServiceRoutine( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**/\* The interrupt has occurred - change the period of xTimer to 500ms.**

**xHigherPriorityTaskWoken was set to pdFALSE where it was defined**

**(within this function). As this is an interrupt service routine, only**

**FreeRTOS API functions that end in "FromISR" can be used. \*/**

**if( xTimerChangePeriodFromISR( xTimer,**

**pdMS\_TO\_TICKS( 500 ),**

**&xHigherPriorityTaskWoken ) != pdPASS )**

**{**

**/\* The command to change the timers period was not executed**

**successfully. Take appropriate action here. \*/**

**}**

**/\* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch**

**should be performed. The syntax required to perform a context switch**

**from inside an ISR varies from port to port, and from compiler to**

**compiler. Inspect the demos for the port you are using to find the**

**actual syntax required. \*/**

**if( xHigherPriorityTaskWoken != pdFALSE )**

**{**

**/\* Call the interrupt safe yield function here (actual function**

**depends on the FreeRTOS port being used). \*/**

**}**

**}**

# xTimerResetFromISR [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerResetFromISR**

**(**

**TimerHandle\_t xTimer,**

**BaseType\_t \*pxHigherPriorityTaskWoken**

**);**

可以从中断服务程序中调用 的[xTimerReset（）](https://www.freertos.org/FreeRTOS-timers-xTimerReset.html)版本。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 要启动，重置或重新启动的计时器的句柄。 |
| pxHigherPriorityTaskWoken | 计时器服务/守护程序任务的大部分时间都处于“阻塞”状态，等待消息到达计时器命令队列。调用xTimerResetFromISR（）会将消息写入计时器命令队列，因此有可能将计时器服务/守护程序任务转换为“已阻止”状态。如果调用xTimerResetFromISR（）导致计时器服务/守护程序任务退出“阻塞”状态，并且计时器服务/守护程序任务的优先级等于或大于当前正在执行的任务（被中断的任务），则\* pxHigherPriorityTaskWoken将获得在xTimerResetFromISR（）函数内部将其设置为pdTRUE。如果xTimerResetFromISR（）将此值设置为pdTRUE，则应在中断退出之前执行上下文切换。 |

**返回值：**

如果无法将重置命令发送到计时器命令队列，则将返回pdFAIL。如果命令已成功发送到计时器命令队列，则将返回pdPASS。实际处理命令的时间将取决于计时器服务/守护程序任务相对于系统中其他任务的优先级，尽管计时器的到期时间与实际调用xTimerResetFromISR（）有关。计时器服务/守护程序任务优先级由configTIMER\_TASK\_PRIORITY配置常量设置。

**用法示例：**

**/\* This scenario assumes xBacklightTimer has already been created. When a**

**key is pressed, an LCD back-light is switched on. If 5 seconds pass**

**without a key being pressed, then the LCD back-light is switched off. In**

**this case, the timer is a one-shot timer, and unlike the example given for**

**the xTimerReset() function, the key press event handler is an interrupt**

**service routine. \*/**

**/\* The callback function assigned to the one-shot timer. In this case the**

**parameter is not used. \*/**

**void vBacklightTimerCallback( TimerHandle\_t pxTimer )**

**{**

**/\* The timer expired, therefore 5 seconds must have passed since a key**

**was pressed. Switch off the LCD back-light. \*/**

**vSetBacklightState( BACKLIGHT\_OFF );**

**}**

**/\* The key press interrupt service routine. \*/**

**void vKeyPressEventInterruptHandler( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**/\* Ensure the LCD back-light is on, then reset the timer that is**

**responsible for turning the back-light off after 5 seconds of**

**key inactivity. This is an interrupt service routine so can only**

**call FreeRTOS API functions that end in "FromISR". \*/**

**vSetBacklightState( BACKLIGHT\_ON );**

**/\* xTimerStartFromISR() or xTimerResetFromISR() could be called here**

**as both cause the timer to re-calculate its expiry time.**

**xHigherPriorityTaskWoken was initialised to pdFALSE when it was**

**declared (in this function). \*/**

**if( xTimerResetFromISR( xBacklightTimer,**

**&xHigherPriorityTaskWoken ) != pdPASS )**

**{**

**/\* The reset command was not executed successfully. Take appropriate**

**action here. \*/**

**}**

**/\* Perform the rest of the key processing here. \*/**

**/\* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch**

**should be performed. The syntax required to perform a context switch**

**from inside an ISR varies from port to port, and from compiler to**

**compiler. Inspect the demos for the port you are using to find the**

**actual syntax required. \*/**

**if( xHigherPriorityTaskWoken != pdFALSE )**

**{**

**/\* Call the interrupt safe yield function here (actual function**

**depends on the FreeRTOS port being used). \*/**

**}**

**}**

# pvTimerGetTimerID [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**void \*pvTimerGetTimerID( TimerHandle\_t xTimer );**

返回分配给[软件计时器](https://www.freertos.org/RTOS-software-timer.html)的ID 。

使用用于创建计时器的[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)调用的pvTimerID参数将ID分配给 计时器。

创建计时器时，[将为](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html)计时器分配一个标识符（ID），并且可以使用[vTimerSetTimerID（）](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html) API函数随时对其进行更改。

如果将相同的回调函数分配给多个计时器，则可以在回调函数内部检查计时器标识符，以确定哪个计时器实际到期。

计时器标识符还可用于在两次调用计时器的回调函数之间将数据存储在计时器中。

**参数：**

|  |  |
| --- | --- |
| 计时器 | 正在查询计时器。 |

**返回值：**

分配给要查询的计时器的ID。

**用法示例：**

请参阅[xTimerCreate（）文档页面](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html) 和[vTimerSetTimerID（）文档页面](https://www.freertos.org/FreeRTOS-timers-vTimerSetTimerID.html)上提供的示例。

# vTimerSetReloadMode [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**void vTimerSetReloadMode( TimerHandle\_t xTimer,**

**const UBaseType\_t uxAutoReload );**

将[软件计时器](https://www.freertos.org/RTOS-software-timer.html)的“模式”更新为自动重载计时器或单次计时器。

自动重载计时器在每次到期时都会自行重置，从而导致该计时器定期到期（并因此执行其回调）。

一键式定时器不会自动重置自身，因此除非手动重新启动，否则它将仅过期一次（并因此执行其回调）。

仅当所构建项目中包含FreeRTOS / Source / timers.c源文件时，此API函数才可用。

**参数：**

|  |  |
| --- | --- |
| xTimer | 计时器的句柄进行更新。该句柄将从 用于创建计时器的[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html) 或[xTimerCreateStatic（）](https://www.freertos.org/xTimerCreateStatic.html)的调用中返回。 |
| uxAutoReload | 将uxAutoReload设置为pdTRUE可将计时器设置为自动重载模式，或将pdFALSE设置为将计时器设置为单发模式。 |

# vTimerSetTimerID [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**void vTimerSetTimerID( TimerHandle\_t xTimer, void \*pvNewID );**

创建[计时器](https://www.freertos.org/RTOS-software-timer.html)时，将为[软件计时器](https://www.freertos.org/RTOS-software-timer.html) 分配一个标识符（ID），并且可以使用vTimerSetTimerID（）API函数随时对其进行更改。

如果将相同的回调函数分配给多个计时器，则可以在回调函数内部检查计时器标识符，以确定哪个计时器实际到期。

计时器标识符还可用于在两次调用计时器的回调函数之间将数据存储在计时器中。

**参数：**

|  |  |
| --- | --- |
| xTimer | 计时器正在更新。 |
| pvNewID | 设置计时器标识符的句柄。 |

**用法示例：**

**/\* A callback function assigned to a timer. \*/**

**void TimerCallbackFunction( TimerHandle\_t pxExpiredTimer )**

**{**

**uint32\_t ulCallCount;**

**/\* A count of the number of times this timer has expired**

**and executed its callback function is stored in the**

**timer's ID. Retrieve the count, increment it, then save**

**it back into the timer's ID. \*/**

**ulCallCount =**

**( uint32\_t )** [**pvTimerGetTimerID**](https://www.freertos.org/FreeRTOS-timers-pvTimerGetTimerID.html)**( pxExpiredTimer );**

**ulCallCount++;**

**vTimerSetTimerID( pxExpiredTimer, ( void \* ) ulCallCount );**

**}**

# xTimerPendFunctionCall() [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerPendFunctionCall(**

**PendedFunction\_t xFunctionToPend,**

**void \*pvParameter1,**

**uint32\_t ulParameter2,**

**TickType\_t xTicksToWait );**

Used to pend the execution of a function to the RTOS daemon task (the timer service task, hence this function is pre-fixed with ‘Timer’).

Functions that can be deferred to the RTOS daemon task must have the following prototype:

用于将功能的执行附加到RTOS守护程序任务（计时器服务任务，因此该功能以“ Timer”为前缀）。

可以推迟到RTOS守护程序任务的函数必须具有以下原型：

void vPendableFunction（void \* pvParameter1，uint32\_t ulParameter2）;

该pvParameter1和ulParameter2提供了应用程序代码中使用。

为了使xTimerPendFunctionCall（）可用，必须将INCLUDE\_xTimerPendFunctionCall（）和configUSE\_TIMERS都设置为1。

**参数：**

|  |  |
| --- | --- |
| *xFunctionToPend* | 从计时器服务/守护程序任务执行的功能。该函数必须符合PendedFunction\_t原型，如上所示。 |
| *pvParameter1* | 回调函数的第一个参数的值。该参数具有void \*类型，以使其可以用于传递任何类型。例如，可以将整数类型转换为void \*，或者可以使用void \*指向结构。 |
| *ulParameter2* | 回调函数的第二个参数的值。 |
| *xTicksToWait* | 调用此函数将导致消息发送到队列中的计时器守护程序任务。xTicksToWait是调用任务应保持在“阻塞”状态（因此不使用任何处理时间）以使计时器队列中的空间变满（如果发现队列已满）的时间。队列的长度由FreeRTOSConfig.h中的configTIMER\_QUEUE\_LENGTH的值设置。 |

**返回值：**

如果消息已成功发送到RTOS计时器守护程序任务，则返回pdPASS，否则返回pdFALSE。

# xTimerPendFunctionCallFromISR() [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**BaseType\_t xTimerPendFunctionCallFromISR(**

**PendedFunction\_t xFunctionToPend,**

**void \*pvParameter1,**

**uint32\_t ulParameter2,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

从应用程序中断服务例程中使用，以将功能的执行推迟到RTOS守护程序任务（计时器服务任务，因此此功能在timers.c中实现，并以“ Timer”作为前缀）。

理想情况下，中断服务程序（ISR）保持尽可能短，但是有时ISR要么要做很多处理，要么需要执行不确定的处理。在这些情况下，可以使用xTimerPendFunctionCallFromISR（）将函数的处理推迟到RTOS守护程序任务。

提供了一种机制，该机制允许中断直接返回到随后将执行挂起的功能的任务。这允许回调函数在中断的时间连续执行-就像回调函数是在中断本身中执行一样。

可以推迟到RTOS守护程序任务的函数必须具有以下原型：

void vPendableFunction（void \* pvParameter1，uint32\_t ulParameter2）;

该pvParameter1和ulParameter2提供了应用程序代码中使用。

为了使xTimerPendFunctionCallFromISR（）可用，必须将INCLUDE\_xTimerPendFunctionCall（）和configUSE\_TIMERS都设置为1。

**参数：**

|  |  |
| --- | --- |
| xFunctionToPend | 从计时器服务/守护程序任务执行的功能。该函数必须符合PendedFunction\_t原型，如上所示。 |
| pvParameter1 | 回调函数的第一个参数的值。该参数具有void \*类型，以使其可以用于传递任何类型。例如，可以将整数类型转换为void \*，或者可以使用void \*指向结构。 |
| ulParameter2 | 回调函数的第二个参数的值。 |
| pxHigherPriorityTaskWoken | 如上所述，调用xTimerPendFunctionCallFromISR（）将导致消息发送到RTOS计时器守护程序任务。如果守护程序任务的优先级（ 在FreeRTOSConfig.h中使用[configTIMER\_TASK\_PRIORITY](https://www.freertos.org/a00110.html" \l "configTIMER_TASK_PRIORITY)设置 ）高于当前正在运行的任务（中断被中断的任务）的优先级，则\* pxHigherPriorityTaskWoken将在xTimerPendFunctionCallFromISR（）中设置为pdTRUE。在中断退出之前，应该请求上下文切换。因此，必须将\* pxHigherPriorityTaskWoken初始化为pdFALSE。请参见下面的示例代码。 |

**返回值：**

如果消息已成功发送到RTOS计时器守护程序任务，则返回pdPASS，否则返回pdFALSE。

**用法示例：**

**/\* The callback function that will execute in the context of the daemon task.**

**Note callback functions must all use this same prototype. \*/**

**void vProcessInterface( void \*pvParameter1, uint32\_t ulParameter2 )**

**{**

**BaseType\_t xInterfaceToService;**

**/\* The interface that requires servicing is passed in the second**

**parameter. The first parameter is not used in this case. \*/**

**xInterfaceToService = ( BaseType\_t ) ulParameter2;**

**/\* ...Perform the processing here... \*/**

**}**

**/\* An ISR that receives data packets from multiple interfaces \*/**

**void vAnISR( void )**

**{**

**BaseType\_t xInterfaceToService, xHigherPriorityTaskWoken;**

**/\* Query the hardware to determine which interface needs processing. \*/**

**xInterfaceToService = prvCheckInterfaces();**

**/\* The actual processing is to be deferred to a task. Request the**

**vProcessInterface() callback function is executed, passing in the**

**number of the interface that needs processing. The interface to**

**service is passed in the second parameter. The first parameter is**

**not used in this case. \*/**

**xHigherPriorityTaskWoken = pdFALSE;**

**xTimerPendFunctionCallFromISR( vProcessInterface,**

**NULL,**

**( uint32\_t ) xInterfaceToService,**

**&xHigherPriorityTaskWoken );**

**/\* If xHigherPriorityTaskWoken is now set to pdTRUE then a context**

**switch should be requested. The macro used is port specific and will**

**be either portYIELD\_FROM\_ISR() or portEND\_SWITCHING\_ISR() - refer to**

**the documentation page for the port being used. \*/**

**portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

# pcTimerGetName [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**const char \* pcTimerGetName( TimerHandle\_t xTimer );**

返回[软件计时器的](https://www.freertos.org/RTOS-software-timer.html)可读文本名称。

使用用于创建计时器的[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html) 调用的pcTimerName参数将文本名称分配给 计时器。

**参数：**

|  |  |
| --- | --- |
| xTimer | 要查询定时器句柄。 |

**返回值：**

指向定时器文本名称的指针，该指针为标准NULL结尾的C字符串。

**用法示例：**

**const char \*pcTimerName = "ExampleTimer";**

**/\* A function that creates a timer. \*/**

**static void prvCreateTimer( void )**

**{**

**TimerHandle\_t xTimer;**

**/\* Create a timer. \*/**

**xTimer =** [**xTimerCreate**](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)**( pcTimerName, /\* Text name. \*/**

**pdMS\_TO\_TICKS( 500 ), /\* Period. \*/**

**pdTRUE, /\* Autoreload. \*/**

**NULL, /\* No ID. \*/**

**prvExampleCallback ); /\* Callback function. \*/**

**if( xTimer != NULL )**

**{**

**xTimerStart( xTimer, portMAX\_DELAY );**

**/\* Just to demonstrate pcTimerGetName(), query the timer's name and**

**assert if it does not equal pcTimerName. \*/**

[**configASSERT**](https://www.freertos.org/a00110.html#configASSERT)**( strcmp( pcTimerGetName( xTimer ), pcTimerName ) == 0 );**

**}**

**}**

# xTimerGetPeriod [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**TickType\_t xTimerGetPeriod( TimerHandle\_t xTimer );**

返回[软件计时器](https://www.freertos.org/RTOS-software-timer.html)的周期。周期以刻度为单位。

最初使用用于创建计时器的[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)调用的xTimerPeriod参数设置计时器的时间段 。然后可以使用[xTimerChangePeriod（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriod.html) 和[xTimerChangePeriodFromISR（）](https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriodFromISR.html) API函数进行更改 。

**参数：**

|  |  |
| --- | --- |
| xTimer | 正在查询计时器。 |

**返回值：**

计时器的周期，以滴答为单位。

**用法示例：**

**/\* A callback function assigned to a software timer. \*/**

**static void prvExampleTimerCallback( TimerHandle\_t xTimer )**

**{**

**TickType\_t xTimerPeriod;**

**/\* Query the period of the timer that expires. \*/**

**xTimerPeriod = xTimerGetPeriod( xTimer );**

**}**

# xTimerGetExpiryTime [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

timers.h

**TickType\_t xTimerGetExpiryTime( TimerHandle\_t xTimer );**

返回[软件计时器](https://www.freertos.org/RTOS-software-timer.html)将到期的时间，这是计时器的回调函数将执行的时间。

如果xTimerGetExpiryTime（）返回的值小于当前时间，则计时器将在滴答计数溢出并换回为0后到期。溢出是在RTOS实现本身中处理的，因此计时器的回调函数将在正确的时间执行时间是滴答计数溢出之前还是之后。

**参数：**

|  |  |
| --- | --- |
| xTimer | 正在查询计时器。 |

**返回值：**

如果xTimer引用的计时器[处于活动状态](https://www.freertos.org/FreeRTOS-timers-xTimerIsTimerActive.html)，则返回计时器下一次到期的时间（可能在当前滴答计数溢出之后，请参见上面的注释）。

如果xTimer引用的计时器未激活，则返回值不确定。

**用法示例：**

**static void prvAFunction( TimerHandle\_t xTimer )**

**{**

**TickType\_t xRemainingTime;**

**/\* Calculate the time that remains before the timer referenced by xTimer**

**expires. TickType\_t is an unsigned type, so the subtraction will result in**

**the correct answer even if the timer will not expire until after the tick**

**count has overflowed. \*/**

**xRemainingTime = xTimerGetExpiryTime( xTimer ) -** [**xTaskGetTickCount()**](https://www.freertos.org/a00021.html#xTaskGetTickCount)**;**

**}**

# uxTimerGetReloadMode [[Timer API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)]

task.h

**UBaseType\_t uxTimerGetReloadMode( TimerHandle\_t xTimer );**

查询xTimer句柄引用的[软件计时器](https://www.freertos.org/RTOS-software-timer.html)的“模式” 。

该模式可以是自动重新加载的计时器，该计时器在每次到期时都会自动重置，也可以是单次使用的计时器，除非手动重新启动，否则它只会过期一次。

这个API功能仅当FreeRTOS的“timers.c”源文件包含在内置项目，[configUSE\_TIMERS](https://www.freertos.org/a00110.html" \l "configUSE_TIMERS)在FreeRTOSConfig.h中设置为1。

**参数：**

|  |  |
| --- | --- |
| xTimer | 要查询的计时器的句柄。该句柄将从用于创建计时器的[xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html) 或[xTimerCreateStatic（）](https://www.freertos.org/xTimerCreateStatic.html)的调用中返回。 |

**返回值：**

如果具有句柄xTimer的计时器是自动重载计时器，则为pdTRUE，否则为pdFALSE。

# 事件组

|  |
| --- |
| [**提示：在许多情况下，“任务通知”可以为事件组提供轻巧的选择**](https://www.freertos.org/RTOS_Task_Notification_As_Event_Group.html) |

## 事件组和事件位API函数

* [**xEventGroupCreate**](https://www.freertos.org/xEventGroupCreate.html)
* [**xEventGroupCreateStatic**](https://www.freertos.org/xEventGroupCreateStatic.html)
* [**xEventGroupWaitBits**](https://www.freertos.org/xEventGroupWaitBits.html)
* [**xEventGroupSetBits**](https://www.freertos.org/xEventGroupSetBits.html)
* [**xEventGroupSetBitsFromISR**](https://www.freertos.org/xEventGroupSetBitsFromISR.html)
* [**xEventGroupClearBits**](https://www.freertos.org/xEventGroupClearBits.html)
* [**xEventGroupClearBitsFromISR**](https://www.freertos.org/xEventGroupClearBitsFromISR.html)
* [**xEventGroupGetBits**](https://www.freertos.org/xEventGroupGetBits.html)
* [**xEventGroupGetBitsFromISR**](https://www.freertos.org/xEventGroupGetBitsFromISR.html)
* [**xEventGroupSync**](https://www.freertos.org/xEventGroupSync.html)
* [**vEventGroupDelete**](https://www.freertos.org/vEventGroupDelete.html)

# xEventGroupCreate() [[事件组API](https://www.freertos.org/event-groups-API.html) ]

|  |
| --- |
| [**提示：在许多情况下，“任务通知”可以为事件组提供轻巧的选择**](https://www.freertos.org/RTOS_Task_Notification_As_Event_Group.html) |

event\_groups.h

**EventGroupHandle\_t xEventGroupCreate( void );**

创建一个新的RTOS[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)，并返回一个句柄，通过它可以引用新创建的事件组。

要使此RTOS API函数可用：

1. 必须在FreeRTOSConfig.h[中将configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)设置为1，或者未定义（在这种情况下，它将默认为1）。
2. RTOS源文件FreeRTOS/source/event\_groups.c必须包含在构建中。

每个事件组需要[非常]少量的RAM，用于保留事件组的状态。如果使用xEventGroupCreate（）创建事件组，则将从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用[xEventGroupCreateStatic](https://www.freertos.org/xEventGroupCreateStatic.html)（）创建事件组，则RAM由应用程序[编写](https://www.freertos.org/xEventGroupCreateStatic.html)器提供，它需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

事件组存储在EventBits\_t类型的变量中。如果将[configUSE\_16\_BIT\_TICKS](https://www.freertos.org/a00110.html" \l "configUSE_16_BIT_TICKS)设置为1，则在事件组内实现的位数（或标志）为8；如果将[configUSE\_16\_BIT\_TICKS](https://www.freertos.org/a00110.html" \l "configUSE_16_BIT_TICKS)设置为0，则为24。对configUSE\_16\_BIT\_TICKS的依赖性是由于在内部实现中用于线程本地存储的数据类型RTOS任务。

**参数：**

|  |  |
| --- | --- |
| 没有 |  |

**返回值：**

如果创建了事件组，则返回事件组的句柄。如果没有足够的[FreeRTOS堆](https://www.freertos.org/a00111.html) 来创建事件组，则返回NULL。

**用法示例：**

**/\* Declare a variable to hold the created event group. \*/**

**EventGroupHandle\_t xCreatedEventGroup;**

**/\* Attempt to create the event group. \*/**

**xCreatedEventGroup = xEventGroupCreate();**

**/\* Was the event group created successfully? \*/**

**if( xCreatedEventGroup == NULL )**

**{**

**/\* The event group was not created because there was insufficient**

**FreeRTOS heap available. \*/**

**}**

**else**

**{**

**/\* The event group was created. \*/**

**}**

# xEventGroupCreateStatic [[事件组API](https://www.freertos.org/event-groups-API.html) ]

|  |
| --- |
| [**提示：在许多情况下，“任务通知”可以为事件组提供轻巧的选择**](https://www.freertos.org/RTOS_Task_Notification_As_Event_Group.html) |

event\_groups.h

**EventGroupHandle\_t xEventGroupCreateStatic(**

**StaticEventGroup\_t \*pxEventGroupBuffer );**

创建一个新的RTOS[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)，并返回一个句柄，通过它可以引用新创建的事件组。 必须在FreeRTOSConfig.h[中将configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_STATIC_ALLOCATION)设置为1，并且必须将RTOS源文件FreeRTOS / source / event\_groups.c包含在构建中，以便xEventGroupCreateStatic（）函数可用。

每个事件组需要[非常]少量的RAM，用于保留事件组的状态。如果使用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)创建事件组，[则将](https://www.freertos.org/xEventGroupCreate.html) 从[FreeRTOS堆中](https://www.freertos.org/a00111.html)自动分配所需的RAM 。如果使用xEventGroupCreateStatic（）创建事件组，则RAM由应用程序编写器提供，这需要附加参数，但允许在编译时静态分配RAM。有关更多信息，请参见[静态与动态分配](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)页面。

事件组存储在EventBits\_t类型的变量中。如果将[configUSE\_16\_BIT\_TICKS](https://www.freertos.org/a00110.html" \l "configUSE_16_BIT_TICKS)设置为1，则在事件组内实现的位数（或标志）为8；如果将[configUSE\_16\_BIT\_TICKS](https://www.freertos.org/a00110.html" \l "configUSE_16_BIT_TICKS)设置为0，则为24。对configUSE\_16\_BIT\_TICKS的依赖性是由于在内部实现中用于线程本地存储的数据类型RTOS任务。

**参数：**

|  |  |
| --- | --- |
| pxEventGroupBuffer | 必须指向StaticEventGroup\_t类型的变量，事件组数据结构将存储在该变量中。 |

**返回值：**

如果事件组创建成功，则返回事件组的句柄。如果pxEventGroupBuffer为NULL，则返回NULL。

**用法示例：**

**/\* Declare a variable to hold the handle of the created event group. \*/**

**EventGroupHandle\_t xEventGroupHandle;**

**/\* Declare a variable to hold the data associated with the created**

**event group. \*/**

**StaticEventGroup\_t xCreatedEventGroup;**

**/\* Attempt to create the event group. \*/**

**xEventGroupHandle = xEventGroupCreateStatic( &xCreatedEventGroup );**

**/\* pxEventGroupBuffer was not null so expect the event group to have**

**been created? \*/**

[**configASSERT**](https://www.freertos.org/a00110.html#configASSERT)**( xEventGroupHandle );**

# vEventGroupDelete() [[Event Group API](https://www.freertos.org/event-groups-API.html)]

event\_groups.h

**void vEventGroupDelete( EventGroupHandle\_t xEventGroup );**

删除以前使用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)调用创建的[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)。

在要删除的事件组上被阻止的任务将被取消阻止，并报告事件组值为0。

RTOS源文件FreeRTOS / source / event\_groups.c必须包含在构建中，以便vEventGroupDelete（）函数可用。

不能从中断调用此函数。

**参数：**

|  |  |
| --- | --- |
| xEventGroup | 事件组被删除。 |

**返回值：**

没有。

# xEventGroupWaitBits() [[Event Group API](https://www.freertos.org/event-groups-API.html)]

event\_groups.h

**EventBits\_t xEventGroupWaitBits(**

**const EventGroupHandle\_t xEventGroup,**

**const EventBits\_t uxBitsToWaitFor,**

**const BaseType\_t xClearOnExit,**

**const BaseType\_t xWaitForAllBits,**

**TickType\_t xTicksToWait );**

读取RTOS[事件组中的位](https://www.freertos.org/FreeRTOS-Event-Groups.html)，可以选择进入Blocked状态（带有超时），以等待位或位组的置位。

不能从中断调用此函数。

RTOS源文件FreeRTOS / source / event\_groups.c必须包含在构建中，以便xEventGroupWaitBits（）函数可用。

**参数：**

|  |  |
| --- | --- |
| xEventGroup | 在其中测试位的事件组。必须事先通过调用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)创建事件组 。 |
| uxBitsToWaitFor | 指示事件组内部要测试的一位或多位的按位值。例如，要等待位0和/或位2，请将uxBitsToWaitFor设置为0x05。要等待位0和/或位1和/或位2，请将uxBitsToWaitFor设置为0x07。等等。  uxBitsToWaitFor**不得**设置为0。 |
| xClearOnExit | 如果xClearOnExit设置为pdTRUE，则如果xEventGroupWaitBits（）由于超时以外的其他原因返回，则xEventGroupWaitBits（）返回之前，将在事件组中清除作为uxBitsToWaitFor参数传递的值中设置的任何位。超时值由xTicksToWait参数设置。  如果xClearOnExit设置为pdFALSE，则当对xEventGroupWaitBits（）的调用返回时，事件组中设置的位不会更改。 |
| xWaitForAllBits | xWaitForAllBits用于创建逻辑AND测试（必须设置所有位）或逻辑OR测试（必须设置一个或多个位），如下所示：  如果将xWaitForAllBits设置为pdTRUE，则当在事件组中设置了作为uxBitsToWaitFor参数传递的值中设置的**所有**位或者指定的阻止时间到期时，xEventGroupWaitBits（）将返回。  如果将xWaitForAllBits设置为pdFALSE，则在事件组中设置了作为uxBitsToWaitFor参数传递的值中设置的**任何**位或指定的阻止时间到期时，xEventGroupWaitBits（）将返回。 |
| xTicksToWait | 等待设置uxBitsToWaitFor指定的位中的一个/全部（取决于xWaitForAllBits值）的最长时间（以“ ticks”指定）。 |

**返回值：**

等待的事件位被置位或块时间到期时事件组的值。如果较高优先级的任务或中断更改了调用任务离开阻塞状态与退出xEventGroupWaitBits（）函数之间的事件位的值，则事件组中事件位的当前值将不同于返回值。

测试返回值以了解设置了哪些位。如果xEventGroupWaitBits（）由于其超时到期而返回，则不会设置所有等待的位。如果由于设置了xEventGroupWaitBits（）而等待的位被返回，则返回值是在自动清除任何位之前的事件组值，因为xClearOnExit参数设置为pdTRUE。

**用法示例：**

**#define BIT\_0 ( 1 << 0 )**

**#define BIT\_4 ( 1 << 4 )**

**void aFunction( EventGroupHandle\_t xEventGroup )**

**{**

**EventBits\_t uxBits;**

**const TickType\_t xTicksToWait = 100 / portTICK\_PERIOD\_MS;**

**/\* Wait a maximum of 100ms for either bit 0 or bit 4 to be set within**

**the event group. Clear the bits before exiting. \*/**

**uxBits = xEventGroupWaitBits(**

**xEventGroup, /\* The event group being tested. \*/**

**BIT\_0 | BIT\_4, /\* The bits within the event group to wait for. \*/**

**pdTRUE, /\* BIT\_0 & BIT\_4 should be cleared before returning. \*/**

**pdFALSE, /\* Don't wait for both bits, either bit will do. \*/**

**xTicksToWait );/\* Wait a maximum of 100ms for either bit to be set. \*/**

**if( ( uxBits & ( BIT\_0 | BIT\_4 ) ) == ( BIT\_0 | BIT\_4 ) )**

**{**

**/\* xEventGroupWaitBits() returned because both bits were set. \*/**

**}**

**else if( ( uxBits & BIT\_0 ) != 0 )**

**{**

**/\* xEventGroupWaitBits() returned because just BIT\_0 was set. \*/**

**}**

**else if( ( uxBits & BIT\_4 ) != 0 )**

**{**

**/\* xEventGroupWaitBits() returned because just BIT\_4 was set. \*/**

**}**

**else**

**{**

**/\* xEventGroupWaitBits() returned because xTicksToWait ticks passed**

**without either BIT\_0 or BIT\_4 becoming set. \*/**

**}**

**}**

# xEventGroupSetBits()

event\_groups.h

**EventBits\_t xEventGroupSetBits( EventGroupHandle\_t xEventGroup,**

**const EventBits\_t uxBitsToSet );**

在RTOS[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)内设置位（标志）。不能从中断调用此函数。 [xEventGroupSetBitsFromISR（）](https://www.freertos.org/xEventGroupSetBitsFromISR.html) 是可以从中断中调用的版本。

在事件组中设置位将自动解除阻止等待这些位的任务。

RTOS源文件FreeRTOS / source / event\_groups.c必须包含在构建中，以便xEventGroupSetBits（）函数可用。

**参数：**

|  |  |
| --- | --- |
| xEventGroup | 将在其中设置位的事件组。必须事先通过调用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)创建事件组 。 |
| uxBitsToSet | 指示事件组中要设置的一个或多个位的按位值。例如，将uxBitsToSet设置为0x08仅设置第3位。将uxBitsToSet设置为0x09设置第3位和第0位。 |

**返回值：**

**调用xEventGroupSetBits（）返回时** 事件组的值。

返回值可能清除了uxBitsToSet参数指定的位的原因有两个：

1. 如果设置一个位导致正在等待该位离开阻塞状态的任务，则该位可能已被自动清除（请参阅[xEventGroupWaitBits（）](https://www.freertos.org/xEventGroupWaitBits.html)的xClearBitOnExit参数 ）。
2. 优先级高于被称为xEventGroupSetBits（）的任务的任何优先级的任何非阻塞（或“就绪”状态）任务都将执行，并且可以在调用xEventGroupSetBits（）返回之前更改事件组的值。

**用法示例：**

**#define BIT\_0 ( 1 << 0 )**

**#define BIT\_4 ( 1 << 4 )**

**void aFunction( EventGroupHandle\_t xEventGroup )**

**{**

**EventBits\_t uxBits;**

**/\* Set bit 0 and bit 4 in xEventGroup. \*/**

**uxBits = xEventGroupSetBits(**

**xEventGroup, /\* The event group being updated. \*/**

**BIT\_0 | BIT\_4 );/\* The bits being set. \*/**

**if( ( uxBits & ( BIT\_0 | BIT\_4 ) ) == ( BIT\_0 | BIT\_4 ) )**

**{**

**/\* Both bit 0 and bit 4 remained set when the function returned. \*/**

**}**

**else if( ( uxBits & BIT\_0 ) != 0 )**

**{**

**/\* Bit 0 remained set when the function returned, but bit 4 was**

**cleared. It might be that bit 4 was cleared automatically as a**

**task that was waiting for bit 4 was removed from the Blocked**

**state. \*/**

**}**

**else if( ( uxBits & BIT\_4 ) != 0 )**

**{**

**/\* Bit 4 remained set when the function returned, but bit 0 was**

**cleared. It might be that bit 0 was cleared automatically as a**

**task that was waiting for bit 0 was removed from the Blocked**

**state. \*/**

**}**

**else**

**{**

**/\* Neither bit 0 nor bit 4 remained set. It might be that a task**

**was waiting for both of the bits to be set, and the bits were cleared**

**as the task left the Blocked state. \*/**

**}**

**}**

# xEventGroupSetBitsFromISR() [[Event Group API](https://www.freertos.org/event-groups-API.html)]

event\_groups.h

**BaseType\_t xEventGroupSetBitsFromISR(**

**EventGroupHandle\_t xEventGroup,**

**const EventBits\_t uxBitsToSet,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

在RTOS[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)内设置位（标志）。可以从中断服务程序（ISR）调用的[xEventGroupSetBits（）](https://www.freertos.org/xEventGroupSetBits.html)版本。

在事件组中设置位将自动解除阻止等待这些位的任务。

在事件组中设置位不是确定性操作，因为可能有数量未知的任务正在等待设置一个或多个位。FreeRTOS不允许在中断或关键部分执行不确定的操作。因此，xEventGroupSetBitFromISR（）向RTOS守护程序任务发送一条消息，以在守护程序任务的上下文中执行设置操作-在此任务中，使用调度程序锁代替关键节。

**注意：**如以上段落所述，来自ISR的设置位会将设置操作延迟到RTOS守护程序任务（也称为计时器服务任务）。与其他任何RTOS任务一样，RTOS守护程序任务是根据其优先级安排的。因此，如果必须立即完成设置操作（在应用程序创建的任务执行之前），则RTOS守护程序任务的优先级必须高于使用事件组的任何应用程序任务的优先级。RTOS守护程序任务的优先级由 FreeRTOSConfig.h中的[configTIMER\_TASK\_PRIORITY](https://www.freertos.org/Configuring-a-real-time-RTOS-application-to-use-software-timers.html)定义设置 。

必须将FreeRTOSConfig.h中的INCLUDE\_xEventGroupSetBitFromISR，configUSE\_TIMERS和INCLUDE\_xTimerPendFunctionCall都设置为1，xEventGroupSetBitsFromISR（）函数才可用。

RTOS源文件FreeRTOS / source / event\_groups.c必须包含在构建中，以便xEventGroupSetBitsFromISR（）函数可用。

**参数：**

|  |  |
| --- | --- |
| xEventGroup | 将在其中设置位的事件组。必须事先通过调用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)创建事件组 。 |
| uxBitsToSet | 指示要设置的一个或多个位的按位值。例如，将uxBitsToSet设置为0x08仅设置第3位。将uxBitsToSet设置为0x09设置第3位和第0位。 |
| pxHigherPriorityTaskWoken | 如上所述，调用此函数将导致消息发送到RTOS守护程序任务。如果守护程序任务的优先级高于当前正在运行的任务（被中断的任务）的优先级，则xEventGroupSetBitsFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE，指示应在中断退出之前请求上下文切换。因此，必须将\* pxHigherPriorityTaskWoken初始化为pdFALSE。请参见下面的示例代码。 |

**返回值：**

如果消息已发送到RTOS守护程序任务，则返回pdPASS，否则返回pdFAIL。如果[计时器服务队列](https://www.freertos.org/RTOS-software-timer-service-daemon-task.html)已满，将返回pdFAIL 。

**用法示例：**

**#define BIT\_0 ( 1 << 0 )**

**#define BIT\_4 ( 1 << 4 )**

**/\* An event group which it is assumed has already been created by a call to**

**xEventGroupCreate(). \*/**

**EventGroupHandle\_t xEventGroup;**

**void anInterruptHandler( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken, xResult;**

**/\* xHigherPriorityTaskWoken must be initialised to pdFALSE. \*/**

**xHigherPriorityTaskWoken = pdFALSE;**

**/\* Set bit 0 and bit 4 in xEventGroup. \*/**

**xResult = xEventGroupSetBitsFromISR(**

**xEventGroup, /\* The event group being updated. \*/**

**BIT\_0 | BIT\_4, /\* The bits being set. \*/**

**&xHigherPriorityTaskWoken );**

**/\* Was the message posted successfully? \*/**

**if( xResult != pdFAIL )**

**{**

**/\* If xHigherPriorityTaskWoken is now set to pdTRUE then a context**

**switch should be requested. The macro used is port specific and will**

**be either portYIELD\_FROM\_ISR() or portEND\_SWITCHING\_ISR() - refer to**

**the documentation page for the port being used. \*/**

**portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

**}**

# xEventGroupClearBits()

event\_groups.h

**EventBits\_t xEventGroupClearBits(**

**EventGroupHandle\_t xEventGroup,**

**const EventBits\_t uxBitsToClear );**

清除RTOS[事件组中的](https://www.freertos.org/FreeRTOS-Event-Groups.html)位（标志）。不能从中断调用此函数。有关可以从中断调用的版本，请参见[xEventGroupClearBitsFromISR（）](https://www.freertos.org/xEventGroupClearBitsFromISR.html)。

RTOS源文件FreeRTOS / source / event\_groups.c必须包含在构建中，以便xEventGroupClearBits（）函数可用。

**参数：**

|  |  |
| --- | --- |
| xEventGroup | 要清除其中位的事件组。必须事先通过调用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)创建事件组 。 |
| uxBitsToClear | 指示事件组中要清除的一个或多个位的按位值。例如，将uxBitsToClear设置为0x08仅清除位3。将uxBitsToClear设置为0x09以清除位3和位0。 |

**返回值：**

清除指定位**之前** 的事件组的值。

**用法示例：**

**#define BIT\_0 ( 1 << 0 )**

**#define BIT\_4 ( 1 << 4 )**

**void aFunction( EventGroupHandle\_t xEventGroup )**

**{**

**EventBits\_t uxBits;**

**/\* Clear bit 0 and bit 4 in xEventGroup. \*/**

**uxBits = xEventGroupClearBits(**

**xEventGroup, /\* The event group being updated. \*/**

**BIT\_0 | BIT\_4 );/\* The bits being cleared. \*/**

**if( ( uxBits & ( BIT\_0 | BIT\_4 ) ) == ( BIT\_0 | BIT\_4 ) )**

**{**

**/\* Both bit 0 and bit 4 were set before xEventGroupClearBits()**

**was called. Both will now be clear (not set). \*/**

**}**

**else if( ( uxBits & BIT\_0 ) != 0 )**

**{**

**/\* Bit 0 was set before xEventGroupClearBits() was called. It will**

**now be clear. \*/**

**}**

**else if( ( uxBits & BIT\_4 ) != 0 )**

**{**

**/\* Bit 4 was set before xEventGroupClearBits() was called. It will**

**now be clear. \*/**

**}**

**else**

**{**

**/\* Neither bit 0 nor bit 4 were set in the first place. \*/**

**}**

**}**

# xEventGroupClearBitsFromISR() [[Event Group API](https://www.freertos.org/event-groups-API.html)]

event\_groups.h

**BaseType\_t xEventGroupClearBitsFromISR(**

**EventGroupHandle\_t xEventGroup,**

**const EventBits\_t uxBitsToClear );**

可以从中断中调用 的[xEventGroupClearBits（）](https://www.freertos.org/xEventGroupClearBits.html)版本。清除操作将推迟到RTOS守护程序任务，也称为计时器服务任务。守护程序任务的优先级由 FreeRTOSConfig.h中的 [configTIMER\_TASK\_PRIORITY](https://www.freertos.org/a00110.html" \l "configTIMER_TASK_PRIORITY)设置来设置。

RTOS源文件FreeRTOS / source / event\_groups.c必须包含在构建中，以便xEventGroupClearBitsFromISR（）函数可用。

**参数：**

|  |  |
| --- | --- |
| xEventGroup | 要清除其中位的事件组。必须事先通过调用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)创建事件组 。 |
| uxBitsToClear | 指示事件组中要清除的一个或多个位的按位值。例如，将uxBitsToClear设置为0x08仅清除位3。将uxBitsToClear设置为0x09以清除位3和位0。 |

**返回值：**

pdPASS，如果操作已成功推迟到RTOS守护程序任务。否则为pdFALSE。仅当[计时器命令队列](https://www.freertos.org/RTOS-software-timer-service-daemon-task.html)已满时才返回pdFALSE 。

**用法示例：**

**#define BIT\_0 ( 1 << 0 )**

**#define BIT\_4 ( 1 << 4 )**

**/\* This code assumes the event group referenced by the**

**xEventGroup variable has already been created using a call to**

**xEventGroupCreate(). \*/**

**void anInterruptHandler( void )**

**{**

**BaseType\_t xSuccess;**

**/\* Clear bit 0 and bit 4 in xEventGroup. \*/**

**xSuccess = xEventGroupClearBitsFromISR(**

**xEventGroup, /\* The event group being updated. \*/**

**BIT\_0 | BIT\_4 );/\* The bits being cleared. \*/**

**if( xSuccess == pdPASS )**

**{**

**/\* The command was sent to the daemon task. \*/**

**}**

**else**

**{**

**/\* The clear bits command was not sent to the daemon task. \*/**

**}**

**}**

# xEventGroupGetBits() [[Event Group API](https://www.freertos.org/event-groups-API.html)]

event\_groups.h

**EventBits\_t xEventGroupGetBits( EventGroupHandle\_t xEventGroup );**

返回RTOS[事件组中](https://www.freertos.org/FreeRTOS-Event-Groups.html)事件位（事件标志）的当前值。不能从中断使用此功能。有关 可以在中断中使用的版本，请参见[xEventGroupGetBitsFromISR（）](https://www.freertos.org/xEventGroupGetBitsFromISR.html)。

RTOS源文件FreeRTOS / source / event\_groups.c必须包含在构建中，以便xEventGroupGetBits（）函数可用。

**参数：**

|  |  |
| --- | --- |
| xEventGroup | 正在查询的事件组。必须事先通过调用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)创建事件组 。 |

**返回值：**

调用xEventGroupGetBits（）时，事件组中事件位的值。

# xEventGroupGetBitsFromISR() [[Event Group API](https://www.freertos.org/event-groups-API.html)]

event\_groups.h

**EventBits\_t xEventGroupGetBitsFromISR(**

**EventGroupHandle\_t xEventGroup );**

可以从中断中调用 的[xEventGroupGetBits（）](https://www.freertos.org/xEventGroupGetBits.html)版本。

RTOS源文件FreeRTOS / source / event\_groups.c必须包含在构建中，以便xEventGroupGetBitsFrom（）函数可用。

**参数：**

|  |  |
| --- | --- |
| xEventGroup | 正在查询的事件组。必须事先通过调用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)创建事件组 。 |

**返回值：**

调用xEventGroupGetBitsFromISR（）时，事件组中事件位的值。

# xEventGroupSync() [[Event Group API](https://www.freertos.org/event-groups-API.html)]

event\_groups.h

**EventBits\_t xEventGroupSync( EventGroupHandle\_t xEventGroup,**

**const EventBits\_t uxBitsToSet,**

**const EventBits\_t uxBitsToWaitFor,**

**TickType\_t xTicksToWait );**

以原子方式设置RTOS[事件组中的](https://www.freertos.org/FreeRTOS-Event-Groups.html)位（标志），然后等待在同一事件组中设置位的组合。此功能通常用于同步多个任务（通常称为任务集合），其中每个任务必须先等待其他任务到达同步点，然后才能继续。

不能从中断使用此功能。

如果设置了uxBitsToWait参数指定的位，或者在该时间内设置了该位，则该函数将在其块时间到期之前返回。在这种情况下，uxBitsToWait指定的所有位将在函数返回之前自动清除。

RTOS源文件FreeRTOS/source/event\_groups.c必须包含在构建中，以便xEventGroupSync（）函数可用。

**参数：**

|  |  |
| --- | --- |
| xEventGroup | 设置和测试位的事件组。必须事先通过调用[xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)创建事件组 。 |
| uxBitsToSet | 在确定（并可能等待）是否已设置uxBitsToWait参数指定的所有位之前，事件组中要设置的位。例如，将uxBitsToSet设置为0x04以设置事件组中的位2。 |
| uxBitsToWaitFor | 指示事件组内部要测试的一位或多位的按位值。例如，将uxBitsToWaitFor设置为0x05以等待位0和位2。将uxBitsToWaitFor设置为0x07以等待位0和位1和位2等。 |
| xTicksToWait | 等待设置uxBitsToWaitFor参数值指定的所有位的最长时间（以“滴答”表示）。 |

**返回值：**

在等待位设置或块时间到期时，事件组的值。测试返回值以了解设置了哪些位。

如果xEventGroupSync（）由于超时而返回，则不会设置所有等待的位。

如果由于设置了xEventGroupSync（）而已等待的所有位而返回，则返回的值是事件组值，**然后**自动清除任何位。

**用法示例：**

**/\* Bits used by the three tasks. \*/**

**#define TASK\_0\_BIT ( 1 << 0 )**

**#define TASK\_1\_BIT ( 1 << 1 )**

**#define TASK\_2\_BIT ( 1 << 2 )**

**#define ALL\_SYNC\_BITS ( TASK\_0\_BIT | TASK\_1\_BIT | TASK\_2\_BIT )**

**/\* Use an event group to synchronise three tasks. It is assumed this event**

**group has already been created elsewhere. \*/**

**EventGroupHandle\_t xEventBits;**

**void vTask0( void \*pvParameters )**

**{**

**EventBits\_t uxReturn;**

**TickType\_t xTicksToWait = 100 / portTICK\_PERIOD\_MS;**

**for( ;; )**

**{**

**/\* Perform task functionality here. \*/**

**. . .**

**/\* Set bit 0 in the event group to note this task has reached the**

**sync point. The other two tasks will set the other two bits defined**

**by ALL\_SYNC\_BITS. All three tasks have reached the synchronisation**

**point when all the ALL\_SYNC\_BITS are set. Wait a maximum of 100ms**

**for this to happen. \*/**

**uxReturn = xEventGroupSync( xEventBits,**

**TASK\_0\_BIT,**

**ALL\_SYNC\_BITS,**

**xTicksToWait );**

**if( ( uxReturn & ALL\_SYNC\_BITS ) == ALL\_SYNC\_BITS )**

**{**

**/\* All three tasks reached the synchronisation point before the call**

**to xEventGroupSync() timed out. \*/**

**}**

**}**

**}**

**void vTask1( void \*pvParameters )**

**{**

**for( ;; )**

**{**

**/\* Perform task functionality here. \*/**

**. . .**

**/\* Set bit 1 in the event group to note this task has reached the**

**synchronisation point. The other two tasks will set the other two**

**bits defined by ALL\_SYNC\_BITS. All three tasks have reached the**

**synchronisation point when all the ALL\_SYNC\_BITS are set. Wait**

**indefinitely for this to happen. \*/**

**xEventGroupSync( xEventBits, TASK\_1\_BIT, ALL\_SYNC\_BITS, portMAX\_DELAY );**

**/\* xEventGroupSync() was called with an indefinite block time, so**

**this task will only reach here if the syncrhonisation was made by all**

**three tasks, so there is no need to test the return value. \*/**

**}**

**}**

**void vTask2( void \*pvParameters )**

**{**

**for( ;; )**

**{**

**/\* Perform task functionality here. \*/**

**. . .**

**/\* Set bit 2 in the event group to note this task has reached the**

**synchronisation point. The other two tasks will set the other two**

**bits defined by ALL\_SYNC\_BITS. All three tasks have reached the**

**synchronisation point when all the ALL\_SYNC\_BITS are set. Wait**

**indefinitely for this to happen. \*/**

**xEventGroupSync( xEventBits, TASK\_2\_BIT, ALL\_SYNC\_BITS, portMAX\_DELAY );**

**/\* xEventGroupSync() was called with an indefinite block time, so**

**this task will only reach here if the syncrhonisation was made by all**

**three tasks, so there is no need to test the return value. \*/**

**}**

**}**

# RTOS流缓冲区

## [RTOS流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)API函数：

* [**xStreamBufferCreate（）**](https://www.freertos.org/xStreamBufferCreate.html)
* [**xStreamBufferCreateStatic（）**](https://www.freertos.org/xStreamBufferCreateStatic.html)
* [**xStreamBufferSend（）**](https://www.freertos.org/xStreamBufferSend.html)
* [**xStreamBufferSendFromISR（）**](https://www.freertos.org/xStreamBufferSendFromISR.html)
* [**xStreamBufferReceive（）**](https://www.freertos.org/xStreamBufferReceive.html)
* [**xStreamBufferReceiveFromISR（）**](https://www.freertos.org/xStreamBufferReceiveFromISR.html)
* [**vStreamBufferDelete（）**](https://www.freertos.org/vStreamBufferDelete.html)
* [**xStreamBufferBytesAvailable（）**](https://www.freertos.org/xStreamBufferBytesAvailable.html)
* [**xStreamBufferSpacesAvailable（）**](https://www.freertos.org/xStreamBufferSpacesAvailable.html)
* [**xStreamBufferSetTriggerLevel（）**](https://www.freertos.org/xStreamBufferSetTriggerLevel.html)
* [**xStreamBufferReset（）**](https://www.freertos.org/xStreamBufferReset.html)
* [**xStreamBufferIsEmpty（）**](https://www.freertos.org/xStreamBufferIsEmpty.html)
* [**xStreamBufferIsFull（）**](https://www.freertos.org/xStreamBufferIsFull.html)

# xStreamBufferCreate() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**StreamBufferHandle\_t xStreamBufferCreate( size\_t xBufferSizeBytes,**

**size\_t xTriggerLevelBytes );**

使用动态分配的内存 创建一个新的[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)。有关使用静态分配的内存（在编译时分配的内存[）](https://www.freertos.org/xStreamBufferCreateStatic.html)的版本，请参见[xStreamBufferCreateStatic（](https://www.freertos.org/xStreamBufferCreateStatic.html)）。

必须将[configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)设置为1或在FreeRTOSConfig.h中未定义才能使xStreamBufferCreate（）可用。

通过 在构建中包含FreeRTOS/source/stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xBufferSizeBytes | 流缓冲区在任何一次都可以保留的字节总数。 |
| xTriggerLevelBytes | 在流缓冲区中被阻塞以等待数据的任务移出阻塞状态之前，流缓冲区中必须存在的字节数。例如，如果在读取具有触发级别1的空流缓冲区时阻止了任务，则当将单个字节写入缓冲区或任务的阻止时间到期时，任务将被取消阻止。作为另一个示例，如果在读取触发器级别为10的空流缓冲区时阻止了任务，则直到该流缓冲区包含至少10个字节或任务的阻止时间到期后，该任务才会被取消阻止。如果读取任务的阻止时间在达到触发级别之前已过期，则该任务仍将接收，但是实际上有许多字节可用。将触发级别设置为0将导致使用触发级别1。指定的触发级别大于缓冲区大小是无效的。 |

**返回值：**

如果返回NULL，则无法创建流缓冲区，因为FreeRTOS没有足够的堆内存来分配流缓冲区的数据结构和存储区域。返回的非NULL值表示流缓冲区已成功创建-返回的值应存储为创建的流缓冲区的句柄。

**用法示例：**

**void vAFunction( void )**

**{**

**StreamBufferHandle\_t xStreamBuffer;**

**const size\_t xStreamBufferSizeBytes = 100, xTriggerLevel = 10;**

**/\* Create a stream buffer that can hold 100 bytes. The memory used to hold**

**both the stream buffer structure and the data in the stream buffer is**

**allocated dynamically. \*/**

**xStreamBuffer = xStreamBufferCreate( xStreamBufferSizeBytes, xTriggerLevel );**

**if( xStreamBuffer == NULL )**

**{**

**/\* There was not enough heap memory space available to create the**

**stream buffer. \*/**

**}**

**else**

**{**

**/\* The stream buffer was created successfully and can now be used. \*/**

**}**

**}**

# xStreamBufferCreateStatic() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**StreamBufferHandle\_t xStreamBufferCreateStatic(**

**size\_t xBufferSizeBytes,**

**size\_t xTriggerLevelBytes,**

**uint8\_t \*pucStreamBufferStorageArea,**

**StaticStreamBuffer\_t \*pxStaticStreamBuffer );**

使用静态分配的内存 创建一个新的[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)。有关使用动态分配的内存的版本，请参见 [xStreamBufferCreate（）](https://www.freertos.org/xStreamBufferCreate.html)。

[configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html#configSUPPORT_STATIC_ALLOCATION) 必须FreeRTOSConfig.h中被设置为1用于xStreamBufferCreateStatic（）是可用的。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xBufferSizeBytes | pucStreamBufferStorageArea参数指向的缓冲区的大小（以字节为单位）。 |
| xTriggerLevelBytes | 在流缓冲区中被阻塞以等待数据的任务移出阻塞状态之前，流缓冲区中必须存在的字节数。例如，如果在读取具有触发级别1的空流缓冲区时阻止了任务，则当将单个字节写入缓冲区或任务的阻止时间到期时，任务将被取消阻止。作为另一个示例，如果在读取触发器级别为10的空流缓冲区时阻止了任务，则直到该流缓冲区包含至少10个字节或任务的阻止时间到期后，该任务才会被取消阻止。如果读取任务的阻止时间在达到触发级别之前已过期，则该任务仍将接收，但是实际上有许多字节可用。将触发级别设置为0将导致使用触发级别1。指定的触发级别大于缓冲区大小是无效的。 |
| pucStreamBufferStorageArea | 必须指向至少xBufferSizeBytes + 1个大的uint8\_t数组。这是将流写入流缓冲区时将复制到的数组。 |
| pxStaticStreamBuffer | 必须指向StaticStreamBuffer\_t类型的变量，该变量将用于保存流缓冲区的数据结构。 |

**返回值：**

如果成功创建了流缓冲区，则返回创建的流缓冲区的句柄。如果pucStreamBufferStorageArea或pxStaticstreamBuffer为NULL，则返回NULL。

**用法示例：**

**/\* Used to dimension the array used to hold the streams. The available space**

**will actually be one less than this, so 999. \*/**

**#define STORAGE\_SIZE\_BYTES 1000**

**/\* Defines the memory that will actually hold the streams within the stream**

**buffer. \*/**

**static uint8\_t ucBufferStorage[ STORAGE\_SIZE\_BYTES ];**

**/\* The variable used to hold the stream buffer structure. \*/**

**StaticStreamBuffer\_t xStreamBufferStruct;**

**void MyFunction( void )**

**{**

**StreamBufferHandle\_t xStreamBuffer;**

**const size\_t xTriggerLevel = 1;**

**xStreamBuffer = xStreamBufferCreateStatic( sizeof( ucBufferStorage ),**

**xTriggerLevel,**

**ucBufferStorage,**

**&xStreamBufferStruct );**

**/\* As neither the pucStreamBufferStorageArea or pxStaticStreamBuffer**

**parameters were NULL, xStreamBuffer will not be NULL, and can be used to**

**reference the created stream buffer in other stream buffer API calls. \*/**

**/\* Other code that uses the stream buffer can go here. \*/**

**}**

# xStreamBufferSend() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**size\_t xStreamBufferSend( StreamBufferHandle\_t xStreamBuffer,**

**const void \*pvTxData,**

**size\_t xDataLengthBytes,**

**TickType\_t xTicksToWait );**

将字节发送到[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)。字节被复制到流缓冲区中。

**注意**：在FreeRTOS对象中唯一的是，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）假定只有一个任务或中断将写入缓冲区（写程序），并且仅一个将从缓冲区（读取器）读取的任务或中断。对于作者和读者来说，不同的任务或中断是安全的，但是，与其他FreeRTOS对象不同，拥有多个不同的作者或多个不同的读者是不安全的。如果要有多个不同的编写器，则应用程序编写器必须将对编写API函数（例如xStreamBufferSend（））的每个调用放在关键部分内，并使用0的发送块时间。同样，

使用xStreamBufferSend（）从任务写入流缓冲区。使用xStreamBufferSendFromISR（）从中断服务例程（ISR）写入流缓冲区。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 将流发送到的流缓冲区的句柄。 |
| pvTxData | 指向缓冲区的指针，该缓冲区保存要复制到流缓冲区的字节。 |
| xDataLengthBytes | 从pvTxData复制到流缓冲区的最大字节数。 |
| xTicksToWait | 如果流缓冲区包含的空间太小而无法容纳其他xDataLengthBytes字节，则任务应保持在“阻塞”状态的最大时间，以等待流缓冲区中有足够的空间可用。块时间以滴答周期指定，因此它表示的绝对时间取决于滴答频率。宏pdMS\_TO\_TICKS（）可用于将以毫秒为单位的时间转换为以刻度为单位的时间。如果在FreeRTOSConfig.h中将INCLUDE\_vTaskSuspend设置为1，则将xTicksToWait设置为portMAX\_DELAY将导致任务无限期等待（无超时）。如果某个任务在可以将所有xDataLengthBytes写入缓冲区之前超时，它仍将写入尽可能多的字节。 |

**返回值：**

写入流缓冲区的字节数。如果某个任务在可以将所有xDataLengthBytes写入缓冲区之前超时，它仍将写入尽可能多的字节。

**用法示例：**

**void vAFunction( StreamBufferHandle\_t xStreamBuffer )**

**{**

**size\_t xBytesSent;**

**uint8\_t ucArrayToSend[] = { 0, 1, 2, 3 };**

**char \*pcStringToSend = "String to send";**

**const TickType\_t x100ms = pdMS\_TO\_TICKS( 100 );**

**/\* Send an array to the stream buffer, blocking for a maximum of 100ms to**

**wait for enough space to be available in the stream buffer. \*/**

**xBytesSent = xStreamBufferSend( xStreamBuffer,**

**( void \* ) ucArrayToSend,**

**sizeof( ucArrayToSend ),**

**x100ms );**

**if( xBytesSent != sizeof( ucArrayToSend ) )**

**{**

**/\* The call to xStreamBufferSend() times out before there was enough**

**space in the buffer for the data to be written, but it did**

**successfully write xBytesSent bytes. \*/**

**}**

**/\* Send the string to the stream buffer. Return immediately if there is not**

**enough space in the buffer. \*/**

**xBytesSent = xStreamBufferSend( xStreamBuffer,**

**( void \* ) pcStringToSend,**

**strlen( pcStringToSend ), 0 );**

**if( xBytesSent != strlen( pcStringToSend ) )**

**{**

**/\* The entire string could not be added to the stream buffer because**

**there was not enough free space in the buffer, but xBytesSent bytes**

**were sent. Could try again to send the remaining bytes. \*/**

**}**

**}**

# xStreamBufferSendFromISR() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**size\_t xStreamBufferSendFromISR( StreamBufferHandle\_t xStreamBuffer,**

**const void \*pvTxData,**

**size\_t xDataLengthBytes,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

API函数的中断安全版本，用于将字节流发送到[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)。

**注意**：在FreeRTOS对象中唯一的是，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）假定只有一个任务或中断将写入缓冲区（写程序），并且仅一个将从缓冲区（读取器）读取的任务或中断。对于作者和读者来说，不同的任务或中断是安全的，但是，与其他FreeRTOS对象不同，拥有多个不同的作者或多个不同的读者是不安全的。如果要有多个不同的编写器，则应用程序编写器必须将对编写API函数（例如xStreamBufferSend（））的每个调用放在关键部分内，并使用0的发送块时间。同样，

使用xStreamBufferSend（）从任务写入流缓冲区。使用xStreamBufferSendFromISR（）从中断服务例程（ISR）写入流缓冲区。

通过 在构建中包含FreeRTOS/source/stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 将流发送到的流缓冲区的句柄。 |
| pvTxData | 指向缓冲区的指针，该缓冲区保存要复制到流缓冲区的字节。 |
| xDataLengthBytes | 从pvTxData复制到流缓冲区的最大字节数。 |
| pxHigherPriorityTaskWoken | 流缓冲区可能会阻塞任务以等待数据。调用xStreamBufferSendFromISR（）可以使数据可用，从而导致正在等待数据的任务离开“已阻止”状态。如果调用xStreamBufferSendFromISR（）导致任务退出“阻塞”状态，并且未阻塞的任务的优先级高于当前正在执行的任务（被中断的任务），则内部，xStreamBufferSendFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。如果xStreamBufferSendFromISR（）将此值设置为pdTRUE，则通常应在退出中断之前执行上下文切换。这将确保中断直接返回到最高优先级的就绪状态任务。在将pxHigherPriorityTaskWoken传递给函数之前，应将其设置为pdFALSE。有关示例，请参见下面的示例代码。 |

**返回值：**

写入流缓冲区的字节数。如果某个任务在可以将所有xDataLengthBytes写入缓冲区之前超时，它仍将写入尽可能多的字节。

**用法示例：**

**/\* A stream buffer that has already been created. \*/**

**StreamBufferHandle\_t xStreamBuffer;**

**void vAnInterruptServiceRoutine( void )**

**{**

**size\_t xBytesSent;**

**char \*pcStringToSend = "String to send";**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE; /\* Initialised to pdFALSE. \*/**

**/\* Attempt to send the string to the stream buffer. \*/**

**xBytesSent = xStreamBufferSendFromISR( xStreamBuffer,**

**( void \* ) pcStringToSend,**

**strlen( pcStringToSend ),**

**&xHigherPriorityTaskWoken );**

**if( xBytesSent != strlen( pcStringToSend ) )**

**{**

**/\* There was not enough free space in the stream buffer for the entire**

**string to be written, ut xBytesSent bytes were written. \*/**

**}**

**/\* If xHigherPriorityTaskWoken was set to pdTRUE inside**

**xStreamBufferSendFromISR() then a task that has a priority above the**

**priority of the currently executing task was unblocked and a context**

**switch should be performed to ensure the ISR returns to the unblocked**

**task. In most FreeRTOS ports this is done by simply passing**

**xHigherPriorityTaskWoken into taskYIELD\_FROM\_ISR(), which will test the**

**variables value, and perform the context switch if necessary. Check the**

**documentation for the port in use for port specific instructions. \*/**

**taskYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

# xStreamBufferReceive() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**size\_t xStreamBufferReceive( StreamBufferHandle\_t xStreamBuffer,**

**void \*pvRxData,**

**size\_t xBufferLengthBytes,**

**TickType\_t xTicksToWait );**

从[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)接收字节。

**注意**：在FreeRTOS对象中唯一的是，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）假定只有一个任务或中断将写入缓冲区（写程序），并且仅一个将从缓冲区（读取器）读取的任务或中断。对于作者和读者来说，不同的任务或中断是安全的，但是，与其他FreeRTOS对象不同，拥有多个不同的作者或多个不同的读者是不安全的。如果要有多个不同的编写器，则应用程序编写器必须将对编写API函数（例如xStreamBufferSend（））的每个调用放在关键部分内，并使用0的发送块时间。同样，

使用xStreamBufferReceive（）从任务的流缓冲区中读取。使用xStreamBufferReceiveFromISR（）从中断服务例程（ISR）读取流缓冲区。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 要从中接收字节的流缓冲区的句柄。 |
| pvRxData | 指向将接收到的字节复制到的缓冲区的指针。 |
| xBufferLengthBytes | pvRxData参数指向的缓冲区的长度。设置一次呼叫中要接收的最大字节数。xStreamBufferReceive将返回尽可能多的字节，直到xBufferLengthBytes设置的最大值为止。 |
| xTicksToWait | 如果流缓冲区为空，任务应保持在“阻塞”状态以等待数据变为可用的最长时间。如果xTicksToWait为零，则xStreamBufferReceive（）将立即返回。块时间以滴答周期指定，因此它表示的绝对时间取决于滴答频率。宏pdMS\_TO\_TICKS（）可用于将以毫秒为单位的时间转换为以刻度为单位的时间。如果在FreeRTOSConfig.h中将INCLUDE\_vTaskSuspend设置为1，则将xTicksToWait设置为portMAX\_DELAY将导致任务无限期等待（无超时）。处于“阻塞”状态的任务不占用任何CPU时间。 |

**返回值：**

从流缓冲区实际读取的字节数，如果对xStreamBufferReceive（）的调用在xBufferLengthBytes可用之前超时，则将小于xBufferLengthBytes。

**用法示例：**

**void vAFunction( StreamBuffer\_t xStreamBuffer )**

**{**

**uint8\_t ucRxData[ 20 ];**

**size\_t xReceivedBytes;**

**const TickType\_t xBlockTime = pdMS\_TO\_TICKS( 20 );**

**/\* Receive up to another sizeof( ucRxData ) bytes from the stream buffer.**

**Wait in the Blocked state (so not using any CPU processing time) for a**

**maximum of 100ms for the full sizeof( ucRxData ) number of bytes to be**

**available. \*/**

**xReceivedBytes = xStreamBufferReceive( xStreamBuffer,**

**( void \* ) ucRxData,**

**sizeof( ucRxData ),**

**xBlockTime );**

**if( xReceivedBytes > 0 )**

**{**

**/\* A ucRxData contains another xRecievedBytes bytes of data, which can**

**be processed here.... \*/**

**}**

**}**

# xStreamBufferReceiveFromISR() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**size\_t xStreamBufferReceiveFromISR( StreamBufferHandle\_t xStreamBuffer,**

**void \*pvRxData,**

**size\_t xBufferLengthBytes,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

API函数的中断安全版本，可从[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)接收字节 。

**注意**：在FreeRTOS对象中唯一的是，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）假定只有一个任务或中断将写入缓冲区（写程序），并且仅一个将从缓冲区（读取器）读取的任务或中断。对于作者和读者来说，不同的任务或中断是安全的，但是，与其他FreeRTOS对象不同，拥有多个不同的作者或多个不同的读者是不安全的。如果要有多个不同的编写器，则应用程序编写器必须将对编写API函数（例如xStreamBufferSend（））的每个调用放在关键部分内，并使用0的发送块时间。同样，

使用xStreamBufferReceive（）从任务的流缓冲区中读取。使用xStreamBufferReceiveFromISR（）从中断服务例程（ISR）读取流缓冲区。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 要从中接收字节的流缓冲区的句柄。 |
| pvRxData | 指向将接收到的字节复制到的缓冲区的指针。 |
| xBufferLengthBytes | pvRxData参数指向的缓冲区的长度。设置一次呼叫中要接收的最大字节数。xStreamBufferReceive将返回尽可能多的字节，直到xBufferLengthBytes设置的最大值为止。 |
| pxHigherPriorityTaskWoken | 流缓冲区可能会阻塞任务，以等待空间可用。调用xStreamBufferReceiveFromISR（）可以使空间可用，因此导致正在等待空间的任务离开“已阻塞”状态。如果调用xStreamBufferReceiveFromISR（）导致任务退出“阻塞”状态，并且未阻塞任务的优先级高于当前正在执行的任务（被中断的任务），则内部xStreamBufferReceiveFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。如果xStreamBufferReceiveFromISR（）将此值设置为pdTRUE，则通常应在退出中断之前执行上下文切换。这样可以确保中断直接返回到最高优先级的就绪状态任务。在将pxHigherPriorityTaskWoken传递给函数之前，应将其设置为pdFALSE。有关示例，请参见下面的代码示例。 |

**返回值：**

从流缓冲区读取的字节数（如果有）。

**用法示例：**

**/\* A stream buffer that has already been created. \*/**

**StreamBuffer\_t xStreamBuffer;**

**void vAnInterruptServiceRoutine( void )**

**{**

**uint8\_t ucRxData[ 20 ];**

**size\_t xReceivedBytes;**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE; /\* Initialised to pdFALSE. \*/**

**/\* Receive the next stream from the stream buffer. \*/**

**xReceivedBytes = xStreamBufferReceiveFromISR( xStreamBuffer,**

**( void \* ) ucRxData,**

**sizeof( ucRxData ),**

**&xHigherPriorityTaskWoken );**

**if( xReceivedBytes > 0 )**

**{**

**/\* ucRxData contains xReceivedBytes read from the stream buffer.**

**Process the stream here.... \*/**

**}**

**/\* If xHigherPriorityTaskWoken was set to pdTRUE inside**

**xStreamBufferReceiveFromISR() then a task that has a priority above the**

**priority of the currently executing task was unblocked and a context**

**switch should be performed to ensure the ISR returns to the unblocked**

**task. In most FreeRTOS ports this is done by simply passing**

**xHigherPriorityTaskWoken into taskYIELD\_FROM\_ISR(), which will test the**

**variables value, and perform the context switch if necessary. Check the**

**documentation for the port in use for port specific instructions. \*/**

**taskYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

# vStreamBufferDelete() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**void vStreamBufferDelete( StreamBufferHandle\_t xStreamBuffer );**

删除以前使用[xStreamBufferCreate（）](https://www.freertos.org/xStreamBufferCreate.html)或[xStreamBufferCreateStatic（）](https://www.freertos.org/xStreamBufferCreateStatic.html)调用创建 的[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)。如果流缓冲区是使用动态内存创建的（即通过xStreamBufferCreate（）创建），则释放已分配的内存。

删除流缓冲区后，不得使用流缓冲区句柄。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 流缓冲区的句柄要删除。 |

# xStreamBufferBytesAvailable() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**size\_t xStreamBufferBytesAvailable( StreamBufferHandle\_t xStreamBuffer );**

查询[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)以查看其包含多少数据，该数据等于在流缓冲区为空之前可以从流缓冲区读取的字节数。

通过 在构建中包含FreeRTOS/source/stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 正在查询的流缓冲区的句柄。 |

**返回值：**

流缓冲区为空之前可以从流缓冲区读取的字节数。

# xStreamBufferSpacesAvailable() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**size\_t xStreamBufferSpacesAvailable( StreamBufferHandle\_t xStreamBuffer );**

查询[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)以查看其包含的可用空间，该可用空间等于在流缓冲区已满之前可以发送给流缓冲区的数据量。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 正在查询的流缓冲区的句柄。 |

**返回值：**

在流缓冲区已满之前可以写入流缓冲区的字节数。

# xStreamBufferSetTriggerLevel() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**BaseType\_t xStreamBufferSetTriggerLevel( StreamBufferHandle\_t xStreamBuffer,**

**size\_t xTriggerLevel );**

一个[流缓冲区的](https://www.freertos.org/RTOS-stream-buffer-API.html)触发级别是指在流缓冲区中被阻止以等待数据移出阻止状态的任务之前，流缓冲区中必须存在的字节数。例如，如果在读取具有触发级别1的空流缓冲区时阻止了任务，则当将单个字节写入缓冲区或任务的阻止时间到期时，任务将被取消阻止。作为另一个示例，如果在读取触发器级别为10的空流缓冲区时阻止了任务，则直到该流缓冲区包含至少10个字节或任务的阻止时间到期后，该任务才会被取消阻止。如果读取任务的阻止时间在达到触发级别之前已过期，则该任务仍将接收，但是实际上有许多字节可用。将触发级别设置为0将导致使用触发级别1。

创建流缓冲区时设置触发级别，并且可以使用xStreamBufferSetTriggerLevel（）进行修改。

通过在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 流缓冲区的句柄正在更新。 |
| xTriggerLevel | 流缓冲区的新触发级别。 |

**返回值：**

如果xTriggerLevel小于或等于流缓冲区的长度，则将更新触发级别并返回pdTRUE。否则，返回pdFALSE。

# xStreamBufferReset() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**BaseType\_t xStreamBufferReset( StreamBufferHandle\_t xStreamBuffer );**

将[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)重置为其初始的空状态。流缓冲区中的所有数据都将被丢弃。仅当没有阻塞等待发送或接收流缓冲区的任务时，才可以重置流缓冲区。

通过 在构建中包含FreeRTOS/source/stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 流缓冲区的句柄被重置。 |

**返回值：**

如果流缓冲区被重置，则返回pdPASS。如果某个任务被阻塞，等待发送到流缓冲区或从流缓冲区中读取，则该流缓冲区将不会被重置，并且将返回pdFAIL。

# xStreamBufferIsEmpty() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**BaseType\_t xStreamBufferIsEmpty( StreamBufferHandle\_t xStreamBuffer );**

查询[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)以查看其是否为空。如果流缓冲区不包含任何数据，则为空。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 正在查询的流缓冲区的句柄。 |

**返回值：**

如果流缓冲区为空，则返回pdTRUE。否则，返回pdFALSE。

# xStreamBufferIsFull() [[RTOS Stream Buffer API](https://www.freertos.org/RTOS-stream-buffer-API.html)]

stream\_buffer.h

**BaseType\_t xStreamBufferIsFull( StreamBufferHandle\_t xStreamBuffer );**

查询[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)以查看其是否已满。如果流缓冲区没有可用空间，则它已满，因此不能再接受任何数据。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

**参数：**

|  |  |
| --- | --- |
| xStreamBuffer | 正在查询的流缓冲区的句柄。 |

**返回值：**

如果流缓冲区已满，则返回pdTRUE。否则，返回pdFALSE。

# RTOS消息缓冲区

## [RTOS消息缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)API函数：

* [**xMessageBufferCreate（）**](https://www.freertos.org/xMessageBufferCreate.html)
* [**xMessageBufferCreateStatic（）**](https://www.freertos.org/xMessageBufferCreateStatic.html)
* [**xMessageBufferSend（）**](https://www.freertos.org/xMessageBufferSend.html)
* [**xMessageBufferSendFromISR（）**](https://www.freertos.org/xMessageBufferSendFromISR.html)
* [**xMessageBufferReceive（）**](https://www.freertos.org/xMessageBufferReceive.html)
* [**xMessageBufferReceiveFromISR（）**](https://www.freertos.org/xMessageBufferReceiveFromISR.html)
* [**vMessageBufferDelete（）**](https://www.freertos.org/vMessageBufferDelete.html)
* [**xMessageBufferSpacesAvailable（）**](https://www.freertos.org/xMessageBufferSpacesAvailable.html)
* [**xMessageBufferReset（）**](https://www.freertos.org/xMessageBufferReset.html)
* [**xMessageBufferIsEmpty（）**](https://www.freertos.org/xMessageBufferIsEmpty.html)
* [**xMessageBufferIsFull（）**](https://www.freertos.org/xMessageBufferIsFull.html)

# xMessageBufferCreate() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**MessageBufferHandle\_t xMessageBufferCreate( size\_t xBufferSizeBytes );**

使用动态分配的内存 创建一个新的[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)。有关使用静态分配的内存（在编译时分配的内存[）](https://www.freertos.org/xMessageBufferCreateStatic.html)的版本，请参见 [xMessageBufferCreateStatic（](https://www.freertos.org/xMessageBufferCreateStatic.html)）。

必须将[configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html#configSUPPORT_DYNAMIC_ALLOCATION)设置为1或在FreeRTOSConfig.h中未定义才能使xMessageBufferCreate（）可用。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xBufferSizeBytes | 消息缓冲区可以随时保留的字节总数（不是消息）。将消息写入消息缓冲区时，还会写入另外的sizeof（size\_t）字节来存储消息的长度。sizeof（size\_t）在32位体系结构上通常为4个字节，因此在大多数32位体系结构上，一个10字节的消息将占用14个字节的消息缓冲区空间。 |

**返回值：**

如果返回NULL，则无法创建消息缓冲区，因为FreeRTOS没有足够的堆内存来分配消息缓冲区的数据结构和存储区域。返回的非NULL值表示消息缓冲区已成功创建-返回的值应存储为创建的消息缓冲区的句柄。

**用法示例：**

**void vAFunction( void )**

**{**

**MessageBufferHandle\_t xMessageBuffer;**

**const size\_t xMessageBufferSizeBytes = 100;**

**/\* Create a message buffer that can hold 100 bytes. The memory used to hold**

**both the message buffer structure and the data in the message buffer is**

**allocated dynamically. \*/**

**xMessageBuffer = xMessageBufferCreate( xMessageBufferSizeBytes );**

**if( xMessageBuffer == NULL )**

**{**

**/\* There was not enough heap memory space available to create the**

**message buffer. \*/**

**}**

**else**

**{**

**/\* The message buffer was created successfully and can now be used. \*/**

**}**

**}**

# xMessageBufferCreateStatic() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**MessageBufferHandle\_t xMessageBufferCreateStatic(**

**size\_t xBufferSizeBytes,**

**uint8\_t \*pucMessageBufferStorageArea,**

**StaticMessageBuffer\_t \*pxStaticMessageBuffer );**

使用静态分配的内存 创建一个新的[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)。有关使用动态分配的内存的版本，请参见 [xMessageBufferCreate（）](https://www.freertos.org/xMessageBufferCreate.html)。

[configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html#configSUPPORT_STATIC_ALLOCATION) 必须FreeRTOSConfig.h中被设置为1用于xMessageBufferCreateStatic（）是可用的。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xBufferSizeBytes | pucMessageBufferStorageArea参数指向的缓冲区的大小（以字节为单位）。将消息写入消息缓冲区时，还会写入另外的sizeof（size\_t）字节来存储消息的长度。sizeof（size\_t）在32位体系结构上通常为4个字节，因此在大多数32位体系结构上，一个10字节的消息将占用14个字节的消息缓冲区空间。实际上，可以在消息缓冲区中存储的最大字节数为（xBufferSizeBytes – 1）。 |
| pucMessageBufferStorageArea | 必须指向至少xBufferSizeBytes + 1个大的uint8\_t数组。这是将消息写入消息缓冲区时要复制到的数组。 |
| pxStaticMessageBuffer | 必须指向类型为StaticMessageBuffer\_t的变量，该变量将用于保存消息缓冲区的数据结构。 |

**返回值：**

如果成功创建了消息缓冲区，则返回创建的消息缓冲区的句柄。如果pucMessageBufferStorageArea或pxStaticMessageBuffer为NULL，则返回NULL。

**用法示例：**

**/\* Used to dimension the array used to hold the messages. The available space**

**will actually be one less than this, so 999. \*/**

**#define STORAGE\_SIZE\_BYTES 1000**

**/\* Defines the memory that will actually hold the messages within the message**

**buffer. Should be one more than the value passed in the xBufferSizeBytes**

**parameter. \*/**

**static uint8\_t ucStorageBuffer[ STORAGE\_SIZE\_BYTES ];**

**/\* The variable used to hold the message buffer structure. \*/**

**StaticMessageBuffer\_t xMessageBufferStruct;**

**void MyFunction( void )**

**{**

**MessageBufferHandle\_t xMessageBuffer;**

**xMessageBuffer = xMessageBufferCreateStatic( sizeof( ucStoragegBuffer ),**

**ucBufferStorage,**

**&xMessageBufferStruct );**

**/\* As neither the pucMessageBufferStorageArea or pxStaticMessageBuffer**

**parameters were NULL, xMessageBuffer will not be NULL, and can be used to**

**reference the created message buffer in other message buffer API calls. \*/**

**/\* Other code that uses the message buffer can go here. \*/**

**}**

# xMessageBufferSend() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**size\_t xMessageBufferSend( MessageBufferHandle\_t xMessageBuffer,**

**const void \*pvTxData,**

**size\_t xDataLengthBytes,**

**TickType\_t xTicksToWait );**

将离散消息发送到[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)。该消息的长度可以适合缓冲区的可用空间，并且可以复制到缓冲区中。

**注意**：在FreeRTOS对象中唯一的是，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）假定只有一个任务或中断将写入缓冲区（写程序），并且仅一个将从缓冲区（读取器）读取的任务或中断。对于作者和读者来说，不同的任务或中断是安全的，但是，与其他FreeRTOS对象不同，拥有多个不同的作者或多个不同的读者是不安全的。如果要有多个不同的编写器，则应用程序编写器必须将对编写API函数（例如xMessageBufferSend（））的每个调用放在关键部分内，并使用0的发送块时间。同样，

使用xMessageBufferSend（）从任务写入消息缓冲区。使用 [xMessageBufferSendFromISR（）](https://www.freertos.org/xMessageBufferSendFromISR.html)从中断服务例程（ISR）写入消息缓冲区。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xMessageBuffer | 将消息发送到的消息缓冲区的句柄。 |
| pvTxData | 指向要复制到消息缓冲区的消息的指针。 |
| xDataLengthBytes | 消息的长度。即，从pvTxData复制到消息缓冲区的字节数。将消息写入消息缓冲区时，还会写入另外的sizeof（size\_t）字节来存储消息的长度。sizeof（size\_t）在32位体系结构上通常为4个字节，因此在大多数32位体系结构上，将xDataLengthBytes设置为20会将消息缓冲区中的可用空间减少24个字节（20个字节的消息数据和4个字节来保存字节）。邮件长度）。 |
| xTicksToWait | xTicksToWait如果在调用xMessageBufferSend（）时消息缓冲区没有足够的空间，则调用任务应保持在“阻塞”状态以等待足够的空间以等待消息缓冲区中可用的最长时间。如果xTicksToWait为零，则调用任务将永远不会阻塞。块时间以滴答周期指定，因此它表示的绝对时间取决于滴答频率。宏pdMS\_TO\_TICKS（）可用于将以毫秒为单位的时间转换为以刻度为单位的时间。如果在FreeRTOSConfig.h中将INCLUDE\_vTaskSuspend设置为1，则将xTicksToWait设置为portMAX\_DELAY将导致任务无限期等待（无超时）。处于“阻塞”状态的任务不占用任何CPU时间。 |

**返回值：**

写入消息缓冲区的字节数。如果在有足够的空间将消息写入消息缓冲区之前，对xMessageBufferSend（）的调用超时，则返回零。如果呼叫没有超时，则返回xDataLengthBytes。

**用法示例：**

**void vAFunction( MessageBufferHandle\_t xMessageBuffer )**

**{**

**size\_t xBytesSent;**

**uint8\_t ucArrayToSend[] = { 0, 1, 2, 3 };**

**char \*pcStringToSend = "String to send";**

**const TickType\_t x100ms = pdMS\_TO\_TICKS( 100 );**

**/\* Send an array to the message buffer, blocking for a maximum of 100ms to**

**wait for enough space to be available in the message buffer. \*/**

**xBytesSent = xMessageBufferSend( xMessageBuffer,**

**( void \* ) ucArrayToSend,**

**sizeof( ucArrayToSend ),**

**x100ms );**

**if( xBytesSent != sizeof( ucArrayToSend ) )**

**{**

**/\* The call to xMessageBufferSend() times out before there was enough**

**space in the buffer for the data to be written. \*/**

**}**

**/\* Send the string to the message buffer. Return immediately if there is**

**not enough space in the buffer. \*/**

**xBytesSent = xMessageBufferSend( xMessageBuffer,**

**( void \* ) pcStringToSend,**

**strlen( pcStringToSend ), 0 );**

**if( xBytesSent != strlen( pcStringToSend ) )**

**{**

**/\* The string could not be added to the message buffer because there was**

**not enough free space in the buffer. \*/**

**}**

**}**

# xMessageBufferSendFromISR() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**size\_t xMessageBufferSendFromISR( MessageBufferHandle\_t xMessageBuffer,**

**const void \*pvTxData,**

**size\_t xDataLengthBytes,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

API函数的中断安全版本，可将离散消息发送到[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)。该消息的长度可以适合缓冲区的可用空间，并且可以复制到缓冲区中。

**注意**：在FreeRTOS对象中唯一的是，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）假定只有一个任务或中断将写入缓冲区（写程序），并且仅一个将从缓冲区（读取器）读取的任务或中断。对于作者和读者来说，不同的任务或中断是安全的，但是，与其他FreeRTOS对象不同，拥有多个不同的作者或多个不同的读者是不安全的。如果要有多个不同的编写器，则应用程序编写器必须将对编写API函数（例如xMessageBufferSend（））的每个调用放在关键部分内，并使用0的发送块时间。同样，

使用[xMessageBufferSend（）](https://www.freertos.org/xMessageBufferSend.html)从任务写入消息缓冲区。使用xMessageBufferSendFromISR（）从中断服务例程（ISR）写入消息缓冲区。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xMessageBuffer | 将消息发送到的消息缓冲区的句柄。 |
| pvTxData | 指向要复制到消息缓冲区的消息的指针。 |
| xDataLengthBytes | 消息的长度。即，从pvTxData复制到消息缓冲区的字节数。将消息写入消息缓冲区时，还会写入另外的sizeof（size\_t）字节来存储消息的长度。sizeof（size\_t）在32位体系结构上通常为4个字节，因此在大多数32位体系结构上，将xDataLengthBytes设置为20会将消息缓冲区中的可用空间减少24个字节（20个字节的消息数据和4个字节来保存字节）。邮件长度）。 |
| pxHigherPriorityTaskWoken | 消息缓冲区可能会阻塞任务以等待数据。调用xMessageBufferSendFromISR（）可以使数据可用，因此导致正在等待数据的任务离开“已阻止”状态。如果调用xMessageBufferSendFromISR（）导致任务退出“阻塞”状态，并且未阻塞任务的优先级高于当前正在执行的任务（被中断的任务），则内部，xMessageBufferSendFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。如果xMessageBufferSendFromISR（）将此值设置为pdTRUE，则通常应在退出中断之前执行上下文切换。这将确保中断直接返回到最高优先级的就绪状态任务。在将pxHigherPriorityTaskWoken传递给函数之前，应将其设置为pdFALSE。有关示例，请参见下面的代码示例。 |

**返回值：**

实际写入消息缓冲区的字节数。如果消息缓冲区没有足够的可用空间来存储消息，则返回0，否则返回xDataLengthBytes。

**用法示例：**

**/\* A message buffer that has already been created. \*/**

**MessageBufferHandle\_t xMessageBuffer;**

**void vAnInterruptServiceRoutine( void )**

**{**

**size\_t xBytesSent;**

**char \*pcStringToSend = "String to send";**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE; /\* Initialised to pdFALSE. \*/**

**/\* Attempt to send the string to the message buffer. \*/**

**xBytesSent = xMessageBufferSendFromISR( xMessageBuffer,**

**( void \* ) pcStringToSend,**

**strlen( pcStringToSend ),**

**&xHigherPriorityTaskWoken );**

**if( xBytesSent != strlen( pcStringToSend ) )**

**{**

**/\* The string could not be added to the message buffer because there was**

**not enough free space in the buffer. \*/**

**}**

**/\* If xHigherPriorityTaskWoken was set to pdTRUE inside**

**xMessageBufferSendFromISR() then a task that has a priority above the**

**priority of the currently executing task was unblocked and a context**

**switch should be performed to ensure the ISR returns to the unblocked**

**task. In most FreeRTOS ports this is done by simply passing**

**xHigherPriorityTaskWoken into taskYIELD\_FROM\_ISR(), which will test the**

**variables value, and perform the context switch if necessary. Check the**

**documentation for the port in use for port specific instructions. \*/**

**taskYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

# xMessageBufferReceive() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**size\_t xMessageBufferReceive( MessageBufferHandle\_t xMessageBuffer,**

**void \*pvRxData,**

**size\_t xBufferLengthBytes,**

**TickType\_t xTicksToWait );**

从RTOS[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)接收离散消息。消息的长度可变，并且可以从缓冲区中复制出来。

**注意**：在FreeRTOS对象中唯一的是，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）假定只有一个任务或中断将写入缓冲区（写程序），并且仅一个将从缓冲区（读取器）读取的任务或中断。对于写入对象和读取对象来说，不同的任务或中断是安全的，但是，与其他FreeRTOS对象不同，拥有多个不同的写入对象或多个不同的读取对象是不安全的。如果要有多个不同的写入对象，则应用程序编写器必须将对编写API函数（例如xMessageBufferSend（））的每个调用放在关键部分内，并使用0的发送阻塞时间。同样，使用xMessageBufferReceive（）从任务的消息缓冲区中读取。使用 [xMessageBufferReceiveFromISR（）](https://www.freertos.org/xMessageBufferReceiveFromISR.html)从中断服务例程（ISR）读取消息缓冲区。

通过在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xMessageBuffer | 从中接收消息的消息缓冲区的句柄。 |
| pvRxData | 指向要将接收到的消息复制到的缓冲区的指针。 |
| xBufferLengthBytes | pvRxData参数指向的缓冲区的长度。这设置了可以接收的最大消息长度。如果xBufferLengthBytes太小而无法容纳下一条消息，则该消息将保留在消息缓冲区中，并且将返回0。 |
| xTicksToWait | 如果在调用xMessageBufferReceive（）时消息缓冲区为空，则任务应保持在“阻塞”状态以等待消息的最长时间。如果xTicksToWait为零并且消息缓冲区为空，则xMessageBufferReceive（）将立即返回。块时间以滴答周期指定，因此它表示的绝对时间取决于滴答频率。宏pdMS\_TO\_TICKS（）可用于将以毫秒为单位的时间转换为以刻度为单位的时间。如果在FreeRTOSConfig.h中将INCLUDE\_vTaskSuspend设置为1，则将xTicksToWait设置为portMAX\_DELAY将导致任务无限期等待（无超时）。处于“阻塞”状态的任务不占用任何CPU时间。 |

**返回值：**

从消息缓冲区读取的消息的长度（以字节为单位）（如果有）。如果xMessageBufferReceive（）在消息可用之前超时，则返回零。如果消息的长度大于xBufferLengthBytes，则消息将保留在消息缓冲区中，并返回零。

**用法示例：**

**void vAFunction( MessageBuffer\_t xMessageBuffer )**

**{**

**uint8\_t ucRxData[ 20 ];**

**size\_t xReceivedBytes;**

**const TickType\_t xBlockTime = pdMS\_TO\_TICKS( 20 );**

**/\* Receive the next message from the message buffer. Wait in the Blocked state (so not using any CPU processing time) for a maximum of 100ms for**

**a message to become available. \*/**

**xReceivedBytes = xMessageBufferReceive( xMessageBuffer,**

**( void \* ) ucRxData,**

**sizeof( ucRxData ),**

**xBlockTime );**

**if( xReceivedBytes > 0 )**

**{**

**/\* A ucRxData contains a message that is xReceivedBytes long. Process the message here.... \*/**

**}**

**}**

# xMessageBufferReceiveFromISR() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**size\_t xMessageBufferReceiveFromISR( MessageBufferHandle\_t xMessageBuffer,**

**void \*pvRxData,**

**size\_t xBufferLengthBytes,**

**BaseType\_t \*pxHigherPriorityTaskWoken );**

API函数的中断安全版本，可从[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)接收离散消息。消息的长度可以可变，并且可以从缓冲区中复制出来。

**注意**：在FreeRTOS对象中唯一的是，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）假定只有一个任务或中断将写入缓冲区（写程序），并且仅一个将从缓冲区（读取器）读取的任务或中断。对于作者和读者来说，不同的任务或中断是安全的，但是，与其他FreeRTOS对象不同，拥有多个不同的作者或多个不同的读者是不安全的。如果要有多个不同的编写器，则应用程序编写器必须将对编写API函数（例如xMessageBufferSend（））的每个调用放在关键部分内，并使用0的发送块时间。同样，

使用[xMessageBufferReceive（）](https://www.freertos.org/xMessageBufferReceive.html)从任务的消息缓冲区中读取。使用xMessageBufferReceiveFromISR（）从中断服务例程（ISR）读取消息缓冲区。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xMessageBuffer | 从中接收消息的消息缓冲区的句柄。 |
| pvRxData | 指向将接收到的消息复制到的缓冲区的指针。 |
| xBufferLengthBytes | pvRxData参数指向的缓冲区的长度。这设置了可以接收的最大消息长度。如果xBufferLengthBytes太小而无法容纳下一条消息，则该消息将保留在消息缓冲区中，并且将返回0。 |
| pxHigherPriorityTaskWoken | 消息缓冲区可能会阻塞任务，以等待空间可用。调用xMessageBufferReceiveFromISR（）可以使空间可用，因此导致正在等待空间的任务离开“已阻止”状态。如果调用xMessageBufferReceiveFromISR（）导致任务退出“阻塞”状态，并且未阻塞任务的优先级高于当前正在执行的任务（被中断的任务），则内部xMessageBufferReceiveFromISR（）会将\* pxHigherPriorityTaskWoken设置为pdTRUE。如果xMessageBufferReceiveFromISR（）将此值设置为pdTRUE，则通常应在退出中断之前执行上下文切换。这样可以确保中断直接返回到最高优先级的就绪状态任务。在将pxHigherPriorityTaskWoken传递给函数之前，应将其设置为pdFALSE。有关示例，请参见下面的代码示例。 |

**返回值：**

从消息缓冲区读取的消息的长度（以字节为单位）（如果有）。

**用法示例：**

**/\* A message buffer that has already been created. \*/**

**MessageBuffer\_t xMessageBuffer;**

**void vAnInterruptServiceRoutine( void )**

**{**

**uint8\_t ucRxData[ 20 ];**

**size\_t xReceivedBytes;**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE; /\* Initialised to pdFALSE. \*/**

**/\* Receive the next message from the message buffer. \*/**

**xReceivedBytes = xMessageBufferReceiveFromISR( xMessageBuffer,**

**( void \* ) ucRxData,**

**sizeof( ucRxData ),**

**&xHigherPriorityTaskWoken );**

**if( xReceivedBytes > 0 )**

**{**

**/\* A ucRxData contains a message that is xReceivedBytes long. Process**

**the message here.... \*/**

**}**

**/\* If xHigherPriorityTaskWoken was set to pdTRUE inside**

**xMessageBufferReceiveFromISR() then a task that has a priority above the**

**priority of the currently executing task was unblocked and a context**

**switch should be performed to ensure the ISR returns to the unblocked**

**task. In most FreeRTOS ports this is done by simply passing**

**xHigherPriorityTaskWoken into taskYIELD\_FROM\_ISR(), which will test the**

**variables value, and perform the context switch if necessary. Check the**

**documentation for the port in use for port specific instructions. \*/**

**taskYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

**}**

# vMessageBufferDelete() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**void vMessageBufferDelete( MessageBufferHandle\_t xMessageBuffer );**

删除以前使用[xMessageBufferCreate（）](https://www.freertos.org/xMessageBufferCreate.html)或[xMessageBufferCreateStatic（）](https://www.freertos.org/xMessageBufferCreateStatic.html)调用创建 的[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)。如果消息缓冲区是使用动态内存创建的（即通过xMessageBufferCreate（）创建），则释放已分配的内存。

删除消息缓冲区后，不得使用消息缓冲区句柄。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xMessageBuffer | 要删除的消息缓冲区的句柄。 |

# xMessageBufferSpacesAvailable() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**size\_t xMessageBufferSpacesAvailable( MessageBufferHandle\_t xMessageBuffer );**

查询[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)以查看其包含多少可用空间，该可用空间等于在消息缓冲区已满之前可以发送到消息缓冲区的数据量。返回的值比可以发送到消息缓冲区的最大消息大小大4个字节。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xMessageBuffer | 正在查询的消息缓冲区的句柄。 |

**返回值：**

在消息缓冲区已满之前可以写入消息缓冲区的字节数。将消息写入消息缓冲区时，还会写入另外的sizeof（size\_t）字节来存储消息的长度。在32位体系结构上，sizeof（size\_t）通常为4个字节，因此，如果xMessageBufferSpacesAvailable（）返回10，则可以写入消息缓冲区的最大消息的大小为6个字节。

# xMessageBufferReset() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**BaseType\_t xMessageBufferReset( MessageBufferHandle\_t xMessageBuffer );**

将[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)重置为其初始的空状态。消息缓冲区中的所有数据都将被丢弃。仅当没有阻塞等待发送或接收消息缓冲区的任务时，才可以重置消息缓冲区。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xMessageBuffer | 消息缓冲区的句柄被重置。 |

**返回值：**

如果消息缓冲区被重置，则返回pdPASS。如果某个任务被阻止等待发送到消息缓冲区或从消息缓冲区中读取消息，则消息缓冲区将不会重置，并返回pdFAIL。

# xMessageBufferIsEmpty() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**BaseType\_t xMessageBufferIsEmpty( MessageBufferHandle\_t xMessageBuffer );**

查询[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)以查看其是否为空。如果消息缓冲区不包含任何消息，则为空。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xMessageBuffer | 正在查询的消息缓冲区的句柄。 |

**返回值：**

如果消息缓冲区为空，则返回pdTRUE。否则，返回pdFALSE。

# xMessageBufferIsFull() [[RTOS Message Buffer API](https://www.freertos.org/RTOS-message-buffer-API.html)]

message\_buffer.h

**BaseType\_t xMessageBufferIsFull( MessageBufferHandle\_t xMessageBuffer );**

查询[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html)以查看其是否已满。如果消息缓冲区不能再接受任何大小的消息，则该消息缓冲区已满，直到通过从消息缓冲区中删除消息来腾出空间为止。

通过 在构建中包括FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能（因为消息缓冲区使用流缓冲区）。

**参数：**

|  |  |
| --- | --- |
| xMessageBuffer | 正在查询的消息缓冲区的句柄。 |

**返回值：**

如果消息缓冲区已满，则返回pdTRUE。否则，返回pdFALSE。

# 特定于例程的 [ [API](https://www.freertos.org/a00106.html) ]

## 模组

* [**CoRoutineHandle\_t**](https://www.freertos.org/crcreate.html#crhandle)
* [**xCoRoutineCreate**](https://www.freertos.org/crcreate.html)
* [**crDELAY**](https://www.freertos.org/crdelay.html)
* [**crQUEUE\_SEND**](https://www.freertos.org/crqueuesend.html)
* [**crQUEUE\_RECEIVE**](https://www.freertos.org/crqueuerec.html)
* [**crQUEUE\_SEND\_FROM\_ISR**](https://www.freertos.org/crsendisr.html)
* [**crQUEUE\_RECEIVE\_FROM\_ISR**](https://www.freertos.org/crrecisr.html)
* [**vCoRoutineSchedule**](https://www.freertos.org/crschedule.html)

# xCoRoutineCreate [[Co-Routine Specific](https://www.freertos.org/croutineapi.html)]

croutine.h

**BaseType\_t xCoRoutineCreate**

**(**

**crCOROUTINE\_CODE pxCoRoutineCode,**

**UBaseType\_t uxPriority,**

**UBaseType\_t uxIndex**

**);**

创建一个新的协同例程并将其添加到准备运行的协同例程列表中。

**参数：**

|  |  |
| --- | --- |
| *pxCoRoutineCode* | 指向协同例程功能的指针。协同例程功能需要特殊的语法–有关更多信息，请参见Web文档的协同例程部分。 |
| *优先级* | 相对于运行其他例程的其他例程的优先级。 |
| *uxIndex* | 用于区分执行相同功能的不同例程。有关更多信息，请参见下面的示例和Web文档的协同例程部分。 |

**返回值：**

如果成功创建了该例程并将其添加到就绪列表，则为pdPASS，否则使用ProjDefs.h定义错误代码。

**用法示例：**

**// Co-routine to be created.**

**void vFlashCoRoutine( CoRoutineHandle\_t xHandle, UBaseType\_t uxIndex )**

**{**

**// Variables in co-routines must be declared static if they must**

**// maintain value across a blocking call. This may not be necessary**

**// for const variables.**

**static const char cLedToFlash[ 2 ] = { 5, 6 };**

**static const TickType\_t uxFlashRates[ 2 ] = { 200, 400 };**

**// Must start every co-routine with a call to crSTART();**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// This co-routine just delays for a fixed period, then toggles**

**// an LED. Two co-routines are created using this function, so**

**// the uxIndex parameter is used to tell the co-routine which**

**// LED to flash and how long to delay. This assumes xQueue has**

**// already been created.**

**vParTestToggleLED( cLedToFlash[ uxIndex ] );**

**crDELAY( xHandle, uxFlashRates[ uxIndex ] );**

**}**

**// Must end every co-routine with a call to crEND();**

**crEND();**

**}**

**// Function that creates two co-routines.**

**void vOtherFunction( void )**

**{**

**unsigned char ucParameterToPass;**

**TaskHandle\_t xHandle;**

**// Create two co-routines at priority 0. The first is given index 0**

**// so (from the code above) toggles LED 5 every 200 ticks. The second**

**// is given index 1 so toggles LED 6 every 400 ticks.**

**for( uxIndex = 0; uxIndex < 2; uxIndex++ )**

**{**

**xCoRoutineCreate( vFlashCoRoutine, 0, uxIndex );**

**}**

**}**

### CoRoutineHandle\_t

引用协同例程的类型。协同例程句柄会自动传递到每个协同例程函数中。

# crDELAY [[Co-Routine Specific](https://www.freertos.org/croutineapi.html)]

croutine.h

**void crDELAY( CoRoutineHandle\_t xHandle,**

**TickType\_t xTicksToDelay )**

crDELAY是一个宏。上面原型中的数据类型仅供参考。

将协同例程延迟固定的时间。

只能从协同例程函数本身调用crDELAY，而不能从协同例程函数调用的函数内部调用。这是因为协同例程不维护自己的堆栈。

**参数：**

|  |  |
| --- | --- |
| *xHandle* | 协同例程的句柄要延迟。这是协同例程函数的xHandle参数。 |
| *xTickToDelay* | 协同例程应延迟的滴答数。等于的实际时间由configTICK\_RATE\_HZ（在FreeRTOSConfig.h中设置）定义。常量portTICK\_PERIOD\_MS可用于将滴答转换为毫秒。 |

**用法示例：**

**// Co-routine to be created.**

**void vACoRoutine( CoRoutineHandle\_t xHandle,**

**UBaseType\_t uxIndex )**

**{**

**// Variables in co-routines must be declared static if they must maintain**

**// value across a blocking call. This may not be necessary for const**

**// variables. We are to delay for 200ms.**

**static const xTickType xDelayTime = 200 / portTICK\_PERIOD\_MS;**

**// Must start every co-routine with a call to crSTART();**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// Delay for 200ms.**

**crDELAY( xHandle, xDelayTime );**

**// Do something here.**

**}**

**// Must end every co-routine with a call to crEND();**

**crEND();**

**}**

# crQUEUE\_SEND [[Co-Routine Specific](https://www.freertos.org/croutineapi.html)]

croutine.h

**crQUEUE\_SEND(**

**CoRoutineHandle\_t xHandle,**

**QueueHandle\_t xQueue,**

**void \*pvItemToQueue,**

**TickType\_t xTicksToWait,**

**BaseType\_t \*pxResult**

**)**

crQUEUE\_SEND是一个宏。数据类型在上面的原型中显示，仅供参考。

宏的crQUEUE\_SEND（）和crQUEUE\_RECEIVE（）是等效于任务使用的xQueueSend（）和xQueueReceive（）函数的例程。

crQUEUE\_SEND和crQUEUE\_RECEIVE只能在协同例程中使用，而xQueueSend（）和xQueueReceive（）只能在任务中使用。 **请注意**，协同例程只能将数据发送到其他协同例程。协同例程不能使用队列将数据发送到任务，反之亦然。

只能从协同例程函数本身中调用crQUEUE\_SEND，而不能从协同例程函数所调用的函数内部调用。这是因为协同例程不维护自己的堆栈。

有关在任务和协同程序之间以及ISR和协同程序之间传递数据的信息，请参见Web文档的协同程序部分。

**参数：**

|  |  |
| --- | --- |
| *xHandle* | 调用协同例程的句柄。这是协同例程函数的xHandle参数。 |
| *xQueue* | 将在其上发布数据的队列的句柄。使用xQueueCreate（）API函数创建队列时，将获得该句柄作为返回值。 |
| *pvItemToQueue* | 指向要发布到队列中的数据的指针。创建队列时，将指定每个排队项目的字节数。该字节数从pvItemToQueue复制到队列本身。 |
| *xTickToDelay* | 如果没有立即可用的空间，则该例程应阻塞以等待队列中可用空间的滴答数。等于的实际时间由configTICK\_RATE\_HZ（在FreeRTOSConfig.h中设置）定义。常量portTICK\_PERIOD\_MS可用于将滴答转换为毫秒（请参见下面的示例）。 |
| *pxResult* | 如果数据成功发布到队列中，则pxResult指向的变量将设置为pdPASS，否则它将设置为ProjDefs.h中定义的错误。 |

**用法示例：**

**// Co-routine function that blocks for a fixed period then posts a number onto**

**// a queue.**

**static void prvCoRoutineFlashTask( CoRoutineHandle\_t xHandle,**

**UBaseType\_t uxIndex )**

**{**

**// Variables in co-routines must be declared static if they must maintain**

**// value across a blocking call.**

**static BaseType\_t xNumberToPost = 0;**

**static BaseType\_t xResult;**

**// Co-routines must begin with a call to crSTART().**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// This assumes the queue has already been created.**

**crQUEUE\_SEND( xHandle,**

**xCoRoutineQueue,**

**&xNumberToPost,**

**NO\_DELAY,**

**&xResult );**

**if( xResult != pdPASS )**

**{**

**// The message was not posted!**

**}**

**// Increment the number to be posted onto the queue.**

**xNumberToPost++;**

**// Delay for 100 ticks.**

**crDELAY( xHandle, 100 );**

**}**

**// Co-routines must end with a call to crEND().**

**crEND();**

**}**

# crQUEUE\_RECEIVE [[Co-Routine Specific](https://www.freertos.org/croutineapi.html)]

croutine.h

**void crQUEUE\_RECEIVE(**

**CoRoutineHandle\_t xHandle,**

**QueueHandle\_t xQueue,**

**void \*pvBuffer,**

**TickType\_t xTicksToWait,**

**BaseType\_t \*pxResult**

**)**

crQUEUE\_RECEIVE是一个宏。数据类型在上面的原型中显示，仅供参考。

宏的crQUEUE\_SEND（）和crQUEUE\_RECEIVE（）是等效于任务使用的xQueueSend（）和xQueueReceive（）函数的例程。

crQUEUE\_SEND和crQUEUE\_RECEIVE只能在协同例程中使用，而xQueueSend（）和xQueueReceive（）只能在任务中使用。 **请注意**，协同例程只能将数据发送到其他协同例程。协同例程不能使用队列将数据发送到任务，反之亦然。

crQUEUE\_RECEIVE只能从协同例程函数本身调用，而不能从协同例程函数调用的函数内部调用。这是因为协同例程不维护自己的堆栈。

有关在任务和协同程序之间以及ISR和协同程序之间传递数据的信息，请参见Web文档的协同程序部分。

**参数：**

|  |  |
| --- | --- |
| *xHandle* | 调用协同例程的句柄。这是协同例程函数的xHandle参数。 |
| *xQueue* | 将从中接收数据的队列的句柄。使用xQueueCreate（）API函数创建队列时，将获得该句柄作为返回值。 |
| *pvBuffer* | 将接收到的项目复制到的缓冲区。创建队列时，将指定每个排队项目的字节数。该字节数被复制到pvBuffer中。 |
| *xTickToDelay* | 如果数据不能立即可用，则该例程应阻塞以等待队列中的数据可用的滴答数。等于的实际时间由configTICK\_RATE\_HZ（在FreeRTOSConfig.h中设置）定义。常量portTICK\_PERIOD\_MS可用于将刻度线转换为毫秒（请参见crQUEUE\_SEND示例）。 |
| *pxResult* | 如果已成功从队列中检索数据，则pxResult指向的变量将设置为pdPASS，否则将设置为ProjDefs.h中定义的错误代码。 |

**用法示例：**

**// A co-routine receives the number of an LED to flash from a queue. It**

**// blocks on the queue until the number is received.**

**static void prvCoRoutineFlashWorkTask( CoRoutineHandle\_t xHandle,**

**UBaseType\_t uxIndex )**

**{**

**// Variables in co-routines must be declared static if they must maintain**

**// value across a blocking call.**

**static BaseType\_t xResult;**

**static UBaseType\_t uxLEDToFlash;**

**// All co-routines must start with a call to crSTART().**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// Wait for data to become available on the queue.**

**crQUEUE\_RECEIVE( xHandle,**

**xCoRoutineQueue,**

**&uxLEDToFlash,**

**portMAX\_DELAY,**

**&xResult );**

**if( xResult == pdPASS )**

**{**

**// We received the LED to flash - flash it!**

**vParTestToggleLED( uxLEDToFlash );**

**}**

**}**

**crEND();**

**}**

# crQUEUE\_SEND\_FROM\_ISR [[Co-Routine Specific](https://www.freertos.org/croutineapi.html)]

croutine.h

**BaseType\_t crQUEUE\_SEND\_FROM\_ISR**

**(**

**QueueHandle\_t xQueue,**

**void \*pvItemToQueue,**

**BaseType\_t xCoRoutinePreviouslyWoken**

**)**

crQUEUE\_SEND\_FROM\_ISR（）是一个宏。数据类型在上面的原型中显示，仅供参考。

宏的crQUEUE\_SEND\_FROM\_ISR（）和crQUEUE\_RECEIVE\_FROM\_ISR（）是等效于任务使用的xQueueSendFromISR（）和xQueueReceiveFromISR（）函数的例程。

crQUEUE\_SEND\_FROM\_ISR（）和crQUEUE\_RECEIVE\_FROM\_ISR（）仅可用于在协同例程和ISR之间传递数据，而xQueueSendFromISR（）和xQueueReceiveFromISR（）仅可用于在任务与ISR之间传递数据。

只能从ISR调用crQUEUE\_SEND\_FROM\_ISR，以将数据发送到协同例程中正在使用的队列。

有关在任务和协同程序之间以及ISR和协同程序之间传递数据的信息，请参见Web文档的协同程序部分。

**参数：**

|  |  |
| --- | --- |
| *xQueue* | 要在其上发布项目的队列的句柄。 |
| *pvItemToQueue* | 指向要放在队列中的项目的指针。创建队列时已定义了队列将要容纳的项目的大小，因此，这许多字节将从pvItemToQueue复制到队列存储区域。 |
| *xCoRoutinePreviously唤醒* | 包括在内，因此ISR可以从单个中断中多次发布到同一队列中。第一次调用应始终传递pdFALSE。后续调用应传入从先前调用返回的值。 |

**返回值：**

如果通过发布到队列中唤醒了一个例程，则为pdTRUE。ISR使用它来确定在ISR之后是否可能需要上下文切换。

**用法示例：**

**// A co-routine that blocks on a queue waiting for characters to be received.**

**static void vReceivingCoRoutine( CoRoutineHandle\_t xHandle,**

**UBaseType\_t uxIndex )**

**{**

**char cRxedChar;**

**BaseType\_t xResult;**

**// All co-routines must start with a call to crSTART().**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// Wait for data to become available on the queue. This assumes the**

**// queue xCommsRxQueue has already been created!**

**crQUEUE\_RECEIVE( xHandle,**

**xCommsRxQueue,**

**&uxLEDToFlash,**

**portMAX\_DELAY,**

**&xResult );**

**// Was a character received?**

**if( xResult == pdPASS )**

**{**

**// Process the character here.**

**}**

**}**

**// All co-routines must end with a call to crEND().**

**crEND();**

**}**

**// An ISR that uses a queue to send characters received on a serial port to**

**// a co-routine.**

**void vUART\_ISR( void )**

**{**

**char cRxedChar;**

**BaseType\_t xCRWokenByPost = pdFALSE;**

**// We loop around reading characters until there are none left in the UART.**

**while( UART\_RX\_REG\_NOT\_EMPTY() )**

**{**

**// Obtain the character from the UART.**

**cRxedChar = UART\_RX\_REG;**

**// Post the character onto a queue. xCRWokenByPost will be pdFALSE**

**// the first time around the loop. If the post causes a co-routine**

**// to be woken (unblocked) then xCRWokenByPost will be set to pdTRUE.**

**// In this manner we can ensure that if more than one co-routine is**

**// blocked on the queue only one is woken by this ISR no matter how**

**// many characters are posted to the queue.**

**xCRWokenByPost = crQUEUE\_SEND\_FROM\_ISR( xCommsRxQueue,**

**&cRxedChar,**

**xCRWokenByPost );**

**}**

**}**

# crQUEUE\_RECEIVE\_FROM\_ISR [[Co-Routine Specific](https://www.freertos.org/croutineapi.html)]

croutine.h

**BaseType\_t crQUEUE\_SEND\_FROM\_ISR**

**(**

**QueueHandle\_t xQueue,**

**void \*pvBuffer,**

**BaseType\_t \* pxCoRoutineWoken**

**)**

宏的crQUEUE\_SEND\_FROM\_ISR（）和crQUEUE\_RECEIVE\_FROM\_ISR（）是等效于任务使用的xQueueSendFromISR（）和xQueueReceiveFromISR（）函数的例程。

crQUEUE\_SEND\_FROM\_ISR（）和crQUEUE\_RECEIVE\_FROM\_ISR（）仅可用于在协同例程和ISR之间传递数据，而xQueueSendFromISR（）和xQueueReceiveFromISR（）仅可用于在任务与ISR之间传递数据。

只能从ISR调用crQUEUE\_RECEIVE\_FROM\_ISR来从正在从协同例程（发布到该队列的协同例程）中使用的队列接收数据。

有关在任务和协同程序之间以及ISR和协同程序之间传递数据的信息，请参见Web文档的协同程序部分。

**参数：**

|  |  |
| --- | --- |
| *xQueue* | 要在其上发布项目的队列的句柄。 |
| *pvBuffer* | 指向将放置接收到的项目的缓冲区的指针。创建队列时已定义了队列将容纳的项目的大小，因此将从队列中将这许多字节复制到pvBuffer中。 |
| *pxCoRoutineWoken* | 协同例程可能会被阻塞，以等待队列中的可用空间。如果crQUEUE\_RECEIVE\_FROM\_ISR导致此类协例解除阻塞，则\* pxCoRoutineWoken将设置为pdTRUE，否则\* pxCoRoutineWoken将保持不变 |

**返回值：**

pdTRUE从队列中成功接收到一个项目，否则为pdFALSE。

**用法示例：**

**// A co-routine that posts a character to a queue then blocks for a fixed**

**// period. The character is incremented each time.**

**static void vSendingCoRoutine( CoRoutineHandle\_t xHandle,**

**UBaseType\_t uxIndex )**

**{**

**// cChar holds its value while this co-routine is blocked and must therefore**

**// be declared static.**

**static char cCharToTx = 'a';**

**BaseType\_t xResult;**

**// All co-routines must start with a call to crSTART().**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// Send the next character to the queue.**

**crQUEUE\_SEND( xHandle,**

**xCoRoutineQueue,**

**&cCharToTx,**

**NO\_DELAY,**

**&xResult );**

**if( xResult == pdPASS )**

**{**

**// The character was successfully posted to the queue.**

**}**

**else**

**{**

**// Could not post the character to the queue.**

**}**

**// Enable the UART Tx interrupt to cause an interrupt in this**

**// hypothetical UART. The interrupt will obtain the character**

**// from the queue and send it.**

**ENABLE\_RX\_INTERRUPT();**

**// Increment to the next character then block for a fixed period.**

**// cCharToTx will maintain its value across the delay as it is**

**// declared static.**

**cCharToTx++;**

**if( cCharToTx > 'x' )**

**{**

**cCharToTx = 'a';**

**}**

**crDELAY( 100 );**

**}**

**// All co-routines must end with a call to crEND().**

**crEND();**

**}**

**// An ISR that uses a queue to receive characters to send on a UART.**

**void vUART\_ISR( void )**

**{**

**char cCharToTx;**

**BaseType\_t xCRWokenByPost = pdFALSE;**

**while( UART\_TX\_REG\_EMPTY() )**

**{**

**// Are there any characters in the queue waiting to be sent?**

**// xCRWokenByPost will automatically be set to pdTRUE if a co-routine**

**// is woken by the post - ensuring that only a single co-routine is**

**// woken no matter how many times we go around this loop.**

**if( crQUEUE\_RECEIVE\_FROM\_ISR( xQueue, &cCharToTx, &xCRWokenByPost ) )**

**{**

**SEND\_CHARACTER( cCharToTx );**

**}**

**}**

**}**

# vCoRoutineSchedule [[Co-Routine Specific](https://www.freertos.org/croutineapi.html)]

croutine.h

**void vCoRoutineSchedule( void );**

运行协同程序。

vCoRoutineSchedule（）执行能够运行的最高优先级的协同例程。协同例程将一直执行，直到阻塞，屈服或被任务抢占为止。协同例程可以协同执行，因此一个协同例程不能被另一个协同例程抢占，而可以被一个任务抢占。

如果应用程序同时包含任务和例程，则应从空闲任务（在空闲任务挂钩中）调用vCoRoutineSchedule。

**用法示例：**

**void vApplicationIdleHook( void )**

**{**

**vCoRoutineSchedule( void );**

**}**

Alternatively, if the idle task is not performing any other function it would be more efficient to call vCoRoutineSchedule() from within a loop as:

**void vApplicationIdleHook( void )**

**{**

**for( ;; )**

**{**

**vCoRoutineSchedule( void );**

**}**

**}**