FreeRTOS_Reference

> 2021年11月12日
>
> FreeRTOS参考手册中文版本，根据官网v9.0.0英文原版译制，仅供参考

章节1　关于本手册

本文档提供了主要的FreeRTOS API1和FreeRTOS内核配置选项的技术参考。本文假设读者已经熟悉编写多任务应用程序的概念，以及实时内核提供的原语。对于不熟悉这些基本概念的读者，建议阅读《使用FreeRTOS实时内核—实用指南》这本书，它有更多描述性、实践性和教程风格的文本。这本书可以从 http://www.FreeRTOS.org/Documentation获取。

函数在本手册中出现的顺序

在本文档中，API函数被分为5组——与任务和调度程序相关的函数、与队列相关的函数、与信号量相关的函数、与软件计时器相关的函数和与事件组相关的函数。每一组都在其各自的章节中进行了记录，并且在每个章节中，API函数都按字母顺序列出。但是请注意，每个API函数的名称前面都有一个或多个字母，这些字母指定了函数的返回类型，并且每个章节中API函数的字母顺序忽略了函数的返回类型前缀。附录1:更详细地描述了前缀。

例如，考虑用于创建FreeRTOS任务的API函数。它的名称是xTaskCreate()。' x '前缀指定xTaskCreate()返回一个非标准类型。次要的"任务"前缀指定该函数是一个与任务相关的函数，因此，将在包含任务和调度程序相关函数的章节中进行记录。' x '不是按字母顺序考虑的，所以xTaskCreate()将出现在任务和调度器章节中，就像它的名字只是TaskCreate()一样。

API使用限制

以下规则适用于使用FreeRTOS API:

1. 不以"FromISR"结尾的API函数不能用于中断服务例程(ISR)。一些FreeRTOS港口进一步限制,即使API函数,最终"FromISR"不能用于一个中断服务例程(硬件)的优先级高于设定的优先级configMAX_SYSCALL_INTERRUPT_PRIORITY(或configMAX_API_CALL_INTERRUPT_PRIORITY,根据端口)内核配置不变,在本文档的7.1节中有描述。第二个限制是确保优先级高于configMAX_SYSCALL_INTERRUPT_PRIORITY设置的中断的时间、确定性和延迟不受FreeRTOS的影响。
2. 在挂起调度器时，绝不能调用可能导致上下文切换的API函数。
3. 可能导致上下文切换的API函数不能在临界区域内调用。

章节2　任务和调度器

2.1 portSWITCH_TO_USER_MODE()

```
#include “FreeRTOS.h”
#include “task.h”

void portSWITCH_TO_USER_MODE( void );
```

概要

本功能仅供高级用户使用，仅与FreeRTOS MPU端口(使用内存保护单元的FreeRTOS端口)相关。

提供给xTaskCreateRestricted()的参数指定要创建的任务应该是User(非特权)模式任务还是Supervisor(特权)模式任务。Supervisor模式任务可以调用portSWITCH_TO_USER_MODE()将自己从Supervisor模式任务转换为User模式任务。

参数

无

返回值

无

提示

不存在与portSWITCH_TO_USER_MODE()等价的对等项，允许任务将自己从用户模式转换为Supervisor模式任务。

2.2 vTaskAllocateMPURegions()

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskAllocateMPURegions( TaskHandle_t xTaskToModify,
                              const MemoryRegion_t * const xRegions );
```

概要

定义一组内存保护单元(MPU)区域，供MPU受限任务使用。

本功能仅供高级用户使用，仅与FreeRTOS MPU端口(使用内存保护单元的FreeRTOS端口)相关。

当使用xTaskCreateRestricted()函数创建任务时，可以将MPU控制的内存区域分配给MPU受限任务。然后可以在运行时使用vTaskAllocateMPURegions()函数重新定义(或重新分配)它们。

参数

xTaskToModify

- 被修改的受限任务的句柄(被赋予访问由xRegions参数定义的内存区域的任务)。
- 任务的句柄是通过xTaskCreateRestricted() API函数的pxCreatedTask参数获得的。
- 任务可以通过传递NULL来代替有效的任务句柄来修改自己的内存区域访问定义。

xRegions

- MemoryRegion_t结构的数组。数组中的位置数由端口特定的portNUM_CONFIGURABLE_REGIONS常量定义。在Cortex-M3上，portNUM_CONFIGURABLE_REGIONS被定义为3。
- 数组中的每个MemoryRegion_t结构都定义了一个单独的MPU内存区域，供xTaskToModify参数引用的任务使用。

返回值

无

提示

MPU内存区域是使用所示的MemoryRegion_t结构定义的。

```
typedef struct xMEMORY_REGION
{
    void *pvBaseAddress;
    unsigned long ulLengthInBytes;
    unsigned long ulParameters;
} MemoryRegion_t;
```

2.3 xTaskAbortDelay()

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskAbortDelay( TaskHandle_t xTask );
```

概要

调用包含超时参数的API函数可能导致调用任务进入Blocked状态。处于Blocked状态的任务要么等待超时时间，要么等待超时事件发生，事件发生后任务将自动离开Blocked状态，进入Ready状态。这种行为有很多例子，其中两个是:

- 如果任务调用vTaskDelay()，它将进入Blocked状态，直到函数参数指定的超时时间过去，此时任务将自动离开Blocked状态并进入Ready状态。
- 如果一个任务调用ulTaskNotifyTake()时通知值为零将进入阻塞状态,直到它收到一个通知或一个函数的参数所指定的超时运行,在这段时间的任务将自动离开阻塞状态,进入就绪状态。

xTaskAbortDelay()将任务从Blocked状态移动到Ready状态，即使任务正在等待的事件没有发生，并且任务进入Blocked状态时指定的超时时间还没有过去。

当任务处于Blocked状态时，它对调度程序不可用，并且不会占用任何处理时间。

参数

xTask

- 将从Blocked状态移出的任务的句柄。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

返回值

如果xTask引用的任务从Blocked状态中移除，则返回pdPASS。如果xTask引用的任务没有从Blocked状态中删除，因为它不是处于Blocked状态，则返回pdFAIL。

提示

为了xTaskAbortDelay()可用，必须在FreeRTOSConfig.h中将INCLUDE_xTaskAbortDelay设置为1。

示例

```
void vAFunction( TaskHandle_t xTask )
{
/* The task referenced by xTask is blocked to wait for something that the task
calling
this function has determined will never happen. Force the task referenced by xTask
out of the Blocked state. */
    if( xTaskAbortDelay( xTask ) == pdFAIL )
    {
        /* The task referenced by xTask was not in the Blocked state anyway. */
    }
    else
    {
        /* The task referenced by xTask was in the Blocked state, but is not now. */
    }
}
```

2.4 xTaskCallApplicationTaskHook()

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskCallApplicationTaskHook( TaskHandle_t xTask, void *pvParameters );
```

概要

本功能仅供高级用户使用。

标记值的含义和使用由应用程序编写人员定义。内核本身通常不会访问标记值。

作为一种特殊情况，标签值可以用来关联一个任务挂钩(或回调)函数到一个任务。当此操作完成时，钩子函数将使用xTaskCallApplicationTaskHook()调用。

任务钩子函数可以用于任何目的。本节中展示的示例演示了一个用于输出调试跟踪信息的任务钩子。

任务钩子函数必须有原型如下：

```
BaseType_t xAnExampleTaskHookFunction( void *pvParameters );
```

xTaskCallApplicationTaskHook()仅当configUSE_APPLICATION_TASK_TAG在FreeRTOSConfig.h中设置为1时可用。

参数

xTask

- The handle of the task whose associated hook function is being called.
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用中使用任务的名称xTaskGetHandle().
- 任务可以通过传递NULL来代替有效的任务句柄来调用自己的钩子函数。

pvParameters

- 作为任务钩子函数本身的形参的值。
- 该形参的类型为“void指针”，允许任务钩子函数形参有效地、间接地通过强制类型转换接收任何类型的形参。例如，整数类型可以通过在调用钩子函数时将整型转换为void指针，然后在钩子函数内部将void指针形参转换回整型来传递给钩子函数。

示例

```
/* Define a hook (callback) function – using the required prototype as
demonstrated by Listing 8 */
static BaseType_t prvExampleTaskHook( void * pvParameter )
{
/* Perform an action - this could be anything. In this example the hook
is used to output debug trace information. pxCurrentTCB is the handle
of the currently executing task. (vWriteTrace() is not an API function,
its just used as an example.) */
vWriteTrace( pxCurrentTCB );
/* This example does not make use of the hook return value so just returns
0 in every case. */
return 0;
}
/* Define an example task that makes use of its tag value. */
void vAnotherTask( void *pvParameters )
{
```

```
/* vTaskSetApplicationTaskTag() sets the ‘tag’ value associated with a task.
NULL is used in place of a valid task handle to indicate that it should be
the tag value of the calling task that gets set. In this example the ‘value’
being set is the hook function. */
vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );
for( ;; )
{
/* The rest of the task code goes here. */
}
}
/* Define the traceTASK_SWITCHED_OUT() macro to call the hook function of each
task that is switched out. pxCurrentTCB points to the handle of the currently
running task. */
#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )
```

2.5 xTaskCheckForTimeOut()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskCheckForTimeOut( TimeOut_t *
const pxTimeOut,                                TickType_t * const pxTicksToWait );
```

概要

本功能仅供高级用户使用。

任务可以进入“Blocked”状态，等待事件发生。通常，任务不会在Blocked状态下无限期等待，而是指定一个超时时间。如果在任务等待的事件发生之前超时，则任务将从阻塞状态中移除。

如果一个任务进入和退出阻塞状态时不止一次等待事件发生,那么每一次使用的超时任务进入阻塞状态必须进行调整,以确保所有阻塞状态的时间不超过原指定的超时时间。xTaskCheckForTimeOut()执行调整，考虑到偶尔发生的情况，如tick计数溢出，否则手动调整容易出错。

xTaskCheckForTimeOut()与vTaskSetTimeOutState()一起使用。调用vTaskSetTimeOutState()来设置初始条件，之后可以调用xTaskCheckForTimeOut()来检查超时条件，如果没有超时则调整剩余的块时间。

参数

pxTimeOut

- 一种指向结构的指针，该结构包含确定是否发生超时所必需的信息。pxTimeOut使用vTaskSetTimeOutState()初始化。

pxTicksToWait

- 用于传递调整后的块时间，这是在考虑了已花费在Blocked状态中的时间后剩余的块时间。

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，则还剩下一些块时间，因此没有发生超时。

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 Blockedstate. Adjusting xTicksToWait to account for any time that has been spent in theBlocked state within this function so far to ensure the total amount of time spentin 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 receiveinterrupt has placed more data into the buffer. */   ulTaskNotifyTake( pdTRUE, xTicksToWait );    }    /* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The actualnumber of bytes read (which might be less than uxWantedBytes) is returned. */    uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, uxWantedBytes    ); return uxReceived;}
```

2.6 xTaskCreate()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,                        const char * const pcName,                        unsigned short usStackDepth,                        void *pvParameters,                        UBaseType_t uxPriority,                        TaskHandle_t *pxCreatedTask );
```

概要

创建任务的新实例。

每个任务都需要用于保存任务状态(任务控制块，或TCB)的RAM，并被任务用作其堆栈。如果使用xTaskCreate()创建任务，则从FreeRTOS堆中自动分配所需的RAM。如果使用xTaskCreateStatic()创建任务，那么应用程序编写器将提供RAM，这将导致两个额外的函数参数，但允许在编译时静态分配RAM。

新创建的任务最初处于就绪状态，但如果没有能够运行的高优先级任务，则会立即变为运行状态任务。

可以在启动调度程序之前和之后创建任务。

参数

pvTaskCode

- pvTaskCode参数只是一个指向实现任务的函数(实际上只是函数名)的指针。

pcName

- 这主要用于方便调试，但也可以在调用xTaskGetHandle()中使用，以获得任务句柄。
- 应用程序定义的常量configMAX_TASK_NAME_LEN定义了名称的最大字符长度——包括NULL结束符。提供一个超过这个最大值的字符串将导致该字符串被截断。

usStackDepth

- usStackDepth值告诉内核堆栈的大小。
- 例如，在一个4字节堆栈宽度的架构上，如果usStackDepth以100传入，那么将分配400字节的堆栈空间(100 * 4字节)。堆栈深度乘以堆栈宽度不能超过size_t类型的变量所能包含的最大值。

  > 译者注：传入参数的值是字数，而非字节数，实际字节数和处理器位宽相关。

- 在演示应用程序中，为所选微控制器架构提供的赋值为该架构上的任何任务的最小推荐值。如果您的任务使用了大量的堆栈空间，那么您必须分配一个更大的值。

pvParameters

- 任务函数接受void指针类型的形参(void*)。赋给pvParameters的值将是传递给任务的值。
- 该参数的类型为“void指针”，允许任务参数有效地、间接地通过强制类型转换接收任何类型的参数。例如，可以通过在任务创建点将整型转换为空指针，然后在任务函数定义本身将空指针形参转换回整型，将整型类型传递给任务函数。

uxPriority

- 定义任务将执行的优先级。优先级可以从0(最低优先级)分配到(configMAX_PRIORITIES - 1)(最高优先级)。

  > 译者注：FreeRTOS任务最高优先级是0，其他操作系统任务最低优先级是0，这点需要特别注意。

- configMAX_PRIORITIES是一个用户定义的常量。如果configUSE_PORT_OPTIMISED_TASK_SELECTION设置为0,那么优先的数量没有上限,可以使用的限制(除了数据类型和可用的RAM单片机使用),但建议使用所需的最低数量的优先级,以避免浪费内存。

- 传递以上的uxPriority值(configMAX_PRIORITIES - 1)将导致分配给任务的优先级被静默地上限为最大合法值

pxCreatedTask

- pxCreatedTask可用于将句柄传递给正在创建的任务。这个句柄可用于在API调用中引用任务，例如，更改任务优先级或删除任务。
- 如果您的应用程序不使用任务句柄，那么pxCreatedTask可以设置为NULL。

返回值

pdPASS 表示任务创建成功。

errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY指示无法创建任务，因为没有足够的堆内存供FreeRTOS分配任务数据结构和堆栈。

如果heap_1.c, heap_2.c或heap_4.c被包含在项目中，那么堆可用的总数量是由FreeRTOSConfig.h中的configTOTAL_HEAP_SIZE定义的，并且分配内存失败可以使用vApplicationMallocFailedHook()回调(或'hook ')函数捕获，剩余的空闲堆内存数量可以使用xPortGetFreeHeapSize() API函数查询。

如果项目中包含heap_3.c，那么总堆大小是由连接器配置定义的。

提示

configSUPPORT_DYNAMIC_ALLOCATION必须在FreeRTOSConfig.h中设置为1，或者不设置为未定义，以使该函数可用。

2.7 xTaskCreateStatic()

```
#include “FreeRTOS.h”#include “task.h”TaskHandle_t xTaskCreateStatic( TaskFunction_t
pvTaskCode,                                const char * const pcName,
                 uint32_t ulStackDepth,                                 void
*pvParameters,                               UBaseType_t uxPriority,
                 StackType_t * const puxStackBuffer,
StaticTask_t * const pxTaskBuffer );
```

概要

创建任务的新实例.

每个任务都需要用于保存任务状态(任务控制块，或TCB)的RAM，并被任务用作其堆栈。如果使用xTaskCreate()创建任务，则从FreeRTOS堆中自动分配所需的RAM。如果使用xTaskCreateStatic()创建任务，那么应用程序编写器将提供RAM，这将导致两个额外的函数参数，但允许在编译时静态分配RAM。

新创建的任务最初处于就绪状态，但如果没有能够运行的高优先级任务，则会立即变为运行状态任务。

可以在启动调度程序之前和之后创建任务。

参数

pvTaskCode

- pvTaskCode参数只是一个指向实现任务的函数(实际上只是函数名)的指针。

pcName

- 这主要用于方便调试，但也可以在调用xTaskGetHandle()中使用，以获得任务句柄。
- 应用程序定义的常量configMAX_TASK_NAME_LEN定义了名称的最大字符长度——包括NULL结束符。提供一个超过这个最大值的字符串将导致该字符串被截断。

usStackDepth

- usStackDepth值告诉内核堆栈的大小。
- 例如，在一个4字节堆栈宽度的架构上，如果usStackDepth以100传入，那么将分配400字节的堆栈空间(100 * 4字节)。堆栈深度乘以堆栈宽度不能超过size_t类型的变量所能包含的最大值。

  > 译者注：传入参数的值是字数，而非字节数，实际字节数和处理器位宽相关。

- 在演示应用程序中，为所选微控制器架构提供的赋值为该架构上的任何任务的最小推荐值。如果您的任务使用了大量的堆栈空间，那么您必须分配一个更大的值。

pvParameters

- 任务函数接受void指针类型的形参(void*)。赋给pvParameters的值将是传递给任务的值。
- 该参数的类型为“void指针”，允许任务参数有效地、间接地通过强制类型转换接收任何类型的参数。例如，可以通过在任务创建点将整型转换为空指针，然后在任务函数定义本身将空指针形参转换回整型，将整型类型传递给任务函数。

uxPriority

- 定义任务将执行的优先级。优先级可以从0(最低优先级)分配到(configMAX_PRIORITIES - 1)(最高优先级)。

  > 译者注：FreeRTOS任务最高优先级是0，其他操作系统任务最低优先级是0，这点需要特别注意。

- configMAX_PRIORITIES是一个用户定义的常量。如果configUSE_PORT_OPTIMISED_TASK_SELECTION设置为0,那么优先的数量没有上限,可以使用的限制(除了数据类型和可用的RAM单片机使用),但建议使用所需的最低数量的优先级,以避免浪费内存。
- 传递以上的uxPriority值(configMAX_PRIORITIES - 1)将导致分配给任务的优先级被静默地上限为最大合法值

puxStackBuffer

- 必须指向至少具有ulStackDepth索引的StackType_t变量数组(参见上面的ulStackDepth参数)。该数组将被用作创建的任务的堆栈，因此必须是持久的(不能在函数创建的堆栈框架中声明，也不能在应用程序执行时可以合法重写的任何其他内存中声明)。

pxTaskBuffer

- 必须指向StaticTask_t类型的变量。该变量将用于保存创建的任务的数据结构(TCB)，因此它必须是持久的(不能在函数创建的堆栈框架中声明，也不能在应用程序执行时可以合法重写的任何其他内存中声明)。

返回值

NULL 无法创建任务，因为puxStackBuffer或pxTaskBuffer为NULL。

其他返回值 如果返回非null值，则创建了任务，返回的值是创建的任务的句柄。

提示

configSUPPORT_STATIC_ALLOCATION必须在FreeRTOSConfig.h中设置为1，才能使用该函数。

2.8 xTaskCreateRestricted()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskCreateRestricted(
TaskParameters_t *pxTaskDefinition,                                    TaskHandle_t
*pxCreatedTask );
```

概要

本功能仅供高级用户使用，仅与FreeRTOS MPU端口(使用内存保护单元的FreeRTOS端口)相关。

创建新的MPU受限任务。

新创建的任务最初处于就绪状态，但如果没有能够运行的高优先级任务，则会立即变为运行状态任务。

可以在启动调度程序之前和之后创建任务。

参数

pxTaskDefinition

- 指向定义任务的结构的指针。该结构在参考手册的这一节的注释标题下进行了描述。

pxCreatedTask

- pxCreatedTask可用于将句柄传递给正在创建的任务。这个句柄可用于在API调用中引用任务，例如，更改任务优先级或删除任务。
- 如果您的应用程序不使用任务句柄，那么pxCreatedTask可以设置为NULL。

返回值

pdPASS 表示任务创建成功。

其他返回值 指示不能按指定的方式创建任务，可能是因为可用的FreeRTOS堆内存不足，无法分配任务数据结构。

提示

xTaskCreateRestricted()利用了下面所示的两个数据结构。

```
typedef struct xTASK_PARAMTERS{    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;
```

2.9 TaskDelay()

```
#include “FreeRTOS.h”#include “task.h”void vTaskDelay( TickType_t xTicksToDelay );
```

概要

对于固定数量的tick中断，将调用vTaskDelay()的任务置于Blocked状态。

指定零节拍的延迟时间不会导致调用任务被置于Blocked状态，但会导致调用任务屈服于与之共享其优先级的任何Ready状态任务。调用vTaskDelay(0)等价于调用taskYIELD()。

参数

xTicksToDelay

- 调用任务在转换回就绪状态之前将保持在Blocked状态的tick中断的数量。例如，如果一个名为vTaskDelay(100)的任务在tick计数为10,000时，那么它将立即进入Blocked状态，并保持Blocked状态，直到tick计数达到10,100。
- 在调用vTaskDelay()和发生下一个滴答中断之间的任何时间，都算作一个完整的滴答周期。因此，在最坏的情况下，指定延迟周期时所能达到的最高时间分辨率等于一个完整的tick中断周期
- 宏pdMS_TO_TICKS()可用于将毫秒转换为节拍。本节的示例演示了这一点。

返回值

无

提示

必须在FreeRTOSConfig.h中将INCLUDE_vTaskDelay设置为1，以便vTaskDelay() API函数可用。

示例

```
void vAnotherTask( void * pvParameters ){    for( ;; )    {    /* Perform some processing here. */    …    /* Enter the Blocked state for 20 tick interrupts – the actual time spent    in the Blocked state is dependent on the tick frequency. */    vTaskDelay( 20 );    /* 20 ticks will have passed since the first call to vTaskDelay() was    executed. */    /* Enter the Blocked state for 20 milliseconds. Using the    pdMS_TO_TICKS() macro means the tick frequency can change without effecting the time spent in the blocked state (other than due to the    resolution of the tick frequency). */        vTaskDelay( pdMS_TO_TICKS( 20 ) );    }}
```

2.10 vTaskDelayUntil()

```
#include “FreeRTOS.h”#include “task.h”void vTaskDelayUntil( TickType_t *pxPreviousWakeTime, TickType_t xTimeIncrement );
```

概要

将调用vTaskDelayUntil()的任务置于Blocked状态，直到达到绝对时间。

周期性任务可以使用vTaskDelayUntil()来实现固定的执行频率。

vTaskDelay()和vTaskDelayUntil()的区别

- vTaskDelay()导致调用任务进入Blocked状态，然后在vTaskDelay()被调用后的指定时间间隔内保持Blocked状态。调用vTaskDelay()的任务退出Blocked状态的时间与调用vTaskDelay()的时间有关。
- vTaskDelayUntil()导致调用任务进入Blocked状态，然后保持Blocked状态，直到达到绝对时间。调用vTaskDelayUntil()的任务恰好在指定的时间退出Blocked状态，而不是相对于vTaskDelayUntil()被调用的时间。

参数

pxPreviousWakeTime

- 这个参数的命名是基于这样的假设:vTaskDelayUntil()用于实现一个定期执行并以固定频率执行的任务。在这种情况下，pxPreviousWakeTime保持任务最后离开Blocked状态(被“唤醒”)的时间。此时间用作参考点，以计算任务下一次离开Blocked状态的时间。
- pxPreviousWakeTime所指向的变量会在vTaskDelayUntil()函数中自动更新;应用程序代码通常不会修改它，除非变量第一次初始化。本节中的示例演示如何执行初始化。

xTimeIncrement

- 这个参数的命名也是基于这样一个假设:vTaskDelayUntil()被用于实现一个周期性执行的任务，并且具有固定的频率—频率由xTimeIncrement值设置。
- xTimeIncrement在' ticks '中指定。可以使用pdMS_TO_TICKS()宏将毫秒转换为节拍。

返回值

无

提示

INCLUDE_vTaskDelayUntil必须在FreeRTOSConfig.h中设置为1，以便vTaskDelay() API函数可用。

2.11 vTaskDelete()

```
#include “FreeRTOS.h”#include “task.h”void vTaskDelete( TaskHandle_t pxTask );
```

概要

删除以前通过调用xTaskCreate()或xTaskCreateStatic()创建的任务实例。

删除的任务不再存在，不能进入运行状态。

不要尝试使用任务句柄来引用已删除的任务。

当一个任务被删除时，空闲任务负责释放用来保存被删除任务的堆栈和数据结构(任务控制块)的内存。因此，如果应用程序使用vTaskDelete() API函数，应用程序还必须确保空闲任务不会缺乏处理时间(空闲任务必须在Running状态下分配时间)。

删除任务时，只有内核本身分配给任务的内存会自动释放。应用程序(而不是内核)分配给任务的内存或任何其他资源，必须在删除任务时由应用程序显式释放。

参数

pxTask

- 被删除任务的句柄(主题任务)。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。
- 任务可以通过传递NULL来代替有效的任务句柄来删除自己。

返回值

无

示例

```
void vAnotherFunction( void ){    TaskHandle_t xHandle;/* Create a task, storing the handle to the created task in xHandle. */    if(        xTaskCreate(
vTaskCode,            "Demo task",            STACK_SIZE,            NULL,
PRIORITY,            &xHandle             /* The address of xHandle is passed in as the                last parameter to xTaskCreate() to obtain a handle
to the task being created. */        )        != pdPASS )        {            /* The task could not be created because there was not enough FreeRTOS heap
memory available for the task data structures and stack to be allocated. */        }
       else        {            /* Delete the task just created. Use the handle passed out of xTaskCreate()            to reference the subject task. */
vTaskDelete( xHandle );         }/* Delete the task that called this function by passing NULL in as thevTaskDelete() parameter. The same task (this task) could also be deleted bypassing in a valid handle to itself. */}
```

2.12 taskDISABLE_INTERRUPTS()

```
#include “FreeRTOS.h”#include “task.h”void taskDISABLE_INTERRUPTS( void );
```

概要

如果正在使用的FreeRTOS端口没有使用configMAX_SYSCALL_INTERRUPT_PRIORITY(或configMAX_API_CALL_INTERRUPT_PRIORITY，取决于端口)内核配置常量，那么调用taskDISABLE_INTERRUPTS()将使中断全局禁用。

如果正在使用的FreeRTOS端口确实使用了configMAX_SYSCALL_INTERRUPT_PRIORITY内核配置常量，那么调用taskDISABLE_INTERRUPTS()将使中断处于或低于configMAX_SYSCALL_INTERRUPT_PRIORITY设置的中断优先级禁用，并启用所有高优先级的中断。

configMAX_SYSCALL_INTERRUPT_PRIORITY通常在FreeRTOSConfig.h中定义。

taskDISABLE_INTERRUPTS()和taskENABLE_INTERRUPTS()的调用不是为了嵌套而设计的。例如，如果taskDISABLE_INTERRUPTS()被调用两次，对taskENABLE_INTERRUPTS()的单个调用仍然会导致中断被启用。如果需要嵌套，则分别使用taskENTER_CRITICAL()和taskEXIT_CRITICAL()来代替taskDISABLE_INTERRUPTS()和taskENABLE_INTERRUPTS()。

一些FreeRTOS API函数使用临界区，如果临界区嵌套计数为0，它将重新启用中断——即使在调用API函数之前调用taskDISABLE_INTERRUPTS()禁用了中断。当中断已经被禁用时，不建议调用FreeRTOS API函数。

参数

无

返回值

无

2.13 taskENABLE_INTERRUPTS()

```
#include “FreeRTOS.h”#include “task.h”void taskENABLE_INTERRUPTS( void );
```

概要

调用taskENABLE_INTERRUPTS()将导致所有中断优先级被启用。

taskDISABLE_INTERRUPTS()和taskENABLE_INTERRUPTS()的调用不是为了嵌套而设计的。例如，如果taskDISABLE_INTERRUPTS()被调用两次，对taskENABLE_INTERRUPTS()的单个调用仍然会导致中断被启用。如果需要嵌套，则分别使用taskENTER_CRITICAL()和taskEXIT_CRITICAL()来代替taskDISABLE_INTERRUPTS()和taskENABLE_INTERRUPTS()。

一些FreeRTOS API函数使用临界区，如果临界区嵌套计数为0，它将重新启用中断——即使在调用API函数之前调用taskDISABLE_INTERRUPTS()禁用了中断。当中断已经被禁用时，不建议调用FreeRTOS API函数。

参数

无

返回值

无

2.14 taskENTER_CRITICAL()

```
#include “FreeRTOS.h”#include “task.h”void taskENTER_CRITICAL( void );
```

摘要

通过调用taskENTER_CRITICAL()进入临界区，然后通过调用taskEXIT_CRITICAL()退出临界区。

taskENTER_CRITICAL()不能从中断服务程序中调用。请参阅taskENTER_CRITICAL_FROM_ISR()来获得一个中断安全的等效值。

taskENTER_CRITICAL()和taskEXIT_CRITICAL()宏提供了一个基本的临界区实现，它通过禁用中断来工作，可以是全局的，也可以是特定的中断优先级。有关创建临界区而不禁用中断的信息，请参阅vTaskSuspendAll() API函数。

如果正在使用的FreeRTOS端口没有使用configMAX_SYSCALL_INTERRUPT_PRIORITY内核配置常量，那么调用taskENTER_CRITICAL()将使全局中断处于禁用状态。

如果FreeRTOS端口正在使用利用configMAX_SYSCALL_INTERRUPT_PRIORITY(或configMAX_API_CALL_INTERRUPT_PRIORITY,根据端口)内核配置不变,然后调用taskENTER_CRITICAL()将中断和低于configMAX_SYSCALL_INTERRUPT_PRIORITY禁用中断优先级,和所有高优先级中断启用。

抢占式上下文切换只发生在中断中，因此在中断被禁用时不会发生。因此，调用taskENTER_CRITICAL()的任务被保证保持在Running状态，直到临界区退出，除非任务显式地试图阻塞或屈服(它不应该在临界区内部这样做)。

taskENTER_CRITICAL()和taskEXIT_CRITICAL()的调用被设计成嵌套。因此，只有当前面每次调用taskENTER_CRITICAL()都执行一次对taskEXIT_CRITICAL()的调用时，临界区才会退出。

临界段必须保持很短，否则它们将对中断响应时间产生不利影响。对taskENTER_CRITICAL()的每次调用都必须与对taskEXIT_CRITICAL()的调用紧密匹配。

不能在临界区内调用FreeRTOS API函数。

参数

无

返回值

无

2.15 taskENTER_CRITICAL_FROM_ISR()

```
#include “FreeRTOS.h”#include “task.h”UBaseType_t taskENTER_CRITICAL_FROM_ISR( void );
```

摘要

taskENTER_CRITICAL()的一个版本，可以用于中断服务例程(ISR)。

在ISR中，临界区通过调用taskENTER_CRITICAL_FROM_ISR()进入，随后通过调用taskEXIT_CRITICAL_FROM_ISR()退出。

taskENTER_CRITICAL_FROM_ISR()和taskEXIT_CRITICAL_FROM_ISR()宏提供了一个基本的临界区实现，它通过禁用中断来工作，可以是全局的，也可以是特定的中断优先级。

如果FreeRTOS端口用于支持中断嵌套调用taskENTER_CRITICAL_FROM_ISR下面()将禁用中断和中断优先级设定的configMAX_SYSCALL_INTERRUPT_PRIORITY(或configMAX_API_CALL_INTERRUPT_PRIORITY)内核配置不变,并将所有其他中断优先级启用。如果使用的FreeRTOS端口不支持中断嵌套，那么taskENTER_CRITICAL_FROM_ISR()和taskEXIT_CRITICAL_FROM_ISR()将不起作用。

taskENTER_CRITICAL_FROM_ISR()和taskEXIT_CRITICAL_FROM_ISR()的调用被设计为嵌套，但是宏的使用方式与taskENTER_CRITICAL()和taskEXIT_CRITICAL()的等价语义是不同的。

临界段必须保持很短，否则它们将对嵌套的高优先级中断的响应时间产生不利影响。对taskENTER_CRITICAL_FROM_ISR()的每次调用都必须与对taskEXIT_CRITICAL_FROM_ISR()的调用紧密匹配。

不能在临界区内调用FreeRTOS API函数。

参数

无

返回值

返回调用taskENTER_CRITICAL_FROM_ISR()时的中断掩码状态。必须保存返回值，以便将其传递给匹配的taskEXIT_CRITICAL_FROM_ISR()调用。

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 fromwithin a critical section, so entering this critical section will result in a nestingdepth of 2. Save the value returned by taskENTER_CRITICAL_FROM_ISR() into a localstack 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 acritical section, so interrupts will have already been disabled before a value wasstored in uxSavedInterruptStatus, and therefore passing uxSavedInterruptStatus intotaskEXIT_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 thereturned value into a local stack. */    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 afunction that includes its own calls to taskENTER_CRITICAL_FROM_ISR() andtaskEXIT_CRITICAL_FROM_ISR(). */    vDemoFunction();/* The operation that required the critical section is complete so exit thecritical section. Assuming interrupts were enabled on entry to this ISR, the valuesaved in uxSavedInterruptStatus will result in interrupts being re-enabled.*/ taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );}
```

2.16 taskEXIT_CRITICAL()

```
#include “FreeRTOS.h”#include “task.h”void taskEXIT_CRITICAL( void );
```

概要

通过调用taskENTER_CRITICAL()进入临界区，然后通过调用taskEXIT_CRITICAL()退出临界区。

taskEXIT_CRITICAL()不能从中断服务程序中调用。请参阅taskEXIT_CRITICAL_FROM_ISR()来获得一个中断安全的等效值。

taskENTER_CRITICAL()和taskEXIT_CRITICAL()宏提供了一个基本的临界区实现，它通过禁用中断来工作，可以是全局的，也可以是特定的中断优先级。

如果正在使用的FreeRTOS端口没有使用configMAX_SYSCALL_INTERRUPT_PRIORITY内核配置常量，那么调用taskENTER_CRITICAL()将使全局中断处于禁用状态。

如果正在使用的FreeRTOS端口确实使用了configMAX_SYSCALL_INTERRUPT_PRIORITY内核配置常量，那么调用taskENTER_CRITICAL()将使中断处于或低于configMAX_SYSCALL_INTERRUPT_PRIORITY设置的中断优先级禁用，并启用所有更高优先级的中断。

抢占式上下文切换只发生在中断中，因此在中断被禁用时不会发生。因此，调用taskENTER_CRITICAL()的任务被保证保持在Running状态，直到临界区退出，除非任务显式地试图阻塞或屈服(它不应该在临界区内部这样做)。

taskENTER_CRITICAL()和taskEXIT_CRITICAL()的调用被设计成嵌套。因此，只有当前面每次调用taskENTER_CRITICAL()都执行一次对taskEXIT_CRITICAL()的调用时，临界区才会退出。

临界段必须保持很短，否则它们将对中断响应时间产生不利影响。对taskENTER_CRITICAL()的每次调用都必须与对taskEXIT_CRITICAL()的调用紧密匹配。

不能在临界区内调用FreeRTOS API函数。

参数

无

返回值

无

2.17 taskEXIT_CRITICAL_FROM_ISR()

```
#include “FreeRTOS.h”#include “task.h”void taskENTER_CRITICAL_FROM_ISR( UBaseType_t uxSavedInterruptStatus );
```

摘要

退出通过调用taskENTER_CRITICAL_FROM_ISR()进入的临界区。

在ISR中，通过调用taskENTER_CRITICAL_FROM_ISR()进入临界区，然后通过调用taskEXIT_CRITICAL_FROM_ISR()退出临界区。

taskENTER_CRITICAL_FROM_ISR()和taskEXIT_CRITICAL_FROM_ISR()宏提供了一个基本的临界区实现，它通过禁用中断来工作，可以是全局的，也可以是特定的中断优先级。

如果FreeRTOS端口用于支持中断嵌套调用taskENTER_CRITICAL_FROM_ISR下面()将禁用中断和中断优先级设定的configMAX_SYSCALL_INTERRUPT_PRIORITY(或configMAX_API_CALL_INTERRUPT_PRIORITY)内核配置不变,并将所有其他中断优先级启用。如果使用的FreeRTOS端口不支持中断嵌套，那么taskENTER_CRITICAL_FROM_ISR()和taskEXIT_CRITICAL_FROM_ISR()将不起作用。

taskENTER_CRITICAL_FROM_ISR()和taskEXIT_CRITICAL_FROM_ISR()的调用被设计为嵌套，但是宏的使用方式与taskENTER_CRITICAL()和taskEXIT_CRITICAL()的等价语义是不同的。

临界段必须保持很短，否则它们将对嵌套的高优先级中断的响应时间产生不利影响。对taskENTER_CRITICAL_FROM_ISR()的每次调用都必须与对taskEXIT_CRITICAL_FROM_ISR()的调用紧密匹配。

不能在临界区内调用FreeRTOS API函数。

参数

uxSavedInterruptStatus

- 从对taskENTER_CRITICAL_FROM_ISR()的匹配调用返回的值必须用作uxSavedInterruptStatus值。

返回值

无

示例

2.18 xTaskGetApplicationTaskTag()

```
#include “FreeRTOS.h”#include “task.h”TaskHookFunction_t xTaskGetApplicationTaskTag(
TaskHandle_t xTask );
```

摘要

返回与任务关联的' tag '值。标记值的含义和使用由应用程序编写人员定义。内核本身通常不会访问标记值。

本功能仅供高级用户使用。

参数

xTask

- 正在查询的任务的句柄。这是主题任务。
- 任务可以通过使用自己的任务句柄或使用NULL代替有效的任务句柄来获得自己的标记值。

返回值

正在查询的任务的' tag '值。

提示

标记值可以用来保存函数指针。完成之后，可以使用xTaskCallApplicationTaskHook() API函数调用分配给标签值的函数。这种技术实际上是给任务分配一个回调函数。这样的回调通常与traceTASK_SWITCHED_IN()宏一起使用，以实现执行跟踪特性。

configUSE_APPLICATION_TASK_TAG必须在FreeRTOSConfig.h中设置为1，以便xTaskGetApplicationTaskTag()可用。

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 *) castis used to prevent compiler warnings. */    vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );    for( ;; )    {    /* Rest of task code goes here. */ }}void vAFunction( void ){    TaskHandle_t xHandle;    long lReturnedTaskHandle;/* Create a task from the vATask() function, storing the handle to the createdtask 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 inxHandle. */                                                ) == pdPASS )    {/* The task was created successfully. Delay for a short period to allowthe task to run. */            vTaskDelay( 100 );/* What tag value is assigned to the task? The returned tag value isstored in an integer, so cast to an integer to prevent compiler warnings. */            lReturnedTaskHandle = ( long ) xTaskGetApplicationTaskTag( xHandle );    }}
```

2.19 xTaskGetCurrentTaskHandle()

```
#include “FreeRTOS.h”#include “task.h”TaskHandle_t xTaskGetCurrentTaskHandle( void );
```

概要

返回处于Running状态的任务句柄——它将是调用xTaskGetCurrentTaskHandle()的任务的句柄。

参数

无

返回值

调用xTaskGetCurrentTaskHandle()的任务的句柄。

提示

INCLUDE_xTaskGetCurrentTaskHandle必须在FreeRTOSConfig.h中设置为1，以便xTaskGetCurrentTaskHandle()可用。

2.20 xTaskGetIdleTaskHandle()

```
#include “FreeRTOS.h”#include “task.h”TaskHandle_t xTaskGetIdleTaskHandle( void );
```

概要

返回与空闲任务关联的任务句柄。Idle任务在调度程序启动时自动创建。

参数

无

返回值

Idle任务的句柄。

提示

INCLUDE_xTaskGetIdleTaskHandle()必须在FreeRTOSConfig.h中设置为1，以便xTaskGetIdleTaskHandle()可用。

2.21 xTaskGetHandle()

```
#include “FreeRTOS.h”#include “task.h”TaskHandle_t xTaskGetHandle( const char
*pcNameToQuery );
```

概要

使用xTaskCreate()或xTaskCreateStatic()创建任务。这两个函数都有一个名为pcName的参数，用于将人类可读的文本名称分配给正在创建的任务。xTaskGetHandle()从任务的人类可读文本名称中查找并返回任务的句柄。

参数

pcNameToQuery

- 正在查询的任务名称。该名称被指定为一个标准的以NULL结尾的C字符串。

返回值

如果任务的名称与pcNameToQuery参数指定的名称完全相同，则将返回该任务的句柄。如果没有任务具有由pcNameToQuery参数指定的名称，则返回NULL。

提示

xTaskGetHandle()可能需要相当长的时间才能完成。因此，建议对每个任务名只使用xTaskGetHandle()一次——然后可以存储xTaskGetHandle()返回的任务句柄，以便以后重用。

xTaskGetHandle()的行为是未定义的，因为有多个任务具有相同的名称。为了让xTaskGetHandle()可用，必须在FreeRTOSConfig.h中将INCLUDE_xTaskGetHandle设置为1。

示例

```
void vATask( void *pvParameters ){    const char *pcNameToLookup = "MyTask";
TaskHandle_t xHandle;/* Find the handle of the task that has the name MyTask,
storing the returned handle locallyso it can be re-used later. */    xHandle =
xTaskGetHandle( pcNameToLookup );    if( xHandle != NULL )    {/* The handle of the
task was found, and can now be used in any other FreeRTOS APIfunction that takes a
TaskHandle_t parameter. */    }    for( ;; )    {    /* The rest of the task code
goes here. */    }}
```

2.22 uxTaskGetNumberOfTasks()

```
#include “FreeRTOS.h”#include “task.h”UBaseType_t uxTaskGetNumberOfTasks( void );
```

概要

返回在调用uxTaskGetNumberOfTasks()时存在的任务总数。

参数

无

返回值

返回值是调用uxTaskGetNumberOfTasks()时FreeRTOS内核控制的任务总数。这是挂起状态任务的数量，加上阻塞状态任务的数量，加上就绪状态任务的数量，加上空闲状态任务的数量，加上运行状态任务的数量。

2.23 vTaskGetRunTimeStats()

```
#include “FreeRTOS.h”#include “task.h”void vTaskGetRunTimeStats( char *pcWriteBuffer
);
```

概要

FreeRTOS可以配置为收集任务运行时统计信息。任务运行时统计信息提供关于每个任务收到的处理时间的信息。提供了绝对时间和应用程序总运行时间的百分比。vTaskGetRunTimeStats() API函数将收集到的运行时统计数据格式化为人类可读的表。为任务名称、分配给该任务的绝对时间以及分配给该任务的应用程序总运行时间的百分比生成列。系统中的每个任务都会生成一行，包括“Idle”任务。示例输出如下所示：

参数

pcWriteBuffer

- 一种指向字符缓冲区的指针，将格式化的可读表写入其中。缓冲区必须大到足以容纳整个表，因为不执行边界检查。

返回值

无

提示

vTaskGetRunTimeStats()是一个实用函数，提供它只是为了方便。它不被认为是内核的一部分。vTaskGetRunTimeStats()使用xTaskGetSystemState() API函数获取原始数据。

configGENERATE_RUN_TIME_STATS和configUSE_STATS_FORMATTING_FUNCTIONS都必须在FreeRTOSConfig.h中设置为1，以便vTaskGetRunTimeStats()可用。设置configGENERATE_RUN_TIME_STATS还需要应用程序定义以下宏:

- portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()
    - 必须提供此宏来初始化用于生成时间基数的任何外设。运行时统计数据所使用的时间基数必须具有比tick中断更高的分辨率，否则所收集的统计数据可能太不准确而不能真正有用。建议使时间基准比tick中断快10到20倍
- portGET_RUN_TIME_COUNTER_VALUE(), or portALT_GET_RUN_TIME_COUNTER_VALUE(Time)
    - 必须提供这两个宏中的一个来返回当前的时间基值——这是应用程序在选择的时间基单元中运行的总时间。如果使用了第一个宏，则必须定义该宏以计算当前的时间基值。如果使用第二个宏，则必须定义它将其"Time"参数设置为当前的时间基值。

这些宏可以在FreeRTOSConfig.h中定义。

示例

```
/* The LM3Sxxxx Eclipse demo application already includes a 20KHz timer
interrupt.The interrupt handler was updated to simply increment a variable
calledulHighFrequencyTimerTicks each time it
executed.portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() then sets this variable to 0
andportGET_RUN_TIME_COUNTER_VALUE() returns its value. To implement this the
following few lines are added to FreeRTOSConfig.h. */extern volatile unsigned long
ulHighFrequencyTimerTicks;/* ulHighFrequencyTimerTicks is already being incremented
at 20KHz. Just setits value back to 0. */#define
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() ( ulHighFrequencyTimerTicks = 0UL )/*
Simply return the high frequency counter value. */#define
portGET_RUN_TIME_COUNTER_VALUE() ulHighFrequencyTimerTicks/* The LPC17xx demo
application does not include the high frequency interrupt test,so
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() is used to configure the timer 0peripheral
to generate the time base. portGET_RUN_TIME_COUNTER_VALUE() simply returnsthe
current timer 0 counter value. This was implemented using the following functionsand
macros. *//* Defined in main.c. */void vConfigureTimerForRunTimeStats( void ){const
unsigned long TCR_COUNT_RESET = 2,CTCR_CTM_TIMER = 0x00,TCR_COUNT_ENABLE = 0x01;/*
Power up and feed the timer with a clock. */PCONP |= 0x02UL;PCLKSEL0 = (PCLKSEL0 &
(~(0x3<<2))) | (0x01 << 2);/* Reset Timer 0 */T0TCR = TCR_COUNT_RESET;/* Just count
up. */T0CTCR = CTCR_CTM_TIMER;/* Prescale to a frequency that is good enough to get
a decent resolution,but not too fast so as to overflow all the time. */T0PR = (
configCPU_CLOCK_HZ / 10000UL ) - 1UL;/* Start the counter. */T0TCR =
TCR_COUNT_ENABLE;}/* Defined in FreeRTOSConfig.h. */extern void
vConfigureTimerForRunTimeStats( void );#define
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() vConfigureTimerForRunTimeStats()#define
portGET_RUN_TIME_COUNTER_VALUE() T0TCvoid vAFunction( void ){/* Define a buffer that
is large enough to hold the generated table. In most casesthe buffer will be too
large to allocate on the stack, hence in this example it isdeclared static. */static
char cBuffer[ BUFFER_SIZE ];/* Pass the buffer into vTaskGetRunTimeStats() to
generate the table of data. */vTaskGetRunTimeStats( cBuffer );/* The generated
information can be saved or viewed here. */}
```

2.24 xTaskGetSchedulerState()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskGetSchedulerState( void );
```

概要

返回一个值，该值指示在调用xTaskGetSchedulerState()时调度程序所处的状态。

参数

无

返回值

taskSCHEDULER_NOT_STARTED

- 这个值只会在调用vTaskStartScheduler()之前调用xTaskGetSchedulerState()时返回。

taskSCHEDULER_RUNNING

- 如果vTaskStartScheduler()已经被调用，只要调度程序不在Suspended状态，则返回。

taskSCHEDULER_SUSPENDED

- 当调度程序处于Suspended状态时返回，因为调用了vTaskSuspendAll()。

提示

INCLUDE_xTaskGetSchedulerState必须在FreeRTOSConfig.h中设置为1，以便xTaskGetSchedulerState()可用。

2.25 uxTaskGetStackHighWaterMark()

```
#include “FreeRTOS.h”#include “task.h”UBaseType_t uxTaskGetStackHighWaterMark(
TaskHandle_t xTask );
```

概要

每个任务维护自己的堆栈，其总大小在创建任务时指定。uxTaskGetStackHighWaterMark()用于查询一个任务离溢出分配给它的堆栈空间有多近。这个值称为堆栈“高水位标志”。

参数

xTask

- 正在查询堆栈高水位的任务(主题任务)的句柄。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。
- 一个任务可以通过传递NULL来代替一个有效的任务句柄来查询它自己的堆栈高水位。

返回值

当任务执行和中断被处理时，任务所使用的堆栈量会增加或减少。uxTaskGetStackHighWaterMark()返回自任务开始执行以来可用的最小剩余堆栈空间。这是当堆栈使用率达到最大(或最深)值时，仍未使用的堆栈数量。高水位越接近于零，任务越接近于溢出它的堆栈。

提示

uxTaskGetStackHighWaterMark()可能需要相当长的时间来执行。因此，建议将其使用限于测试和调试构建。INCLUDE_uxTaskGetStackHighWaterMark()必须在FreeRTOSConfig.h中设置为1，以便uxTaskGetStackHighWaterMark()可用。

示例

```
void vTask1( void * pvParameters ){    UBaseType_t uxHighWaterMark;/* Inspect the high water mark of the calling task when the task starts toexecute. */ uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );    for( ;; )    {/* Call any function. */    vTaskDelay( 1000 );/* Calling a function will have used some stack space, so it will beexpected that uxTaskGetStackHighWaterMark() will return a lower valueat this point than when it was called on entry to the task function. */ uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );    }}
```

2.26 eTaskGetState()

```
#include “FreeRTOS.h”#include “task.h”eTaskState eTaskGetState( TaskHandle_t pxTask );
```

概要

作为枚举类型返回任务在eTaskGetState()执行时存在的状态。

参数

pxTask

- 主题任务的句柄
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

返回值

下表列出了由pxTask参数引用的任务可能存在的每个可能状态的eTaskGetState()返回的值。

State	Return Value
Running	eRunning (the task is querying its own state)
Ready	eReady
Blocked	eBlocked
Suspended	eSuspended
Deleted	eDeleted (the task’s structures are waiting to be cleaned up)

提示

为了让eTaskGetState() API函数可用，必须在FreeRTOSConfig.h中将INCLUDE_eTaskGetState设置为1。

2.27 uxTaskGetSystemState()

```
#include “FreeRTOS.h”#include “task.h”UBaseType_t uxTaskGetSystemState( TaskStatus_t
* const pxTaskStatusArray,                                    const UBaseType_t
uxArraySize,                                  unsigned long * const pulTotalRunTime
);
```

概要

uxTaskGetSystemState()

- 为系统中的每个任务填充一个TaskStatus_t结构。TaskStatus_t结构包含任务的句柄、名称、优先级、状态和消耗的运行时间总量。
- 示例中定义了TaskStatus_t结构。

参数

pxTaskStatusArray

- 一个指向TaskStatus_t结构数组的指针。对于RTOS控制下的每个任务，数组中必须至少包含一个TaskStatus_t结构。RTOS控制下的任务数量可以通过使用uxTaskGetNumberOfTasks() API函数来确定。

uxArraySize

- pxTaskStatusArray参数所指向的数组的大小。指定的大小是数组中索引的数量(数组中包含的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，那么uxTaskGetSystemState()将*pulTotalRunTime设置为自目标启动以来的总运行时间(由运行时统计时钟定义)。pulTotalRunTime可以设置为NULL，以忽略总运行时值。

返回值

由uxTaskGetSystemState()填充的TaskStatus_t结构的数量。这个值应该等于uxTaskGetNumberOfTasks() API函数返回的值，但是如果通过uxArraySize参数传递的值太小，则为零。

提示

此函数仅用于调试，因为使用它会导致调度器在较长时间内保持挂起。

要获取单个任务的信息，而不是系统中所有任务的信息，请使用vTaskGetTaskInfo()而不是uxTaskGetSystemState()。

configUSE_TRACE_FACILITY必须在FreeRTOSConfig.h中定义为1，以便uxTaskGetSystemState()可用。

示例

```
/* This example demonstrates how a human readable table of run time statsinformation 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 *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 thisfunction is executing. */    uxArraySize = uxTaskGetNumberOfTasks();/* Allocate a TaskStatus_t structure for each task. An array could beallocated 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, "%s\t\t%lu\t\t%lu%%\r\n", pxTaskStatusArray[ x ].pcTaskName,                    pxTaskStatusArray[ x ].ulRunTimeCounter,                    ulStatsAsPercentage );                }         else                {/* If the percentage is zero here then the task hasconsumed less than 1% of the total run time. */                    sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r\n", pxTaskStatusArray[ x ].pcTaskName,                    pxTaskStatusArray[ x ].ulRunTimeCounter );       }                pcWriteBuffer += strlen( ( char * ) pcWriteBuffer );   }        }/* The array is no longer needed, free the memory it consumes. */ vPortFree( pxTaskStatusArray );    }}typedef struct xTASK_STATUS{    /* The handle of the task to which the rest of the information in the structurerelates. */ TaskHandle_t xHandle;    /* A pointer to the task's name. This value will be invalid if the task was deletedsince 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 eCurrentState;/* The priority at which the task was running (may be inherited) when the structurewas populated. */    UBaseType_t uxCurrentPriority;/* The priority to which the task will return if the task's current priority has beeninherited to avoid unbounded priority inversion when obtaining a mutex. Only validif 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 time statsclock. Only valid when configGENERATE_RUN_TIME_STATS is defined as 1 inFreeRTOSConfig.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 wascreated. The closer this value is to zero the closer the task has come to overflowingits stack. */    unsigned short usStackHighWaterMark;} TaskStatus_t;
```

2.28 vTaskGetTaskInfo()

```
#include “FreeRTOS.h”#include “task.h”void vTaskGetTaskInfo( TaskHandle_t xTask,
                    TaskStatus_t *pxTaskStatus,                          BaseType_t
xGetFreeStackSpace,                         eTaskState eState );
```

概要

vTaskGetTaskInfo()为单个任务填充TaskStatus_t结构。TaskStatus_t结构包含任务的句柄、名称、优先级、状态和消耗的运行时间总量。

TaskStatus_t在前文已有定义。

参数

xTask

- 正在查询的任务的句柄。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

pxTaskStatus

- 必须指向类型为TaskStatus_t的变量，该变量将被查询的任务信息填充。

xGetFreeStackSpace

- TaskStatus_t结构体包含一个成员，用于报告正在查询的任务的堆栈高水位标记。堆栈高水位标志是任务曾经存在的最小堆栈空间，所以这个数字越接近零，任务越接近溢出其堆栈。计算堆栈高水位线需要相当长的时间，并且可能使系统暂时无响应—因此提供了xGetFreeStackSpace参数以允许跳过高水位线检查。如果xGetFreeStackSpace没有设置为pdFALSE，则高水位值只会写入TaskStatus_t结构。

eState

- TaskStatus_t结构体包含一个成员，用于报告正在查询的任务的状态。获取任务状态不如简单的赋值快——因此提供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，请将eState设置为eInvalid -否则在eState中传递的值将作为TaskStatus_t结构中的任务状态报告。

提示

此函数仅用于调试，因为使用它可能导致调度器在较长时间内保持挂起。

要获取系统中所有任务的TaskStatus_t结构，使用uxTaskGetSystemState()代替vTaskGetTaskInfo()。

configUSE_TRACE_FACILITY必须在FreeRTOSConfig.h中定义为1，以便uxTaskGetSystemState()可用。

示例

```
void vAFunction( void ){    TaskHandle_t xHandle;    TaskStatus_t xTaskDetails;/*
Obtain the handle of a task from its name. */    xHandle = xTaskGetHandle(
"Task_Name" );/* Check the handle is not NULL. */    configASSERT( xHandle );/* Use
the handle to obtain further information about the task. */    vTaskGetTaskInfo( /*
The handle of the task being queried. */                    xHandle,/* The
TaskStatus_t structure to complete with information on xHandle. */
&xTaskDetails,/* Include the stack high water mark value in the
TaskStatus_tstructure. */                    pdTRUE,/* Include the task state in the
TaskStatus_t structure. */                     eInvalid );}
```

2.29 pvTaskGetThreadLocalStoragePointer()

```
#include “FreeRTOS.h”#include “task.h”void *pvTaskGetThreadLocalStoragePointer(
TaskHandle_t xTaskToQuery,                                        BaseType_t
xIndex );
```

概要

线程本地存储(或TLS)允许应用程序编写器将值存储在任务的控制块中，使值特定于任务本身(局部于任务)，并允许每个任务拥有自己的唯一值。

每个任务都有自己的指针数组，可以用作线程本地存储。数组中的索引数由FreeRTOSConfig.h中的configignum_thread_local_storage_pointer编译时配置常量设置。

pvTaskGetThreadLocalStoragePointer()从数组的索引中读取一个值，有效地检索线程本地值。

参数

xTaskToQuery

- 从其中读取线程本地数据的任务的句柄。
- 一个任务可以通过使用NULL作为参数值读取它自己的线程本地数据。

xIndex

- 从其中读取数据的线程本地存储阵列的索引。

返回值

从任务的线程本地存储阵列索引xIndex读取的值。

示例

```
uint32_t ulVariable;/* Read the value stored in index 5 of the calling task's thread
local storagearray into ulVariable. */ulVariable = ( uint32_t )
pvTaskGetThreadLocalStoragePointer( NULL, 5 );
```

2.30 pcTaskGetName()

```
#include “FreeRTOS.h”#include “task.h”char * pcTaskGetName( TaskHandle_t
xTaskToQuery );
```

概要

查询任务的人类可读文本名称。使用用于创建任务的xTaskCreate()或xTaskCreateStatic() API函数调用的pcName参数，将文本名称分配给任务。

参数

xTaskToQuery

- 正在查询的任务的句柄(主题任务)。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。
- 任务可以通过传递NULL来代替有效的任务句柄来查询自己的名称。

返回值

任务名是标准的以NULL结尾的C字符串。返回的值是一个指向主题任务名称的指针。

2.31 xTaskGetTickCount()

```
#include “FreeRTOS.h”#include “task.h”TickType_t xTaskGetTickCount( void );
```

概要

滴答计数是自调度程序启动以来发生的滴答中断的总数。xTaskGetTickCount()返回当前滴答计数值。

参数

无

返回值

xTaskGetTickCount()总是返回调用xTaskGetTickCount()时的tick计数值。

提示

一个tick period代表的实际时间取决于在FreeRTOSConfig.h中分配给configTICK_RATE_HZ的值。pdMS_TO_TICKS()宏可用于将时间以毫秒为单位转换为以“ticks”为单位的时间。

滴答计数最终会溢出并返回到零。这将不会影响内核的内部操作—例如，任务将始终阻塞在指定的时间内，即使在任务处于Blocked状态时时钟计数溢出。但是，如果应用程序直接使用刻度数值，则宿主应用程序必须考虑溢出。

滴答计数溢出的频率取决于滴答频率和用于保存计数值的数据类型。如果configUSE_16_BIT_TICKS设置为1，那么滴答计数将被保存在一个16位变量中。如果configUSE_16_BIT_TICKS设置为0，那么滴答计数将保存在一个32位变量中。

示例

```
void vAFunction( void ){    TickType_t xTime1, xTime2, xExecutionTime;    /* Get the time the function started. */    xTime1 = xTaskGetTickCount();    /* Perform some operation. */    /* Get the time following the execution of the operation. */ xTime2 = xTaskGetTickCount();    /* Approximately how long did the operation take? */    xExecutionTime = xTime2 - xTime1;}
```

2.32 xTaskGetTickCountFromISR()

```
#include “FreeRTOS.h”#include “task.h”TickType_t xTaskGetTickCountFromISR( void );
```

概要

xTaskGetTickCount()的一个版本，可以从ISR调用。

滴答计数是自调度程序启动以来发生的滴答中断的总数。

参数

无

返回值

xTaskGetTickCountFromISR()总是在调用xTaskGetTickCountFromISR()时返回tick计数值。

提示

一个tick period代表的实际时间取决于在FreeRTOSConfig.h中分配给configTICK_RATE_HZ的值。pdMS_TO_TICKS()宏可用于将时间以毫秒为单位转换为以“ticks”为单位的时间。

滴答计数最终会溢出并返回到零。这将不会影响内核的内部操作—例如，任务将始终阻塞在指定的时间内，即使在任务处于Blocked状态时时钟计数溢出。但是，如果应用程序直接使用刻度数值，则宿主应用程序必须考虑溢出。

滴答计数溢出的频率取决于滴答频率和用于保存计数值的数据类型。如果configUSE_16_BIT_TICKS设置为1，那么滴答计数将被保存在一个16位变量中。如果configUSE_16_BIT_TICKS设置为0，那么滴答计数将保存在一个32位变量中。

示例

```
void vAnISR( void ){    static TickType_t xTimeISRLastExecuted = 0;    TickType_t xTimeNow, xTimeBetweenInterrupts;    /* Store the time at which this interrupt was entered. */    xTimeNow = xTaskGetTickCountFromISR();    /* Perform some operation. */    /* How many ticks occurred between this and the previous interrupt? */ xTimeBetweenInterrupts = xTimeISRLastExecuted - xTimeNow;    /* If more than 200 ticks occurred between this and the previous interrupt then    do something. */ if( xTimeBetweenInterrupts > 200 )    {    /* Take appropriate action here. */    }    /* Remember the time at which this interrupt was entered. */ xTimeISRLastExecuted = xTimeNow;}
```

2.33 vTaskList()

```
#include “FreeRTOS.h”#include “task.h”void vTaskList( char *pcWriteBuffer );
```

概要

在字符缓冲区中创建一个人类可读的表，描述在调用vTaskList()时每个任务的状态。下图显示了一个示例。

 image-20211115141907822

该表包括以下信息:

- Name——这是在创建任务时指定给任务的名称。
- State——调用vTaskList()时任务的状态，如下所示:
  - 'B '，如果任务处于阻塞状态。
  - 如果任务处于就绪状态，则使用' R '。
  - ' S '表示任务处于挂起状态，或处于未超时的阻塞状态。
  - 'D '如果任务已经被删除，但空闲任务还没有释放该任务用来保存其数据结构和堆栈的内存。
- Priority ——调用vTaskList()时分配给任务的优先级
- Stack——显示任务堆栈的“高水位线”。这是任务生命周期内可用的最小空闲堆栈数量。该值越接近于零，则任务越接近于溢出其堆栈。
- Num ——这是分配给每个任务的唯一数字。当多个任务被分配了相同的名称时，它除了帮助识别任务之外没有其他用途。

参数

pcWriteBuffer

- 将表文本写入其中的缓冲区。由于不执行边界检查，因此这个值必须大到足以容纳整个表。

返回值

无

提示

vTaskList()是一个实用函数，提供它只是为了方便。它不被认为是内核的一部分。vTaskList()使用xTaskGetSystemState() API函数获取原始数据。

vTaskList()将在其执行期间禁用中断。对于包含硬实时功能的应用程序来说，这可能是不可接受的。

configUSE_TRACE_FACILITY和configUSE_STATS_FORMATTING_FUNCTIONS都必须在FreeRTOSConfig.h中设置为1，以便vTaskList()可用。

默认情况下，vTaskList()使用标准库sprintf()函数。这可能会导致编译后的映像大小和堆栈使用量显著增加。FreeRTOS的下载包括一个名为printf-stdarg.c的文件中sprintf()的开源压缩版本。这可以用来代替标准库sprintf()，以帮助最小化代码大小的影响。注意，printf-stdarg.c是单独授权给FreeRTOS的。它的许可条款包含在文件中。

示例

```
void vAFunction( void ){/* Define a buffer that is large enough to hold the
generated table. In most casesthe buffer will be too large to allocate on the stack,
hence in this example it isdeclared static. */    static char cBuffer[ BUFFER_SIZE
];/* Pass the buffer into vTaskList() to generate the table of information. */
vTaskList( cBuffer );/* The generated information can be saved or viewed here. */}
```

2.34 xTaskNotify()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskNotify( TaskHandle_t
xTaskToNotify,                        uint32_t ulValue,
eNotifyAction eAction );
```

概要

每个任务都有一个32位的通知值，该值在创建任务时初始化为零。xTaskNotify()用于直接向任务发送事件，并可能解除阻塞，并可选地以以下方式之一更新接收任务的通知值:

- 将32位的数字写入通知值
- 增加一个通知值、
- 在通知值中设置一个或多个位
- 保持通知值不变

参数

xTaskToNotify

- 被通知的RTOS任务的句柄。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

ulValue

- 用于更新被通知任务的通知值。如何解释ulValue取决于eAction参数的值。

eAction

- 通知任务时要执行的操作。

  eAction是一种枚举类型，可以取以下值之一:

  - eNoAction -任务被通知，但通知值没有改变。在本例中不使用ulValue。
  - eSetBits -任务的通知值是用ulValue的位或表示的。例如，如果ulValue设置为0x01，那么第0位将在任务的通知值内设置。如果ulValue是0x04，那么将在任务的通知值中设置第2位。使用eSetBits允许任务通知作为一个更快和轻量级的替代事件组。
  - eIncrement -任务的通知值增加1。在本例中不使用ulValue。
  - esetvaluewiththoverwrite—任务的通知值被无条件设置为ulValue的值，即使当调用xTaskNotify()时任务已经有一个通知挂起。
  - eSetValueWithoutOverwrite -如果任务已经有一个通知挂起，那么它的通知值不会改变，xTaskNotify()返回pdFAIL。如果任务还没有挂起通知，则将其通知值设置为ulValue。

返回值

如果eAction设置为eSetValueWithoutOverwrite，并且任务的通知值没有更新，则返回pdFAIL。在所有其他情况下，返回pdPASS。

提示

如果任务的通知值被用作二进制或计数信号量的轻量级和更快的替代，那么使用更简单的xTaskNotifyGive() API函数而不是xTaskNotify()。

RTOS任务通知功能在默认情况下是启用的，可以通过在FreeRTOSConfig.h中将configUSE_TASK_NOTIFICATIONS设置为0来从构建中排除(每个任务节省8字节)。

示例

```
/* Set bit 8 in the notification value of the task referenced by xTask1Handle.
*/xTaskNotify( xTask1Handle, ( 1UL << 8UL ), eSetBits );/* Send a notification to
the task referenced by xTask2Handle, potentiallyremoving the task from the Blocked
state, but without updating the task'snotification 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 notificationvalue before the task had obtained it (by
a call to xTaskNotifyWait()or ulTaskNotifyTake()). */if( xTaskNotify( xTask4Handle,
0xfff, eSetValueWithoutOverwrite ) == pdPASS ){/* The task's notification value was
updated. */}else{/* The task's notification value was not updated. */}
```

2.35 xTaskNotifyAndQuery()

```
include “FreeRTOS.h”#include “task.h”BaseType_t xTaskNotifyAndQuery( TaskHandle_t
xTaskToNotify,                                uint32_t ulValue,
           eNotifyAction eAction,                                      uint32_t
*pulPreviousNotifyValue );
```

概要

xTaskNotifyAndQuery()与xTaskNotify()类似，但包含一个额外的参数，在该参数中返回主题任务以前的通知值。

每个任务都有一个32位的通知值，该值在创建任务时初始化为零。xTaskNotifyAndQuery()用于直接向任务发送事件，并潜在地解除阻塞，并可选地以以下方式之一更新接收任务的通知值:

- 将32位的数字写入通知值
- 增加一个通知值、
- 在通知值中设置一个或多个位
- 保持通知值不变

参数

xTaskToNotify

- 被通知的RTOS任务的句柄。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

ulValue

- 用于更新被通知任务的通知值。如何解释ulValue取决于eAction参数的值。

eAction

- 通知任务时要执行的操作。

  eAction是一种枚举类型，可以取以下值之一:

  - eNoAction -任务被通知，但通知值没有改变。在本例中不使用ulValue。
  - eSetBits -任务的通知值是用ulValue的位或表示的。例如，如果ulValue设置为0x01，那么第0位将在任务的通知值内设置。如果ulValue是0x04，那么将在任务的通知值中设置第2位。使用eSetBits允许任务通知作为一个更快和轻量级的替代事件组。
  - eIncrement -任务的通知值增加1。在本例中不使用ulValue。
  - esetvaluewiththoverwrite—任务的通知值被无条件设置为ulValue的值，即使当调用xTaskNotify()时任务已经有一个通知挂起。
  - eSetValueWithoutOverwrite -如果任务已经有一个通知挂起，那么它的通知值不会改变，xTaskNotify()返回pdFAIL。如果任务还没有挂起通知，则将其通知值设置为ulValue。

pulPreviousNotifyValue

- 用于在xTaskNotifyAndQuery()的操作修改任何位之前传递主题任务的通知值。

- pulPreviousNotifyValue是一个可选参数，如果不需要，可以设置为NULL。如果pulPreviousNotifyValue没有被使用，那么可以考虑使用xTaskNotify()来代替xTaskNotifyAndQuery()。

返回值

如果eAction设置为eSetValueWithoutOverwrite，并且任务的通知值没有更新，则返回pdFAIL。在所有其他情况下，返回pdPASS。

提示

如果任务的通知值被用作二进制或计数信号量的轻量级和更快的替代，那么使用更简单的xTaskNotifyGive() API函数而不是xTaskNotify()。

RTOS任务通知功能在默认情况下是启用的，可以通过在FreeRTOSConfig.h中将configUSE_TASK_NOTIFICATIONS设置为0来从构建中排除(每个任务节省8字节)。

示例

```
uint32_t ulPreviousValue;/* Set bit 8 in the notification value of the task
referenced by xTask1Handle. The task’s previous notification value is not needed, so
the last pulPreviousNotifyValue parameter is set to NULL. */xTaskNotifyAndQuery(
xTask1Handle, ( 1UL << 8UL ), eSetBits, NULL );/* 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. The task’s current notification
value is saved 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. Save the task’s previous notification value (before it was set to 0x50) in
ulPreviousValue. */xTaskNotifyAndQuery( xTask3Handle, 0x50, eSetValueWithOverwrite,
&ulPreviousValue );/* 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()
or ulTaskNotifyTake()). Save the task’s previous notification value (before it was
set to 0xfff) 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. */}
```

2.36 xTaskNotifyAndQueryFromISR()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskNotifyAndQuery( TaskHandle_t
xTaskToNotify,                                  uint32_t ulValue,
           eNotifyAction eAction,                                       uint32_t
*pulPreviousNotifyValue,                                  BaseType_t
*pxHigherPriorityTaskWoken );
```

概要

xTaskNotifyAndQuery()与xTaskNotify()类似，但包含一个额外的参数，在该参数中返回主题任务以前的通知值。xTaskNotifyAndQueryFromISR()是xTaskNotifyAndQuery()的一个版本，可以从中断服务例程(ISR)调用。

每个任务都有一个32位的通知值，该值在创建任务时初始化为零。xTaskNotifyAndQueryFromISR()用于直接向任务发送事件，并潜在地解除阻塞，并可选地以以下方式之一更新接收任务的通知值:

- 将32位的数字写入通知值
- 增加一个通知值、
- 在通知值中设置一个或多个位
- 保持通知值不变

参数

xTaskToNotify

- 被通知的RTOS任务的句柄。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

ulValue

- 用于更新被通知任务的通知值。如何解释ulValue取决于eAction参数的值。

eAction

- 通知任务时要执行的操作。

  eAction是一种枚举类型，可以取以下值之一:

  - eNoAction -任务被通知，但通知值没有改变。在本例中不使用ulValue。
  - eSetBits -任务的通知值是用ulValue的位或表示的。例如，如果ulValue设置为0x01，那么第0位将在任务的通知值内设置。如果ulValue是0x04，那么将在任务的通知值中设置第2位。使用eSetBits允许任务通知作为一个更快和轻量级的替代事件组。
  - eIncrement -任务的通知值增加1。在本例中不使用ulValue。
  - esetvaluewiththoverwrite—任务的通知值被无条件设置为ulValue的值，即使当调用xTaskNotify()时任务已经有一个通知挂起。
  - eSetValueWithoutOverwrite -如果任务已经有一个通知挂起，那么它的通知值不会改变，xTaskNotify()返回pdFAIL。如果任务还没有挂起通知，则将其通知值设置为ulValue。

pulPreviousNotifyValue

- 用于在xTaskNotifyAndQuery()的操作修改任何位之前传递主题任务的通知值。
- pulPreviousNotifyValue是一个可选参数，如果不需要，可以设置为NULL。如果pulPreviousNotifyValue没有被使用，那么可以考虑使用xTaskNotify()来代替xTaskNotifyAndQuery()。

pxHigherPriorityTaskWoken

- *pxHigherPriorityTaskWoken必须初始化为pdFALSE。如果发送通知导致被通知的任务离开Blocked状态，并且被通知的任务的优先级高于当前正在运行的任务，则xTaskNotifyAndQueryFromISR()将设置* pxHigherPriorityTaskWoken为pdTRUE。
- 如果xTaskNotifyAndQueryFromISR()将这个值设置为pdTRUE，那么应该在中断退出之前请求一个上下文切换。下文给出了一个示例。
- pxHigherPriorityTaskWoken是一个可选参数，可以设置为NULL。

返回值

如果eAction设置为eSetValueWithoutOverwrite，并且任务的通知值没有更新，则返回pdFAIL。在所有其他情况下，返回pdPASS。

提示

RTOS任务通知功能在默认情况下是启用的，可以通过在FreeRTOSConfig.h中将configUSE_TASK_NOTIFICATIONS设置为0来从构建中排除(每个任务节省8字节)。

示例

```
uint32_t ulPreviousValue;/* xHigherPriorityTaskWoken must be set to pdFALSE so it
can later be detected if it was set to pdTRUE by any of the functions called within
the interrupt. */BaseType_t xHigherPriorityTaskWoken = pdFALSE;/* Set bit 8 in the
notification value of the task referenced by xTask1Handle. The task's previous
notification value is not needed, so the last pulPreviousNotifyValue parameter is
set to NULL. */xTaskNotifyAndQueryFromISR( xTask1Handle,
( 1UL << 8UL ),                              eSetBits,
NULL,                           &xHigherPriorityTaskWoken );/* 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. The task's
current notification value is saved in ulPreviousValue. */
xTaskNotifyAndQueryFromISR( xTask2Handle,                            0,
               eNoAction,                          &ulPreviousValue,
             &xHigherPriorityTaskWoken );/* 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 );
```

2.37 xTaskNotifyFromISR()

```
#include "FreeRTOS.h"#include "task.h"BaseType_t xTaskNotifyFromISR( TaskHandle_t
xTaskToNotify,                                uint32_t ulValue,
           eNotifyAction eAction                                    BaseType_t
*pxHigherPriorityTaskWoken );
```

概要

xTaskNotify()的一个版本，可以从中断服务例程(ISR)调用。

每个任务都有一个32位的通知值，该值在创建任务时初始化为零。xTaskNotifyFromISR()用于直接向任务发送事件，并潜在地解除阻塞，并可选地以以下方式之一更新接收任务的通知值:

- 将32位的数字写入通知值
- 增加一个通知值、
- 在通知值中设置一个或多个位
- 保持通知值不变

参数

xTaskToNotify

- 被通知的RTOS任务的句柄。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

ulValue

- 用于更新被通知任务的通知值。如何解释ulValue取决于eAction参数的值。

eAction

- 通知任务时要执行的操作。

  eAction是一种枚举类型，可以取以下值之一:

  - eNoAction -任务被通知，但通知值没有改变。在本例中不使用ulValue。
  - eSetBits -任务的通知值是用ulValue的位或表示的。例如，如果ulValue设置为0x01，那么第0位将在任务的通知值内设置。如果ulValue是0x04，那么将在任务的通知值中设置第2位。使用eSetBits允许任务通知作为一个更快和轻量级的替代事件组。
  - eIncrement -任务的通知值增加1。在本例中不使用ulValue。
  - esetvaluewiththoverwrite—任务的通知值被无条件设置为ulValue的值，即使当调用xTaskNotify()时任务已经有一个通知挂起。
  - eSetValueWithoutOverwrite -如果任务已经有一个通知挂起，那么它的通知值不会改变，xTaskNotify()返回pdFAIL。如果任务还没有挂起通知，则将其通知值设置为ulValue。

pxHigherPriorityTaskWoken

- *pxHigherPriorityTaskWoken必须初始化为pdFALSE。如果发送通知导致被通知的任务离开Blocked状态，并且被通知的任务的优先级高于当前正在运行的任务，那么xTaskNotifyFromISR()将把*pxHigherPriorityTaskWoken设置为pdTRUE。
- 如果xTaskNotifyFromISR()将这个值设置为pdTRUE，那么应该在中断退出之前请求一个上下文切换，下文给出了一个示例。
- pxHigherPriorityTaskWoken是一个可选参数，可以设置为NULL。

返回值

如果eAction设置为eSetValueWithoutOverwrite，并且任务的通知值没有更新，则返回pdFAIL。在所有其他情况下，返回pdPASS。

提示

如果任务的通知值被用作二进制或计数信号量的轻量级和更快的替代，那么使用更简单的vTaskNotifyGiveFromISR() API函数而不是xTaskNotifyFromISR()。

RTOS任务通知功能在默认情况下是启用的，可以通过在FreeRTOSConfig.h中将configUSE_TASK_NOTIFICATIONS设置为0来从构建中排除(每个任务节省8字节)。

示例

这个例子演示了一个单独的RTOS任务用来处理两个独立的中断服务程序产生的事件——一个发送中断和一个接收中断。许多外围设备将使用相同的处理程序，在这种情况下，外围设备的中断状态寄存器可以简单地按位或与接收任务的通知值。

```
/* First bits are defined to represent each interrupt source. */#define TX_BIT
0x01#define RX_BIT 0x02/* The handle of the task that will receive notifications
from the interrupts. The handle was obtained when the task was created. */static
TaskHandle_t xHandlingTask;/* The implementation of the transmit interrupt service
routine. */void vTxISR( void ){    BaseType_t xHigherPriorityTaskWoken = pdFALSE;/*
Clear the interrupt source. */    prvClearInterrupt();/* Notify the task that the
transmission is complete by setting the TX_BIT in thetask's notification value. */
xTaskNotifyFromISR( xHandlingTask, TX_BIT, eSetBits, &xHigherPriorityTaskWoken );/*
If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch shouldbe
performed to ensure the interrupt returns directly to the highest prioritytask. The
macro used for this purpose is dependent on the port in use and may becalled
portEND_SWITCHING_ISR(). */    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );}/*---
----------------------------------------------------------*//* The implementation of
the receive interrupt service routine is identical except for the bit that gets set
in the receiving task's notification value. */void vRxISR( void ){    BaseType_t
xHigherPriorityTaskWoken = pdFALSE;/* Clear the interrupt source. */
prvClearInterrupt();/* Notify the task that the reception is complete by setting the
RX_BIT in thetask's notification value. */    xTaskNotifyFromISR( xHandlingTask,
RX_BIT, eSetBits, &xHigherPriorityTaskWoken );/* If xHigherPriorityTaskWoken is now
set to pdTRUE then a context switch shouldbe performed to ensure the interrupt
returns directly to the highest prioritytask. The macro used for this purpose is
dependent on the port in use and may becalled portEND_SWITCHING_ISR(). */
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );}/* The implementation of the task
that is notified by the interrupt serviceroutines. */static void prvHandlingTask(
void *pvParameter ){    const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 500 );
BaseType_t xResult;    for( ;; )    {/* Wait to be notified of an interrupt. */
   xResult = xTaskNotifyWait( pdFALSE, /* Don't clear bits on entry. */
ULONG_MAX, /* Clear all bits on exit. */            &ulNotifiedValue, /* Stores the
notified value. */            xMaxBlockTime );        if( xResult == pdPASS )
{/* A notification was received. See which bits were set. */            if( (
ulNotifiedValue & TX_BIT ) != 0 )            {/* The TX ISR has set a bit. */
         prvProcessTx();            }            if( ( ulNotifiedValue & RX_BIT ) !=
0 )            {/* The RX ISR has set a bit. */                prvProcessRx();
     }          }        else        {/* Did not receive a notification within the
expected time. */            prvCheckForErrors();        }    }}
```

2.38 xTaskNotifyGive()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskNotifyGive( TaskHandle_t
xTaskToNotify );
```

概要

每个任务都有一个32位的通知值，该值在创建任务时初始化为零。任务通知是直接发送给任务的事件，可以解除对接收任务的阻塞，并可选地更新接收任务的通知值。

xTaskNotifyGive()是一个宏，用于当任务通知值被用作二进制信号量或计数信号量的轻量级和更快的替代时使用。FreeRTOS信号量是使用xsemaphoregve () API函数给出的，而xTaskNotifyGive()是等效的，它使用接收任务的通知值而不是单独的信号量对象。

参数

xTaskToNotify

- 正在被通知并使其通知值增加的任务的句柄。

  要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

返回值

xTaskNotifyGive()是一个宏，它调用将eAction参数设置为eIncrement的xTaskNotify()。因此，总是返回pdPASS。

提示

当任务通知值被用作二进制信号量或计数信号量时，被通知的任务应该使用更简单的ulTaskNotifyTake() API函数而不是xTaskNotifyWait() API函数等待通知。

RTOS任务通知功能在默认情况下是启用的，可以通过在FreeRTOSConfig.h中将configUSE_TASK_NOTIFICATIONS设置为0来从构建中排除(每个任务节省8字节)。

示例

```
/* 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, thenstart the RTOS scheduler. */void main( void ){xTaskCreate( prvTask1, "Task1", 200, NULL, tskIDLE_PRIORITY, &xTask1 );xTaskCreate( prvTask2, "Task2", 200, NULL, tskIDLE_PRIORITY, &xTask2 );vTaskStartScheduler();}/*-----------------------------------------------------------*/static void prvTask1( void *pvParameters ){for( ;; ){/* Send a notification to prvTask2(), bringing it out of the Blockedstate. */xTaskNotifyGive( xTask2 );/* Block to wait for prvTask2() to notify this task. */ulTaskNotifyTake( pdTRUE, portMAX_DELAY );}}/*-----------------------------------------------------------*/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 Blockedstate. */xTaskNotifyGive( xTask1 );}}
```

2.39 vTaskNotifyGiveFromISR()

```
#include “FreeRTOS.h”#include “task.h”void vTaskNotifyGiveFromISR( TaskHandle_t xTaskToNotify,BaseType_t *pxHigherPriorityTaskWoken );
```

概要

可以从中断服务例程(ISR)调用的xTaskNotifyGive()的一个版本。

每个任务都有一个32位的通知值，该值在创建任务时初始化为零。任务通知是直接发送给任务的事件，可以解除对接收任务的阻塞，并可选地更新接收任务的通知值。

vTaskNotifyGiveFromISR()用于当任务通知值被用作二进制信号量或计数信号量的轻量级和更快的替代方法时。FreeRTOS信号量是使用xSemaphoreGiveFromISR() API函数给出的，而vTaskNotifyGiveFromISR()是等效的，它使用接收任务的通知值而不是单独的信号量对象。

参数

xTaskToNotify

- RTOS任务的句柄被通知并使其通知值增加。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

pxHigherPriorityTaskWoken

- *pxHigherPriorityTaskWoken必须初始化为pdFALSE。如果发送通知导致正在被通知的任务离开Blocked状态，并且未被阻塞的任务的优先级高于当前正在运行的任务，则vTaskNotifyGiveFromISR()会将*pxhigherprioritytaskkoken设置为pdTRUE。
- 如果vTaskNotifyGiveFromISR()将这个值设置为pdTRUE，那么应该在中断退出之前请求一个上下文切换。下文给出了一个示例。
- pxHigherPriorityTaskWoken是一个可选参数，可以设置为NULL。

提示

当任务通知值被用作二进制信号量或计数信号量时，被通知的任务应该使用ulTaskNotifyTake() API函数而不是xTaskNotifyWait() API函数等待通知。

RTOS任务通知功能在默认情况下是启用的，可以通过在FreeRTOSConfig.h中将configUSE_TASK_NOTIFICATIONS设置为0来从构建中排除(每个任务节省8字节)。

示例

这是一个通用外设驱动程序中的传输函数的示例。任务调用传输函数，然后在Blocked状态中等待(因此不使用CPU时间)，直到收到传输完成的通知。传输由DMA执行，DMA结束中断用于通知任务。

```
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();/* Start the transmission - an interrupt is generated when the transmissionis 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 inprogress. */configASSERT( xTaskToNotify != NULL );/* Notify the task that the transmission is complete. */vTaskNotifyGiveFromISR( xTaskToNotify, &xHigherPriorityTaskWoken );/* There are no transmissions in progress, so no tasks to notify. */xTaskToNotify = NULL;/* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switchshould be performed to ensure the interrupt returns directly to the highestpriority task. The macro used for this purpose is dependent on the port inuse and may be called portEND_SWITCHING_ISR(). */portYIELD_FROM_ISR( xHigherPriorityTaskWoken );}/*-----------------------------------------------------*//* The task that initiates the transmission, then enters the Blocked state (sonot 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 = ulTaskNotifyTake( pdFALSE, xMaxBlockTime );if( ulNotificationValue == 1 ){/* The transmission ended as expected. */}else{/* The call to ulTaskNotifyTake() timed out. */}}
```

2.40 xTaskNotifyStateClear()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskNotifyStateClear( TaskHandle_t xTask )
```

概要

每个任务都有一个32位的通知值，该值在创建任务时初始化为零。任务通知是直接发送给任务的事件，可以解除对接收任务的阻塞，并可选地更新接收任务的通知值。

如果任务处于“阻塞”状态等待通知到达，则该任务立即退出“阻塞”状态，通知不再保持挂起。如果任务在收到通知时没有等待通知，那么通知将一直挂起，直到:

- 接收任务读取通知值。
- 接收任务是调用xTaskNotifyStateClear()中的主题任务。

xTaskNotifyStateClear()将清除挂起的通知，但不更改通知值。

参数

xTask

- 将清除挂起通知的任务的句柄。将xTask设置为NULL将清除调用xTaskNotifyStateClear()的任务中一个挂起的通知。

返回值

如果xTask引用的任务有一个通知挂起，则返回pdPASS。如果xTask引用的任务没有挂起通知，则返回pdFAIL。

示例

```
/* An example UART transmit 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,uint16_t usStringLength ){const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );/* xSendingTask holds the handle of the task waiting for the transmission tocomplete. If xSendingTask is NULL then a transmission is not in progress.Don't start to send a new string unless transmission of the previous stringis complete. */if( ( xSendingTask == NULL ) && ( usStringLength > 0 ) ){/* Ensure the calling task’s notification state is not already pending. */xTaskNotifyStateClear( NULL );/* Store the handle of the transmitting task. This is used to unblockthe task when the transmission has completed. */xSendingTask = xTaskGetCurrentTaskHandle();/* Start sending the string - the transmission is then controlled by aninterrupt. */UARTSendString( pcStringToSend, usStringLength );/* Wait in the Blocked state (so not using any CPU time) until the UARTISR sends a notification to xSendingTask to notify (and unblock) the taskwhen the transmission is complete. */ulTaskNotifyTake( pdTRUE, xMaxBlockTime );}}
```

2.41 ulTaskNotifyTake()

```
#include “FreeRTOS.h”#include “task.h”uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t xTicksToWait );
```

概要

每个任务都有一个32位的通知值，该值在创建任务时初始化为零。任务通知是直接发送给任务的事件，可以解除对接收任务的阻塞，并可选地更新接收任务的通知值。

ulTaskNotifyTake()用于当任务通知被用作二进制信号量或计数信号量的更快和更轻的替代方法时。FreeRTOS信号量是使用xSemaphoreTake() API函数获取的，ulTaskNotifyTake()相当于使用任务通知值而不是单独的信号量对象。

正如xTaskNotifyWait()将在通知挂起时返回，ulTaskNotifyTake()将在任务的通知值不为零时返回，在它返回之前递减任务的通知值。

任务可以使用ulTaskNotifyTake()来选择性地阻塞以等待任务的通知值为非零。任务处于阻塞状态时不占用CPU时间。

ulTaskNotifyTake()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;也可以在退出时减少任务的通知值，在这种情况下，通知值的作用更类似于计数信号量。

参数

xClearCountOnExit

- 如果xClearCountOnExit设置为pdFALSE，那么任务的通知值在ulTaskNotifyTake()退出之前递减。这相当于通过成功调用xSemaphoreTake()来减少一个计数信号量的值。
- 如果xClearCountOnExit设置为pdTRUE，那么在ulTaskNotifyTake()退出之前，任务的通知值将重置为0。这相当于在成功调用xSemaphoreTake()之后，二进制信号量的值被保留为零(或空，或'不可用')。

xTicksToWait

- 当调用ulTaskNotifyTake()时，如果通知尚未挂起，则在Blocked状态下等待接收通知的最大时间。
- RTOS任务处于Blocked状态时不占用CPU时间。
- 时间以RTOS的时间周期指定。可以使用pdMS_TO_TICKS()宏将以毫秒为单位的时间转换为以节拍为单位的时间。

返回值

任务的通知值在减少或清除之前的值(参见xClearCountOnExit的描述)。

提示

当一个任务使用它的通知值作为二进制信号或计数信号量时，其他任务和中断应该使用xTaskNotifyGive()宏或xTaskNotify()函数(函数的eAction参数设置为eIncrement)向它发送通知(这两者是等价的)。

RTOS任务通知功能在默认情况下是启用的，可以通过在FreeRTOSConfig.h中将configUSE_TASK_NOTIFICATIONS设置为0来从构建中排除(每个任务节省8字节)。

示例

```
/* An interrupt handler that 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 then 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 = pdFALSE;/* Clear the interrupt. */prvClearInterruptSource();/* Unblock the handling task so the task can perform any processing necessitatedby the interrupt. xHandlingTask is the task's handle, which was obtainedwhen the task was created. */vTaskNotifyGiveFromISR( xHandlingTask, &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 calledportEND_SWITCHING_ISR(). */portYIELD_FROM_ISR( xHigherPriorityTaskWoken );}/*-------------------------------------------------------------*//* Task that blocks waiting to be notified that the peripheral needs servicing. */void vHandlingTask( void *pvParameters ){BaseType_t xEvent;for( ;; ){/* Block indefinitely (without a timeout, so no need to check the function'sreturn value) to wait for a notification. Here the RTOS task notificationis being used as a binary semaphore, so the notification value is clearedto zero on exit. NOTE! Real applications should not block indefinitely,but instead time out occasionally in order to handle error conditionsthat may prevent the interrupt from sending any more notifications. */ulTaskNotifyTake( pdTRUE, /* Clear the notification value on exit. */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 eventspending in the peripheral have been processed. */do{xEvent = xQueryPeripheral();if( xEvent != NO_MORE_EVENTS ){vProcessPeripheralEvent( xEvent );}} while( xEvent != NO_MORE_EVENTS );}}
```

2.42 xTaskNotifyWait()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,uint32_t ulBitsToClearOnExit,uint32_t *pulNotificationValue,TickType_t xTicksToWait );
```

概要

每个任务都有一个32位的通知值，该值在创建任务时初始化为零。任务通知是直接发送给任务的事件，它可以解除对接收任务的阻塞，并可以通过多种不同的方式更新接收任务的通知值。例如，通知可以覆盖接收任务的通知值，或者只在接收任务的通知值中设置一个或多个比特。有关示例，请参阅xTaskNotify() API文档。

xTaskNotifyWait()使用可选超时等待调用任务接收通知。

如果接收任务已经被阻塞，等待通知，当收到通知时，接收任务将从阻塞状态中移除，通知将被清除。

参数

ulBitsToClearOnEntry

- 在ulBitsToClearOnEntry中设置的任何位都将在进入xTaskNotifyWait()函数时在调用任务的通知值中被清除(在任务等待一个新的通知之前)，前提是当xTaskNotifyWait()被调用时，通知还没有挂起。
- 例如，如果ulBitsToClearOnEntry是0x01，那么任务的通知值的第0位将在进入函数时被清除。
- 将ulBitsToClearOnEntry设置为0xffffffff (ULONG_MAX)将清除任务通知值中的所有位，有效地将该值清除为0。

ulBitsToClearOnExit

- 如果收到通知，在ulBitsToClearOnExit中设置的任何位都将在xTaskNotifyWait()函数退出之前在调用任务的通知值中清除。
- 当任务的通知值保存在*pulNotificationValue中(参见下面pulNotificationValue的描述)后，这些位将被清除。
- 例如，如果ulBitsToClearOnExit为0x03，那么在函数退出之前，任务通知值的第0位和第1位将被清除。
- 设置ulBitsToClearOnExit为0xffffffff (ULONG_MAX)将清除任务通知值中的所有位，有效地将该值清除为0。

pulNotificationValue

- 用于传递任务的通知值。复制到*pulNotificationValue的值是任务的通知值，因为ulBitsToClearOnExit设置导致任何位被清除之前的值。
- pulNotificationValue是一个可选参数，如果不需要，可以设置为NULL。

xTicksToWait

- 当调用xTaskNotifyWait()时，如果通知尚未挂起，则在Blocked状态下等待接收通知的最大时间。
- 任务处于阻塞状态时不占用CPU时间。
- 时间以RTOS的时间周期指定。可以使用pdMS_TO_TICKS()宏将以毫秒为单位的时间转换为以节拍为单位的时间。

返回值

如果收到了通知，或者当调用xTaskNotifyWait()时通知已经挂起，则返回pdTRUE。

如果在收到通知之前对xTaskNotifyWait()的调用超时，则返回pdFALSE。

提示

如果您使用任务通知来实现二进制或计数信号量类型的行为，那么使用更简单的ulTaskNotifyTake() API函数而不是xTaskNotifyWait()。

RTOS任务通知功能在默认情况下是启用的，可以通过在FreeRTOSConfig.h中将configUSE_TASK_NOTIFICATIONS设置为0来从构建中排除(每个任务节省8字节)。

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 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'sreturn value) to wait for a notification.Bits in this RTOS task's notification value are set by the notifyingtasks and interrupts to indicate which events have occurred. */xTaskNotifyWait( 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. */}}
```

2.43 uxTaskPriorityGet()

```
#include “FreeRTOS.h”#include “task.h”UBaseType_t uxTaskPriorityGet( TaskHandle_t pxTask );
```

概要

查询在调用uxTaskPriorityGet()时分配给任务的优先级。

参数

pxTask

- 正在查询的任务的句柄(主题任务)。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。
- 任务可以通过传递NULL来代替有效的任务句柄来查询自己的优先级。

返回值

返回的值是调用uxTaskPriorityGet()时查询的任务的优先级。

示例

```
void vAFunction( void ){TaskHandle_t xHandle;UBaseType_t uxCreatedPriority,
uxOurPriority;/* Create a task, storing the handle of the created task in xHandle.
*/if( xTaskCreate( vTaskCode,"Demo task",STACK_SIZE, NULL, PRIORITY,&xHandle) !=
pdPASS ){/* The task was not created successfully. */}else{/* Use the handle to
query the priority of the created task. */uxCreatedPriority = uxTaskPriorityGet(
xHandle );/* Query the priority of the calling task by using NULL in place ofa valid
task handle. */uxOurPriority = uxTaskPriorityGet( NULL );/* Is the priority of this
task higher than the priority of the taskjust created? */if( uxOurPriority >
uxCreatedPriority ){/* Yes. */}}}
```

2.44 vTaskPrioritySet()

```
#include “FreeRTOS.h”#include “task.h”void vTaskPrioritySet( TaskHandle_t pxTask,
UBaseType_t uxNewPriority );
```

概要

修改任务的优先级。

参数

pxTask

- 正在修改的任务(主题任务)的句柄。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。
- 任务可以通过传递NULL来代替有效的任务句柄来改变自己的优先级。

uxNewPriority

- 主题任务将被设置的优先级。优先级可以从0(最低优先级)分配到(configMAX_PRIORITIES - 1)(最高优先级)。
- configMAX_PRIORITIES在FreeRTOSConfig.h中定义。传递一个以上的值(configMAX_PRIORITIES - 1)将导致分配给任务的优先级上限为最大合法值。

返回值

无

提示

vTaskPrioritySet()只能从正在执行的任务中调用，因此不能在调度器处于初始化状态(在启动调度器之前)时调用。

可能有一组任务都被阻塞，等待同一个队列或信号量事件。这些任务将根据它们的优先级排序——例如，第一个事件将解除阻塞正在等待事件的优先级最高的任务，第二个事件将解除阻塞最初正在等待事件的优先级第二高的任务，等等。使用vTaskPrioritySet()更改这样一个被阻塞任务的优先级不会导致被阻塞任务的评估顺序被重新评估。

示例

```
void vAFunction( void ){TaskHandle_t xHandle;/* Create a task, storing the handle of the created task in xHandle. */if( xTaskCreate( vTaskCode,"Demo task",STACK_SIZE,NULL,PRIORITY,&xHandle) != pdPASS ){/* The task was not created successfully. */}else{/* Use the handle to raise the priority of the created task. */vTaskPrioritySet( xHandle, PRIORITY + 1 );/* Use NULL in place of a valid task handle to set the priority of thecalling task to 1. */vTaskPrioritySet( NULL, 1 );}}
```

2.45 vTaskResume()

```
#include “FreeRTOS.h”#include “task.h”void vTaskResume( TaskHandle_t pxTaskToResume );
```

概要

将任务从挂起状态转移到就绪状态。之前必须使用vTaskSuspend()调用将任务置于Suspended状态。

参数

pxTaskToResume

- 正在恢复的任务的句柄(从挂起状态转换出来)。这是主题任务。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

返回值

无

提示

可以通过阻塞任务来等待队列事件，指定超时时间。通过调用vTaskSuspend()将这样的Blocked任务移到Suspended状态，然后通过调用vTaskResume()将其移出Suspended状态并移到Ready状态是合法的。在此场景下，下一次任务进入Running状态时，它将检查其超时时间是否已经(同时)过期。如果超时时间还未到期，任务将再次进入Blocked状态，等待队列事件直到最初指定的超时时间的剩余时间。

也可以使用vTaskDelay()或vTaskDelayUntil() API函数阻塞任务以等待临时事件。通过调用vTaskSuspend()将这样的Blocked任务移到Suspended状态，然后通过调用vTaskResume()将其移出Suspended状态并移到Ready状态是合法的。按照这个场景，下一次任务进入Running状态时，它将退出vTaskDelay()或vTaskDelayUntil()函数，就像指定的延迟时间已经过期一样，即使实际情况并非如此。

vTaskResume()只能从正在执行的任务中调用，因此不能在调度器处于初始化状态(在启动调度器之前)时调用。

示例

```
void vAFunction( void ){TaskHandle_t xHandle;/* Create a task, storing the handle to the created task in xHandle. */if( xTaskCreate( vTaskCode,"Demo task",STACK_SIZE,NULL,PRIORITY,&xHandle) != pdPASS ){/* The task was not created successfully. */}else{/* Use the handle to suspend the created task. */vTaskSuspend( xHandle );/* The suspended task will not run during this period, unless another taskcalls vTaskResume( xHandle ). *//* Resume the suspended task again. */vTaskResume( xHandle );/* The created task is again available to the scheduler and can enterThe Running state. */}}
```

2.46 xTaskResumeAll()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskResumeAll( void );
```

概要

在先前调用vTaskSuspendAll()之后，通过将调度器从挂起状态转换为活动状态，恢复调度器活动。

参数

无

返回值

pdTRUE

- 调度程序已转换为活动状态。该转换导致发生挂起的上下文切换。

pdFALSE

- 调度程序被转换到Active状态且转换没有导致发生上下文切换，或者由于对vTaskSuspendAll()的嵌套调用，调度程序处于Suspended状态。

提示

调度程序可以通过调用vTaskSuspendAll()来挂起。当调度程序被挂起时，中断仍然是启用的，但是不会发生上下文切换。如果在调度器挂起时请求上下文切换，则请求将被挂起，直到调度器恢复(取消挂起)。

对vTaskSuspendAll()的调用可以嵌套。在调度器将离开Suspended状态并重新进入Active状态之前，必须对xTaskResumeAll()进行与之前对vTaskSuspendAll()相同数量的调用。

xTaskResumeAll()只能从正在执行的任务中调用，因此不能在调度器处于初始化状态(在启动调度器之前)时调用。

当调度器挂起时，不应该调用其他FreeRTOS API函数。

示例

```
/* A function that suspends then resumes the scheduler. */void vDemoFunction( void ) {/* This function suspends the scheduler. When it is called from vTask1 thescheduler is already suspended, so this call creates a nesting depth of 2. */vTaskSuspendAll();/* Perform an action here. *//* As calls to vTaskSuspendAll() are now nested, resuming the scheduler heredoes 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 itdoes not want to get swapped out, or it wants to access data which is alsoaccessed from another task (but not from an interrupt). It cannot usetaskENTER_CRITICAL()/taskEXIT_CRITICAL() as the length of the operation maycause interrupts to be missed. *//* Prevent the scheduler from performing a context switch. */vTaskSuspendAll();/* Perform the operation here. There is no need to use critical sectionsas the task has all the processing time other than that utilized by interruptservice routines.*//* Calls to vTaskSuspendAll() can be nested, so it is safe to call a (nonAPI) function that also calls vTaskSuspendAll(). API functions should notbe called while the scheduler is suspended. */vDemoFunction();/* The operation is complete. Set the scheduler back into the Activestate. */if( xTaskResumeAll() == pdTRUE ){/* A context switch occurred within xTaskResumeAll(). */}else{/* A context switch did not occur within xTaskResumeAll(). */}}}
```

2.47 xTaskResumeFromISR()

```
#include “FreeRTOS.h”#include “task.h”BaseType_t xTaskResumeFromISR( TaskHandle_t pxTaskToResume );
```

概要

vTaskResume()的一个版本，可以从中断服务程序中调用。

参数

pxTaskToResume

- 正在恢复的任务的句柄(从挂起状态转换出来)。这是主题任务。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。

返回值

pdTRUE

- 如果正在恢复(未阻塞)的任务的优先级大于或等于当前正在执行的任务(被中断的任务)，则返回——这意味着在退出中断之前应该执行上下文切换。

pdFALSE

- 如果正在恢复的任务的优先级低于当前正在执行的任务(被中断的任务)，则返回-这意味着在退出中断之前不需要执行上下文切换。

提示

可以通过调用vTaskSuspend()来挂起任务。当处于挂起状态时，将不会选择任务以进入运行状态。vTaskResume()和xTaskResumeFromISR()可以用来恢复(取消挂起)挂起的任务。可以从中断中调用xtaskresume()，但vTaskResume()不能。

对vTaskSuspend()的调用不维护嵌套计数。一个被多个vTaskSuspend()调用挂起的任务总是会被一个vTaskResume()或xTaskResumeFromISR()调用取消挂起。

xTaskResumeFromISR()不能用于将任务与中断同步。如果中断事件发生的速度快于相关任务级处理函数的执行，那么这样做将导致中断事件被遗漏。任务和中断同步可以使用二进制信号量或计数信号量安全地实现，因为信号量将锁存事件。

示例

```
TaskHandle_t xHandle;void vAFunction( void ){/* Create a task, storing the handle of the created task in xHandle. */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 by using NULL as the parameter to vTaskSuspend()in place of a valid task handle. */vTaskSuspend( NULL );/* The task is now suspended, so will not reach here until the ISR resumes(un-suspends) it. */}}void vAnExampleISR( void ){BaseType_t xYieldRequired;/* Resume the suspended task. */xYieldRequired = xTaskResumeFromISR( xHandle );if( xYieldRequired == pdTRUE ){/* A context switch should now be performed so the ISR returns directly tothe resumed task. This is because the resumed task had a priority that wasequal to or higher than the task that is currently in the Running state.NOTE: The syntax required to perform a context switch from an ISR variesfrom port to port, and from compiler to compiler. Check the documentation andexamples for the port being used to find the syntax required by yourapplication. It is likely that this if() statement can be replaced by asingle call to portYIELD_FROM_ISR() [or portEND_SWITCHING_ISR()] usingxYieldRequired as the macro parameter:portYIELD_FROM_ISR( xYieldRequired );*/portYIELD_FROM_ISR();}}
```

2.48 vTaskSetApplicationTaskTag()

```
#include “FreeRTOS.h”#include “task.h”void vTaskSetApplicationTaskTag( TaskHandle_t xTask, TaskHookFunction_t pxTagValue );
```

概要

本功能仅供高级用户使用。

API函数vTaskSetApplicationTaskTag()可用于为任务分配一个' tag '值。标记值的含义和使用由应用程序编写人员定义。内核本身通常不会访问标记值。

参数

xTask

- 要为其分配标记值的任务的句柄。这是主题任务。
- 任务可以通过使用自己的任务句柄或使用NULL代替有效的任务句柄来为自己分配标记值。

pxTagValue

- 指定为主题任务标记值的值。它的类型是TaskHookFunction_t，允许将函数指针赋给标记，尽管通过强制类型转换，标记值可以是任何类型。

返回值

无

提示

标记值可以用来保存函数指针。完成之后，可以使用xTaskCallApplicationTaskHook() API函数调用分配给标签值的函数。这种技术实际上是给任务分配一个回调函数。这样的回调通常与traceTASK_SWITCHED_IN()宏一起使用，以实现执行跟踪特性。

configUSE_APPLICATION_TASK_TAG必须在FreeRTOSConfig.h中设置为1，以便vTaskSetApplicationTaskTag()可用。

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 *) castis used to prevent compiler warnings. */vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );for( ;; ){/* Rest of task code goes here. */}}/* In this example a callback function is assigned as the task tag. First define thecallback 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, updatingthe task state, outputting a value, etc. */return 0;}/* Now define the task that sets prvExampleTaskHook() as its hook/tag value. This isin effect registering the task callback function. */void vAnotherTask( void *pvParameters ){/* Register a callback function for the currently running (calling) task. */vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );for( ;; ){/* Rest of task code goes here. */}}/* [As an example use of the hook (callback)] Define the traceTASK_SWITCHED_OUT()macro to call the hook function. The kernel will then automatically call the taskhook each time the task is switched out. This technique can be used to generatean execution trace. pxCurrentTCB references the currently executing task. */#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )
```

2.49 vTaskSetThreadLocalStoragePointer()

```
#include “FreeRTOS.h”#include “task.h”void vTaskSetThreadLocalStoragePointer(
TaskHandle_t xTaskToSet,BaseType_t xIndex,void *pvValue );
```

概要

线程本地存储(或TLS)允许应用程序编写器将值存储在任务的控制块中，使值特定于任务本身(局部于任务)，并允许每个任务拥有自己的唯一值。

每个任务都有自己的指针数组，可以用作线程本地存储。数组中的索引数由FreeRTOSConfig.h中的configignum_thread_local_storage_pointer编译时配置常量设置。

vTaskSetThreadLocalStoragePointer()设置数组中索引的值，有效地存储线程本地值。

参数

xTaskToSet

- 正在向其写入线程本地数据的任务的句柄。
- 一个任务可以通过使用NULL作为参数值写入它自己的线程本地数据。

xIndex

- 要写入数据的线程本地存储阵列的索引。

pvValue

- 要写入到xIndex指定的索引中的值。

返回值

无

示例

```
uint32_t ulVariable;/* Write the 32-bit 0x12345678 value directly into index 1 of
the thread localstorage array. Passing NULL as the task handle has the effect of
writing to thecalling task's thread local storage array.
*/vTaskSetThreadLocalStoragePointer( NULL, /* Task handle. */1, /* Index into the
array. */( void * ) 0x12345678 );/* Store the value of the 32-bit variable
ulVariable to index 0 of the callingtask's thread local storage array. */ulVariable
= ERROR_CODE;vTaskSetThreadLocalStoragePointer( NULL, /* Task handle. */0, /* Index
into the array. */( void * ) &ulVariable );
```

2.50 vTaskSetTimeOutState()

```
#include “FreeRTOS.h”#include “task.h”void vTaskSetTimeOutState( TimeOut_t * const pxTimeOut );
```

概要

本功能仅供高级用户使用。

任务可以进入“Blocked”状态，等待事件发生。通常，任务不会在Blocked状态下无限期等待，而是指定一个超时时间。如果在任务等待的事件发生之前超时，则任务将从阻塞状态中移除。

如果一个任务进入和退出阻塞状态时不止一次等待事件发生,那么每一次使用的超时任务进入阻塞状态必须进行调整,以确保所有阻塞状态的时间不超过原指定的超时时间。xTaskCheckForTimeOut()执行调整，考虑到偶尔发生的情况，如tick计数溢出，否则手动调整容易出错。

vTaskSetTimeOutState()与xTaskCheckForTimeOut()一起使用。调用vTaskSetTimeOutState()来设置初始条件，之后可以调用xTaskCheckForTimeOut()来检查超时条件，如果没有超时则调整剩余的块时间。

参数

pxTimeOut

- 一个指向结构体的指针，该结构体将被初始化以保存确定是否发生超时所必需的信息。

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 Blockedstate. Adjusting xTicksToWait to account for any time that has been spent in theBlocked state within this function so far to ensure the total amount of time spentin 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 receiveinterrupt has placed more data into the buffer. */ulTaskNotifyTake( pdTRUE, xTicksToWait );}/* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The actualnumber of bytes read (which might be less than uxWantedBytes) is returned. */uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );return uxReceived;}
```

2.51 vTaskStartScheduler()

```
#include “FreeRTOS.h”#include “task.h”void vTaskStartScheduler( void );
```

概要

启动FreeRTOS调度器运行。

通常，在调度程序启动之前，main()(或main()调用的函数)将被执行。启动调度程序后，只有任务和中断才会执行。

启动调度器会导致在调度器处于初始化状态时创建的最高优先级任务进入运行状态。

参数

无

返回值

Idle任务在调度程序启动时自动创建。vTaskStartScheduler()只会在没有足够的FreeRTOS堆内存可用来创建空闲任务时返回。

提示

在ARM7和ARM9微控制器上执行的端口要求处理器在vTaskStartScheduler()被调用之前处于Supervisor模式。

示例

```
TaskHandle_t xHandle;/* Define a task function. */void vATask( void ){for( ;; ){/* Task code goes here. */}}void main( void ){/* Create at least one task, in this case the task function defined above iscreated. Calling vTaskStartScheduler() before any tasks have been createdwill cause the idle task to enter the Running state. */xTaskCreate( vTaskCode, "task name", STACK_SIZE, NULL, TASK_PRIORITY, NULL );/* Start the scheduler. */vTaskStartScheduler();/* This code will only be reached if the idle task could not be created insidevTaskStartScheduler(). An infinite loop is used to assist debugging byensuring this scenario does not result in main() exiting. */for( ;; );}
```

2.52 vTaskStepTick()

```
#include “FreeRTOS.h”#include “task.h”void vTaskStepTick( TickType_t xTicksToJump );
```

概要

如果RTOS被配置为使用无tick空闲功能，那么tick中断将被停止，当idle任务是唯一能够执行的任务时，微控制器将处于低功耗状态。在退出低功率状态时，必须修正滴答计数值，以反映停止时所经过的时间。

如果FreeRTOS端口包含默认的portSUPPRESS_TICKS_AND_SLEEP()实现，则在内部使用vTaskStepTick()以确保维护正确的tick计数值。vTaskStepTick()是一个公共API函数，允许重写默认的portSUPPRESS_TICKS_AND_SLEEP()实现，如果使用的端口没有提供默认值，则提供一个portSUPPRESS_TICKS_AND_SLEEP()。

参数

xTicksToJump

- 在时钟中断停止和重新启动之间经过的RTOS时钟周期的数量(时钟中断被抑制的时间)。为了正确操作，参数必须小于或等于portSUPPRESS_TICKS_AND_SLEEP()参数。

返回值

无

提示

configUSE_TICKLESS_IDLE必须在FreeRTOSConfig.h中设置为1，以便vTaskStepTick()可用。

示例

```
/* This is an example of how portSUPPRESS_TICKS_AND_SLEEP() might be implemented by an application writer. This basic implementation will introduce inaccuracies in the tracking of the time maintained by the kernel in relation to calendar time. Official FreeRTOS implementations account for these inaccuracies as much as possible.Only vTaskStepTick() is part of the FreeRTOS API. The other function calls are for demonstration only. *//* First define the portSUPPRESS_TICKS_AND_SLEEP() macro. 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 whenthe 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 stateat the time the kernel next needs to execute. The interrupt must be generatedfrom a source that remains operational when the microcontroller is in a lowpower 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 outof low power mode by an interrupt other than that configured by thevSetWakeTimeInterrupt() call. Note that the scheduler is suspended beforeportSUPPRESS_TICKS_AND_SLEEP() is called, and resumed whenportSUPPRESS_TICKS_AND_SLEEP() returns. Therefore no other tasks will executeuntil this function completes. */ulLowPowerTimeAfterSleep = ulGetExternalTime();/* Correct the kernels tick count to account for the time the microcontrollerspent in its low power state. */vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );/* Restart the timer that is generating the tick interrupt. */prvStartTickInterruptTimer();}
```

2.53 vTaskSuspend()

```
#include “FreeRTOS.h”#include “task.h”void vTaskSuspend( TaskHandle_t pxTaskToSuspend );
```

概要

将任务置于挂起状态。处于挂起状态的任务永远不会被选择进入运行状态。

从Suspended状态删除任务的唯一方法是使其成为vTaskResume()调用的主题。

参数

pxTaskToSuspend

- 被挂起的任务的句柄。
- 要获得任务的句柄，使用xTaskCreate()创建任务并使用pxCreatedTask参数，或者使用xTaskCreateStatic()创建任务并存储返回值，或者在调用xTaskGetHandle()时使用任务的名称。
- 任务可以通过传递NULL来代替有效的任务句柄来挂起自己。

返回值

无

提示

如果使用的是FreeRTOS版本6.1.0或更高版本，那么在调度程序启动之前(在调用vTaskStartScheduler()之前)，可以调用vTaskSuspend()将任务置于Suspended状态。这将导致任务(有效地)以挂起状态启动。

示例

```
void vAFunction( void ){TaskHandle_t xHandle;/* Create a task, storing the handle of the created task in xHandle. */if( xTaskCreate( vTaskCode,"Demo task",STACK_SIZE,NULL,PRIORITY,&xHandle) != pdPASS ){/* The task was not created successfully. */}else{/* Use the handle of the created task to place the task in the Suspendedstate. From FreeRTOS version 6.1.0, this can be done before the Schedulerhas been started. */vTaskSuspend( xHandle );/* The created task will not run during this period, unless another taskcalls vTaskResume( xHandle ). *//* Use a NULL parameter to suspend the calling task. */vTaskSuspend( NULL );/* This task can only execute past the call to vTaskSuspend( NULL ) ifanother task has resumed (un-suspended) it using a call to vTaskResume(). */}}
```

2.54 vTaskSuspendAll()

```
#include “FreeRTOS.h”#include “task.h”void vTaskSuspendAll( void );
```

概要

暂停调度器。暂停调度程序可以防止发生上下文切换，但保留中断。如果一个中断请求在调度器挂起时进行上下文切换，那么该请求将被挂起并仅在调度器恢复(未挂起)时执行。

参数

无

返回值

无

提示

在之前调用vTaskSuspendAll()之后，对xTaskResumeAll()的调用将调度器从Suspended状态转移出来。

对vTaskSuspendAll()的调用可以嵌套。在调度器将离开Suspended状态并重新进入Active状态之前，必须对xTaskResumeAll()进行与之前对vTaskSuspendAll()相同数量的调用。

xTaskResumeAll()只能从正在执行的任务中调用，因此不能在调度器处于初始化状态(在启动调度器之前)时调用。

当调度器挂起时，不能调用其他的FreeRTOS API函数。

示例

```
/* A function that suspends then resumes the scheduler. */void vDemoFunction( void ){/* This function suspends the scheduler. When it is called from vTask1 thescheduler 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 willnot 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 doesnot want to get swapped out, or it wants to access data which is alsoaccessed from another task (but not from an interrupt). It cannot usetaskENTER_CRITICAL()/taskEXIT_CRITICAL() as the length of the operation maycause 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 asthe task has all the processing time other than that utilized by interruptservice routines.*//* Calls to vTaskSuspendAll() can be nested so it is safe to call a (non API)function which also contains calls to vTaskSuspendAll(). API functionsshould not be called while the scheduler is suspended. */vDemoFunction();/* The operation is complete. Set the scheduler back into the Activestate. */if( xTaskResumeAll() == pdTRUE ){/* A context switch occurred within xTaskResumeAll(). */}else{/* A context switch did not occur within xTaskResumeAll(). */}}}
```

2.55 taskYIELD()

```
#include “FreeRTOS.h”#include “task.h”void taskYIELD( void );
```

概要

让位于另一个同等重要的任务。

让步是指任务自愿离开Running状态，而不被抢占，并且在它的时间片被充分利用之前。

参数

无

返回值

无

提示

taskYIELD()只能从正在执行的任务中调用，因此不能在调度器处于初始化状态(在启动调度器之前)时调用。

当一个任务调用taskYIELD()时，调度程序将选择另一个同等优先级的Ready状态任务，以进入Running状态。如果没有其他同等优先级的Ready状态任务，那么调用taskYIELD()的任务本身将直接转换回Running状态。

调度器只会选择一个与调用taskYIELD()的任务同等优先级的任务，因为如果有任何优先级更高的任务处于Ready状态，调用taskYIELD()的任务就不会首先执行。

示例

```
void vATask( void * pvParameters){for( ;; ){/* Perform some actions. *//* If there are any tasks of equal priority to this task that are in theReady state then let them execute now - even though this task has not usedall of its time slice. */taskYIELD();/* If there were any tasks of equal priority to this task in the Ready state,then they will have executed before this task reaches here. */}}
```

章节3 队列

3.1 vQueueAddToRegistry()

```
#include “FreeRTOS.h”#include “queue.h”void vQueueAddToRegistry( QueueHandle_t xQueue, char *pcQueueName );
```

概要

队列分配一个可读的名称，并将该队列添加到队列注册表。

参数

xQueue

- 将被添加到注册表中的队列句柄。也可以使用信号量句柄。

pcQueueName

- 队列或信号量的描述性名称。这不是FreeRTOS以任何方式使用的。它纯粹是作为调试辅助而包含的。通过人类可读的名称标识队列或信号量比尝试通过其句柄标识队列或信号量简单得多。

返回值

无

提示

队列注册表被内核感知调试器使用:

1. 它允许将文本名称与队列或信号量关联起来，以便在调试接口中轻松识别队列和信号量。
2. 它为调试器提供了一种方法来定位队列和信号量结构。

configQUEUE_REGISTRY_SIZE内核配置常量定义了一次可以注册的队列和信号量的最大数量。只有需要在内核感知调试接口中查看的队列和信号量才需要注册。

只有在使用内核感知调试器时才需要队列注册表。在其他任何时候，它都没有作用，可以通过将configQUEUE_REGISTRY_SIZE设置为0来省略它，或者完全省略configQUEUE_REGISTRY_SIZE配置常量定义。

删除已注册队列将自动将其从注册表中删除。

示例

```
void vAFunction( void ){QueueHandle_t xQueue;/* Create a queue big enough to hold 10 chars. */xQueue = xQueueCreate( 10, sizeof( char ) );/* The created queue needs to be viewable in a kernel aware debugger, soadd it to the registry. */vQueueAddToRegistry( xQueue, "AMeaningfulName" );
```

3.2 xQueueAddToSet()

```
include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueAddToSet( QueueSetMemberHandle_t xQueueOrSemaphore,QueueSetHandle_t xQueueSet );
```

概要

将队列或信号量添加到以前由调用xQueueCreateSet()创建的队列集。

接收(在队列中)或接受(在信号量中)操作不能在队列集合的成员上执行，除非对xQueueSelectFromSet()的调用首先返回了该集合成员的句柄。

参数

xQueueOrSemaphore

- 添加到队列集的队列句柄或信号量(转换为QueueSetMemberHandle_t类型)。

xQueueSet

- 要向其添加队列或信号量的队列设置的句柄。

返回值

pdPASS

- 队列或信号量成功添加到队列集。

pdFAIL

- 无法将队列或信号量添加到队列集中，因为它已经是不同集合的成员。

提示

为了使xQueueAddToSet() API函数可用，必须在FreeRTOSConfig.h中将configUSE_QUEUE_SETS设置为1。

示例

请参阅本手册中为xQueueCreateSet()函数提供的示例。

3.3 xQueueCreate()

```
#include “FreeRTOS.h”#include “queue.h”QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,UBaseType_t uxItemSize );
```

概要

创建一个新队列并返回一个句柄，该句柄可用于引用该队列。

每个队列都需要RAM, RAM用于保存队列状态，并保存队列中包含的项(队列存储区域)。如果使用xQueueCreate()创建队列，则从FreeRTOS堆中自动分配所需的RAM。如果使用xQueueCreateStatic()创建队列，则由应用程序编写器提供RAM，这将导致更多的参数，但允许在编译时静态分配RAM。

参数

uxQueueLength

- 正在创建的队列在任何时候可以容纳的最大项数。

uxItemSize

- 可存储在队列中的每个数据项的大小，以字节为单位。

返回值

NULL

- 无法创建队列，因为没有足够的堆内存供FreeRTOS分配队列数据结构和存储区域。

任何其他值

- 队列创建成功。返回值是一个句柄，可通过该句柄引用创建的队列。

提示

队列用于在任务之间、任务和中断之间传递数据。

可以在启动调度程序之前或之后创建队列。

configSUPPORT_DYNAMIC_ALLOCATION必须在FreeRTOSConfig.h中设置为1，或者不设置为未定义，以使该函数可用。

示例

```
/* Define the data type that will be queued. */typedef struct A_Message{char ucMessageID;char ucData[ 20 ];} AMessage;/* Define the queue parameters. */#define QUEUE_LENGTH 5#define QUEUE_ITEM_SIZE sizeof( AMessage )int main( void ) {QueueHandle_t xQueue;/* Create the queue, storing the returned handle in the xQueue variable. */xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );if( xQueue == NULL ){/* The queue could not be created. */}/* Rest of code goes here. */}
```

3.4 xQueueCreateSet()

```
#include “FreeRTOS.h”#include “queue.h”QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength );
```

概要

队列集提供了一种机制，允许RTOS任务阻塞(pend)来自多个RTOS队列或信号量的读操作。请注意，除了使用队列集，还有更简单的替代方法。更多信息请参见FreeRTOS.org网站上的“屏蔽多对象”页面。

在使用队列集之前，必须通过调用xQueueCreateSet()显式创建队列集。一旦创建，标准的FreeRTOS队列和信号量就可以通过调用xQueueAddToSet()添加到集合中。然后使用xQueueSelectFromSet()来确定集合中包含的哪个队列或信号量处于读取队列或信号量操作将成功的状态(如果有的话)。

参数

uxEventQueueLength

- 队列集存储发生在队列和集合中包含的信号量上的事件。uxEventQueueLength指定一次可以排队的最大事件数。

  为了绝对确保事件不会丢失，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:

- 如果一个队列集容纳一个长度为5的队列，另一个长度为12的队列，以及一个二进制信号量，那么uxEventQueueLength应该设置为(5 + 12 + 1)或18。
- 如果一个队列设置为容纳三个二进制信号量，那么uxeventqueuellength应该设置为(1 + 1 + 1)或3。
- 如果一个队列集要保存一个最大计数量为5的计数信号量和一个最大计数量为3的计数信号量，那么uxEventQueueLength应该设置为(5 + 3)或8。

返回值

NULL

- 无法创建队列集。

其他任何值

- 成功创建队列集。返回值是一个句柄，可通过该句柄引用创建的队列集。

提示

在包含互斥锁的队列集中阻塞不会导致互斥锁持有者继承阻塞任务的优先级。

添加到队列集的每个队列中的每个空间都需要额外的4字节RAM。因此，具有较高的最大计数值的计数信号量不应该被添加到队列集中。

接收(在队列中)或接受(在信号量中)操作不能在队列集合的成员上执行，除非对xQueueSelectFromSet()的调用首先返回了该集合成员的句柄。

为了使xQueueCreateSet() API函数可用，必须在FreeRTOSConfig.h中将configUSE_QUEUE_SETS设置为1。

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 a queue set large enough to hold an event for every space in everyqueue 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();/* Take the semaphore, so it starts empty. A block time of zero can be usedas the semaphore is guaranteed to be available - it has just been created. */xSemaphoreTake( xSemaphore, 0 );/* Add the queues and semaphores to the set. Reading from these queues andsemaphore can only be performed after a call to xQueueSelectFromSet() hasreturned the queue or semaphore handle from this point on. */xQueueAddToSet( xQueue1, xQueueSet );xQueueAddToSet( xQueue2, xQueueSet );xQueueAddToSet( xSemaphore, xQueueSet );/* CONTINUED ON NEXT PAGE */
```

3.5 xQueueCreateStatic()

```
#include “FreeRTOS.h”#include “queue.h”QueueHandle_t xQueueCreateStatic( UBaseType_t uxQueueLength,UBaseType_t uxItemSize,uint8_t *pucQueueStorageBuffer,StaticQueue_t *pxQueueBuffer );
```

概要

创建一个新队列并返回一个句柄，该句柄可用于引用该队列。

每个队列都需要RAM, RAM用于保存队列状态，并保存队列中包含的项(队列存储区域)。如果使用xQueueCreate()创建队列，则从FreeRTOS堆中自动分配所需的RAM。如果使用xQueueCreateStatic()创建队列，则由应用程序编写器提供RAM，这将导致更多的参数，但允许在编译时静态分配RAM。

参数

uxQueueLength

- 正在创建的队列在任何时候可以容纳的最大项数。

uxItemSize

- 可存储在队列中的每个数据项的大小，以字节为单位。

pucQueueStorageBuffer

- 如果uxItemSize不为零，那么pucQueueStorageBuffer必须指向一个uint8_t数组，该数组至少大到足以在任何时间容纳队列中最大的项目数量-即(uxQueueLength * uxItemSize)字节。
- 如果uxItemSize为0，则pucQueueStorageBuffer可以为NULL，因为没有日期将被复制到队列存储区域。

pxQueueBuffer

- 必须指向StaticQueue_t类型的变量，该变量将用于保存队列的数据结构。

返回值

NULL

- 因为pxQueueBuffer为NULL，所以没有创建队列。

其他任何值

- 已创建队列，返回的值是已创建队列的句柄。

提示

队列用于在任务之间、任务和中断之间传递数据。

可以在启动调度程序之前或之后创建队列。

configSUPPORT_STATIC_ALLOCATION必须在FreeRTOSConfig.h中设置为1，才能使用该函数。

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( xQueue );}
```

3.6 vQueueDelete()

```
#include “FreeRTOS.h”#include “queue.h”void vQueueDelete( TaskHandle_t pxQueueToDelete );
```

概要

删除以前通过调用xQueueCreate()或xQueueCreateStatic()创建的队列。vqueueddelete()也可以用于删除信号量。

参数

pxQueueToDelete

- 被删除队列的句柄。也可以使用信号量句柄。

返回值

无

提示

队列用于在任务之间以及任务和中断之间传递数据。

任务可以选择块队列上/信号(用一个可选的超时)如果他们试图发送数据到队列/信号量和队列/信号量已经满了,或者他们试图从队列接收数据/信号量和队列/信号已经空了。如果队列/信号量上有当前阻塞的任务，则不能删除该队列/信号量。

示例

```
/* Define the data type that will be queued. */typedef struct A_Message{char ucMessageID;char ucData[ 20 ];} AMessage;/* Define the queue parameters. */#define QUEUE_LENGTH 5#define QUEUE_ITEM_SIZE sizeof( AMessage )int main( void ) {QueueHandle_t xQueue;/* Create the queue, storing the returned handle in the xQueue variable. */xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );if( xQueue == NULL ){/* The queue could not be created. */}else{/* Delete the queue again by passing xQueue to vQueueDelete(). */vQueueDelete( xQueue );}}
```

3.7 pcQueueGetName()

```
#include “FreeRTOS.h”#include “queue.h”const char *pcQueueGetName( QueueHandle_t xQueue );
```

概要

查询队列的可读文本名称。

如果队列已被添加到队列注册表，那么它将只有一个文本名称。请参阅vQueueAddToRegistry() API函数。

参数

xQueue

- 被查询队列的句柄。

返回值

队列名是标准的以NULL结尾的C字符串。返回的值是一个指向正在查询的队列名称的指针。

3.8 xQueueIsQueueEmptyFromISR()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueIsQueueEmptyFromISR( const QueueHandle_t pxQueue );
```

概要

查询队列以查看它是否包含项目，或者它是否已经为空。如果队列为空，则无法从队列接收项目。

这个函数只能在ISR中使用。

参数

pxQueue

- 正在查询的队列。

返回值

pdFALSE

- 在调用xQueueIsQueueEmptyFromISR()时，查询的队列是空的(不包含任何数据项)。

其他任何值

- 在调用xQueueIsQueueEmptyFromISR()时，查询的队列不是空的(包含数据项)。

3.9 QueueIsQueueFullFromISR()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueIsQueueFullFromISR( const QueueHandle_t pxQueue );
```

概要

查询一个队列，看看它是否已经满了，或者它是否有空间接收一个新项目。队列只有在未满时才能成功接收新项目。

这个函数只能在ISR中使用。

参数

pxQueue

- 正在查询的队列。

返回值

pdFALSE

- 调用xQueueIsQueueFullFromISR()时，查询的队列还没有满。

其他任何值

- 在调用xQueueIsQueueFullFromISR()时，查询的队列已满。

3.10 uxQueueMessagesWaiting()

```
#include “FreeRTOS.h”#include “queue.h”UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );
```

概要

返回当前保存在队列中的项的数量。

参数

xQueue

- 被查询队列的句柄。

返回值

在调用uxQueueMessagesWaiting()时查询的队列中持有的项的数量。

示例

```
void vAFunction( QueueHandle_t xQueue ){UBaseType_t uxNumberOfItems;/* How many items are currently in the queue referenced by the xQueue handle? */uxNumberOfItems = uxQueueMessagesWaiting( xQueue );}
```

3.11 uxQueueMessagesWaitingFromISR()

```
#include “FreeRTOS.h”#include “queue.h”UBaseType_t uxQueueMessagesWaitingFromISR( const QueueHandle_t xQueue );
```

概要

uxQueueMessagesWaiting()的一个版本，可以在中断服务例程中使用。

参数

xQueue

- 被查询队列的句柄。

返回值

在调用uxQueueMessagesWaitingFromISR()时查询的队列中包含的项的数量。

示例

```
void vAnInterruptHandler( void ){UBaseType_t uxNumberOfItems;BaseType_t
xHigherPriorityTaskWoken = pdFALSE;/* Check the status of the queue, if it contains
more than 10 items then wake thetask that will drain the queue. *//* How many items
are currently in the queue referenced by the xQueue handle? */uxNumberOfItems =
uxQueueMessagesWaitingFromISR( xQueue );if( uxNumberOfItems > 10 ){/* The task being
woken is currently blocked on xSemaphore. Giving thesemaphore will unblock the task.
*/xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );}/* If
xHigherPriorityTaskWoken is equal to pdTRUE at this point then the taskthat was
unblocked by the call to xSemaphoreGiveFromISR() had a priority eitherequal to or
greater than the currently executing task (the task that was inthe Running state
when this interrupt occurred). In that case a context switchshould be performed
before leaving this interrupt service routine to ensure theinterrupt returns to the
highest priority ready state task (the task that wasunblocked). The syntax required
to perform a context switch from inside aninterrupt varies from port to port, and
from compiler to compiler. Check theweb documentation and examples for the port in
use to find the correct syntaxfor your application. */}
```

3.12 xQueueOverwrite()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueOverwrite( QueueHandle_t
xQueue, const void *pvItemToQueue );
```

概要

xQueueSendToBack()的一个版本，即使队列已满也将写入队列，覆盖队列中已经保存的数据。

xQueueOverwrite()用于长度为1的队列，这意味着队列要么是空的，要么是满的。

这个函数不能从中断服务程序中调用。请参阅xQueueOverwriteFromISR()以获得可用于中断服务例程的替代方案。

参数

xQueue

- 将数据发送到的队列的句柄。

pvItemToQueue

- 指向要放入队列中的项的指针。队列可以容纳的每个项的大小是在创建队列时设置的，这些字节将从pvItemToQueue复制到队列存储区域。

返回值

xQueueOverwrite()是一个调用xQueueGenericSend()的宏，因此具有与xQueueSendToFront()相同的返回值。然而，pdPASS是唯一可以返回的值，因为xQueueOverwrite()将写入队列，即使队列已经满了。

示例

```
void vFunction( void *pvParameters ){QueueHandle_t xQueue;unsigned long ulVarToSend, ulValReceived;/* Create a queue to hold one unsigned long value. It is stronglyrecommended *not* to use xQueueOverwrite() on queues that cancontain more than one value, and doing so will trigger an assertionif 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 inthe queue. A block time of zero is used as it is known that thequeue 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 thevalue 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 thequeue was already full when the value was written. */if( ulValReceived != 100 ){/* Error unless another task is using the same queue. */}/* ... */}
```

2.13 3.13 xQueueOverwriteFromISR()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueOverwriteFromISR( QueueHandle_t xQueue,const void *pvItemToQueue,BaseType_t *pxHigherPriorityTaskWoken );
```

概要

可以在ISR中使用的xQueueOverwrite()版本。xQueueOverwriteFromISR()类似于xQueueSendToBackFromISR()，但即使队列已满也会写入队列，覆盖队列中已经保存的数据。

xQueueOverwriteFromISR()用于长度为1的队列，这意味着队列要么是空的，要么是满的。

参数

xQueue

- 将数据发送到的队列的句柄。

pvItemToQueue

- 指向要放入队列中的项的指针。队列可以容纳的每个项的大小是在创建队列时设置的，这些字节将从pvItemToQueue复制到队列存储区域。

pxHigherPriorityTaskWoken

- xQueueOverwriteFromISR()将把*pxHigherPriorityTaskWoken设置为pdTRUE，如果发送到队列导致一个任务被解除阻塞，并且被解除阻塞的任务的优先级高于当前正在运行的任务。如果xQueueOverwriteFromISR()将这个值设置为pdTRUE，那么应该在中断退出之前请求一个上下文切换。请参阅文档中的中断服务例程部分，了解如何使用端口。

返回值

xQueueOverwriteFromISR()是一个调用xQueueGenericSendFromISR()的宏，因此具有与xQueueSendToFrontFromISR()相同的返回值。然而，pdPASS是唯一可以返回的值，因为xQueueOverwriteFromISR()将写入队列，即使队列已经满了。

示例

```
QueueHandle_t xQueue;void vFunction( void *pvParameters ){/* Create a queue to hold one unsigned long value. It is stronglyrecommended not to use xQueueOverwriteFromISR() on queues that cancontain more than one value, and doing so will trigger an assertionif 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 stillpass because the value held in the queue will be overwritten with thenew 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 taskhas a priority higher than or equal to the priority of the currentlyexecuting task (the task this interrupt interrupted). Perform a contextswitch so this interrupt returns directly to the unblocked task. */portYIELD_FROM_ISR(); /* or portEND_SWITCHING_ISR() depending on the port.*/}}
```

3.14 xQueuePeek()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueuePeek( QueueHandle_t
xQueue,void *pvBuffer, TickType_
```

概要

下一次使用xQueueReceive()或xQueuePeek()从相同队列中获取项时，将返回相同的项。

参数

xQueue

- 要从其中读取数据的队列的句柄。

pvBuffer

- 从队列中读取的数据将被复制到其中的内存指针。
- 缓冲区的长度必须至少等于队列项的大小。项目大小将由用于创建队列的xQueueCreate()或xQueueCreateStatic()调用的uxItemSize参数设置。

xTicksToWait

- 当队列已经为空时，任务应保持在Blocked状态以等待队列上的数据可用的最大时间。
- 如果xTicksToWait为零，那么如果队列已经为空，xQueuePeek()将立即返回。
- 块时间以滴答周期指定，因此它所代表的绝对时间依赖于滴答频率。可以使用pdMS_TO_TICKS()宏将以毫秒为单位的时间转换为以节拍为单位的时间。
- 将xTicksToWait设置为portMAX_DELAY将导致任务无限期等待(不会超时)，前提是在FreeRTOSConfig.h中将INCLUDE_vTaskSuspend设置为1。

返回值

pdPASS

- 如果从队列成功读取数据，则返回。
- 如果指定了块时间(xTicksToWait不为零)，那么调用任务可能被置于Blocked状态，以等待队列上的数据可用，但数据在块时间到期之前成功地从队列中读取。

errQUEUE

- 如果由于队列已为空而无法从队列读取数据，则返回。、
- 如果指定了块时间(xTicksToWait不为零)，则调用任务将被置于Blocked状态，以等待另一个任务或中断向队列发送数据，但在此之前块时间已过期。

示例

```
struct AMessage{char ucMessageID;char ucData[ 20 ];} xMessage;QueueHandle_t xQueue;/* Task that creates a queue and posts a value. */void vATask( void *pvParameters ){struct AMessage *pxMessage;/* Create a queue capable of containing 10 pointers to AMessage structures.Store the handle to the created queue in the xQueue variable. */xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );if( xQueue == 0 ){/* The queue was not created because there was not enough FreeRTOS heapmemory available to allocate the queues data structures or storage area. */}else{/* ... *//* Send a pointer to a struct AMessage object to the queue referenced bythe xQueue variable. Don't block if the queue is already full (the thirdparameter to xQueueSend() is zero, so not block time is specified). */pxMessage = &xMessage;xQueueSend( xQueue, ( void * ) &pxMessage, 0 );}/* ... Rest of the task code. */for( ;; ){}}/* 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 isnot available immediately. */if( xQueuePeek( xQueue, &( pxRxedMessage ), 10 ) == pdPASS ){/* pxRxedMessage now points to the struct AMessage variable posted byvATask, but the item still remains on the queue. */}}else{/* The queue could not or has not been created. */}/* ... Rest of the task code. */for( ;; ){}}
```

3.15 xQueuePeekFromISR()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueuePeekFromISR( QueueHandle_t xQueue, void *pvBuffer );
```

概要

xQueuePeek()的一个版本，可以从中断服务例程(ISR)中使用。

从队列中读取项，但不从队列中删除项。下一次使用xQueueReceive()或xQueuePeek()从相同队列中获取项时，将返回相同的项。

参数

xQueue

- 要从其中读取数据的队列的句柄。

pvBuffer

- 从队列中读取的数据将被复制到其中的内存指针。
- 缓冲区的长度必须至少等于队列项的大小。项目大小将由用于创建队列的xQueueCreate()或xQueueCreateStatic()调用的uxItemSize参数设置。

返回值

pdPASS

- 如果从队列成功读取数据，则返回。

errQUEUE_EMPTY

- 如果由于队列已为空而无法从队列读取数据，则返回。

3.16 xQueueReceive()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueReceive( QueueHandle_t
xQueue,                         void *pvBuffer,                         TickType_t
xTicksToWait );
```

概要

从队列中接收(读取)一个项目。

参数

xQueue

- 队列句柄将从用于创建队列的xqueueccreate()或xQueueCreateStatic()调用中返回。

pvBuffer

- 缓冲区的长度必须至少等于队列项的大小。
- 项目大小将通过调用xqueueccreate()或用于创建队列的xQueueCreateStatic()的uxItemSize参数设置。

xTicksToWait

- 如果xTicksToWait为零，那么如果队列已经为空，则xqueuerreceive()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。
- 可以使用pdMS_TO_TICKS()宏将以毫秒为单位指定的时间转换为以ticks为单位指定的时间。
- 如果在FreeRTOSConfig.h中将INCLUDE_vTaskSuspend设置为1，将xTicksToWait设置为portMAX_DELAY会导致任务无限期地等待(没有超时)。

返回值

pdPASS

- 如果成功从队列中读取数据，则返回。
- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务可能被放置到Blocked状态，以等待队列上的数据可用，但在块时间过期之前成功地从队列中读取了数据。

errQUEUE_EMPTY

- 如果由于队列已为空而无法从队列中读取数据，则返回。

- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务将被置于Blocked状态，等待另一个任务或中断向队列发送数据，但在此之前块时间已经过期。

示例

```
/* Define the data type that will be queued. */typedef struct A_Message{char ucMessageID;char ucData[ 20 ];} AMessage;/* Define the queue parameters. */#define QUEUE_LENGTH 5#define QUEUE_ITEM_SIZE sizeof( AMessage )int main( void ) {QueueHandle_t xQueue;/* Create the queue, storing the returned handle in the xQueue variable. */xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );if( xQueue == NULL ){/* The queue could not be created – do something. */}/* Create a task, passing in the queue handle as the task parameter. */xTaskCreate( vAnotherTask,“Task”,STACK_SIZE,( void * ) xQueue, /* The queue handle is used as the task parameter. */TASK_PRIORITY,NULL );/* Start the task executing. */vTaskStartScheduler();/* Execution will only reach here if there was not enough FreeRTOS heap memoryremaining for the idle task to be created. */for( ;; );}void vAnotherTask( void *pvParameters ){QueueHandle_t xQueue;AMessage xMessage;/* The queue handle is passed into this task as the task parameter. Cast thevoid * parameter back to a queue handle. */xQueue = ( QueueHandle_t ) pvParameters;for( ;; ){/* Wait for the maximum period for data to become available on the queue.The period will be indefinite if INCLUDE_vTaskSuspend is set to 1 inFreeRTOSConfig.h. */if( xQueueReceive( xQueue, &xMessage, portMAX_DELAY ) != pdPASS ){/* Nothing was received from the queue – even after blocking to waitfor data to arrive. */}else{/* xMessage now contains the received data. */}}}
```

3.17 xQueueReceiveFromISR()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueReceiveFromISR( QueueHandle_t xQueue,                                    void *pvBuffer,
BaseType_t *pxHigherPriorityTaskWoken );
```

概要

可以从ISR调用的xqueuerreceive()版本。与xqueuerreceive()不同，xQueueReceiveFromISR()不允许指定块时间。

参数

xQueue

- 接收(读取)数据的队列的句柄。队列句柄将从用于创建队列的xqueueccreate()或xQueueCreateStatic()调用中返回。

pvBuffer

- 一个指向内存的指针，接收的数据将被复制到其中。
- 缓冲区的长度必须至少等于队列项的大小。项目大小将通过调用xqueueccreate()或用于创建队列的xQueueCreateStatic()的uxItemSize参数设置。

pxHigherPriorityTaskWoken

- 单个队列上可能有一个或多个任务阻塞，等待队列上的空间可用。调用xQueueReceiveFromISR()可以使空间可用，从而导致这样的任务离开阻塞状态。如果调用API函数导致一个任务离开阻塞状态，并且未阻塞的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)，那么，在内部，API函数将* pxhigherprioritytask叫醒设置为pdTRUE。
- 如果xQueueReceiveFromISR()将此值设置为pdTRUE，那么应该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FreeRTOS V7.3.0中，pxhigherprioritytaskawake是一个可选参数，可以设置为NULL。

返回值

pdPASS

- 成功从队列接收数据。

pdFAIL

- 没有从队列接收数据，因为队列已经为空。

提示

在中断服务例程中调用xQueueReceiveFromISR()可能会导致队列上被阻塞的任务离开阻塞状态。如果这样一个未阻塞的任务的优先级高于或等于当前正在执行的任务(被中断的任务)，则应该执行上下文切换。上下文切换将确保中断直接返回到优先级最高的就绪状态任务。与xqueuerreceive () API函数不同，xQueueReceiveFromISR()本身不会执行上下文切换。相反，它将指示是否需要进行上下文切换。

在启动调度程序之前，不能调用xQueueReceiveFromISR()。因此，在启动调度程序之前，必须不允许执行调用xQueueReceiveFromISR()的中断。

示例

为了清晰地演示，本节中的示例多次调用xQueueReceiveFromISR()来接收多个小数据项。这是低效的，因此不推荐用于大多数应用程序。更可取的方法是将结构中的多个数据项以单个post的方式发送到队列中，允许只调用xQueueReceiveFromISR()一次。或者，最好将处理推迟到任务级别。

```
/* vISR is an interrupt service routine that empties a queue of values, sending eachto a peripheral. It might be that there are multiple tasks blocked on the queuewaiting for space to write more data to the queue. */void vISR( void ){char cByte;BaseType_t xHigherPriorityTaskWoken;/* No tasks have yet been unblocked. */xHigherPriorityTaskWoken = pdFALSE;/* Loop until the queue is empty.xHigherPriorityTaskWoken will get set to pdTRUE internally withinxQueueReceiveFromISR() if calling xQueueReceiveFromISR()caused a task to leavethe Blocked state, and the unblocked task has a priority equal to or greater thanthe task currently in the Running state (the task this ISR interrupted). */while( xQueueReceiveFromISR( xQueue,&cByte,&xHigherPriorityTaskWoken ) == pdPASS ){/* Write the received byte to the peripheral. */OUTPUT_BYTE( TX_REGISTER_ADDRESS, cByte );}/* Clear the interrupt source. *//* Now the queue is empty and we have cleared the interrupt we can perform acontext switch if one is required (if xHigherPriorityTaskWoken has been set topdTRUE. NOTE: The syntax required to perform a context switch from an ISR variesfrom port to port, and from compiler to compiler. Check the web documentation andexamples for the port being used to find the correct syntax required for yourapplication. */portYIELD_FROM_ISR( xHigherPriorityTaskWoken );}
```

3.18 xQueueRemoveFromSet()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueRemoveFromSet( QueueSetMemberHandle_t xQueueOrSemaphore,
QueueSetHandle_t xQueueSet );
```

概要

从队列集中移除队列或信号量。

只有当队列或信号量为空时，队列或信号量才能从队列集中移除。

参数

xQueueOrSemaphore

- 从队列集中删除的队列或信号量的句柄(转换为QueueSetMemberHandle_t类型)。

xQueueSet

- 包含队列或信号量的队列集的句柄。

返回值

pdPASS

- 队列或信号量已成功从队列集中移除。

pdFAIL

- 队列或信号量没有从队列集中移除，因为队列或信号量不在队列集中，或者队列或信号量不是空的。

提示

要使xQueueRemoveFromSet() API函数可用，在FreeRTOSConfig.h中configUSE_QUEUE_SETS必须设置为1。

示例

本示例假设xQueueSet是一个已经创建的队列集，而xQueue是一个已经创建并添加到xQueueSet的队列。

```
if( xQueueRemoveFromSet( xQueue, xQueueSet ) != pdPASS ){/* Either xQueue was not a member of the xQueueSet set, or xQueue isnot empty and therefore cannot be removed from the set. */}else{/* The queue was successfully removed from the set. */}
```

3.19 xQueueReset()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueReset( QueueHandle_t xQueue );
```

概要

将队列重置为其初始空状态。在重置队列时，队列中包含的任何数据都会被丢弃。

参数

xQueue

- 正在重置的队列的句柄。队列句柄将从用于创建队列的xqueueccreate()或xQueueCreateStatic()调用中返回。

返回值

原始版本的xQueueReset()返回pdPASS或pdFAIL。因为FreeRTOS V7.2.0 xQueueReset()总是返回pdPASS。

3.20 xQueueSelectFromSet()

```
#include “FreeRTOS.h”#include “queue.h”QueueSetMemberHandle_t xQueueSelectFromSet( QueueSetHandle_t xQueueSet,                                                const TickType_t xTicksToWait );
```

概要

xQueueSelectFromSet()从队列集合的成员中选择一个队列或信号量，该队列或信号量要么包含数据(在队列的情况下)，要么可以接受数据(在信号量的情况下)。xQueueSelectFromSet()有效地允许任务在一个队列集中同时阻塞(挂起)所有队列和信号量的读操作。

参数

xQueueSet

- 任务将(可能)阻塞的队列设置。

xTicksToWait

- 调用任务将保持在Blocked状态(其他任务正在执行)，等待队列集的一个成员为成功读取队列或信号量进行操作做好准备的最大时间(以滴数计)。

返回值

NULL

- 在xTicksToWait参数指定的块时间过期之前，集合中包含的队列或信号量不可用。

其他任何值

- 包含在包含数据的队列集中的队列句柄(强制转换为QueueSetMemberHandle_t类型)，或者包含在可用队列集中的信号量句柄(强制转换为QueueSetMemberHandle_t类型)。

提示

为了使xQueueSelectFromSet() API函数可用，必须在FreeRTOSConfig.h中将configUSE_QUEUE_SETS设置为1。

除了使用队列集，还有更简单的替代方法。更多信息请参见FreeRTOS.org网站上的屏蔽多个对象页面。

在包含互斥量的队列集上阻塞不会导致互斥量持有者继承阻塞任务的优先级。

接收(在队列中)或接受(在信号量中)操作不能在队列集合的成员上执行，除非对xQueueSelectFromSet()的调用首先返回了该集合成员的句柄。

示例

请参阅本手册中为xQueueCreateSet()函数提供的示例。

3.21 xQueueSelectFromSetFromISR()

```
#include “FreeRTOS.h”#include “queue.h”QueueSetMemberHandle_t
xQueueSelectFromSetFromISR( QueueSetHandle_t xQueueSet );
```

概要

可以从中断服务例程中使用的xQueueSelectFromSet()的一个版本

参数

xQueueSet

- 正在查询的队列集。不可能在读取时阻塞，因为这个函数被设计为从中断中使用。

返回值

NULL

- 没有可用的队列集成员。

其他任何值

- 包含在包含数据的队列集中的队列句柄(强制转换为QueueSetMemberHandle_t类型)，或者包含在可用队列集中的信号量句柄(强制转换为QueueSetMemberHandle_t类型)。

提示

为了使xQueueSelectFromSetFromISR() API函数可用，在FreeRTOSConfig.h中configUSE_QUEUE_SETS必须设置为1。

示例

```
void vReceiveFromQueueInSetFromISR( void ){ QueueSetMemberHandle_t xActivatedQueue;
unsigned long ulReceived;   /* See if any of the queues in the set contain data. */
xActivatedQueue = xQueueSelectFromSetFromISR( xQueueSet );  if( xActivatedQueue !=
NULL )   {       /* Reading from the queue returned by xQueueSelectFromSetFormISR().
*/      if( xQueueReceiveFromISR( xActivatedQueue, &ulReceived, NULL ) != pdPASS )
{           /* Data should have been available as the handle was returned from
xQueueSelectFromSetFromISR(). */        }   }}
```

3.22 xQueueSend(), xQueueSendToFront(), xQueueSendToBack()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueSend( QueueHandle_t xQueue,
  const void * pvItemToQueue,                     TickType_t xTicksToWait
);BaseType_t xQueueSendToFront( QueueHandle_t xQueue,
const void * pvItemToQueue,                             TickType_t xTicksToWait
);BaseType_t xQueueSendToBack( QueueHandle_t xQueue,                          const
void * pvItemToQueue,                           TickType_t xTicksToWait );
```

概要

将一个项目发送(写入)到队列的前面或后面。

xQueueSend()和xQueueSendToBack()执行相同的操作，因此是等效的。两者都将数据发送到队列的后面。xQueueSend()是最初的版本，现在建议使用xQueueSendToBack()代替它。

参数

xQueue

- 将数据发送(写入)到的队列的句柄。队列句柄将从用于创建队列的xqueueccreate()或xQueueCreateStatic()调用中返回。

pvItemToQueue

- 一个指向要复制到队列中的数据的指针。
- 队列可以容纳的每个条目的大小是在创建队列时设置的，许多字节将从pvItemToQueue复制到队列存储区域。

xTicksToWait

- 当队列已满时，任务保持在Blocked状态以等待队列上的空间可用的最大时间。
- 如果xTicksToWait为零且队列已满，则xQueueSend()、xQueueSendToFront()和xQueueSendToBack()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。可以使用pdMS_TO_TICKS()宏将以毫秒为单位指定的时间转换为以ticks为单位指定的时间。
- 将xTicksToWait设置为portMAX_DELAY将导致任务无限期等待(没有超时)，前提是FreeRTOSConfig.h中INCLUDE_vTaskSuspend设置为1。

返回值

pdPASS

- 如果数据成功发送到队列，则返回。
- 如果一块指定时间(xTicksToWait不是零),然后调用任务可能被进入阻塞状态,等待空间可用队列中的函数返回之前,但数据成功写入队列阻塞时间到期之前。

errQUEUE_FULL

- 如果由于队列已满而无法将数据写入队列，则返回。
- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务将被置于Blocked状态，等待另一个任务或中断在队列中腾出空间，但指定的块时间在此之前已经过期。

示例

```
/* Define the data type that will be queued. */typedef struct A_Messagechar ucMessageID;char ucData[ 20 ];} AMessage;/* Define the queue parameters. */#define QUEUE_LENGTH 5#define QUEUE_ITEM_SIZE sizeof( AMessage )int main( void ) {QueueHandle_t xQueue;/* Create the queue, storing the returned handle in the xQueue variable. */xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );if( xQueue == NULL ){/* The queue could not be created – do something. */}/* Create a task, passing in the queue handle as the task parameter. */xTaskCreate( vAnotherTask,“Task”,STACK_SIZE,( void * ) xQueue, /* xQueue is used as the task parameter. */TASK_PRIORITY,NULL );/* Start the task executing. */vTaskStartScheduler();/* Execution will only reach here if there was not enough FreeRTOS heap memoryremaining for the idle task to be created. */for( ;; );}void vATask( void *pvParameters ){QueueHandle_t xQueue;AMessage xMessage;/* The queue handle is passed into this task as the task parameter. Castthe parameter back to a queue handle. */xQueue = ( QueueHandle_t ) pvParameters;for( ;; ){/* Create a message to send on the queue. */xMessage.ucMessageID = SEND_EXAMPLE;/* Send the message to the queue, waiting for 10 ticks for space to becomeavailable if the queue is already full. */if( xQueueSendToBack( xQueue, &xMessage, 10 ) != pdPASS ){/* Data could not be sent to the queue even after waiting 10 ticks. */}}}
```

3.23 xQueueSendFromISR(), xQueueSendToBackFromISR(), xQueueSendToFrontFromISR()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xQueueSendFromISR( QueueHandle_t xQueue,                    const void *pvItemToQueue,
BaseType_t *pxHigherPriorityTaskWoken );BaseType_t xQueueSendToBackFromISR(
QueueHandle_t xQueue,                                  const void *pvItemToQueue,
BaseType_t *pxHigherPriorityTaskWoken );BaseType_t xQueueSendToFrontFromISR(
QueueHandle_t xQueue,                                    const void *pvItemToQueue,
BaseType_t *pxHigherPriorityTaskWoken );
```

概要

可以从ISR调用的xQueueSend()、xQueueSendToFront()和xQueueSendToBack() API函数的版本。与xQueueSend()、xQueueSendToFront()和xQueueSendToBack()不同，ISR安全版本不允许指定块时间。

xQueueSendFromISR()和xQueueSendToBackFromISR()执行相同的操作，所以是等效的。两者都将数据发送到队列的后面。xQueueSendFromISR()是最初的版本，现在建议使用xQueueSendToBackFromISR()代替它。

参数

xQueue

- 将数据发送(写入)到的队列的句柄。队列句柄将从用于创建队列的xqueueccreate()或xQueueCreateStatic()调用中返回。

pvItemToQueue

- 一个指向要复制到队列中的数据的指针。

- 队列可以容纳的每个条目的大小是在创建队列时设置的，许多字节将从pvItemToQueue复制到队列存储区域。

pxHigherPriorityTaskWoken

- 一个队列可能有一个或多个任务阻塞在上面，等待数据可用。调用xQueueSendFromISR()、xQueueSendToFrontFromISR()或xQueueSendToBackFromISR()可以使数据可用，从而导致这样的任务离开阻塞状态。如果调用API函数导致一个任务离开阻塞状态，并且未阻塞的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)，那么，在内部，API函数将* pxhigherprioritytask叫醒设置为pdTRUE。如果xQueueSendFromISR()， xQueueSendToFrontFromISR()或xQueueSendToBackFromISR()将此值设置为pdTRUE，那么应该在中断退出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FreeRTOS V7.3.0中，pxhigherprioritytaskawake是一个可选参数，可以设置为NULL。

返回值

pdTRUE

- 数据已成功发送到队列。

errQUEUE_FULL

- 无法将数据发送到队列，因为队列已满。

提示

在中断服务例程中调用xQueueSendFromISR()、xQueueSendToBackFromISR()或xQueueSendToFrontFromISR()可能会导致队列上被阻塞的任务离开阻塞状态。如果这样一个未阻塞的任务的优先级高于或等于当前正在执行的任务(被中断的任务)，则应该执行上下文切换。上下文切换将确保中断直接返回到优先级最高的就绪状态任务。与xQueueSend()、xQueueSendToBack()和xQueueSendToFront() API函数不同，xQueueSendFromISR()、xQueueSendToBackFromISR()和xQueueSendToFrontFromISR()本身不会执行上下文切换。相反，它们将指示是否需要进行上下文切换。

xQueueSendFromISR()， xQueueSendToBackFromISR()和xQueueSendToFrontFromISR()在调度程序启动之前不能被调用。因此，在启动调度程序之前，必须不允许调用任何这些函数的中断执行。

示例

为了清晰地演示，下面的示例多次调用xQueueSendToBackFromISR()来发送多个小数据项。这是低效的，因此不推荐使用。更可取的方法包括:

1. 将多个数据项打包到一个结构中，然后使用单个调用xQueueSendToBackFromISR()将整个结构发送到队列。这种方法仅适用于数据项较少的情况。
2. 将数据项写入循环RAM缓冲区，然后使用对xQueueSendToBackFromISR()的单个调用来让任务知道缓冲区中包含多少新数据项。

```
/* vBufferISR() is an interrupt service routine that empties a buffer of values,writing each value to a queue. It might be that there are multiple tasks blockedon the queue waiting for the data. */void vBufferISR( void ){   char cIn; BaseType_t xHigherPriorityTaskWoken;    /* No tasks have yet been unblocked. */ xHigherPriorityTaskWoken = pdFALSE; /* Loop until the buffer is empty. */   do  { /* Obtain a byte from the buffer. */        cIn = INPUT_BYTE( RX_REGISTER_ADDRESS ); /* Write the byte to the queue. xHigherPriorityTaskWoken will get set to pdTRUE if writing to the queue causes a task to leave the Blocked state,        and the task leaving the Blocked state has a priority higher than the       currently executing task (the task that was interrupted). */        xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );   } while( INPUT_BYTE( BUFFER_COUNT ) );  /* Clear the interrupt source here. */  /* Now the buffer is empty, and the interrupt source has been cleared, a context    switch should be performed if xHigherPriorityTaskWoken is equal to pdTRUE.  NOTE: The syntax required to perform a context switch from an ISR varies from   port to port, and from compiler to compiler. Check the web documentation and    examples for the port being used to find the syntax required for your   application. */ portYIELD_FROM_ISR( xHigherPriorityTaskWoken );}
```

3.24 uxQueueSpacesAvailable()

```
#include “FreeRTOS.h”#include “queue.h”UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );
```

概要

返回队列中可用的空闲空间数量。也就是说，在队列满之前可以发送到队列的项数。

参数

xQueue

- 被查询队列的句柄。

返回值

调用uxQueueSpacesAvailable()时正在查询的队列中可用空闲空间的数量。

示例

```
void vAFunction( QueueHandle_t xQueue ){    UBaseType_t uxNumberOfFreeSpaces;   /* How many free spaces are currently available in the queue referenced by the  xQueue handle? */   uxNumberOfFreeSpaces = uxQueueSpacesAvailable( xQueue );}
```

章节4 信号量

4.1 vSemaphoreCreateBinary()

```
#include “FreeRTOS.h”#include “semphr.h”void vSemaphoreCreateBinary(
SemaphoreHandle_t xSemaphore );
```

概要

vSemaphoreCreateBinary()宏保留在源代码中，以确保向后兼容，但不应该在新设计中使用它。请使用xSemaphoreCreateBinary()函数。

创建二进制信号量的宏。在使用信号量之前，必须显式地创建它。

参数

xSemaphore

- 类型为SemaphoreHandle_t的变量，它将存储正在创建的信号量的句柄。

返回值

如果在调用vSemaphoreCreateBinary()之后，xSemaphore等于NULL，那么就不能创建信号量，因为没有足够的堆内存供FreeRTOS分配信号量数据结构。在所有其他情况下，xSemaphore将持有创建的信号量的句柄

提示

二进制信号量和互斥量非常相似，但有一些细微的区别。互斥锁包含优先级继承机制，而二进制信号量没有。这使得二进制信号量成为实现同步(在任务之间或任务与中断之间)的更好选择，而互斥体则成为实现简单互斥的更好选择。

Binary Semaphores - 用于同步的二进制信号量在成功地“获取”(获取)之后不需要被“给予”。任务同步是通过一个任务或中断“给出”信号量，另一个任务“获取”信号量来实现的(参见xsemaphoregvefromisr()文档)。

Mutexes - 持有互斥锁的任务的优先级将被提高，如果另一个具有更高优先级的任务试图获取相同的互斥锁。已经拥有互斥锁的任务被认为“继承”了试图“获取”相同互斥锁的任务的优先级。当互斥锁被返回时，继承的优先级将被取消(持有互斥锁时继承的优先级更高的任务将在互斥锁被返回时返回其原来的优先级)。

一个获得互斥锁的任务必须将该互斥锁返回——否则其他任务将无法获得相同的互斥锁。本手册的xSemaphoreTake()一节提供了一个互斥对象用来实现互斥的示例。

互斥锁和二进制信号量都是使用具有SemaphoreHandle_t类型的变量引用的，并且可以在任何接受该类型参数的API函数中使用。

使用旧的vSemaphoreCreateBinary()宏(而不是首选的xSemaphoreCreateBinary()函数)创建的互斥锁和二进制信号量，都是这样创建的，即对该信号量或互斥锁的xSemaphoreTake()的第一个调用将通过。注意vSemaphoreCreateBinary()已弃用，不能在新的应用程序中使用。使用xSemaphoreCreateBinary()函数创建的二进制信号量被创建为“空”，因此必须先给出信号量，然后才能通过调用xSemaphoreTake()来获取(获取)信号量。

示例

```
SemaphoreHandle_t xSemaphore;void vATask( void * pvParameters ){    /* Attempt to create a semaphore.   NOTE: New designs should use the xSemaphoreCreateBinary() function, not the vSemaphoreCreateBinary() macro. */  vSemaphoreCreateBinary( xSemaphore );   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 xSemaphore        variable. */    }}
```

4.2 xSemaphoreCreateBinary()

```
#include “FreeRTOS.h”#include “semphr.h”SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

概要

创建一个二进制信号量，并返回一个可以引用该信号量的句柄。

每个二进制信号量都需要少量的RAM来保存信号量的状态。如果使用xSemaphoreCreateBinary()创建二进制信号量，那么所需的RAM将自动从FreeRTOS堆中分配。如果使用xSemaphoreCreateBinaryStatic()创建二进制信号量，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

信号量是在“空”状态下创建的，这意味着必须先给出信号量，然后才能使用xSemaphoreTake()函数获取(获取)信号量。

参数

无

返回值

NULL

- 无法创建信号量，因为没有足够的堆内存供FreeRTOS分配信号量数据结构。

其他任何值

- 成功创建信号量。返回值是一个句柄，创建的信号量可以通过它来引用。

提示

直接到任务通知通常提供比二进制信号量更轻、更快的替代方法。

二进制信号量和互斥量非常相似，但有一些细微的区别。互斥锁包含优先级继承机制，而二进制信号量没有。这使得二进制信号量成为实现同步(任务之间或中断与任务之间)的更好选择，而互斥是实现简单互斥的更好选择。

Binary Semaphores（二进制信号量）——用于同步的二进制信号量在被成功“获取”(获得)后不需要“返回”。任务同步是通过一个任务或中断“给出”信号量，另一个任务“获取”信号量来实现的(参见xsemaphoregvefromisr()文档)。注意，使用直接到任务通知通常可以更有效地实现相同的功能。

Mutexes（互斥锁）——持有互斥锁的任务的优先级将被提高，如果另一个优先级更高的任务试图获取相同的互斥锁。已经拥有互斥锁的任务被认为“继承”了试图“获取”相同互斥锁的任务的优先级。当互斥锁被返回时，继承的优先级将被取消(持有互斥锁时继承的优先级更高的任务将在互斥锁被返回时返回其原来的优先级)。

一个获得互斥锁的任务必须将该互斥锁返回——否则其他任务将无法获得相同的互斥锁。本手册的xSemaphoreTake()一节提供了一个互斥对象用来实现互斥的示例。

互斥锁和二进制信号量都是使用具有SemaphoreHandle_t类型的变量引用的，并且可以在任何接受该类型参数的API函数中使用。

使用vSemaphoreCreateBinary()宏(而不是首选的xSemaphoreCreateBinary()函数)创建的互斥锁和二进制信号量都是这样创建的，以便通过对该信号量或互斥锁的xSemaphoreTake()的第一个调用。注意vSemaphoreCreateBinary()已弃用，不能在新的应用程序中使用。使用xSemaphoreCreateBinary()函数创建的二进制信号量被创建为“空”，因此必须先给出信号量，然后才能通过调用xSemaphoreTake()来获取(获取)信号量。

configSUPPORT_DYNAMIC_ALLOCATION必须在FreeRTOSConfig.h中设置为1，或者不设置为未定义，以使该函数可用。

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 xSemaphore        variable. Calling xSemaphoreTake() on the semaphore here will fail until        the semaphore has first been given. */  }}
```

4.3 xSemaphoreCreateBinaryStatic()

```
#include “FreeRTOS.h”#include “semphr.h”SemaphoreHandle_t xSemaphoreCreateBinaryStatic( StaticSemaphore_t *pxSemaphoreBuffer );
```

概要

创建一个二进制信号量，并返回一个可以引用该信号量的句柄。

每个二进制信号量都需要少量的RAM来保存信号量的状态。如果使用xSemaphoreCreateBinary()创建二进制信号量，那么所需的RAM将自动从FreeRTOS堆中分配。如果使用xSemaphoreCreateBinaryStatic()创建二进制信号量，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

信号量是在“空”状态下创建的，这意味着必须先给出信号量，然后才能使用xSemaphoreTake()函数获取(获取)信号量。

参数

pxSemaphoreBuffer

- 必须指向类型为StaticSemaphore_t的变量，该变量将用于保存信号量的状态。

返回值

NULL

- 无法创建信号量，因为pxSemaphoreBuffer为NULL。

其他任何值

- 成功创建信号量。返回值是一个句柄，创建的信号量可以通过它来引用。

提示

直接到任务通知通常提供比二进制信号量更轻、更快的替代方法。

二进制信号量和互斥量非常相似，但有一些细微的区别。互斥锁包含优先级继承机制，而二进制信号量没有。这使得二进制信号量成为实现同步(任务之间或中断与任务之间)的更好选择，而互斥是实现简单互斥的更好选择。

二进制信号量——用于同步的二进制信号量在被成功“获取”(获得)后不需要“返回”。任务同步是通过一个任务或中断“给出”信号量，另一个任务“获取”信号量来实现的(参见xsemaphoregvefromisr()文档)。注意，使用直接到任务通知通常可以更有效地实现相同的功能。

互斥锁——持有互斥锁的任务的优先级将被提高，如果另一个优先级更高的任务试图获取相同的互斥锁。已经拥有互斥锁的任务被认为“继承”了试图“获取”相同互斥锁的任务的优先级。当互斥锁被返回时，继承的优先级将被取消(持有互斥锁时继承的优先级更高的任务将在互斥锁被返回时返回其原来的优先级)。

一个获得互斥锁的任务必须将该互斥锁返回——否则其他任务将无法获得相同的互斥锁。本手册的xSemaphoreTake()一节提供了一个互斥对象用来实现互斥的示例。

互斥锁和二进制信号量都是使用具有SemaphoreHandle_t类型的变量引用的，并且可以在任何接受该类型参数的API函数中使用。

使用vSemaphoreCreateBinary()宏(而不是首选的xSemaphoreCreateBinary()函数)创建的互斥锁和二进制信号量都是这样创建的，以便通过对该信号量或互斥锁的xSemaphoreTake()的第一个调用。注意vSemaphoreCreateBinary()已弃用，不能在新的应用程序中使用。使用xSemaphoreCreateBinary()函数创建的二进制信号量被创建为“空”，因此

在使用xSemaphoreTake()调用获取(获取)信号量之前，必须先给出信号量。

示例

```
SemaphoreHandle_t xSemaphoreHandle;StaticSemaphore_t xSemaphoreBuffer;void vATask( void * pvParameters ){/* Create a binary semaphore without using any dynamic memory allocation. */xSemaphoreHandle = xSemaphoreCreateBinaryStatic( &xSemaphoreBuffer );/* pxSemaphoreBuffer was not NULL so the binary semaphore will have been created,and xSemaphoreHandle will be a valid handle.The rest of the task code goes here. */}
```

4.4 xSemaphoreCreateCounting()

```
#include “FreeRTOS.h”#include “semphr.h”SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount,                                        UBaseType_t uxInitialCount );
```

概要

创建一个计数信号量，并返回一个句柄，通过该句柄可以引用该信号量。

每个计数信号量都需要少量的RAM来保存信号量的状态。如果使用xSemaphoreCreateCounting()创建计数信号量，那么将自动从FreeRTOS堆中分配所需的RAM。如果使用xSemaphoreCreateCountingStatic()创建计数信号量，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

参数

uxMaxCount

- 可以达到的最大计数值。当信号量达到这个值时，它就不再被“给定”了。

uxInitialCount

- 创建信号量时分配给它的计数值。

返回值

NULL

- 如果因为没有足够的堆内存供FreeRTOS分配信号量数据结构而不能创建信号量时返回。

其他任何值

- 成功创建信号量。返回值是一个句柄，创建的信号量可以通过它来引用。

提示

直接到任务通知通常提供了比计数信号量更轻、更快的替代方法。

计数信号量通常用于两种情况:

1. 计算事件

   在这个使用场景中，事件处理程序将在每次事件发生时'给出'信号量，而处理程序任务将在每次处理事件时'获取'信号量。

信号量的计数值将在每次“给定”时递增，在每次“获取”时递减。因此，计数值就是已经发生的事件数和已经处理的事件数之间的差值。

为计数事件而创建的信号量的初始计数值应该为0，因为在创建信号量之前没有事件被计数。

2. 资源管理

在这个使用场景中，信号量的计数值表示可用资源的数量。

为了获得对资源的控制，任务必须首先成功地“获取”信号量。“获取”信号量的操作将减少信号量的计数值。当count值达到0时，没有更多可用的资源，进一步尝试“获取”信号量将失败。

当一个任务用一个资源完成时，它必须“给出”信号量。“给出”信号量的操作将增加信号量的计数值，表明一个资源是可用的，并允许未来“获取”信号量的尝试成功。

创建用于管理资源的信号量的初始计数值应该等于可用资源的数量。

configSUPPORT_DYNAMIC_ALLOCATION必须在FreeRTOSConfig.h中设置为1，或者不设置为未定义，以使该函数可用。

示例

```
void vATask( void * pvParameters ){ SemaphoreHandle_t xSemaphore;   /* The semaphore cannot be used before it is created using a call to    xSemaphoreCreateCounting(). The maximum value to which the semaphore can    count in this example case is set to 10, and the initial value assigned to  the count is set to 0. */   xSemaphore = xSemaphoreCreateCounting( 10, 0 ); if( xSemaphore != NULL )    {       /* The semaphore was created successfully. The semaphore can now be used. */    }}
```

4.5 xSemaphoreCreateCountingStatic()

```
#include “FreeRTOS.h”#include “semphr.h”SemaphoreHandle_t xSemaphoreCreateCountingStatic( UBaseType_t uxMaxCount, UBaseType_t uxInitialCount, StaticSemaphore_t pxSempahoreBuffer );
```

概要

创建一个计数信号量，并返回一个句柄，通过该句柄可以引用该信号量。

每个计数信号量都需要少量的RAM来保存信号量的状态。如果使用xSemaphoreCreateCounting()创建计数信号量，那么将自动从FreeRTOS堆中分配所需的RAM。如果使用xSemaphoreCreateCountingStatic()创建计数信号量，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

参数

uxMaxCount

- 可以达到的最大计数值。当信号量达到这个值时，它就不再被“给定”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，该变量将用于保存信号量的状态。

返回值

NULL

- 无法创建信号量，因为pxSemaphoreBuffer为NULL。

其他任何值

- 成功创建信号量。返回值是一个句柄，创建的信号量可以通过它来引用。

提示

与4.4一致

示例

```
void vATask( void * pvParameters ){ SemaphoreHandle_t xSemaphoreHandle;
StaticSemaphore_t xSemaphoreBuffer; /* Create a counting semaphore without using
dynamic memory allocation. The maximum value to which the semaphore can count in
this example case is set to   10, and the initial value assigned to the count is set
to 0. */ xSemaphoreHandle = xSemaphoreCreateCountingStatic( 10, 0, &xSemaphoreBuffer
);  /* The pxSemaphoreBuffer parameter was not NULL, so the semaphore will have been
created and is now ready for use. */}
```

4.6 xSemaphoreCreateMutex()

```
#include “FreeRTOS.h”#include “semphr.h”SemaphoreHandle_t xSemaphoreCreateMutex(
void );
```

概要

创建一个互斥量类型信号量，并返回一个可以引用该互斥量的句柄。

每个互斥量类型的信号量都需要少量的RAM来保存信号量的状态。如果使用xSemaphoreCreateMutex()创建互斥锁，则会自动从FreeRTOS堆中分配所需的RAM。如果使用xSemaphoreCreateMutexStatic()创建一个互斥锁，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

参数

无

返回值

NULL

- 如果因为没有足够的堆内存供FreeRTOS分配信号量数据结构而不能创建信号量时返回。

其他任何值

- 成功创建信号量。返回值是一个句柄，创建的信号量可以通过它来引用。

提示

二进制信号量和互斥量非常相似，但有一些细微的区别。互斥锁包含优先级继承机制，而二进制信号量没有。这使得二进制信号量成为实现同步(在任务之间或任务与中断之间)的更好选择，而互斥体则成为实现简单互斥的更好选择。

二进制信号量——用于同步的二进制信号量在被成功“获取”(获得)后不需要“返回”。任务同步是通过一个任务或中断“给出”信号量，另一个任务“获取”信号量来实现的(参见xsemaphoregvefromisr()文档)。

互斥锁——持有互斥锁的任务的优先级将被提高，如果另一个优先级更高的任务试图获取相同的互斥锁。已经拥有互斥锁的任务被认为“继承”了试图“获取”相同互斥锁的任务的优先级。当互斥锁被返回时，继承的优先级将被取消(持有互斥锁时继承的优先级更高的任务将在互斥锁被返回时返回其原来的优先级)。

一个获得互斥锁的任务必须将该互斥锁返回——否则其他任务将无法获得相同的互斥锁。本手册的xSemaphoreTake()一节提供了一个互斥对象用来实现互斥的示例。

互斥锁和二进制信号量都是使用具有SemaphoreHandle_t类型的变量引用的，并且可以在任何接受该类型参数的API函数中使用。

在FreeRTOSConfig.h中，configSUPPORT_DYNAMIC_ALLOCATION必须设置为1，或者保持未定义，这样这个函数才可用。

示例

```
SemaphoreHandle_t xSemaphore;void vATask( void * pvParameters ){    /* Attempt to create a mutex type semaphore. */ xSemaphore = xSemaphoreCreateMutex();   if( xSemaphore == NULL )    {       /* There was insufficient heap memory available for the mutex to be     created. */ }   else    {       /* The mutex can now be used. The handle of the created mutex will be       stored in the xSemaphore variable. */ }}
```

4.7 xSemaphoreCreateMutexStatic()

```
#include “FreeRTOS.h”#include “semphr.h”SemaphoreHandle_t xSemaphoreCreateMutexStatic( StaticSemaphore_t *pxMutexBuffer );
```

概要

创建一个互斥量类型信号量，并返回一个可以引用该互斥量的句柄。

每个互斥量类型的信号量都需要少量的RAM来保存信号量的状态。如果使用xSemaphoreCreateMutex()创建互斥锁，则会自动从FreeRTOS堆中分配所需的RAM。如果使用xSemaphoreCreateMutexStatic()创建一个互斥锁，那么应用程序编写器将提供RAM

参数

pxMutexBuffer

- 必须指向类型为StaticSemaphore_t的变量，该变量将用于保存互斥锁的状态。

返回值

NULL

- 如果因为没有足够的堆内存供FreeRTOS分配信号量数据结构而不能创建信号量时返回。

其他任何值

- 成功创建信号量。返回值是一个句柄，创建的信号量可以通过它来引用。

提示

与4.6一致

示例

```
SemaphoreHandle_t xSemaphoreHandle;StaticSemaphore_t xSemaphoreBuffer;void vATask( void * pvParameters ){   /* Create a mutex without using any dynamic memory allocation. */   xSemaphoreHandle = xSemaphoreCreateMutexStatic( &xSemaphoreBuffer );    /* The pxMutexBuffer parameter was not NULL so the mutex will have been created and is now ready for use. */}
```

4.8 xSemaphoreCreateRecursiveMutex()

```
#include “FreeRTOS.h”#include “semphr.h”SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( void );
```

概要

创建递归互斥量类型信号量，并返回一个句柄，通过该句柄可以引用递归互斥量。

每个递归互斥锁都需要少量的RAM来保存互斥锁的状态。如果使用xSemaphoreCreateRecursiveMutex()创建递归互斥锁，则会自动从FreeRTOS堆中分配所需的RAM。如果使用xSemaphoreCreateRecursiveMutexStatic()创建递归互斥锁，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

参数

无

返回值

NULL

- 如果因为没有足够的堆内存供FreeRTOS分配信号量数据结构而不能创建信号量时返回。

其他任何值

- 成功创建信号量。返回值是一个句柄，创建的信号量可以通过它来引用。

提示

configUSE_RECURSIVE_MUTEXES必须在FreeRTOSConfig.h中设置为1，以便xSemaphoreCreateRecursiveMutex() API函数可用。

使用xSemaphoreTakeRecursive()函数“获取”递归互斥，使用xSemaphoreGiveRecursive()函数“给出”递归互斥。xSemaphoreTake()和xSemaphoreGive()函数不能与递归互斥体一起使用。

对xSemaphoreTakeRecursive()的调用可以嵌套。因此，一旦一个任务成功地“获取”了一个递归互斥锁，同样的任务对xSemaphoreTakeRecursive()的进一步调用也会成功。必须对xsemaphoregverecursive()进行与之前对xSemaphoreTakeRecursive()进行相同数量的调用，才能让互斥对象对任何其他任务可用。例如，如果一个任务成功地递归地“获取”了相同的互斥锁5次，那么这个互斥锁将不能被其他任何任务使用，直到成功获取互斥锁的任务同时将互斥锁“返回”5次。

与标准互斥锁一样，递归互斥锁在任何时候都只能被单个任务持有或获取。

持有递归互斥锁的任务，如果另一个具有更高优先级的任务试图获取相同的互斥锁，则该任务的优先级将被提高。已经拥有递归互斥锁的任务被称为“继承”试图“获取”相同互斥锁的任务的优先级。当互斥锁被返回时，继承的优先级将被取消(持有互斥锁时继承的优先级更高的任务将在互斥锁被返回时返回其原来的优先级)。

在FreeRTOSConfig.h中，configSUPPORT_DYNAMIC_ALLOCATION必须设置为1，或者保持未定义，这样这个函数才可用。

示例

```
void vATask( void * pvParameters ){SemaphoreHandle_t xSemaphore;/* Recursive
semaphores cannot be used before being explicitly created using acall to
xSemaphoreCreateRecursiveMutex(). */xSemaphore =
xSemaphoreCreateRecursiveMutex();if( xSemaphore != NULL ){/* The recursive mutex
semaphore was created successfully and its handlewill be stored in xSemaphore
variable. The recursive mutex can now beused. */}}
```

4.9 xSemaphoreCreateRecursiveMutexStatic()

```
#include “FreeRTOS.h”#include “semphr.h”SemaphoreHandle_t
xSemaphoreCreateRecursiveMutex( StaticSemaphore_t pxMutexBuffer );
```

概要

创建递归互斥量类型信号量，并返回一个句柄，通过该句柄可以引用递归互斥量。

每个递归互斥锁都需要少量的RAM来保存互斥锁的状态。如果使用xSemaphoreCreateRecursiveMutex()创建递归互斥锁，则会自动从FreeRTOS堆中分配所需的RAM。如果使用xSemaphoreCreateRecursiveMutexStatic()创建递归互斥锁，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

pxMutexBuffer

- 必须指向类型为StaticSemaphore_t的变量，该变量将用于保存互斥锁的状态。

返回值

NULL

- 如果因为没有足够的堆内存供FreeRTOS分配信号量数据结构而不能创建信号量时返回。

其他任何值

- 成功创建信号量。返回值是一个句柄，创建的信号量可以通过它来引用。

提示

与4.8一致

示例

```
void vATask( void * pvParameters ){SemaphoreHandle_t
xSemaphoreHandle;StaticSemaphore_t xSemaphoreBuffer;/* Create a recursive mutex
without using any dynamic memory allocation. */xSemaphoreHandle =
xSemaphoreCreateRecursiveMutexStatic( &xSemaphoreBuffer );/* The pxMutexBuffer
parameter was not NULL so the recursive mutex will havebeen created and is now ready
for use. */}
```

4.10 vSemaphoreDelete()

```
#include “FreeRTOS.h”#include “semphr.h”void vSemaphoreDelete( SemaphoreHandle_t
xSemaphore );
```

概要

删除先前使用vSemaphoreCreateBinary()、xSemaphoreCreateCounting()、xSemaphoreCreateRecursiveMutex()或xSemaphoreCreateMutex()创建的信号量。

参数

xSemaphore

- 正在删除的信号量的句柄。

返回值

无

提示

如果任务试图获取一个不可用的信号量，它们可以选择阻塞一个信号量(有一个可选的超时)。如果一个信号量上有任何当前阻塞的任务，那么这个信号量不能被删除。

4.11 uxSemaphoreGetCount()

```
#include “FreeRTOS.h”
#include “semphr.h”
UBaseType_t uxSemaphoreGetCount( SemaphoreHandle_t xSemaphore );
```

概要

返回信号量的计数。

二进制信号量的计数只能为0或1。计数信号量的计数可以在0和创建计数信号量时指定的最大计数之间。

参数

xSemaphore

- 正在查询的信号量的句柄。

返回值

在xSemaphore参数中传递的句柄所引用的信号量的计数。

4.12 xSemaphoreGetMutexHolder()

```
#include “FreeRTOS.h”#include “semphr.h”TaskHandle_t xSemaphoreGetMutexHolder(
SemaphoreHandle_t xMutex );
```

概要

返回包含由函数参数指定的互斥锁的任务句柄(如果有)。

参数

xMutex

- 正在查询的互斥锁句柄。

返回值

NULL(二选一)

- 由xMutex参数指定的信号量不是互斥量类型的信号量
- 信号量是可用的，并且不被任何任务占用。

其他任何值

- 保存由xMutex参数指定的信号量的任务句柄。

提示

xSemaphoreGetMutexHolder()可以可靠地用于确定调用任务是否是互斥锁的持有者，但如果互斥锁由调用任务以外的任何任务持有，则不能可靠地使用它。这是因为互斥锁保持器可能在调用任务调用函数和调用任务测试函数返回值之间发生变化。

为了xSemaphoreGetMutexHolder()可用，在FreeRTOSConfig.h中configUSE_MUTEXES和INCLUDE_xSemaphoreGetMutexHolder必须都设置为1。

4.13 xSemaphoreGive()

```
#include “FreeRTOS.h”#include “semphr.h”BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

概要

给出(或释放)一个先前通过调用vSemaphoreCreateBinary()、xSemaphoreCreateCounting()或xSemaphoreCreateMutex()创建的信号量，并且已经成功“获取”。

参数

xSemaphore

- 信号量被“给定”。信号量由类型为SemaphoreHandle_t的变量引用，在使用之前必须显式地创建。

返回值

pdPASS

- 信号量' give '操作成功。

pdFAIL

- 信号量' give '操作没有成功，因为调用xsemaphoregve()的任务不是信号量持有者。一个任务必须成功地“获取”一个信号量，然后才能成功地“返回”它。

示例

```
SemaphoreHandle_t xSemaphore = NULL;void vATask( void * pvParameters ){/* A
semaphore is going to be used to guard a shared resource. In this case amutex type
semaphore is created because it includes priority inheritancefunctionality.
*/xSemaphore = xSemaphoreCreateMutex();for( ;; ){if( xSemaphore != NULL ){if(
xSemaphoreGive( xSemaphore ) != pdTRUE ){/* This call should fail because the
semaphore has not yet been‘taken’. */}/* Obtain the semaphore – don’t block if the
semaphore is notimmediately available (the specified block time is zero). */if(
xSemaphoreTake( xSemaphore, 0 ) == pdPASS ){/* The semaphore was ‘taken’
successfully, so the resource it isguarding can be accessed safely. *//* ... *//*
Access to the resource the semaphore is guarding is complete, sothe semaphore must
be ‘given’ back. */if( xSemaphoreGive( xSemaphore ) != pdPASS ){/* This call should
not fail because the calling task hasalready successfully ‘taken’ the semaphore.
*/}}}else{/* The semaphore was not created successfully because there is notenough
FreeRTOS heap remaining for the semaphore data structures to beallocated. */}}}
```

4.14 xSemaphoreGiveFromISR()

```
#include “FreeRTOS.h”#include “semphr.h”BaseType_t xSemaphoreGiveFromISR(
SemaphoreHandle_t xSemaphore,BaseType_t *pxHigherPriorityTaskWoken );
```

概要

可以在ISR中使用的xSemaphoreGive()版本。与xSemaphoreGive()不同，xSemaphoreGiveFromISR()不允许指定块时间。

参数

xSemaphore

- 信号量被“给定”。
- 信号量由类型为SemaphoreHandle_t的变量引用，在使用之前必须显式地创建。

*pxHigherPriorityTaskWoken

- 一个信号量可能会有一个或多个任务阻塞在其上，等待该信号量变得可用。调用xsemaphoregvefromisr()可以使信号量可用，从而导致这样的任务离开Blocked状态。如果调用xSemaphoreGiveFromISR()导致任务离开Blocked状态，并且未阻塞的任务的优先级高于或等于当前正在执行的任务(被中断的任务)，那么，在内部，xSemaphoreGiveFromISR()将*pxHigherPriorityTaskWoken设置为pdTRUE。
- 如果xSemaphoreGiveFromISR()将这个值设置为pdTRUE，那么应该在中断退出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FreeRTOS V7.3.0中，pxhigherprioritytaskawake是一个可选参数，可以设置为NULL。

返回值

pdTRUE

- 对xsemaphoregvefromisr()的调用成功。

errQUEUE_FULL

- 如果一个信号量已经可用，则不能给出它，并且xsemaphoregvefromisr()将返回errQUEUE_FULL。

提示

在中断服务例程中调用xsemaphoregvefromisr()可能会导致被阻塞的任务等待接收信号量离开阻塞状态。如果这样一个未阻塞的任务的优先级高于或等于当前正在执行的任务(被中断的任务)，则应该执行上下文切换。上下文切换将确保中断直接返回到最高优先级的就绪状态任务。

与xSemaphoreGive() API函数不同，xSemaphoreGiveFromISR()本身不会执行上下文切换。相反，它将指示是否需要进行上下文切换。

xsemaphoregvefromisr()不能在启动调度程序之前调用。因此，在启动调度程序之前，必须不允许调用xSemaphoreGiveFromISR()的中断执行。

示例

```
#define LONG_TIME 0xffff#define TICKS_TO_WAIT 10SemaphoreHandle_t xSemaphore = NULL;/* Define a task that performs an action each time an interrupt occurs. TheInterrupt processing is deferred to this task. The task is synchronized with theinterrupt using a semaphore. */void vATask( void * pvParameters ){/* It is assumed the semaphore has already been created outside of this task. */for( ;; ){/* Wait for the next event. */if( xSemaphoreTake( xSemaphore, portMAX_DELAY ) == pdTRUE ){/* The event has occurred, process it here. */.../* Processing is complete, return to wait for the next event. */}}}/* An ISR that defers its processing to a task by using a semaphore to indicatewhen events that require processing have occurred. */void vISR( void * pvParameters ){BaseType_t xHigherPriorityTaskWoken = pdFALSE;/* The event has occurred, use the semaphore to unblock the task so the taskcan process the event. */xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );/* Clear the interrupt here. *//* Now the task has been unblocked a context switch should be performed ifxHigherPriorityTaskWoken is equal to pdTRUE. NOTE: The syntax required to performa context switch from an ISR varies from port to port, and from compiler tocompiler. Check the web documentation and examples for the port being used tofind the syntax required for your application. */portYIELD_FROM_ISR( xHigherPriorityTaskWoken );}
```

4.15 xSemaphoreGiveRecursive()

```
#include “FreeRTOS.h”#include “semphr.h”BaseType_t xSemaphoreGiveRecursive( SemaphoreHandle_t xMutex );
```

概要

'给出'(或释放)一个先前使用xSemaphoreCreateRecursiveMutex()创建的递归互斥量类型信号量。

参数

xMutex

- 信号量被“给定”。信号量由类型为SemaphoreHandle_t的变量引用，在使用之前必须显式地创建。

返回值

pdPASS

- 对xSemaphoreGiveRecursive()的调用成功。

pdFAIL

- 对xsemaphoregiverrecursive()的调用失败，因为调用任务不是互斥锁的持有者。

提示

使用xSemaphoreTakeRecursive()函数“获取”递归互斥，使用xSemaphoreGiveRecursive()函数“给出”递归互斥。xSemaphoreTake()和xSemaphoreGive()函数不能与递归互斥体一起使用。

对xSemaphoreTakeRecursive()的调用可以嵌套。因此，一旦一个任务成功地“获取”了一个递归互斥锁，同样的任务对xSemaphoreTakeRecursive()的进一步调用也会成功。必须对xsemaphoregverecursive()进行与之前对xSemaphoreTakeRecursive()进行相同数量的调用，才能让互斥对象对任何其他任务可用。例如，如果一个任务成功地递归地“获取”相同的互斥锁5次，那么该互斥锁将不能被其他任务使用任务，直到成功获得互斥锁的任务也会将互斥锁“返回”五次。

xSemaphoreGiveRecursive()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时(在启动调度程序之前)不能调用它。

xSemaphoreGiveRecursive()不能在临界区段内或调度程序挂起时调用。

示例

```
/* A task that creates a recursive mutex. */void vATask( void * pvParameters ){/*
Recursive mutexes cannot be used before being explicitly created using a callto
xSemaphoreCreateRecursiveMutex(). */xMutex = xSemaphoreCreateRecursiveMutex();/*
Rest of task code goes here. */for( ;; ){}}/* A function (called by a task) that
uses the mutex. */void vAFunction( void ){/* ... Do other things. */if( xMutex !=
NULL ){/* See if the mutex can be obtained. If the mutex is not available wait
10ticks to see if it becomes free. */if( xSemaphoreTakeRecursive( xMutex, 10 ) ==
pdTRUE ){/* The mutex was successfully 'taken'. */.../* For some reason, due to the
nature of the code, further calls toxSemaphoreTakeRecursive() are made on the same
mutex. In real code thesewould not be just sequential calls, as that would serve no
purpose.Instead, the calls are likely to be buried inside a more complex
callstructure, for example in a TCP/IP stack.*/xSemaphoreTakeRecursive( xMutex, (
TickType_t ) 10 );xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );/* The mutex
has now been 'taken' three times, so will not be availableto another task until it
has also been given back three times. Again itis unlikely that real code would have
these calls sequentially, butinstead buried in a more complex call structure. This
is just forillustrative purposes. */xSemaphoreGiveRecursive( xMutex
);xSemaphoreGiveRecursive( xMutex );xSemaphoreGiveRecursive( xMutex );/* Now the
mutex can be taken by other tasks. */}else{/* The mutex was not successfully
'taken'. */}}}
```

4.16 xSemaphoreTake()

```
#include "FreeRTOS.h"#include "semphr.h"BaseType_t xSemaphoreTake( SemaphoreHandle_t
xSemaphore, TickType_t xTicksToWait );
```

概要

'Takse'(或obtains)先前通过调用vSemaphoreCreateBinary()、xSemaphoreCreateCounting()或xSemaphoreCreateMutex()创建的信号量。

参数

xSemaphore

- 信号量正在被"获取"。信号量由类型为SemaphoreHandle_t的变量引用，在使用之前必须显式地创建。

xTicksToWait

- 如果信号量不能立即可用，则任务应该保持在Blocked状态等待信号量可用的最长时间。
- 如果xTicksToWait为零，那么如果信号量不可用，xSemaphoreTake()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。可以使用pdMS_TO_TICKS()宏将以毫秒为单位指定的时间转换为以ticks为单位指定的时间。
- 如果在FreeRTOSConfig.h中将INCLUDE_vTaskSuspend设置为1，将xTicksToWait设置为portMAX_DELAY会导致任务无限期地等待(没有超时)。

返回值

pdPASS

- 仅当对xSemaphoreTake()的调用成功获取信号量时返回。
- 如果指定了一个块时间(xTicksToWait不为零)，那么有可能调用任务被放置到Blocked状态，以便在信号量不能立即可用的情况下等待信号量，但信号量在块时间过期之前已经可用。

pdFAIL

- 如果对xSemaphoreTake()的调用没有成功获得信号量，则返回。
- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务将被放置到Blocked状态，等待信号量可用，但在此之前块时间已经过期。

提示

xSemaphoreTake()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时(在启动调度程序之前)不能调用它。

xSemaphoreTake()不能在关键段内或调度程序挂起时调用。

示例

```
SemaphoreHandle_t xSemaphore = NULL;/* A task that creates a mutex type semaphore.
*/void vATask( void * pvParameters ){/* A semaphore is going to be used to guard a
shared resource. In this casea mutex type semaphore is created because it includes
priority inheritancefunctionality. */xSemaphore = xSemaphoreCreateMutex();/* The
rest of the task code goes here. */for( ;; ){/* ... */}}/* A task that uses the
mutex. */void vAnotherTask( void * pvParameters ){for( ;; ){/* ... Do other things.
*/if( xSemaphore != NULL ){/* See if the mutex can be obtained. If the mutex is not
availablewait 10 ticks to see if it becomes free. */if( xSemaphoreTake( xSemaphore,
10 ) == pdTRUE ){/* The mutex was successfully obtained so the shared resource can
beaccessed safely. *//* ... *//* Access to the shared resource is complete, so the
mutex isreturned. */xSemaphoreGive( xSemaphore );}else{/* The mutex could not be
obtained even after waiting 10 ticks, sothe shared resource cannot be accessed.
*/}}}}
```

4.17 xSemaphoreTakeFromISR()

```
#include “FreeRTOS.h”#include “queue.h”BaseType_t xSemaphoreTakeFromISR(
SemaphoreHandle_t xSemaphore,signed BaseType_t *pxHigherPriorityTaskWoken );
```

概要

可以从ISR调用的xSemaphoreTake()的一个版本。与xSemaphoreTake()不同，xSemaphoreTakeFromISR()不允许指定块时间。

参数

xSemaphore

- 信号量正在被“获取”。信号量由类型为SemaphoreHandle_t的变量引用，在使用之前必须显式地创建。

pxHigherPriorityTaskWoken

- 有可能(虽然不太可能，取决于信号量的类型)一个信号量上有一个或多个阻塞的任务等待信号量。调用xSemaphoreTakeFromISR()将使一个等待信号量释放的被阻塞的任务离开阻塞状态。如果调用API函数导致一个任务离开阻塞状态，并且未阻塞的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)，那么，在内部，API函数将* pxhigherprioritytask叫醒设置为pdTRUE。
- 如果xSemaphoreTakeFromISR()将* pxhigherprioritytaskwork设置为pdTRUE，那么应该在中断退出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()文档进行进一步解释。
- 在FreeRTOS V7.3.0中，pxhigherprioritytaskawake是一个可选参数，可以设置为NULL。

返回值

pdPASS

- 已成功获取(获取)信号量。

pdFAIL

- 由于信号量不可用，所以没有成功获取。

4.18 xSemaphoreTakeRecursive()

```
#include “FreeRTOS.h”#include “semphr.h”BaseType_t xSemaphoreTakeRecursive(
SemaphoreHandle_t xMutex,TickType_t xTicksToWait );
```

概要

获取'(或获取)先前使用xSemaphoreCreateRecursiveMutex()创建的递归互斥量类型信号量。

参数

xMutex

- 信号量正在被“获取”。信号量由类型为SemaphoreHandle_t的变量引用，在使用之前必须显式地创建。

xTicksToWait

- 如果信号量不能立即可用，则任务应该保持在Blocked状态等待信号量可用的最长时间。
- 如果xTicksToWait为零，那么如果信号量不可用，xSemaphoreTakeRecursive()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。可以使用pdMS_TO_TICKS()宏将以毫秒为单位指定的时间转换为以ticks为单位指定的时间。

- 如果在FreeRTOSConfig.h中将INCLUDE_vTaskSuspend设置为1，将xTicksToWait设置为portMAX_DELAY会导致任务无限期地等待(没有超时)。

返回值

pdPASS

- 仅当对xSemaphoreTakeRecursive()的调用成功获得信号量时返回。
- 如果指定了一个块时间(xTicksToWait不为零)，那么有可能调用任务被放置到Blocked状态，以便在信号量不能立即可用的情况下等待信号量，但信号量在块时间过期之前已经可用。

pdFAIL

- 如果对xSemaphoreTakeRecursive()的调用没有成功获得信号量，则返回。
- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务将被放置到Blocked状态，等待信号量可用，但在此之前块时间已经过期。

提示

使用xSemaphoreTakeRecursive()函数“获取”递归互斥，使用xSemaphoreGiveRecursive()函数“给出”递归互斥。xSemaphoreTake()和xSemaphoreGive()函数不能与递归互斥体一起使用。

对xSemaphoreTakeRecursive()的调用可以嵌套。因此，一旦一个任务成功地“获取”了一个递归互斥锁，同样的任务对xSemaphoreTakeRecursive()的进一步调用也会成功。必须对xsemaphoregverecursive()进行与之前对xSemaphoreTakeRecursive()进行相同数量的调用，才能让互斥对象对任何其他任务可用。例如，如果一个任务成功地递归地“获取”了相同的互斥锁5次，那么这个互斥锁将不能被其他任何任务使用，直到成功获取互斥锁的任务同时将互斥锁“返回”5次。

xSemaphoreTakeRecursive()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时(在启动调度程序之前)不能调用它。

xSemaphoreTakeRecursive()不能在关键段内或调度程序挂起时调用。

示例

```
/* A task that creates a recursive mutex. */void vATask( void * pvParameters ){/*
Recursive mutexes cannot be used before being explicitly created using a callto
xSemaphoreCreateRecursiveMutex(). */xMutex = xSemaphoreCreateRecursiveMutex();/*
Rest of task code goes here. */for( ;; ){}}/* A function (called by a task) that
uses the mutex. */void vAFunction( void ){/* ... Do other things. */if( xMutex !=
NULL ){/* See if the mutex can be obtained. If the mutex is not available wait
10ticks to see if it becomes free. */if( xSemaphoreTakeRecursive( xMutex, 10 ) ==
pdTRUE ){/* The mutex was successfully 'taken'. */.../* For some reason, due to the
nature of the code, further calls toxSemaphoreTakeRecursive() are made on the same
mutex. In real code thesewould not be just sequential calls, as that would serve no
purpose.Instead, the calls are likely to be buried inside a more complex
callstructure, for example in a TCP/IP stack.*/xSemaphoreTakeRecursive( xMutex, (
TickType_t ) 10 );xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );/* The mutex
has now been 'taken' three times, so will not be availableto another task until it
has also been given back three times. Again itis unlikely that real code would have
these calls sequentially, butinstead buried in a more complex call structure. This
is just forillustrative purposes. */xSemaphoreGiveRecursive( xMutex
);xSemaphoreGiveRecursive( xMutex );xSemaphoreGiveRecursive( xMutex );/* Now the
mutex can be taken by other tasks. */}else{/* The mutex was not successfully
'taken'. */}}}
```

章节5 软件定时器

5.1 xTimerChangePeriod()

```
#include "FreeRTOS.h"
#include "timers.h"
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer,
                               TickType_t xNewPeriod,
                               TickType_t xTicksToWait );
```

概要

修改定时器的周期。xTimerChangePeriodFormISR()是一个等效的函数，可以从中断服务例程中调用。

如果xTimerChangePeriod()用于更改已经运行的计时器的周期，则计时器将使用新的周期值重新计算其过期时间。重新计算的到期时间将相对于调用xTimerChangePeriod()时，而不是相对于计时器最初启动时。

如果使用xTimerChangePeriod()来更改尚未运行的计时器的周期，则计时器将使用新的period值来计算过期时间，计时器将开始运行。

参数

xTimer

- 新周期被分配给的计时器。

xNewPeriod

- 由xTimer参数引用的定时器的新周期。

- 计时器周期以多个计时周期指定。可以使用pdMS_TO_TICKS()宏将以毫秒为单位的时间转换为以计时为单位的时间。例如，如果计时器必须在100嘀嗒之后过期，那么xNewPeriod可以直接设置为100。或者，如果计时器必须在500ms之后过期，那么可以将xNewPeriod设置为pdMS_TO_TICKS(500)，前提是configTICK_RATE_HZ小于或等于1000。

xTicksToWait

- 计时器功能不是由核心FreeRTOS代码提供的，而是由一个计时器服务(或守护进程)任务提供的。FreeRTOS定时器API通过一个名为定时器命令队列的队列向定时器服务任务发送命令。xTicksToWait指定当计时器命令队列已经满时，任务应保持在Blocked状态以等待空间可用的最大时间量。
- 阻塞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。与xNewPeriod参数一样，可以使用pdMS_TO_TICKS()宏将指定的时间(以毫秒为单位)转换为以ticks为单位的时间。
- 将xTicksToWait设置为portMAX_DELAY将导致任务无限期等待(没有超时)，前提是FreeRTOSConfig.h中INCLUDE_vTaskSuspend设置为1。
- 如果在启动调度程序之前调用xTimerChangePeriod()，则xTicksToWait将被忽略。

返回值

pdPASS

- 日志含义change period命令成功下发到定时器命令队列。
- 如果一块指定时间(xTicksToWait不是零),然后调用任务可能被进入阻塞状态等待空间可用定时器命令队列函数返回之前,但数据成功写入队列阻塞时间到期之前。
- 该命令实际被处理的时间取决于计时器服务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相对于实际调用xTimerChangePeriod()的时间。定时器服务任务的优先级由configTIMER_TASK_PRIORITY配置常量设置。

pdFAIL

- change period命令没有发送到计时器命令队列，因为队列已经满了。
- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务将被置于Blocked状态，等待计时器服务任务在队列中腾出空间，但指定的块时间在此之前已经过期。

提示

为了使xTimerChangePeriod()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

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 ) {   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, pdMS_TO_TICKS( 500 ), 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. */      }   }}
```

5.2 xTimerChangePeriodFromISR()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerChangePeriodFromISR( TimerHandle_t xTimer,                                    TickType_t xNewPeriod,                                    BaseType_t *pxHigherPriorityTaskWoken );
```

概要

可以从中断服务例程调用的xTimerChangePeriod()的一个版本。

参数

xTimer

- 新周期被分配给的计时器。

xNewPeriod

- 由xTimer参数引用的定时器的新周期。
- 计时器周期以多个计时周期指定。可以使用pdMS_TO_TICKS()宏将以毫秒为单位的时间转换为以计时为单位的时间。例如，如果计时器必须在100嘀嗒之后过期，那么xNewPeriod可以直接设置为100。或者，如果计时器必须在500ms之后过期，那么可以将xNewPeriod设置为pdMS_TO_TICKS(500)，前提是configTICK_RATE_HZ小于或等于1000。

pxHigherPriorityTaskWoken

- xTimerChangePeriodFromISR()将命令写入计时器命令队列。如果写定时器命令队列使定时器服务任务离开阻塞状态,和定时器服务任务优先级等于或大于当前执行的任务(任务被打断),然后*pxHigherPriorityTaskWoken将被设置在xTimerChangePeriodFromISR pdTRUE内部()函数。如果xTimerChangePeriodFromISR()将这个值设置为pdTRUE，那么应该在中断退出之前执行上下文切换。

返回值

pdPASS

- change period命令成功下发到定时器命令队列。当命令被实际处理时，将取决于计时器服务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是相对于实际调用xTimerChangePeriodFromISR()的时间。定时器服务任务的优先级由configTIMER_TASK_PRIORITY配置常量设置。

pdFAIL

- change period命令没有发送到计时器命令队列，因为队列已经满了。

提示

为了使xTimerChangePeriodFromISR()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

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, &xHigherPriorityTaskWoken ) != pdPASS )  {       /* The command to change the timer's 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). */ }}
```

5.3 xTimerCreate()

```
#include “FreeRTOS.h”#include “timers.h”TimerHandle_t xTimerCreate( const char *pcTimerName,                            const TickType_t xTimerPeriod,                            const UBaseType_t uxAutoReload,                            void * const pvTimerID,                            TimerCallbackFunction_t pxCallbackFunction );
```

概要

创建一个新的软件计时器，并返回一个句柄，通过该句柄可以引用创建的软件计时器。

每个软件计时器都需要少量的RAM来保存计时器的状态。如果使用xTimerCreate()创建一个软件计时器，那么这个RAM将自动从FreeRTOS堆中分配。如果使用xTimerCreateStatic()创建软件计时器，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

创建定时器不会启动定时器。xTimerStart()、xTimerReset()、xTimerStartFromISR()、xTimerResetFromISR()、xTimerChangePeriod()和xTimerChangePeriodFromISR() API函数都可以用来启动计时器运行。

参数

pcTimerName

- 分配给计时器的纯文本名称，纯粹是为了辅助调试。

xTimerPeriod

- 计时器。
- 计时器周期以多个计时周期指定。可以使用pdMS_TO_TICKS()宏将以毫秒为单位的时间转换为以计时为单位的时间。例如，如果计时器必须在100嘀嗒之后过期，那么xNewPeriod可以直接设置为100。或者，如果计时器必须在500ms之后过期，那么可以将xNewPeriod设置为pdMS_TO_TICKS(500)，前提是configTICK_RATE_HZ小于或等于1000。

uxAutoReload

- 设置为pdTRUE以创建自动重新加载计时器。设置为pdFALSE以创建一次性计时器。
- 一旦启动，自动重新加载计时器将以xTimerPeriod参数设置的频率重复过期。
- 一旦启动，一次性计时器将只过期一次。一次性计时器可以在过期后手动重新启动。

pvTimerID

- 分配给正在创建的计时器的标识符。稍后可以使用vTimerSetTimerID() API函数更新该标识符。
- 如果同一个回调函数被分配给多个计时器，那么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上已经过期。此外，计时器标识符可以用于在计时器的回调函数调用之间存储一个值。

pxCallbackFunction

- 计时器到期时要调用的函数。回调函数必须有TimerCallbackFunction_t类型定义的原型。所需的原型如下所示。

```
void vCallbackFunctionExample( TimerHandle_t xTimer );
```

返回值

NULL

- 无法创建软件计时器，因为没有足够的FreeRTOS堆内存来成功分配计时器数据结构。

其他任何值

- 软件定时器创建成功，返回值为所创建的软件定时器引用的句柄。

提示

为了使xTimerCreate()可用，在FreeRTOSConfig.h中configUSE_TIMERS和configSUPPORT_DYNAMIC_ALLOCATION必须同时设置为1。如果未定义，configSUPPORT_DYNAMIC_ALLOCATION将默认为1。

示例

```
/* 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;   /* The number of times this timer has expired is saved as the timer's ID. Obtain the    count. */ ulCount = ( uint32_t ) pvTimerGetTimerID( 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 >= xMaxExpiryCountBeforeStopping )  {       /* 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( 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( xTimer, ( void * ) ulCount ); }}
```

5.4 xTimerCreateStatic()

```
#include “FreeRTOS.h”#include “timers.h”TimerHandle_t xTimerCreateStatic( const char *pcTimerName,                                  const TickType_t xTimerPeriod,  const UBaseType_t uxAutoReload,                                    void * const pvTimerID,                                  TimerCallbackFunction_t pxCallbackFunction,                                  StaticTimer_t *pxTimerBuffer );
```

概要

创建一个新的软件计时器，并返回一个句柄，可通过该句柄引用创建的软件计时器。

每个软件定时器都需要少量的RAM来保存定时器的状态。如果使用xTimerCreate()创建一个软件计时器，那么这个RAM将自动从FreeRTOS堆中分配。如果使用xTimerCreateStatic()创建软件计时器，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

创建定时器不会启动定时器。xTimerStart()、xTimerReset()、xTimerStartFromISR()、xTimerResetFromISR()、xTimerChangePeriod()和xTimerChangePeriodFromISR() API函数都可以用来启动计时器运行。

参数

pcTimerName

- 分配给计时器的纯文本名称，纯粹是为了辅助调试。

xTimerPeriod

- 计时器。
- 计时器周期以多个计时周期指定。可以使用pdMS_TO_TICKS()宏将以毫秒为单位的时间转换为以计时为单位的时间。例如，如果计时器必须在100嘀嗒之后过期，那么xNewPeriod可以直接设置为100。或者，如果计时器必须在500ms之后过期，那么可以将xNewPeriod设置为pdMS_TO_TICKS(500)，前提是configTICK_RATE_HZ小于或等于1000。

uxAutoReload

- 设置为pdTRUE以创建自动重新加载计时器。设置为pdFALSE以创建一次性计时器。
- 一旦启动，自动重新加载计时器将以xTimerPeriod参数设置的频率重复过期。
- 一旦启动，一次性计时器将只过期一次。一次性计时器可以在过期后手动重新启动。

pvTimerID

- 分配给正在创建的计时器的标识符。稍后可以使用vTimerSetTimerID() API函数更新该标识符。
- 如果同一个回调函数被分配给多个计时器，那么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上已经过期。此外，计时器标识符可以用于在计时器的回调函数调用之间存储一个值。

pxCallbackFunction

- 计时器到期时要调用的函数。回调函数必须有TimerCallbackFunction_t类型定义的原型。所需的原型如下所示。

```
void vCallbackFunctionExample( TimerHandle_t xTimer );
```

pxTimerBuffer

- 必须指向类型为StaticTimer_t的变量，然后使用该变量保存计时器的状态。

返回值

NULL

- 无法创建软件计时器，因为pxTimerBuffer为NULL。

任何其他值

- 软件定时器创建成功，返回值为所创建的软件定时器引用的句柄。

提示

为了使xTimerCreateStatic()可用，在FreeRTOSConfig.h中configUSE_TIMERS和configSUPPORT_STATIC_ALLOCATION必须同时设置为1。

示例

```
/* 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;   /* The number of
times this timer has expired is saved as the timer's ID. Obtain the    count. */
ulCount = ( uint32_t ) pvTimerGetTimerID( 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 >= xMaxExpiryCountBeforeStopping )  {       /* 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( 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(
xTimer, ( void * ) ulCount ); }}
```

5.5 xTimerDelete()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerDelete( TimerHandle_t
xTimer, TickType_t xTicksToWait );
```

概要

删除一个计时器。计时器必须首先使用xTimerCreate() API函数创建。

参数

xTimer

- 正在删除的定时器的句柄。

xTicksToWait

- 计时器功能不是由核心FreeRTOS代码提供的，而是由一个计时器服务(或守护进程)任务提供的。FreeRTOS定时器API通过一个名为定时器命令队列的队列向定时器服务任务发送命令。xTicksToWait指定任务应该保持在Blocked状态，等待计时器命令队列上的空间可用的最大时间，如果队列已经满了。
- 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。可以使用pdMS_TO_TICKS()宏将以毫秒为单位指定的时间转换为以ticks为单位指定的时间。
- 将xTicksToWait设置为portMAX_DELAY将导致任务无限期等待(没有超时)，前提是FreeRTOSConfig.h中INCLUDE_vTaskSuspend设置为1。
- 如果在启动调度程序之前调用xTimerDelete()，则xTicksToWait将被忽略

返回值

pdPASS

- delete命令成功发送到定时器命令队列。如果一块指定时间(xTicksToWait不是零),然后调用任务可能被进入阻塞状态等待空间可用定时器命令队列函数返回之前,但数据成功写入队列阻塞时间到期之前。
- 当命令被实际处理时，将取决于定时器服务任务相对于系统中其他任务的优先级。定时器服务任务的优先级由configTIMER_TASK_PRIORITY配置常量设置。

pdFAIL

- 删除命令没有发送到计时器命令队列，因为队列已经满了。
- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务将被置于Blocked状态，等待计时器服务任务在队列中腾出空间，但指定的块时间在此之前已经过期。

提示

为了使xTimerDelete()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

示例

请参阅提供的xTimerChangePeriod() API函数的示例。

5.6 xTimerGetExpiryTime()

```
#include “FreeRTOS.h”#include “timers.h”TickType_t xTimerGetExpiryTime(
TimerHandle_t xTimer );
```

概要

返回软件计时器将过期的时间，这是软件计时器的回调函数将执行的时间。

参数

xTimer

- 正在查询的定时器的句柄。

返回值

如果xTimer引用的定时器是活动的，那么将返回该定时器的回调函数下一次执行的时间。时间以RTOS节拍指定。

如果xTimer所引用的定时器未激活，则返回值为undefined。可以使用xTimerIsTimerActive() API函数来确定计时器是否处于活动状态。

提示

如果xTimerGetExpiryTime()返回的值小于当前计时计数，那么计时器将不会过期，直到计时计数溢出并包装回0。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，无论它是在tick计数溢出之前还是之后。

为了使xTimerGetExpiryTime()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

示例

```
static void vAFunction( TimerHandle_t xTimer ){ TickType_t xRemainingTime;  /*
Calculate the time that remains before the timer referenced by xTimer        Expires
and executes its callback function.     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();}
```

5.7 pcTimerGetName()

```
#include “FreeRTOS.h”#include “timers.h”const char * pcTimerGetName( TimerHandle_t
xTimer );
```

概要

返回创建计时器时分配给计时器的人类可读文本名称。更多信息请参见xTimerCreate() API函数。

参数

xTimer

- 正在查询的定时器。

返回值

计时器名称是标准的以NULL结尾的C字符串。返回的值是一个指向主题计时器名称的指针。

提示

为了使pcTimerGetName()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

5.8 xTimerGetPeriod()

```
#include “FreeRTOS.h”#include “timers.h”TickType_t xTimerGetPeriod( TimerHandle_t
xTimer );
```

概要

返回软件计时器的周期。周期以RTOS节拍指定。

软件计时器的周期最初由用于创建计时器的xTimerCreate()调用的xTimerPeriod参数指定。随后可以使用xTimerChangePeriod()和xTimerChangePeriodFromISR() API函数更改它。

参数

xTimer

- 正在查询的定时器。

返回值

定时器的周期，以滴答为单位。

提示

为了使xTimerGetPeriod()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

示例

```
/* A callback function assigned to a software timer. */static void prvTimerCallback(
TimerHandle_t xTimer ){    TickType_t xTimerPeriod;    /* Query the period of the
timer that expired. */   xTimerPeriod = xTimerGetPeriod( xTimer );}
```

5.9 xTimerGetTimerDaemonTaskHandle()

```
#include “FreeRTOS.h”#include “timers.h”TaskHandle_t xTimerGetTimerDaemonTaskHandle(
void );
```

概要

返回与软件定时器守护进程(或服务)任务相关联的任务句柄。如果在FreeRTOSConfig.h中将configUSE_TIMERS设置为1，那么计时器守护进程任务将在启动调度程序时自动创建。所有FreeRTOS软件定时器回调函数都运行在定时器守护进程任务的上下文中。

参数

无

返回值

定时器守护进程任务的句柄。FreeRTOS软件定时器回调函数运行在软件守护进程任务的上下文中。

提示

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerGetTimerDaemonTaskHandle() to be available.

5.10 pvTimerGetTimerID()

```
#include “FreeRTOS.h”#include “timers.h”void *pvTimerGetTimerID( TimerHandle_t xTimer );
```

概要

返回分配给计时器的标识符(ID)。一个标识符在创建计时器时被分配给该计时器，并且可以使用vTimerSetTimerID() API函数进行更新。更多信息请参见xTimerCreate() API函数。

如果同一个回调函数被分配给多个计时器，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上已经过期。这在为xTimerCreate() API函数提供的示例代码中进行了演示。

此外，计时器的标识符可以用于存储计时器回调函数调用之间的值。

参数

xTimer

- 正在查询的定时器。

返回值

分配给正在查询的计时器的标识符。

提示

如果pvTimerGetTimerID()可用，则在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

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( pxExpiredTimer ); ulCallCount++;  vTimerSetTimerID( pxExpiredTimer, ( void * ) ulCallCount );}
```

5.11 xTimerIsTimerActive()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerIsTimerActive( TimerHandle_t xTimer );
```

概要

查询计时器以确定计时器是否正在运行。

如果:计时器将不会运行:

1. 计时器已经创建，但尚未启动。
2. 该计时器是一个一次性计时器，它在过期后还没有重新启动。

xTimerStart()、xTimerReset()、xTimerStartFromISR()、xTimerResetFromISR()、xTimerChangePeriod()和xTimerChangePeriodFromISR() API函数都可以用来启动计时器运行。

参数

xTimer

- 正在查询的定时器。

返回值

pdFALSE

- 计时器没有运行。

任何其他值

- 计时器正在运行。

提示

为了使xTimerIsTimerActive()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

示例

```
/* This function assumes xTimer has already been created. */void vAFunction( TimerHandle_t xTimer ){    /* The following line could equivalently be written as: "if( xTimerIsTimerActive( xTimer ) )" */    if( xTimerIsTimerActive( xTimer ) != pdFALSE )  {       /* xTimer is active, do something. */   }   else    {       /* xTimer is not active, do something else. */  }}
```

5.12 xTimerPendFunctionCall()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerPendFunctionCall(
PendedFunction_t xFunctionToPend,                                    void
*pvParameter1,                                uint32_t ulParameter2,
  TickType_t xTicksToWait );
```

概要

用于延迟一个函数的执行到RTOS守护进程任务(也称为定时器服务任务，因此该函数在timers.c中实现，并以' timer '作为前缀)。

这个函数不能从中断服务程序中调用。请参见xTimerPendFunctionCallFromISR()获取可以从中断服务例程调用的版本。

可以延迟到RTOS守护进程任务的函数必须具有下述原型。

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

pvParameter1和ulParameter2参数供应用程序代码使用。

参数

xFunctionToPend

- 要从计时器服务/守护进程任务中执行的函数。该函数必须符合概要所示的PendedFunction_t原型。

pvParameter1

- 作为函数的第一个参数传递给回调函数的值。该形参具有void *类型，允许它用于传递任何类型。例如，整型可以被转换为void *，或者void *可以用来指向一个结构。

ulParameter2

- 作为函数的第二个参数传递给回调函数的值。

xTicksToWait

- 调用xTimerPendFunctionCall()将导致在队列中向计时器守护进程任务(也称为计时器服务任务)发送消息。xTicksToWait指定调用任务在Blocked状态下等待(因此不消耗任何处理时间)的时间量，以便在队列已满时队列上有可用空间。

返回值

pdPASS

- 消息已成功发送给RTOS守护进程任务。

任何其他值

- 消息没有发送到RTOS守护进程任务，因为消息队列已经满了。队列的长度由FreeRTOSConfig.h中的configTIMER_QUEUE_LENGTH的值设置。

提示

INCLUDE_xTimerPendFunctionCall()和configUSE_TIMERS必须同时在FreeRTOSConfig.h中设置为1，以便xTimerPendFunctionCall()可用。

5.13 xTimerPendFunctionCallFromISR()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerPendFunctionCallFromISR(
PendedFunction_t xFunctionToPend,                                    void
*pvParameter1,                                  uint32_t ulParameter2,
BaseType_t *pxHigherPriorityTaskWoken );
```

概要

用于应用程序中断服务程序，将函数的执行延迟到RTOS守护进程任务(也称为定时器服务任务，因此该函数在timers.c中实现，并以' timer '作为前缀)。

理想情况下，中断服务程序(ISR)应该保持尽可能短的时间，但有时ISR要么有很多处理要做，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守护进程任务。

提供了一种机制，允许中断直接返回到随后将执行挂起函数的任务。这允许回调函数在中断时间内连续执行——就像回调函数在中断本身中执行一样。

可以延迟到RTOS守护进程任务的函数必须具有下述原型。

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

pvParameter1和ulParameter2参数供应用程序代码使用。

参数

xFunctionToPend

- 要从计时器服务/守护进程任务中执行的函数。该函数必须符合清单204所示的PendedFunction_t原型。

pvParameter1

- 将作为函数的第一个参数传递给回调函数的值。该形参具有void *类型，允许它用于传递任何类型。例如，整型可以被转换为void *，或者void *可以用来指向一个结构。

ulParameter2

- 将作为函数的第二个参数传递给回调函数的值。

pxHigherPriorityTaskWoken

- 调用xTimerPendFunctionCallFromISR()将导致一条消息在队列中被发送到RTOS计时器守护进程任务。如果守护进程的优先级的任务(这是由configTIMER_TASK_PRIORITY FreeRTOSConfig.h)的价值高于当前运行的任务的优先级(任务中断打断了)然后* pxHigherPriorityTaskWoken将被设置在xTimerPendFunctionCallFromISR pdTRUE (),指示在中断退出之前应该请求一个上下文切换。因此，*

pxhigherprioritytask唤醒必须初始化为pdFALSE。

返回值

pdPASS

- 消息已成功发送给RTOS守护进程任务。

任何其他值

- 消息没有发送到RTOS守护进程任务，因为消息队列已经满了。队列的长度由FreeRTOSConfig.h中的configTIMER_QUEUE_LENGTH的值设置。

提示

INCLUDE_xTimerPendFunctionCall()和configUSE_TIMERS必须同时在FreeRTOSConfig.h中设置为1，以便xTimerPendFunctionCallFromISR()可用。

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 );}
```

5.14 xTimerReset()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

概要

重启一个计时器。xTimerResetFromISR()是一个可以从中断服务例程调用的等价函数。

如果计时器已经在运行，那么计时器将根据调用xTimerReset()的时间重新计算其到期时间。

如果计时器没有运行，那么该计时器将计算一个相对于调用xTimerReset()时的到期时间，然后计时器将开始运行。在本例中，xTimerReset()在功能上等同于xTimerStart()。

重置计时器可确保计时器正在运行。如果在此期间计时器没有停止、删除或重置，则在调用xTimerReset()之后，与计时器关联的回调函数将被调用' n ' ticks，其中' n '是计时器定义的周期。

如果在启动调度器之前调用xTimerReset()，那么定时器在启动调度器之前不会开始运行，并且定时器的到期时间将相对于启动调度器的时间。

参数

xTimer

- 定时器正在重置、启动或重新启动。

xTicksToWait

- 计时器功能不是由核心FreeRTOS代码提供的，而是由计时器服务(或守护进程)任务提供的。FreeRTOS定时器API通过一个名为定时器命令队列的队列向定时器服务任务发送命令。xTicksToWait指定任务应该保持在Blocked状态，等待计时器命令队列上的空间可用的最大时间，如果队列已经满了。
- 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。可以使用pdMS_TO_TICKS()宏将以毫秒为单位指定的时间转换为以ticks为单位指定的时间。
- 将xTicksToWait设置为portMAX_DELAY将导致任务无限期等待(没有超时)，前提是FreeRTOSConfig.h中INCLUDE_vTaskSuspend设置为1。
- 如果在启动调度器之前调用xTimerReset()，则xTicksToWait将被忽略。

返回值

pdPASS

- 成功将reset命令发送到定时器命令队列。
- 如果一块指定时间(xTicksToWait不是零),然后调用任务可能被进入阻塞状态等待空间可用定时器命令队列函数返回之前,但数据成功写入队列阻塞时间到期之前。
- 当命令被实际处理时，将取决于计时器服务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是相对于实际调用xTimerReset()时的。定时器服务任务的优先级由configTIMER_TASK_PRIORITY配置常量设置。

pdFAIL

- reset命令没有发送到定时器命令队列，因为队列已经满了。
- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务将被置于Blocked状态，等待计时器服务任务在队列中腾出空间，但指定的块时间在此之前已经过期。

提示

为了使xTimerReset()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

示例

```
/* In this example, 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 by 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     reset 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 ){   /* 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( "BcklghtTmr" /* Just a text name, not used by the kernel. */                                    pdMS_TO_TICKS( 5000 ), /* The timer period in ticks. */                                pdFALSE, /* It is a one-shot timer. */ 0, /* ID not used by the callback so can take any value. */                                    vBacklightTimerCallback /* The callback function that                                                switches the LCD back-light off. */                                       );  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 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 scheduler will start the timer running as xTimerStart has already       been called. */ xTaskStartScheduler();}
```

5.15 xTimerResetFromISR()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerResetFromISR( TimerHandle_t xTimer,                                BaseType_t *pxHigherPriorityTaskWoken );
```

概要

xTimerReset()的一个版本，可以从中断服务例程中调用。

参数

xTimer

- 正在启动、重置或重新启动的定时器的句柄。

pxHigherPriorityTaskWoken

- xTimerResetFromISR()将命令写入计时器命令队列。如果写定时器命令队列使定时器服务任务离开阻塞状态,和定时器服务任务优先级等于或大于当前执行的任务(任务被打断),然后* pxHigherPriorityTaskWoken将被设置在xTimerResetFromISR pdTRUE内部()函数。如果xTimerResetFromISR()将这个值设置为pdTRUE，那么应该在中断退出之前执行上下文切换。

返回值

pdPASS

- 成功将reset命令发送到定时器命令队列。该命令实际被处理的时间取决于定时器服务任务相对于系统中其他任务的优先级，尽管定时器的到期时间相对于实际调用xTimerResetFromISR()的时间。定时器服务任务的优先级由configTIMER_TASK_PRIORITY配置常量设置。

pdFAIL

- reset命令没有发送到定时器命令队列，因为队列已经满了。

提示

为了xTimerResetFromISR()可用，必须在FreeRTOSConfig.h中将configUSE_TIMERS设置为1。

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 by a one-shot timer. 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   initialized 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). */   }}
```

5.16 vTimerSetTimerID()

```
#include “FreeRTOS.h”#include “timers.h”void vTimerSetTimerID( TimerHandle_t xTimer, void *pvNewID );
```

概要

一个标识符(ID)在创建计时器时被分配给一个计时器，并且可以使用vTimerSetTimerID() API函数在任何时候更改它。

如果同一个回调函数被分配给多个计时器，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上已经过期。

计时器标识符还可以用于在计时器的回调函数调用之间存储数据。

参数

xTimer

- 定时器的句柄被更新为一个新的标识符。

pvNewID

- 定时器标识符要设置的值。

提示

为了xTimerSetTimerID()可用，必须在FreeRTOSConfig.h中将configUSE_TIMERS设置为1。

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( pxExpiredTimer );
ulCallCount++;  vTimerSetTimerID( pxExpiredTimer, ( void * ) ulCallCount );}
```

5.17 xTimerStart()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerStart( TimerHandle_t
xTimer, TickType_t xTicksToWait );
```

概要

启动计时器运行。xTimerStartFromISR()是一个可以从中断服务例程调用的等价函数。

如果计时器尚未运行，那么该计时器将计算相对于调用xTimerStart()时的到期时间。

如果计时器已经运行，则xTimerStart()在功能上等同于xTimerReset()。

如果在此期间没有停止、删除或重置计时器，则在调用xTimerStart()后，与计时器关联的回调函数将被称为“n”ticks，其中“n”是计时器定义的周期。

参数

xTimer

- 要重置、启动或重启的定时器。

xTicksToWait

- 计时器功能不是由核心FreeRTOS代码提供的，而是由计时器服务(或守护进程)任务提供的。FreeRTOS定时器API通过一个名为定时器命令队列的队列向定时器服务任务发送命令。xTicksToWait指定当计时器命令队列已经满时，任务应保持在Blocked状态以等待空间可用的最大时间量。
- 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。可以使用pdMS_TO_TICKS()宏将以毫秒为单位指定的时间转换为以ticks为单位指定的时间。
- 将xTicksToWait设置为portMAX_DELAY将导致任务无限期等待(没有超时)，前提是FreeRTOSConfig.h中INCLUDE_vTaskSuspend设置为1。
- 如果在启动调度程序之前调用xTimerStart()，则xTicksToWait将被忽略。

返回值

pdPASS

- start命令成功发送到定时器命令队列。
- 如果一块指定时间(xTicksToWait不是零),然后调用任务可能被进入阻塞状态等待空间可用定时器命令队列函数返回之前,但数据成功写入队列阻塞时间到期之前。
- 当命令被实际处理时，将取决于计时器服务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是相对于实际调用xTimerStart()的时间。定时器服务任务的优先级由configTIMER_TASK_PRIORITY配置常量设置。

pdFAIL

- 启动命令没有发送到定时器命令队列，因为队列已经满了。
- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务将被置于Blocked状态，等待计时器服务任务在队列中腾出空间，但指定的块时间在此之前已经过期。

提示

为了使xTimerStart()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

示例

请参阅xTimerCreate() API函数的示例。

5.18 xTimerStartFromISR()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerStartFromISR( TimerHandle_t
xTimer,                                BaseType_t *pxHigherPriorityTaskWoken );
```

概要

可以从中断服务例程调用的xTimerStart()版本。

参数

xTimer

- 正在启动、重置或重新启动的定时器的句柄。

pxHigherPriorityTaskWoken

- xTimerStartFromISR()将命令写入计时器命令队列。如果写定时器命令队列使定时器服务任务离开阻塞状态,和定时器服务任务优先级等于或大于当前执行的任务(任务被打断),然后* pxHigherPriorityTaskWoken将被设置在xTimerStartFromISR pdTRUE内部()函数。如果xTimerStartFromISR()将这个值设置为pdTRUE，那么应该在中断退出之前执行上下文切换。

返回值

pdPASS

- start命令成功发送到定时器命令队列。当命令被实际处理时，将取决于计时器服务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是相对于实际调用xTimerStartFromISR()的时间。定时器服务任务的优先级由configTIMER_TASK_PRIORITY配置常量设置。

pdFAIL

- 启动命令没有发送到定时器命令队列，因为队列已经满了。

提示

为了使xTimerStartFromISR()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

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 by a one-shot timer. 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   initialized 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). */   }}
```

5.19 xTimerStop()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerStop( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

概要

停止计时器运行。xTimerStopFromISR()是一个可以从中断服务例程调用的等价函数。

参数

xTimer

- 要停止的计时器。

xTicksToWait

- 计时器功能不是由核心FreeRTOS代码提供的，而是由计时器服务(或守护进程)任务提供的。FreeRTOS定时器API通过一个名为定时器命令队列的队列向定时器服务任务发送命令。xTicksToWait指定当计时器命令队列已经满时，任务应保持在Blocked状态以等待空间可用的最大时间量。
- 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。可以使用pdMS_TO_TICKS()宏将以毫秒为单位指定的时间转换为以ticks为单位指定的时间。
- 将xTicksToWait设置为portMAX_DELAY将导致任务无限期等待(没有超时)，前提是FreeRTOSConfig.h中INCLUDE_vTaskSuspend设置为1。
- 如果在启动调度程序之前调用xTimerStop()，则xTicksToWait将被忽略。

返回值

pdPASS

- stop命令成功发送到定时器命令队列。
- 如果一块指定时间(xTicksToWait不是零),然后调用任务可能被进入阻塞状态等待空间可用定时器命令队列函数返回之前,但数据成功写入队列阻塞时间到期之前。
- 当命令被实际处理时，将取决于定时器服务任务相对于系统中其他任务的优先级。定时器服务任务的优先级由configTIMER_TASK_PRIORITY配置常量设置。

pdFAIL

- 停止命令没有发送到计时器命令队列，因为队列已经满了。
- 如果指定了一个块时间(xTicksToWait不为零)，那么调用任务将被置于Blocked状态，等待计时器服务任务在队列中腾出空间，但指定的块时间在此之前已经过期。

提示

为了使xTimerStop()可用，在FreeRTOSConfig.h中configUSE_TIMERS必须设置为1。

示例

请参阅xTimerCreate() API函数的示例。

5.20 xTimerStopFromISR()

```
#include “FreeRTOS.h”#include “timers.h”BaseType_t xTimerStopFromISR( TimerHandle_t
xTimer,                          BaseType_t *pxHigherPriorityTaskWoken );
```

概要

可以从中断服务例程调用的xTimerStop()版本。

参数

xTimer

- 正在停止的定时器的句柄。

pxHigherPriorityTaskWoken

- xTimerStopFromISR()将命令写入计时器命令队列。如果写定时器命令队列使定时器服务任务离开阻塞状态,和定时器服务任务优先级等于或大于当前执行的任务(任务被打断),然后* pxHigherPriorityTaskWoken将被设置在xTimerStopFromISR pdTRUE内部()函数。如果xTimerStopFromISR()将这个值设置为pdTRUE，那么应该在中断退出之前执行上下文切换。

返回值

pdPASS

- stop命令成功发送到定时器命令队列。当命令被实际处理时，将取决于定时器服务任务相对于系统中其他任务的优先级。定时器服务任务的优先级由configTIMER_TASK_PRIORITY配置常量设置。

pdFAIL

- 停止命令没有发送到计时器命令队列，因为队列已经满了。

提示

为了xTimerStopFromISR()可用，必须在FreeRTOSConfig.h中将configUSE_TIMERS设置为1。

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). */
}}
```

章节6 事件组

6.1 xEventGroupClearBits()

```
#include “FreeRTOS.h”#include “event_groups.h”EventBits_t xEventGroupClearBits(
EventGroupHandle_t xEventGroup,                                const EventBits_t
uxBitsToClear );
```

概要

清除RTOS事件组中的位(标志)。这个函数不能从中断调用。请参阅xEventGroupClearBitsFromISR()了解可以从中断调用的版本。

参数

xEventGroup

- 要清除的位所在的事件组。之前必须使用对xEventGroupCreate()的调用创建事件组。

uxBitsToClear

- 一个位值，指示事件组中要清除的位或位。例如，将uxBitsToClear设置为0x08以清除第3位。设置uxBitsToClear为0x09以清除位3和位0。

返回值

所有的值

- 清除事件组中任何位之前的位数。

提示

为了使xEventGroupClearBits()函数可用，RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中。

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. */  }}
```

6.2 xEventGroupClearBitsFromISR()

```
#include “FreeRTOS.h”#include “event_groups.h”BaseType_t xEventGroupClearBitsFromISR( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToClear );
```

概要

可以从中断调用的xEventGroupClearBits()的一个版本。

xEventGroupClearBitsFromISR()向RTOS守护任务发送一条消息，让它在守护任务的上下文中执行清除操作。守护任务的优先级由FreeRTOSConfig.h中的configTIMER_TASK_PRIORITY设置。

参数

xEventGroup

- 要清除的位所在的事件组。之前必须使用对xEventGroupCreate()的调用创建事件组。

uxBitsToClear

- 一个位值，指示事件组中要清除的位或位。例如，将uxBitsToClear设置为0x08以清除第3位。设置uxBitsToClear为0x09以清除位3和位0。

返回值

pdPASS

- 消息被发送到RTOS守护任务。

pdFAIL

- 消息无法发送到RTOS守护进程任务(也称为计时器服务任务)，因为计时器命令队列已满。队列的长度由FreeRTOSConfig.h中的configTIMER_QUEUE_LENGTH设置。

提示

为了使xEventGroupClearBitsFromISR()函数可用，RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中。

示例

```
#define BIT_0 ( 1 << 0 )#define BIT_4 ( 1 << 4 )/* This code assumes the event group referenced by the xEventGroup variable hasalready 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 clear bits message was sent to the daemon task. */   }   else    {       /* The clear bits message was not sent to the daemon task. */   }}
```

6.3 xEventGroupCreate()

```
#include “FreeRTOS.h”#include “event_groups.h”EventGroupHandle_t xEventGroupCreate( void );
```

概要

创建一个新的事件组，并返回一个句柄，可通过该句柄引用已创建的事件组。

每个事件组需要[非常]少量的RAM，用于保存事件组的状态。如果使用xEventGroupCreate()创建一个事件组，那么这个RAM将自动从FreeRTOS堆中分配。如果使用xEventGroupCreateStatic()创建事件组，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

事件组存储在类型为EventGroupHandle_t的变量中。如果configUSE_16_BIT_TICKS设置为1，则事件组中实现的位(或标志)的数量为8，如果configUSE_16_BIT_TICKS设置为0，则为24。configUSE_16_BIT_TICKS依赖于RTOS任务内部实现中用于线程本地存储的数据类型。

这个函数不能从中断调用。

参数

无

返回值

NULL

- 无法创建事件组，因为没有足够的可用FreeRTOS堆。

任何其他值

- 创建事件组，返回的值是创建的事件组的句柄。

提示

configSUPPORT_DYNAMIC_ALLOCATION必须在FreeRTOSConfig.h中设置为1(或未定义，在这种情况下它将默认为1)，并且RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中，以便xEventGroupCreate()函数可用。

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. */}
```

6.4 xEventGroupCreateStatic()

```
#include “FreeRTOS.h”#include “event_groups.h”EventGroupHandle_t xEventGroupCreateStatic( StaticEventGroup_t *pxEventGroupBuffer );
```

概要

创建一个新的事件组，并返回一个句柄，可通过该句柄引用已创建的事件组。

每个事件组需要[非常]少量的RAM，用于保存事件组的状态。如果使用xEventGroupCreate()创建一个事件组，那么这个RAM将自动从FreeRTOS堆中分配。如果使用xEventGroupCreateStatic()创建事件组，那么应用程序编写器将提供RAM，这需要一个额外的参数，但允许在编译时静态地分配RAM。

事件组存储在类型为EventGroupHandle_t的变量中。如果configUSE_16_BIT_TICKS设置为1，则事件组中实现的位(或标志)的数量为8，如果configUSE_16_BIT_TICKS设置为0，则为24。configUSE_16_BIT_TICKS依赖于RTOS任务内部实现中用于线程本地存储的数据类型。

参数

pxEventGroupBuffer

- 必须指向类型为StaticEventGroup_t的变量，事件组的数据结构将存储在其中。

返回值

NULL

- 无法创建事件组，因为pxEventGroupBuffer为空。

任何其他值

- 创建事件组，返回的值是创建的事件组的句柄。

提示

在FreeRTOSConfig.h中，configSUPPORT_STATIC_ALLOCATION必须设置为1，并且RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中，以便xEventGroupCreateStatic()函数可用。

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;void vAFunction( void ){    /* Attempt to create the event group.
*/    xEventGroupHandle = xEventGroupCreate( &xCreatedEventGroup );   /*
pxEventGroupBuffer was not null so expect the event group to have beencreated. */
configASSERT( xEventGroupHandle );}
```

6.5 vEventGroupDelete()

```
#include “FreeRTOS.h”#include “event_groups.h”void vEventGroupDelete(
EventGroupHandle_t xEventGroup );
```

概要

删除先前通过调用xEventGroupCreate()创建的事件组。

在被删除的事件组上被阻塞的任务将被解除阻塞，并报告事件组值为0。

这个函数不能从一个中断调用。

参数

xEventGroup

- 需要删除的事件组。

返回值

无

提示

为了使vEventGroupDelete()函数可用，RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中。

6.6 xEventGroupGetBits()

```
#include “FreeRTOS.h”#include “event_groups.h”EventBits_t xEventGroupGetBits(
EventGroupHandle_t xEventGroup );
```

概要

返回事件组中事件位(事件标志)的当前值。这个函数不能在中断中使用。可以在中断中使用的版本请参见xEventGroupGetBitsFromISR()。

参数

xEventGroup

- 正在查询的事件组。之前必须使用对xEventGroupCreate()的调用创建事件组。

返回值

调用xEventGroupGetBits()时事件组中的事件位的值。

提示

为了使xEventGroupGetBits()函数可用，RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中。

6.7 xEventGroupGetBitsFromISR()

```
#include “FreeRTOS.h”#include “event_groups.h”EventBits_t xEventGroupGetBitsFromISR(
EventGroupHandle_t xEventGroup );
```

概要

可以从中断调用的xEventGroupGetBits()的一个版本。

参数

xEventGroup

- 正在查询的事件组。之前必须使用对xEventGroupCreate()的调用创建事件组。

返回值

调用xEventGroupGetBitsFromISR()时事件组中的事件位的值。

提示

为了使xEventGroupGetBitsFromISR()函数可用，RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中。

6.8 xEventGroupSetBits()

```
#include “FreeRTOS.h”#include “event_groups.h”EventBits_t xEventGroupSetBits(
EventGroupHandle_t xEventGroup,                              const EventBits_t
uxBitsToSet );
```

概要

设置RTOS事件组中的位(标志)。这个函数不能从中断调用。请参阅xEventGroupSetBitsFromISR()了解可以从中断调用的版本。

在事件组中设置位将自动解除阻塞的任何等待位设置的任务。

参数

xEventGroup

- 要在其中设置位的事件组。之前必须使用对xEventGroupCreate()的调用创建事件组。

uxBitsToSet

- 一个按位值，表示要在事件组中设置的一个或多个位。例如，将uxBitsToSet设置为0x08，只设置第3位。设置uxBitsToSet为0x09以设置位3和位0。

返回值

对xEventGroupSetBits()的调用返回时事件组中的位的值。

有两个原因可以解释为什么返回值可能有uxBitsToSet参数指定的位被清除:

1. 如果设置一个位会导致一个等待位离开阻塞状态的任务，那么这个位可能会被自动清除(参见xEventGroupWaitBits()的xClearBitsOnExit参数)。
2. 任何任务,阻塞状态的位被设置(或其它任何任务就绪状态),任务的优先级高于叫xEventGroupSetBits()将执行并可能改变前的事件组值调用xEventGroupSetBits()返回。

提示

为了使xEventGroupSetBits()函数可用，RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中。

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. */  }}
```

6.9 xEventGroupSetBitsFromISR()

```
#include “FreeRTOS.h”#include “event_groups.h”BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup,                                    const EventBits_t uxBitsToSet,                                  BaseType_t *pxHigherPriorityTaskWoken );
```

概要

设置事件组中的位(标志)。可以从中断服务例程(ISR)调用的xEventGroupSetBits()的一个版本。

在事件组中设置位将自动解除阻塞的任何等待位设置的任务。

在事件组中设置位不是确定性操作，因为可能有未知数量的任务等待正在设置的位或位。FreeRTOS不允许在中断或临界区中执行非确定性操作。因此，xEventGroupSetBitsFromISR()向RTOS守护任务发送一条消息，让它在守护任务的上下文中执行set操作——在守护任务中使用调度器锁来代替临界区。守护任务的优先级由FreeRTOSConfig.h中的configTIMER_TASK_PRIORITY设置。

参数

xEventGroup

- 要在其中设置位的事件组。之前必须使用对xEventGroupCreate()的调用创建事件组。

uxBitsToSet

- 一个按位值，表示要在事件组中设置的一个或多个位。例如，将uxBitsToSet设置为0x08，只设置第3位。设置uxBitsToSet为0x09以设置位3和位0。

pxHigherPriorityTaskWoken

- 调用xEventGroupSetBitsFromISR()会导致一条消息被发送到RTOS守护进程任务。如果守护进程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，那么*pxHigherPriorityTaskWoken将被xEventGroupSetBitsFromISR()设置为pdTRUE，这表明应该在中断退出之前请求一个上下文切换。因此，* pxhigherprioritytask唤醒必须初始化为pdFALSE。请参阅下面的示例代码。

返回值

pdPASS

- 消息被发送到RTOS守护任务。

pdFAIL

- 消息无法发送到RTOS守护进程任务(也称为计时器服务任务)，因为计时器命令队列已满。队列的长度由FreeRTOSConfig.h中的configTIMER_QUEUE_LENGTH设置。

提示

为了使xEventGroupSetBitsFromISR()函数可用，RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中。

INCLUDE_xEventGroupSetBitsFromISR, configUSE_TIMERS和INCLUDE_xTimerPendFunctionCall都必须在FreeRTOSConfig.h中设置为1，以便xEventGroupSetBitsFromISR()函数可用。

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 toxEventGroupCreate(). */EventGroupHandle_t xEventGroup;void anInterruptHandler( void ){   BaseType_t xHigherPriorityTaskWoken, xResult;   /* xHigherPriorityTaskWoken must be initialized 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 ); }}
```

6.10 xEventGroupSync()

```
#include “FreeRTOS.h”#include “event_groups.h”EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup,                              const EventBits_t uxBitsToSet,                          const EventBits_t uxBitsToWaitFor, TickType_t xTicksToWait );
```

概要

原子地设置事件组中的位(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)，其中每个任务必须等待其他任务到达一个同步点后才能继续。

如果uxBitsToWaitFor参数指定的位被设置，或者在该时间内被设置，函数将在它的块时间到期之前返回。在这种情况下，所有uxBitsToWaitFor指定的位在函数返回之前将被自动清除。

此功能不能在中断中使用。

参数

xEventGroup

- 正在其中设置和测试比特的事件组。之前必须使用对xEventGroupCreate()的调用创建事件组。

uxBitsToSet

- 一个按位的值，它指示在事件组中设置的一个或多个位，然后确定是否(并可能等待)设置了uxBitsToWaitFor参数所指定的所有位。例如，将uxBitsToSet设置为0x04以设置事件组中的第2位。

uxBitsToWaitFor

- 一个位值，指示事件组内要测试的位或位。例如，设置uxBitsToWaitFor为0x05，等待第0位和第2位。设置uxBitsToWaitFor为0x07来等待第0位、第1位和第2位。等。

xTicksToWait

- 等待uxBitsToWaitFor参数值指定的所有位的最大时间(在'ticks'中指定)。

返回值

所有值

- 事件组在等待的位被设置或阻塞时间过期时的值。测试返回值以知道设置了哪些位。
- 如果xEventGroupSync()因为超时过期而返回，那么返回值中不会设置所有等待的位。
- 如果xEventGroupSync()返回是因为它所等待的所有位都被设置了，那么返回的值是任何位被自动清除之前的事件组值。

提示

为了使xEventGroupSync()函数可用，RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中。

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 synchronize three tasks. It is assumed this eventgroup has already been created elsewhere. */EventGroupHandle_t xEventBits;void vTask0( void *pvParameters ){  EventBits_t uxReturn;   TickType_t xTicksToWait = pdMS_TO_TICKS( 100 ); 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 synchronization      point when all the ALL_SYNC_BITS bits are set. Wait a maximum of 100ms      for this to happen. */ uxReturn = xEventGroupSync( xEventBits,                                    TASK_0_BIT, /* The bit to set. */                                    ALL_SYNC_BITS, /* The bits to wait for. */                                    xTicksToWait );/* Timeout value. */ if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )     {           /* All three tasks reached the synchronization point before the call           to xEventGroupSync() timed out. */      }   }}void vTask1( void *pvParameters ){ for( ;; )   {       /* Perform task functionality here. */      . . .       /* Set bit 2 in the event group to note this task has reached the       synchronization point. The other two tasks will set the other two       bits defined by ALL_SYNC_BITS. All three tasks have reached the     synchronization 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 synchronization 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       synchronization point. The other two tasks will set the other two       bits defined by ALL_SYNC_BITS. All three tasks have reached the     synchronization 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 synchronization was made by all       three tasks, so there is no need to test the return value. */   }}
```

6.11 xEventGroupWaitBits()

```
#include “FreeRTOS.h”#include “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 );
```

概要

在一个RTOS事件组中读取位，可以选择进入阻塞状态(有一个超时)来等待一个或一组位被设置。

这个函数不能从中断调用。

参数

xEventGroup

- 正在其中测试比特的事件组。之前必须使用对xEventGroupCreate()的调用创建事件组。

uxBitsToWaitFor

- 一个位值，指示事件组内要测试的位或位。例如，要等待第0位和/或第2位，设置uxBitsToWaitFor为0x05。要等待第0位和/或第1位和/或第2位，设置uxBitsToWaitFor为0x07等。
- uxBitsToWaitFor不能设置为0。

xClearOnExit

- 如果xClearOnExit设置为pdTRUE，那么在xEventGroupWaitBits()返回之前，如果xEventGroupWaitBits()返回除了超时以外的任何原因，在uxBitsToWaitFor参数的值中设置的任何位将在事件组中清除。超时值由xTicksToWait参数设置。
- 如果xClearOnExit设置为pdFALSE，那么当调用xEventGroupWaitBits()返回时，在事件组中设置的位不会改变。

xWaitAllBits

- xWaitForAllBits用于创建逻辑与测试(必须设置所有位)或逻辑或测试(必须设置一个或多个位)，如下所示:
    - 如果xWaitForAllBits被设置为pdTRUE，那么当所有通过uxBitsToWaitFor参数的值设置在事件组中或者指定的块时间过期时，xEventGroupWaitBits()将返回。
    - 如果xWaitForAllBits设置为pdFALSE，那么当uxBitsToWaitFor参数传递的任何位在事件组中设置或指定的块时间过期时，xEventGroupWaitBits()将返回。

xTicksToWait

- 等待uxBitsToWaitFor所指定的位的一个/所有位(取决于xWaitForAllBits的值)的最大时间量(在'ticks'中指定)成为设置。

返回值

任何值

- 设置等待的事件位或块时间过期时的事件组值。如果较高优先级的任务或中断改变了调用任务离开Blocked状态和退出xEventGroupWaitBits()函数之间的事件位值，则事件组中事件位的当前值将与返回值不同。
- 测试返回值以知道设置了哪些位。如果xEventGroupWaitBits()因为超时而返回，那么不是所有正在等待的位都会被设置。如果xEventGroupWaitBits()返回，因为它正在等待的位被设置，那么在xClearOnExit参数被设置为pdTRUE的情况下，在任何位被自动清除之前，返回的值是事件组值。

提示

为了使xEventGroupWaitBits()函数可用，RTOS源文件FreeRTOS/source/event_groups.c必须包含在构建中。

示例

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )
void aFunction( EventGroupHandle_t xEventGroup )
{
    EventBits_t uxBits;
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );
    /* 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 and 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. */
    }
}
```

章节7 内核配置

7.1 FreeRTOSConfig.h

内核配置是通过在FreeRTOSConfig.h中设置#define constants实现的。每个使用FreeRTOS的应用程序必须提供一个FreeRTOSConfig.h头文件。

FreeRTOS下载中包含的所有演示应用程序项目都包含一个预定义的FreeRTOSConfig.h，可以用作引用或简单复制。然而，请注意，有些演示项目是在本章中所有记录的选项可用之前生成的，因此它们包含的FreeRTOSConfig.h头文件将不会包含以下小节中记录的所有常量和选项。

7.2 Constants that Start “INCLUDE_”

以文本“INCLUDE_”开头的常量用于从应用程序中包含或排除FreeRTOS API函数。例如，将INCLUDE_vTaskPrioritySet设置为0将从构建中排除vTaskPrioritySet() API函数，这意味着应用程序不能调用vTaskPrioritySet()。将INCLUDE_vTaskPrioritySet设置为1将在构建中包含vTaskPrioritySet() API函数，因此应用程序可以调用vTaskPrioritySet()。

在某些情况下，一个INCLUDE_配置常量将包含或排除多个API函数。

“INCLUDE” *常量的提供是为了通过删除不需要的FreeRTOS函数和特性来减少代码的大小。然而，大多数链接器在默认情况下会自动删除未引用的代码，除非优化完全关闭。没有此默认行为的链接器通常可以配置为删除未引用的代码。因此，在大多数实际情况下，INCLUDE*配置常量对可执行代码的大小几乎没有影响。

从应用程序中排除API函数也可能减少FreeRTOS内核使用的RAM数量。例如，删除vTaskSuspend() API函数也将阻止结构，否则将引用挂起的任务。

INCLUDE_xEventGroupSetBitsFromISR

configUSE_TIMERS, INCLUDE_xTimerPendFunctionCall和INCLUDE_xEventGroupSetBitsFromISR必须全部设置为1，以便xEventGroupSetBitsFromISR () API函数可用。

INCLUDE_xSemaphoreGetMutexHolder

INCLUDE_xSemaphoreGetMutexHolder必须设置为1才能使用xSemaphoreGetMutexHolder() API函数。

INCLUDE_xTaskAbortDelay

INCLUDE_xTaskAbortDelay必须设置为1才能使用xTaskAbortDelay() API函数。

INCLUDE_vTaskDelay

INCLUDE_vTaskDelay必须设置为1，以便vTaskDelay() API函数可用。

INCLUDE_vTaskDelayUntil

为了使vTaskDelayUntil() API函数可用，INCLUDE_vTaskDelayUntil必须设置为1。

INCLUDE_vTaskDelete

INCLUDE_vTaskDelete必须设置为1，以便vTaskDelete() API函数可用。

INCLUDE_xTaskGetCurrentTaskHandle

include_xtaskgetcurrentttaskhandle必须设置为1才能使用xTaskGetCurrentTaskHandle() API函数。

INCLUDE_xTaskGetHandle

为了让xTaskGetHandle() API函数可用，INCLUDE_xTaskGetHandle必须设置为1。

INCLUDE_xTaskGetIdleTaskHandle

INCLUDE_xTaskGetIdleTaskHandle必须设置为1，以便xTaskGetIdleTaskHandle() API函数可用。

INCLUDE_xTaskGetSchedulerState

INCLUDE_xTaskGetSchedulerState必须设置为1才能使用xTaskGetSchedulerState() API函数。

INCLUDE_uxTaskGetStackHighWaterMark

INCLUDE_uxTaskGetStackHighWaterMark必须设置为1，这样uxTaskGetStackHighWaterMark() API函数才可用。

INCLUDE_uxTaskPriorityGet

INCLUDE_uxTaskPriorityGet必须设置为1，以便uxTaskPriorityGet() API函数可用。

INCLUDE_vTaskPrioritySet

INCLUDE_vTaskPrioritySet必须设置为1，以便vTaskPrioritySet() API函数可用。

INCLUDE_xTaskResumeFromISR

INCLUDE_xTaskResumeFromISR和INCLUDE_vTaskSuspend都必须设置为1，这样xTaskResumeFromISR() API函数才可用。

INCLUDE_eTaskGetState

INCLUDE_eTaskGetState必须设置为1，以便eTaskGetState() API函数可用。

INCLUDE_vTaskSuspend

INCLUDE_vTaskSuspend必须设置为1，以便vTaskSuspend()、vTaskResume()和xTaskIsTaskSuspended() API函数可用。

INCLUDE_vTaskSuspend和INCLUDE_xTaskResumeFromISR必须都设置为1，以便xTaskResumeFromISR() API函数可用。

一些队列和信号量API函数允许调用任务选择将其置于Blocked状态，以等待队列或信号量事件的发生。这些API函数需要指定一个最大的块时间，或者超时时间。然后，调用任务将保持在Blocked状态，直到队列或信号量事件发生，或者块周期到期。可指定的最大块周期由portMAX_DELAY定义。如果INCLUDE_vTaskSuspend被设置为0，那么指定一个portMAX_DELAY的块周期将会导致调用任务被放置到阻塞状态，以获得最大的portMAX_DELAY节拍。如果INCLUDE_vTaskSuspend被设置为1，那么指定一个

portMAX_DELAY的块周期将导致调用任务被无限期地放置到Blocked状态(没有超时)。在第二种情况下，块周期是不确定的，因此摆脱Blocked状态的唯一方法是发生队列或信号量事件。

INCLUDE_xTimerPendFunctionCall

configUSE_TIMERS和INCLUDE_xTimerPendFunctionCall都必须设置为1，以便xTimerPendFunctionCall()和xTimerPendFunctionCallFromISR () API函数可用。

7.3 Constants that Start “config”

以“config”文本开头的常量定义内核的属性，或包含或排除内核的特性。

configAPPLICATION_ALLOCATED_HEAP

默认情况下，FreeRTOS堆由FreeRTOS声明并由链接器放置在内存中。将configAPPLICATION_ALLOCATED_HEAP设置为1允许应用程序编写器声明堆，这允许应用程序编写器将堆放置在内存中他们喜欢的任何地方。

如果使用heap_1.c、heap_2.c或heap_2.c，并且configAPPLICATION_ALLOCATED_HEAP被设置为1，那么应用程序编写器必须提供具有名称和维数的uint8_t数组，如下所示。该数组将用作FreeRTOS堆。数组如何放置在特定的内存位置取决于所使用的编译器——请参阅编译器文档。

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

configASSERT

对configASSERT(x)的调用存在于FreeRTOS内核代码的关键位置。

如果FreeRTOS功能正常，并且被正确使用，那么configASSERT()参数将是非零。如果发现该参数等于零，则错误发生了。

configASSERT()所捕获的大多数错误很可能是一个无效参数被传递到FreeRTOS API函数的结果。因此，configASSERT()可以帮助进行运行时调试。然而，定义configASSERT()也会增加应用程序代码的大小，并降低其执行速度。

configASSERT()等价于标准C中的assert()宏。它被用来代替标准的C assert()宏，因为不是所有可以用来构建FreeRTOS的编译器都提供assert.h头文件。

configASSERT()应该在FreeRTOSConfig.h中定义。下文显示了一个示例configASSERT()定义，它假定vAssertCalled()是由应用程序在其他地方定义的。

```
#define configASSERT( ( x ) ) if( ( x ) == 0 ) vAssertCalled( __FILE__, __LINE__ )
```

configCHECK_FOR_STACK_OVERFLOW

每个任务都有一个唯一的堆栈。如果使用xTaskCreate() API函数创建任务，那么堆栈将自动从FreeRTOS堆中分配，并且堆栈的大小由xTaskCreate() usStackDepth参数指定。如果任务是使用xTaskCreateStatic() API函数创建的，那么堆栈将由应用程序编写器预先分配。

堆栈溢出是导致应用程序不稳定的常见原因。FreeRTOS提供了两种可选机制，可用于帮助进行堆栈溢出检测和调试。使用哪个选项(如果有的话)是由configCHECK_FOR_STACK_OVERFLOW配置常量配置的。

如果configCHECK_FOR_STACK_OVERFLOW没有被设置为0，那么应用程序还必须提供一个堆栈溢出钩子(或回调)函数。每当检测到堆栈溢出时，内核将调用堆栈溢出钩子。

堆栈溢出钩子函数必须被称为vApplicationStackOverflowHook()，其原型如下所示：

```
void vApplicationStackOverflowHook( TaskHandle_t *pxTask,
                                    signed char *pcTaskName );
```

已经超过其堆栈空间的任务的名称和句柄分别使用pcTaskName和pxTask参数传递给堆栈溢出钩子函数。应该注意的是，堆栈溢出可能破坏这些参数，在这种情况下，可以检查pxcurrentttcb变量，以确定是哪个任务导致调用堆栈溢出钩子函数。

堆栈溢出检查只能用于具有线性(而不是分段)内存映射的架构。

在调用堆栈溢出回调函数之前，一些处理器会生成一个错误异常来响应堆栈损坏。

堆栈溢出检查会增加执行上下文切换所花费的时间。

栈溢出检测方法一

- 方法一通过将configCHECK_FOR_STACK_OVERFLOW设置为1来选择。
- 当在上下文切换期间将任务上下文保存到堆栈时，任务堆栈的利用率可能会达到最大。堆栈溢出检测方法一检查当时的堆栈利用率，以确保任务堆栈指针保持在有效的堆栈区域内。如果堆栈指针包含一个无效值(一个引用有效堆栈区域之外内存的值)，堆栈溢出钩子函数将被调用。
- 方法一是快速的，但不一定会捕获所有堆栈溢出事件。

栈溢出检测方法二

- 方法二通过将configCHECK_FOR_STACK_OVERFLOW设置为2来选择。
- 方法二包括方法一执行的检查。此外，方法二还将验证有效堆栈区域的限制没有被覆盖。
- 分配给任务的堆栈在创建任务时用一个已知的模式填充。方法2检查有效堆栈范围内的最后n个字节，以确保该模式保持未修改(没有被覆盖)。如果这n个字节中的任何一个发生了改变，就会调用堆栈溢出钩子函数。
- 方法二比方法一效率低，但仍然快。它将捕获大多数堆栈溢出发生，尽管可以想象有些可能会丢失(例如，在没有写入最后n个字节的情况下发生堆栈溢出)。

configCPU_CLOCK_HZ

这必须设置为驱动外设的时钟频率，外设用来生成内核的周期性滴答中断。这通常等于(但不总是)主系统时钟频率。

configSUPPORT_DYNAMIC_ALLOCATION

如果configSUPPORT_DYNAMIC_ALLOCATION设置为1，那么可以使用从FreeRTOS堆中自动分配的RAM创建RTOS对象。如果configSUPPORT_DYNAMIC_ALLOCATION设置为0，那么RTOS对象只能使用应用程序编写器提供的RAM创建。也看到configSUPPORT_STATIC_ALLOCATION。

如果没有定义configSUPPORT_DYNAMIC_ALLOCATION，那么它将默认为1。

configENABLE_BACKWARD_COMPATIBILITY

FreeRTOS.h头文件包含一组#define宏，这些宏将FreeRTOS 8.0.0之前版本中使用的数据类型名称映射到FreeRTOS 8.0.0版本中使用的名称。这些宏允许应用程序代码将FreeRTOS的版本从8.0.0之前的版本更新到8.0.0之后的版本，而无需修改。在FreeRTOSConfig.h中将configENABLE_BACKWARD_COMPATIBILITY设置为0会将宏从构建中排除，这样做会允许验证没有使用8.0.0之前版本的名称。

configGENERATE_RUN_TIME_STATS

任务运行时统计特性收集关于每个任务接收的处理时间的信息。该特性要求应用程序配置一个运行时统计数据时基。运行时统计信息时基的频率必须至少是tick中断频率的十倍。

将configGENERATE_RUN_TIME_STATS设置为1将包含运行时统计信息收集功能和构建中的相关API。将configGENERATE_RUN_TIME_STATS设置为0将从构建中排除运行时统计信息收集功能和相关API。

如果configGENERATE_RUN_TIME_STATS设置为1，那么应用程序还必须提供下表中描述的宏的定义。如果configGENERATE_RUN_TIME_STATS设置为0，则应用程序必须不定义表2中描述的任何宏，否则就有可能导致应用程序无法编译和/或链接。

宏	描述
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	必须提供此宏来初始化用于生成运行时统计数据时基的任何外设。
portGET_RUN_TIME_COUNTER_VALUE(), or portALT_GET_RUN_TIME_COUNTER_VALUE(Time)	必须提供这两个宏中的一个来返回当前的时基值——这是应用程序在选择的时基单元中运行的总时间。如果使用了第一个宏，则必须定义该宏以计算当前的时间基值。如果使用第二个宏，则必须定义它将其“Time”参数设置为当前的时间基值。(宏名称中的“ALT”是“ALTernative”的缩写)。

configIDLE_SHOULD_YIELD

configIDLE_SHOULD_YIELD控制空闲任务的行为，如果有应用程序任务也以空闲优先级运行。只有在使用了抢占式调度程序时，它才会起作用。

共享优先级的任务使用轮循，时间切片，算法调度。每个任务将被依次选择以进入运行状态，但可能不会在整个计时周期内保持运行状态。例如，一个任务可能被抢占，选择放弃，或者选择在下一个tick中断之前进入阻塞状态。

如果configIDLE_SHOULD_YIELD设置为0，那么空闲任务将永远不会让出给其他任务，只有当它被抢占时才会离开Running状态。

如果configIDLE_SHOULD_YIELD设置为1，那么如果有另一个idle优先级任务处于Ready状态，空闲任务将永远不会执行其定义的功能的不止一次迭代，而不会屈服于另一个任务。这确保当应用程序任务可以运行时，在空闲任务上花费的时间最少。

Idle任务一致地让位于另一个Idle优先级Ready状态任务，其副作用如下图所示。

上图显示了四个任务的执行模式，它们都以空闲优先级运行。任务A、B和C是应用程序任务。任务1为空闲任务。tick中断以固定的间隔启动上下文切换，在T0、T1、T2等时刻显示。可以看到Idle任务在T2时间开始执行。它执行一个时间片的一部分，然后让任务a执行同一个时间片的其余部分，然后在时间T3被抢占。任务I和任务A有效地共享一个时间片，从而导致任务B和任务C始终比任务A占用更多的处理时间。

将configIDLE_SHOULD_YIELD设置为0可以通过确保Idle任务在整个tick周期内保持运行状态来防止这种行为(除非被除tick中断之外的其他中断抢占)。在这种情况下，按时间平均值计算，共享空闲优先级的其他任务将获得相同的处理时间份额，但执行空闲任务的时间也会更多。使用Idle任务钩子函数可以确保有效地使用Idle任务执行所花费的时间。

configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS

configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS只被FreeRTOS MPU使用。

如果configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS被设置为1，那么应用程序编写器必须提供一个名为“application_defined_privileged_functions.h”的头文件，在这个头文件中应用程序编写器需要以特权模式执行的函数可以被实现。请注意，尽管有.h扩展名，头文件应该包含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prvraisprivilege()函数和portRESET_PRIVILEGE()宏保存和恢复处理器的特权状态。例如，如果库中提供的print函数访问不在应用程序编写器控制范围内的RAM，因此不能分配给受内存保护的用户模式任务，那么可以使用以下代码将print函数封装到一个特权函数中:

```
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 );
}
```

这种技术应该只在开发期间使用，而不是部署期间，因为它绕过了内存保护。

configKERNEL_INTERRUPT_PRIORITY, configMAX_SYSCALL_INTERRUPT_PRIORITY, configMAX_API_CALL_INTERRUPT_PRIORITY

configMAX_API_CALL_INTERRUPT_PRIORITY是configMAX_SYSCALL_INTERRUPT_PRIORITY的新名称，它只被较新的端口使用。两者是等价的。

configKERNEL_INTERRUPT_PRIORITY和configMAX_SYSCALL_INTERRUPT_PRIORITY只与实现中断嵌套的端口相关。

如果一个端口只实现configKERNEL_INTERRUPT_PRIORITY配置常量，那么configKERNEL_INTERRUPT_PRIORITY设置内核本身使用的中断的优先级。在这种情况下，ISR安全的FreeRTOS API函数(那些以“FromISR”结尾的函数)不能从任何被分配了优先级高于configKERNEL_INTERRUPT_PRIORITY设置的中断调用。不调用API函数的中断可以以更高的优先级执行，以确保中断时间、确定性和延迟不会受到内核正在执行的任何事情的不利影响。

如果一个端口同时实现了configKERNEL_INTERRUPT_PRIORITY和configMAX_SYSCALL_INTERRUPT_PRIORITY配置常量，那么configKERNEL_INTERRUPT_PRIORITY设置内核本身使用的中断的优先级，configMAX_SYSCALL_INTERRUPT_PRIORITY设置中断的最大优先级，ISR安全的FreeRTOS API函数(那些以“FromISR”结尾的函数)可以从这些中断调用。通过将configMAX_SYSCALL_INTERRUPT_PRIORITY设置在比configKERNEL_INTERRUPT_PRIORITY更高的优先级之上(即在更高的优先级级别上)，可以实现一个完整的中断嵌套模型。不调用API函数的中断可以以高于configMAX_SYSCALL_INTERRUPT_PRIORITY的优先级执行，以确保中断计时、确定性和延迟不受内核正在执行的任何事情的不利影响。

作为一个例子——假设一个微控制器有七个中断优先级。在这个假设的情况下，1是最低的中断优先级，7是最高的中断优先级

> 注意，在为configKERNEL_INTERRUPT_PRIORITY和configMAX_SYCALL_INTERRUPT_PRIORITY分配值时必须小心，因为有些微控制器使用0或1表示最低优先级，而其他微控制器使用0或1表示最高优先级。

下图描述了当configKERNEL_INTERRUPT_PRIORITY和configMAX_SYSCALL_INTERRUPT_PRIORITY分别被设置为1和3时，在每个优先级级别上可以做什么和不能做什么。


image-20211119143838760

运行在configMAX_SYSCALL_INTERRUPT_PRIORITY之上的isr不会被屏蔽内核本身，所以它们的响应性不受内核功能的影响。这是非常适合需要非常高时间精度的中断，例如，中断执行电动机换向。然而，中断具有上面的优先级

configMAX_SYSCALL_INTERRUPT_PRIORITY不能调用任何FreeRTOS API函数，即使是以“FromISR”结尾的也不能使用。

configKERNEL_INTERRUPT_PRIORITY几乎总是(不是总是)被设置为最低

configMAX_CO_ROUTINE_PRIORITIES

设置可分配给协同例程的最大优先级。可以分配协同例程优先级从0(最低优先级)到(configMAX_CO_ROUTINE_PRIORITIES -1)，这是最高优先级。

configMAX_PRIORITIES

设置可分配给任务的最大优先级。任务可以被分配优先级从0(最低优先级)到最高的(configMAX_PRIORITIES - 1)

优先级。

configMAX_TASK_NAME_LEN

设置可用于任务名称的最大字符数。NULL结束符包含在字符计数中。

configMAX_SYSCALL_INTERRUPT_PRIORITY

请参阅configKERNEL_INTERRUPT_PRIORITY配置常量的描述。

configMINIMAL_STACK_SIZE

设置分配给Idle任务的堆栈大小。该值是用单词而不是字节指定的。

内核本身不将configMINIMAL_STACK_SIZE用于任何其他目的，尽管标准演示任务广泛使用这个常量。

每个官方FreeRTOS端口都提供了一个演示应用程序。在这样一个特定于端口的演示应用程序中使用的configMINIMAL_STACK_SIZE的值是使用该端口创建的任何任务所建议的最小堆栈大小。

configNUM_THREAD_LOCAL_STORAGE_POINTERS

线程本地存储(或TLS)允许应用程序编写器在任务的控制块中存储值，使该值特定于(本地到)任务本身，并允许每个任务拥有自己独特的值。

每个任务都有自己的指针数组，可以用作线程本地存储。数组中的索引数量由configNUM_THREAD_LOCAL_STORAGE_POINTERS设置。

configQUEUE_REGISTRY_SIZE

设置任何时候可以从队列注册表中引用的队列和信号量的最大数量。只有需要在内核感知调试接口中查看的队列和信号量才需要注册。

队列注册表仅在使用内核感知调试器时才需要。在其他所有时候，它都没有作用，可以通过将configQUEUE_REGISTRY_SIZE设置为0或完全忽略configQUEUE_REGISTRY_SIZE配置常量定义来忽略它。

configSUPPORT_STATIC_ALLOCATION

如果configSUPPORT_STATIC_ALLOCATION设置为1，那么就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被设置为0，那么RTOS对象只能使用从FreeRTOS堆中分配的RAM创建。也看到configSUPPORT_DYNAMIC_ALLOCATION。

如果没有定义configSUPPORT_STATIC_ALLOCATION，那么它将默认为0。

configTICK_RATE_HZ

设置tick中断频率。该值以Hz为单位。

可以使用pdMS_TO_TICKS()宏将以毫秒为单位指定的时间转换为以ticks为单位指定的时间。以这种方式指定的块时间将保持不变，即使configTICK_RATE_HZ定义被更改。当configTICK_RATE_HZ小于等于1000时，才可以使用pdMS_TO_TICKS()。标准演示任务大量使用pdMS_TO_TICKS()，因此它们也只能在configTICK_RATE_HZ小于或等于1000时使用。

configTIMER_QUEUE_LENGTH

FreeRTOS定时器API通过一个名为定时器命令队列的队列向定时器服务任务发送命令。configTIMER_QUEUE_LENGTH设置定时器命令队列一次可以容纳的未处理命令的最大数量。

定时器命令队列可能被填满的原因包括:

- 在启动调度器之前，也就是在创建计时器服务任务之前，执行多个计时器API函数调用。
- 从中断服务例程(ISR)调用多个(中断安全)计时器API函数，因此不允许计时器服务任务处理命令。
- 从优先级高于计时器服务任务的任务发出的多个计时器API函数调用。

configTIMER_TASK_PRIORITY

计时器功能不是由核心FreeRTOS代码提供的，而是由计时器服务(或守护进程)任务提供的。FreeRTOS定时器API通过一个名为定时器命令队列的队列向定时器服务任务发送命令。configTIMER_TASK_PRIORITY设置定时器服务任务的优先级。像所有任务一样，计时器服务任务可以在0到(configMAX_PRIORITIES - 1)之间的任意优先级运行。

需要仔细选择这个值，以满足应用程序的需求。例如，如果计时器服务任务被设置为系统中优先级最高的任务，那么发送给计时器服务任务(当调用计时器API函数时)的命令和过期的计时器都将立即得到处理。相反，如果计时器服务任务被赋予低优先级，那么发送给计时器服务任务的命令和过期的计时器将不会被处理，直到计时器服务任务成为能够运行的最高优先级任务。然而，值得注意的是，计时器到期时间是相对于发送命令时计算的，而不是相对于处理命令时计算的。

configTIMER_TASK_STACK_DEPTH

计时器功能不是由核心FreeRTOS代码提供的，而是由计时器服务(或守护进程)任务提供的。FreeRTOS定时器API通过一个名为定时器命令队列的队列向定时器服务任务发送命令。configTIMER_TASK_STACK_DEPTH设置分配给计时器服务任务的堆栈大小(用单词表示，而不是字节)。

计时器回调函数在计时器服务任务的上下文中执行。因此，计时器服务任务的堆栈需求取决于计时器回调函数的堆栈需求。

configTOTAL_HEAP_SIZE

每次创建任务、队列或信号量时，内核都会从堆中分配内存。FreeRTOS的官方下载包括三个示例内存分配方案。该方案分别在heap_1.c、heap_2.c、heap_3.c和heap_4.c源文件中实现。heap_1.c、heap_2.c和heap_4.c定义的方案从一个静态分配的数组(称为FreeRTOS堆)中分配内存。configTOTAL_HEAP_SIZE设置该数组的大小。大小以字节为单位指定。

除非应用程序正在使用heap_1.c、heap_2.c或heap_4.c，否则configTOTAL_HEAP_SIZE设置没有作用。

configUSE_16_BIT_TICKS

滴答计数保存在类型为TickType_t的变量中。当configUSE_16_BIT_TICKS设置为1时，TickType_t被定义为一个无符号16位类型。当configUSE_16_BIT_TICKS设置为0时，TickType_t被定义为32位无符号类型。

使用16位类型可以大大提高8位和16位微控制器的效率，但代价是限制可以指定的最大块时间。

configUSE_ALTERNATIVE_API

提供了两组API函数用于向队列发送信息和从队列接收信息——标准API和“替代”API。本手册中只记录了标准API。不再推荐使用替代API。

将configUSE_ALTERNATIVE_API设置为1将在构建中包含可选API函数。将configUSE_ALTERNATIVE_API设置为0将从构建中排除可选API函数。

注意:不建议使用替代API，因此不推荐使用。

configUSE_APPLICATION_TASK_TAG

将configUSE_APPLICATION_TASK_TAG设置为1将包括构建中的vTaskSetApplicationTaskTag()和xTaskCallApplicationTaskHook() API函数。将configUSE_APPLICATION_TASK_TAG设置为0将从构建中排除vTaskSetApplicationTaskTag()和xTaskCallApplicationTaskHook() API函数。

configUSE_CO_ROUTINES

协同例程是一种轻量级任务，通过共享堆栈来节省内存，但功能有限。它们的使用从本手册中省略了。

将configUSE_CO_ROUTINES设置为1将包括构建中所有的协同例程功能及其关联的API函数。将configUSE_CO_ROUTINES设置为0将从构建中排除所有协同例程功能及其相关的API函数。

configUSE_COUNTING_SEMAPHORES

将configUSE_COUNTING_SEMAPHORES设置为1将包含计数信号量功能及其在构建中的关联API。将configUSE_COUNTING_SEMAPHORES设置为0将从构建中排除计数信号量功能及其关联的API。

configUSE_DAEMON_TASK_STARTUP_HOOK

如果configUSE_TIMERS和configUSE_DAEMON_TASK_STARTUP_HOOK都被设置为1，那么应用程序必须定义一个钩子函数，其名称和原型完全相同，如下所示。当RTOS守护进程任务(也称为定时器服务)第一次执行时，钩子函数将被调用一次。任何需要运行RTOS的应用程序初始化代码都可以放在钩子函数中。

```
void vApplicationDaemonTaskStartupHook( void );
```

configUSE_IDLE_HOOK

空闲任务钩子函数是一个钩子(或回调)函数，如果定义和配置了它，空闲任务将在其实现的每次迭代中调用它。

如果configUSE_IDLE_HOOK设置为1，那么应用程序必须定义一个空闲任务钩子函数。如果configUSE_IDLE_HOOK被设置为0，那么空闲任务钩子函数将不会被调用，即使已经定义了空闲任务钩子函数。

空闲任务钩子函数必须具有如下所示的名称和原型

```
void vApplicationIdleHook( void );
```

configUSE_MALLOC_FAILED_HOOK

每次创建任务、队列或信号量时，内核使用对pvPortMalloc()的调用从堆中分配内存。FreeRTOS的官方下载包括三个示例内存分配方案。该方案分别在heap_1.c、heap_2.c和heap_4.c源文件中实现。configUSE_MALLOC_FAILED_HOOK仅在使用这三个示例方案中的一个时才相关。

malloc()失败钩子函数是一个钩子(或回调函数)，如果pvPortMalloc()返回NULL，它将被定义和配置。只有当剩余的FreeRTOS堆内存不足以让请求的分配成功时，才返回NULL。

如果configUSE_MALLOC_FAILED_HOOK设置为1，那么应用程序必须定义一个malloc()失败的钩子函数。如果configUSE_MALLOC_FAILED_HOOK被设置为0，那么malloc()失败的钩子函数将不会被调用，即使已经定义了。

失败的Malloc()钩子函数必须具有如下所示的名称和原型。

```
void vApplicationMallocFailedHook( void );
```

configUSE_MUTEXES

将configUSE_MUTEXES设置为1将在构建中包含互斥功能及其关联的API。将configUSE_MUTEXES设置为0将从构建中排除互斥功能及其关联的API。

configUSE_NEWLIB_REENTRANT

如果configUSE_NEWLIB_REENTRANT设置为1，那么将为每个创建的任务分配一个newlib reent结构。

注意，Newlib支持是根据流行的需求而包含的，但FreeRTOS的维护者本身并不使用Newlib。FreeRTOS不对产生的newlib操作负责。用户必须熟悉newlib，并且必须提供必要存根的系统范围实现。请注意(在撰写本文时)当前的newlib设计实现了一个系统范围的malloc()，它必须提供锁。

configUSE_PORT_OPTIMISED_TASK_SELECTION

一些FreeRTOS端口有两种方法来选择要执行的下一个任务—一个泛型方法和一个特定于该端口的方法。

的一般方法:

- 在configUSE_PORT_OPTIMISED_TASK_SELECTION被设置为0或没有实现特定于端口的方法时使用。
- 可用于所有FreeRTOS端口。
- 完全是用C编写的，这使得它的效率低于特定于端口的方法。
- 不限制可用优先级的最大数量。

特定端口的方法:

- 不能用于所有端口。
- 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(通常是等效指令的Count Leading Zeros [CLZ])，因此只能用于专门为其编写的体系结构。
- 比泛型方法更有效。
- 通常对可用优先级的最大数量施加32个限制。

configUSE_PREEMPTION

将configUSE_PREEMPTION设置为1将导致使用抢占式调度程序。将configUSE_PREEMPTION设置为0将导致使用合作调度程序。

当使用抢占式调度程序时，内核将在每次tick中断期间执行，这可能导致在tick中断中发生上下文切换。

当使用合作调度器时，上下文切换只会在以下情况发生:

1. 任务显式地调用taskYIELD()。
2. 任务显式地调用API函数，导致其进入Blocked状态。
3. 应用程序定义的中断显式地执行上下文切换。

configUSE_QUEUE_SETS

将configUSE_QUEUE_SETS设置为1将包括队列集功能(同时阻塞多个队列的能力)及其在构建中的关联API。将configUSE_QUEUE_SETS设置为0将从构建中排除队列集功能及其关联的API。

configUSE_RECURSIVE_MUTEXES

将configUSE_RECURSIVE_MUTEXES设置为1将导致递归互斥功能及其关联的API被包含在构建中。将configUSE_RECURSIVE_MUTEXES设置为0将导致递归互斥功能及其关联的API被排除在构建中。

configUSE_STATS_FORMATTING_FUNCTIONS

将configUSE_TRACE_FACILITY和configUSE_STATS_FORMATTING_FUNCTIONS设置为1，以包含vTaskList()和vTaskGetRunTimeStats()。

构建中的函数。设置为0将在构建中省略vTaskList()和vTaskGetRunTimeStates()。

configUSE_TASK_NOTIFICATIONS

将configUSE_TASK_NOTIFICATIONS设置为1(或未定义configUSE_TASK_NOTIFICATIONS)将包括直接到任务的通知功能及其在构建中的相关API。将configUSE_TASK_NOTIFICATIONS设置为0将从构建中排除直接指向任务的通知功能及其关联的API。

当构建中直接包含任务通知时，每个任务会消耗额外的8字节RAM。

configUSE_TICK_HOOK

tick钩子函数是一个钩子(或回调)函数，如果定义并配置，将在每个tick中断期间调用它。

如果configUSE_TICK_HOOK被设置为1，那么应用程序必须定义一个tick hook函数。如果configUSE_TICK_HOOK被设置为0，那么tick钩子函数将不会被调用，即使已经定义了一个。

TICK_HOOK函数必须具有下文所示的名称和原型。

```
void vApplicationTickHook( void );
```

configUSE_TICKLESS_IDLE

将configUSE_TICKLESS_IDLE设置为1以使用低功耗的无tick模式，或设置为0以保持tick中断一直运行。没有为所有FreeRTOS端口提供低功耗的无tickless实现。

configUSE_TIMERS

将configUSE_TIMERS设置为1将在构建中包含软件计时器功能及其相关的API。将configUSE_TIMERS设置为0将从构建中排除软件计时器功能及其相关API。

如果configUSE_TIMERS设置为1，则configTIMER_TASK_PRIORITY, configTIMER_QUEUE_LENGTH和configTIMER_TASK_STACK_DEPTH也必须定义。

configUSE_TIME_SLICING

默认情况下(如果configUSE_TIME_SLICING没有定义，或者如果configUSE_TIME_SLICING被定义为1)FreeRTOS使用优先抢占式调度和时间切片。这意味着RTOS调度程序将始终运行处于就绪状态的优先级最高的任务，并将在每个RTOS tick中断上在同等优先级的任务之间切换。如果configUSE_TIME_SLICING被设置为0，那么RTOS调度程序将仍然运行处于Ready状态的最高优先级的任务，但是不会因为执行了tick中断而在同等优先级的任务之间切换。

configUSE_TRACE_FACILITY

将configUSE_TRACE_FACILITY设置为1将导致额外的结构成员和函数，这些成员和函数有助于在构建中包含执行可视化和跟踪。