

---

# FreeRTOS 内核

## 开发人员指南

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

## Table of Contents

关于 FreeRTOS 内核 .....	1
价值主张 .....	1
关于术语的注释 .....	1
为什么使用实时内核？ .....	1
FreeRTOS 内核功能 .....	2
许可 .....	3
包含的源文件和项目 .....	3
FreeRTOS 内核分发版 .....	4
了解 FreeRTOS 内核分发版 .....	4
构建 FreeRTOS 内核 .....	4
FreeRTOSConfig.h .....	4
官方 FreeRTOS 内核分发版 .....	4
FreeRTOS 分发版中的顶级目录 .....	4
所有端口通用的 FreeRTOS 源文件 .....	5
特定于端口的 FreeRTOS 源文件 .....	6
标头文件 .....	7
演示应用程序 .....	7
创建 FreeRTOS 项目 .....	8
从头开始创建新项目 .....	9
数据类型和编码样式指南 .....	10
变量名称 .....	10
函数名称 .....	10
格式设置 .....	10
宏名称 .....	10
过量类型强制转换的基本原理 .....	11
堆内存管理 .....	12
先决条件 .....	12
动态内存分配及其与 FreeRTOS 的关联性 .....	12
动态内存分配的选项 .....	12
示例内存分配方案 .....	13
Heap_1 .....	13
Heap_2 .....	14
Heap_3 .....	15
Heap_4 .....	15
为 Heap_4 使用的数组设置起始地址 .....	16
Heap_5 .....	17
vPortDefineHeapRegions() API 函数 .....	17
与堆相关的实用程序函数 .....	20
xPortGetFreeHeapSize() API 函数 .....	20
xPortGetMinimumEverFreeHeapSize() API 函数 .....	20
Malloc 失败挂钩函数 .....	21
任务管理 .....	22
任务函数 .....	22
顶级任务状态 .....	23
创建任务 .....	23
xTaskCreate() API 函数 .....	23
创建任务 (示例 1) .....	24
使用任务参数 (示例 2) .....	28
任务优先级 .....	29
时间测量和时钟周期中断 .....	30
试用优先级 (示例 3) .....	31
“未运行”状态详述 .....	33
“被阻止”状态 .....	33
“暂停”状态 .....	33

“准备就绪”状态 .....	34
完成状态转换示意图 .....	34
使用“被阻止”状态创建延迟 (示例 4) .....	34
vTaskDelayUntil() API 函数 .....	38
将示例任务转换为使用 vTaskDelayUntil() (示例 5) .....	39
组合使用阻止和非阻止任务 (示例 6) .....	40
空闲任务和空闲任务挂钩 .....	41
空闲任务挂钩函数 .....	42
对实现空闲任务挂钩函数的限制 .....	42
定义空闲任务挂钩函数 (示例 7) .....	42
更改任务优先级 .....	44
vTaskPrioritySet() API 函数 .....	44
uxTaskPriorityGet() API 函数 .....	44
更改任务优先级 (示例 8) .....	45
删除任务 .....	48
vTaskDelete() API 函数 .....	48
删除任务 (示例 9) .....	48
计划算法 .....	51
任务状态和事件回顾 .....	51
配置计划算法 .....	51
带时间切片的固定优先级抢先计划 .....	52
固定优先级抢先计划 (无时间切片) .....	54
协作计划 .....	55
队列管理 .....	58
队列的特征 .....	58
数据存储 .....	58
由多个任务访问 .....	60
队列读取时阻止 .....	60
队列写入时阻止 .....	60
在多个队列中阻止 .....	60
使用队列 .....	61
xQueueCreate() API 函数 .....	61
xQueueSendToBack() 和 xQueueSendToFront() API 函数 .....	61
xQueueReceive() API 函数 .....	63
uxQueueMessagesWaiting() API 函数 .....	64
从队列接收时阻止 (示例 10) .....	64
接收多个源的数据 .....	68
向队列发送数据和通过队列发送结构时阻止 (示例 11) .....	68
处理大型或大小可变的数据 .....	73
指针排队 .....	73
使用队列来发送不同类型和长度的数据 .....	75
从多个队列接收 .....	78
队列集 .....	78
xQueueCreateSet() API 函数 .....	78
xQueueAddToSet() API 函数 .....	79
xQueueSelectFromSet() API 函数 .....	79
使用队列集 (示例 12) .....	80
更真实的队列集使用案例 .....	84
使用队列来创建邮箱 .....	85
xQueueOverwrite() API 函数 .....	86
xQueuePeek() API 函数 .....	87
软件计时器管理 .....	89
软件计时器回调函数 .....	89
软件计时器的属性和状态 .....	89
软件计时器的周期 .....	89
一次性和自动重新加载计时器 .....	90
软件计时器状态 .....	90

软件计时器上下文 .....	91
RTOS 守护程序 ( 计时器服务 ) 任务 .....	91
计时器命令队列 .....	91
守护程序任务计划 .....	92
创建并启动软件计时器 .....	94
xTimerCreate() API 函数 .....	94
xTimerStart() API 函数 .....	95
创建一次性和自动重新加载计时器 ( 示例 13 ) .....	96
计时器 ID .....	99
vTimerSetTimerID() API 函数 .....	99
pvTimerGetTimerID() API 函数 .....	99
使用回调函数参数和软件计时器 ID ( 示例 14 ) .....	99
更改计时器的周期 .....	101
xTimerChangePeriod() API 函数 .....	101
重置软件计时器 .....	104
xTimerReset() API 函数 .....	104
重置软件计时器 ( 示例 15 ) .....	105
中断管理 .....	108
中断安全 API .....	108
使用单独的中断安全 API 的优点 .....	108
使用单独的中断安全 API 的缺点 .....	109
xHigherPriorityTaskWoken 参数 .....	109
portYIELD_FROM_ISR() 和 portEND_SWITCHING_ISR() 宏 .....	110
委托中断处理 .....	111
用于同步的二进制信号灯 .....	112
xSemaphoreCreateBinary() API 函数 .....	114
xSemaphoreTake() API 函数 .....	114
xSemaphoreGiveFromISR() API 函数 .....	115
使用二进制信号灯使任务与中断同步 ( 示例 16 ) .....	116
改进示例 16 中所用任务的实现 .....	119
计数信号灯 .....	124
xSemaphoreCreateCounting() API 函数 .....	126
使用计数信号灯使任务与中断同步 ( 示例 17 ) .....	126
将工作委托给 RTOS 守护程序任务 .....	128
xTimerPendFunctionCallFromISR() API 函数 .....	128
集中式委托中断处理 ( 示例 18 ) .....	129
在中断服务例程内使用队列 .....	132
xQueueSendToFrontFromISR() 和 xQueueSendToBackFromISR() API 函数 .....	132
从 ISR 中使用队列时的注意事项 .....	133
从中断内部发送和接收队列 ( 示例 19 ) .....	133
中断嵌套 .....	137
ARM Cortex-M 和 ARM GIC 用户 .....	139
资源管理 .....	141
互斥 .....	143
关键部分和暂停计划程序 .....	143
基本关键部分 .....	143
暂停 ( 或锁定 ) 计划程序 .....	145
vTaskSuspendAll() API 函数 .....	145
xTaskResumeAll() API 函数 .....	145
互斥锁 ( 和二进制信号灯 ) .....	146
xSemaphoreCreateMutex() API 函数 .....	147
将 vPrintString() 重新编写为使用信号灯 ( 示例 20 ) .....	147
优先级反转 .....	150
优先级继承 .....	151
死锁 ( 或抱死 ) .....	151
递归互斥锁 .....	152
互斥锁和任务计划 .....	153

网关守卫任务 .....	156
将 vPrintString() 重新编写为使用网关守卫任务 (示例 21) .....	156
事件组 .....	161
事件组的特性 .....	161
事件组、事件标记和事件位 .....	161
关于 EventBits_t 数据类型的更多信息 .....	162
由多个任务访问 .....	60
使用事件组的实用示例 .....	162
使用事件组管理事件 .....	162
xEventGroupCreate() API 函数 .....	162
xEventGroupSetBits() API 函数 .....	163
xEventGroupSetBitsFromISR() API 函数 .....	163
xEventGroupWaitBits() API 函数 .....	164
试用事件组 (示例 22) .....	167
使用事件组同步任务 .....	171
xEventGroupSync() API 函数 .....	173
同步任务 (示例 23) .....	175
任务通知 .....	178
通过中间对象进行通信 .....	178
任务通知 : 直接到任务通信 .....	178
任务通知的优势和限制 .....	179
任务通知的限制 .....	179
使用任务通知 .....	180
任务通知 API 选项 .....	180
xTaskNotifyGive() API 函数 .....	180
vTaskNotifyGiveFromISR() API 函数 .....	180
ulTaskNotifyTake() API 函数 .....	181
使用任务通知代替信号灯的方法 1 (示例 24) .....	182
使用任务通知代替信号灯的方法 2 (示例 25) .....	185
xTaskNotify() 和 xTaskNotifyFromISR() API 函数 .....	187
xTaskNotifyWait() API 函数 .....	188
外围设备驱动程序中使用的任务通知 : UART 示例 .....	190
外围设备驱动程序中使用的任务通知 : ADC 示例 .....	195
直接在应用程序中使用的任务通知 .....	196
开发人员支持 .....	202
configASSERT() .....	202
示例 configASSERT() 定义 .....	202
Tracealyzer .....	203
调试相关的挂钩 (回调) 函数 .....	206
查看运行时和任务状态信息 .....	206
任务运行时统计数据 .....	206
运行时统计数据时钟 .....	206
将应用程序配置为收集运行时统计数据 .....	207
uxTaskGetSystemState() API 函数 .....	207
vTaskList() 帮助程序函数 .....	209
vTaskGetRunTimeStats() 帮助程序函数 .....	210
生成和显示运行时统计数据 , 一个工作示例 .....	211
跟踪挂钩宏 .....	213
可用的跟踪挂钩宏 .....	214
定义跟踪挂钩宏 .....	215
FreeRTOS 感知调试程序插件 .....	215
疑难解答 .....	217
本章简介和范围 .....	217
中断优先级 .....	217
堆栈溢出 .....	217
uxTaskGetStackHighWaterMark() API 函数 .....	218
运行时堆栈检查概述 .....	218

运行时堆栈检查方法 1 .....	218
运行时堆栈检查方法 2 .....	219
不当使用 printf() 和 sprintf() .....	219
Printf-stdarg.c .....	219
其他常见错误 .....	219
症状：向演示添加简单任务导致演示崩溃 .....	219
症状：在中断中使用 API 函数导致应用程序崩溃 .....	220
症状：应用程序有时在中断服务例程中崩溃 .....	220
症状：计划程序在尝试启动第一个任务时崩溃 .....	220
症状：中断被意外保持为禁用或关键部分嵌套不正确 .....	220
症状：应用程序甚至在计划程序启动之前就崩溃 .....	220
症状：在计划程序暂停时调用 API 函数或从关键部分内部调用 API 函数导致应用程序崩溃 .....	221

# 关于 FreeRTOS 内核

FreeRTOS 内核是由 Amazon 维护的一个开源软件。

FreeRTOS 内核非常适用于使用微控制器或小型微处理器的深度嵌入式实时应用程序。这种类型的应用程序通常包括软/硬实时要求混用的情况。

软实时要求规定了一个时间截止期限，但违反此截止期限不会使系统呈现无用状态。例如，对击键操作响应过慢可能会使系统看起来不响应而烦人，但并不会真的使其变得无法使用。

硬实时要求规定了一个时间截止期限，但违反此截止期限确实会导致系统完全失败。例如，如果驾驶员安全气囊对碰撞传感器输入的响应过慢，则其害处可能高于益处。

FreeRTOS 内核是一个实时内核（或实时计划程序），可以在其基础之上构建嵌入式应用程序以满足硬实时要求。借助于此内核，可以将应用程序组织成独立执行线程的集合。在只有一个核心的处理器上，在任一时间只能执行一个线程。内核通过检查应用程序设计人员分配给每个线程的优先级来决定哪个线程应执行。在最简单的情况下，应用程序设计人员可以向实现硬实时要求的线程分配较高优先级，而向实现软实时要求的线程分配较低优先级。这可确保硬实时线程始终比软实时线程先执行，但优先级分配并非始终如此简单。

如果您尚不了解上一段落中的概念，请勿担心。本指南将详细介绍这些内容并提供许多示例，以帮助您了解如何使用实时内核，尤其是 FreeRTOS 内核。

## 价值主张

FreeRTOS 内核在全球取得空前的成功源自于其具有吸引力的价值主张。FreeRTOS 内核经过专业化开发和严格的质量控制、稳定且享有支持服务，知识产权非常明晰，且真正免费用于商业应用程序中而不要求您披露专有源代码。您可以使用 FreeRTOS 内核将产品推向市场，而无需支付任何费用，成千上万的人就是这样做的。在任何时候，如果您希望收到额外备份，或者您的法律团队需要额外的书面保证或保障，则可以采用一个简单的低成本商业升级路径。因为您知道可以随时选择采用这一商业途径，所以您可以安心使用。

## 关于术语的注释

在 FreeRTOS 内核中，每个执行线程都称为任务。尽管在嵌入式社区中对术语并没有达成共识，但线程在应用程序的某些领域具有更特定的含义。

## 为什么使用实时内核？

要编写出色的嵌入式软件，可以采用许多非常成熟的方法而不使用内核；并且，如果正在开发的系统较为简单，这些方法可能会提供最适合的解决方案。在较复杂的情况下，使用内核会更好些，但究竟复杂到什么程度应该优先使用内核则始终凭主观确定。

除了通过任务优先级帮助确保应用程序满足其处理截止期限，内核还可以带来其他一些不太明显的优势：

- 将计时信息从应用程序中抽离出来

内核负责执行计时，并向应用程序提供与时间相关的 API。这可让应用程序代码的结构更为简单，总体代码大小更小。

- 可维护性/可扩展性

将计时详细信息从应用程序中抽离出来会导致模块之间的相互依赖性降低，并让软件能够以可控且可预测的方式发展。此外，由于内核负责计时，因此应用程序性能不太易受底层硬件更改的影响。

- 模块性

任务是独立的模块，每个模块都应有定义良好的用途。

- 团队开发

任务还应具有定义良好的接口，从而让团队更轻松地进行开发。

- 更易于测试

如果任务是定义良好的独立模块且具有整洁的接口，则可以在隔离环境中对它们进行测试。

- 代码重用

由于模块性更高且依赖关系更少，因此可以轻松地重用代码。

- 效率提高

使用内核可允许软件完全由事件驱动，因此，不会由于轮询尚未发生的事件而浪费处理时间。仅当有任务必须完成时，才执行代码。

虽然该方法需要处理 RTOS 计时中断和切换任务执行（从一个任务切换到另一个任务），因此会影响效率，但是，不利用 RTOS 的应用程序也通常包含某种形式的计时中断。

- 闲置时间

当启动计划程序时，会自动创建闲置任务。只要没有要执行的应用程序任务，就会执行闲置任务。闲置任务可用于衡量备用处理容量，执行背景检查，或只是将处理器置于低能耗模式。

- 电源管理

使用 RTOS 获得的效率收益让处理器可以在更多时间处于低功耗模式。

通过在每次运行闲置任务时将处理器置于低功耗状态，可以显著降低功耗。FreeRTOS 内核还具有非计时模式，可让处理器进入并保留在低功耗模式下更长时间。

- 灵活的中断处理

通过将处理委托给由应用程序编写者创建的任务或 FreeRTOS 守护程序任务，可以将中断处理程序保留非常短的时间。

- 混合处理要求

简单设计模式可以在应用程序内实现定期、连续和事件驱动型处理的混合。此外，可以通过选择适当的任务和中断优先级来满足硬实时要求和软实时要求。

## FreeRTOS 内核功能

FreeRTOS 内核具有以下标准功能：

- 优先操作或合作操作
- 非常灵活的任务优先级分配
- 灵活、快速和轻量级任务通知机制
- 队列
- 二进制信号灯
- 计数信号灯
- 互斥对象
- 递归互斥
- 软件计时器
- 事件组
- 计时挂钩函数
- 闲置挂钩函数
- 堆栈溢出检查
- 跟踪记录
- 任务运行时统计数据收集
- 可选的商业许可和支持
- 完全中断嵌套模型（对于某些架构）
- 极低功耗应用程序的非计时功能
- 适时提供的软件管理的中断堆栈（这可帮助节省 RAM）

## 许可

FreeRTOS 内核按照 MIT 许可证的条款向用户提供。

## 包含的源文件和项目

在随附的 zip 文件中为所有展示的示例提供了源代码、预配置的项目文件和完整的构建说明。如果您没有随本手册收到一份该 ZIP 文件，可以从 <http://www.FreeRTOS.org/Documentation/code> 下载。该 zip 文件可能不包含最新版本的 FreeRTOS 内核。

本手册中包含的屏幕截图是在 Microsoft Windows 环境中使用 FreeRTOS Windows 端口执行这些示例时截取的。使用 FreeRTOS Windows 端口的项目预配置为使用免费的 Visual Studio Express 版本生成，您可以从 <https://www.visualstudio.com/vs/community/> 下载此版本。尽管 FreeRTOS Windows 端口提供了方便的评估、测试和开发平台，但它不提供真正的实时行为。

# FreeRTOS 内核分发版

FreeRTOS 内核以单个 zip 文件存档形式分发，其中包含所有 FreeRTOS 官方内核端口和大量预配置的演示应用程序。

## 了解 FreeRTOS 内核分发版

FreeRTOS 可以使用大约 20 种不同的内核编译器进行构建，并可以在 30 多种不同的处理器架构上运行。每个受支持的编译器和处理器组合被视为一个单独的 FreeRTOS 端口。

## 构建 FreeRTOS 内核

您可以将 FreeRTOS 看作一个库，它可向原本是裸机应用程序的内容提供多任务处理功能。

FreeRTOS 作为一组 C 语言源文件提供。一些源文件是所有端口通用的，而其他一些源文件特定于端口。将源文件构建为您的项目的一部分，以使 FreeRTOS API 可用于您的应用程序。为便于操作，每个官方 FreeRTOS 端口都提供有演示应用程序。演示应用程序预配置为构建正确的源文件并包含正确的标头文件。

演示应用程序应该是现成可构建的。但是，演示发布之后对构建工具进行的更改可能会导致问题。有关更多信息，请参阅本主题后面的“演示应用程序”。

## FreeRTOSConfig.h

FreeRTOS 通过称为 FreeRTOSConfig.h 的标头文件进行配置。

FreeRTOSConfig.h 用于定制 FreeRTOS 以用于特定应用程序。例如，FreeRTOSConfig.h 包含如 configUSEPREEMPTION 等常量，其设置定义是使用合作计划算法，还是使用优先计划算法。FreeRTOSConfig.h 包含特定于应用程序的定义，因此它应位于作为所构建的应用程序的一部分的目录中，而不位于包含 FreeRTOS 源代码的目录中。

为每个 FreeRTOS 端口都提供了一个演示应用程序，每个演示应用程序都包含一个 FreeRTOSConfig.h 文件。因此，您从不需要从头开始创建 FreeRTOSConfig.h 文件。相反，我们建议您从为正在使用的 FreeRTOS 端口提供的演示应用程序所用的 FreeRTOSConfig.h 着手并进行改写。

## 官方 FreeRTOS 内核分发版

FreeRTOS 以单个 zip 文件的方式分发。该 zip 文件包含所有 FreeRTOS 端口的源代码和所有 FreeRTOS 演示应用程序的项目文件。它还包含一些精选的 FreeRTOS+ 生态系统组件和 FreeRTOS+ 生态系统演示应用程序。

不必担心 FreeRTOS 分发版中的文件数量。任何应用程序都只需少量数目的文件。

## FreeRTOS 分发版中的顶级目录

以下是 FreeRTOS 分发版的第一级和第二级目录及说明。

FreeRTOS

| |

| |└Source 目录，包含 FreeRTOS 源文件

| |

| |└Demo 目录，包含预配置且特定于端口的 FreeRTOS 演示项目

| |

FreeRTOS-Plus

| |

| |└Source 目录，包含一些 FreeRTOS + 生态系统组件的源代码

| |

| |└Demo 目录，包含 FreeRTOS+ 生态系统组件的演示项目

该 zip 文件仅包含 FreeRTOS 源文件的一份副本、所有 FreeRTOS 演示项目以及所有 FreeRTOS+ 演示项目。您应该可以在 FreeRTOS/Source 目录中找到 FreeRTOS 源文件。如果目录结构发生变化，则文件可能无法构建。

## 所有端口通用的 FreeRTOS 源文件

核心 FreeRTOS 源代码只包含在两个 C 语言文件中，这些文件是所有 FreeRTOS 端口通用的。这两个文件分别称为 tasks.c 和 list.c。它们直接位于 FreeRTOS/Source 目录中。以下源文件位于同一个目录中：

- queue.c

queue.c 同时提供队列和信号灯服务。queue.c 几乎始终是必需的。

- timers.c

timers.c 提供软件计时器功能。仅当要使用软件计时器时，才将其包含在构建过程中。

- eventgroups.c

eventgroups.c 提供事件组功能。仅当要使用事件组时，才将其包含在构建过程中。

- croutine.c

croutine.c 实现 FreeRTOS 协同例程功能。仅当要使用协同例程时，才将其包含在构建过程中。协同例程旨在用于非常小的微控制器上。现在它们很少使用，因此没有按与其他 FreeRTOS 功能相同的级别进行维护。本指南中未涉及协同例程。

FreeRTOS

| |

| |└Source

| |

| |└tasks.c FreeRTOS 源文件 - 始终是必需的

| |└list.c FreeRTOS 源文件 - 始终是必需的

```
|---queue.c FreeRTOS 源文件 - 几乎始终是必需的  
|---timers.c FreeRTOS 源文件 - 可选  
|---eventgroups.c FreeRTOS 源文件 - 可选  
└---croutine.c FreeRTOS 源文件 - 可选
```

我们认识到文件名可能会导致名称空间冲突，因为许多项目将已经包含同名的文件。但是，现在更改文件名可能会带来很多问题，因为这样做会破坏数以千计使用 FreeRTOS 以及自动化工具和 IDE 插件的项目的兼容性。

## 特定于端口的 FreeRTOS 源文件

特定于 FreeRTOS 端口的源文件包含在 FreeRTOS/Source/portable 目录中。portable 目录按层次结构排列，首先按编译器，然后按处理器架构。

如果您在架构为“架构”的处理器上使用编译器“编译器”运行 FreeRTOS，则除了核心 FreeRTOS 源文件外，您还必须构建位于 FreeRTOS/Source/portable/[编译器]/[架构] 目录中的文件。

如将在第 2 章“堆内存管理”中介绍的那样，FreeRTOS 还将堆内存分配视为可移植层的一部分。使用早于 V9.0.0 的 FreeRTOS 版本的项目必须包含堆内存管理器。从 FreeRTOS V9.0.0 开始，仅当在 FreeRTOSConfig.h 中将 configSUPPORTDYNAMICALLOCATION 设置为 1 或者将 configSUPPORTDYNAMICALLOCATION 保留为未定义时，才需要堆内存管理器。

FreeRTOS 提供五个示例堆分配方案。五个方案的名称为 heap1 到 heap5，分别由源文件 heap1.c 到 heap5.c 实现。这些示例堆分配方案包含在 FreeRTOS/Source/portable/MemMang 目录中。如果您已将 FreeRTOS 配置为使用动态内存分配，则需要在项目中构建这五个源文件之一，除非您的应用程序提供替代实现。

下图显示 FreeRTOS 目录树中特定于端口的源文件。

```
FreeRTOS  
|  
└---Source  
|  
└---portable 目录，包含所有特定于端口的源文件  
|  
└---MemMang 目录，包含 5 个替代堆分配源文件  
|  
└---[编译器 1] 目录，包含特定于编译器 1 的端口文件  
| |  
|   |---[架构 1] 包含编译器 1 架构 1 端口的文件  
|   |---[架构 2] 包含编译器 1 架构 2 端口的文件  
|   |---[架构 3] 包含编译器 1 架构 3 端口的文件  
| |
```

└ [编译器 2] 目录，包含特定于编译器 2 的端口文件

|

└ [架构 1] 包含编译器 2 架构 1 端口的文件

└ [架构 2] 包含编译器 2 架构 2 端口的文件

└ [等等]

包含路径

FreeRTOS 要求将三个目录加入编译器的包含路径中：

1. 核心 FreeRTOS 标头文件的路径，始终是 FreeRTOS/Source/include。
2. 特定于所用 FreeRTOS 端口的源文件的路径。如上所述，这是 FreeRTOS/Source/portable/[编译器]/[架构]。
3. FreeRTOSConfig.h 标头文件的路径。

## 标头文件

使用 FreeRTOS API 的源文件必须包括“FreeRTOS.h”，后跟标头文件，其中包括所用 API 函数的原型—“task.h”、“queue.h”、“semphr.h”、“timers.h”或“eventgroups.h”。

## 演示应用程序

每个 FreeRTOS 端口附带了至少一个演示应用程序，演示应用程序构建时应该不会生成错误或警告；尽管一些演示比其他演示更早，但有时自演示发布后对构建工具进行的更改可能会导致出现问题。

Linux 用户请注意：FreeRTOS 是在 Windows 主机上开发和测试的。有时，这会导致在 Linux 主机上构建演示项目时出现构建错误。构建错误几乎都与引用文件名时所用的字母大小写或文件路径中使用的斜杠字符方向有关。请使用 FreeRTOS 联系表格 (<http://www.FreeRTOS.org/contact>) 向我们提醒任何此类错误。

演示应用程序具有以下几个作用：

- 提供一个有效的预配置项目示例，其中包含正确的文件且设置了正确的编译器选项。
- 允许通过极少的设置或先前学到的知识获得开箱即用体验。
- 演示如何可以使用 FreeRTOS API。
- 作为从中创建真实应用程序的基础。

每个演示项目都位于 FreeRTOS/Demo 目录下的一个唯一子目录中。子目录的名称指示演示项目所关联的端口。

每个演示应用程序也由 FreeRTOS.org 网站上的一个网页进行描述。该网页包含有关以下各项的信息：

- 如何在 FreeRTOS 目录结构中找到演示的项目文件。
- 项目配置为使用哪种硬件。
- 如何设置硬件来运行演示。
- 如何构建演示。
- 演示的预期行为。

所有演示项目都会创建一些常见演示任务，它们的实现包含在 FreeRTOS/Demo/Common/Minimal 目录中。常见演示任务的作用纯粹是为了演示如何可以使用 FreeRTOS API — 它们不实现任何特定的有用功能。

更近期的演示项目也可以构建一个初学者“blinky”项目。Blinky 项目是非常基本的。通常情况下，它们将仅创建两个任务和一个队列。

每个演示项目包含一个名为 main.c 的文件。这包含 main() 函数，所有演示应用程序任务都从中创建。有关特定于该演示的信息，请参阅各 main.c 文件中的注释。

下面的内容显示了 FreeRTOS/Demo 目录层次结构。

FreeRTOS

|

└─Demo 目录，包含所有演示项目

|

  └─[演示 x] 包含构建演示“x”的项目文件

|

  └─[演示 y] 包含构建演示“y”的项目文件

|

  └─[演示 z] 包含构建演示“z”的项目文件

|

└─Common 目录，包含由所有演示应用程序构建的文件

## 创建 FreeRTOS 项目

每个 FreeRTOS 端口附带至少一个预配置的演示应用程序，构建时应无错误或警告。建议通过改写这些现有项目来创建新项目；这样，项目就包含了正确的文件，安装了正确的中断处理程序，并设置了正确的编译器选项。

要从现有演示项目开始新的应用程序请执行以下操作：

1. 打开提供的演示项目并确保它按预期方式构建和执行。
2. 删除定义演示任务的源文件。可以从项目中删除任何位于 Demo/Common 目录中的文件。
3. 删除 main() 中的所有函数调用，但 prvSetupHardware() 和 vTaskStartScheduler() 除外，如列表 1 中所示。
4. 检查此项目仍然构建。

按照这些步骤操作将创建一个项目，其中包含正确的 FreeRTOS 源文件，但不定义任何功能。

```
int main( void )
{
    /* Perform any hardware setup necessary. */

    prvSetupHardware();

    /* --- APPLICATION TASKS CAN BE CREATED HERE --- */

    /* Start the created tasks running. */
```

```
vTaskStartScheduler();

/* Execution will only reach here if there was insufficient heap to start the scheduler. */

for( ; ; );
return 0;
}
```

## 从头开始创建新项目

如前所述，建议从现有的演示项目创建新项目。如果不这样做，则可以使用以下过程创建一个新项目：

1. 使用您选择的工具链创建一个新项目，其中尚未包含任何 FreeRTOS 源文件。
2. 确保可以构建新项目，并可将其下载到您的目标硬件中执行。
3. 仅当您确定您已有一个有效的项目时，才按照表 1 中的详细说明将 FreeRTOS 源文件添加到项目中。
4. 将为所用端口提供的演示项目中使用的 FreeRTOSConfig.h 标头文件复制到项目目录中。
5. 将以下目录添加到项目将在其中查找标头文件的路径：
  - FreeRTOS/Source/include
  - FreeRTOS/Source/portable/[编译器]/[架构] ( 其中，  
[编译器] 和 [架构] 对应于您选择的端口 )
  - 包含 FreeRTOSConfig.h 标头文件的目录
1. 从相关的演示项目中复制编译器设置。
2. 安装任何可能需要的 FreeRTOS 中断处理程序。使用介绍所用端口的网页和为所用端口提供的演示项目作为参考。

下表列出了要包含在项目中的 FreeRTOS 源文件。

File	位置
tasks.c	FreeRTOS/Source
queue.c	FreeRTOS/Source
list.c	FreeRTOS/Source
timers.c	FreeRTOS/Source
eventgroups.c	FreeRTOS/Source
所有 C 语言和汇编器文件	FreeRTOS/Source/portable/[编译器]/[架构]
heapn.c	FreeRTOS/Source/portable/MemMang , 其中 n 是 1、2、3、4 或 5。此文件从 FreeRTOS V9.0.0 开 始成为可选的。

使用早于 V9.0.0 的 FreeRTOS 版本的项目必须构建其中一个 heapn.c 文件。从 FreeRTOS V9.0.0 开始，仅当在 FreeRTOSConfig.h 中将 configSUPPORTDYNAMICALLOCATION 设置为 1 或者将

configSUPPORTDYNAMICALLOCATION 保留为未定义时，才需要 heapn.c 文件。有关更多信息，请参阅第 2 章“堆内存管理”。

## 数据类型和编码样式指南

FreeRTOS 的每个端口都具有唯一的 portmacro.h 标头文件，该文件除包含其他内容外，还包含以下两个特定于端口的数据类型的定义：TickType 和 BaseType。

下表列出了 FreeRTOS 所使用的数据类型。

一些编译器使所有非限定 char 变量成为未签名变量，而其他一些编译器使这些变量成为已签名变量。因此，FreeRTOS 源代码显式限定“签名”或“未签名”char 的每次使用，除非相应 char 用于保留 ASCII 字符或指向 char 的指针用于指向字符串。

从不使用 Plain int 类型。

## 变量名称

变量的前缀是其类型：“c”表示 char、“s”表示 int16t (short)、“l”表示 int32t (long)、“x”表示 BaseType 和任何其他非标准类型（结构、任务句柄、队列句柄等）。

如果某个变量未签名，则其前缀中还有一个“u”。如果某个变量是指针，则其前缀中还有一个“p”。例如，类型为 uint8t 的变量将以“uc”为前缀，而类型为 char 指针的变量将以“pc”为前缀。

## 函数名称

函数的前缀是它们返回的类型和在其中定义它们的文件。例如：

- vTaskPrioritySet() 返回 Void 并在 task.c 内定义。
- xQueueReceive() 返回类型为 BaseType\_t 的变量并在 queue.c 内定义。
- pvTimerGetTimerID() 返回指向 Void 的指针并在 timers.c 内定义。

文件作用域（私有）函数的前缀为“prv”。

## 格式设置

一个制表符始终设置为等于四个空格。

## 宏名称

大多数宏以大写格式编写，前缀采用小写字母以指示定义宏的位置。

下表列举了宏前缀。

前缀	宏定义的位置
----	--------

port ( 例如 , portMAXDELAY )	portable.h 或 portmacro.h
task ( 例如 , taskENTERCRITICAL() )	task.h
pd ( 例如 , pdTRUE )	projdefs.h
config ( 例如 , configUSEPREEMPTION )	FreeRTOSConfig.h
err ( 例如 , errQUEUEFULL )	projdefs.h

信号灯 API 几乎完全编写为一组宏，但遵循函数命名惯例，而不是宏命名约定。

下表列出了 FreeRTOS 源代码中使用的宏。

宏	值
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

## 过量类型强制转换的基本原理

可以使用许多不同的编译器编译 FreeRTOS 源代码，所有编译器在生成警告的方式和时间方面都不同。尤其是，不同的编译器需要以不同的方式使用强制转换。因此，FreeRTOS 源代码包含的类型强制转换比通常保证的多。

# 堆内存管理

从 V9.0.0 开始，FreeRTOS 应用程序可以完全静态分配，这意味着无需包含堆内存管理器。

本节内容：

- FreeRTOS 何时分配 RAM。
- FreeRTOS 随附的五个示例内存分配方案。
- 每个内存分配方案的使用案例。

## 先决条件

您需要充分了解 C 语言编程以便使用 FreeRTOS。具体来说，您应该熟悉：

- 如何构建 C 语言项目，包括编译和链接阶段。
- 堆栈和堆的概念。
- 标准 C 语言库函数 malloc() 和 free()。

## 动态内存分配及其与 FreeRTOS 的关联性

本指南介绍内核对象，如任务、队列、信号灯和事件组。要让 FreeRTOS 对象变得尽可能易于使用，这些内核对象不是在编译时静态分配，而是在运行时动态分配。每次创建内核对象时，FreeRTOS 都会分配 RAM；每次删除内核对象时，都会释放 RAM。此策略可减少设计和规划工作量，简化了 API，并最大程度地减少 RAM 开销。

动态内存分配是一个 C 语言编程概念。它不是特定于 FreeRTOS 或多任务处理的概念。它与 FreeRTOS 相关，因为内核对象是动态分配的，通用编译器提供的动态内存分配方案并非始终适合实时应用程序。

可以使用标准 C 语言库函数 malloc() 和 free() 分配内存，但由于以下一个或多个原因，这些函数可能不适合或不适用：

- 它们并非始终适用于小型嵌入式系统。
- 它们的实现可能相当大，占用宝贵的代码空间。
- 它们极少是线程安全的。
- 它们不是确定性的。执行函数所花的时间量将因调用不同而异。
- 它们可能会碎片化。如果堆中的可用 RAM 被拆分为若干较小且彼此分离的块，则认为堆已碎片化。当堆已碎片化时，如果堆中没有一个可用块的大小足以包含某个数据块，则尝试分配此数据块时将失败，即使堆中所有单独块的总大小比无法分配的数据块大小大很多倍，也是如此。
- 它们可能使链接器配置变得更复杂。
- 如果允许堆空间增长而占用了其他变量使用的内存，则这些函数可能成为一些难以调试的错误的来源。

## 动态内存分配的选项

早期版本的 FreeRTOS 使用内存池分配方案，在编译时预分配由不同大小的内存块组成的池，然后由内存分配函数返回。虽然这是实时系统中常用的方法，但它产生了大量支持请求。由于该方法使用 RAM 的效率无法满足非常小型的嵌入式系统的需要，因此已被放弃。

FreeRTOS 现在将内存分配视为可移植层的一部分（而不是核心代码库的一部分）。这样做的原因是已认识到嵌入式系统具有变化多端的动态内存分配和计时要求。单个动态内存分配算法仅适用于一部分应用程序。此外，通过从核心代码库中删除动态内存分配，应用程序编写者可以适时提供其自己的特定实现。

当 FreeRTOS 需要 RAM 时，它调用 `pvPortMalloc()` 而不是 `malloc()`。释放 RAM 时，内核调用 `vPortFree()`，而不是 `free()`。`pvPortMalloc()` 与标准 C 语言库函数 `malloc()` 具有相同的原型。`vPortFree()` 与标准 C 语言库函数 `free()` 具有相同的原型。

`pvPortMalloc()` 和 `vPortFree()` 是公共函数，因此，也可以从应用程序代码中调用它们。

FreeRTOS 附带了 `pvPortMalloc()` 和 `vPortFree()` 的五个示例实现，本指南将一一介绍。FreeRTOS 应用程序可以使用其中一个示例实现或提供自己的实现。

这五个示例在 `heap_1.c`、`heap_2.c`、`heap_3.c`、`heap_4.c` 和 `heap_5.c` 源文件中定义，这些源文件位于 `FreeRTOS/Source/portable/MemMang` 目录中。

## 示例内存分配方案

因为 FreeRTOS 应用程序可以完全静态分配，您无需包含堆内存管理器。

### Heap\_1

它常用于小型专用嵌入式系统，以便仅在启动计划程序之前创建任务和其他内核对象。在应用程序开始执行任何实时功能之前，内核动态分配内存，并且内存存在应用程序的生命周期内保持已分配状态。这意味着所选分配方案不必考虑任何更复杂的内存分配问题（例如确定性和碎片化），而是可以考虑如代码大小和简单性等属性。

`Heap_1.c` 实现 `pvPortMalloc()` 的一个非常基本的版本。它不实现 `vPortFree()`。从不删除任务或其他内核对象的应用程序可以使用 `heap_1`。

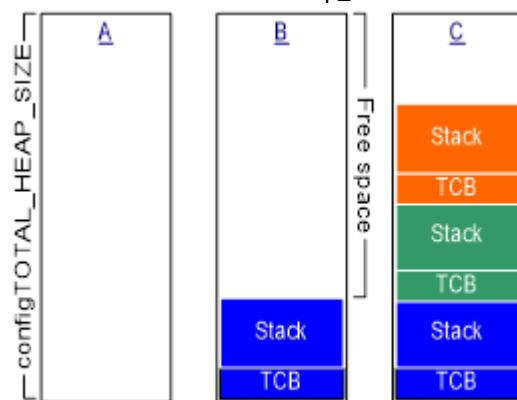
某些商业关键和安全关键型系统可能禁止动态内存分配，这些系统也可能能够使用 `heap_1`。由于存在与非确定性、内存碎片化以及分配失败等相关的不确定性，因此，关键系统通常禁止动态内存分配，但 `heap_1` 始终是确定性的且无法对内存进行碎片化。

当调用 `pvPortMalloc()` 时，`heap_1` 分配方案将一个简单的数组细分成更小的块。此数组称为 FreeRTOS 堆。

数组的总大小（以字节为单位）由定义 `configTOTAL_HEAP_SIZE` 在 `FreeRTOSConfig.h` 中设置。以这种方式定义大型数组可能让应用程序看起来会消耗大量 RAM，甚至从数组中分配任何内存之前就是如此。

每个创建的任务都要求从堆中分配一个任务控制块 (TCB) 和一个堆栈。

下图显示了在创建任务时 `heap_1` 如何细分简单的数组。每次创建任务时，都会从 `heap_1` 数组分配 RAM。



- A 显示创建任何任务之前的数组。整个数组都可用。

- B 显示已创建一个任务后的数组。
- C 显示已创建三个任务后的数组。

## Heap\_2

Heap\_2 包含在 FreeRTOS 发行版中以保持向后兼容性。建议不要用于新设计，而是考虑使用 heap\_4，因为其中提供了更多功能。

也可以使用 Heap\_2.c，但要细分由 configTOTAL\_HEAP\_SIZE 确定大小的数组。它使用最适合算法分配内存。与 heap\_1 不同，它允许释放内存。再次说明，数组是静态声明的，因此应用程序看起来会消耗大量 RAM，甚至在从数组中分配任何内存之前就是如此。

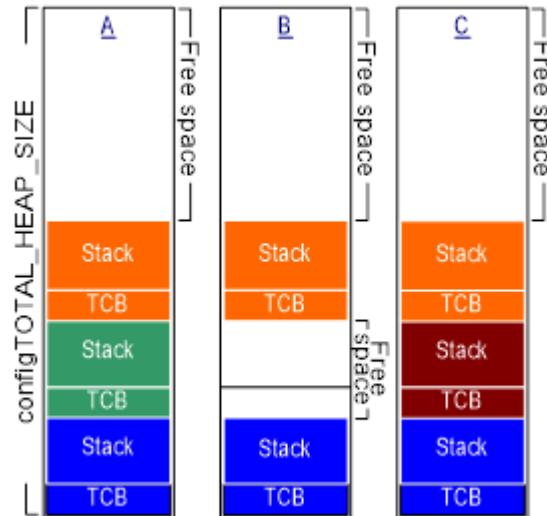
最适合算法可确保 pvPortMalloc() 使用的可用内存块在大小方面与所要求的字节数最接近。例如，考虑以下情形：

- 堆包含三个可用内存块，大小分别为 5 字节、25 字节和 100 字节。
- 调用 pvPortMalloc() 以请求 20 字节的 RAM。

适合所请求字节数的最小可用 RAM 块是 25 字节块，因此，pvPortMalloc() 将 25 字节块拆分成一个 20 字节块和一个 5 字节块，然后返回一个指向 20 字节块的指针。（上面是过于简化了，因为 heap\_2 要存储堆区域中块大小的信息，因此两个拆分块的总和实际上将小于 25。）新的 5 字节块保持可用于将来对 pvPortMalloc() 的调用。

与 heap\_4 不同，heap\_2 不将相邻的可用块合并为单个更大的块。因此，它更容易碎片化。但是，如果分配的块与后续释放的块大小始终相同，则碎片化不是问题。Heap\_2 适合反复创建和删除任务的应用程序，但前提是分配给所创建的任务的堆栈大小不发生变化。

下图显示在创建和删除任务时从 heap\_2 数组中分配和释放 RAM 的过程。



图中显示当创建、删除以及后续再次创建任务时，最适合算法的工作原理。

- A 显示已创建三个任务后的数组。大型可用块保持在数组的顶部。
- B 显示已删除其中一个任务后的数组。这些区域有：

数组顶部的大型可用块保持原样。此外，目前有两个较小的可用块，它们之前分配给了已删除任务的 TCB 和堆栈。

- C 显示已创建另一个任务后的数组。创建

任务导致两次调用 `pvPortMalloc()`：一次是分配新的 TCB，一次是分配任务堆栈。使用 `xTaskCreate()` API 函数创建任务，如[创建任务 \(p. 23\)](#)中所述。在 `xTaskCreate()` 内部发生两次 `pvPortMalloc()` 调用。

每个 TCB 的大小完全相同，因此，最适合算法可确保之前分配给已删除任务的 TCB 的 RAM 块可重用于分配新任务的 TCB。

分配给新创建任务的堆栈大小与分配给以前被删除任务的堆栈大小完全相同，因此，最适合算法可确保之前分配给已删除任务的堆栈的 RAM 块可重用于分配新任务的堆栈。

数组顶部的较大的未分配块保持不变。

Heap\_2 是非确定性的，但它比 `malloc()` 和 `free()` 的大多数标准库实现都要快。

## Heap\_3

Heap\_3.c 使用标准库函数 `malloc()` 和 `free()`，因此堆大小由链接器配置定义。`configTOTAL_HEAP_SIZE` 设置不起作用。

Heap\_3 通过临时暂停 FreeRTOS 计划程序使 `malloc()` 和 `free()` 成为线程安全的。有关线程安全和计划程序暂停的信息，请参阅[资源管理 \(p. 141\)](#)部分。

## Heap\_4

与 heap\_1 和 heap\_2 一样，heap\_4 将数组细分成较小的块。数组是静态声明的并由 `configTOTAL_HEAP_SIZE` 确定大小，因此应用程序看起来会消耗大量 RAM，甚至在从数组中分配任何内存之前就是如此。

Heap\_4 使用首个适合算法分配内存。与 heap\_2 不同，它会将相邻的可用内存块组合（合并）成一个较大的块。这样可以最大程度地减小内存碎片化风险。

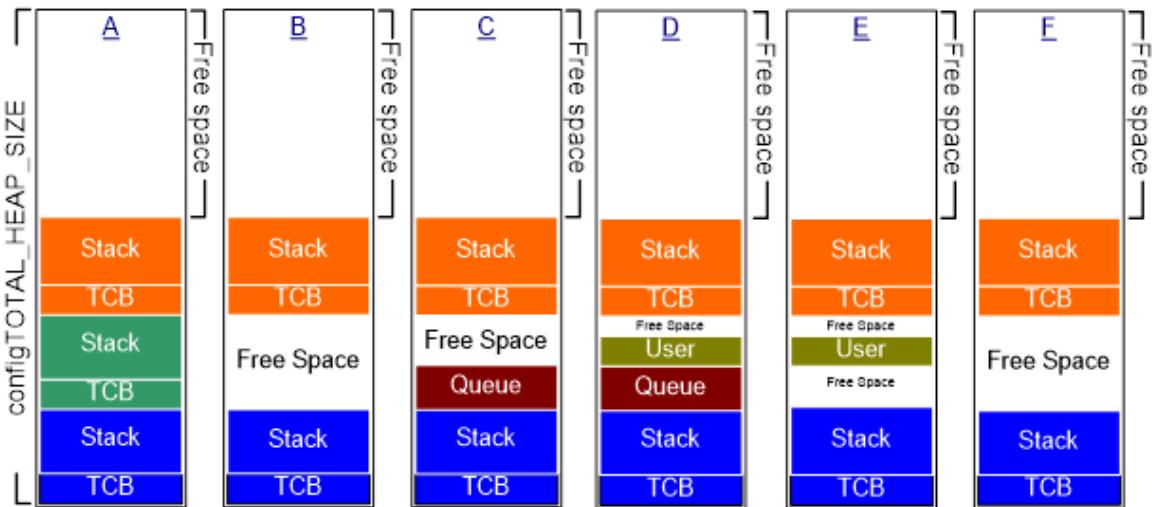
首个适合算法可确保 `pvPortMalloc()` 使用第一个大小足以容纳所要求的字节数的可用内存块。例如，考虑以下情形：

- 堆包含三个可用内存块。它们在数组中以下面的顺序显示：5 字节、200 字节和 100 字节。
- 调用 `pvPortMalloc()` 以请求 20 字节的 RAM。

适合所请求字节数的第一个可用 RAM 块是 200 字节块，因此，`pvPortMalloc()` 将 200 字节块拆分成一个 20 字节块和一个 180 字节块，然后返回一个指向 20 字节块的指针。（上面是过于简化了，因为 heap\_4 要存储堆区域中块大小的信息，因此两个拆分块的总和将小于 200 字节。）新的 180 字节块保持可用于将来对 `pvPortMalloc()` 的调用。

Heap\_4 会将相邻的可用内存块组合（合并）成一个较大的块，同时最大限度地降低碎片化风险。Heap\_4 适用于反复分配和释放不同大小的 RAM 块的应用程序。

下图显示从 heap\_4 数组中分配和释放 RAM 的过程。它演示 heap\_4 首个适合算法（带内存合并）在分配和释放内存时的工作方式。



A 显示已创建三个任务后的数组。大型可用块保持在数组顶部。

B 显示已删除其中一个任务后的数组。数组顶部的大型可用块保持原样。还有一个可用块，之前分配给了已删除任务的 TCB 和堆栈。删除 TCB 时释放的内存和删除堆栈释放的内存不会保留为两个单独的可用块，而是合并以创建一个更大的单个可用块。

C 显示已创建一个 FreeRTOS 队列后的数组。队列是使用 `xQueueCreate()` API 函数创建的，如[使用队列 \(p. 61\)](#)中所述。`xQueueCreate()` 调用 `pvPortMalloc()` 以分配队列使用的 RAM。由于 `heap_4` 使用首个适合算法，因此 `pvPortMalloc()` 从第一个大小足以容纳队列的可用 RAM 块中分配 RAM。在图中，这是删除任务时释放的 RAM。该队列不消耗可用块中的所有 RAM，因此块一分为二。未使用的部分保持可用于将来对 `pvPortMalloc()` 的调用。

D 显示在从应用程序代码中直接调用 `pvPortMalloc()` (而不是通过间接调用 FreeRTOS API 函数) 后的数组。用户分配的块足够小，第一个可用块(介于分配给队列的内存与分配给上方 TCB 的内存之间的块)就可以满足。删除任务时释放的内存现在已拆分成三个单独的块。第一个块容纳队列。第二个块容纳用户分配的内存。第三个保持可用。

E 显示删除队列后的数组，此时会自动释放分配给已删除队列的内存。此时，用户分配的块的两侧都有可用内存。

F 显示的也是已释放用户分配的内存之后的数组。用户分配的块所用的内存已与两侧的可用内存组合成一个更大的可用块。

`Heap_4` 是非确定性的，但比 `malloc()` 和 `free()` 的大多数标准库实现都要快。

## 为 Heap\_4 使用的数组设置起始地址

注意：此部分包含高级信息。使用 `heap_4` 并不需要阅读本节内容。

有时，应用程序编写者必须将 `heap_4` 使用的数组放在特定的内存地址处。例如，FreeRTOS 任务使用的堆栈是从堆中分配的，因此，可能需要确保堆位于快速内部内存中，而不是慢速外部内存中。

默认情况下，`heap_4` 使用的数组在 `heap_4.c` 源文件中声明。其起始地址由链接器自动设置。但是，如果在 `FreeRTOSConfig.h` 中将 `configAPPLICATION_ALLOCATED_HEAP` 编译时间配置常量设置为 1，则应用程序必须使用 FreeRTOS 声明数组。如果将数组声明为应用程序的一部分，则应用程序编写者可以设置其起始地址。

如果在 `FreeRTOSConfig.h` 中将 `configAPPLICATION_ALLOCATED_HEAP` 设置为 1，则名为 `ucHeap` 且由 `configTOTAL_HEAP_SIZE` 设置确定大小的 `uint8_t` 数组必须在应用程序的一个源文件中声明。

将变量放到特定内存地址所需的语法取决于编译器。有关信息，请参阅编译器的相应文档。

后面介绍两个编译器示例。

下面是 GCC 编译器声明数组和将其放入名为 .my\_heap 的内存分配所需的语法：

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] __attribute__ ( ( section( ".my_heap" ) ) );
```

下面是 IAR 编译器声明数组和将其放入绝对内存地址 0x20000000 所需的语法：

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] @ 0x20000000;
```

## Heap\_5

heap\_5 用来分配和释放内存的算法与 heap\_4 的完全相同。与 heap\_4 不同，heap\_5 不限于从单个静态声明的数组分配内存。Heap\_5 可以从多个单独的内存空间分配内存。当运行 FreeRTOS 的系统提供的 RAM 在系统的内存映射中未作为单个邻接的（没有空间）块出现时，Heap\_5 很有用。

Heap\_5 是唯一一个必须在调用 pvPortMalloc() 之前显式初始化的内存分配方案。它使用 vPortDefineHeapRegions() API 函数进行初始化。当使用 heap\_5 时，必须先调用 vPortDefineHeapRegions()，然后才可创建任何内核对象（任务、队列、信号灯等）。

## vPortDefineHeapRegions() API 函数

vPortDefineHeapRegions() 用于指定每个单独内存区域的起始地址和大小。它们共同构成 heap\_5 所使用的总内存量。

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

每个单独的内存区域都由一个类型为 HeapRegion\_t 的结构描述。所有可用内存区域的描述作为 HeapRegion\_t 结构的数组传递到 vPortDefineHeapRegions()。

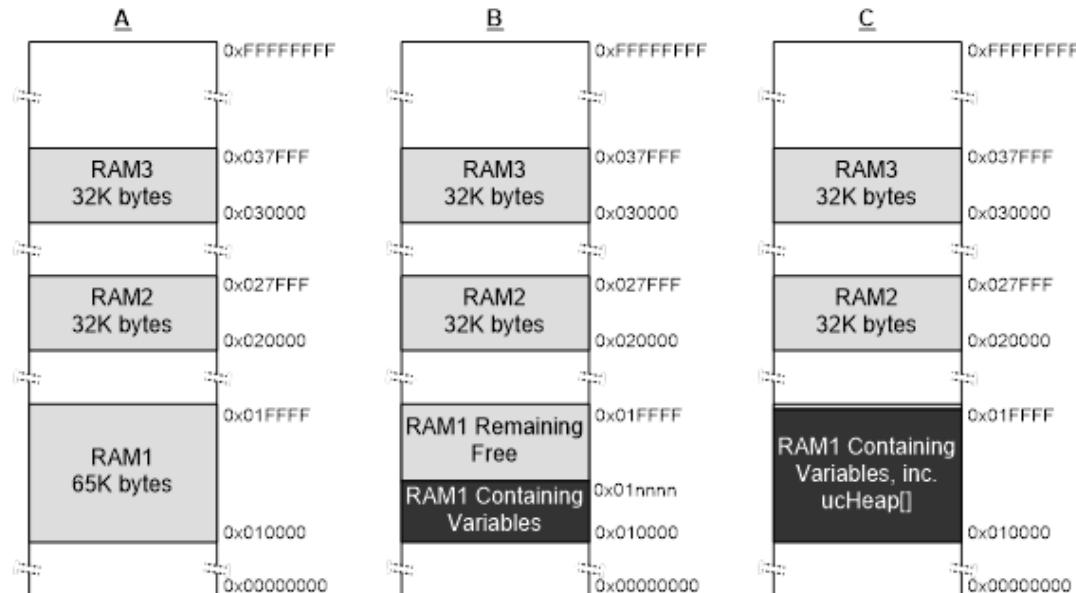
```
typedef struct HeapRegion
{
    /* The start address of a block of memory that will be part of the heap.*/
    uint8_t *pucStartAddress;
    /* The size of the block of memory in bytes. */
    size_t xSizeInBytes;
} HeapRegion_t;
```

下表中列出了 vPortDefineHeapRegions() 参数。

参数名称/返回的值	描述
pxHeapRegions	指向 HeapRegion_t 结构的数组开头的指针。数组中的每个结构描述当使用 heap_5 时将成为堆的一部分的内存区域的起始地址和长度。数组中的 HeapRegion_t 结构必须按起始地址排序。描述具有最低起始地址的内存区域的 HeapRegion_t

结构必须是数组中的第一个结构，而描述具有最高起始地址的内存区域的 HeapRegion\_t 结构必须是数组中的最后一个结构。数组的末尾由已将其 pucStartAddress 成员设置为 NULL 的 HeapRegion\_t 结构标记。

举例来说，考虑下图中显示的一个假设内存映射，其中包含三个单独的 RAM 块：RAM1、RAM2 和 RAM3。可执行文件代码放置在只读内存中，但未显示。



下面的代码演示了 HeapRegion\_t 结构的数组。它们共同描述三个 RAM 块。

```

/* Define the start address and size of the three RAM regions. */

#define RAM1_START_ADDRESS ( ( uint8_t * ) 0x00010000 )
#define RAM1_SIZE ( 65 * 1024 )

#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )
#define RAM2_SIZE ( 32 * 1024 )

#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )
#define RAM3_SIZE ( 32 * 1024 )

/* Create an array of HeapRegion_t definitions, with an index for each of the three RAM
regions, and terminating the array with a NULL address. The HeapRegion_t structures must
appear in start address order, with the structure that contains the lowest start address
appearing first. */

const HeapRegion_t xHeapRegions[ ] =
{
    { RAM1_START_ADDRESS, RAM1_SIZE },
    { RAM2_START_ADDRESS, RAM2_SIZE },
    { RAM3_START_ADDRESS, RAM3_SIZE },
    { 0 }
};

```

```

    { NULL, 0 } /* Marks the end of the array. */

};

int main( void )
{
    /* Initialize heap_5. */

    vPortDefineHeapRegions( xHeapRegions );

    /* Add application code here. */
}

```

尽管代码正确地描述了 RAM，但这是不实用的示例，因为它将所有 RAM 都分配给堆，让其他变量无 RAM 可用。

构建项目时，构建过程的链接阶段会向每个变量分配 RAM 地址。可供链接器使用的 RAM 变量正常情况下由链接器配置文件（如链接器脚本）描述。在上图的 B 中，假定链接器脚本包括有关 RAM1 的信息，但不包括有关 RAM2 或 RAM3 的信息。因此，链接器将变量放入 RAM1 中，只留下 RAM1 中地址在 0x0001nnnn 之上的部分供 heap\_5 使用。0x0001nnnn 的实际值将取决于所链接的应用程序中包含的所有变量的组合大小。链接器已离开所有未使用的 RAM2 和 RAM3，因此它们可供 heap\_5 使用。

如果使用上述代码，则分配给 heap\_5 且地址低于 0x0001nnnn 的 RAM 将与用于保留变量的 RAM 重叠。为避免发生这种情况，xHeapRegions[] 数组中的第一个 HeapRegion\_t 结构可以使用起始地址 0x0001nnnn，而不是 0x00010000。

这不是推荐的解决方案，因为：

- 起始地址可能不容易确定。
- 链接器使用的 RAM 量在将来的内部版本中可能会发生变化，这就需要更新 HeapRegion\_t 结构中使用的起始地址。
- 构建工具将不知道这一点，因此无法警告应用程序编写者 heap\_5 使用的 RAM 是否重叠。

下面的代码演示了一个更方便和可维护的示例。它声明名为 ucHeap 的数组。ucHeap 是一个正常变量，因此它会成为链接器分配给 RAM1 的数据的一部分。xHeapRegions 数组中的第一个 HeapRegion\_t 结构描述 ucHeap 的起始地址和大小，因此 ucHeap 成为 heap\_5 管理的内存的一部分。可以增加 ucHeap 的大小，直到链接器使用的 RAM 消耗掉所有 RAM1，如上图中的 C 所示。

```

/* Define the start address and size of the two RAM regions not used by the linker. */

#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )

#define RAM2_SIZE ( 32 * 1024 )

#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )

#define RAM3_SIZE ( 32 * 1024 )

/* Declare an array that will be part of the heap used by heap_5. The array will be placed
   in RAM1 by the linker. */

#define RAM1_HEAP_SIZE ( 30 * 1024 )

static uint8_t ucHeap[ RAM1_HEAP_SIZE ];

/* Create an array of HeapRegion_t definitions. Whereas in previous code listing, the first
   entry described all of RAM1, so heap_5 will have used all of RAM1, this time the first

```

```
entry only describes the ucHeap array, so heap_5 will only use the part of RAM1 that
contains the ucHeap array. The HeapRegion_t structures must still appear in start address
order, with the structure that contains the lowest start address appearing first. */

const HeapRegion_t xHeapRegions[ ] =
{
    { ucHeap, RAM1_HEAP_SIZE },
    { RAM2_START_ADDRESS, RAM2_SIZE },
    { RAM3_START_ADDRESS, RAM3_SIZE },
    { NULL, 0 } /* Marks the end of the array. */
};
```

在前面的代码中，HeapRegion\_t 结构的数组描述所有 RAM2 和所有 RAM3，但只描述 RAM1 的一部分。

此处演示的方法具有以下优势：

- 您不需要使用硬编码的起始地址。
- HeapRegion\_t 结构中使用的地址将由链接器自动设置，因此它将始终是正确的，即使在将来的内部版本中更改链接器使用的 RAM 量也是如此。
- 分配给 heap\_5 的 RAM 不可能与链接器放入 RAM1 的数据发生重叠。
- 如果 ucHeap 太大，则应用程序不会链接。

## 与堆相关的实用程序函数

### xPortGetFreeHeapSize() API 函数

调用 xPortGetFreeHeapSize() API 函数时，此函数将返回堆中的可用字节数。它可用于优化堆大小。例如，如果在创建所有内核对象后 xPortGetFreeHeapSize() 返回 2000，则 configTOTAL\_HEAP\_SIZE 的值可能会减少 2000。

使用 heap\_3 时，xPortGetFreeHeapSize() 不可用。

xPortGetFreeHeapSize() API 函数原型

```
size_t xPortGetFreeHeapSize( void );
```

下表列出了 xPortGetFreeHeapSize() 返回值。

参数名称/返回的值	描述
返回的值	调用 xPortGetFreeHeapSize() 时堆中保留的未分配字节数。

### xPortGetMinimumEverFreeHeapSize() API 函数

xPortGetMinimumEverFreeHeapSize() API 函数返回自 FreeRTOS 应用程序开始执行以来堆中已存在的最 小数量的未分配字节。

xPortGetMinimumEverFreeHeapSize() 返回的值指示应用程序接近用尽堆空间的程度。例如，如果 xPortGetMinimumEverFreeHeapSize() 返回 200，则自应用程序开始执行后的某个时间，离用尽堆空间还不足 200 字节。

仅当使用 heap\_4 或 heap\_5 时，xPortGetMinimumEverFreeHeapSize() 才可用。

下表列出了 xPortGetMinimumEverFreeHeapSize() 返回值。

参数名称/返回的值	描述
返回的值	自 FreeRTOS 应用程序开始执行以来堆中已存在的最小数量的未分配字节。

## Malloc 失败挂钩函数

可以直接从应用程序代码中调用 pvPortMalloc()。在每次创建内核对象时，也可以在 FreeRTOS 源文件中调用它。内核对象包括任务、队列、信号灯和事件组，所有这些将在后面介绍。

就像标准库函数 malloc() 一样，如果 pvPortMalloc() 因为所请求大小的块不存在而无法返回 RAM 块，它将返回 NULL。如果因为应用程序编写者正在创建内核对象而执行 pvPortMalloc()，并且对 pvPortMalloc() 的调用返回 NULL，则不会创建内核对象。

如果对 pvPortMalloc() 的调用返回 NULL，则所有示例堆分配方案都可以配置为调用一个挂钩（或回调）函数。

如果在 FreeRTOSConfig.h 中将 configUSE\_MALLOC\_FAILED\_HOOK 设置为 1，则应用程序必须提供具有如下所示名称和原型的 malloc 失败挂钩函数。该函数可以通过任何适合应用程序的方式实现。

```
void vApplicationMallocFailedHook( void );
```

# 任务管理

本节介绍的概念是了解如何使用 FreeRTOS 以及 FreeRTOS 应用程序如何运行的基础知识。本节内容：

- FreeRTOS 如何为应用程序中的每个任务分配处理时间。
- 在任何给定时间，FreeRTOS 如何选择哪个任务应该执行。
- 每个任务的相对优先级如何影响系统行为。
- 任务可能具有的状态。

本节还介绍：

- 如何实现任务。
- 如何为一个任务创建一个或多个实例。
- 如何使用任务参数。
- 如何更改已创建的任务的优先级。
- 如何删除任务。
- 如何使用任务实现周期性处理。有关更多信息，请参阅 software\_tier\_management。
- 空闲任务何时执行，以及如何使用空闲任务。

## 任务函数

任务以 C 函数形式实现。它们唯一特别的一点，是它们的原型必须返回 void 并使用 void 指针参数，如下所示。

```
void ATaskFunction( void *pvParameters );
```

每个任务本身都是一个小程序。它有一个入口点，通常会在一个无限循环中永远运行，而不会退出。

FreeRTOS 任务不能允许以任何方式从其实现函数返回。它们不得包含“return”语句，也不得在函数结束之后执行。如果不再需要某个任务，应该将它显式删除。

单个任务函数定义可用来创建任意数量的任务。已创建的每个任务都是一个单独的执行实例，具有自己的堆栈，以及在任务本身之内定义的所有自动化（堆栈）变量的副本。

下面是典型任务的结构。

```
void ATaskFunction( void *pvParameters )

{
    /* Variables can be declared just as per a normal function. Each instance of a task created
       using this example function will have its own copy of the lVariableExample variable. This
       would not be true if the variable was declared static, in which case only one copy of the
       variable would exist, and this copy would be shared by each created instance of the task.
       (The prefixes added to variable names are described in Data Types and Coding Style Guide.)
    */

    int32_t lVariableExample = 0;
    /* A task will normally be implemented as an infinite loop. */
    for( ;; )
}
```

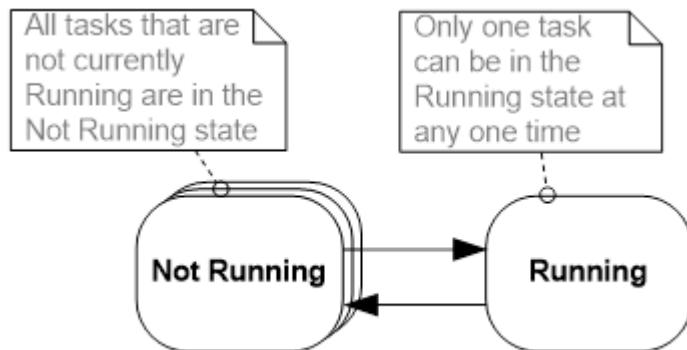
```
{  
    /* The code to implement the task functionality will go here. */  
  
}  
  
/* Should the task implementation ever break out of the above loop, then the task must be  
deleted before reaching the end of its implementing function. The NULL parameter passed to  
the vTaskDelete() API function indicates that the task to be deleted is the calling (this)  
task. The convention used to name API functions is described in Data Types and Coding  
Style Guide. */  
  
vTaskDelete( NULL );  
  
}
```

## 顶级任务状态

一个应用程序可以包含很多任务。如果运行应用程序的处理器只包含单个核心，则在任何给定时间只能有一个任务在执行。这意味着任务可以处于两种状态之一：运行和未运行。请注意，这是很粗略的划分。在本节稍后部分，您会了解到，“未运行”状态实际上包含很多子状态。

当任务处于“运行”状态时，处理器执行任务代码。当任务处于“未运行”状态时，任务休眠，其状态已保存，因此，当下次计划程序决定它应该进入“运行”状态时，它可以恢复执行。当任务恢复执行时，它从上一次离开“运行”状态之前将要执行的那一条指令开始恢复执行。

下图介绍顶层任务状态和转换。



从“未运行”状态转换为“运行”状态的任务称为进行了切入 或换入。反过来，从“未运行”状态转换为“运行”状态的任务称为进行了切出 或换出。FreeRTOS 计划程序是唯一可以切入和切出任务的实体。

## 创建任务

### xTaskCreate() API 函数

FreeRTOS V9.0.0 还包含 xTaskCreateStatic() 函数，该函数用于分配在编译时静态创建任务所需的内存。任务是使用 FreeRTOS xTaskCreate() API 函数创建的。这可能是所有 API 函数中最复杂的。因为任务是多任务处理系统最基本的组成部分，所以首先必须掌握相关知识。本指南中有很多 xTaskCreate() 函数的示例。

有关所使用的数据类型和命名约定的信息，请参阅 [FreeRTOS 内核分发版 \(p. 4\)](#) 中的“数据类型和编程风格指南”。

下面是 xTaskCreate() API 函数原型。

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode, const char * const pcName, uint16_t usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask );
```

下面列出了 xTaskCreate() 参数和返回值。

- **pvTaskCode** : 任务只是永不退出的 C 函数，因此通常以无限循环形式实现。pvTaskCode 参数只是一个指针，它指向实现任务的函数（实际上就是该函数的名称）。
- **pcName** : 任务的描述性名称。FreeRTOS 不会以任何方式使用此参数。它只是用于在调试时提供辅助。以人类易读的名称来标识任务，比尝试用任务句柄进行标识要简单得多。应用程序定义的常量 configMAX\_TASK\_NAME\_LEN 定义任务名称的最大长度，包括 NULL 终止符。如果提供的字符串长度超过此最大值，字符串将以无提示方式被截断。
- **usStackDepth** : 每个任务都有自己唯一的堆栈，这是在创建任务时，由内核为每个任务分配的。usStackDepth 值表明内核应该分配多大的堆栈。该值指定堆栈可容纳的字（而不是字节）的数量。例如，如果堆栈是 32 位宽，传入的 usStackDepth 为 100，则将分配 400 字节（ $100 \times 4$  字节）的堆栈空间。堆栈深度乘以堆栈宽度不得超过 uint16\_t 类型变量可以包含的最大值。空闲任务使用的堆栈的大小是由应用程序定义的常量 configMINIMAL\_STACK\_SIZE 所定义的。这是 FreeRTOS 源代码使用 configMINIMAL\_STACK\_SIZE 设置的唯一方式。该常量也在演示应用程序内使用，以帮助使这些演示可以跨多个处理器架构移植。在 FreeRTOS 演示应用程序中针对所使用的处理器架构分配给此常量的值是对任何任务建议使用的最小值。如果您的任务需要使用大量堆栈空间，则必须分配更大的值。确定某个任务所需的堆栈空间没有什么简单的方法。这是可以计算出来的，但是大多数用户只是简单地分配他们认为的合理值，然后使用 FreeRTOS 功能来确保分配的空间的确足够，并且不浪费 RAM。有关如何查询任务已使用的最大堆栈空间的信息，请参阅“[疑难解答](#)”部分中的 [堆栈溢出 \(p. 217\)](#)。
- **pvParameters** : 任务函数接受一个 void 指针类型 (void\*) 的参数。分配给 pvParameters 的值是传递到任务中的值。本指南中有一些示例介绍如何使用该参数。
- **uxPriority** : 定义任务将在哪个优先级执行。优先级的分配范围可以是从 0 (最低优先级) 到 (configMAX\_PRIORITIES - 1) (最高优先级)。configMAX\_PRIORITIES 是用户定义的常量，[任务优先级 \(p. 29\)](#) 中对该常量进行了介绍。如果传递的 uxPriority 值超过 (configMAX\_PRIORITIES - 1)，将导致分配给任务的优先级会以无提示方式被限制为最大合法值。
- **pxCreatedTask** : 此参数可用于传递出所创建的任务的句柄。然后，此句柄可在 API 调用（例如，更改任务优先级或删除任务）中用来引用该任务。如果任务句柄对您的应用程序而言没有用，可将 pxCreatedTask 设置为 NULL。

有两个可能的返回值：一个是 pdPASS，表示任务已成功创建；另一个是 pdFAIL，表示任务尚未创建，因为没有足够的堆内存可供 FreeRTOS 分配足够的 RAM 来保存任务数据结构和堆栈。有关更多信息，请参阅 [堆内存管理 \(p. 12\)](#)。

## 创建任务 (示例 1 )

本示例介绍创建两个简单的任务并开始执行它们所需的步骤。这些任务只是使用一个粗略的 null 循环创建周期延迟来周期性地输出一个字符串。两个任务以相同优先级创建，除了输出的字符串外，并无不同。下面的代码介绍示例 1 中使用的第一任务的实现。

```
void vTask1( void *pvParameters )  
{  
    const char *pcTaskName = "Task 1 is running\r\n";  
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */  
    /* As per most tasks, this task is implemented in an infinite loop. */  
    for( ;; )
```

```
{  
    /* Print out the name of this task. */  
  
    vPrintString( pcTaskName );  
  
    /* Delay for a period. */  
  
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )  
  
    {  
  
        /* This loop is just a very crude delay implementation. There is nothing to do in  
        here. Later examples will replace this crude loop with a proper delay/sleep function. */  
  
    }  
  
}
```

下面的代码介绍示例 1 中使用的第二个任务的实现。

```
void vTask2( void *pvParameters )  
  
{  
  
    const char *pcTaskName = "Task 2 is running\r\n";  
  
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */  
  
    /* As per most tasks, this task is implemented in an infinite loop. */  
  
    for( ;; )  
  
    {  
  
        /* Print out the name of this task. */  
  
        vPrintString( pcTaskName );  
  
        /* Delay for a period. */  
  
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )  
  
        {  
  
            /* This loop is just a very crude delay implementation. There is nothing to do  
            in here. Later examples will replace this crude loop with a proper delay/sleep function.  
            */  
  
        }  
  
    }  
  
}
```

main() 函数在启动计划程序之前创建这些任务。

```
int main( void )  
  
{
```

```
/* Create one of the two tasks. Note that a real application should check the return
value of the xTaskCreate() call to ensure the task was created successfully. */

xTaskCreate( vTask1, /* Pointer to the function that implements the task. */ "Task 1",
* Text name for the task. This is to facilitate debugging only. */ 1000, /* Stack depth -
small microcontrollers will use much less stack than this. */ NULL, /* This example does
not use the task parameter. */ 1, /* This task will run at priority 1. */ NULL ); /* This
example does not use the task handle. */

/* Create the other task in exactly the same way and at the same priority. */

xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

/* Start the scheduler so the tasks start executing. */

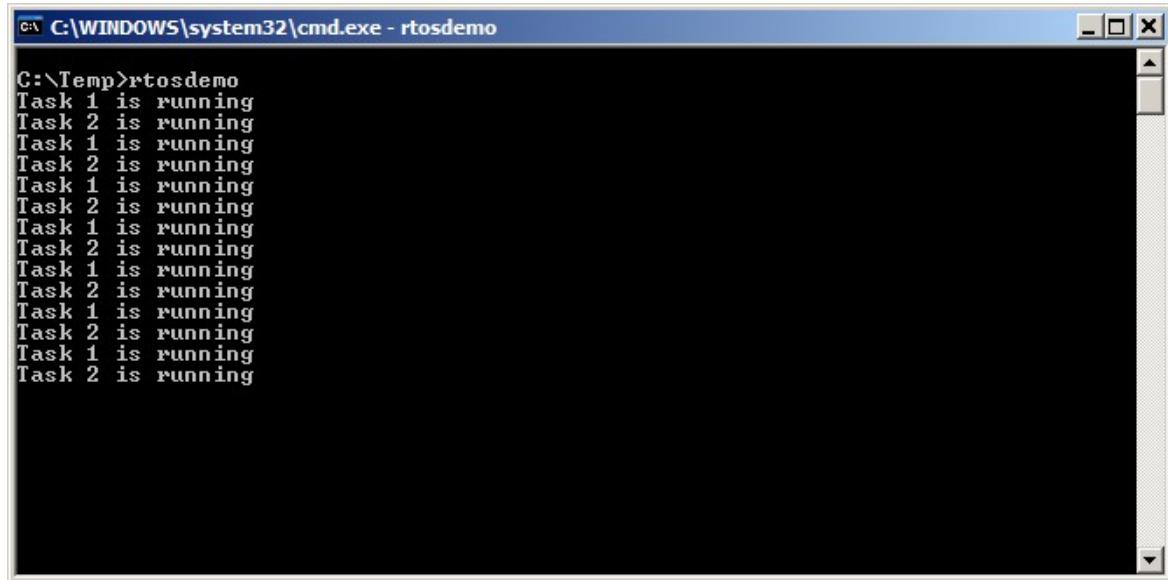
vTaskStartScheduler();

/* If all is well then main() will never reach here as the scheduler will now be
running the tasks. If main() does reach here then it is likely that there was insufficient
heap memory available for the idle task to be created. For more information, see Heap
Memory Management. */

for( ;; );

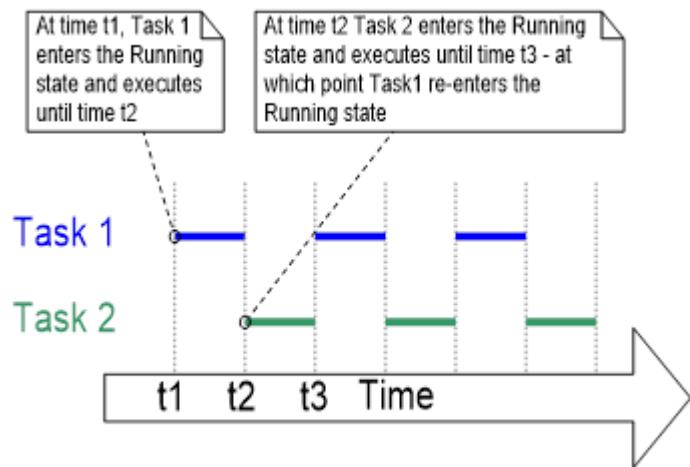
}
```

下面是输出的内容。



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
C:\Temp>rtosdemo
Task 1 is running
Task 2 is running
```

两个任务似乎是同时执行的。但是，同一个处理器核心上不能同时运行两个任务，因此不可能是这样。实际上，两个任务都在快速进入和退出“运行”状态。两个任务在相同优先级运行，因此分享同一个处理器核心上的时间。它们的执行模式如下图中所示。



此图底部的箭头表示从时间  $t_1$  开始的时间。彩色线显示每个时间点哪个任务在执行（例如，在时间  $t_1$  和时间  $t_2$  之间，Task 1 在执行）。

任何时候只能有一个任务处于“运行”状态，因此当一个任务进入“运行”状态（任务切入）时，另一个任务就进入“未运行”状态（任务切出）。

两个任务都是在计划程序启动之前在 main() 中创建的。也可以从任务中创建另一个任务。例如，Task 2 就可以从 Task 1 中创建。

下面的代码介绍计划程序启动后从一个任务中创建另一个任务。

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* If this task code is executing then the scheduler must already have been started.
    Create the other task before entering the infinite loop. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is nothing to do
            in here. Later examples will replace this crude loop with a proper delay/sleep function.
            */
        }
    }
}
```

```
        }
    }
}
```

## 使用任务参数 (示例 2)

示例 1 中创建的两个任务几乎相同。唯一的区别是它们输出的文本字符串不一样。通过创建单个任务实现的两个实例，可以消除这种重复现象。然后可以使用任务参数向每个任务传入它应该输出的字符串。

下面包含单个任务函数 (vTaskFunction) 的代码。该单个函数代替示例 1 中使用的两个任务函数 ( vTask2 和 vTask1 )。任务参数强制转换为 char \* 以获取任务应输出的字符串。

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* The string to print out is passed in via the parameter. Cast this to a character
       pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )

    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )

        {
            /* This loop is just a very crude delay implementation. There is nothing to do
               in here. Later exercises will replace this crude loop with a delay/sleep function. */
        }
    }
}
```

即使现在只有一个任务实现 (vTaskFunction)，也可以为所定义的任务创建多个实例。创建的每个实例都将在 FreeRTOS 计划程序的控制下独立执行。

下面的代码介绍如何使用 xTaskCreate() 函数的 pvParameters 参数向任务传入文本字符串。

```
/* Define the strings that will be passed in as the task parameters. These are defined
   const and not on the stack to ensure they remain valid when the tasks are executing. */

static const char *pcTextForTask1 = "Task 1 is running\r\n";
```

```
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create one of the two tasks. */

    xTaskCreate( vTaskFunction, /* Pointer to the function that implements the task.
    */ "Task 1", /* Text name for the task. This is to facilitate debugging only. */
    1000, /* Stack depth - small microcontrollers will use much less stack than this. */
    (void*)pcTextForTask1, /* Pass the text to be printed into the task using the task
    parameter. */ 1, /* This task will run at priority 1. */ NULL );
    /* The task handle is not used in this example. */

    /* Create the other task in exactly the same way. Note this time that multiple tasks
    are being created from the SAME task implementation (vTaskFunction). Only the value passed
    in the parameter is different. Two instances of the same task are being created. */

    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

    /* Start the scheduler so the tasks start executing. */

    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will now be
    running the tasks. If main() does reach here then it is likely that there was insufficient
    heap memory available for the idle task to be created. For more information, see Heap
    Memory Management. */

    for( ;; );
}
```

## 任务优先级

xTaskCreate() API 函数的 uxPriority 参数向要创建的任务分配初始优先级。在计划程序启动后，您可使用 vTaskPrioritySet() API 函数更改优先级。

可用优先级的最大数量由 FreeRTOSConfig.h 中应用程序定义的 configMAX\_PRIORITIES 编译时配置常量设置。低数字优先级值表示低优先级任务，优先级 0 是最低优先级。因此，可用优先级的范围是 0 到 (configMAX\_PRIORITIES – 1)。任意数量的任务可以具有同一个优先级，以确保最大的设计灵活性。

FreeRTOS 计划程序可以使用两种方法之一来确定哪个任务将处于“运行”状态。configMAX\_PRIORITIES 可设置的最大值取决于所使用的方法：

### 1. 泛型方法

此方法用 C 实现，可用于 FreeRTOS 架构的所有端口。

使用泛型方法时，FreeRTOS 不会限制 configMAX\_PRIORITIES 可设置的最大值。但是，建议将 configMAX\_PRIORITIES 保持为所需的最小值，因为其值越大，所使用的 RAM 将越多，最差情况下的执行时间也将越长。

如果 FreeRTOSConfig.h 中的 configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 设置为 0，或者 configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 保留为未定义，或者泛型方法是为所使用的 FreeRTOS 端口提供的唯一方法，则将使用此方法。

### 2. 针对架构优化的方法

此方法使用少量汇编程序代码，比泛型方法快。configMAX\_PRIORITIES 设置不影响最差情况的执行时间。

如果使用此方法，configMAX\_PRIORITIES 不能大于 32。使用泛型方法时，应将 configMAX\_PRIORITIES 保持为所需的最小值，因为其值越大，所使用的 RAM 越多。

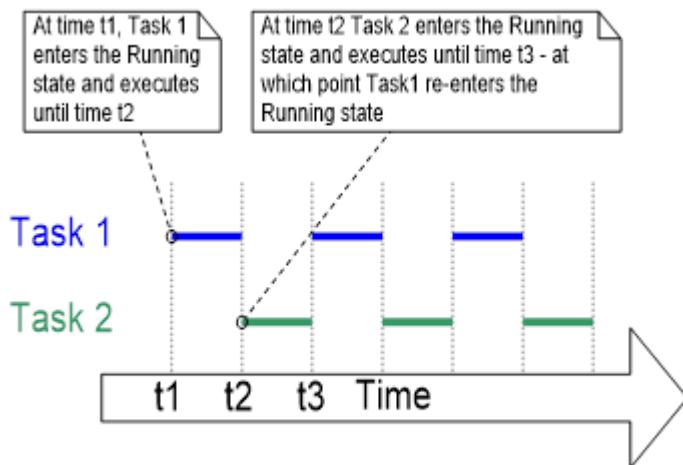
如果在 FreeRTOSConfig.h 中，configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 设置为 1，则将使用此方法。

并非所有 FreeRTOS 端口都提供针对架构优化的方法。

FreeRTOS 计划程序将始终确保能够运行的最高优先级任务是选为要进入“运行”状态的任务。在多个相同优先级的任务能够运行的情况下，计划程序轮流将每个任务转入和转出“运行”状态。

## 时间测量和时钟周期中断

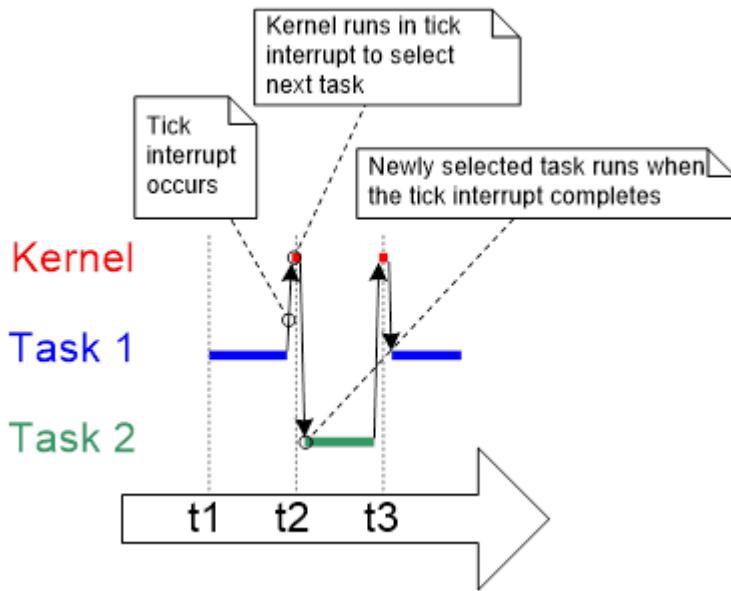
计划算法 (p. 51)一节介绍了一项称为时间切片的可选功能。到现在为止，示例中已经使用了时间切片。这是在示例生成的输出中出现的行为。在这些示例中，两个任务以相同的优先级创建，并且两个任务始终都能够运行。因此，每个任务在一个时间切片内执行，在时间切片开始时进入“运行”状态，在时间切片结束时退出“运行”状态。在此图中，t1 和 t2 之间的时间就等于一个时间切片。



为了能够选择下一个要运行的任务，计划程序本身在每个时间切片结束时必须执行。时间切片的末尾并不是计划程序可以选择新任务来运行的唯一位置。计划程序也会在当前正在执行的任务进入“被阻止”状态之后或当中断将较高优先级任务转入“准备就绪”状态时选择新任务立即运行。称为时钟周期中断的周期性中断就用于这一目的。时间切片的长度实际上是按时钟周期中断频率设置的，此频率由 FreeRTOSConfig.h 中应用程序定义的 configTICK\_RATE\_HZ 编译时配置常量配置。例如，如果 configTICK\_RATE\_HZ 设置为 100 (Hz)，则时间切片为 10 毫秒。两个时钟周期中断之间的时间称为时钟周期。一个时间切片等于一个时钟周期。

下图是展开的执行顺序，以说明时钟周期中断执行情况。最上面一行显示计划程序的执行时间。细箭头指示从任务到时钟周期中断，再从时间周期中断回到不同任务的执行顺序。

configTICK\_RATE\_HZ 的最佳值取决于正在开发的应用程序，但 100 是典型值。



FreeRTOS API 调用始终以时钟周期的倍数来指定时间，也简称为时钟周期数。pdMS\_TO\_TICKS() 宏将以毫秒为单位指定的时间转换为以时钟周期数指定的时间。可用的分辨率取决于定义的时钟周期频率。如果时钟周期频率高于 1 KHz ( 如果 configTICK\_RATE\_HZ 大于 1000 )，则不能使用 pdMS\_TO\_TICKS()。下面的代码介绍如何使用 pdMS\_TO\_TICKS() 将指定为 200 毫秒的时间转换为以时钟周期数指定的等效时间。

```
/* pdMS_TO_TICKS() takes a time in milliseconds as its only parameter, and evaluates to the equivalent time in tick periods. This example shows xTimeInTicks being set to the number of tick periods that are equivalent to 200 milliseconds. */

TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```

注意：建议不要在应用程序中直接以时钟周期为单位指定时间。而应使用 pdMS\_TO\_TICKS() 宏来指定时间（以毫秒为单位）。这可确保应用程序中指定的时间不会因为时钟频率改变而改变。

“时钟计数”值是计划程序启动以来发生的时钟周期中断的总数（假定时钟计数未溢出）。在指定延迟时段时，用户应用程序不必考虑溢出，因为时间一致性由 FreeRTOS 在内部进行管理。

有关会影响计划程序何时选择要运行的新任务以及时钟周期中断何时执行的配置常量的信息，请参阅[计划算法 \(p. 51\)](#)。

## 试用优先级 (示例 3)

计划程序将始终确保能够运行的最高优先级任务是选为要进入“运行”状态的任务。到目前为止使用的示例中，两个任务是以相同优先级创建的，因此这两个任务轮流进入和退出“运行”状态。此示例说明当示例 2 中创建的两个任务之一的优先级更改时，会发生什么。这一次，第一个任务以优先级 1 创建，第二个任务以优先级 2 创建。

下面是以不同优先级创建任务的示例代码。实现两个任务的单个函数并未更改。它仍然只是使用一个 null 循环生成延迟来周期性地输出一个字符串。

```
/* Define the strings that will be passed in as the task parameters. These are defined const and not on the stack to ensure they remain valid when the tasks are executing. */

static const char *pcTextForTask1 = "Task 1 is running\r\n";
```

```
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create the first task at priority 1. The priority is the second to last parameter.
     */
    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2, which is higher than a priority of 1. The
     priority is the second to last parameter. */

    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

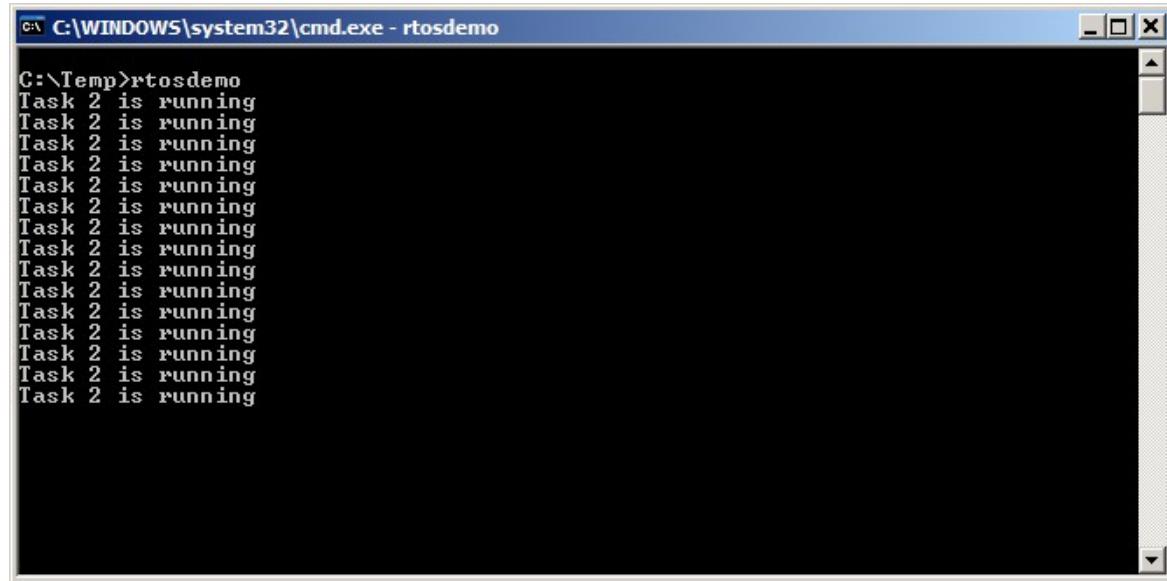
    /* Start the scheduler so the tasks start executing. */

    vTaskStartScheduler();

    /* Will not reach here. */

    return 0;
}
```

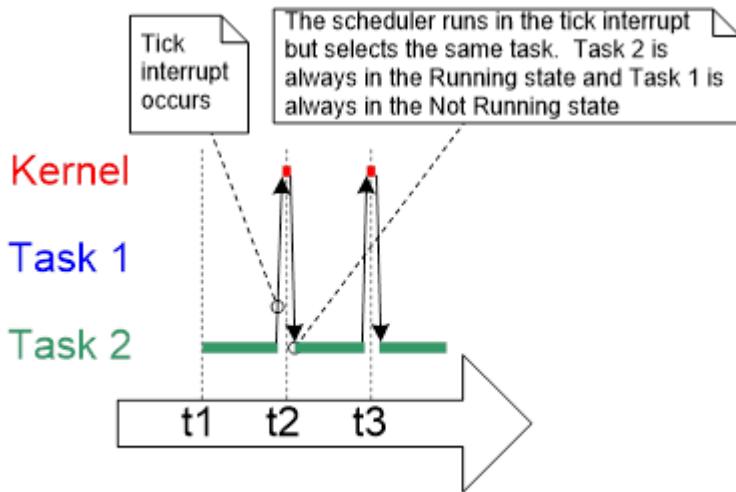
下面是此示例生成的输出。计划程序始终选择能够运行的最高优先级任务。Task 2 的优先级比 Task 1 高，因此始终能够运行。因此，Task 2 是唯一可以进入“运行”状态的任务。由于 Task 1 无法进入“运行”状态，它不会输出其字符串。Task 1 称为因 Task 2 而缺乏处理时间。



```
C:\Temp>rtosdemo
Task 2 is running
```

Task 2 始终能够运行，因为它从不需要等待。它通过一个 null 循环来循环运行或输出到终端。

下图是先前的示例代码的执行顺序。



## “未运行”状态详述

到目前为止，创建的任务始终可以执行处理，从不需要等待。因为它们从不需要等待，所以始终能够进入“运行”状态。这种连续处理任务用处有限，因为这些任务只能以最低优先级创建。如果它们以任何其他优先级运行，会令优先级更低的任务根本无法运行。

为使这些任务有用，必须将它们重新编写为用事件驱动。事件驱动型任务仅在发生触发它的事件后有工作（处理）要执行，在该事件发生之前无法进入“运行”状态。计划程序始终选择能够运行的最高优先级任务。高优先级任务无法运行意味着计划程序无法选择它们，而必须选择优先级更低并且能够运行的任务。因此，使用事件驱动型任务意味着任务可在不同优先级创建，而优先级最高的任务不会使所有较低优先级任务缺少处理时间。

## “被阻止”状态

正在等待事件发生的任务称为处于“被阻止”状态，这是“未运行”状态的一个子状态。

任务可以进入“被阻止”状态等待两类事件：

1. 临时（时间相关）事件，即事件延迟时段到期或到达绝对时间。例如，一个任务可能进入“被阻止”状态等待 10 毫秒。
2. 同步事件，即事件源自另一个任务或中断。例如，一个任务进入“被阻止”状态等待数据到达队列。同步事件包括非常广泛的事件类型。

FreeRTOS 队列、二进制信号灯、计数信号灯、互斥锁、递归互斥锁、事件组和“直接到任务”通知都可以用于创建同步事件。

任务可能因具有超时的同步事件而被阻止，这实际上是同时因两类事件而被阻止。例如，任务可以选择在最多 10 毫秒时间内等待数据到达队列。如果数据在 10 毫秒内到达或 10 毫秒之后数据仍未到达，该任务将离开“被阻止”状态。

## “暂停”状态

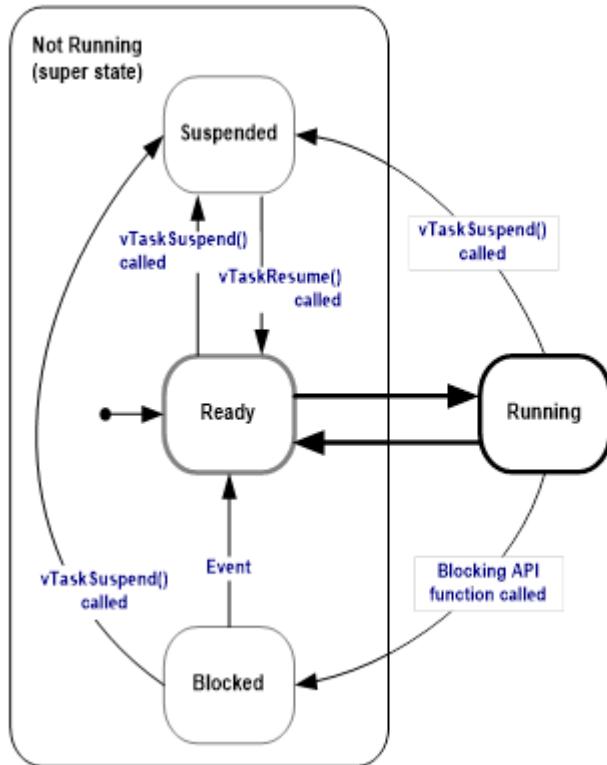
“暂停”也是一种“未运行”子状态。对计划程序而言，处于“暂停”状态的任务不可用。进入“暂停”状态的唯一方法是调用 vTaskSuspend() API 函数。离开“暂停”状态的唯一方法是调用 vTaskResume() 或 xTaskResumeFromISR() API 函数。大多数应用程序不使用“暂停”状态。

## “准备就绪”状态

处于“未运行”状态、但不是“被阻止”或“暂停”状态的任务称为处于“准备就绪”状态。它们能够运行，因此可随时运行，但当前并不处于“运行”状态。

## 完成状态转换示意图

下图对粗略的状态图进行展开介绍，其中包含本节所述的所有“未运行”子状态。到目前为止，示例中创建的任务都未使用“被阻止”或“暂停”状态。如此处的粗线所示，它们只是在“准备就绪”和“运行”状态之间来回转换。



## 使用“被阻止”状态创建延迟（示例 4）

到现在为止，提供的示例中创建的所有任务都是周期性任务。它们延迟一个时段，在下一次延迟之前输出字符串，如此循环。延迟是通过 null 循环非常粗略地生成的。任务实际上会轮询一个递增循环计数器，直至它达到某个固定值。示例 3 清楚地说明了这种方法的缺点。较高优先级的任务在执行 null 循环时仍保持“运行”状态，导致较低优先级的任务缺乏处理时间。

任何形式的轮询都有其他几方面的缺点，效率低下还不是最糟糕的。在轮询期间，任务不是真的有任何工作要做，但它使用的处理时间仍然最多，从而浪费处理器周期。示例 4 更正了这一行为，方法是将轮询 null 循环替换为对 vTaskDelay() API 函数的调用。vTaskDelay() API 函数仅当 FreeRTOSConfig.h 中的 INCLUDE\_vTaskDelay 设置为 1 时才可用。

vTaskDelay() 在固定的时钟周期中断数之内将调用任务置于“被阻止”状态。该任务处于“被阻止”状态时不会使用任何处理时间，因此它仅在有实际工作要做时才使用处理时间。

下面是 vTaskDelay() API 函数原型。

```
void vTaskDelay( TickType_t xTickToDelay );
```

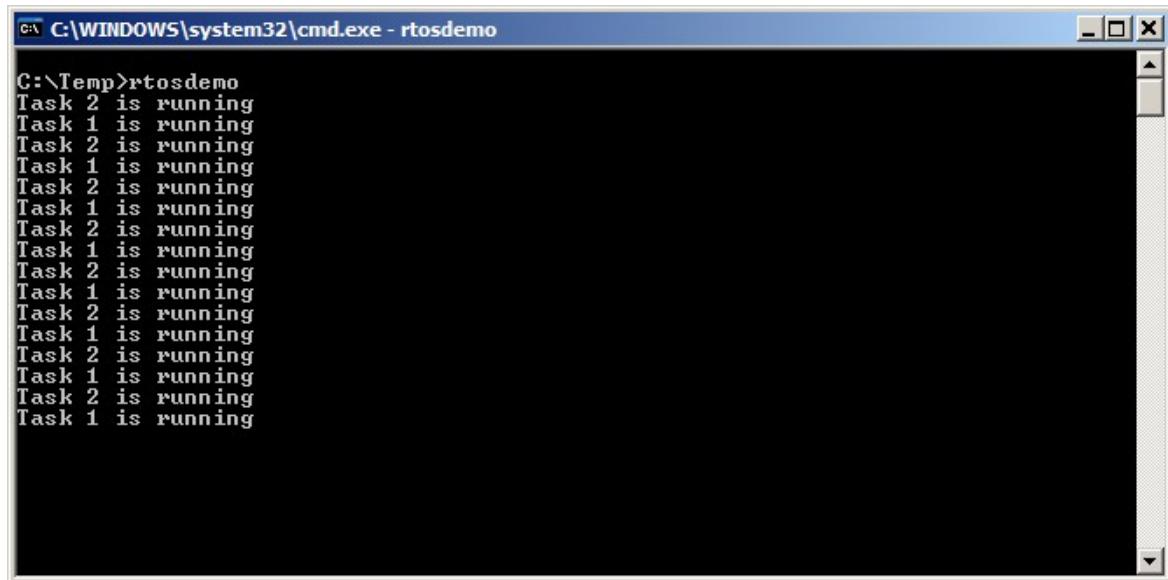
下表列出了 vTaskDelay() 参数。

参数名称	描述
xTicksToDelay	转换回“准备就绪”状态之前，调用任务将保持“被阻止”状态的时钟周期中断数。  例如，如果任务在时钟计数为 10000 时调用 vTaskDelay(100)，将立即进入并保持“被阻止”状态，直到时钟计数达到 10100。  宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为计时周期数。例如，调用 vTaskDelay( pdMS_TO_TICKS( 100 ) ) 将导致调用任务保持“被阻止”状态 100 毫秒时间。

在以下代码中，null 循环延迟后的示例任务已经替换为 vTaskDelay() 调用。这段代码说明新的任务定义。

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );
    /* The string to print out is passed in via the parameter. Cast this to a character
     pointer. */
    pcTaskName = ( char * ) pvParameters;
    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* Delay for a period. This time a call to vTaskDelay() is used which places the
         task into the Blocked state until the delay period has expired. The parameter takes a
         time specified in 'ticks'，and the pdMS_TO_TICKS() macro is used (where the xDelay250ms
         constant is declared) to convert 250 milliseconds into an equivalent time in ticks. */
        vTaskDelay( xDelay250ms );
    }
}
```

即使这两个任务仍以不同优先级创建，现在也都可运行。输出确认了预期的行为。

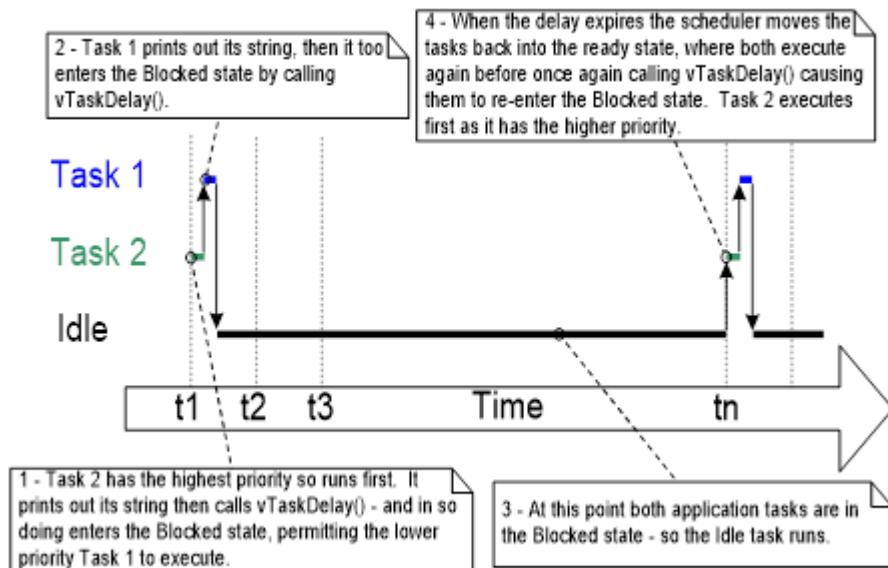


```
C:\Temp>rtosdemo
Task 2 is running
Task 1 is running
```

下图所示的执行顺序说明为何这两个任务即使以不同优先级创建，也都能够运行。为简单起见，省略了计划程序本身的执行。

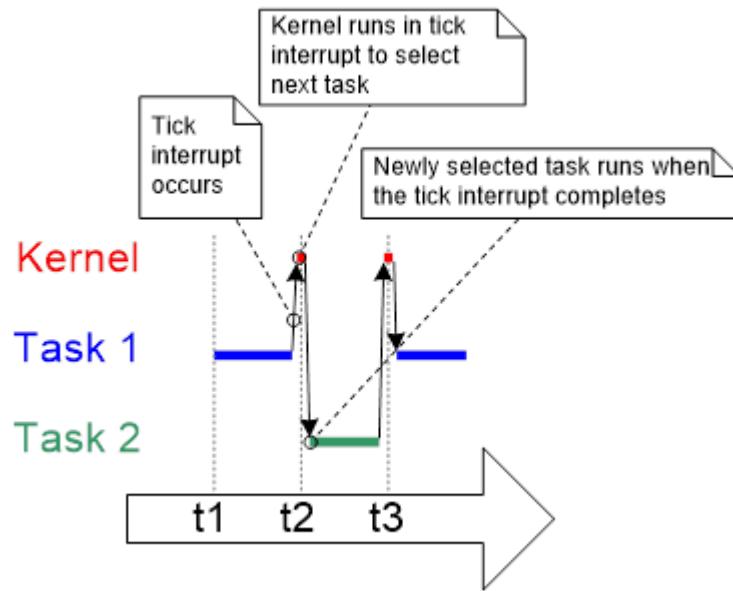
当计划程序启动时，将自动创建空闲任务，以确保始终至少有一个任务能够运行（至少一个任务处于“准备就绪”状态）。有关空闲任务的更多信息，请参阅[空闲任务和空闲任务挂钩 \(p. 41\)](#)。

此图说明当任务使用 vTaskDelay() 代替 NULL 循环时的执行顺序。



只更改了这两个任务的实现，并未改变其功能。如果将此图与[时间测量和时钟周期中断 \(p. 30\)](#)中的图比较，可以看到现在以更加高效的方式实现了此功能。

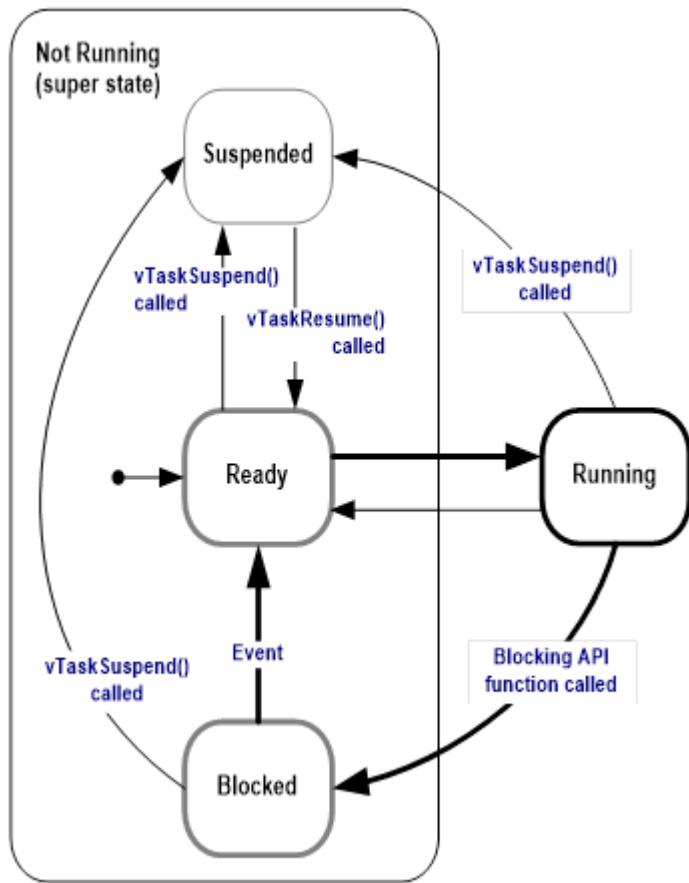
此图说明任务进入“被阻止”状态并在其整个延迟时段内保持该状态时的执行模式，因此它们仅当实际有必须执行的工作时才使用处理器时间（在本例中，它的工作只是输出一条消息）。



这些任务每次离开“被阻止”状态，在重新进入“被阻止”状态之前，都在一个时钟周期的一小部分时间内执行操作。在大部分时间里，没有应用程序任务能够运行（没有应用程序任务处于“准备就绪”状态），因此没有应用程序任务可以被选择进入“运行”状态。虽然是这种情况，空闲任务仍将运行。分配给空闲任务的处理时间量是衡量系统中备用处理容量的一个指标。使用 RTOS 可以大幅增加备用处理容量，方法只是允许应用程序完全由事件驱动。

下图中的粗线指示示例 4 中的任务所执行的状态转换，现在每个任务在返回到“准备就绪”状态之前都要先转换为“被阻止”状态。

粗线指示任务执行的状态转换。



## vTaskDelayUntil() API 函数

vTaskDelayUntil() 类似于 vTaskDelay()。vTaskDelay() 参数指定从一个任务调用 vTaskDelay() 到相同任务再次通过状态转换离开“被阻止”状态之间的时钟周期中断数。任务保持“被阻止”状态的时间长度由 vTaskDelay() 参数指定，但是该任务离开“被阻止”状态的时间是相对于 vTaskDelay() 被调用的时间。

vTaskDelayUntil() 的参数指定调用任务应从“被阻止”状态转入“准备就绪”状态的精确时钟计数值。vTaskDelayUntil() 是在需要固定执行周期（需要任务以固定频率周期性执行）时应该使用的 API 函数，因为调用任务的解除阻止时间是绝对的，而函数（在本例中为 vTaskDelay()）被调用的时间是相对的。

下面是 vTaskDelayUntil() API 函数原型。

```
void vTaskDelayUntil( TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement );
```

下表列出了 vTaskDelayUntil() 的参数。

参数名称	描述
pxPreviousWakeTime	此参数的名称来自一个假设：将使用 vTaskDelayUntil() 来实现以某个固定频率周期性执行的任务。在本例中，pxPreviousWakeTime 保存任务上次离开“被阻止”状态（被唤醒）的时间。此时间用作一个参考点来计算任务接下来应离开“被阻止”状态的时间。

	pxPreviousWakeTime 指向的变量在 vTaskDelayUntil() 函数中自动更新。它通常不由应用程序代码修改，但在首次使用前必须初始化为当前的时钟计数。
xTimeIncrement	此参数的名称也来自一个假设：将使用 vTaskDelayUntil() 来实现以某个固定频率周期性执行的任务。频率由 xTimeIncrement 值设置。  xTimeIncrement 以时钟周期为单位指定。宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为计时周期数。

## 将示例任务转换为使用 vTaskDelayUntil() ( 示例 5 )

示例 4 中创建的两个任务是周期性任务，但使用 vTaskDelay() 并不能保证它们运行的频率是固定的，因为任务离开“被阻止”状态的时间是相对于其调用 vTaskDelay() 的时间。将这些任务改为使用 vTaskDelayUntil() 而不是 vTaskDelay() 可解决此潜在问题。

下面的代码介绍如何使用 vTaskDelayUntil() 实现示例任务。

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    TickType_t xLastWakeTime;
    /* The string to print out is passed in by the parameter. Cast this to a character
    pointer. */
    pcTaskName = ( char * ) pvParameters;
    /* The xLastWakeTime variable needs to be initialized with the current tick count.
    This is the only time the variable is written to explicitly. After this, xLastWakeTime is
    automatically updated within vTaskDelayUntil(). */
    xLastWakeTime = xTaskGetTickCount();
    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* This task should execute every 250 milliseconds exactly. As per the
        vTaskDelay() function, time is measured in ticks, and the pdMS_TO_TICKS() macro is
        used to convert milliseconds into ticks. xLastWakeTime is automatically updated within
        vTaskDelayUntil(), so is not explicitly updated by the task. */
        vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );
    }
}
```

## 组合使用阻止和非阻止任务 (示例 6 )

之前的示例已单独说明了轮询和阻止任务的行为。此示例说明组合使用两种模式时的执行顺序，进一步说明前述的预期系统行为。

1. 两个任务都以优先级 1 创建。这些任务除了不断输出字符串外，不执行其他任何操作。

这些任务不会调用任何可能导致它们进入“被阻止”状态的 API 函数，因此它们始终处于“准备就绪”或“运行”状态。这类任务称为连续处理任务，因为它们始终有工作要做（即使在本例中只是一些无关紧要的工作）。

2. 然后以优先级 2 创建第三个任务，因此它的优先级高于其他两个任务。第三个任务也只是输出字符串，但这次是周期性输出，因此它使用 vTaskDelayUntil() API 函数在两次输出迭代之间的时间内将自身置于“被阻止”状态。

下面的代码是连续处理任务。

```
void vContinuousProcessingTask( void *pvParameters )
{
    char *pcTaskName;

    /* The string to print out is passed in by the parameter. Cast this to a character
pointer. */

    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. This task just does this repeatedly without
ever blocking or delaying. */

        vPrintString( pcTaskName );
    }
}
```

下面的代码是周期性任务。

```
void vPeriodicTask( void *pvParameters )
{
    TickType_t xLastWakeTime;
    const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );

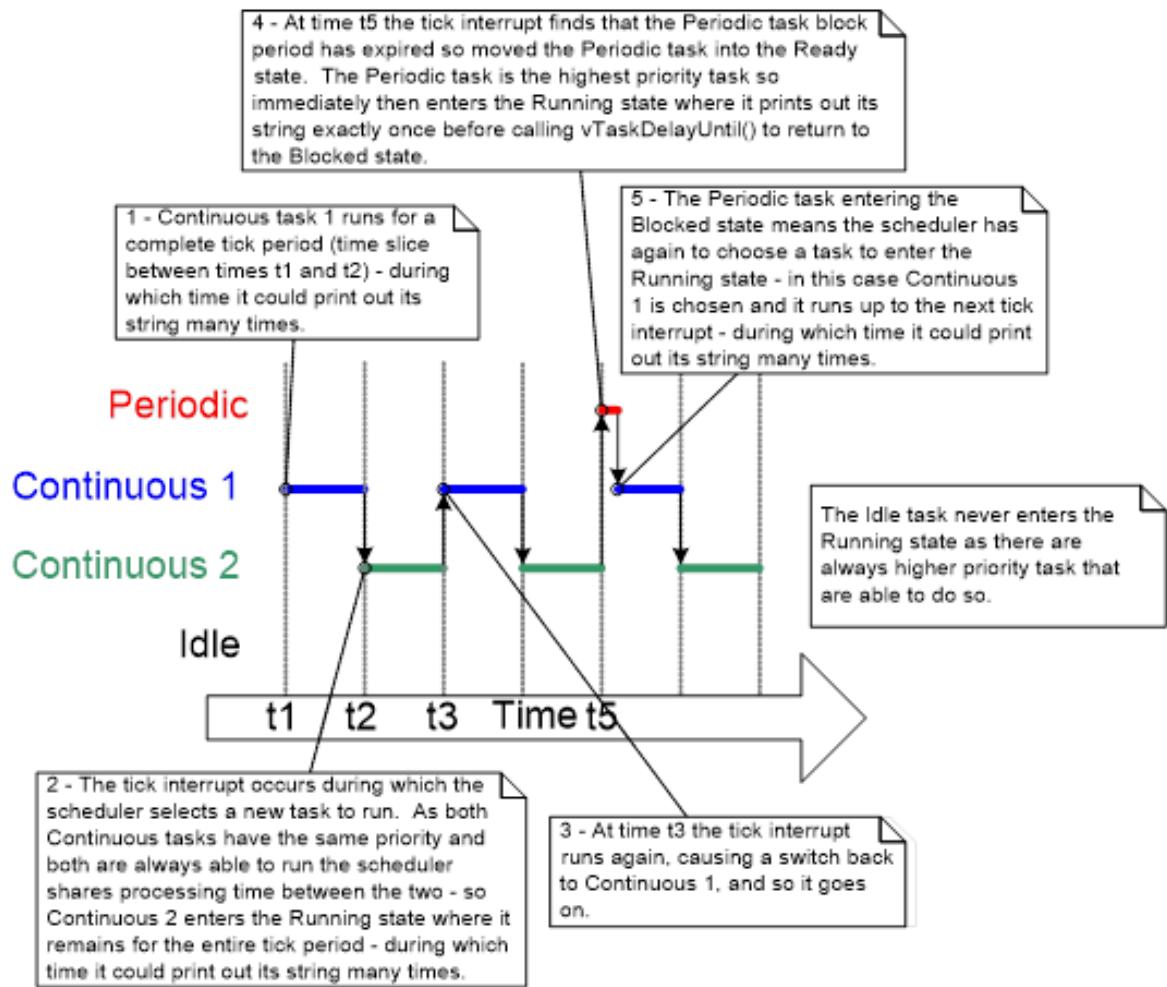
    /* The xLastWakeTime variable needs to be initialized with the current tick count. Note
that this is the only time the variable is explicitly written to. After this xLastWakeTime
is managed automatically by the vTaskDelayUntil() API function. */

    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */
}
```

```
for( ; ; )  
{  
    /* Print out the name of this task. */  
  
    vPrintString( "Periodic task is running\r\n" );  
  
    /* The task should execute every 3 milliseconds exactly. See the declaration of  
    xDelay3ms in this function. */  
  
    vTaskDelayUntil( &xLastWakeTime, xDelay3ms );  
}  
}
```

下图是执行顺序。



## 空闲任务和空闲任务挂钩

示例 4 中创建的任务大部分时间都处于“被阻止”状态。处于这种状态时，它们无法运行，因此计划程序不能选择它们。

至少必须有一个任务可以进入“运行”状态。即使是使用 FreeRTOS 特殊的低功耗功能，也是如此。如果应用程序创建的所有任务都无法执行，正在执行 FreeRTOS 的微控制器将置于低功耗模式。

为确保至少有一个任务可以进入“运行”状态，计划程序会在调用 vTaskStartScheduler() 时创建空闲任务。空闲任务在循环中几乎只是等待，就像第一个示例中的任务一样，它随时能够运行。

空闲任务具有尽可能最低的优先级（优先级零），以确保它不会使得较高优先级的应用程序任务无法进入“运行”状态。不过，您完全可以在空闲任务优先级创建任务，与之共享同一优先级。FreeRTOSConfig.h 中的 configIDLE\_SHOULD\_YIELD 编译时配置常量可用于防止空闲任务消耗本可以更有成效地分配给应用程序任务的处理时间。有关 configIDLE\_SHOULD\_YIELD 的更多信息，请参阅[计划算法 \(p. 51\)](#)。

在最低优先级运行可确保空闲任务只要有更高的优先级任务进入“准备就绪”状态就能通过状态转换离开“运行”状态。在示例 4 的执行顺序图中可以看到，在时间 tn，空闲任务立即换出，让 Task 2 能够在一离开“被阻止”状态时就能执行。我们称 Task 2 抢先于空闲任务。抢先是自动发生的，也不知道是哪个任务被抢先。

注意：如果应用程序使用 vTaskDelete() API 函数，则重要的是空闲任务不会缺乏处理时间。这是因为空闲任务在有任务被删除后要负责清理内核资源。

## 空闲任务挂钩函数

您可以通过使用空闲挂钩（或空闲回调）函数将特定于应用程序的功能直接添加到空闲任务。该函数由空闲任务循环在每次迭代都自动调用一次。

空闲任务挂钩的常见用途包括：

- 执行低优先级、后台或连续处理功能。
- 测量备用处理容量。（仅当较高优先级的所有应用程序任务都没有要执行的工作时，空闲任务才运行，因此测量分配给空闲任务的处理时间量将能够清楚地看到有多少备用的处理时间。）
- 只要没有应用程序处理要执行，处理器就置于低功耗模式，这是一种方便的自动节能方法。（使用非时钟空闲模式可以更加节能。）

## 对实现空闲任务挂钩函数的限制

空闲任务挂钩函数必须遵守以下规则。

1. 切勿尝试阻止或暂停空闲任务挂钩函数。

注意：以任何方式阻止空闲任务都可能会导致任何任务都无法进入“运行”状态的情况。

2. 如果应用程序使用 vTaskDelete() API 函数，则空闲任务挂钩必须始终在合理的时间范围内返回到其调用方。这是因为空闲任务在有任务被删除后要负责清理内核资源。如果空闲任务永远保持在空闲挂钩函数中，则无法进行这种清除。

空闲任务挂钩函数必须具有如下所示的名称和原型。

```
void vApplicationIdleHook( void );
```

## 定义空闲任务挂钩函数（示例 7）

示例 4 中使用阻止 vTaskDelay() API 调用产生了大量空闲时间，因为两个应用程序任务都处于“被阻止”状态，这段时间内空闲任务在执行。示例 7 通过增加空闲挂钩函数使用此空闲时间，下面是该函数的源代码。

以下代码介绍了一个非常简单的空闲挂钩函数。

```
/* Declare a variable that will be incremented by the hook function.*/

volatile uint32_t ulIdleCycleCount = 0UL;

/* Idle hook functions MUST be called vApplicationIdleHook(), take no parameters, and
return void. */

void vApplicationIdleHook( void )
{

    /* This hook function does nothing but increment a counter. */

    ulIdleCycleCount++;
}

To call the idle
hook function, called configUSE_IDLE_HOOK must be set to 1 in FreeRTOSConfig.h.
```

实现已创建任务的函数已稍作修改，以输出 ulIdleCycleCount 值，如下所示。下面的代码介绍如何输出 ulIdleCycleCount 值。

```
void vTaskFunction( void *pvParameters )

{
    char *pcTaskName;

    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* The string to print out is passed in by the parameter. Cast this to a character
pointer. */

    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )

    {
        /* Print out the name of this task AND the number of times ulIdleCycleCount has
been incremented. */

        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

        /* Delay for a period of 250 milliseconds. */

        vTaskDelay( xDelay250ms );
    }
}
```

下面是示例 7 生成的输出。可以看到，在每次应用程序任务迭代之间，空闲任务挂钩函数被调用约 400 万次。（迭代次数取决于执行演示的硬件的速度。）

```
C:\Temp>rtosdemo
Task 2 is running
ullidleCycleCount = 0
Task 1 is running
ullidleCycleCount = 0
Task 2 is running
ullidleCycleCount = 3869504
Task 1 is running
ullidleCycleCount = 3869504
Task 2 is running
ullidleCycleCount = 8564623
Task 1 is running
ullidleCycleCount = 8564623
Task 2 is running
ullidleCycleCount = 13181489
Task 1 is running
ullidleCycleCount = 13181489
Task 2 is running
ullidleCycleCount = 17838406
Task 1 is running
ullidleCycleCount = 17838406
```

## 更改任务优先级

### vTaskPrioritySet() API 函数

vTaskPrioritySet() API 函数可用于在计划程序启动后更改任何任务的优先级。vTaskPrioritySet() API 函数仅当 FreeRTOSConfig.h 中的 INCLUDE\_vTaskPrioritySet 设置为 1 时才可用。

下面是 vTaskPrioritySet() API 函数原型。

```
void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority);
```

下表列出了 vTaskPrioritySet() 参数。

参数名称	描述
pxTask	要修改其优先级的任务（主体任务）的句柄。有关获取任务句柄的信息，请参阅 xTaskCreate() API 函数的 pxCreatedTask 参数。 任务可以传递 NULL 代替有效的任务句柄来更改自己的优先级。
uxNewPriority	要为主体任务设置的优先级。此优先级最大上限自动设定为最大可用优先级 (configMAX_PRIORITIES - 1)，其中 configMAX_PRIORITIES 是在 FreeRTOSConfig.h 头文件中设置的编译时间常量。

### uxTaskPriorityGet() API 函数

uxTaskPriorityGet() API 函数可用于查询任务优先级。uxTaskPriorityGet() API 函数仅当 FreeRTOSConfig.h 中的 INCLUDE\_uxTaskPriorityGet 设置为 1 时才可用。

下面是 uxTaskPriorityGet() API 函数原型。

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t pxTask );
```

下表列出了 uxTaskPriorityGet() 参数和返回值。

参数名称/返回值	描述
pxTask	要查询其优先级的任务 ( 主体任务 ) 的句柄。有关获取任务句柄的信息 , 请参阅 xTaskCreate() API 函数的 pxCreatedTask 参数。 任务可以传递 NULL 代替有效的任务句柄来查询自己的优先级。
返回的值	当前分配给要查询的任务的优先级。

## 更改任务优先级 (示例 8 )

计划程序将始终选择处于“准备就绪”状态、优先级最高的任务作为进入“运行”状态的任务。示例 8 使用 vTaskPrioritySet() API 函数来更改两个任务之间的相对优先级 , 对此进行说明。

下面使用的示例代码以两个不同优先级创建两个任务。两个任务都没有调用可能导致其进入“被阻止”状态的任何 API 函数 , 因此它们都始终处于“准备就绪”状态或“运行”状态。因此 , 相对优先级最高的任务将始终是计划程序选为进入“运行”状态的任务。

1. Task 1 ( 如紧接着的代码中所示 ) 以最高优先级创建 , 因此能确保首先运行。Task 1 在 Task 2 的优先级提高到超过它自己的优先级之前输出几个字符串。
2. Task 2 只要获得相对最高优先级 , 就开始运行 ( 进入“运行”状态 ) 。在任何同一时间 , 只有一个任务可以处于“运行”状态 , 因此当 Task 2 处于“运行”状态时 , Task 1 处于“准备就绪”状态。
3. Task 2 在将自己的优先级重新设置为低于 Task 1 的优先级之前 , 输出一条消息。
4. 当 Task 2 重新降低其优先级设置时 , Task 1 再次成为最高优先级任务。Task 1 重新进入“运行”状态 , 从而强制 Task 2 回到“准备就绪”状态。

```
/*The implementation of Task 1*/
void vTask1( void *pvParameters )

{
    UBaseType_t uxPriority;

    /* This task will always run before Task 2 because it is created with the higher
    priority. Task 1 and Task 2 never block, so both will always be in either the Running or
    the Ready state. Query the priority at which this task is running. Passing in NULL means
    "return the calling task's priority". */

    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )

    {
        /* Print out the name of this task. */

        vPrintString( "Task 1 is running\r\n" );

        /* Setting the Task 2 priority above the Task 1 priority will cause Task 2 to
        immediately start running (Task 2 will have the higher priority of the two created tasks). */
    }
}
```

```
Note the use of the handle to Task 2 (xTask2Handle) in the call to vTaskPrioritySet(). The code that follows shows how the handle was obtained. */
```

```
vPrintString( "About to raise the Task 2 priority\r\n" );  
  
vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );  
  
/* Task 1 will only run when it has a priority higher than Task 2. Therefore, for this task to reach this point, Task 2 must already have executed and set its priority back down to below the priority of this task. */  
  
}  
  
}
```

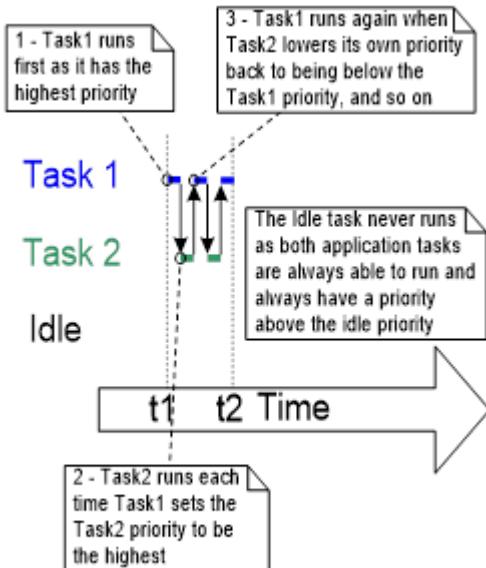
```
/*The implementation of Task 2 */  
  
void vTask2( void *pvParameters )  
{  
  
    UBaseType_t uxPriority;  
  
    /* Task 1 will always run before this task because Task 1 is created with the higher priority. Task 1 and Task 2 never block so they will always be in either the Running or the Ready state. Query the priority at which this task is running. Passing in NULL means "return the calling task's priority". */  
  
    uxPriority = uxTaskPriorityGet( NULL );  
  
    for( ;; )  
  
    {  
  
        /* For this task to reach this point Task 1 must have already run and set the priority of this task higher than its own. Print out the name of this task. */  
  
        vPrintString( "Task 2 is running\r\n" );  
  
        /* Set the priority of this task back down to its original value. Passing in NULL as the task handle means "change the priority of the calling task". Setting the priority below that of Task 1 will cause Task 1 to immediately start running again, preempting this task. */  
  
        vPrintString( "About to lower the Task 2 priority\r\n" );  
  
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );  
  
    }  
  
}
```

每个任务都可以直接使用 NULL，而不使用有效的任务句柄来查询和设置自己的优先级。仅当任务引用自身之外的其他任务时，才需要任务句柄。如当 Task 1 更改 Task 2 的优先级时。为允许 Task 1 执行此操作，Task 2 创建时，将获取并保存 Task 2 的句柄，如下面的代码注释强调的那样。

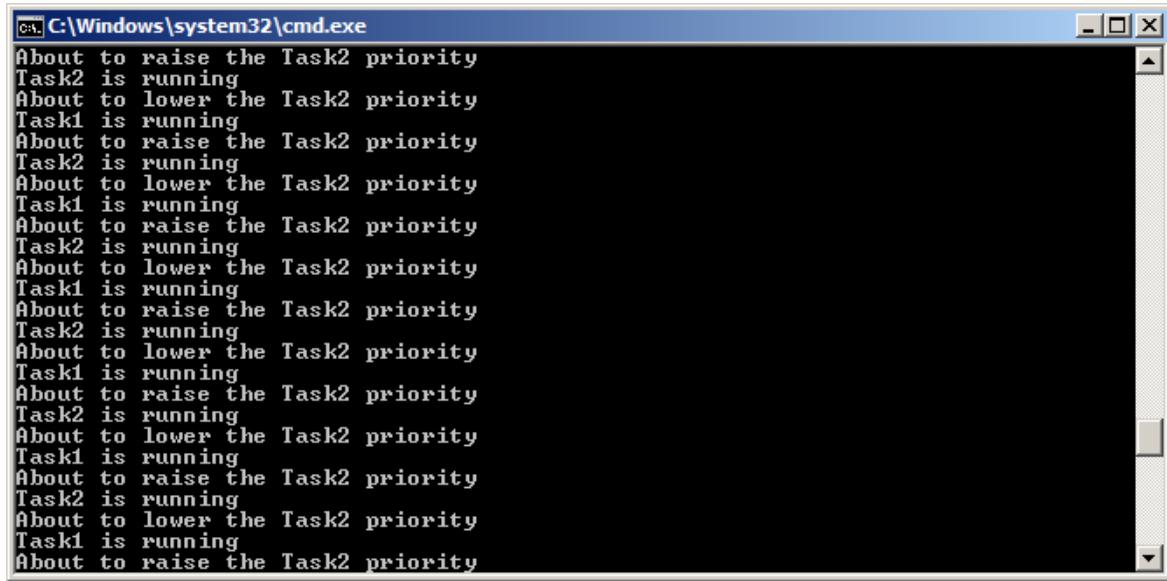
```
/* Declare a variable that is used to hold the handle of Task 2. */  
  
TaskHandle_t xTask2Handle = NULL;  
  
int main( void )  
{
```

```
/* Create the first task at priority 2. The task parameter is not used and set to NULL.  
The task handle is also not used so is also set to NULL. */  
  
xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );  
  
/* The task is created at priority 2 ____^. */  
  
/* Create the second task at priority 1, which is lower than the priority given to  
Task 1. Again, the task parameter is not used so it is set to NULL, but this time the task  
handle is required so the address of xTask2Handle is passed in the last parameter. */  
  
xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );  
  
/* The task handle is the last parameter _____^~~~~~ */  
  
/* Start the scheduler so the tasks start executing. */  
  
vTaskStartScheduler();  
  
/* If all is well, then main() will never reach here because the scheduler will now be  
running the tasks. If main() does reach here, then it is likely there was insufficient  
heap memory available for the idle task to be created. For information, see Heap Memory  
Management. */  
  
for( ; );  
  
}
```

下图是任务的执行顺序。



下面是输出的内容。



A screenshot of a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains a series of log messages from FreeRTOS tasks. The messages show Task2 raising and lowering its priority, while Task1 runs continuously. The log output is as follows:

```
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
```

## 删除任务

### vTaskDelete() API 函数

任务可以使用 vTaskDelete() API 函数来删除自身或任何其他任务。vTaskDelete() API 函数仅当 FreeRTOSConfig.h 中的 INCLUDE\_vTaskDelete 设置为 1 时才可用。

已删除的任务不再存在，不能再次进入“运行”状态。

空闲任务负责释放分配给已删除任务的内存。因此，有一点很重要，使用 vTaskDelete() API 函数的应用程序不要完全挤占空闲任务的所有处理时间。

注意：只有内核分配给任务的内存才会在任务被删除时自动释放。通过任务的实现分配的任何内存或其他资源都必须显式释放。

下面是 vTaskDelete() API 函数原型。

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

下表列出了 vTaskDelete() 参数。

参数名称/返回值	描述
pxTaskToDelete	要删除的任务（主体任务）的句柄。有关获取任务句柄的信息，请参阅 xTaskCreate() API 函数的 pxCreatedTask 参数。任务通过传递 NULL 代替有效的任务句柄可以删除其自身。

### 删除任务（示例 9）

下面是一个非常简单的示例。

1. Task 1 是由 main() 以优先级 1 创建的。当它运行时，它以优先级 2 创建 Task 2。Task 2 现在是优先级最高的任务，因此它立即开始执行。main() 的源代码在第一个代码清单中。Task 1 的源代码在第二个代码清单中。
2. Task 2 除了自行删除之外不执行其他任何操作。它可以通过向 vTaskDelete() 传递 NULL 而不是出于演示目的使用自己的任务句柄，自行删除。Task 2 的源代码在第三个代码清单中。
3. 当 Task 2 已被删除时，Task 1 再次成为优先级最高的任务，因此它继续执行，此时它调用 vTaskDelay() 阻止一小段时间。
4. 当 Task 1 处于“被阻止”状态时，空闲任务执行，并释放分配给现在已删除的 Task 2 的内存。
5. 当 Task 1 离开“被阻止”状态时，它再次变为优先级最高的“准备就绪”状态任务，因此它抢先于空闲任务执行。当它进入“运行”状态时，它再次创建 Task 2，如此继续执行。

```
int main( void )
{
    /* Create the first task at priority 1. The task parameter is not used so is set to
NULL. The task handle is also not used so likewise is set to NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );

    /* The task is created at priority 1 ____^. */

    /* Start the scheduler so the task starts executing. */
    vTaskStartScheduler();

    /* main() should never reach here as the scheduler has been started.*/
    for( ;; );
}
```

```
TaskHandle_t xTask2Handle = NULL;

void vTask1( void *pvParameters )
{
    const TickType_t xDelay100ms = pdMS_TO_TICKS( 100UL );
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\r\n" );

        /* Create task 2 at a higher priority. Again, the task parameter is not used so it
is set to NULL, but this time the task handle is required so the address of xTask2Handle
is passed as the last parameter. */

        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );

        /* The task handle is the last parameter ____^^^^^^^^^ */

        /* Task 2 has/had the higher priority, so for Task 1 to reach here, Task 2 must
have already executed and deleted itself. Delay for 100 milliseconds. */

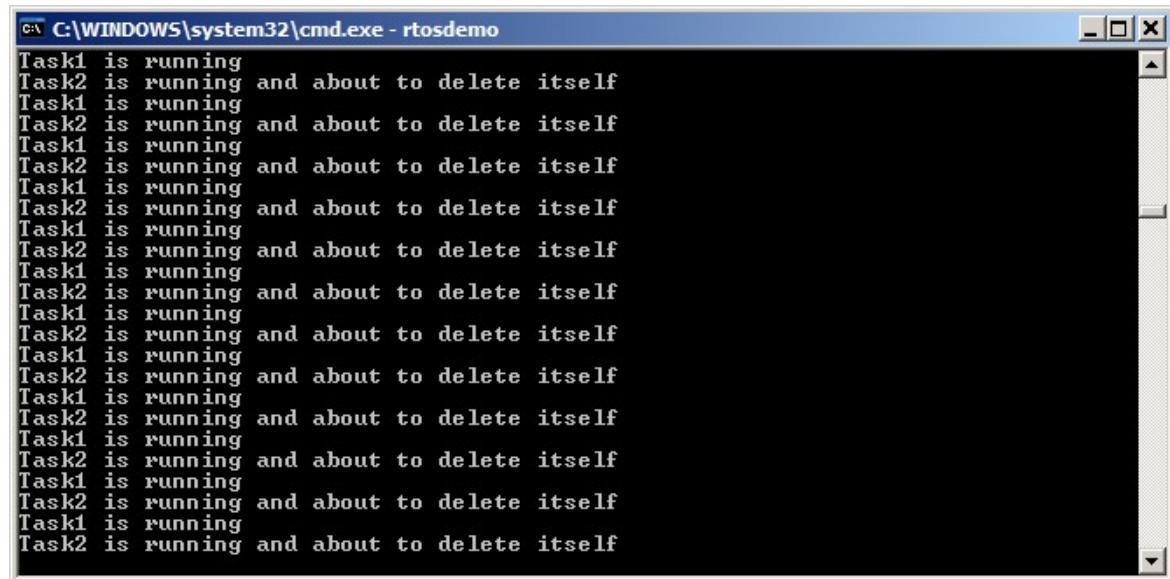
        vTaskDelay( xDelay100ms );
}
```

}

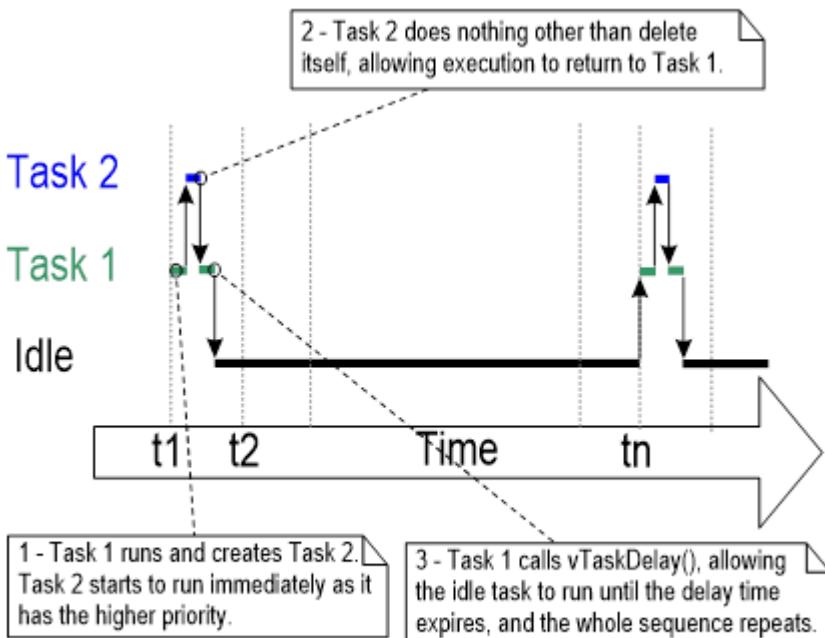
```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself. To do this, it could call vTaskDelete() using
NULL as the parameter, but for demonstration purposes, it calls vTaskDelete(), passing its
own task handle. */

    vPrintString( "Task 2 is running and about to delete itself\r\n" );
    vTaskDelete( xTask2Handle );
}
```

下面是输出的内容。



下图是执行顺序。



## 计划算法

### 任务状态和事件回顾

实际在运行（使用处理时间）的任务处于“运行”状态。在任何给定时间，在单核处理器中只能有一个任务处于“运行”状态。

未在运行、但不是“被阻止”或“暂停”状态的任务处于“准备就绪”状态。处于“准备就绪”状态的任务可由计划程序选择为要进入“运行”状态的任务。计划程序将始终选择优先级最高的“准备就绪”状态任务进入“运行”状态。

任务可以在“被阻止”状态下等待事件发生，当事件发生时，将自动退回到“准备就绪”状态。临时事件在特定时间（例如，阻止时间到期时）发生，通常用于实现周期性或超时行为。当任务或中断服务例程使用任务通知、队列、事件组或诸多类型的信号灯中的一种发送信息时，将发生同步事件。它们通常用于发送异步活动信号，如数据到达某个外设。

### 配置计划算法

计划算法是确定哪个“准备就绪”状态的任务可以转换为“运行”状态的软件例程。

到目前为止，介绍的所有示例都使用相同的计划算法，不过，您可以使用 `configUSE_PREEMPTION` 和 `configUSE_TIME_SLICING` 配置常量来更改此算法。这两个常量都在 `FreeRTOSConfig.h` 中定义。

第三个配置常量 `configUSE_TICKLESS_IDLE` 也会影响计划算法。使用它可能导致时钟周期中断完全关闭更长的时间。`configUSE_TICKLESS_IDLE` 是一个高级选项，专为在必须最小化功耗的应用程序中使用而提供。本节中提供的说明假定 `configUSE_TICKLESS_IDLE` 设置为 0，这是常量保留为未定义状态时的默认设置。

在所有可能的配置中，FreeRTOS 计划程序都确保将同一优先级的任务选为进入“运行”状态。这种“轮流”策略通常称为循环计划。循环计划算法不保证相同优先级的两个任务平均分享时间，只是相同优先级的“准备就绪”状态任务轮流进入“运行”状态。

## 带时间切片的固定优先级抢先计划

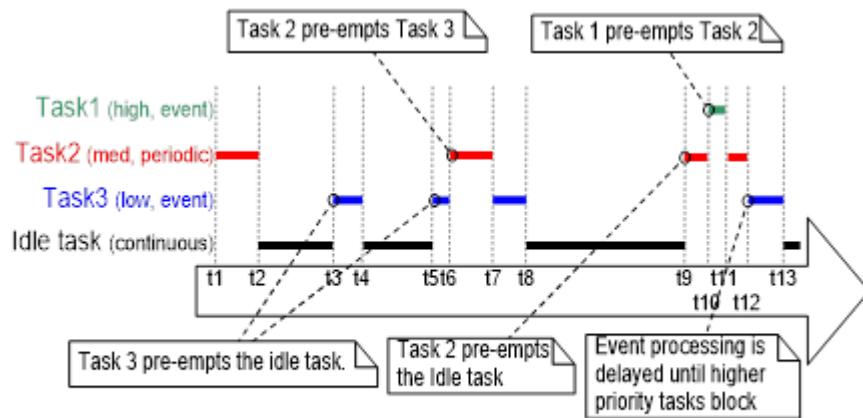
下表所示的配置将 FreeRTOS 计划程序设置为使用称为“带时间切片的固定优先级抢先计划”的计划算法，这是大多数小 RTOS 应用程序使用的计划算法，也是到目前为止提供的所有示例使用的算法。

常量	值
configUSE_PREEMPTION	1
configUSE_TIME_SLICING	1

下面是该算法中使用的术语：

- 固定优先级任务：描述为固定优先级的计划算法不会更改分配给所计划的任务的优先级，但也不阻止任务更改自己或其他任务的优先级。
- 抢先状态：如果较高优先级的任务进入“准备就绪”状态，抢先计划算法将立即抢先于“运行”状态的任务运行。被抢先意味着不知不觉地（没有显式的让步或阻止）从“运行”状态转为“准备就绪”状态，以便另一个不同任务进入“运行”状态。
- 时间切片：时间切片用于使相同优先级的任务之间共享处理时间，即使任务未显式让步或进入“被阻止”状态。描述为使用时间切片的计划算法在每一个时间切片结束时选择一个新任务进入“运行”状态（如果存在其他与正在运行的任务具有相同优先级，并且处于“准备就绪”状态的任务）。一个时间切片等于两个 RTOS 时钟周期中断之间的时间。

下图是当应用程序中所有任务都具有唯一优先级时任务被选择进入“运行”状态的顺序。



此图中的执行模式重点说明在每个任务都分配了唯一优先级的假想应用程序中的任务优先顺序和抢先情况。

### 1. 空闲任务

空闲任务在最低优先级运行，因此每次有较高优先级的任务进入“准备就绪”状态时都会被抢先（例如在时间 t3、t5 和 t9 时）。

### 2. Task 3

Task 3 是事件驱动型任务，它以相对较低的优先级运行，但是它高于空闲任务的优先级。它的大部分时间都处于“被阻止”状态等待其相关事件，每次该事件发生时，都从“被阻止”状态转换为“准备就绪”状态。所有 FreeRTOS 任务间通信机制（任务通知、队列、信号灯、事件组等）都可以这种方式用于发送事件信号和解除阻止任务。

事件在时间 t3 和 t5 发生，以及 t9 和 t12 之间某个时刻发生。在时间 t3 和 t5 发生的事件会立即处理，因为在这些时间，Task 3 是能够运行的最高优先级任务。在时间 t9 与 t12 之间的某个时刻发生的事件要等到 t12 才能处理，因为优先级较高的任务（即 Task 1 和 Task 2）在那之前仍在执行。只有在时间 t12 时，Task 1 和 Task 2 都处于“被阻止”状态，这使 Task 3 成为优先级最高的“准备就绪”状态任务。

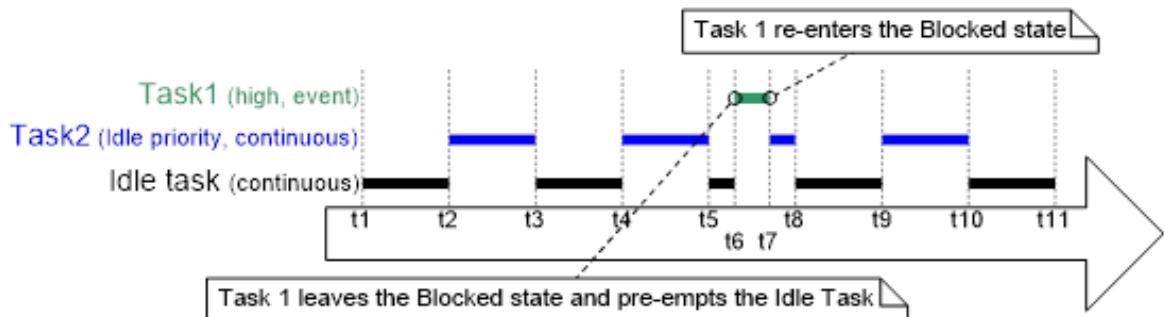
### 3. Task 2

Task 2 是一个周期性任务，其执行优先级高于 Task 3，但是低于 Task 1。该任务的周期间隔表示 Task 2 需要在时间 t1、t6 和 t9 执行。在时间 t6，Task 3 处于“运行”状态，但 Task 2 相对优先级较高，因此它抢先于 Task 3 运行，并立即开始执行。Task 2 完成其处理并在时间 t7 重新进入“被阻止”状态，此时 Task 3 可以再次进入“运行”状态以完成处理。Task 3 在时间 t8 阻止。

### 4. Task 1

Task 1 也是事件驱动型任务。它以最高的优先级执行，因此它可以抢先于系统中的其他所有任务执行。显示的唯一一个 Task 1 事件发生在时间 t10，那时 Task 1 抢先于 Task 2 运行。Task 2 只有等 Task 1 在时间 t11 重新进入“被阻止”状态后才能完成其处理。

下图中的执行模式重点说明在两个任务以相同优先级运行的假想应用程序中的任务优先顺序和时间切片。



### 1. 空闲任务和 Task 2

空闲任务和 Task 2 都是连续处理任务，优先级都是 0（最低优先级）。当没有更高优先级的任务可运行时，计划程序仅向优先级为 0 的任务分配处理时间，并通过时间切片使优先级为 0 的任务分享这些时间。每次发生时钟周期中断时（在时间 t1、t2、t3、t4、t5、t8、t9、t10 和 t11）都会开始一个新的时间切片。

空闲任务和 Task 2 轮流进入“运行”状态，这可能导致这两个任务在同一个时间切片的一部分时间内同时处于“运行”状态，就像时间 t5 和时间 t8 之间那样。

### 2. Task 1

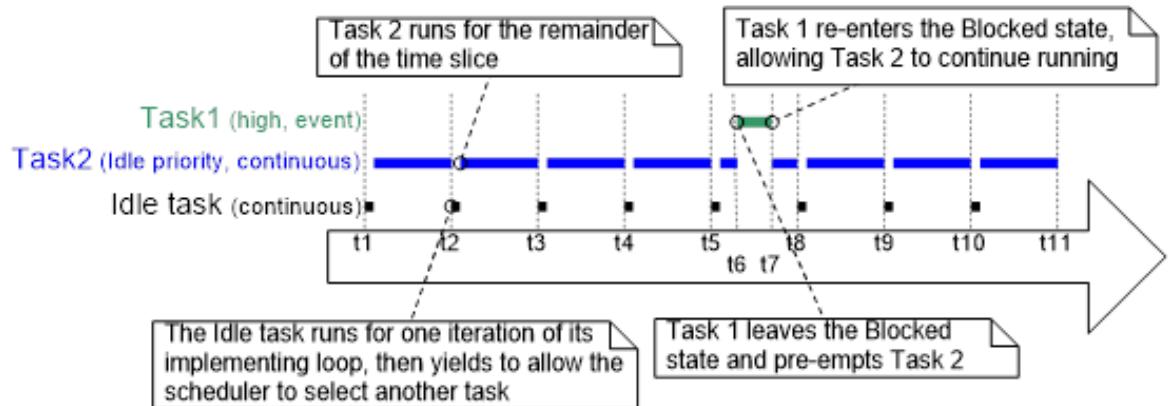
Task 1 的优先级高于空闲任务优先级。Task 1 是事件驱动型任务，它的大部分时间都处于“被阻止”状态等待其相关事件，每次该事件发生时，它都从“被阻止”状态转换为“准备就绪”状态。相关事件在时间 t6 发生，这时 Task 1 变为优先级最高的能够运行的任务，因此 Task 1 在空闲任务的部分时间切片中抢先执行。事件处理在时间 t7 完成，此时 Task 1 重新进入“被阻止”状态。

上图是空闲任务与应用程序编写者创建的任务分享处理时间。如果应用程序编写者所创建的空闲优先级任务有工作要做，但空闲任务并没有工作，您可能不想为空闲任务分配那么多处理时间。可以使用编译时配置常量 configIDLE\_SHOULD\_YIELD 更改空闲任务的计划：

- 如果 configIDLE\_SHOULD\_YIELD 设置为 0，则空闲任务在其整个时间切片中都保持“运行”状态，除非它被优先级更高的任务抢先。
- 如果 configIDLE\_SHOULD\_YIELD 设置为 1，则当存在其他处于“准备就绪”状态的空闲优先级任务时，空闲任务将在其每一次循环迭代中让步（自愿放弃它所获得的时间切片的剩余部分）。

configIDLE\_SHOULD\_YIELD 设置为 0 时，会出现上图所示的执行模式。

configIDLE\_SHOULD\_YIELD 设置为 1 时，在相同情况下会出现下图所示的执行模式。它是当应用程序中的两个任务具有相同优先级时任务被选择进入“运行”状态的顺序。



此图还说明当 configIDLE\_SHOULD\_YIELD 设置为 0 时，在空闲任务之后被选择进入“运行”状态的任务不是在整个时间切片中执行，而是在空闲任务让步期间的剩余时间切片部分执行。

## 固定优先级抢先计划 ( 无时间切片 )

无时间切片的固定优先级抢先计划仍然使用上一节所述的任务选择和抢先算法，但是不使用时间切片在相同优先级任务之间分享处理时间。

下表列出了将 FreeRTOS 计划程序配置为使用无时间切片的固定优先级抢先计划的 FreeRTOSConfig.h 设置。

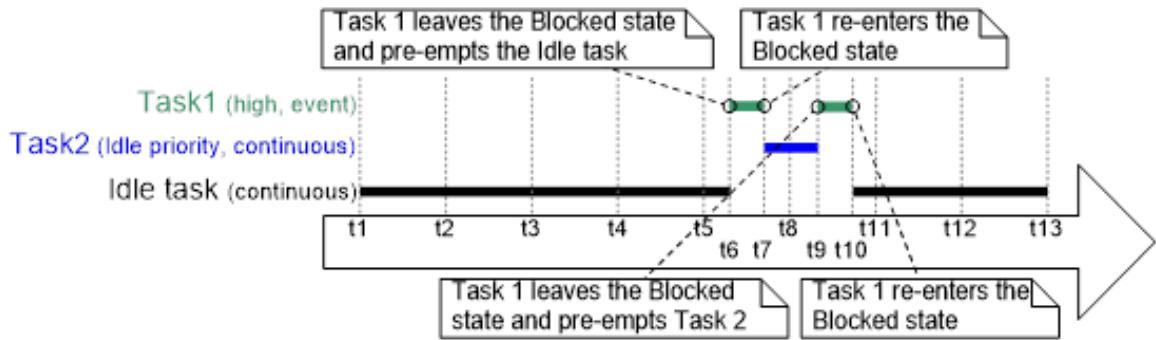
常量	值
configUSE_PREEMPTION	1
configUSE_TIME_SLICING	0

如果使用时间切片，并且存在多个优先级最高的“准备就绪”任务能够运行，则在每个 RTOS 时钟周期中断（时钟周期中断标记时间切片的结束）期间，计划程序将选择新任务进入“运行”状态。如果不使用时间切片，则计划程序仅在以下情况下选择新任务进入“运行”状态：

- 一个优先级较高的任务进入“准备就绪”状态。
- “运行”状态的任务进入“被阻止”或“暂停”状态。

不使用时间切片时，任务上下文切换较少。因此，关闭时间切片可减少计划程序的处理开销。但是，关闭时间切片也可能导致相同优先级的任务获得的处理时间量存在非常大的差异。不使用时间切片运行计划程序被视为一种高级方法。只应由经验丰富的用户使用。

此图说明在不使用时间切片时，相同优先级的任务如何获得差异非常大的处理时间量。



在上图中，configIDLE\_SHOULD\_YIELD 设置为 0。

#### 1. 时钟周期中断

时钟周期中断发生在时间 t1、t2、t3、t4、t5、t8、t11、t12 和 t13。

#### 2. Task 1

Task 1 是事件驱动型高优先级任务，它的大部分时间都处于“被阻止”状态，等待其相关事件发生。每次发生该事件时，Task 1 的状态从“被阻止”转换为“准备就绪”状态（随后，因为它是优先级最高的“准备就绪”状态任务，会进入“运行”状态）。上图说明 Task 1 在时间 t6 和 t7 之间，然后在 t9 和 t10 之间处理事件。

#### 3. 空闲任务和 Task 2

空闲任务和 Task 2 都是连续处理任务，优先级都是 0（空闲优先级）。连续处理任务不会进入“被阻止”状态。

未使用时间切片，因此处于“运行”状态的空闲优先级将保持“运行”状态，直至被优先级更高的 Task 1 抢先。

在上图中，空闲任务在时间 t1 开始运行，并保持“运行”状态，直至在时间 t6 被 Task 1 抢先，这时空闲任务进入“运行”状态已经超过四个完整的时钟周期。

Task 2 在时间 t7 开始运行，这时 Task 1 重新进入“被阻止”状态等待另一个事件发生。Task 2 保持“运行”状态，直至在时间 t9 被 Task 1 抢先，这时它进入“运行”状态还不到一个时钟周期。

在时间 t10，空闲任务重新进入“运行”状态，即使它获得的处理时间比 Task 2 多四倍。

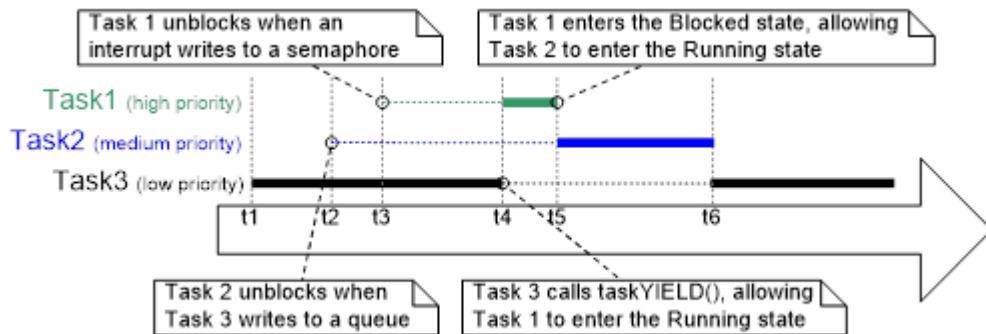
## 协作计划

FreeRTOS 还可以使用协作计划。下表列出了配置 FreeRTOS 计划程序使用协作计划的 FreeRTOSConfig.h 设置。

常量	值
configUSE_PREEMPTION	0
configUSE_TIME_SLICING	任意值

使用协作计划程序时，仅当“运行”状态的任务进入“被阻止”状态，或“运行”状态的任务通过调用 taskYIELD() 显式让步（手动请求重新计划）时，才会发生上下文切换。任务从不抢先，因此无法使用时间切片。

下图是协作计划程序的行为。水平虚线指示任务何时处于“准备就绪”状态。



### 1. Task 1

Task 1 具有最高优先级。它以“被阻止”状态开始，等待信号灯。

在时间 t3，中断释放信号灯，使 Task 1 离开“被阻止”状态，进入“准备就绪”状态。有关从中断释放信号灯的信息，请参阅[中断管理 \(p. 108\)](#)。

在时间 t3，Task 1 是优先级最高的“准备就绪”状态任务，如果使用的是抢先计划程序，Task 1 将变为“运行”状态任务。但是，由于使用的是协作计划程序，Task 1 保持“准备就绪”状态，直至时间 t4，此时“运行”状态的任务调用 taskYIELD()。

### 2. Task 2

Task 2 的优先级介于 Task 1 和 Task 3 之间。它以“被阻止”状态开始，等待 Task 3 在时间 t2 发送消息给它。

在时间 t2，Task 2 是优先级最高的“准备就绪”状态任务，如果使用的是抢先计划程序，Task 2 将变为“运行”状态任务。但是，由于使用协作计划程序，Task 2 会保持“准备就绪”状态，直至“运行”状态的任务进入“被阻止”状态或调用 taskYIELD()。

“运行”状态的任务在时间 t4 调用 taskYIELD()，但是那时 Task 1 是优先级最高的“准备就绪”状态任务，因此 Task 2 实际上不会变为“运行”状态，直至 Task 1 再次在时间 t5 重新进入“被阻止”状态。

在时间 t6，Task 2 重新进入“被阻止”状态等待下一条消息，此时 Task 3 再次成为优先级最高的“准备就绪”状态任务。

在多任务处理应用程序中，应用程序编写者必须注意，资源不能被多个任务同时访问，因为同时访问可能损坏资源。考虑以下情况：所访问的资源是 UART（串口）。两个任务将字符串写入 UART。Task 1 写入“abcdefghijklmноп”，Task 2 写入“123456789”：

1. Task 1 处于“运行”状态并开始写入其字符串。它向 UART 写入“abcdefg”，但还未写入更多字符就离开“运行”状态。
2. Task 2 进入“运行”状态，在离开“运行”状态之前向 UART 写入“123456789”。
3. Task 1 重新进入“运行”状态，将其字符串中的剩余字符写入 UART。

在这种情况下，实际写入 UART 的内容是“abcdefghijklmноп123456789”。Task 1 写入的字符串未如愿以未断开的顺序写入 UART。该字符串已损坏，因为 Task 2 写入 UART 的字符串出现在它之内。

可以使用协作计划程序来避免同时访问导致的问题。本指南稍后会介绍在任务之间安全共享资源的方法。FreeRTOS 提供的资源（如队列和信号灯）始终能够在任务之间安全共享。

- 如果使用抢先计划程序，“运行”状态的任务随时（包括当它与其他任务共享的资源处于不一致状态时）都可能被抢先。如 UART 示例所示，使资源处于不一致状态可能导致数据损坏。

- 如果使用协作计划程序，应用程序编写者控制何时可以切换到另一个任务。因此，应用程序编写者可以确保在资源处于不一致状态时，不会切换到另一个任务。
- 在 UART 示例中，应用程序编写者可以确保 Task 1 不会离开“运行”状态，直至其整个字符串已写入 UART，并且在此过程中，避免字符串因另一个任务激活而被损坏。

使用协作计划程序时，系统响应能力可能较低。

- 使用抢先计划程序时，计划程序在一个任务变为优先级最高的“准备就绪”状态时立即开始运行该任务。在规定时间段内必须响应高优先级事件的实时系统中，这通常是非常重要的。
- 使用协作计划程序时，只有在“运行”状态的任务进入“被阻止”状态或调用 `taskYIELD()` 时，才会切换到已成为优先级最高的“准备就绪”状态任务。

# 队列管理

队列提供任务到任务、任务到中断，以及中断到任务通信机制。本节介绍任务到任务通信。有关任务到中断和中断到任务通信的信息，请参阅[中断管理 \(p. 108\)](#)。

本节内容：

- 如何创建队列。
- 队列如何管理它所包含的数据。
- 如何向队列发送数据。
- 如何从队列接收数据。
- 在队列中阻止是什么意思。
- 如何在多个队列中阻止。
- 如何覆盖队列中的数据。
- 如何清除队列。
- 写入队列和从队列中读取时的任务优先级的效果。

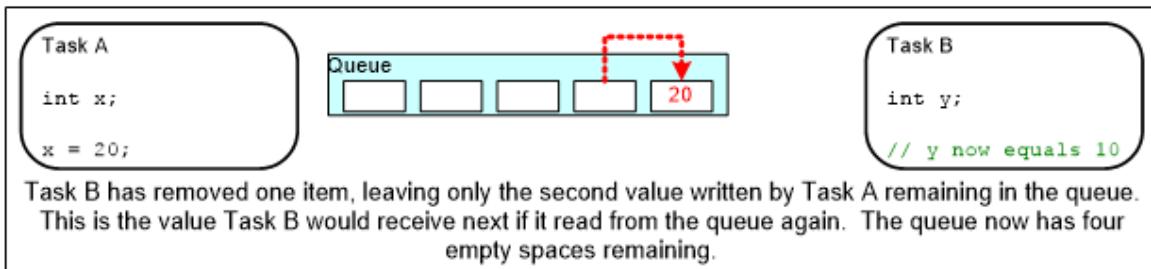
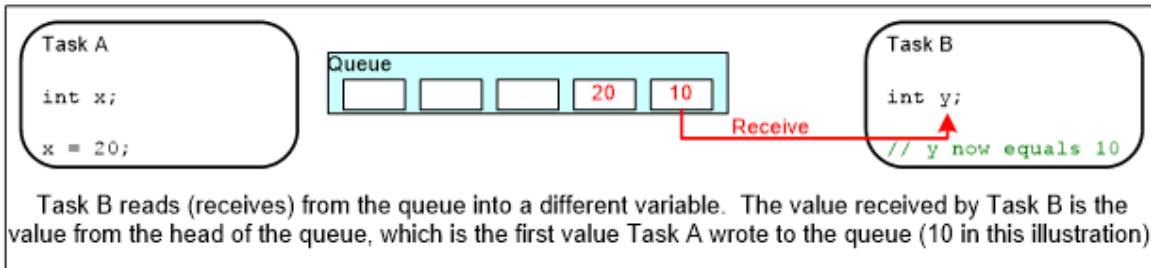
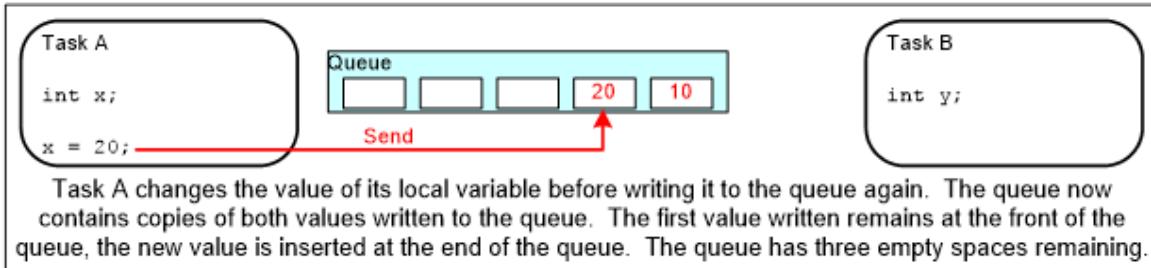
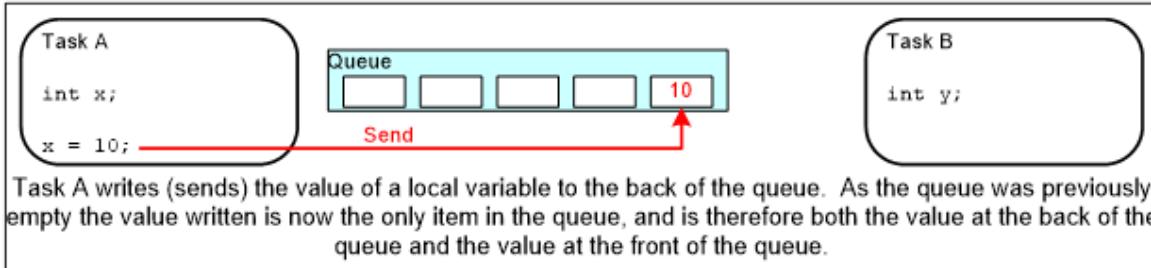
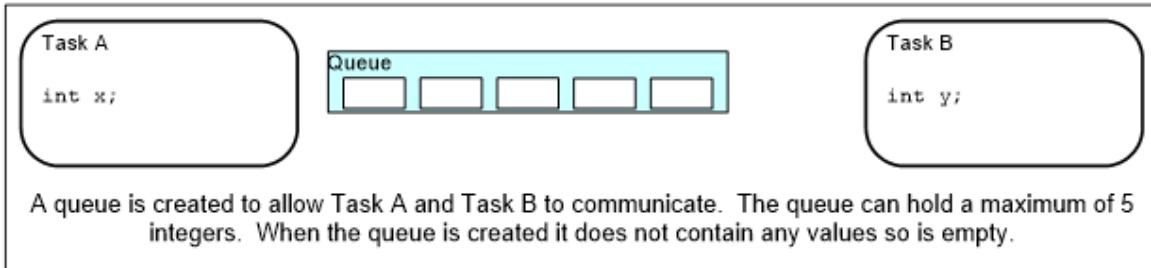
## 队列的特征

### 数据存储

队列可以容纳数量有限、大小固定的数据项目。队列最多可以容纳的项目数称为它的长度。每个数据项目的长度和大小都在创建队列时设置。

队列通常以先进先出 (FIFO) 缓冲区的形式使用，数据写入到队列后端（尾部），从队列前端（头部）移除。

下图说明对 FIFO 队列的数据写入和读取。也可以写入队列的前端和覆盖已在队列前端的数据。



可通过两种方式实现队列行为：

1. 通过复制排队

发送到队列的数据是以字节为单位复制到队列中的。

## 2. 通过引用排队

队列保存的是指针（而不是数据本身），这些指针指向发送给队列的数据。

FreeRTOS 通过复制使用队列。与通过引用排队相比，人们认为这种方法更强大、更简单，原因是：

- 堆栈变量可以直接发送到队列，即使变量在声明它的函数已退出后不会存在也是如此。
- 不必先分配缓冲区来保存数据，然后将数据复制到分配的缓冲区，就可将数据发送到队列。
- 发送任务可以立即重新使用已发送到队列的变量或缓冲区。
- 发送任务和接收任务是完全分离的。应用程序设计者不需要关心哪个任务拥有数据，或哪个任务负责释放数据。
- 通过复制排队不影响同时通过引用排队。例如，如果要排队的数据的大小使数据不能复制到队列中时，可以将指向数据的指针复制到队列中。
- RTOS 全面负责分配用于存储数据的内存。
- 在内存保护的系统中，任务可以访问的 RAM 是受限制的。在这种情况下，仅当发送任务和接收任务可以访问存储数据的 RAM 时才能使用通过引用排队。通过复制排队没有此限制。内核始终以完全权限运行，因此队列可以用来跨内存保护边界传递数据。

# 由多个任务访问

队列是可以由知道其存在的任何任务或中断服务寄存器 (ISR) 访问的对象。任意数量的任务可以向同一队列写入，并且任意数量的任务可以从同一队列读取。一个队列有多个写入方是很常见的，但是一个队列有多个读取方的情况却很少见。

# 队列读取时阻止

当任务尝试读取队列时，可以选择指定阻止时间。这是任务在队列已空时将保持“被阻止”状态以等待队列中数据可用的时间。当有任务或中断向队列中放入数据时，等待队列中数据变为可用、处于“被阻止”状态的任务将自动变为“准备就绪”状态。如果数据还未变为可用，指定的阻止时间已到期，该任务也将从“被阻止”状态自动变为“准备就绪”状态。

队列可以有多个读取方，因此单个队列可能会使多个任务被阻止以等待数据。在这种情况下，当有数据可用时，将只取消阻止一个任务。取消阻止的始终是正在等待数据的最高优先级任务。如果被阻止的多个任务具有相同的优先级，则取消阻止等待数据时间最长的任务。

# 队列写入时阻止

和从队列中读取数据时一样，任务可以选择指定写入队列时的阻止时间。在这种情况下，阻止时间是当队列已满时任务应该保持“被阻止”状态以等待队列空间变为可用的最长时间。

队列可以有多个写入方，因此一个已满的队列可能使多个任务被阻止以等待完成发送操作。在这种情况下，当队列中有空间可用时，将只取消阻止一个任务。取消阻止的始终是正在等待空间的最高优先级任务。如果被阻止的多个任务具有相同的优先级，则取消阻止等待空间的时间最长的任务。

# 在多个队列中阻止

多个队列可以分在一组形成队列集，这样进入“被阻止”状态的任务可以等待队列集里任何队列有数据可用。有关队列集的更多信息，请参阅[从多个队列接收 \(p. 78\)](#)。

# 使用队列

## xQueueCreate() API 函数

队列必须先显式创建，然后才能使用。

队列通过句柄（类型为 QueueHandle\_t 的变量）引用。xQueueCreate() API 函数创建一个队列，并返回用于引用所创建队列的 QueueHandle\_t。

FreeRTOS V9.0.0 还包含 xQueueCreateStatic() 函数，该函数分配在编译时静态创建队列所需的内存。当有队列创建时，FreeRTOS 从 FreeRTOS 堆中分配 RAM。RAM 用于保存队列数据结构和队列中包含的项目。如果可用堆 RAM 不足以用于创建队列，xQueueCreate() 将返回 NULL。

下面是 xQueueCreate() API 函数原型。

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

下表列出了 xQueueCreate() 参数和返回值。

参数名称	描述
uxQueueLength	要创建的队列在同一时间最多可以包含的项目的数量。
uxItemSize	队列中可以存储的每个数据项目的大小（以字节为单位）。
返回值	如果返回 NULL，则表示无法创建队列，因为没有足够的堆内存可供 FreeRTOS 分配队列数据结构和存储区域。 如果返回非 NULL 值，表示队列已成功创建。返回的值应存储为所创建队列的句柄。

在创建队列之后，可使用 xQueueReset() API 函数使队列返回到其初始空状态。

## xQueueSendToBack() 和 xQueueSendToFront() API 函数

xQueueSendToBack() 用于将数据发送到队列后端（尾部）。xQueueSendToFront() 用于将数据发送到队列前端（头部）。

xQueueSend() 等效于 xQueueSendToBack()，实际上两者完全相同。

注意：请勿从中断服务例程中调用 xQueueSendToFront() 或 xQueueSendToBack()。而应使用中断安全版本 xQueueSendToFrontFromISR() 和 xQueueSendToBackFromISR()。

下面是 xQueueSendToFront() API 函数原型。

```
BaseType_t xQueueSendToFront( QueueHandle_t xQueue,
                           const void * pvItemToQueue,
                           TickType_t xTicksToWait );
```

下面是 xQueueSendToBack() API 函数原型。

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue,
                           const void * pvItemToQueue,
                           TickType_t xTicksToWait );
```

下表列出了 xQueueSendToFront() 和 xQueueSendToBack() 的函数参数和返回值。

参数名称/返回的值	描述
xQueue	将向其发送（写入）数据的队列的句柄。该队列句柄从 xQueueCreate() 调用返回，用于创建队列。
pvItemToQueue	指针，它指向要复制到队列的数据。  队列创建时就设置了队列可以保存的每个项目的大 小，因此相应数量的字节将从 pvItemToQueue 复制 到队列存储区域。
xTicksToWait	当队列已满时任务应该保持为“被阻止”状态以等待队 列空间变为可用的最长时间。  如果 xTicksToWait 为零且队列 已满，xQueueSendToFront() 和 xQueueSendToBack() 都将立即返回。  阻止时间以计时周期指定，因此它表示的绝对时间 取决于计时频率。宏 pdMS_TO_TICKS() 可用于将 以毫秒为单位指定的时间转换为以计时周期指定的 时间。  如果 FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 设置为 1，将 xTicksToWait 设置为 portMAX_DELAY 将导致任务 无限期等待（无超时）。
返回的值	可能的返回值有两个：  1. pdPASS  仅当数据已成功发送到队列时返回此值。  如果指定了阻止时间（xTicksToWait 不为零）， 则可能是在函数返回之前，发出调用的任务置 于“被阻止”状态以等待队列中的空间变为可用，但 是在阻止时间到期之前数据已成功写入队列。  2. errQUEUE_FULL  如果因为队列已满而导致数据无法写入队列，则 返回此值。

如果指定了阻止时间 ( xTicksToWait 不为零 ) , 则发出调用的任务会置于“被阻止”状态以等待另一个任务或中断在队列中腾出空间 , 但是在空间可用之前指定的阻止时间就已到期。

## xQueueReceive() API 函数

xQueueReceive() 用于接收 ( 读取 ) 队列中的项目。接收到的项目会从队列中移除。

注意 : 请勿从中断服务例程中调用 xQueueReceive() 。而应使用中断安全 xQueueReceiveFromISR() API 函数。

下面是 xQueueReceive() API 函数原型。

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,
                          void * const pvBuffer,
                          TickType_t xTicksToWait );
```

下表列出了 xQueueReceive() 函数的参数和返回值。

参数名称/返回的值	描述
xQueue	从中接收 ( 读取 ) 数据的队列的句柄。该队列句柄从 xQueueCreate() 调用返回 , 用于创建队列。
pvBuffer	内存指针 , 接收到的数据将复制到该内存位置。 队列保存的每个数据项目的大小都在创建队列时设置。pvBuffer 指向的内存必须至少足以容纳相应字节数的项目。
xTicksToWait	当队列已空时任务应该保持“被阻止”状态以等待队列中有数据可用的最长时间。 如果 xTicksToWait 为零 , 并且队列已空 , 则 xQueueReceive() 将立即返回。 阻止时间以计时周期指定 , 因此它表示的绝对时间取决于计时频率。宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为计时周期数。 如果 FreeRTOSConfig.h 中的 INCLUDE_TaskSuspend 设置为 1 , 将 xTicksToWait 设置为 portMAX_DELAY 将导致任务无限期等待 ( 无超时 ) 。
返回的值	可能的返回值有两个 : 1. pdPASS 仅当数据已成功从队列中读取时返回此值。 如果指定了阻止时间 ( xTicksToWait 不为零 ) , 则可能是发出调用的任务置于“被阻止”状态以等待

队列中有数据可用，但是在阻止时间到期之前已成功从队列中读取数据。

2. errQUEUE\_EMPTY

如果因为队列已空而导致无法从队列中读取数据，则返回此值。

如果指定了阻止时间（xTicksToWait 不为零），则发出调用的任务会置于“被阻止”状态以等待另一个任务或中断向队列发送数据，但是在数据发送之前阻止时间就已到期。

## uxQueueMessagesWaiting() API 函数

uxQueueMessagesWaiting() 用于查询当前队列中的项目的数量。

注意：请勿从中断服务例程中调用 uxQueueMessagesWaiting()。而应使用中断安全 uxQueueMessagesWaitingFromISR()。

下面是 uxQueueMessagesWaiting() API 函数原型。

```
UBaseType_t uxQueueMessagesWaiting( QueueHandle_t xQueue );
```

下表列出了 uxQueueMessagesWaiting() 函数的参数和返回值。

参数名称/返回的值	描述
xQueue	要查询的队列的句柄。该队列句柄从 xQueueCreate() 调用返回，用于创建队列。
返回的值	查询的队列当前包含的项目的数量。如果返回零，说明队列为空。

## 从队列接收时阻止（示例 10）

此示例介绍如何创建队列，如何从多个任务向队列发送数据以及如何接收队列中的数据。创建队列以保存 int32\_t 类型的数据项目。向队列发送数据的任务不指定阻止时间，但是从队列接收数据的任务要指定。

向队列发送数据的任务的优先级低于从队列中接收数据的任务的优先级。这意味着队列不会包含多个项目，因为只要将数据发送到队列，接收任务就会取消阻止，优先于发送任务执行，并移除数据，让队列再次为空。

下面的代码说明写入队列的任务的实现。创建此任务的两个实例：一个实例连续将值 100 写入队列，另一个任务连续将值 200 写入到同一个队列。任务参数用于将这些值传递至每个任务实例。

```
static void vSenderTask( void *pvParameters )
{
    int32_t lValueToSend;
    BaseType_t xStatus;
```

```

/* Two instances of this task are created so the value that is sent to the queue
is passed in through the task parameter. This way, each instance can use a different
value. The queue was created to hold values of type int32_t, so cast the parameter to the
required type. */

lValueToSend = ( int32_t ) pvParameters;

/* As per most tasks, this task is implemented within an infinite loop. */

for( ;; )

{

    /* Send the value to the queue. The first parameter is the queue to which data is
being sent. The queue was created before the scheduler was started, so before this task
started to execute. The second parameter is the address of the data to be sent, in this
case the address of lValueToSend. The third parameter is the Block time, the time the task
should be kept in the Blocked state to wait for space to become available on the queue
should the queue already be full. In this case a block time is not specified because the
queue should never contain more than one item, and therefore never be full. */

    xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );

    if( xStatus != pdPASS )

    {

        /* The send operation could not complete because the queue was full. This must
be an error because the queue should never contain more than one item! */

        vPrintString( "Could not send to the queue.\r\n" );

    }

}

}

```

下面的代码说明从队列接收数据的任务的实现。接收任务指定 100 毫秒的阻止时间，因此它将进入“被阻止”状态以等待有数据可用。当队列中有数据可用，或 100 毫秒后数据仍不可用，它将离开“被阻止”状态。在本示例中，100 毫秒超时不会到期，因为有两个任务不断写入队列。

```

static void vReceiverTask( void *pvParameters )

{

    /* Declare the variable that will hold the values received from the queue. */

    int32_t lReceivedValue;

    BaseType_t xStatus;

    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* This task is also defined within an infinite loop. */

    for( ;; )

    {

        /* This call should always find the queue empty because this task will immediately
remove any data that is written to the queue. */

        if( uxQueueMessagesWaiting( xQueue ) != 0 )

```

```
{  
  
    vPrintString( "Queue should have been empty!\r\n" );  
  
}  
  
/* Receive data from the queue. The first parameter is the queue from which data is  
to be received. The queue is created before the scheduler is started, and therefore before  
this task runs for the first time. The second parameter is the buffer into which the  
received data will be placed. In this case the buffer is simply the address of a variable  
that has the required size to hold the received data. The last parameter is the block  
time, the maximum amount of time that the task will remain in the Blocked state to wait  
for data to be available should the queue already be empty. */  
  
xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );  
  
if( xStatus == pdPASS )  
  
{  
  
    /* Data was successfully received from the queue, print out the received value.  
    */  
  
    vPrintStringAndNumber( "Received = ", lReceivedValue );  
  
}  
  
else  
  
{  
  
    /* Data was not received from the queue even after waiting for 100ms.This must  
be an error because the sending tasks are free running and will be continuously writing to  
the queue. */  
  
    vPrintString( "Could not receive from the queue.\r\n" );  
  
}  
  
}  
}
```

下面的代码包含 main() 函数的定义。它只是在启动计划程序之前创建队列和三个任务。创建的队列用于保存最多 5 个 int32\_t 值，即使任务优先级的设置使得队列无法同时包含超过一个项目也是如此。

```
/* Declare a variable of type QueueHandle_t. This is used to store the handle to the queue  
that is accessed by all three tasks. */  
  
QueueHandle_t xQueue;  
  
int main( void )  
  
{  
  
    /* The queue is created to hold a maximum of 5 values, each of which is large enough to  
hold a variable of type int32_t. */  
  
    xQueue = xQueueCreate( 5, sizeof( int32_t ) );  
  
    if( xQueue != NULL )  
  
    {
```

```

/* Create two instances of the task that will send to the queue. The task parameter
is used to pass the value that the task will write to the queue, so one task will
continuously write 100 to the queue while the other task will continuously write 200 to
the queue. Both tasks are created at priority 1. */

xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );

xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

/* Create the task that will read from the queue. The task is created with priority
2, so above the priority of the sender tasks. */

xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

/* Start the scheduler so the created tasks start executing. */

vTaskStartScheduler();

}

else

{

    /* The queue could not be created. */

}

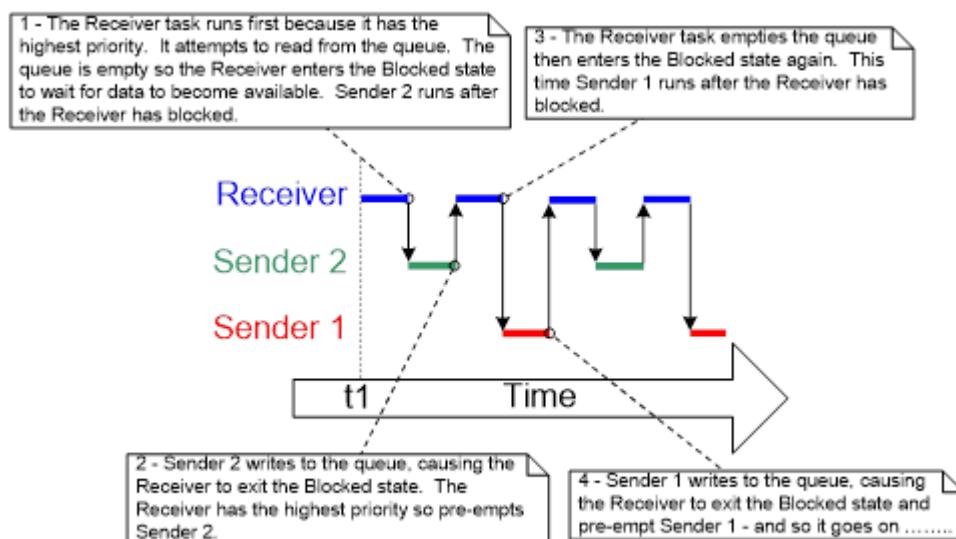
/* If all is well then main() will never reach here as the scheduler will now be
running the tasks. If main() does reach here then it is likely that there was insufficient
FreeRTOS heap memory available for the idle task to be created. For more information, see
Heap Memory Management. */

for( ; );
}

```

两个向队列发送数据的任务具有相同的优先级。这会导致两个发送任务轮流将数据发送到队列。

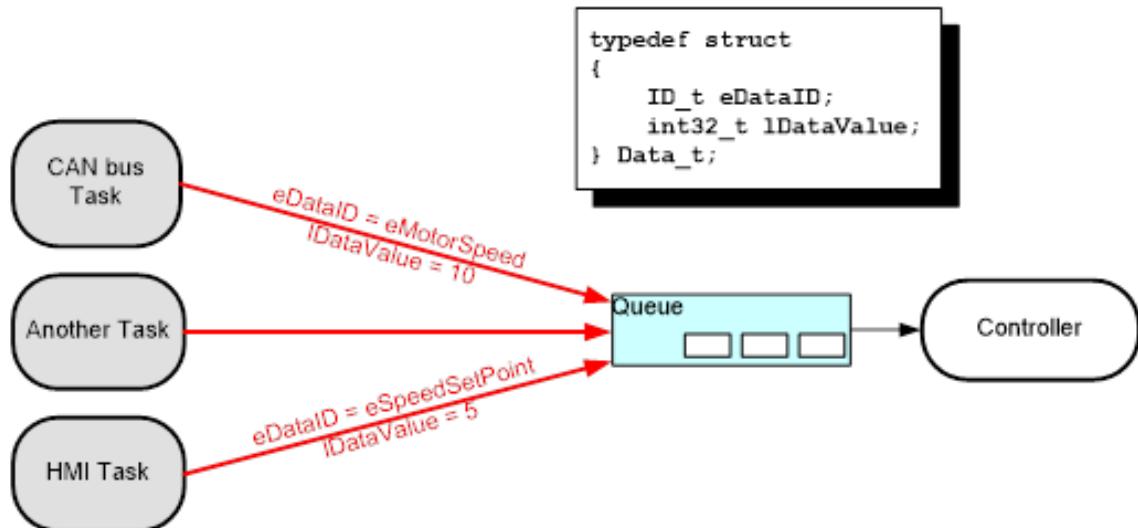
下图是执行顺序。



## 接收多个源的数据

一个任务从多个源接收数据的 FreeRTOS 设计是很常见的。接收任务必须知道数据源于何处，以确定应如何处理这些数据。一种简单的设计解决方案是使用单个队列来传输其字段同时包含数据值和数据源的结构。

下图是通过队列发送结构的示例情况。



- 创建一个队列，用于保存类型为 Data\_t 的结构。结构成员允许在一条消息中将数据值和枚举类型发送到队列。枚举类型用于指示数据的含义。
- 使用一个中心 Controller 任务来执行主要系统函数。它必须对通过队列传递给它的输入和系统状态更改进行响应。
- CAN 总线任务用来封装 CAN 总线接口功能。当 CAN 总线任务接收并解码一条消息后，它以 Data\_t 结构将已解码的消息发送给 Controller 任务。传输的结构中，eDataID 成员用于向 Controller 任务告知数据是什么（在本例中为电机转速值）。传输的结构中，lDataValue 成员用于向 Controller 任务告知电机转速值。
- 人机接口（HMI）任务用来封装所有 HMI 功能。设备操作员可能通过在 HMI 任务中可检测和解释的多种方式来输入命令和查询值。当输入新命令时，HMI 任务以 Data\_t 结构将命令发送到 Controller 任务。传输的结构中，eDataID 成员用于向 Controller 任务告知数据是什么（在本例中为新设定点值）。传输的结构中，lDataValue 成员用于向 Controller 任务告知设定点值。

## 向队列发送数据和通过队列发送结构时阻止（示例 11）

此示例与前例类似，但任务优先级相反，也就是说，接收任务的优先级低于发送任务的优先级。队列也用于传递结构而不是整数。

下面的代码说明要通过队列传递的结构的定义以及两个变量的声明。

```
/* Define an enumerated type used to identify the source of the data.*/
typedef enum
{
```

```
    eSender1,
    eSender2
} DataSource_t;

/* Define the structure type that will be passed on the queue. */

typedef struct
{
    uint8_t ucValue;
    DataSource_t eDataSource;
} Data_t;

/* Declare two variables of type Data_t that will be passed on the queue. */

static const Data_t xStructsToSend[ 2 ] =
{
    { 100, eSender1 }, /* Used by Sender1. */
    { 200, eSender2 } /* Used by Sender2. */
};
```

在前例中，接收任务的优先级最高，因此队列不会包含多个项目。这是因为，只要有数据放入队列，接收任务就优先于发送任务执行。在下面的示例中，发送任务的优先级更高，因此队列通常是满的。这是因为，只要接收任务从队列中移除项目，其中一个发送任务就优先于接收任务运行，立即重新填满队列。然后该发送任务重新进入“被阻止”状态，等待队列中再次有空间可用。

下面的代码说明发送任务的实现。发送任务指定 100 毫秒的阻止时间，因此每当队列变满时，它都将进入“被阻止”状态等待有空间可用。当队列中有可用空间，或 100 毫秒后仍没有空间可用时，它将离开“被阻止”状态。在本示例中，100 毫秒超时不会到期，因为接收任务不断从队列中移除项目，从而腾出空间。

```
static void vSenderTask( void *pvParameters )
{
    BaseType_t xStatus;

    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );
    /* As per most tasks, this task is implemented within an infinite loop.*/
    for( ;; )
    {
        /* Send to the queue. The second parameter is the address of the structure being
        sent. The address is passed in as the task parameter so pvParameters is used directly. The
        third parameter is the Block time, the time the task should be kept in the Blocked state
        to wait for space to become available on the queue if the queue is already full. A block
        time is specified because the sending tasks have a higher priority than the receiving task
        so the queue is expected to become full. The receiving task will remove items from the
        queue when both sending tasks are in the Blocked state. */
        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );
    }
}
```

```
    if( xStatus != pdPASS )  
  
    {  
  
        /* The send operation could not complete, even after waiting for 100 ms. This  
        must be an error because the receiving task should make space in the queue as soon as both  
        sending tasks are in the Blocked state. */  
  
        vPrintString( "Could not send to the queue.\r\n" );  
  
    }  
  
}  
}
```

接收任务优先级最低，因此它仅在两个发送任务均处于“被阻止”状态时运行。仅当队列已满时，发送任务才会进入“被阻止”状态，因此接收任务仅在队列已满时执行。因此，即使它不指定阻止时间，也始终会接收到数据。

下面的代码说明接收任务的实现。

```
static void vReceiverTask( void *pvParameters )  
  
{  
  
    /* Declare the structure that will hold the values received from the queue. */  
  
    Data_t xReceivedStructure;  
  
    BaseType_t xStatus;  
  
    /* This task is also defined within an infinite loop. */  
  
    for( ;; )  
  
    {  
  
        /* Because it has the lowest priority, this task will only run when the sending  
        tasks are in the Blocked state. The sending tasks will only enter the Blocked state when  
        the queue is full so this task always expects the number of items in the queue to be equal  
        to the queue length, which is 3 in this case. */  
  
        if( uxQueueMessagesWaiting( xQueue ) != 3 )  
  
        {  
  
            vPrintString( "Queue should have been full!\r\n" );  
  
        }  
  
        /* Receive from the queue. The second parameter is the buffer into which the  
        received data will be placed. In this case, the buffer is simply the address of a variable  
        that has the required size to hold the received structure. The last parameter is the block  
        time, the maximum amount of time that the task will remain in the Blocked state to wait  
        for data to be available if the queue is already empty. In this case, a block time is not  
        required because this task will only run when the queue is full. */  
  
        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );  
  
        if( xStatus == pdPASS )  
  
        {
```

```
/* Data was successfully received from the queue, print out the received value
and the source of the value. */

if( xReceivedStructure.eDataSource == eSender1 )

{

    vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );

}

else

{

    vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );

}

else

{

    /* Nothing was received from the queue. This must be an error because this task
should only run when the queue is full. */

    vPrintString( "Could not receive from the queue.\r\n" );

}

}

}

}
```

与前例相比，main() 仅略有更改。创建的队列用于保存三个 Data\_t 结构，发送和接收任务的优先级与之前相反。下面是 main() 的实现。

```
int main( void )

{

    /* The queue is created to hold a maximum of 3 structures of type Data_t. */

    xQueue = xQueueCreate( 3, sizeof( Data_t ) );

    if( xQueue != NULL )

    {

        /* Create two instances of the task that will write to the queue. The parameter
        is used to pass the structure that the task will write to the queue, so one task will
        continuously send xStructsToSend[ 0 ] to the queue while the other task will continuously
        send xStructsToSend[ 1 ]. Both tasks are created at priority 2, which is above the
        priority of the receiver. */

        xTaskCreate( vSenderTask, "Sender1", 1000, &( xStructsToSend[ 0 ] ), 2, NULL );

        xTaskCreate( vSenderTask, "Sender2", 1000, &( xStructsToSend[ 1 ] ), 2, NULL );

        /* Create the task that will read from the queue. The task is created with priority
        1, so below the priority of the sender tasks. */

    }

}
```

```

xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

/* Start the scheduler so the created tasks start executing. */

vTaskStartScheduler();

}

else

{

    /* The queue could not be created. */

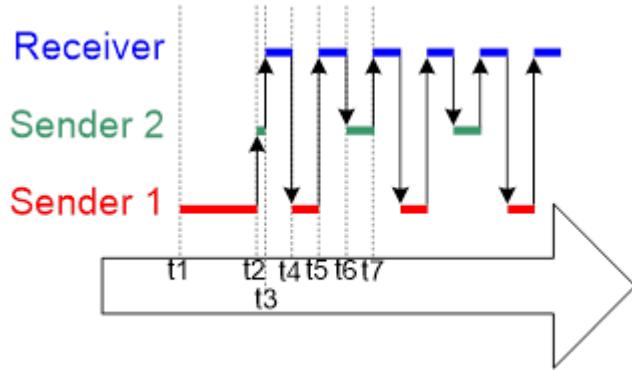
}

/* If all is well then main() will never reach here as the scheduler will now be
running the tasks. If main() does reach here then it is likely that there was insufficient
heap memory available for the idle task to be created. Chapter 2 provides more information
on heap memory management. */

for( ;; );
}

```

下图显示了发送任务优先级高于接收任务优先级时的执行顺序。



下表说明为什么前四条消息来自同一个任务。

时间	描述
t1	任务 Sender 1 执行，将三个数据项目发送到队列。
t2	队列满，因此 Sender 1 进入“被阻止”状态，等待完成其下一次发送。任务 Sender 2 现在是可以运行的优先级最高的任务，因此进入“运行”状态。
t3	任务 Sender 2 发现队列已满，因此进入“被阻止”状态，等待完成其首次发送。任务 Receiver 现在是可以运行的优先级最高的任务，因此进入“运行”状态。
t4	优先级高于接收任务优先级的两个任务正在等待队列中的空间变为可用，这样，只要任务 Receiver 从队列中移除一个项目，这两个任务就优先于任务 Receiver 运行。任务 Sender 1 和 Sender 2 具有相同的优先级，因此计划程序选择等待了最长时间

	的任务（在本例中为任务 Sender 1）使之进入“运行”状态。
t5	<p>任务 Sender 1 向队列发送另一个数据项目。队列中只有一个空间，因此任务 Sender 1 进入“被阻止”状态，等待完成其下一次发送。任务 Receiver 又成为可以运行的优先级最高的任务，因此进入“运行”状态。</p> <p>任务 Sender 1 现在已经向队列发送了 4 个项目，任务 Sender 2 仍在等待向队列发送其第一个项目。</p>
t6	优先级高于接收任务优先级的两个任务正在等待队列中的空间变为可用，这样，只要任务 Receiver 从队列中移除一个项目，这两个任务就优先于任务 Receiver 运行。这次，Sender 2 等待的时间比 Sender 1 长，因此 Sender 2 进入“运行”状态。
t7	任务 Sender 2 向队列发送一个数据项目。队列中只有一个空间，因此 Sender 2 进入“被阻止”状态，等待完成其下一次发送。任务 Sender 1 和 Sender 2 都在等待队列空间变为可用，因此任务 Receiver 是唯一可以进入“运行”状态的任务。

## 处理大型或大小可变的数据

### 指针排队

如果队列中存储的数据较大，最好使用队列来传输指向数据的指针，而不是逐个字节地将数据复制到队列中和从队列中移出。无论从处理时间，还是从创建队列所需的 RAM 量来看，传输指针都更为高效。但是，要进行指针排队，请确保：

- 所指向的 RAM 的所有者是明确定义的。

如果使用指向任务共享内存的指针，必须确保两个任务不会同时修改内存内容或执行任何其他可能导致内存内容无效或不一致的操作。理想情况下，在内存指针进入队列之前，只应该允许发送任务访问内存，从队列中接收到指针后，只应该允许接收任务访问内存。

- 指针指向的 RAM 保持有效。

如果指针指向的内存是动态分配的，或是从预分配缓冲区池获得的，则应该只由一个任务负责释放内存。所有任务都不应尝试访问已释放的内存。

不可使用指针来访问分配在任务堆栈上的数据。如果堆栈框架更改，数据将无效。

下面的代码示例说明如何使用队列将指向缓冲区的指针从一个任务发送到另一个任务。

下面的代码创建一个最多可容纳五个指针的队列。

```
/* Declare a variable of type QueueHandle_t to hold the handle of the queue being created.
 */
QueueHandle_t xPointerQueue;
```

```
/* Create a queue that can hold a maximum of 5 pointers (in this case, character pointers). */
*xPointerQueue = xQueueCreate( 5, sizeof( char * ) );
```

下面的代码分配一个缓冲区，向该缓冲区写入一个字符串，然后将指向该缓冲区的指针发送到队列。

```
/* A task that obtains a buffer, writes a string to the buffer, and then sends the address
   of the buffer to the queue created in the previous listing. */

void vStringSendingTask( void *pvParameters )

{
    char *pcStringToSend;

    const size_t xMaxStringLength = 50;

    BaseType_t xStringNumber = 0;

    for( ;; )

    {
        /* Obtain a buffer that is at least xMaxStringLength characters big. The
           implementation of prvGetBuffer() is not shown. It might obtain the buffer from a pool of
           preallocated buffers or just allocate the buffer dynamically. */

        pcStringToSend = ( char * ) prvGetBuffer( xMaxStringLength );

        /* Write a string into the buffer. */

        sprintf( pcStringToSend, xMaxStringLength, "String number %d\r\n",
xStringNumber );
        /* Increment the counter so the string is different on each iteration of this task.
       */

        xStringNumber++;

        /* Send the address of the buffer to the queue that was created in the previous
           listing. The address of the buffer is stored in the pcStringToSend variable.*/

        xQueueSend( xPointerQueue, /* The handle of the queue. */ &pcStringToSend, /* The
           address of the pointer that points to the buffer. */ portMAX_DELAY );
    }
}
```

下面的代码从队列接收指向缓冲区的指针，然后输出该缓冲区包含的字符串。

```
/* A task that receives the address of a buffer from the queue created in the first listing
   and written to in the second listing. The buffer contains a string, which is printed out.
   */

void vStringReceivingTask( void *pvParameters )

{
    char *pcReceivedString;

    for( ;; )
```

```
{  
    /* Receive the address of a buffer. */  
  
    xQueueReceive( xPointerQueue, /* The handle of the queue. */ &pcReceivedString, /*  
    Store the buffer's address in pcReceivedString. */ portMAX_DELAY );  
  
    /* The buffer holds a string. Print it out. */  
  
    vPrintString( pcReceivedString );  
  
    /* The buffer is not required anymore. Release it so it can be freed or reused. */  
  
    prvReleaseBuffer( pcReceivedString );  
  
}  
}
```

## 使用队列来发送不同类型和长度的数据

向队列发送结构和向队列发送指针是两种强大的设计模式。结合使用这两种方法，任务可以使用单个队列从任何数据源接收任何数据类型。FreeRTOS+TCP TCP/IP 堆栈的实现就是一个实际的示例。

TCP/IP 堆栈在自己的任务中运行，它必须处理来自许多不同源的事件。不同事件类型与不同的数据类型和数据长度关联。在 TCP/IP 任务外发生的所有事件都通过类型为 IPStackEvent\_t 的结构来描述，并通过队列发送到 TCP/IP 任务。IPStackEvent\_t 结构的 pvData 成员是一个指针，可用来直接保存一个值或指向一个缓冲区。

下面是 FreeRTOS+TCP 中用于将事件发送到 TCP/IP 堆栈任务的 IPStackEvent\_t 结构。

```
/* A subset of the enumerated types used in the TCP/IP stack to identify events. */  
  
typedef enum  
{  
  
    eNetworkDownEvent = 0, /* The network interface has been lost or needs (re)connecting.  
*/  
  
    eNetworkRxEvent, /* A packet has been received from the network. */  
  
    eTCPAcceptEvent, /* FreeRTOS_accept() called to accept or wait for a new client. */  
  
    /* Other event types appear here but are not shown in this listing. */  
  
} eIPEvent_t;  
  
/* The structure that describes events and is sent on a queue to the TCP/IP task. */  
  
typedef struct IP_TASK_COMMANDS  
{  
  
    /* An enumerated type that identifies the event. See the eIPEvent_t definition. */  
  
    eIPEvent_t eEventType;  
  
    /* A generic pointer that can hold a value or point to a buffer. */  
  
    void *pvData;
```

```
} IPStackEvent_t;
```

示例 TCP/IP 事件及其关联数据包括：

- eNetworkRxEvent：已从网络接收到数据包。

从网络接收到的数据将通过类型为 IPStackEvent\_t 的结构发送到 TCP/IP 任务。该结构的 eEventType 成员设置为 eNetworkRxEvent。该结构的 pvData 成员用于指向包含所接收数据的缓冲区。

这段伪代码说明如何使用 IPStackEvent\_t 结构将接收到的网络数据发送到 TCP/IP 任务。

```
void vSendRxDataToTheTCPTask( NetworkBufferDescriptor_t *pxRxedData )  
{  
    IPStackEvent_t xEventStruct;  
  
    /* Complete the IPStackEvent_t structure. The received data is stored in pxRxedData. */  
  
    xEventStruct.eEventType = eNetworkRxEvent;  
    xEventStruct.pvData = ( void * ) pxRxedData;  
  
    /* Send the IPStackEvent_t structure to the TCP/IP task. */  
    xSendEventStructToIPTask( &xEventStruct );  
}
```

- eTCPAcceptEvent：一个套接字将接受或等待客户端连接。

Accept 事件通过 IPStackEvent\_t 类型的结构从调用 FreeRTOS\_accept() 的任务发送到 TCP/IP 任务。该结构的 eEventType 成员设置为 eTCPAcceptEvent。该结构的 pvData 成员设置为接受连接的套接字的句柄。

这段伪代码说明如何使用 IPStackEvent\_t 结构将接受连接的套接字的句柄发送到 TCP/IP 任务。

```
void vSendAcceptRequestToTheTCPTask( Socket_t xSocket )  
{  
    IPStackEvent_t xEventStruct;  
  
    /* Complete the IPStackEvent_t structure. */  
    xEventStruct.eEventType = eTCPAcceptEvent;  
    xEventStruct.pvData = ( void * ) xSocket;  
  
    /* Send the IPStackEvent_t structure to the TCP/IP task. */  
    xSendEventStructToIPTask( &xEventStruct );  
}
```

- eNetworkDownEvent：网络需要连接或重新连接。

网络断开事件通过类型为 IPStackEvent\_t 的结构从网络接口发送到 TCP/IP 任务。该结构的 eEventType 成员设置为 eNetworkDownEvent。网络断开事件与任何数据都不关联，因此不使用该结构的 pvData 成员。

这段伪代码说明如何使用 IPStackEvent\_t 结构将网络断开事件发送到 TCP/IP 任务。

```
void vSendNetworkDownEventToTheTCPTask( Socket_t xSocket )  
{  
    IPStackEvent_t xEventStruct;  
  
    /* Complete the IPStackEvent_t structure. */  
    xEventStruct.eEventType = eNetworkDownEvent;  
  
    xEventStruct.pvData = NULL; /* Not used, but set to NULL for completeness. */  
  
    /* Send the IPStackEvent_t structure to the TCP/IP task. */  
  
    xSendEventStructToIPTask( &xEventStruct );  
}
```

下面是在 TCP/IP 任务中接收并处理这些事件的代码。从队列接收到的 IPStackEvent\_t 结构的 eEventType 成员用于确定如何解释 pvData 成员。这段伪代码说明如何使用 IPStackEvent\_t 结构将网络断开事件发送到 TCP/IP 任务。

```
IPStackEvent_t xReceivedEvent;  
  
/* Block on the network event queue until either an event is received, or xNextIPSleep  
ticks pass without an event being received. eEventType is set to eNoEvent in case the  
call to xQueueReceive() returns because it timed out, rather than because an event was  
received. */  
  
xReceivedEvent.eEventType = eNoEvent;  
  
xQueueReceive( xNetworkEventQueue, &xReceivedEvent, xNextIPSleep );  
  
/* Which event was received, if any? */  
  
switch( xReceivedEvent.eEventType )  
{  
  
    case eNetworkDownEvent :  
  
        /* Attempt to (re)establish a connection. This event is not associated with any  
data. */  
  
        prvProcessNetworkDownEvent();  
  
        break;  
  
    case eNetworkRxEvent :  
  
        /* The network interface has received a new packet. A pointer to the received data  
is stored in the pvData member of the received IPStackEvent_t structure. Process the  
received data. */  
  
        prvHandleEthernetPacket( ( NetworkBufferDescriptor_t * )  
( xReceivedEvent.pvData ) );  
  
        break;  
}
```

```
case eTCPAcceptEvent:  
  
    /* The FreeRTOS_accept() API function was called. The handle of the socket that  
    is accepting a connection is stored in the pvData member of the received IPStackEvent_t  
    structure. */  
  
    pxSocket = ( FreeRTOS_Socket_t * ) ( xReceivedEvent.pvData );  
  
    xTCPCheckNewClient( pxSocket );  
  
    break;  
  
    /* Other event types are processed in the same way, but are not shown here. */  
}
```

## 从多个队列接收

# 队列集

设计应用程序时，通常只需要单个任务就可接收不同大小的数据、不同含义的数据，以及不同源的数据。上一节介绍了如何通过一个接收结构的队列简洁高效地实现这一目的。但是，有时设计工作可能存在选择限制，需要为某些数据源使用单独的队列。例如，设计中要集成的第三方代码可能存在专用队列。这种情况下，可以使用队列集。

通过队列集，一个任务可以接收多个队列的数据，无需轮询每个队列以确定哪个队列（如果有）包含数据。

与使用单个队列接收结构实现相同功能的设计相比，使用队列集接收多个源的数据的设计不够简洁和高效。因此，建议仅当设计限制使得有必要的时候才使用队列集。

下面几节介绍具体方法：

1. 创建队列集。
2. 将队列添加到该集。

您还可以向队列集添加信号灯。[中断管理 \(p. 108\)](#)中介绍了信号灯。

3. 读取队列集，以确定其中哪些队列包含数据。

当作为队列集成员的队列接收到数据时，该接收队列的句柄将发送到队列集，并在有任务调用从队列集读取的函数时返回。因此，如果从队列集返回队列句柄，说明该句柄所引用的队列包含数据，任务可以直接从该队列读取。

注意：如果某个队列是队列集的成员，从队列集读取该队列的句柄之前，不要从该队列中读取数据。

要启用队列集功能，请在 FreeRTOSConfig.h 中将 configUSE\_QUEUE\_SETS 编译时间配置常量设置为 1。

## xQueueCreateSet() API 函数

队列集必须先显式创建，然后才能使用。

队列集通过句柄（类型为 QueueSetHandle\_t 的变量）引用。xQueueCreateSet() API 函数创建一个队列集，并返回引用所创建队列集的 QueueSetHandle\_t。

下面是 xQueueCreateSet() API 函数原型。

```
QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength);
```

下表列出了 xQueueCreateSet() 参数和返回值。

参数名称	描述
uxEventQueueLength	<p>当作为队列集成员的某个队列接收到数据时，该接收队列的句柄将发送到队列集。uxEventQueueLength 定义所创建的队列集在任何同一时间最多能够包含的队列句柄的数量。</p> <p>队列句柄仅在队列集里的队列接收到数据时才发送到该集。队列已满时将无法接收数据，因此，如果队列集里所有队列都已满，则无法向队列集发送队列句柄。因此，队列集在同一时间最多可以包含的项目数是该队列集里每个队列的长度的总和。</p> <p>例如，如果队列集里有三个空队列，每个队列的长度为 5，则队列集里所有队列都满时，队列集里的队列总共可以接收 15 个项目（3 个队列，每个队列 5 个项目，两数相乘）。在该示例中，uxEventQueueLength 必须设置为 15，才能保证队列集可以接收发送给它的每一个项目。</p> <p>也可以向队列集添加信号灯。本指南稍后介绍二进制信号灯和计数信号灯。为了计算 uxEventQueueLength，二进制信号灯的长度为 1，计数信号灯的长度由信号灯的最大计数值确定。</p> <p>又例如，如果队列集将包含一个长度为 3 的队列和一个长度为 1 的二进制信号灯，则 uxEventQueueLength 必须设置为 4（3 加 1）。</p>
返回值	<p>如果返回 NULL，则表示无法创建队列集，因为没有足够的堆内存可供 FreeRTOS 分配队列集数据结构和存储区域。</p> <p>如果返回非 NULL 值，表示队列集已成功创建。返回的值应存储为所创建队列集的句柄。</p>

## xQueueAddToSet() API 函数

xQueueAddToSet() 将队列或信号灯添加到队列集。有关信号灯的信息，请参阅 [中断管理 \(p. 108\)](#)。

下面是 xQueueAddToSet() API 函数原型。

```
BaseType_t xQueueAddToSet( QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet );
```

下表列出了 xQueueAddToSet() 参数和返回值。

## xQueueSelectFromSet() API 函数

xQueueSelectFromSet() 从队列集读取队列句柄。

当作为集成员的某个队列或信号灯接收到数据时，该接收队列或信号灯的句柄将发送到队列集，并在有任务调用 `xQueueSelectFromSet()` 时返回。如果对 `xQueueSelectFromSet()` 的调用返回了句柄，说明该句柄引用的队列或信号灯包含数据，发出调用的任务必须直接从该队列或信号灯读取。

注意：除非首先通过调用 `xQueueSelectFromSet()` 返回队列或信号灯的句柄，否则不要从作为集成员的队列或信号灯读取数据。每次调用 `xQueueSelectFromSet()` 返回队列句柄或信号灯句柄时，仅从该队列或信号灯读取一个项目。

下面是 `xQueueSelectFromSet()` API 函数原型。

```
QueueSetMemberHandle_t xQueueSelectFromSet( QueueSetHandle_t xQueueSet, const TickType_t xTicksToWait );
```

下表列出了 `xQueueSelectFromSet()` 参数和返回值。

#### xQueueSet

所接收（读取）的队列句柄或信号灯句柄所属的队列集的句柄。队列集句柄是调用 `xQueueCreateSet()`（用于创建队列集）时返回的。

#### xTicksToWait

如果队列集里所有队列和信号灯都为空，发出调用的任务应保持为“被阻止”状态以等待从队列集接收队列或信号灯句柄的最大时间量。如果 `xTicksToWait` 为零，并且队列集里所有队列和信号灯都为空，则 `xQueueSelectFromSet()` 将立即返回。阻止时间以计时周期指定，因此它表示的绝对时间取决于计时频率。宏 `pdMS_TO_TICKS()` 可用于将以毫秒为单位指定的时间转换为计时周期数。如果 `FreeRTOSConfig.h` 中的 `INCLUDE_vTaskSuspend` 设置为 1，将 `xTicksToWait` 设置为 `portMAX_DELAY` 将导致任务无限期等待（无超时）。

#### 返回值

非 NULL 返回值是已知包含数据的队列或信号灯的句柄。如果指定了阻止时间（`xTicksToWait` 不为零），则可能是发出调用的任务置于“被阻止”状态以等待队列集里的队列或信号灯有数据可用，但是在阻止时间到期之前已成功从队列集读取句柄。句柄以 `QueueSetMemberHandle_t` 类型返回，它可以强制转换为 `QueueHandle_t` 类型或 `SemaphoreHandle_t` 类型。

如果返回值为 NULL，则无法从队列集读取句柄。如果指定了阻止时间（`xTicksToWait` 不为零），则可能是发出调用的任务置于“被阻止”状态以等待另一个任务或中断向队列集里的队列或信号灯发送数据，但是在数据发送之前阻止时间就已到期。

## 使用队列集（示例 12）

此示例创建两个发送任务和一个接收任务。发送任务通过两个独立的队列将数据发送到接收任务，每个任务使用一个队列。这两个队列添加到队列集，接收任务从队列集读取，以确定其中哪一个队列包含数据。

任务、队列和队列集都在 `main()` 中创建。

```
/* Declare two variables of type QueueHandle_t. Both queues are added to the same queue set. */

static QueueHandle_t xQueue1 = NULL, xQueue2 = NULL;

/* Declare a variable of type QueueSetHandle_t. This is the queue set to which the two queues are added. */

static QueueSetHandle_t xQueueSet = NULL;

int main( void )
```

```
{  
  
    /* Create the two queues, both of which send character pointers. The priority of the  
    receiving task is above the priority of the sending tasks, so the queues will never have  
    more than one item in them at any one time*/  
  
    xQueue1 = xQueueCreate( 1, sizeof( char * ) );  
  
    xQueue2 = xQueueCreate( 1, sizeof( char * ) );  
  
    /* Create the queue set. Two queues will be added to the set, each of which can contain  
    1 item, so the maximum number of queue handles the queue set will ever have to hold at one  
    time is 2 (2 queues multiplied by 1 item per queue). */  
  
    xQueueSet = xQueueCreateSet( 1 * 2 );  
  
    /* Add the two queues to the set. */  
  
    xQueueAddToSet( xQueue1, xQueueSet );  
  
    xQueueAddToSet( xQueue2, xQueueSet );  
  
    /* Create the tasks that send to the queues. */  
  
    xTaskCreate( vSenderTask1, "Sender1", 1000, NULL, 1, NULL );  
  
    xTaskCreate( vSenderTask2, "Sender2", 1000, NULL, 1, NULL );  
  
    /* Create the task that reads from the queue set to determine which of the two queues  
    contain data. */  
  
    xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );  
  
    /* Start the scheduler so the created tasks start executing. */  
  
    vTaskStartScheduler();  
  
    /* As normal, vTaskStartScheduler() should not return, so the following lines will  
    never execute. */  
  
    for( ;; );  
  
    return 0;  
}
```

第一个发送任务每 100 毫秒使用 xQueue1 将字符指针发送到接收任务。第二个发送任务每 200 毫秒使用 xQueue2 将字符指针发送到接收任务。字符指针设置为指向用于标识发送任务的字符串。下面是这两个发送任务的实现。

```
void vSenderTask1( void *pvParameters )  
  
{  
  
    const TickType_t xBlockTime = pdMS_TO_TICKS( 100 );  
  
    const char * const pcMessage = "Message from vSenderTask1\r\n";  
  
    /* As per most tasks, this task is implemented within an infinite loop. */  
  
    for( ;; )  
  
    {
```

```
/* Block for 100ms. */

vTaskDelay( xBlockTime );

/* Send this task's string to xQueue1. It is not necessary to use a block time,
even though the queue can only hold one item. This is because the priority of the task
that reads from the queue is higher than the priority of this task. As soon as this task
writes to the queue, it will be preempted by the task that reads from the queue, so the
queue will already be empty again by the time the call to xQueueSend() returns. The block
time is set to 0. */

xQueueSend( xQueue1, &pcMessage, 0 );

}

}

/*-----*/
void vSenderTask2( void *pvParameters )

{

const TickType_t xBlockTime = pdMS_TO_TICKS( 200 );

const char * const pcMessage = "Message from vSenderTask2\r\n";

/* As per most tasks, this task is implemented within an infinite loop. */

for( ;; )

{

/* Block for 200ms. */

vTaskDelay( xBlockTime );

/* Send this task's string to xQueue2. It is not necessary to use a block time,
even though the queue can only hold one item. This is because the priority of the task
that reads from the queue is higher than the priority of this task. As soon as this task
writes to the queue, it will be preempted by the task that reads from the queue, so the
queue will already be empty again by the time the call to xQueueSend() returns. The block
time is set to 0. */

xQueueSend( xQueue2, &pcMessage, 0 );

}

}
```

发送任务写入的队列是同一个队列集的成员。每次有任务发送到其中一个队列时，该队列的句柄都会发送到队列集。接收任务调用 `xQueueSelectFromSet()` 从队列集读取队列句柄。接收任务接收到队列集里的队列句柄后，知道所接收到的句柄引用的队列包含数据，因此直接从该队列读取数据。它从队列读取的数据是指向接收任务输出的字符串的指针。

如果 `xQueueSelectFromSet()` 调用超时，则返回 `NULL`。在上述代码中，使用无限期阻止时间调用 `xQueueSelectFromSet()`，因此不会超时，只能返回有效的队列句柄。因此，在使用返回值之前，接收任务不需要检查 `xQueueSelectFromSet()` 是否返回 `NULL`。

如果句柄所引用的队列包含数据，则 `xQueueSelectFromSet()` 只返回队列句柄，因此在从队列中读取时，不必使用阻止时间。

下面是接收任务的实现。

```

void vReceiverTask( void *pvParameters )
{
    QueueHandle_t xQueueThatContainsData;
    char *pcReceivedString;

    /* As per most tasks, this task is implemented within an infinite loop.*/
    for( ;; )

    {
        /* Block on the queue set to wait for one of the queues in the set to contain
        data. Cast the QueueSetMemberHandle_t value returned from xQueueSelectFromSet() to a
        QueueHandle_t because it is known all the members of the set are queues (the queue set
        does not contain any semaphores). */

        xQueueThatContainsData = ( QueueHandle_t ) xQueueSelectFromSet(xQueueSet,
portMAX_DELAY );

        /* An indefinite block time was used when reading from the queue set, so
        xQueueSelectFromSet() will not have returned unless one of the queues in the set contained
        data, and xQueueThatContainsData cannot be NULL. Read from the queue. It is not necessary
        to specify a block time because it is known the queue contains data. The block time is set
        to 0. */

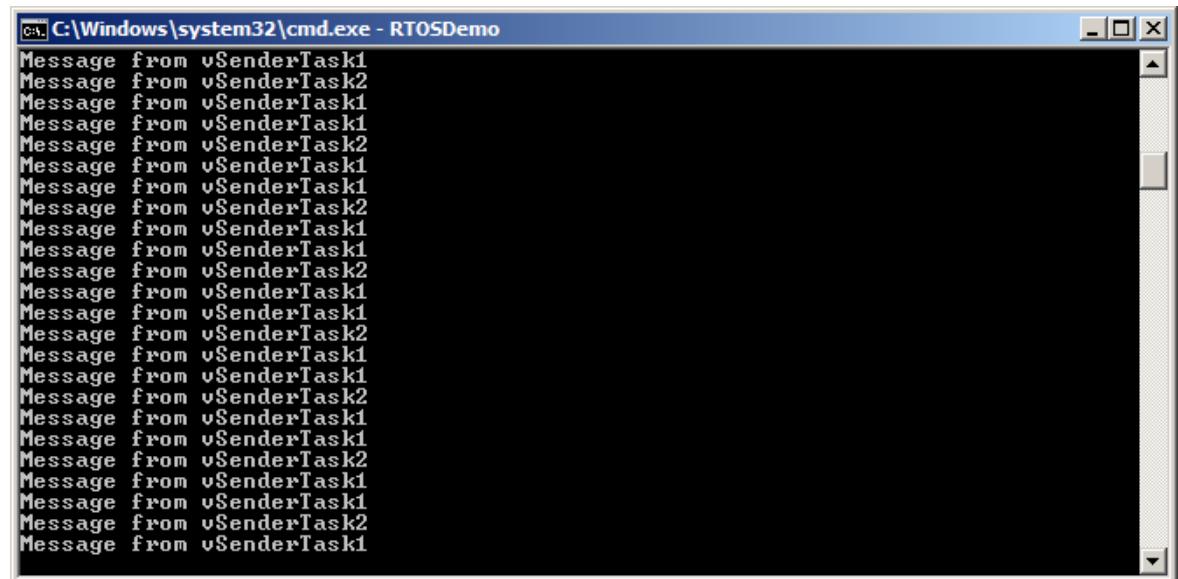
        xQueueReceive( xQueueThatContainsData, &pcReceivedString, 0 );

        /* Print the string received from the queue. */

        vPrintString( pcReceivedString );
    }
}

```

下面是输出的内容。接收任务接收来自两个发送任务的字符串。vSenderTask1() 使用的阻止时间是 vSenderTask2() 使用的阻止时间的一半，因此，vSenderTask1() 发送的字符串比 vSenderTask2() 发送的字符串输出的频率高一倍。



## 更真实的队列集使用案例

在前例中，队列集仅包含两个队列。这两个队列都用于发送字符指针。在实际应用程序中，队列集可能既包含队列又包含信号灯，并且队列可能不只包含相同的数据类型。在这种情况下，需要在使用返回值之前测试 xQueueSelectFromSet() 返回的值。

下面的代码说明当集具有以下成员时，如何使用 xQueueSelectFromSet() 返回的值：

1. 二进制信号灯。
2. 从中读取字符指针的队列。
3. 从中读取 uint32\_t 值的队列。

这段代码假定已创建队列和信号灯并已将它们添加到队列集。

```
/* The handle of the queue from which character pointers are received. */
QueueHandle_t xCharPointerQueue;

/* The handle of the queue from which uint32_t values are received.*/
QueueHandle_t xUint32tQueue;

/* The handle of the binary semaphore. */
SemaphoreHandle_t xBinarySemaphore;

/* The queue set to which the two queues and the binary semaphore belong. */
QueueSetHandle_t xQueueSet;

void vAMoreRealisticReceiverTask( void *pvParameters )
{
    QueueSetMemberHandle_t xHandle;
    char *pcReceivedString;
    uint32_t ulRecievedValue;
    const TickType_t xDelay100ms = pdMS_TO_TICKS( 100 );
    for( ;; )
    {
        /* Block on the queue set for a maximum of 100ms to wait for one of the members of
        the set to contain data. */
        xHandle = xQueueSelectFromSet( xQueueSet, xDelay100ms );

        /* Test the value returned from xQueueSelectFromSet(). If the returned value
        is NULL, then the call to xQueueSelectFromSet() timed out. If the returned value is
        not NULL, then the returned value will be the handle of one of the set's members. The
        QueueSetMemberHandle_t value can be cast to either a QueueHandle_t or a SemaphoreHandle_t.
        Whether an explicit cast is required depends on the compiler. */
        if( xHandle == NULL )
    }
```

```
/* The call to xQueueSelectFromSet() timed out. */
}

else if( xHandle == ( QueueSetMemberHandle_t ) xCharPointerQueue )

{

    /* The call to xQueueSelectFromSet() returned the handle of the queue that
receives character pointers. Read from the queue. The queue is known to contain data, so a
block time of 0 is used. */

    xQueueReceive( xCharPointerQueue, &pcReceivedString, 0 );

    /* The received character pointer can be processed here... */

}

else if( xHandle == ( QueueSetMemberHandle_t ) xUint32tQueue )

{

    /* The call to xQueueSelectFromSet() returned the handle of the queue that
receives uint32_t types. Read from the queue. The queue is known to contain data, so a
block time of 0 is used. */

    xQueueReceive(xUint32tQueue, &ulRecievedValue, 0 );

    /* The received value can be processed here... */

}

else if( xHandle == ( QueueSetMemberHandle_t ) xBinarySemaphore )

{

    /* The call to xQueueSelectFromSet() returned the handle of the binary
semaphore. Take the semaphore now. The semaphore is known to be available, so a block time
of 0 is used. */

    xSemaphoreTake( xBinarySemaphore, 0 );

    /* Whatever processing is necessary when the semaphore is taken can be
performed here... */

}

}

}
```

## 使用队列来创建邮箱

嵌入式社区对邮箱的含义并无共识。在本指南中，我们使用这个术语指长度为 1 的队列。队列因其在应用程序中的使用方式（而不是因其有队列之外的功能）可能被描述为邮箱。

- 队列用于将数据从一个任务发送到另一个任务，或从中断服务例程发送到任务。发送方将项目放入队列，接收方从队列中移除项目。数据通过队列从发送方传递给接收方。
- 邮箱用于保存可由任何任务或中断服务例程读取的数据。数据不会通过邮箱传递。它一直保留在邮箱中，直至被覆盖。发送方将覆盖邮箱中的值。接收方从邮箱中读取值，但是不从邮箱中移除值。

xQueueOverwrite() 和 xQueuePeek() API 函数允许将队列用作邮箱。

下面的代码说明创建用作邮箱的队列。

```

/* A mailbox can hold a fixed-size data item. The size of the data item is set when the
mailbox (queue) is created. In this example, the mailbox is created to hold an Example_t
structure. Example_t includes a timestamp to allow the data held in the mailbox to note
the time at which the mailbox was last updated. The timestamp used in this example is for
demonstration purposes only. A mailbox can hold any data the application writer wants, and
the data does not need to include a timestamp. */

typedef struct xExampleStructure

{

    TickType_t xTimeStamp;

    uint32_t ulValue;

} Example_t;

/* A mailbox is a queue, so its handle is stored in a variable of type
QueueHandle_t. */

QueueHandle_t xMailbox;

void vAFunction( void )

{

    /* Create the queue that is going to be used as a mailbox. The queue has a length of 1
    to allow it to be used with the xQueueOverwrite() API function, which is described below.
    */

    xMailbox = xQueueCreate( 1, sizeof( Example_t ) );
}

```

## xQueueOverwrite() API 函数

就像 xQueueSendToBack() API 函数一样，xQueueOverwrite() API 函数向队列发送数据。与 xQueueSendToBack() 不同，如果队列已满，则 xQueueOverwrite() 将覆盖队列中的数据。

xQueueOverwrite() 应只用于长度为 1 的队列。通过这一限制，如果队列已满，函数的实现不必确定应覆盖队列中的哪一个项目。

注意：请勿从中断服务例程中调用 xQueueOverwrite()。而应使用中断安全版本 xQueueOverwriteFromISR()。

下面是 xQueueOverwrite() API 函数原型。

```
BaseType_t xQueueOverwrite( QueueHandle_t xQueue, const void * pvItemToQueue );
```

下表列出了 xQueueOverwrite() 参数和返回值。

参数名称/返回的值	描述
xQueue	将向其发送（写入）数据的队列的句柄。该队列句柄从 xQueueCreate() 调用返回，用于创建队列。

pvItemToQueue	指针，它指向要复制到队列的数据。
返回的值	队列创建时就设置了队列可以保存的每个项目的大 小，因此相应数量的字节将从 pvItemToQueue 复制 到队列存储区域。  即使队列已满，xQueueOverwrite() 也将写入队列， 因此 pdPASS 是唯一可能的返回值。

下面的代码说明用来写入之前创建的邮箱（队列）的 xQueueOverwrite()。

```
void vUpdateMailbox( uint32_t ulnewValue )

{
    /* Example_t was defined in the earlier code example. */

    Example_t xData;

    /* Write the new data into the Example_t structure.*/
    xData.ulValue = ulnewValue;

    /* Use the RTOS tick count as the timestamp stored in the Example_t structure. */
    xData.xTimeStamp = xTaskGetTickCount();

    /* Send the structure to the mailbox, overwriting any data that is already in the
     mailbox. */
    xQueueOverwrite( xMailbox, &xData );
}
```

## xQueuePeek() API 函数

xQueuePeek() 用于从队列中接收（读取）项目，而不从队列中移除项目。xQueuePeek() 接收队列头部的数据，而不修改存储在队列中的数据或队列的数据存储顺序。

注意：请勿从中断服务例程中调用 xQueuePeek()。而应使用中断安全版本 xQueuePeekFromISR()。

xQueuePeek() 具有与 xQueueReceive() 相同的函数参数和返回值。

下面的代码说明使用 xQueuePeek() 接收已发布到上一示例中创建的邮箱（队列）的项目。

```
BaseType_t vReadMailbox( Example_t *pxData )

{
    TickType_t xPreviousTimeStamp;

    BaseType_t xDataUpdated;

    /* This function updates an Example_t structure with the latest value received from the
     mailbox. Record the timestamp already contained in *pxData before it gets overwritten by
     the new data. */
    xPreviousTimeStamp = pxData->xTimeStamp;
```

```
/* Update the Example_t structure pointed to by pxData with the data contained in
the mailbox. If xQueueReceive() was used here, then the mailbox would be left empty
and the data could not then be read by any other tasks. Using xQueuePeek() instead of
xQueueReceive() ensures the data remains in the mailbox. A block time is specified, so
the calling task will be placed in the Blocked state to wait for the mailbox to contain
data should the mailbox be empty. An infinite block time is used, so it is not necessary
to check the value returned from xQueuePeek(). xQueuePeek() will only return when data is
available. */

xQueuePeek( xMailbox, pxData, portMAX_DELAY );

/* Return pdTRUE if the value read from the mailbox has been updated since this
function was last called. Otherwise, return pdFALSE. */

if( pxData->xTimeStamp > xPreviousTimeStamp )

{
    xDataUpdated = pdTRUE;

}
else

{
    xDataUpdated = pdFALSE;
}

return xDataUpdated;
}
```

# 软件计时器管理

本节内容：

- 软件计时器与任务的特征对比。
- RTOS 守护程序任务。
- 计时器命令队列。
- 一次性软件计时器与周期性软件计时器之间的差异。
- 如何创建、启动、重置以及更改软件计时器周期。

软件计时器用于将函数的执行计划在未来的设定时间或按固定频率周期性执行。由软件计时器执行的函数称为软件计时器的回调函数。

软件计时器由 FreeRTOS 内核实现和控制，无需硬件支持，与硬件计时器或硬件计数器不相关。

根据 FreeRTOS 使用创新设计以确保效率最大化的理念，除非执行软件计时器回调函数，否则软件计时器不使用任何处理时间。

软件计时器功能是可选的。要包括软件计时器功能，请执行以下操作：

1. 构建 FreeRTOS 源文件 FreeRTOS/Source/timers.c，并将其作为项目的组成部分。
2. 在 FreeRTOSConfig.h 中，将 configUSE\_TIMERS 设置为 1。

## 软件计时器回调函数

软件计时器回调函数是以 C 函数形式实现的。它们唯一的特别之处，是它们的原型必须返回 void 并接受软件计时器句柄作为其唯一的参数。

下面是回调函数原型。

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

软件计时器回调函数从头到尾都执行，以正常方式退出。它们应简短，不得进入“被阻止”状态。

注意：软件计时器回调函数在 FreeRTOS 计划程序启动时自动创建的任务的上下文中执行。因此，它们不能调用将导致调用任务进入“被阻止”状态的 FreeRTOS API 函数。您可以调用 xQueueReceive() 这样的函数，但前提是该函数的 xTicksToWait 参数（指定函数的阻止时间）设置为 0。您不能调用 vTaskDelay() 这样的函数，因为这样会将调用任务置于“被阻止”状态。

## 软件计时器的属性和状态

### 软件计时器的周期

软件计时器的周期 是从软件计时器启动到其回调函数执行之间的时间。

## 一次性和自动重新加载计时器

软件计时器有两种：

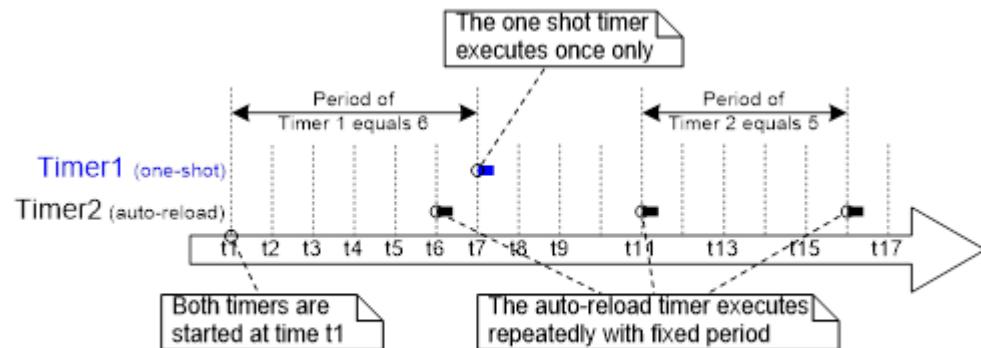
- 一次性计时器

一次性计时器在启动后只执行一次其回调函数。一次性计时器可以手动重启，但不会自行重启。

- 自动重新加载计时器

在启动后，自动重新加载计时器每次到期时都会自行重启，导致其回调函数周期性执行。

下图说明一次性计时器与自动重新加载计时器的行为的差异。垂直虚线标示发生时钟中断的时间。



- Timer 1 是一次性计时器，它的时段包含 6 个时钟周期。它在时间 t1 启动，因此其回调函数在 6 个时钟周期后（时间 t7）执行。由于 Timer 1 是一次性计时器，它的回调函数不会再次执行。
- Timer 2 是自动重新加载计时器，它的周期为 5 个时钟周期。它在时间 t1 启动，因此其回调函数在时间 t1 后每 5 个时钟周期执行一次。在图中就是在时间 t6、t11 和 t16。

## 软件计时器状态

软件计时器可以处于以下两种状态之一：

- 休眠

休眠软件计时器存在，可以通过其句柄引用，但是不会运行，因此其回调函数不会执行。

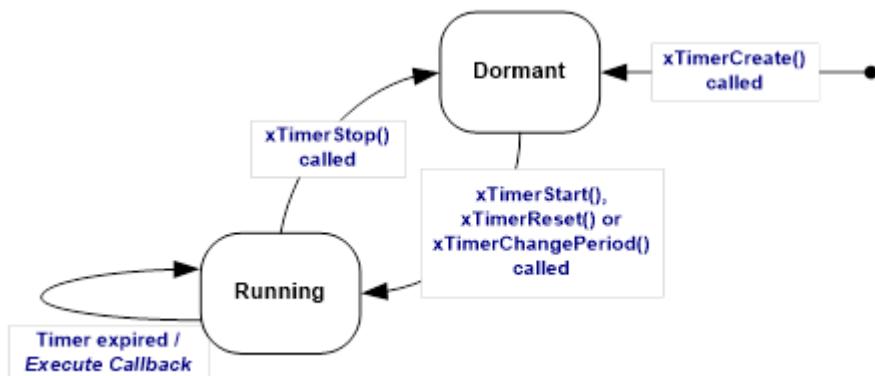
- 正在运行

正在运行的软件计时器将在软件计时器进入“运行”状态或上次重置后经过等于其周期的时间后，执行其回调函数。

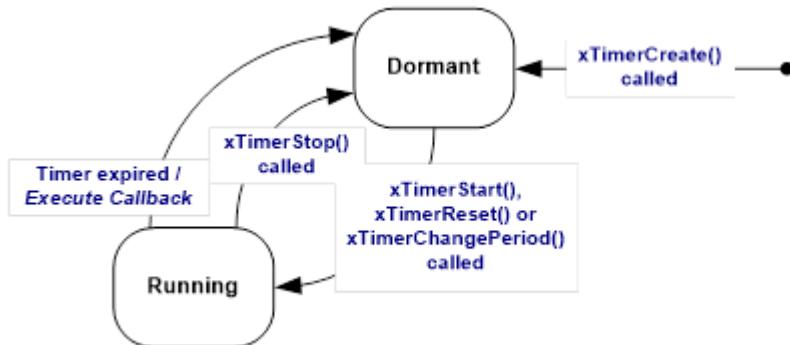
下面两个图分别说明自动重新加载计时器和一次性计时器在“休眠”和“运行”状态之间可能的转换。这两个图之间的主要差异是计时器到期后进入的状态。自动重新加载计时器执行其回调函数，然后重新进入“运行”状态。一次性计时器执行其回调函数，然后进入“休眠”状态。

xTimerDelete() API 函数删除计时器。计时器随时可删除。

下图说明自动重新加载软件计时器的状态和状态转换。



下图说明一次性软件计时器的状态和状态转换。



## 软件计时器上下文

### RTOS 守护程序（计时器服务）任务

所有软件计时器回调函数都在同一个 RTOS 守护程序（即计时器服务）任务的上下文中执行。（该任务以前称为计时器服务任务，因为它最初仅用于执行软件计时器回调函数。现在该任务也用于其他用途，因此以 RTOS 守护程序任务这个更通用的名称为人所知。）

守护程序任务是计划程序启动时自动创建的标准 FreeRTOS 任务。它的优先级和堆栈大小分别由 `configTIMER_TASK_PRIORITY` 和 `configTIMER_TASK_STACK_DEPTH` 编译时间配置常量设置。这两个常量都在 `FreeRTOSConfig.h` 中定义。

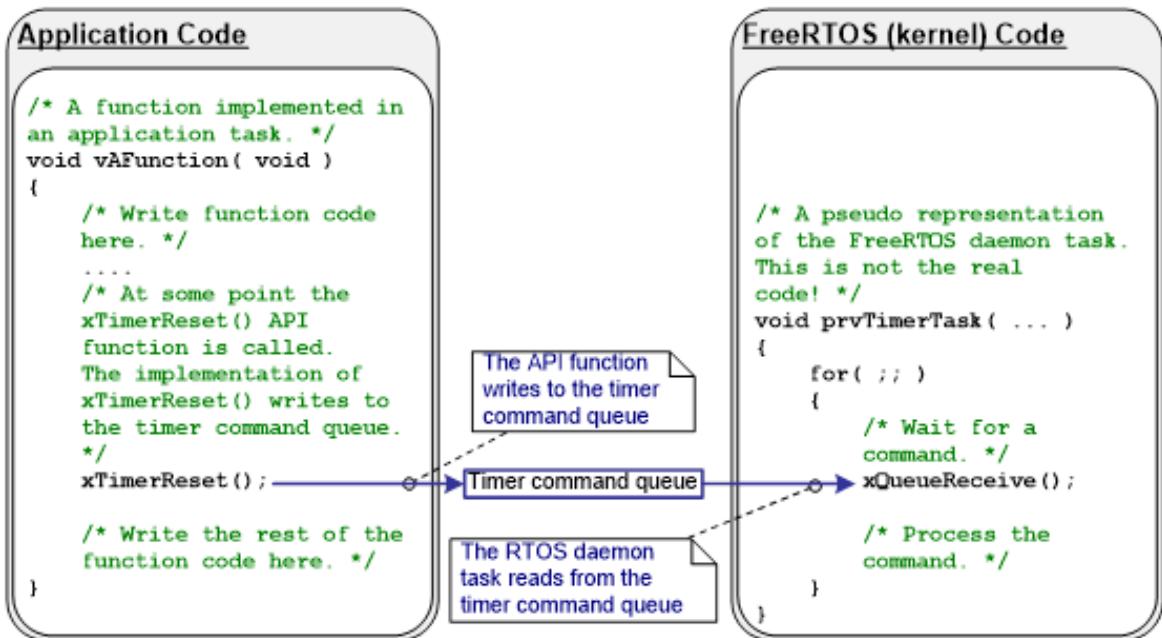
软件计时器回调函数不得调用将导致调用任务进入“被阻止”状态的 FreeRTOS API 函数，因为那样会导致守护程序任务进入“被阻止”状态。

## 计时器命令队列

软件计时器 API 函数通过一个名为计时器命令队列的队列从调用任务向守护程序任务发送命令。举例来说，有“start a timer”、“stop a timer”和“reset a timer”命令。

计时器命令队列是计划程序启动时自动创建的标准 FreeRTOS 队列。计时器命令队列的长度由 `FreeRTOSConfig.h` 中的 `configTIMER_QUEUE_LENGTH` 编译时间配置常量设置。

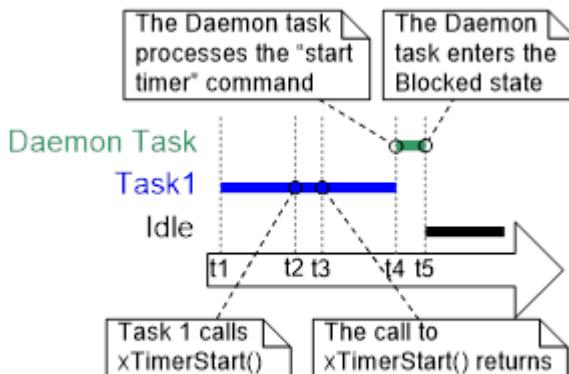
下图说明软件计时器 API 函数用来与 RTOS 守护程序任务通信的命令队列。



## 守护程序任务计划

守护程序任务和其他所有 FreeRTOS 任务一样计划。它仅当是能够运行的最高优先级任务时才处理命令或执行计时器回调函数。下图说明 configTIMER\_TASK\_PRIORITY 设置如何影响执行模式。

此图说明当守护程序任务的优先级低于调用 `xTimerStart()` API 函数的任务的优先级时的执行模式。



### 1. 在时间 t1

Task 1 处于“运行”状态，守护程序任务处于“被阻止”状态。

如果有命令发送到计时器命令队列，守护程序任务将离开“被阻止”状态，在这种情况下它将处理该命令。  
如果软件计时器到期，它将执行软件计时器回调函数。

### 2. 在时间 t2

Task 1 调用 `xTimerStart()`。

`xTimerStart()` 发送一条命令到计时器命令队列，使守护程序任务离开“被阻止”状态。Task 1 的优先级高于守护程序任务的优先级，因此守护程序任务不会抢先于 Task 1 执行。

Task 1 仍处于“运行”状态。守护程序任务已离开“被阻止”状态，进入“准备就绪”状态。

3. 在时间 t3

Task 1 完成 xTimerStart() API 函数的执行。Task 1 从 xTimerStart() 函数的开始到结束，一直在执行该函数，未离开“运行”状态。

4. 在时间 t4

Task 1 调用一个导致它进入“被阻止”状态的 API 函数。现在，守护程序任务是处于“准备就绪”状态的最高优先级任务，因此计划程序选择守护程序任务作为进入“运行”状态的任务。然后，守护程序任务开始处理 Task 1 发送到计时器命令队列的命令。

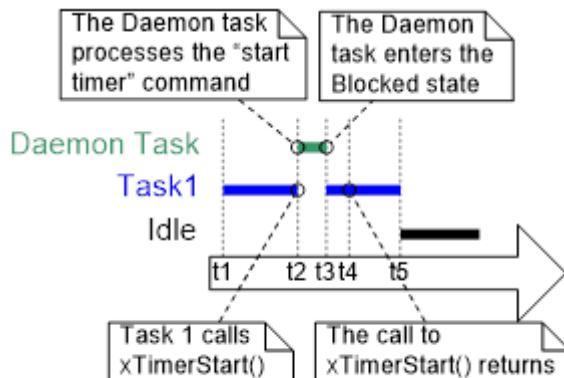
注意：启动的软件计时器的到期时间从“start a timer”命令发送到计时器命令队列时开始计算。而不是从守护程序任务从计时器命令队列接收到“start a timer”时开始计算。

5. 在时间 t5

守护程序任务已完成 Task 1 发送给它的命令的处理，并尝试从计时器命令队列接收更多数据。计时器命令队列为空，因此守护程序任务重新进入“被阻止”状态。如果有命令发送到计时器命令队列或软件计时器到期，守护程序任务将再次离开“被阻止”状态。

现在，空闲任务是处于“准备就绪”状态的最高优先级任务，因此计划程序选择空闲任务作为进入“运行”状态的任务。

下图的情况与上图类似。这次守护程序任务的优先级高于调用 xTimerStart() 的任务的优先级。



1. 在时间 t1

与之前一样，Task 1 处于“运行”状态，守护程序任务处于“被阻止”状态。

2. 在时间 t2

Task 1 调用 xTimerStart()。

xTimerStart() 发送一条命令到计时器命令队列，使守护程序任务离开“被阻止”状态。守护程序任务的优先级高于 Task 1 的优先级，因此计划程序选择守护程序任务作为进入“运行”状态的任务。

Task 1 完成 xTimerStart() 函数的执行之前，守护程序任务抢先执行，因此 Task 1 现在处于“准备就绪”状态。

守护程序任务开始处理 Task 1 发送到计时器命令队列的命令。

3. 在时间 t3

守护程序任务已完成 Task 1 发送给它的命令的处理，并尝试从计时器命令队列接收更多数据。计时器命令队列为空，因此守护程序任务重新进入“被阻止”状态。

现在，Task 1 是处于“准备就绪”状态的最高优先级任务，因此计划程序选择 Task 1 作为进入“运行”状态的任务。

#### 4. 在时间 t4

Task 1 完成 xTimerStart() 函数的执行之前，守护程序任务抢先执行，因此 Task 1 只有在重新进入“运行”状态之后才退出 xTimerStart()（从该函数返回）。

#### 5. 在时间 t5

Task 1 调用一个导致它进入“被阻止”状态的 API 函数。现在，空闲任务是处于“准备就绪”状态的最高优先级任务，因此计划程序选择空闲任务作为进入“运行”状态的任务。

在第一种情况的图中，在 Task 1 向计时器命令队列发送命令到守护程序任务接收并处理该命令之间有一段时间。在 Task 1 从发送命令的函数返回之前，守护程序任务已接收并处理 Task 1 发来的命令。

发送到计时器命令队列的命令包含时间戳。时间戳用于计算从应用程序任务发送命令到守护程序任务处理命令之间的全部时间。例如，如果发送“start a timer”命令要启动的计时器的周期为 10 个时钟周期，则时间戳用来确保启动的计时器的时间戳在该命令发送之后 10 个时钟周期到期，而不是在守护程序处理该命令 10 个时钟周期后到期。

## 创建并启动软件计时器

### xTimerCreate() API 函数

FreeRTOS V9.0.0 还包含 xTimerCreateStatic() 函数，该函数用于分配在编译时静态创建计时器所需的内存。软件计时器必须先显式创建，然后才能使用。

软件计时器通过类型为 TimerHandle\_t 的变量引用。xTimerCreate() 用于创建软件计时器并返回一个 TimerHandle\_t 来引用它所创建的软件计时器。软件计时器以“休眠”状态创建。

软件计时器可以在计划程序运行之前创建，也可以在计划程序已启动后从任务中创建。

下面是 xTimerCreate() API 函数原型。

```
TimerHandle_t xTimerCreate( const char * const pcTimerName, TickType_t xTimerPeriodInTicks,  
                           UBaseType_t uxAutoReload, void * pvTimerID, TimerCallbackFunction_t pxCallbackFunction );
```

下表列出了 xTimerCreate() 参数和返回值。

参数名称/返回的值	描述
pcTimerName	计时器的描述性名称。FreeRTOS 不使用此名称。它只是用于在调试时提供辅助。以人类易读的名称来标识计时器，比尝试用句柄进行标识要简单得多。
xTimerPeriodInTicks	以时钟周期数指定的计时器周期。宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为时钟周期数。

uxAutoReload	将 uxAutoReload 设置为 pdTRUE 可创建自动重新加载计时器。将 uxAutoReload 设置为 pdFALSE 可创建一次性计时器。
pvTimerID	每个软件计时器都有一个 ID 值。该 ID 是一个 void 指针，应用程序编写者可将它用于任何用途。当有多个软件计时器使用同一个回调函数时，此 ID 尤其有用，因为它可用来提供特定于计时器的存储。  pvTimerID 为创建的任务设置初始 ID 值。
pxCallbackFunction	软件计时器回调函数就是符合 <a href="#">软件计时器回调函数 (p. 89)</a> 中所示原型的 C 函数。pxCallbackFunction 参数是一个指针，它指向用作所创建软件计时器的回调函数的函数（实际上就是函数名称）。
返回的值	如果返回 NULL，则表示无法创建软件计时器，因为没有足够的堆内存可供 FreeRTOS 分配数据结构。  如果返回非 NULL 值，表示软件计时器已成功创建。返回的值是所创建的计时器的句柄。

## xTimerStart() API 函数

xTimerStart() 用于启动处于“休眠”状态的软件计时器，或重置（重启）处于“运行”状态的软件计时器。xTimerStop() 用于停止处于“运行”状态的软件计时器。停止软件计时器与将计时器转换到“休眠”状态是一样的。

您可以在计划程序启动前调用 xTimerStart()，但是在计划程序启动之前，软件计时器不会启动。

注意：请勿从中断服务例程中调用 xTimerStart()。而应使用中断安全版本 xTimerStartFromISR()。

下面是 xTimerStart() API 函数原型。

```
TimerHandle_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

下表列出了 xTimerStart() 参数和返回值。

参数名称/返回的值	描述
xTimer	要启动或重置的软件计时器的句柄。该句柄从用于创建软件计时器的 xTimerCreate() 调用返回。
xTicksToWait	xTimerStart() 使用计时器命令队列将“start a timer”命令发送到守护程序任务。xTicksToWait 指定在队列已满的情况下，调用函数保持“被阻止”状态等待计时器命令队列中空间变得可用的最长时间量。  如果 xTicksToWait 为零且计时器命令队列已满，xTimerStart() 将立即返回。  阻止时间以计时周期指定，因此它表示的绝对时间取决于计时频率。宏 pdMS_TO_TICKS() 可用于将毫秒为单位指定的时间转换为计时周期数。

	<p>如果 FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 设置为 1，则将 xTicksToWait 设置为 portMAX_DELAY 会导致调用函数无限期（无超时）保持“被阻止”状态以等待计时器命令队列中空间变得可用。</p> <p>如果在计划程序启动之前调用 xTimerStart()，则将忽略 xTicksToWait 的值，并且 xTimerStart() 将像 xTicksToWait 设置为零一样操作。</p>
返回的值	<p>可能的返回值有两个：</p> <ol style="list-style-type: none"><li>1. pdPASS</li></ol> <p>仅当“start a timer”命令成功发送到计时器命令队列时才会返回 pdPASS。</p> <p>如果守护程序任务的优先级高于调用 xTimerStart() 的任务的优先级，则计划程序将确保在 xTimerStart() 返回之前处理 start 命令。这是因为只要计时器命令队列中有数据，守护程序任务就会抢先于调用 xTimerStart() 的任务执行。</p> <p>如果指定了阻止时间（xTicksToWait 不为零），则可能在函数返回之前，调用任务置于“被阻止”状态以等待计时器命令队列中的空间变为可用，但是在阻止时间到期之前数据已成功写入计时器命令队列。</p> <ol style="list-style-type: none"><li>2. pdFALSE</li></ol> <p>如果因为计时器命令队列已满，导致“start a timer”命令无法写入此队列，则返回 pdFALSE。</p> <p>如果指定了阻止时间（xTicksToWait 不为零），则可能是调用任务置于“被阻止”状态以等待守护程序任务在计时器命令队列中腾出空间，但是在此之前，指定的阻止时间就已到期。</p>

## 创建一次性和自动重新加载计时器 (示例 13 )

此示例创建并启动一次性计时器和自动重新加载计时器。

```
/* The periods assigned to the one-shot and auto-reload timers are 3.333 second and half a
second, respectively. */

#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS( 3333 )

#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 500 )

int main( void )

{
    TimerHandle_t xAutoReloadTimer, xOneShotTimer;

    BaseType_t xTimer1Started, xTimer2Started;
```

```
/* Create the one-shot timer, storing the handle to the created timer in xOneShotTimer.  
*/  
  
xOneShotTimer = xTimerCreate( /* Text name for the software timer - not  
used by FreeRTOS. */ "OneShot", /* The software timer's period, in ticks. */  
mainONE_SHOT_TIMER_PERIOD, /* Setting uxAutoRealod to pdFALSE creates a one-shot software  
timer. */ pdFALSE, /* This example does not use the timer ID. */ 0, /* The callback  
function to be used by the software timer being created. */ prvOneShotTimerCallback );  
  
/* Create the auto-reload timer, storing the handle to the created timer in  
xAutoReloadTimer. */  
  
xAutoReloadTimer = xTimerCreate( /* Text name for the software timer - not  
used by FreeRTOS. */ "AutoReload", /* The software timer's period, in ticks. */  
mainAUTO_RELOAD_TIMER_PERIOD, /* Setting uxAutoRealod to pdTRUE creates an auto-reload  
timer. */ pdTRUE, /* This example does not use the timer ID. */ 0, /* The callback  
function to be used by the software timer being created. */ prvAutoReloadTimerCallback );  
  
/* Check the software timers were created. */  
  
if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )  
{  
  
    /* Start the software timers, using a block time of 0 (no block time). The  
scheduler has not been started yet so any block time specified here would be ignored  
anyway. */  
  
    xTimer1Started = xTimerStart( xOneShotTimer, 0 );  
  
    xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );  
  
    /* The implementation of xTimerStart() uses the timer command queue, and  
xTimerStart() will fail if the timer command queue gets full. The timer service task does  
not get created until the scheduler is started, so all commands sent to the command queue  
will stay in the queue until after the scheduler has been started. Check both calls to  
xTimerStart() passed. */  
  
    if( ( xTimer1Started == pdPASS ) && ( xTimer2Started == pdPASS ) )  
{  
  
        /* Start the scheduler. */  
  
        vTaskStartScheduler();  
  
    }  
  
}  
  
/* As always, this line should not be reached. */  
for( ; );  
}
```

这些计时器的回调函数在每次被调用时输出一条消息。下面是一次性计时器回调函数的实现。

```
static void prvOneShotTimerCallback( TimerHandle_t xTimer )  
{  
  
    TickType_t xTimeNow;  
  
    /* Obtain the current tick count. */
```

```
xTimeNow = xTaskGetTickCount();

/* Output a string to show the time at which the callback was executed. */

vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );

/* File scope variable. */

ulCallCount++;

}
```

下面是自动重新加载计时器回调函数的实现。

```
static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )

{
    TickType_t xTimeNow;

    /* Obtain the current tick count. */

    xTimeNow = xTaskGetTickCount();

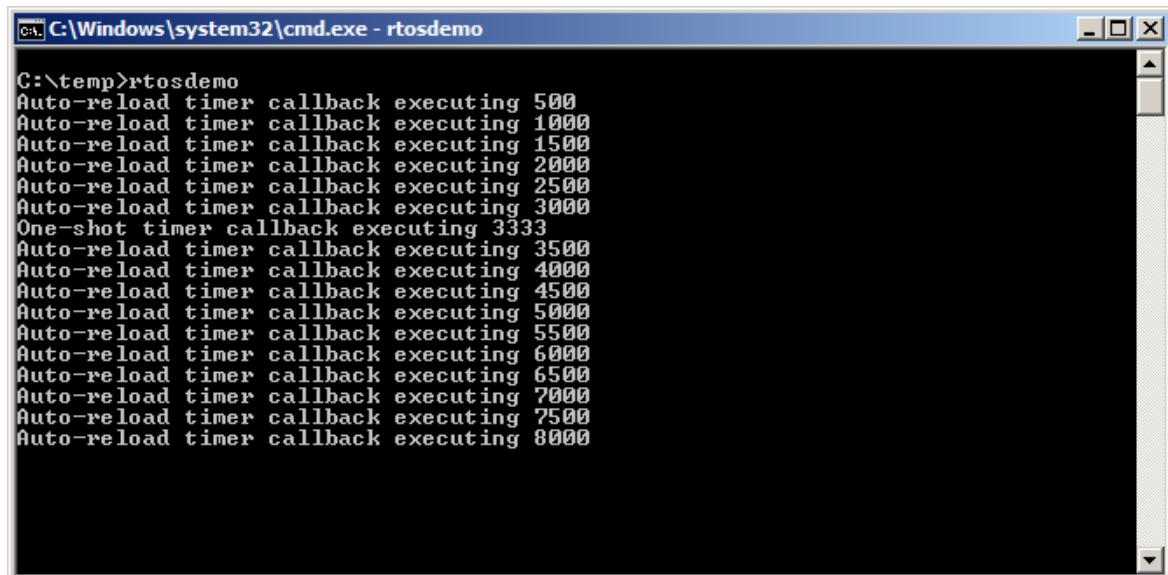
    /* Output a string to show the time at which the callback was executed. */

    vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow);

    ulCallCount++;

}
```

执行此示例将产生以下输出。它说明固定周期为 500 个时钟周期 ( mainAUTO\_RELOAD\_TIMER\_PERIOD 设置为 500 ) 的自动重新加载计时器的回调函数执行，以及计时计数为 3333 ( mainONE\_SHOT\_TIMER\_PERIOD 设置为 3333 ) 的一次性计时器回调函数执行。



```
C:\temp>rtosdemo
Auto-reload timer callback executing 500
Auto-reload timer callback executing 1000
Auto-reload timer callback executing 1500
Auto-reload timer callback executing 2000
Auto-reload timer callback executing 2500
Auto-reload timer callback executing 3000
One-shot timer callback executing 3333
Auto-reload timer callback executing 3500
Auto-reload timer callback executing 4000
Auto-reload timer callback executing 4500
Auto-reload timer callback executing 5000
Auto-reload timer callback executing 5500
Auto-reload timer callback executing 6000
Auto-reload timer callback executing 6500
Auto-reload timer callback executing 7000
Auto-reload timer callback executing 7500
Auto-reload timer callback executing 8000
```

## 计时器 ID

每个软件计时器都有一个 ID，这是可由应用程序编写者用于任何用途的标签值。该 ID 存储在 void 指针 (`void *`) 中，因此它可以直接存储一个整数值，指向任何其他对象，或用作函数指针。

当创建软件计时器时，会对该 ID 分配初始值。该 ID 可以使用 `vTimerSetTimerID()` API 函数更新，可使用 `pvTimerGetTimerID()` API 函数查询。

与其他软件计时器 API 函数不同，`vTimerSetTimerID()` 和 `pvTimerGetTimerID()` 直接访问软件计时器。它们不向计时器命令队列发送命令。

### `vTimerSetTimerID()` API 函数

下面是 `vTimerSetTimerID()` 函数原型。

```
void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID );
```

下表列出了 `vTimerSetTimerID()` 的参数。

参数名称/返回的值	描述
<code>xTimer</code>	要使用新 ID 值更新的软件计时器的句柄。该句柄从用于创建软件计时器的 <code>xTimerCreate()</code> 调用返回。
<code>pvNewID</code>	将设置的软件计时器 ID 值。

### `pvTimerGetTimerID()` API 函数

下面是 `pvTimerGetTimerID()` 函数原型。

```
void *pvTimerGetTimerID( TimerHandle_t xTimer );
```

下表列出了 `pvTimerGetTimerID()` 参数和返回值。

参数名称/返回的值	描述
<code>xTimer</code>	要查询的软件计时器的句柄。该句柄从用于创建软件计时器的 <code>xTimerCreate()</code> 调用返回。
返回的值	要查询的软件计时器的 ID。

## 使用回调函数参数和软件计时器 ID (示例 14)

同一个回调函数可以分配给多个软件计时器。完成后，回调函数的参数用于确定哪一个软件计时器到期。

示例 13 使用了两个单独的回调函数：一个用于一次性计时器，另一个用于自动重新加载计时器。此示例创建类似的功能，但对两个软件计时器分配的是同一个回调函数。

下面的代码说明如何对两个计时器使用同一个回调函数 `prvTimerCallback()`。

```
/* Create the one-shot software timer, storing the handle in xOneShotTimer. */
```

```
xOneShotTimer = xTimerCreate( "OneShot", mainONE_SHOT_TIMER_PERIOD, pdFALSE, /* The
timer's ID is initialized to 0. */ 0, /* prvTimerCallback() is used by both timers. */
prvTimerCallback );

/* Create the auto-reload software timer, storing the handle in xAutoReloadTimer */

xAutoReloadTimer = xTimerCreate( "AutoReload", mainAUTO_RELOAD_TIMER_PERIOD, pdTRUE, /* The
timer's ID is initialized to 0. */ 0, /* prvTimerCallback() is used by both timers. */
prvTimerCallback );
```

其中任一计时器到期时，prvTimerCallback() 都会执行。prvTimerCallback() 的实现使用函数的参数来确定是因为一次性计时器还是自动重新加载计时器到期而被调用。

prvTimerCallback() 还说明如何将软件计时器 ID 用作特定于计时器的存储。每个软件计时器都在自己的 ID 中保留其已到期的次数的计数。自动重新加载计时器使用该计数在它第五次执行时自行停止。

下面是 prvTimerCallback() 的实现。

```
static void prvTimerCallback( TimerHandle_t xTimer )

{
    TickType_t xTimeNow;
    uint32_t ulExecutionCount;

    /* A count of the number of times this software timer has expired is stored in the
    timer's ID. Obtain the ID, increment it, then save it as the new ID value. The ID is a
    void pointer, so is cast to a uint32_t. */

    ulExecutionCount = ( uint32_t ) pvTimerGetTimerID( xTimer );
    ulExecutionCount++;
    vTimerSetTimerID( xTimer, ( void * ) ulExecutionCount );

    /* Obtain the current tick count. */

    xTimeNow = xTaskGetTickCount();

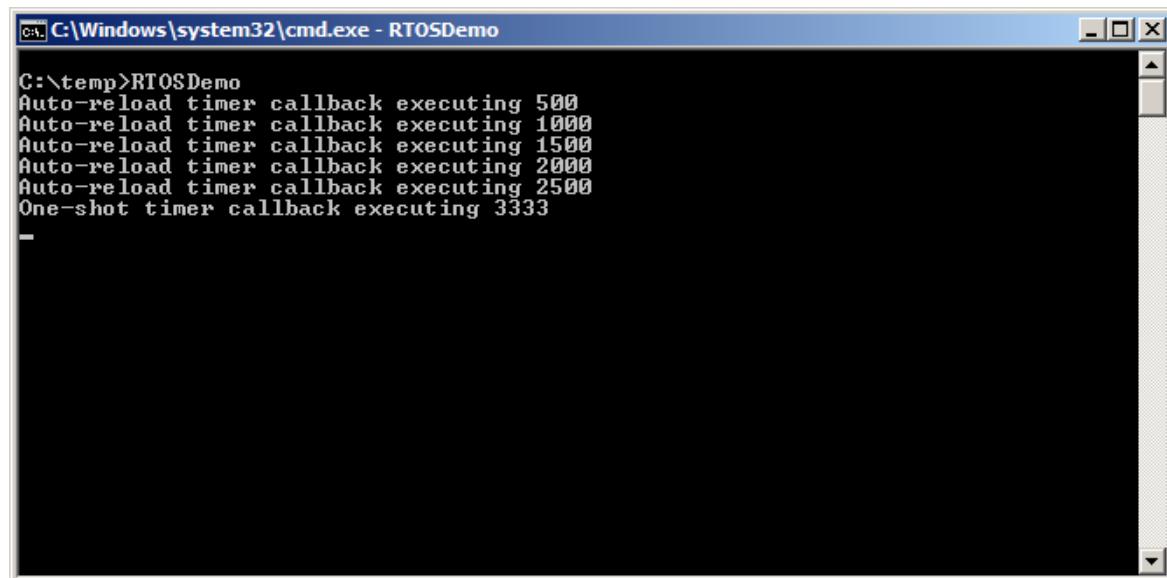
    /* The handle of the one-shot timer was stored in xOneShotTimer when the timer was
    created. Compare the handle passed into this function with xOneShotTimer to determine if
    it was the one-shot or auto-reload timer that expired, then output a string to show the
    time at which the callback was executed. */

    if( xTimer == xOneShotTimer )
    {
        vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );
    }
    else
    {
        /* xTimer did not equal xOneShotTimer, so it must have been the auto-reload timer
        that expired. */

        vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow );
        if( ulExecutionCount == 5 )
```

```
{  
    /* Stop the auto-reload timer after it has executed 5 times. This callback  
    function executes in the context of the RTOS daemon task, so must not call any functions  
    that might place the daemon task into the Blocked state. Therefore, a block time of 0 is  
    used. */  
  
    xTimerStop( xTimer, 0 );  
}  
}  
}
```

下面是输出的内容。自动重新加载计时器仅执行五次。



```
C:\temp>RTOSDemo  
Auto-reload timer callback executing 500  
Auto-reload timer callback executing 1000  
Auto-reload timer callback executing 1500  
Auto-reload timer callback executing 2000  
Auto-reload timer callback executing 2500  
One-shot timer callback executing 3333
```

## 更改计时器的周期

每个官方 FreeRTOS 端口都有相应的一个或多个示例项目。大多数示例项目都可自检查。LED 用于提供项目状态的视觉反馈。如果自检查始终能通过，则 LED 切换会很缓慢。如果自检查失败，则 LED 会快速切换。

有些示例项目在任务中执行自检查，使用 vTaskDelay() 函数来控制 LED 切换的速率。其他示例项目在软件计时器回调函数中执行自检查，使用计时器的周期来控制 LED 切换的速率。

### xTimerChangePeriod() API 函数

您可以使用 xTimerChangePeriod() 函数来更改软件计时器的周期。

如果 xTimerChangePeriod() 用于更改已在运行的计时器的周期，则计时器使用新周期值重新计算其到期时间。重新计算后的到期时间是相对于 xTimerChangePeriod() 被调用时的时间，而不是相对于计时器最初启动时的时间。

如果 xTimerChangePeriod() 用于更改“休眠”状态的计时器（未运行的计时器）的周期，则计时器计算到期时间并转换到“运行”状态（计时器将开始运行）。

注意：请勿从中断服务例程中调用 xTimerChangePeriod()。而应使用中断安全版本 xTimerChangePeriodFromISR()。

下面是 xTimerChangePeriod() API 函数原型。

```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer, TickType_t xNewTimerPeriodInTicks,  
                                TickType_t xTicksToWait );
```

下表列出了 xTimerChangePeriod() 参数和返回值。

参数名称/返回的值	描述
xTimer	要使用新周期值更新的软件计时器的句柄。该句柄从用于创建软件计时器的 xTimerCreate() 调用返回。
xTimerPeriodInTicks	软件计时器的新周期（以时钟周期数指定）。宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为时钟周期数。
xTicksToWait	<p>xTimerChangePeriod() 使用计时器命令队列将“change period”命令发送到守护程序任务。xTicksToWait 指定在队列已满的情况下，调用函数应保持“被阻止”状态等待计时器命令队列中空间变得可用的最长时间量。</p> <p>如果 xTicksToWait 为零且计时器命令队列已满，xTimerChangePeriod() 将立即返回。</p> <p>宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为计时周期数。</p> <p>如果 FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 设置为 1，则将 xTicksToWait 设置为 portMAX_DELAY 会导致调用函数无限期（无超时）保持“被阻止”状态以等待计时器命令队列中空间变得可用。</p> <p>如果在计划程序启动之前调用 xTimerChangePeriod()，则将忽略 xTicksToWait 的值，并且 xTimerChangePeriod() 将像 xTicksToWait 设置为零一样操作。</p>
返回的值	<p>可能的返回值有两个：</p> <ol style="list-style-type: none"><li>pdPASS</li></ol> <p>仅当数据成功发送到计时器命令队列时，才会返回 pdPASS。</p> <p>如果指定了阻止时间（xTicksToWait 不为零），则可能在函数返回之前，调用任务置于“被阻止”状态以等待计时器命令队列中的空间变为可用，但是在阻止时间到期之前数据已成功写入计时器命令队列。</p> <ol style="list-style-type: none"><li>pdFALSE</li></ol> <p>如果因为计时器命令队列已满，导致“change period”命令无法写入该队列，则返回 pdFALSE。</p>

如果指定了阻止时间 ( xTicksToWait 不为零 )，  
则调用任务将置于“被阻止”状态以等待另一个守护  
程序任务在队列中腾出空间，但是在此之前阻止  
时间就已到期。

下面的代码说明在软件计时器回调函数中的自检查功能可以如何使用 xTimerChangePeriod() 来增大自检查失败时 LED 切换的频率。执行自检查的软件计时器称为检查计时器。

```
/* The check timer is created with a period of 3000 milliseconds, resulting in the LED
   toggling every 3 seconds. If the self-checking functionality detects an unexpected state,
   then the check timer's period is changed to just 200 milliseconds, resulting in a much
   faster toggle rate. */

const TickType_t xHealthyTimerPeriod = pdMS_TO_TICKS( 3000 );

const TickType_t xErrorTimerPeriod = pdMS_TO_TICKS( 200 );

/* The callback function used by the check timer. */

static void prvCheckTimerCallbackFunction( TimerHandle_t xTimer )

{

    static BaseType_t xErrorDetected = pdFALSE;

    if( xErrorDetected == pdFALSE )

    {

        /* No errors have yet been detected. Run the self-checking function again.
        The function asks each task created by the example to report its own status, and also
        checks that all the tasks are actually still running (and so able to report their status
        correctly). */

        if( CheckTasksAreRunningWithoutError() == pdFAIL )

        {

            /* One or more tasks reported an unexpected status. An error might have
            occurred. Reduce the check timer's period to increase the rate at which this callback
            function executes, and in so doing also increase the rate at which the LED is toggled.
            This callback function is executing in the context of the RTOS daemon task, so a block
            time of 0 is used to ensure the Daemon task never enters the Blocked state. */

            xTimerChangePeriod( xTimer, /* The timer being updated. */ xErrorTimerPeriod, /
* The new period for the timer. */ 0 ); /* Do not block when sending this command. */

        }

        /* Latch that an error has already been detected. */

        xErrorDetected = pdTRUE;

    }

    /* Toggle the LED. The rate at which the LED toggles will depend on how
    often this function is called, which is determined by the period of the check
    timer. The timer's period will have been reduced from 3000ms to just 200ms if
    CheckTasksAreRunningWithoutError() has ever returned pdFAIL. */

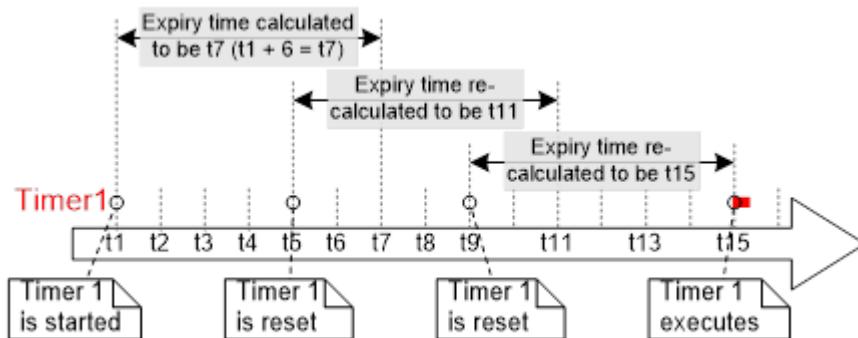
    ToggleLED();

}
```

}

## 重置软件计时器

重置软件计时器意味着重启计时器。重新计算计时器的到期时间，是相对于计时器重置时间而不是计时器最初启动的时间进行计算。下图说明周期为 6 的计时器启动，然后在最终到期并执行其回调函数之前重置两次。



- Timer 1 在时间  $t_1$  启动。其周期为 6，因此，它执行其回调函数的时间最初计算为  $t_7$ ，也就是在它启动后 6 个时钟周期。
- Timer 1 在到达时间  $t_7$  之前重置，因此是在它到期并执行其回调函数之前。Timer 1 在时间  $t_5$  重置，因此，它执行其回调函数的时间重新计算为  $t_{11}$ ，也就是在它重置后 6 个时钟周期。
- Timer 1 在时间  $t_{11}$  之前再次重置，因此是在它到期并执行其回调函数之前。Timer 1 在时间  $t_9$  重置，因此，它执行其回调函数的时间重新计算为  $t_{15}$ ，也就是在它上次重置后 6 个时钟周期。
- Timer 1 不再重置，因此它在时间  $t_{15}$  到期，并相应执行其回调函数。

## xTimerReset() API 函数

您可以使用 xTimerReset() API 函数重置计时器。

您也可以使用 xTimerReset() 启动处于“休眠”状态的计时器。

注意：请勿从中断服务例程中调用 xTimerReset()。而应使用中断安全版本 xTimerResetFromISR()。

下面是 xTimerReset() API 函数原型。

```
BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

下表列出了 xTimerReset() 参数和返回值。

参数名称/返回的值	描述
xTimer	要重置或启动的软件计时器的句柄。该句柄从用于创建软件计时器的 xTimerCreate() 调用返回。
xTicksToWait	xTimerChangePeriod() 使用计时器命令队列将“reset”命令发送到守护程序任务。xTicksToWait 指定在队列已满的情况下，调用函数应保持“被阻

	<p>止”状态等待计时器命令队列中空间变得可用的最长时间量。</p> <p>如果 xTicksToWait 为零且计时器命令队列已满，xTimerReset() 将立即返回。</p> <p>如果 FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 设置为 1，则将 xTicksToWait 设置为 portMAX_DELAY 会导致调用函数无限期（无超时）保持“被阻止”状态以等待计时器命令队列中空间变得可用。</p>
返回的值	<p>可能的返回值有两个：</p> <ol style="list-style-type: none"><li>1. pdPASS</li></ol> <p>仅当数据成功发送到计时器命令队列时，才会返回 pdPASS。</p> <p>如果指定了阻止时间（xTicksToWait 不为零），则可能在函数返回之前，调用任务置于“被阻止”状态以等待计时器命令队列中的空间变为可用，但是在阻止时间到期之前数据已成功写入计时器命令队列。</p> <ol style="list-style-type: none"><li>2. pdFALSE</li></ol> <p>如果因为计时器命令队列已满，导致“reset”命令无法写入该队列，则返回 pdFALSE。</p> <p>如果指定了阻止时间（xTicksToWait 不为零），则调用任务将置于“被阻止”状态以等待另一个守护程序任务在队列中腾出空间，但是在此之前阻止时间就已到期。</p>

## 重置软件计时器（示例 15）

此示例模拟手机背光的行为。背光：

- 在按下某个键时打开。
- 在特定时间段内，如果有其他键按下，将保持打开状态。
- 如果在特定时间段内没有其他键按下，背光将关闭。

使用一次性软件计时器来实现此行为。

- （模拟）背光在某个键按下时打开，并在软件计时器的回调函数中关闭。
- 每次有键按下时都会重置软件计时器。
- 因此，必须按下某个键以防止背光关闭的时间段与软件计时器的周期相等。如果软件计时器到期之前未通过按键重置，则计时器的回调函数将执行，背光关闭。

xSimulatedBacklightOn 变量保存背光状态。如果 xSimulatedBacklightOn 设置为 pdTRUE，则背光打开。如果 xSimulatedBacklightOn 设置为 pdFALSE，则背光关闭。

下面是软件计时器回调函数。

```
static void prvBacklightTimerCallback( TimerHandle_t xTimer )
```

```
{  
  
    TickType_t xTimeNow = xTaskGetTickCount();  
  
    /* The backlight timer expired. Turn the backlight off. */  
  
    xSimulatedBacklightOn = pdFALSE;  
  
    /* Print the time at which the backlight was turned off. */  
  
    vPrintStringAndNumber("Timer expired, turning backlight OFF at time\t\t", xTimeNow );  
  
}
```

使用 FreeRTOS 以允许应用程序通过事件驱动。事件驱动的设计能高效使用处理时间。仅当事件发生时才会使用该时间。不会浪费处理时间来轮询尚未发生的事件。此示例无法设计为由事件驱动，因为使用 FreeRTOS Windows 端口时处理键盘中断是不现实的。而是必须使用效率低得多的轮询方法。

下面的代码说明轮询键盘的任务。如果它是中断服务例程，则使用 xTimerResetFromISR() 代替 xTimerReset()。

```
static void vKeyHitTask( void *pvParameters )  
  
{  
  
    const TickType_t xShortDelay = pdMS_TO_TICKS( 50 );  
  
    TickType_t xTimeNow;  
  
    vPrintString( "Press a key to turn the backlight on.\r\n" );  
  
    /* Ideally an application would be event-driven, and use an interrupt to process key  
    presses. It is not practical to use keyboard interrupts when using the FreeRTOS Windows  
    port, so this task is used to poll for a key press. */  
  
    for( ;; )  
  
    {  
  
        /* Has a key been pressed? */  
  
        if( _kbhit() != 0 )  
  
        {  
  
            /* A key has been pressed. Record the time. */  
  
            xTimeNow = xTaskGetTickCount();  
  
            if( xSimulatedBacklightOn == pdFALSE )  
  
            {  
  
                /* The backlight was off, so turn it on and print the time at which it was  
                turned on. */  
  
                xSimulatedBacklightOn = pdTRUE;  
  
                vPrintStringAndNumber( "Key pressed, turning backlight ON at time\t\t",  
xTimeNow );  
  
            }  
  
        }  
  
    }  
  
}
```

```
    else
    {
        /* The backlight was already on, so print a message to say the timer is
        about to be reset and the time at which it was reset. */

        vPrintStringAndNumber("Key pressed, resetting software timer at time\t\t",
xTimeNow );
    }

    /* Reset the software timer. If the backlight was previously off, then this
call will start the timer. If the backlight was previously on, then this call will restart
the timer. A real application might read key presses in an interrupt. If this function
was an interrupt service routine, then xTimerResetFromISR() must be used instead of
xTimerReset(). */

    xTimerReset( xBacklightTimer, xShortDelay );

    /* Read and discard the key that was pressed. It is not required by this simple
example. */

    ( void ) _getch();
}

}

}
```

下面是输出的内容。

```
C:\temp>rtosdemo
Press a key to turn the backlight on.
Key pressed, turning backlight ON at time      812
Key pressed, resetting software timer at time  1813
Key pressed, resetting software timer at time  3114
Key pressed, resetting software timer at time  4015
Key pressed, resetting software timer at time  5016
Timer expired, turning backlight OFF at time   10016
```

- 当计时计数为 812 时发生第一次按键。此时，背光开启，一次性计时器启动。
- 计时计数为 1813、3114、4015 和 5016 时发生按键。所有这些按键都导致计时器在到期之前重置。
- 当计时计数为 10016 时，计时器到期。那时，背光关闭。

计时器周期为 5000 个时钟周期。上次按键后刚好 5000 个时钟周期时背光关闭，因此是计时器上次重置后 5000 个时钟周期。

# 中断管理

本节内容：

- 哪些 FreeRTOS API 函数可从中断服务例程中使用。
- 用于将中断处理委托给任务的方法。
- 如何创建和使用二进制信号灯和计数信号灯。
- 二进制信号灯和计数信号灯之间的区别。
- 如何使用队列将数据传入和传出中断服务例程。
- 部分 FreeRTOS 端口可使用的中断嵌套模型。

对源于环境的事件，嵌入式实时系统必须采取操作进行响应。例如，到达以太网周边的数据包（事件）可能需要传递到 TCP/IP 堆栈进行处理（操作）。重要系统必须处理来自多个源的事件，所有这些事件都具有不同的处理开销和响应时间要求。选择事件处理实现策略时，请考虑以下问题：

1. 应该如何检测事件？通常会使用中断，但也可以轮询输入。
2. 如果使用中断，中断服务例程内应执行多少处理，其外应执行多少处理？建议尽量缩短每个中断服务例程。
3. 如何将事件通知给主（非 ISR）代码，这些代码应采用什么结构才最适合处理潜在异步事件？

FreeRTOS 不会将任何事件处理策略强加于应用程序设计者，但是它提供了一些功能，供您以可维护的简单方式实现所选策略。

明确任务优先级和中断优先级之间的区别非常重要：

- 任务属于软件功能，与运行 FreeRTOS 的硬件无关。任务优先级由应用程序编写者在软件中指定。软件算法（计划程序）确定哪个任务将处于“运行”状态。
- 尽管中断服务例程是以软件形式编写的，它却属于硬件功能，因为硬件控制哪个中断服务例程将运行，以及何时运行。任务仅当没有中断服务例程在运行时才会运行，因此，优先级最低的中断也会中断优先级最高的任务。任务无法优先于中断服务例程运行。

运行 FreeRTOS 的所有架构都能够处理中断，但有关中断入口和中断优先级分配的详细信息因架构而异。

## 中断安全 API

在中断服务例程中，常常有必要使用 FreeRTOS API 函数提供的功能，但是，很多 FreeRTOS API 函数执行的操作在中断服务例程内是无效的。其中最明显的是，将调用 API 函数的任务置于“被阻止”状态。如果在中断服务例程中调用 API 函数，则不是从任务调用，因此不会有可置于“被阻止”状态的调用任务。FreeRTOS 通过为部分 API 函数提供两个版本来解决这个问题：一个版本用于从任务调用，一个版本用于从中断服务例程调用。用于从 ISR 调用的函数的名称后附有“FromISR”。

注意：在 ISR 中，请勿调用名称中不包含“FromISR”的 FreeRTOS API 函数。

## 使用单独的中断安全 API 的优点

单独用于中断的 API 让任务和代码更高效，中断入口也更简单。考虑另一种解决方案，即为每个 API 函数提供从任务和 ISR 都可以调用的单个版本。如果相同版本的 API 函数可以从任务和中断服务例程调用，则：

- API 函数需要额外的逻辑来确定是从任务还是从 ISR 调用。这样的额外逻辑会为函数引入新的路径，使函数变得更长、更复杂，也更难以测试。
- 从任务调用函数时，会弃用某些 API 函数参数，而从 ISR 调用函数时，又会弃用另一些函数参数。
- 每个 FreeRTOS 端口都需要提供一种机制来确定执行上下文（任务还是 ISR）。
- 对于难以确定执行上下文（任务还是 ISR）的架构，将需要额外的、浪费资源的、更为复杂的非标准中断入口代码，才能由软件提供执行上下文。

## 使用单独的中断安全 API 的缺点

为某些 API 函数提供两个版本，任务和 ISR 都会更高效，但是这也会带来一个新的问题。有时，需要调用不属于 FreeRTOS API 的函数，但是要从任务和 ISR 中使用 FreeRTOS API。

通常，仅当集成第三方代码时，这才会是个问题，因为只有这种情况下，软件设计才不由应用程序编写者控制。您可以使用下面的方法之一：

1. 将中断处理委托给任务，使 API 函数仅从任务上下文调用。
2. 如果使用支持中断嵌套的 FreeRTOS 端口，请使用以“FromISR”结尾的 API 函数版本，因为这种版本可从任务和 ISR 中调用。（反之则不行。不以“FromISR”结尾的 API 函数不能从 ISR 调用）。
3. 第三方代码通常包含一个 RTOS 抽象层，实现这个抽象层可测试所调用函数的上下文（任务或中断），然后调用适合该上下文的 API 函数。

## xHigherPriorityTaskWoken 参数

如果中断执行上下文切换，则中断退出时正在运行的任务可能与中断进入时正在运行的任务不同。中断将中断一个任务，但返回到不同的任务。

某些 FreeRTOS API 函数可以使任务从“被阻止”状态变为“准备就绪”状态。xQueueSendToBack() 等函数就是这样，如果主体队列中有处于“被阻止”状态的任务在等待数据变为可用，则会取消阻止该任务。

如果 FreeRTOS API 函数取消阻止的任务的优先级高于处于“运行”状态的任务的优先级，则根据 FreeRTOS 调度策略，会切换到较高优先级的任务。实际何时切换到较高优先级的任务取决于调用 API 函数的上下文。

- 如果从任务调用 API 函数：

如果 FreeRTOSConfig.h 中的 configUSE\_PREEMPTION 设置为 1，则会在 API 函数内自动切换到较高优先级的任务，因此是在 API 函数退出之前。在 software\_tier\_management 中可以看到这一点，其中，写入计时器命令队列导致在写入该命令队列的函数退出之前切换到 RTOS 守护程序任务。

- 如果从中断调用 API 函数：

中断内部不会自动切换到较高优先级任务，而是会设置一个变量，以通知应用程序编写者，应该执行上下文切换。中断安全 API 函数（以“FromISR”结尾）有一个名为 pxHigherPriorityTaskWoken 的指针参数用于实现这一目的。

如果应执行上下文切换，则中断安全 API 函数将 \*pxHigherPriorityTaskWoken 设置为 pdTRUE。要能够检测是否发生了切换，pxHigherPriorityTaskWoken 指向的变量必须在首次使用之前初始化为 pdFALSE。

如果应用程序编写者选择不从 ISR 中请求上下文切换，则较高优先级的任务将仍然处于“准备就绪”状态，直至下一次计划程序运行（在最差的情况下，将在下一次计时中断期间）。

FreeRTOS API 函数只能将 \*pxHigherPriorityTaskWoken 设置为 pdTRUE。如果 ISR 调用多个 FreeRTOS API 函数，则在每个 API 函数调用中都可将该变量作为 pxHigherPriorityTaskWoken 参数传递。该变量只有在首次使用之前才需要初始化为 pdFALSE。

中断安全版本的 API 函数内部不会自动切换上下文的原因有几个：

1. 避免不必要的上下文切换

在任务必须执行任何处理之前，一个中断可能执行多次。例如，考虑这样一种情况：一个任务处理中断驱动的 UART 接收的字符串。每次接收到一个字符，UART ISR 都切换到该任务，就会很浪费，因为该任务只有在收到完整的字符串之后才需要执行处理。

2. 控制执行顺序

中断可能在不可预测的时间零星发生。FreeRTOS 的专家用户可能需要在其应用程序的某些特定点暂时避免不可预测地切换到另一个任务。这可以通过使用 FreeRTOS 计划程序锁定机制来实现。

3. 可移植性

这是可用于所有 FreeRTOS 端口的最简单机制。

4. 效率

面向较小处理器架构的端口仅允许在 ISR 的最后请求上下文切换。消除这一限制需要额外的更复杂的代码。它还允许在同一个 ISR 中多次调用某个 FreeRTOS API 函数，而不会在这个 ISR 中生成多次上下文切换请求。

5. 在 RTOS 计时中断中执行

您可以在 RTOS 计时中断中添加应用程序代码。在计时中断内尝试切换上下文的结果取决于所使用的 FreeRTOS 端口。最多会导致对计划程序不必要的调用。

是否使用 pxHigherPriorityTaskWoken 参数是可选的。如果 pxHigherPriorityTaskWoken 不是必需的，则将其设置为 NULL。

## portYIELD\_FROM\_ISR() 和 portEND\_SWITCHING\_ISR() 宏

taskYIELD() 是一个宏，可以在任务中调用来请求上下文切换。portYIELD\_FROM\_ISR() 和 portEND\_SWITCHING\_ISR() 都是 taskYIELD() 的中断安全版本。portYIELD\_FROM\_ISR() 和 portEND\_SWITCHING\_ISR() 的使用方式相同，功能相同。过去，portEND\_SWITCHING\_ISR 这个名称用于要求中断处理程序使用程序集代码包装程序的 FreeRTOS 端口。portYIELD\_FROM\_ISR() 这个名称用于允许用 C 语言编写整个中断处理程序的 FreeRTOS 端口。某些 FreeRTOS 端口仅提供两个宏中的一个。较新的 FreeRTOS 端口同时提供这两个宏。

```
portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
```

```
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

在中断安全 API 函数外传递的 xHigherPriorityTaskWoken 参数可在调用 portYIELD\_FROM\_ISR() 时直接用作参数。

如果 portYIELD\_FROM\_ISR() xHigherPriorityTaskWoken 参数是 pdFALSE ( 零 )，则不会请求上下文切换，宏也无效。如果 portYIELD\_FROM\_ISR() xHigherPriorityTaskWoken 参数不是 pdFALSE，则会请求上下文切换，这时，处于“运行”状态的任务可能已不是原来那一个任务。中断始终返回到处于“运行”状态的任务，即使在执行中断时，处于“运行”状态的已不是原来的任务也是如此。

大多数 FreeRTOS 端口都允许在 ISR 中的任意位置调用 portYIELD\_FROM\_ISR()。某些 FreeRTOS 端口（其中大多面向较小的架构）允许仅在 ISR 的最后调用 portYIELD\_FROM\_ISR()。

## 委托中断处理

最佳实践尽可能缩短 ISR，因为：

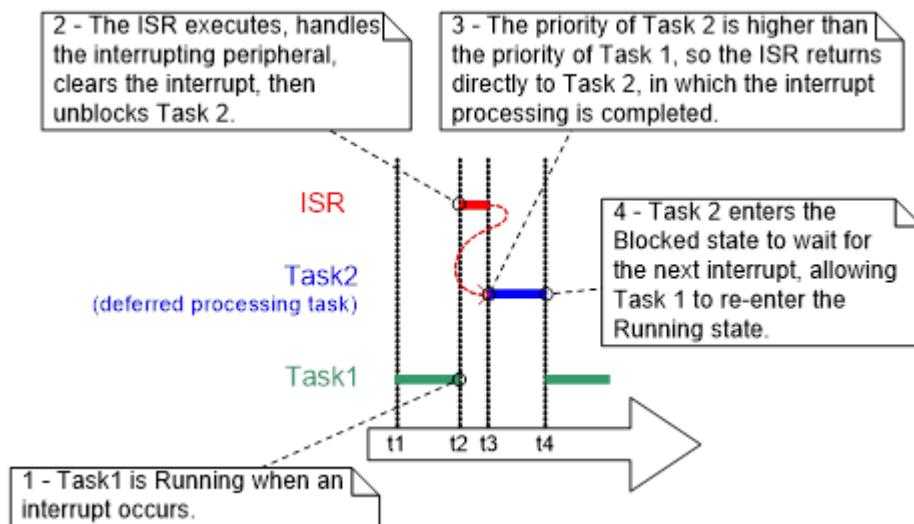
- 即使为任务分配非常高的优先级，它们也只在硬件不为中断服务时才会运行。
- ISR 可能打断（添加抖动到）任务的开头和执行时间。
- 根据运行 FreeRTOS 的架构，在 ISR 执行时，可能无法接受任何新中断，或至少一部分新中断。
- 应用程序编写者必须考虑对于资源（如任务和 ISR 同时访问的变量、外围设备和内存缓冲区）的后果和保护。
- 某些 FreeRTOS 端口允许中断嵌套，但中断嵌套会增加复杂性，降低可预测性。中断越短，嵌套的可能性越小。

中断服务例程必须记录中断的原因并清除中断。中断需要的任何其他处理通常都可以在任务中执行，这样中断服务例程可以尽快退出。这称为委托中断处理，因为中断需要的处理从 ISR 委托给任务进行。

如果将中断处理委托给任务，应用程序编写者还可以在应用程序中将处理操作安排在其他任务之前执行，并且可以使用所有 FreeRTOS API 函数。

如果受委托执行中断处理的任务的优先级高于其他所有任务的优先级，则将立即执行处理，就像处理在 ISR 本身中执行一样。

在下图的方案中，Task 1 是常规应用程序任务，Task 2 是受委托执行中断处理的任务。



在此图中，中断处理在时间 t2 开始，实际上在时间 t4 结束，但是只有 t2 和 t3 之间的时间段由 ISR 使用。如果不使用委托中断处理，则时间 t2 和 t4 之间的整个时间段都由 ISR 使用。

何时最适合在 ISR 中执行中断所需的所有处理，何时最适合将部分处理委托给任务，都没有绝对的规则。将处理委托给任务在以下情况下最有用：

- 中断所需的处理很重要。例如，如果中断只是存储模数转换的结果，则最适合在 ISR 中执行，但是，如果转换结果还必须通过软件滤波器传递，则更适合在任务中执行滤波器。
- 便于中断处理执行无法在 ISR 中执行的操作，如写入控制台或分配内存。
- 中断处理不可确定（即，事先不知道处理需要多长时间）。

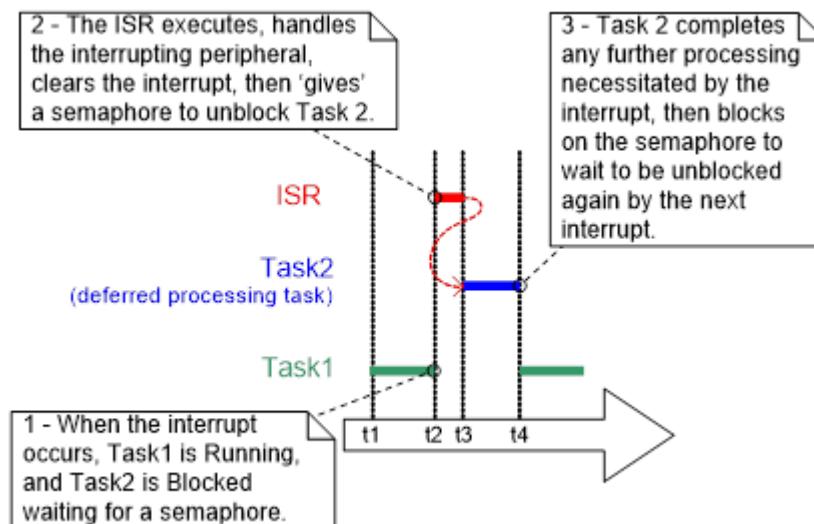
下面几节介绍和说明可用于实现委托中断处理的 FreeRTOS 功能。

## 用于同步的二进制信号灯

使用二进制信号灯 API 的中断安全版本可以在每次发生某个特定中断时取消阻止某个任务，从而有效地使该任务与中断同步。这样，中断事件处理的大部分操作都在同步后的任务中执行，仅在 ISR 中直接保留很少、可以很快执行的部分。可使用二进制信号灯将中断处理委托给任务。使用“直接到任务”通知从中断取消阻止某个任务，是比使用二进制信号灯更高效的方法。有关“直接到任务”通知的更多信息，请参阅[任务通知 \(p. 178\)](#)。

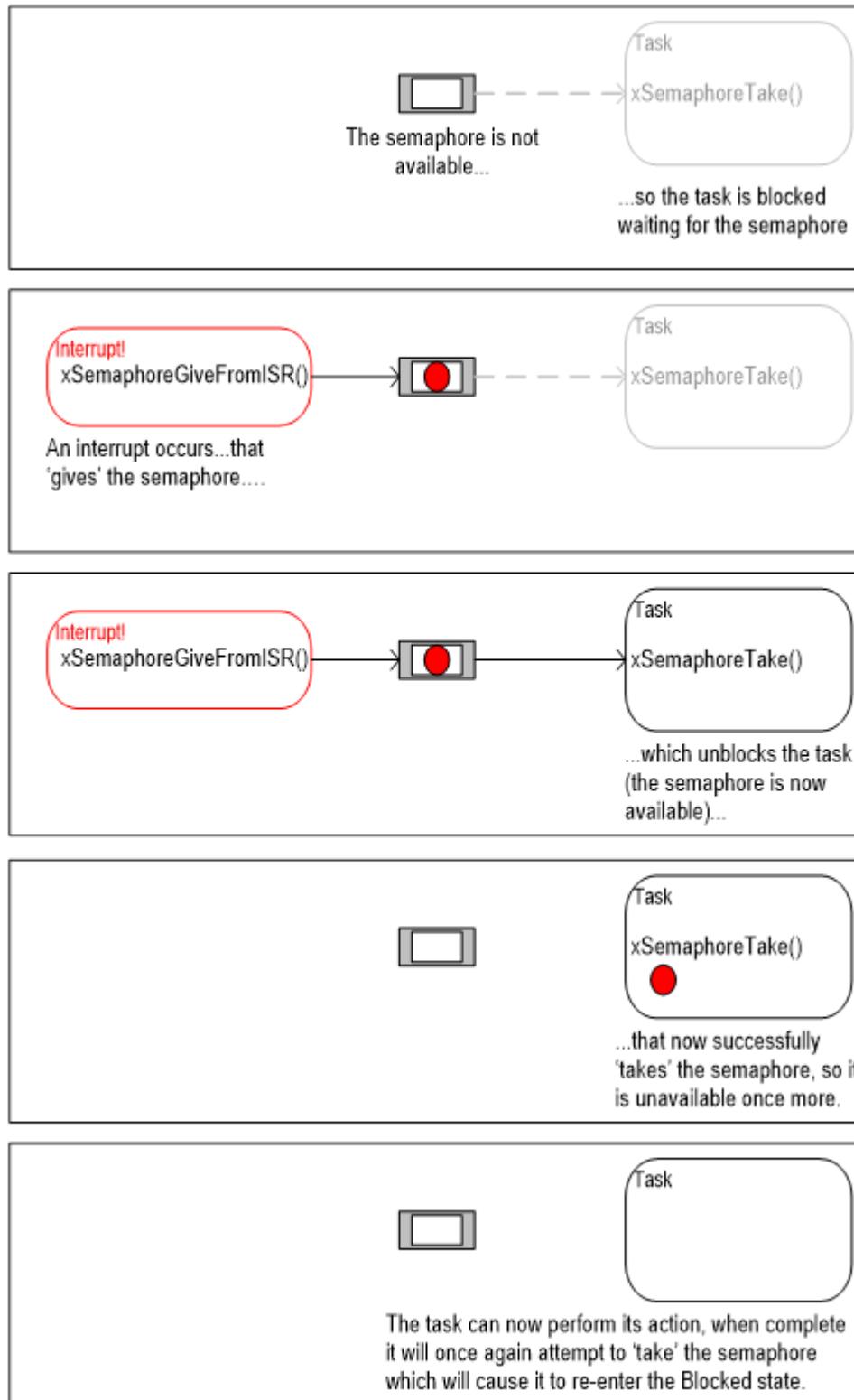
如果中断处理对时间的要求特别苛刻，可以对受委托执行中断处理的任务设置优先级，以确保该任务始终优先于系统中的其他任务。然后可以实现 ISR 以包含 portYIELD\_FROM\_ISR() 调用，从而确保 ISR 直接返回到受委托执行中断处理的任务。这可以确保整个事件处理能够在时间上连续执行（而不中断），就像整个过程都是在 ISR 本身中实现一样。

下图使用上图中所示的方案，但其中的文本已更新为描述如何通过使用信号灯来控制受委托执行处理的任务。



受委托执行处理的任务对信号灯使用阻止性“Take”（获取）调用，从而进入“被阻止”状态以等待事件发生。当事件发生时，ISR 对该信号灯使用“Give”（释放）操作，从而取消阻止任务，以便进行所需的事件处理。

获取信号灯 和 释放信号灯 是含义不同的概念，取决于具体方案。在此中断同步方案中，二进制信号灯可以在概念上视为长度为一的队列。这个队列在任何时候最多都只能包含一个项目，因此始终不是空的就是满的（二进制）。通过调用 xSemaphoreTake()，受委托执行中断处理的任务将使用阻止时间有效地尝试从队列中进行读取，如果队列为空，则任务进入“被阻止”状态。如果事件发生，ISR 使用 xSemaphoreGiveFromISR() 函数将令牌（信号灯）放入队列，使队列变满。这会导致任务退出“被阻止”状态并移除令牌，又一次使队列变空。如果任务完成处理，它再次尝试从队列中读取，会发现队列为空，因而重新进入“被阻止”状态，等待下一个事件发生。下面说明这一顺序。



此图说明释放 信号灯（即使最初并没有获取 信号灯）的中断，以及获取 信号灯但未释放 信号灯的任务。这就是将这种情况描述为在概念上类似于写入队列和从队列读取的原因。这常常会造成困扰，因为不符合其他

信号灯使用情况的规则，在那些情况下，任务获取信号灯后必须要释放，如资源管理 (p. 141) 中所述的情况。

## xSemaphoreCreateBinary() API 函数

FreeRTOS V9.0.0 还包含 xSemaphoreCreateBinaryStatic() 函数，该函数分配在编译时静态创建二进制信号灯所需的内存。各种类型的 FreeRTOS 信号灯的句柄存储在类型为 SemaphoreHandle\_t 的变量中。

信号灯必须先创建，然后才能使用。要创建二进制信号灯，请使用 xSemaphoreCreateBinary() API 函数。

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

下表列出了 xSemaphoreCreateBinary() 返回值。

参数名称	描述
返回的值	如果返回 NULL，则表示无法创建信号灯，因为没有足够的堆内存可供 FreeRTOS 分配信号灯数据结构。 如果返回非 NULL 值，表示信号灯已成功创建。返回的值应存储为所创建信号灯的句柄。

## xSemaphoreTake() API 函数

获取信号灯意味着获得或接收信号灯。只有信号灯可用时才可以获取。

所有类型的 FreeRTOS 信号灯（递归互斥除外）都可以使用 xSemaphoreTake() 函数获取。xSemaphoreTake() 不能从中断服务例程中使用。

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

下表列出了 xSemaphoreTake() 参数和返回值。

参数名称/返回的值	描述
xSemaphore	获取的信号灯。 信号灯由 SemaphoreHandle_t 类型的变量引用。它必须显式创建后才能使用。
xTicksToWait	在信号灯还不可用的情况下，任务应保持“被阻止”状态的最长等待时间。 如果 xTicksToWait 为零，则在信号灯不可用的情况下，xSemaphoreTake() 将立即返回。 阻止时间以计时周期指定，因此它表示的绝对时间取决于计时频率。宏 pdMS_TO_TICKS() 可用于将毫秒为单位指定的时间转换为计时周期数。

	如果 FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 设置为 1，将 xTicksToWait 设置为 portMAX_DELAY 将导致任务无限期等待（无超时）。
返回的值	<p>可能的返回值有两个：</p> <p>1. pdPASS</p> <p>仅当 xSemaphoreTake() 调用成功获取信号灯时，才会返回 pdPASS。</p> <p>如果指定了阻止时间（xTicksToWait 不为零），则可能是信号灯不是立即可用时，发出调用的任务置于“被阻止”状态以等待信号灯，但是在阻止时间到期之前信号灯已变为可用。</p> <p>2. pdFALSE</p> <p>信号灯不可用。</p> <p>如果指定了阻止时间（xTicksToWait 不为零），则发出调用的任务会置于“被阻止”状态以等待信号灯变为可用，但是阻止时间在信号灯可用之前就已到期。</p>

## xSemaphoreGiveFromISR() API 函数

二进制信号灯和计数信号灯可以使用 xSemaphoreGiveFromISR() 函数释放。

xSemaphoreGiveFromISR() 是中断安全版本的 xSemaphoreGive()，因此它有 pxHigherPriorityTaskWoken 参数。

```
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore, BaseType_t *pxHigherPriorityTaskWoken );
```

下表列出了 xSemaphoreGiveFromISR() 参数和返回值。

**xSemaphore**

释放的信号灯。信号灯由 SemaphoreHandle\_t 类型的变量引用，必须在使用之前显式创建。

**pxHigherPriorityTaskWoken**

单个信号灯可能会使一个或多个任务被阻止以等待信号灯变为可用。调用 xSemaphoreGiveFromISR() 可以使信号灯变为可用，并因此使得一个正在等待信号灯的任务离开“被阻止”状态。如果调用 xSemaphoreGiveFromISR() 会导致任务离开“被阻止”状态，并且取消阻止的任务的优先级高于当前正在执行的任务（被中断的任务），那么 xSemaphoreGiveFromISR() 会在内部将 \*pxHigherPriorityTaskWoken 设置为 pdTRUE。如果 xSemaphoreGiveFromISR() 将此值设置为 pdTRUE，则通常情况下，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优先级最高的“准备就绪”状态的任务。

可能的返回值有两个：

**pdPASS**

仅当调用 xSemaphoreGiveFromISR() 成功完成时返回。

pdFAIL

信号灯已可用时不可释放，这种情况下 xSemaphoreGiveFromISR() 将返回 pdFAIL，

## 使用二进制信号灯使任务与中断同步 (示例 16)

此示例使用二进制信号灯从中断服务例程中取消阻止任务，从而有效地使任务与中断同步。

将使用一个简单的定期任务，每 500 毫秒生成一次软件中断。软件中断是为了方便而使用的，因为在某些目标环境中，挂接实际的中断很复杂。

下面的代码说明定期任务的实现。该任务在中断生成前后都输出一个字符串。通过输出可以看到执行顺序。

```
/* The number of the software interrupt used in this example. The code shown is from the
Windows project, where numbers 0 to 2 are used by the FreeRTOS Windows port itself, so 3
is the first number available to the application. */

#define mainINTERRUPT_NUMBER 3

static void vPeriodicTask( void *pvParameters )

{
    const TickType_t xDelay500ms = pdMS_TO_TICKS( 500UL );

    /* As per most tasks, this task is implemented within an infinite loop.*/
    for( ;; )

    {
        /* Block until it is time to generate the software interrupt again. */
        vTaskDelay( xDelay500ms );

        /* Generate the interrupt, printing a message both before and after the interrupt
has been generated, so the sequence of execution is evident from the output. The syntax
used to generate a software interrupt is dependent on the FreeRTOS port being used.
The syntax used below can only be used with the FreeRTOS Windows port, in which such
interrupts are only simulated.*/
        vPrintString( "Periodic task - About to generate an interrupt.\r\n" );
        vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );
        vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n\r\n" );
    }
}
```

下面的代码说明受委托执行中断处理的任务的实现，该任务通过使用二进制信号灯与软件中断同步。同样，该任务的每次迭代都输出一个字符串。通过输出可以看到任务和中断执行的顺序。

尽管作为软件生成中断的示例，这些代码已经足够，对于硬件外围设备生成中断的情况，还是不够的。必须更改这段代码的结构，使之适合硬件生成的中断。

```
static void vHandlerTask( void *pvParameters )
```

```
{  
    /* As per most tasks, this task is implemented within an infinite loop.*/  
  
    for( ;; )  
  
    {  
  
        /* Use the semaphore to wait for the event. The semaphore was created before  
        the scheduler was started, so before this task ran for the first time. The task blocks  
        indefinitely, meaning this function call will only return once the semaphore has been  
        successfully obtained so there is no need to check the value returned by xSemaphoreTake().  
        */  
  
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );  
  
        /* To get here the event must have occurred. Process the event (in this case, just  
        print out a message). */  
  
        vPrintString( "Handler task - Processing event.\r\n" );  
  
    }  
}
```

下面的代码说明 ISR。其中，除了释放信号灯，从而取消阻止受委托执行中断处理的任务之外，没有其他太多操作。

使用了 xHigherPriorityTaskWoken 变量。在调用 xSemaphoreGiveFromISR() 之前，它设置为 pdFALSE，然后在调用 portYIELD\_FROM\_ISR() 时用作参数。如果 xHigherPriorityTaskWoken 等于 pdTRUE，则在 portYIELD\_FROM\_ISR() 宏内部将请求上下文切换。

ISR 的原型和调用来强制进行上下文切换的宏对 FreeRTOS Windows 端口是正确的，但是对于其他 FreeRTOS 端口可能不是这样。要查找您所使用的端口所需的语法，请参阅 FreeRTOS.org 网站上特定于端口的文档页面，以及在 FreeRTOS 下载中提供的示例。

与运行 FreeRTOS 的大多数架构不同，FreeRTOS Windows 端口需要 ISR 返回一个值。随 Windows 端口提供的 portYIELD\_FROM\_ISR() 宏的实现包括返回语句，因此这段代码并不显式显示所返回的值。

```
static uint32_t ulExampleInterruptHandler( void )  
{  
  
    BaseType_t xHigherPriorityTaskWoken;  
  
    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it  
    will get set to pdTRUE inside the interrupt-safe API function if a context switch is  
    required. */  
  
    xHigherPriorityTaskWoken = pdFALSE;  
  
    /* Give the semaphore to unblock the task, passing in the address of  
    xHigherPriorityTaskWoken as the interrupt-safe API function's pxHigherPriorityTaskWoken  
    parameter. */  
  
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );  
  
    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If  
    xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR(), then calling  
    portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still  
    pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS  
    ports, the Windows port requires the ISR to return a value. The return statement is inside  
    the Windows version of portYIELD_FROM_ISR(). */  
}
```

```
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );  
}
```

main() 函数创建二进制信号灯和任务，安装中断处理程序，并启动计划程序。下面的代码说明相关实现。

调用来安装中断处理程序的函数的语法是特定于 FreeRTOS Windows 端口的。对于其他 FreeRTOS 端口，语法可能有所不同。要查找您所使用的端口所需的语法，请参阅 FreeRTOS.org 网站上特定于端口的文档，以及在 FreeRTOS 下载中提供的示例。

```
int main( void )  
{  
  
    /* Before a semaphore is used, it must be explicitly created. In this example, a binary  
    semaphore is created. */  
  
    xBinarySemaphore = xSemaphoreCreateBinary();  
  
    /* Check that the semaphore was created successfully. */  
  
    if( xBinarySemaphore != NULL )  
    {  
  
        /* Create the 'handler' task, which is the task to which interrupt processing is  
        deferred. This is the task that will be synchronized with the interrupt. The handler task  
        is created with a high priority to ensure it runs immediately after the interrupt exits.  
        In this case, a priority of 3 is chosen. */  
  
        xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );  
  
        /* Create the task that will periodically generate a software interrupt. This is  
        created with a priority below the handler task to ensure it will get preempted each time  
        the handler task exits the Blocked state. */  
  
        xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, NULL );  
  
        /* Install the handler for the software interrupt. The syntax required to do this  
        is depends on the FreeRTOS port being used. The syntax shown here can only be used with  
        the FreeRTOS Windows port, where such interrupts are only simulated. */  
  
        vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );  
  
        /* Start the scheduler so the created tasks start executing. */  
  
        vTaskStartScheduler();  
    }  
  
    /* As normal, the following line should never be reached. */  
  
    for( ;; );  
}
```

这段代码会产生以下输出。正如预期的那样，只要生成中断，vHandlerTask() 就进入“运行”状态，因此任务输出将定期任务的输出分隔开了。

```
cmd C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Periodic task - Interrupt generated.

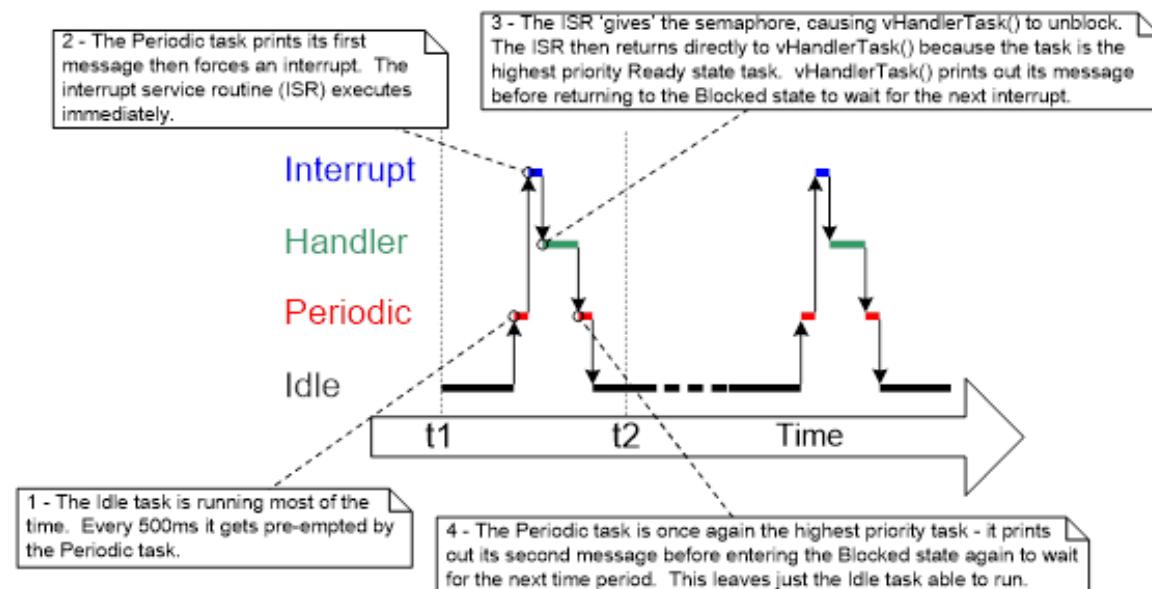
Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.
```

下面是执行的顺序。



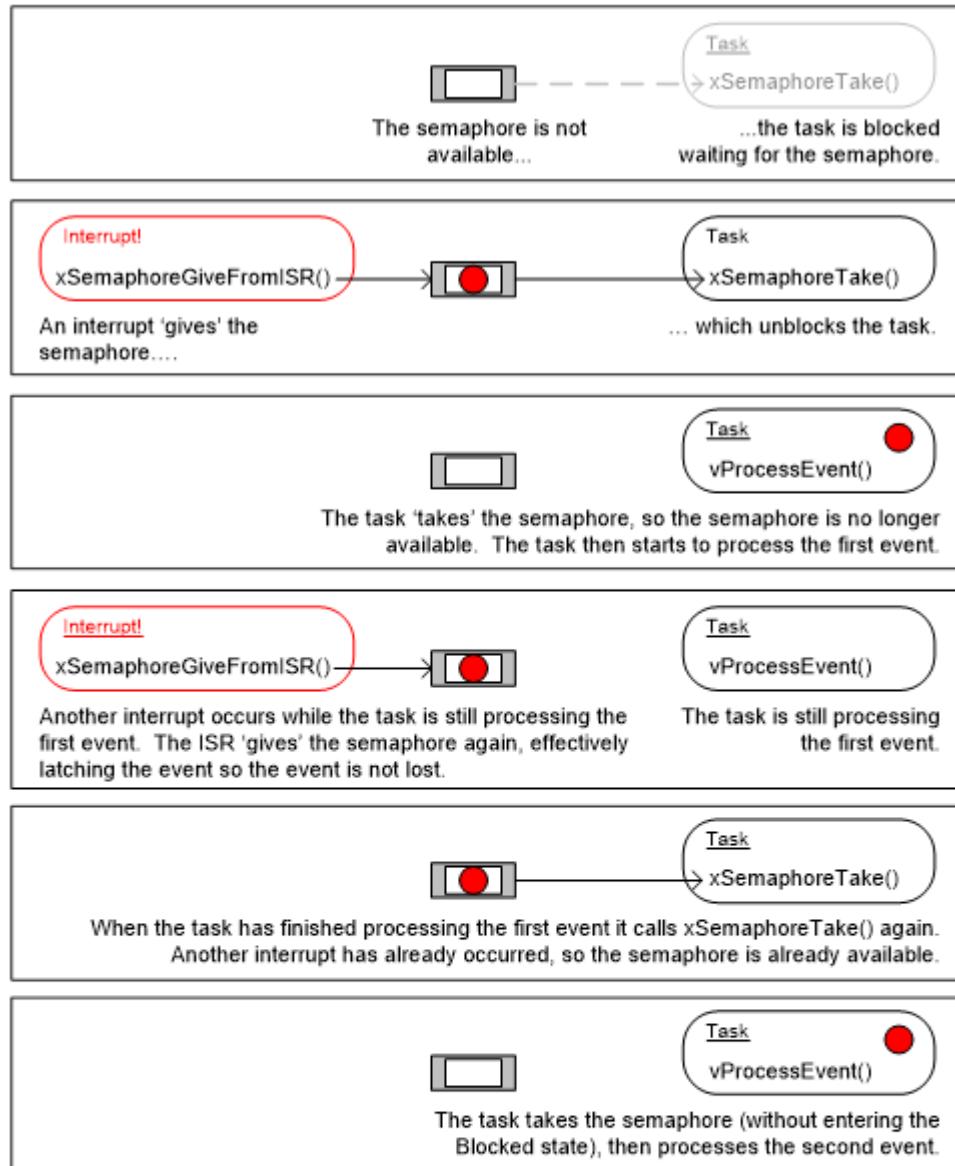
## 改进示例 16 中所用任务的实现

示例 16 使用二进制信号灯使任务与中断同步。下面是执行顺序：

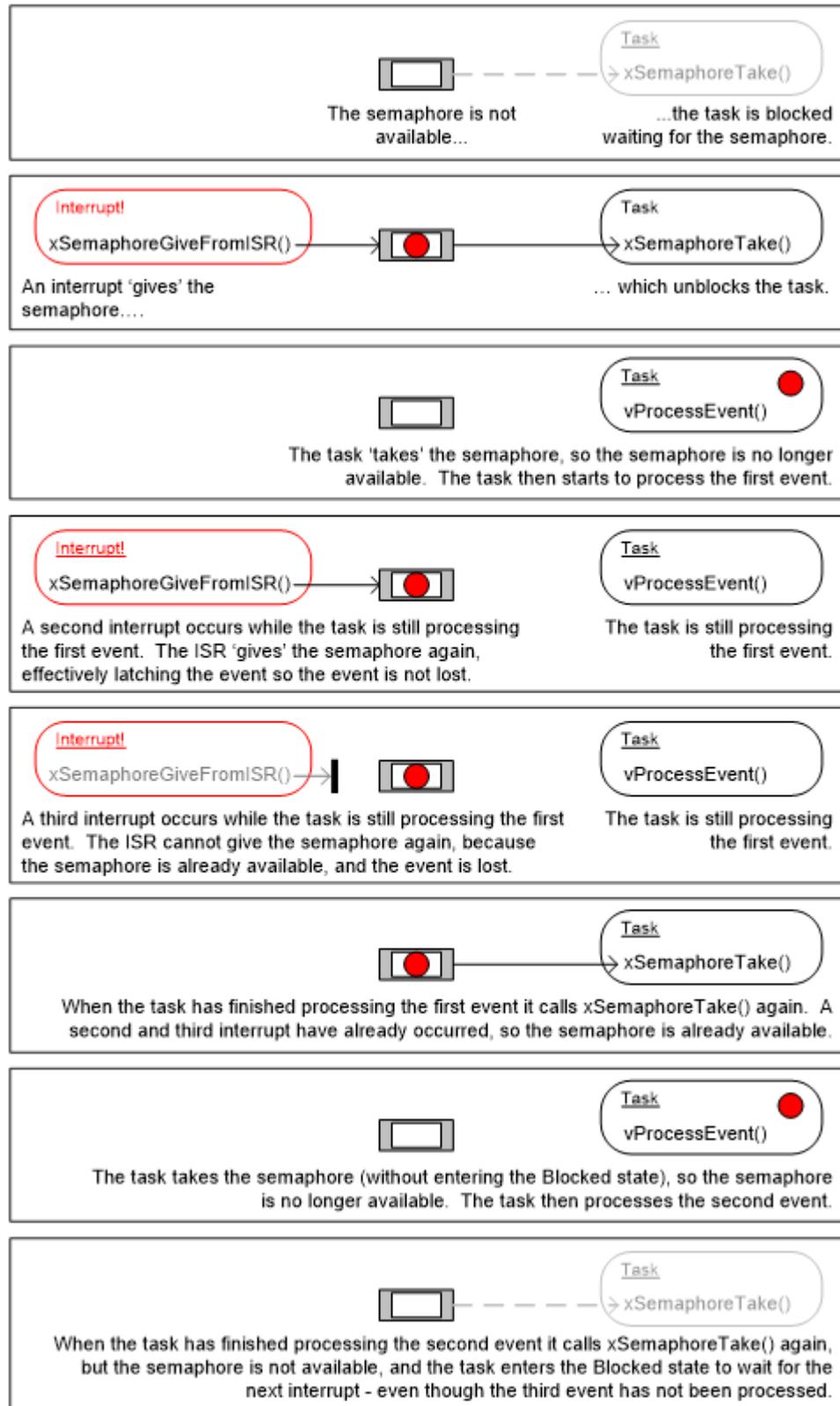
1. 发生中断。
2. ISR 执行，并释放信号灯，以取消阻止任务。
3. 任务在 ISR 后紧接着执行，并获取信号灯。
4. 任务处理事件，然后再次尝试获取信号灯，因为信号灯尚不可用（另一个中断还未发生），所以进入“被阻止”状态。

示例 16 中使用的任务的结构只适用于中断以相对较低的频率发生的情况。考虑一下，如果第二个以及第三个中断已经发生，而任务尚未完成第一个中断的处理，会发生什么情况。

当第二个 ISR 执行时，信号灯为空，因此 ISR 释放信号灯，任务在完成第一个事件的处理后立即开始处理第二个事件。下面说明这种情况。



当第三个 ISR 执行时，信号灯已经可用，使 ISR 无法再次释放信号灯，因此任务并不知道已发生第三个事件。下面说明这种情况。



受委托执行中断处理的任务 `static void vHandlerTask( void *pvParameters )` 是结构化的，这样，它在两次 `xSemaphoreTake()` 调用之间仅处理一个事件。这对示例 16 是足够的，因为生成事件的中断由软件触发，发生的时间可预测。在实际的应用程序中，中断由硬件生成，发生的时间不可预测。因此，要将错过中断的几率降至最低，受委托执行中断处理的任务必须是结构化的，以便它在两次 `xSemaphoreTake()` 调用之间处理所有可用的事件。下面的代码说明 UART 的委托中断处理程序如何实现结构化。它假定 UART 在每次接收到字符时都会接收到中断，并将收到的字符放入硬件 FIFO（一种硬件缓冲区）。

示例 16 中使用的受委托执行中断处理的任务还有一处不足。它在调用 `xSemaphoreTake()` 时未使用超时。该任务将 `portMAX_DELAY` 作为 `xSemaphoreTake()` `xTicksToWait` 参数传递，这将导致任务无限期（无超时）等待信号灯变为可用。无限超时通常在示例代码中使用，因为这样可以简化示例的结构，更便于理解。但是，在实际应用程序中，无限超时通常不是好的做法，因为这样很难从错误中恢复。考虑以下情况：任务在等待中断释放信号灯，但硬件处于错误状态无法生成中断。

- 如果任务的等待没有超时，它不会知道错误状态，将一直等待下去。
- 如果任务的等待有超时，那么超时到期时 `xSemaphoreTake()` 将返回 `pdFAIL`，然后任务可以在下次执行时检测和清除该错误。下面也说明了这种方案。

```
static void vUARTReceiveHandlerTask( void *pvParameters )
{
    /* xMaxExpectedBlockTime holds the maximum time expected between two interrupts. */
    const TickType_t xMaxExpectedBlockTime = pdMS_TO_TICKS( 500 );

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* The semaphore is given by the UART's receive (Rx) interrupt. Wait a maximum of
         * xMaxExpectedBlockTime ticks for the next interrupt. */
        if( xSemaphoreTake( xBinarySemaphore, xMaxExpectedBlockTime ) == pdPASS )
        {
            /* The semaphore was obtained. Process ALL pending Rx events before calling
             * xSemaphoreTake() again. Each Rx event will have placed a character in the UART's receive
             * FIFO, and UART_RxCount() is assumed to return the number of characters in the FIFO. */
            while( UART_RxCount() > 0 )

            {
                /* UART_ProcessNextRxEvent() is assumed to process one Rx character,
                 * reducing the number of characters in the FIFO by 1. */

                UART_ProcessNextRxEvent();
            }

            /* No more Rx events are pending (there are no more characters in the FIFO),
             * so loop back and call xSemaphoreTake() to wait for the next interrupt. Any interrupts
             * occurring between this point in the code and the call to xSemaphoreTake() will be latched
             * in the semaphore, so will not be lost. */
        }
        else
    }
```

```
{\n    /* An event was not received within the expected time. Check for and, if\n    necessary, clear any error conditions in the UART that might be preventing the UART from\n    generating any more interrupts. */\n\n    UART_ClearErrors();\n\n}\n}\n}
```

## 计数信号灯

正如二进制信号灯可视为长度为一的队列，计数信号灯可视为长度大于一的队列。任务对存储在队列中的数据不感兴趣，只对队列中项目的数量感兴趣。要使计数信号灯变为可用，请在 FreeRTOSConfig.h 中将 configUSE\_COUNTING\_SEMAPHORES 设置为 1。

每次释放计数信号灯时，都会使用其队列中的另一个位置。队列中项目的数量是信号灯的“计数”值。

计数信号灯通常用于：

### 1. 对事件计数

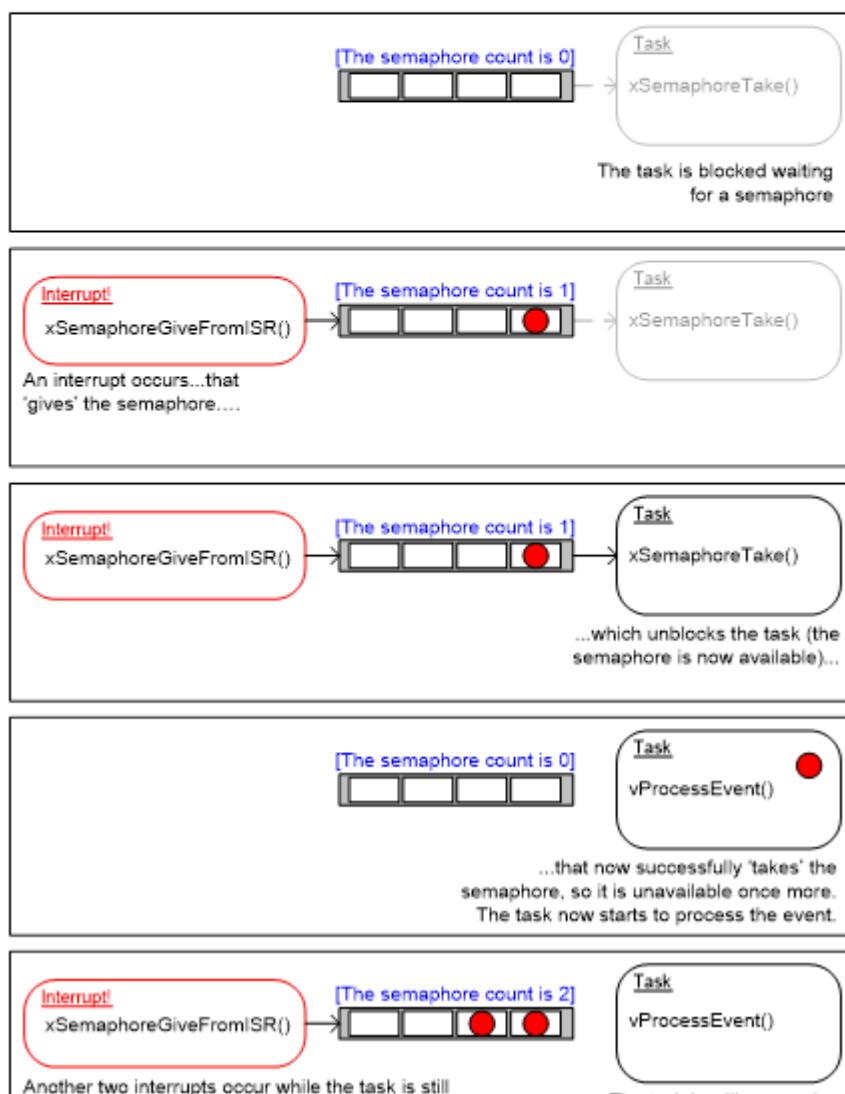
在这种情况下，事件处理程序在每次发生事件时释放信号灯，使信号灯的计数值随每一次释放操作而递增。任务在每次处理事件时都会获取一个信号灯，信号灯的计数值随每一次获取而递减。计数值是已发生的事件数量和已处理的事件数量之间的差值。下图说明这种机制。

用于对事件计数的计数信号灯在创建时的初始计数值为零。

### 2. 资源管理

在这种情况下，计数值表示可用资源的数量。要获取对资源的控制，任务必须首先获取信号灯，并递减信号灯的计数值。计数值达到零表示没有空闲资源。当任务完成资源使用时，会释放信号灯，从而递增信号灯的计数值。

创建用于管理资源的计数信号灯，令其初始计数值等于可用资源的数量。



## xSemaphoreCreateCounting() API 函数

FreeRTOS V9.0.0 还包含 xSemaphoreCreateCountingStatic() 函数，该函数用于分配在编译时静态创建计数信号灯所需的内存。所有不同类型的 FreeRTOS 信号灯的句柄都存储在类型为 SemaphoreHandle\_t 的变量中。

信号灯必须先创建，然后才能使用。要创建计数信号灯，请使用 xSemaphoreCreateCounting() API 函数。

下表列出了 xSemaphoreCreateCounting() 参数和返回值。

参数名称/返回的值	描述
uxMaxCount	信号灯计数的最大值。继续与队列类比，uxMaxCount 值实际上是队列的长度。  当信号灯用于对事件进行计数或锁定时，uxMaxCount 是可锁定的事件的最大数量。  当信号灯用于管理对一组资源的访问权限时，uxMaxCount 应设置为可用资源的总数。
uxInitialCount	信号灯在创建之后的初始计数值。  当信号灯用于对事件计数或锁定事件时，uxInitialCount 应设置为零，因为信号灯刚创建时，还没有事件发生。  当信号灯用于管理对一组资源的访问权限时，uxInitialCount 应设置为等于 uxMaxCount，因为信号灯刚创建时，所有资源都可用。
返回的值	如果返回 NULL，则表示无法创建信号灯，因为没有足够的堆内存可供 FreeRTOS 分配信号灯数据结构。有关更多信息，请参阅 <a href="#">堆内存管理 (p. 12)</a> 。  如果返回非 NULL 值，表示信号灯已成功创建。返回的值应存储为所创建信号灯的句柄。

## 使用计数信号灯使任务与中断同步（示例 17）

示例 17 使用计数信号灯代替二进制信号灯，改进了示例 16 的实现。main() 有变化，用 xSemaphoreCreateCounting() 调用代替 xSemaphoreCreateBinary() 调用。

下面的代码说明新 API 调用。

```
/* Before a semaphore is used it must be explicitly created. In this example, a counting
semaphore is created. The semaphore is created to have a maximum count value of 10, and an
initial count value of 0. */

xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

为了模拟频繁发生的多个事件，中断服务例程更改为在每次中断时多次释放信号灯。每个事件都在信号灯计数值中锁定。

下面的代码说明修改后的中断服务例程。

```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
    will get set to pdTRUE inside the interrupt-safe API function if a context switch is
    required. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Give the semaphore multiple times. The first will unblock the deferred interrupt
    handling task. The following gives are to demonstrate that the semaphore latches the
    events to allow the task to which interrupts are deferred to process them in turn, without
    events getting lost. This simulates multiple interrupts being received by the processor,
    even though in this case the events are simulated within a single interrupt occurrence. */

    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

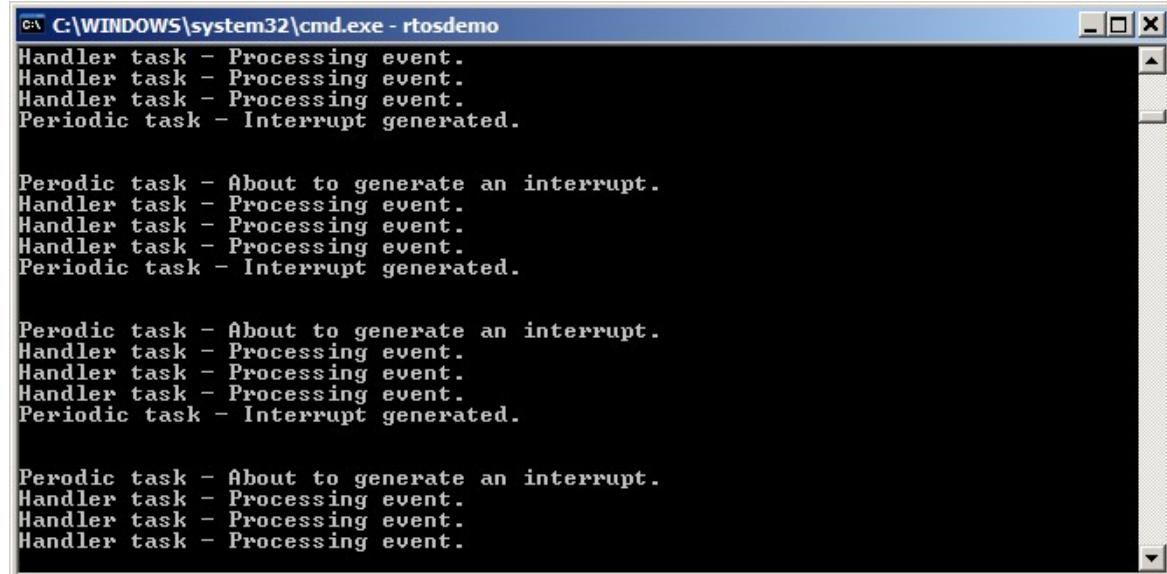
    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
    xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR(), then calling
    portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
    pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS
    ports, the Windows port requires the ISR to return a value. The return statement is inside
    the Windows version of portYIELD_FROM_ISR(). */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

所有其他函数都与在示例 16 中使用的一样。

下面是代码执行时的输出。可以看到，受委托执行中断处理的任务在每次生成中断时处理所有三个（模拟）事件。这些事件都已锁定到信号灯的计数值中，从而让任务能够依次进行处理。



```
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
```

## 将工作委托给 RTOS 守护程序任务

到现在为止，所介绍的委托中断处理示例都需要应用程序编写者为使用委托处理方法的每个中断创建一个任务。使用 `xTimerPendFunctionCallFromISR()` API 函数也可以将中断处理委托给 RTOS 守护程序任务，而无需为每个中断创建单独的任务。将中断处理委托给守护程序任务，被称为集中式委托中断处理。

守护程序任务最初调用计时器服务任务，因为它仅用于执行软件计时器回调函数。因此，`xTimerPendFunctionCall()` 在 `timers.c` 中实现，并遵循函数的命名约定，即，将实现函数的文件的名称作为函数名称的前缀，所以此函数的名称前缀为“Timer”。

`software_tier_management` 部分描述与软件计时器相关的 FreeRTOS API 函数如何向计时器命令队列的守护程序任务发送命令。`xTimerPendFunctionCall()` 和 `xTimerPendFunctionCallFromISR()` API 函数使用相同的计时器命令队列将“执行函数”命令发送到守护程序任务。然后，发送到守护程序的函数在守护程序任务的上下文中执行。

集中式委托中断处理的优点：

- 降低资源使用量  
无需为每个要委托的中断创建单独的任务。
- 简化用户模型  
该委托中断处理函数是标准的 C 语言函数。

集中式委托中断处理的缺点：

- 灵活性较低  
无法为每个受委托执行中断处理的任务单独设置优先级。每个委托中断处理函数都在守护程序任务的优先级上执行。守护程序任务的优先级由 `FreeRTOSConfig.h` 中 `configTIMER_TASK_PRIORITY` 编译时间配置常量设置。
- 确定性较低  
`xTimerPendFunctionCallFromISR()` 向计时器命令队列后端发送命令。在 `xTimerPendFunctionCallFromISR()` 将“执行函数”命令发送到队列之前，已经在计时器命令队列中的命令由守护程序任务处理。

不同的中断有不同的计时限制，因此在同一个应用程序中同时使用两个方法是很常见的。

## xTimerPendFunctionCallFromISR() API 函数

`xTimerPendFunctionCallFromISR()` 是 `xTimerPendFunctionCall()` 的中断安全版本。这两个 API 函数都允许应用程序编写者提供的函数由 RTOS 守护程序任务执行，也因此在该任务的上下文中执行。要执行的函数及其输入参数的值将发送到计时器命令队列的守护程序任务。该函数何时执行取决于守护程序任务的优先级相对于应用程序中其他任务的优先级。

下面是 `xTimerPendFunctionCallFromISR()` API 函数原型。

```
BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken );
```

通过 `xTimerPendFunctionCallFromISR()` 的 `xFunctionToPend` 参数传递的函数必须遵守的原型如下所示。

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

下表列出了 `xTimerPendFunctionCallFromISR()` 参数和返回值。

`xFunctionToPend`

指针，指向将在守护程序任务中执行的函数（实际上只是函数名称）。函数的原型必须与代码中所示的原型相同。

`pvParameter1`

将传递给由守护程序任务执行的函数作为其 `pvParameter1` 参数的值。该参数为 `void*` 类型，可用于传递任何数据类型。例如，`integer` 类型可以直接强制转换为 `void*`。或者，`void*` 可用于指向一个结构。

`ulParameter2`

将传递给由守护程序任务执行的函数作为其 `ulParameter2` 参数的值。

`pxHigherPriorityTaskWoken`

`xTimerPendFunctionCallFromISR()` 写入计时器命令队列。如果 RTOS 守护程序任务处于“被阻止”状态以等待数据在计时器命令队列中变为可用，则写入计数器命令队列将导致守护程序任务离开“被阻止”状态。如果守护程序任务的优先级高于当前正在执行的任务（被中断的任务）的优先级，则 `xTimerPendFunctionCallFromISR()` 在内部将 `*pxHigherPriorityTaskWoken` 设置为 `pdTRUE`。如果 `xTimerPendFunctionCallFromISR()` 将此值设置为 `pdTRUE`，则通常情况下，必须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守护程序任务，因为守护程序任务将是优先级最高的“准备就绪”状态任务。

可能的返回值有两个：

- `pdPASS`

如果“执行函数”命令已写入计时器命令，则返回此值。

- `pdFAIL`

如果因为计时器命令队列已满，“执行函数”命令无法写入计时器命令队列，则返回此值。有关如何设置计时器命令长度的信息，请参阅 `software_tier_management`。

## 集中式委托中断处理 (示例 18)

示例 18 提供的功能类似于示例 16，但不使用信号灯，也不创建任务来执行中断所需的处理，而是由 RTOS 守护程序任务来执行处理。

下面的代码说明示例 18 使用的中断服务例程。它调用 `xTimerPendFunctionCallFromISR()` 将指向名为 `vDeferredHandlingFunction()` 的函数的指针传递给守护程序任务。委托中断处理由 `vDeferredHandlingFunction()` 函数执行。

中断服务例程每次执行时都会递增名为 `ulParameterValue` 的变量。`ulParameterValue` 在 `xTimerPendFunctionCallFromISR()` 调用中用作 `ulParameter2` 的值，因此，当 `vDeferredHandlingFunction()` 由守护程序任务执行时，在 `vDeferredHandlingFunction()` 调用中也用作 `ulParameter2` 的值。本示例没有使用该函数的另一个参数 `pvParameter1`。

```
static uint32_t ulExampleInterruptHandler( void )
{
    static uint32_t ulParameterValue = 0;
    BaseType_t xHigherPriorityTaskWoken;
```

```
/* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
will get set to pdTRUE inside the interrupt-safe API function if a context switch is
required. */

xHigherPriorityTaskWoken = pdFALSE;

/* Send a pointer to the interrupt's deferred handling function to the daemon task.
The deferred handling function's pvParameter1 parameter is not used, so just set to NULL.
The deferred handling function's ulParameterValue parameter is used to pass a number that is
incremented by one each time this interrupt handler executes. */

xTimerPendFunctionCallFromISR( vDeferredHandlingFunction, /* Function to
execute. */ NULL, /* Not used. */ ulParameterValue, /* Incrementing value. */
&xHigherPriorityTaskWoken );

ulParameterValue++;

/* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
xHigherPriorityTaskWoken was set to pdTRUE inside xTimerPendFunctionCallFromISR() then
calling portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is
still pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS
ports, the Windows port requires the ISR to return a value. The return statement is inside
the Windows version of portYIELD_FROM_ISR(). */

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

下面是 vDeferredHandlingFunction() 的实现。它输出一个固定字符串及其 ulParameter2 参数的值。

```
static void vDeferredHandlingFunction( void *pvParameter1, uint32_t ulParameter2 )

{
    /* Process the event - in this case, just print out a message and the value of
    ulParameter2. pvParameter1 is not used in this example. */

    vPrintStringAndNumber( "Handler function - Processing event ", ulParameter2 );
}
```

下面是所用的 main() 函数。vPeriodicTask() 是定期生成软件中断的任务。它以低于守护程序任务优先级的优先级创建，以确保只要守护程序任务离开“被阻止”状态，守护程序任务就能优先于它执行。

```
int main( void )

{
    /* The task that generates the software interrupt is created at a priority below
    the priority of the daemon task. The priority of the daemon task is set by the
    configTIMER_TASK_PRIORITY compile time configuration constant in FreeRTOSConfig.h. */

    const UBaseType_t ulPeriodicTaskPriority = configTIMER_TASK_PRIORITY - 1;

    /* Create the task that will periodically generate a software interrupt. */

    xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, ulPeriodicTaskPriority, NULL );

    /* Install the handler for the software interrupt. The syntax necessary to do this is
    dependent on the FreeRTOS port being used. The syntax shown here can only be used with the
    FreeRTOS windows port, where such interrupts are only simulated. */

    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );
```

```
/* Start the scheduler so the created task starts executing. */

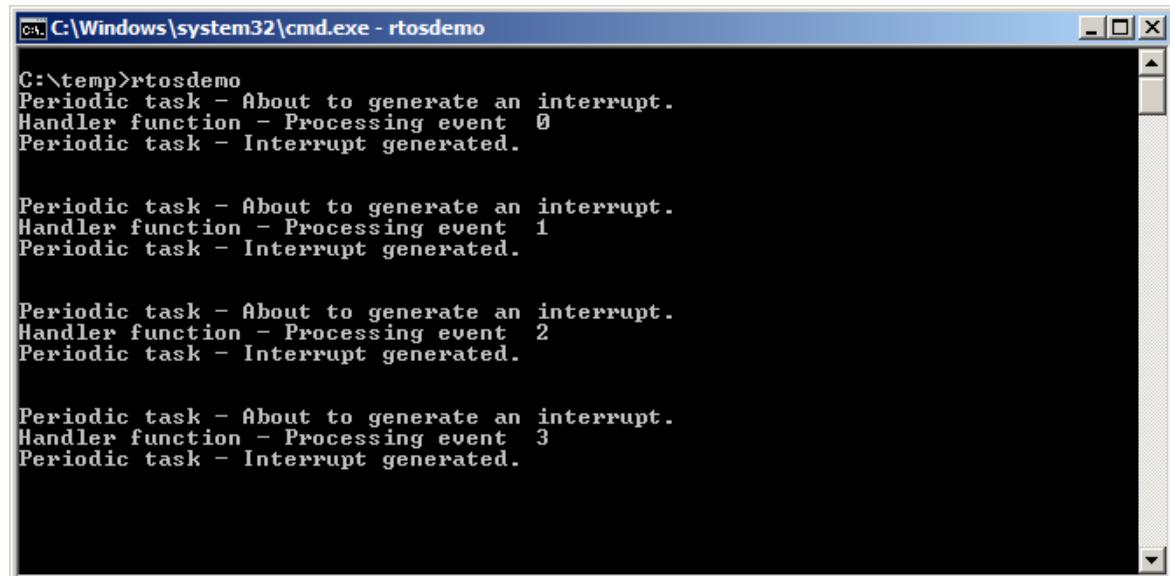
vTaskStartScheduler();

/* As normal, the following line should never be reached. */

for( ;; );

}
```

这段代码产生的输出如下所示。守护程序任务的优先级高于生成软件中断的任务的优先级，因此只要生成中断，守护程序任务就会执行 vDeferredHandlingFunction()。因此，vDeferredHandlingFunction() 输出的消息显示在定期任务输出的两条消息之间，就像使用了信号灯来取消阻止专门的受委托执行中断处理的任务一样。



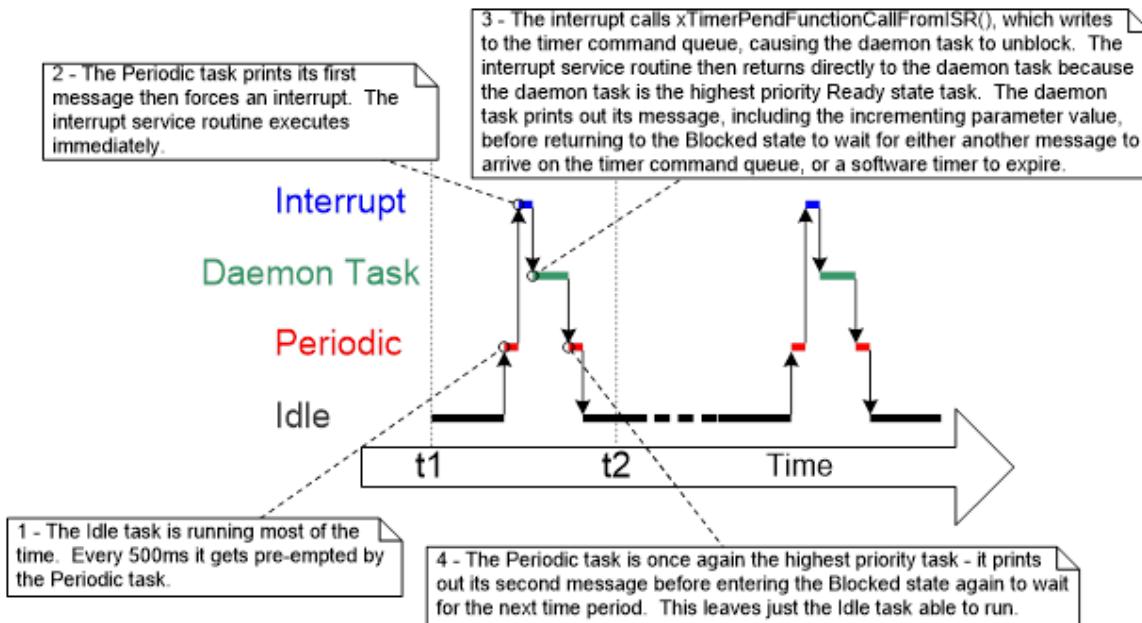
```
C:\temp>rtosdemo
Periodic task - About to generate an interrupt.
Handler function - Processing event  0
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler function - Processing event  1
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler function - Processing event  2
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler function - Processing event  3
Periodic task - Interrupt generated.
```

下面是执行的顺序。



## 在中断服务例程内使用队列

二进制信号灯和计数信号灯用于进行事件通信。队列用于进行事件通信和传输数据。

`xQueueSendToFrontFromISR()` 是在中断服务例程中可以安全使用的 `xQueueSendToFront()` 版本。`xQueueSendToBackFromISR()` 是在中断服务例程中可以安全使用的 `xQueueSendToBack()` 版本。`xQueueReceiveFromISR()` 是在中断服务例程中可以安全使用的 `xQueueReceive()` 版本。

## xQueueSendToFrontFromISR() 和 xQueueSendToBackFromISR() API 函数

下面是 `xQueueSendToFrontFromISR()` API 函数原型。

```
BaseType_t xQueueSendToFrontFromISR( QueueHandle_t xQueue, void *pvItemToQueue BaseType_t *pxHigherPriorityTaskWoken );
```

下面是 `xQueueSendToBackFromISR()` API 函数原型。

```
BaseType_t xQueueSendToBackFromISR( QueueHandle_t xQueue, void *pvItemToQueue BaseType_t *pxHigherPriorityTaskWoken );
```

`xQueueSendFromISR()` 和 `xQueueSendToBackFromISR()` 的功能相同。

下面列出了 `xQueueSendToFrontFromISR()` 和 `xQueueSendToBackFromISR()` 的参数和返回值。

`xQueue`

将向其发送（写入）数据的队列的句柄。该队列句柄从 `xQueueCreate()` 调用返回，用于创建队列。

`pvItemToQueue`

指针，它指向将复制到队列中的数据。队列创建时就设置了队列可以容纳的每个项目的大小，因此相应的字节数将从 pvItemToQueue 复制到队列存储区域。

#### pxHigherPriorityTaskWoken

单个队列可能会使一个或多个任务被阻止以等待数据变为可用。调用 xQueueSendToFrontFromISR() 或 xQueueSendToBackFromISR() 可以使数据变为可用，从而使任务离开“被阻止”状态。如果调用 API 函数会导致任务离开“被阻止”状态，并且取消阻止的任务的优先级高于当前正在执行的任务（被中断的任务），则该 API 函数将 \*pxHigherPriorityTaskWoken 设置为 pdTRUE。如果 xQueueSendToFrontFromISR() 或 xQueueSendToBackFromISR() 将此值设置为 pdTRUE，则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优先级最高的“准备就绪”状态的任务。

可能的返回值有两个：

- pdPASS

仅当数据已成功发送到队列时返回。

- errQUEUE\_FULL

如果因为队列已满而导致数据无法发送到队列，则返回此值。

## 从 ISR 中使用队列时的注意事项

队列是一种简单、方便的将数据从中断传递给任务的方法。如果数据到达的频率较高，使用队列就没有效率。

FreeRTOS 下载中的很多演示应用程序包括一个简单的 UART 驱动程序，它使用队列在 UART 的接收 ISR 外传递字符。在这些演示中，队列用来说明如何从 ISR 中使用队列，以及特意加载系统以便测试 FreeRTOS 端口。以这种方式使用队列的 ISR 并不代表高效的设计，除非数据缓慢到达，否则不建议在生产代码中使用这种方式。更高效的方法包括：

- 使用直接内存访问 (DMA) 硬件来接收和缓冲字符。这种方法几乎没有软件开销。然后，可以使用“直接到任务”通知来取消阻止只有在检测到传输中断后才会处理缓冲区的任务。“直接到任务”通知是从 ISR 取消阻止任务的最有效方法。有关更多信息，请参阅[任务通知 \(p. 178\)](#)。
- 将接收到的每个字符复制到线程安全 RAM 缓冲区。为此，可以使用 FreeRTOS+TCP 中提供的“流缓冲区”。同样，可以使用“直接到任务”通知来取消阻止在接收到完整的消息或检测到传输中断后处理缓冲区的任务。
- 直接在 ISR 中处理接收到的字符，然后使用队列将数据处理结果（而不是原始数据）发送到任务。

## 从中断内部发送和接收队列（示例 19）

此示例说明如何在同一个中断中使用 xQueueSendToBackFromISR() 和 xQueueReceiveFromISR()。为方便起见，中断由软件生成。

创建一个定期任务，该任务每 200 毫秒将五个数字发送到一个队列。它只有在五个值都发送之后才生成软件中断。下面是任务的实现。

```
static void vIntegerGenerator( void *pvParameters )  
{  
    TickType_t xLastExecutionTime;
```

```
uint32_t ulValueToSend = 0;

int i;

/* Initialize the variable used by the call to vTaskDelayUntil(). */

xLastExecutionTime = xTaskGetTickCount();

for( ;; )

{

    /* This is a periodic task. Block until it is time to run again. The task will
execute every 200 ms. */

    vTaskDelayUntil( &xLastExecutionTime, pdMS_TO_TICKS( 200 ) );

    /* Send five numbers to the queue, each value one higher than the previous value.
The numbers are read from the queue by the interrupt service routine. The interrupt
service routine always empties the queue, so this task is guaranteed to be able to write
all five values without needing to specify a block time. */

    for( i = 0; i < 5; i++ )

    {

        xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );

        ulValueToSend++;

    }

    /* Generate the interrupt so the interrupt service routine can read the values from
the queue. The syntax used to generate a software interrupt depends on the FreeRTOS port
being used. The syntax used below can only be used with the FreeRTOS Windows port, in
which such interrupts are only simulated.*/

    vPrintString( "Generator task - About to generate an interrupt.\r\n" );

    vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );

    vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n\r\n" );

}

}
```

中断服务例程反复调用 `xQueueReceiveFromISR()`，直到定期任务写入队列的所有值都已读出，并且队列为空。接收到的每个值的最后两位用作字符串数组的索引。然后，调用 `xQueueSendFromISR()` 将一个指向字符串相应索引位置的指针发送到不同的队列。下面是中断服务例程的实现。

```
static uint32_t ulExampleInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken;

    uint32_t ulReceivedNumber;

    /* The strings are declared static const to ensure they are not allocated on the
interrupt service routine's stack, and so exist even when the interrupt service routine is
not executing. */


```

```
static const char *pcStrings[] =  
{  
    "String 0\r\n",  
    "String 1\r\n",  
    "String 2\r\n",  
    "String 3\r\n"  
};  
  
/* As always, xHigherPriorityTaskWoken is initialized to pdFALSE to be able to detect  
it getting set to pdTRUE inside an interrupt-safe API function. Because an interrupt-safe  
API function can only set xHigherPriorityTaskWoken to pdTRUE, it is safe to use the same  
xHigherPriorityTaskWoken variable in both the call to xQueueReceiveFromISR() and the call  
to xQueueSendToBackFromISR(). */  
  
xHigherPriorityTaskWoken = pdFALSE;  
  
/* Read from the queue until the queue is empty. */  
  
while( xQueueReceiveFromISR( xIntegerQueue, &ulReceivedNumber,  
&xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )  
  
{  
  
    /* Truncate the received value to the last two bits (values 0 to 3 inclusive), and  
then use the truncated value as an index into the pcStrings[] array to select a string  
(char *) to send on the other queue. */  
  
    ulReceivedNumber &= 0x03;  
  
    xQueueSendToBackFromISR( xStringQueue, &pcStrings[ ulReceivedNumber ],  
&xHigherPriorityTaskWoken );  
  
}  
  
/* If receiving from xIntegerQueue caused a task to leave the Blocked state, and if the  
priority of the task that left the Blocked state is higher than the priority of the task  
in the Running state, then xHigherPriorityTaskWoken will have been set to pdTRUE inside  
xQueueReceiveFromISR(). If sending to xStringQueue caused a task to leave the Blocked  
state, and if the priority of the task that left the Blocked state is higher than the  
priority of the task in the Running state, then xHigherPriorityTaskWoken will have been  
set to pdTRUE inside xQueueSendToBackFromISR(). xHigherPriorityTaskWoken is used as the  
parameter to portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken equals pdTRUE, then calling  
portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still  
pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of  
portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why  
this function does not explicitly return a value. */  
  
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );  
}
```

接收来自中断服务例程的字符指针的任务在队列上被阻止，直至有消息到达时，输出其接收到的第一个字符串。下面是其实现。

```
static void vStringPrinter( void *pvParameters )  
{
```

```
char *pcString;  
  
for( ;; )  
  
{  
  
    /* Block on the queue to wait for data to arrive. */  
  
    xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );  
  
    /* Print out the string received. */  
  
    vPrintString( pcString );  
  
}  
  
}
```

和通常情况一样，main() 在启动计划程序之前创建所需的队列和任务。下面是其实现。

```
int main( void )  
  
{  
  
    /* Before a queue can be used, it must first be created. Create both queues used by  
    this example. One queue can hold variables of type uint32_t. The other queue can hold  
    variables of type char*. Both queues can hold a maximum of 10 items. A real application  
    should check the return values to ensure the queues have been successfully created. */  
  
    xIntegerQueue = xQueueCreate( 10, sizeof( uint32_t ) );  
  
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );  
  
    /* Create the task that uses a queue to pass integers to the interrupt service routine.  
    The task is created at priority 1. */  
  
    xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );  
  
    /* Create the task that prints out the strings sent to it from the interrupt service  
    routine. This task is created at the higher priority of 2. */  
  
    xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );  
  
    /* Install the handler for the software interrupt. The syntax required to do this is  
    depends on the FreeRTOS port being used. The syntax shown here can only be used with the  
    FreeRTOS Windows port, where such interrupts are only simulated. */  
  
    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );  
  
    /* Start the scheduler so the created tasks start executing. */  
  
    vTaskStartScheduler();  
  
    /* If all is well, then main() will never reach here because the scheduler will  
    now be running the tasks. If main() does reach here, then it is likely that there was  
    insufficient heap memory available for the idle task to be created. */  
  
    for( ;; );  
  
}
```

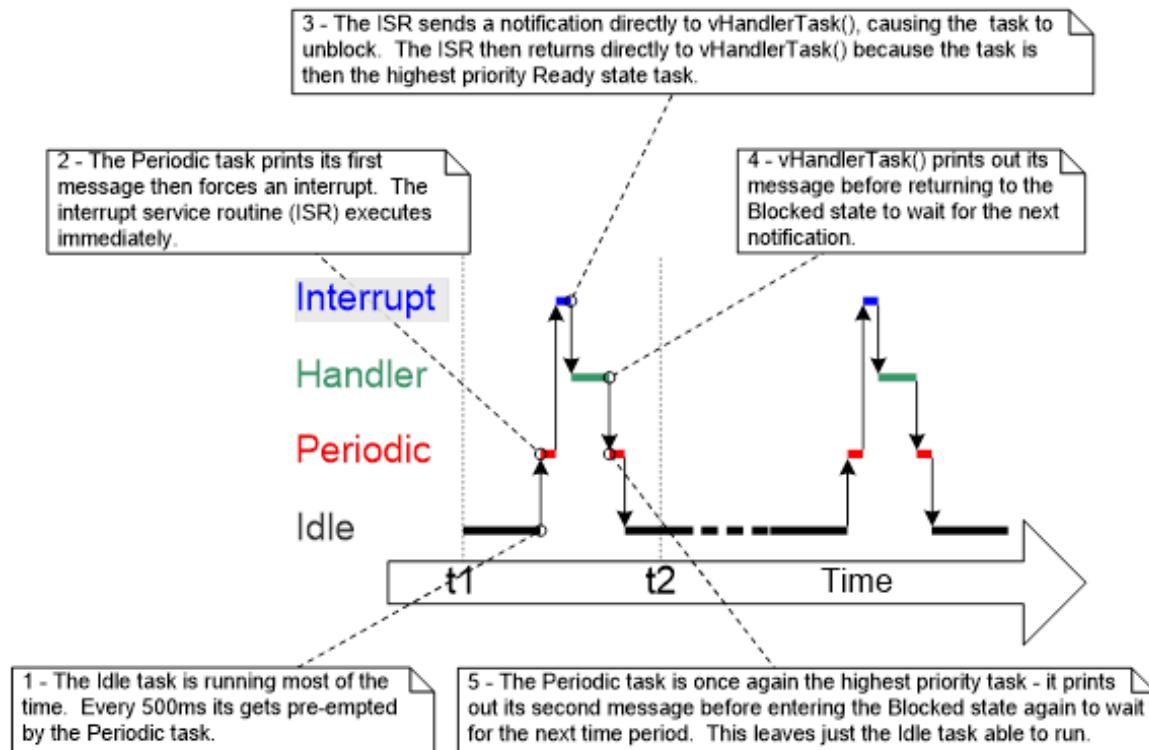
下面是输出的内容。可以看到中断接收所有五个整数，生成五个字符串作为响应。

```
ex C:\WINDOWS\system32\cmd.exe - rtosdemo
String 3
String 0
String 1
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
String 2
String 3
String 0
String 1
String 2
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
String 3
String 0
String 1
String 2
String 3
Generator task - Interrupt generated.
```

下面是执行的顺序。



## 中断嵌套

人们常常混淆任务优先级和中断优先级。本节讨论中断优先级，即 ISR 彼此之间的执行优先级。在任何情况下，分配给任务的优先级与分配给中断的优先级毫无关系。硬件决定 ISR 何时执行。软件决定任务何时执行。为了响应硬件中断而执行 ISR 将中断任务，但是任务不能优先于 ISR 执行。

支持中断嵌套的端口需要在 FreeRTOSConfig.h 中定义下表中列出的一个或两个常量。configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 和 configMAX\_API\_CALL\_INTERRUPT\_PRIORITY 定义同一个属性。较旧的 FreeRTOS 端口使用 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY。较新的 FreeRTOS 端口使用 configMAX\_API\_CALL\_INTERRUPT\_PRIORITY。

下表列出了控制中断嵌套的常量。

常量	描述
configMAX_SYSCALL_INTERRUPT_PRIORITY 或 configMAX_API_CALL_INTERRUPT_PRIORITY	设置可调用中断安全 FreeRTOS API 函数的最高中断优先级。
configKERNEL_INTERRUPT_PRIORITY	设置由计时中断使用的中断优先级，必须始终将它设置为尽可能低的优先级。  如果所使用的 FreeRTOS 端口并不同时使用 configMAX_SYSCALL_INTERRUPT_PRIORITY 常量，则使用中断安全 FreeRTOS API 函数的任何中断也必须以 configKERNEL_INTERRUPT_PRIORITY 定义的优先级执行。

每个中断源都有数值和逻辑优先级。

- 数值

分配给中断优先级的数字。例如，如果为中断分配优先级 7，则其数值优先级为 7。同样，如果为中断分配优先级 200，则其数值优先级为 200。

- 逻辑

描述中断的优先级高于其他中断。如果两个优先级不同的中断同时发生，则处理器为逻辑优先级更高的中断执行 ISR。

中断可以中断（嵌套）任何优先级更低的中断，但是不能中断逻辑优先级相同或更高的中断。

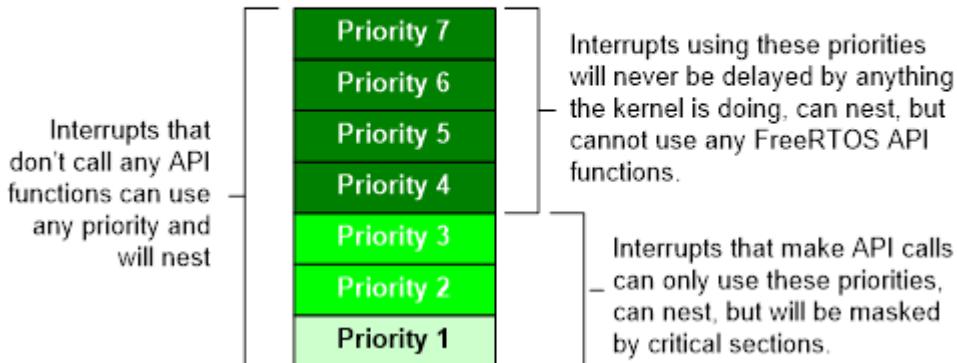
中断的数值优先级和逻辑优先级之间的关系取决于处理器架构。在某些处理器上，分配给中断的数值优先级越大，中断的逻辑优先级越高。在另一些处理器架构中，分配给中断的数值优先级越大，中断的逻辑优先级越低。

通过将 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 设置为比 configKERNEL\_INTERRUPT\_PRIORITY 更高的逻辑中断优先级，可以创建一个完整的中断嵌套模型。在下图的情况下：

- 处理器有 7 个唯一中断优先级。
- 数值优先级为 7 的中断比数值优先级为 1 的中断具有更高的逻辑优先级。
- configKERNEL\_INTERRUPT\_PRIORITY 设置为 1。
- configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 设置为 3。

下图是影响中断嵌套行为的常量。

```
configMAX_SYSCALL_INTERRUPT_PRIORITY = 3  
configKERNEL_INTERRUPT_PRIORITY = 1
```



- 当内核或应用程序执行到关键部分时，无法执行优先级为 1 到 3（含）的中断。在这些优先级运行的 ISR 可以使用中断安全 FreeRTOS API 函数。有关关键部分的信息，请参阅[资源管理 \(p. 141\)](#)。
- 优先级为 4 或更高优先级的中断不受关键部分影响，因此，只要在硬件本身的限制范围内，计划程序完全无法阻止这些中断立即执行。在这些优先级执行的 ISR 不能使用任何 FreeRTOS API 函数。
- 通常情况下，如果功能需要非常严格的计时准确性（例如电机控制），则应使用高于 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 的优先级，以确保计划程序不会对中断响应时间引入抖动。

## ARM Cortex-M 和 ARM GIC 用户

本节内容仅部分适用于 Cortex-M0 和 Cortex-M0+ 内核。Cortex-M 处理器的中断配置令人困惑，容易出错。为帮助进行开发，FreeRTOS Cortex-M 端口会自动检查中断配置，但前提是需要定义 configASSERT()。有关 configASSERT() 的信息，请参阅[开发人员支持 \(p. 202\)](#)。

ARM Cortex 核心和 ARM 通用中断控制器 (GIC) 使用低数值优先级数字表示高逻辑优先级中断。这与直觉相反。如果需要为中断分配低逻辑优先级，则必须为它分配高数值优先级。如果需要为中断分配高逻辑优先级，则必须为它分配低数值优先级。

Cortex-M 中断控制器最多允许使用 8 位来指定每个中断优先级，因此，255 就是最低优先级。零是最高优先级。但是，Cortex-M 微控制器通常仅实现八位中的一部分。实现的位数取决于微控制器系列。

如果只实现部分位，则只能使用该字节中最高有效位。最低有效位不会实现。未实现的位可以使用任何值，但通常设置为 1。下图说明二进制 101 优先级在实现四位优先级的 Cortex-M 微控制器中是如何存储的。

Priority 5, or 95, in a device that implements 4 priority bits

0	1	0	1	1	1	1	1
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

可以看到，二进制值 101 已前移到 4 个最高有效位中，因为 4 个最低有效位未实现。未实现的位已设置为 1。

某些库函数需要在优先级值移到已实现的（最高有效）位之后指定这些值。如果使用这类函数，上图所示的优先级可以指定为十进制 95。十进制 95 是二进制 101 前移 4 位，构成二进制 101nnnn（其中 n 是未实现的位）。未实现的位设置为 1，构成二进制 1011111。

某些库函数需要在优先级值前移至已实现（最高有效）位之前指定这些值。如果使用这类函数，上图所示的优先级必须指定为十进制 5。如果不进行任何转换，十进制 5 就是二进制 101。

`configMAX_SYSCALL_INTERRUPT_PRIORITY` 和 `configKERNEL_INTERRUPT_PRIORITY` 必须以允许直接写入 Cortex-M 寄存器的方式指定（即，优先级值前移到已实现的位之后）。

`configKERNEL_INTERRUPT_PRIORITY` 必须始终设置为最低中断优先级。未实现的优先级位可设置为 1，因此常量始终可以设置为 255，无论实现多少个优先级位都是如此。

Cortex-M 中断默认为优先级零，即最高优先级。Cortex-M 硬件的实现不允许 `configMAX_SYSCALL_INTERRUPT_PRIORITY` 设置为 0，因此使用 FreeRTOS API 的中断的优先级不能保留为其默认值。

# 资源管理

本节内容：

- 何时以及为什么有必要进行资源管理和控制。
- 关键部分。
- 互斥的含义。
- 暂停计划程序的含义。
- 互斥锁的用法。
- 如何创建和使用网关守卫任务。
- 优先级反转的含义以及优先级继承如何减少（但不消除）其影响。

在多任务处理系统中，如果一项任务开始访问资源，但未在退出“运行”状态之前完成其访问，则可能出现错误。如果该任务让资源处于不一致的状态，则其他任何任务或中断访问该资源都可能导致数据损坏或其他类似问题。

以下是一些示例：

## 1. 访问外设

考虑以下情况，有两个任务尝试写入液晶显示器 (LCD)。

- a. 任务 A 执行，开始向 LCD 写入字符串“Hello world”。
- b. 任务 A 输出该字符串开头的“Hello w”后，任务 B 就抢先执行。
- c. 任务 B 将“Abort, Retry, Fail?”写入 LCD 后进入“被阻止”状态。
- d. 任务 A 从被抢先执行的位置继续执行，输出该字符串的其余字符（“orld”）。

现在，LCD 显示损坏的字符串“Hello wAbort, Retry, Fail?orld”。

## 2. 读取、修改、写入操作

下面是一行 C 代码以及说明它通常如何转换为程序集代码的示例。可以看到，PORTA 的值首先从内存读入寄存器，在寄存器中修改，然后写回内存。这称为读取、修改和写入操作。

```
/* The C code being compiled. */
PORTA |= 0x01;

/* The assembly code produced when the C code is compiled. */

LOAD R1,[#PORTA] ; Read a value from PORTA into R1
MOVE R2,#0x01 ; Move the absolute constant 1 into R2
OR R1,R2 ; Bitwise OR R1 (PORTA) with R2 (constant 1)
STORE R1,[#PORTA] ; Store the new value back to PORTA
```

这是一个非原子操作，因为它需要多条指令才能完成，并且可以中断。考虑以下情况，两个任务尝试更新名为 PORTA 的内存映射寄存器。

1. 任务 A 将 PORTA 的值加载到寄存器中，这是操作的读取部分。

2. 任务 A 在完成同一操作的修改和写入部分之前，任务 B 抢先执行。
3. 任务 B 更新 PORTA 的值，然后进入“被阻止”状态。
4. 任务 A 从被抢先执行的位置继续执行。它修改保存在寄存器中的 PORTA 值的副本，然后将更新的值写回 PORTA。

在这种情况下，任务 A 更新并写回 PORTA 的过期值。任务 B 在任务 A 获取 PORTA 值的副本之后、在任务 A 将其修改值写回 PORTA 寄存器之前修改 PORTA。当任务 A 写入 PORTA 时，将覆盖任务 B 已执行的修改，实际上会损坏 PORTA 寄存器值。

本示例使用的是外设寄存器，不过，在对变量执行读取、修改、写入操作时，适用同样的原则。

#### 1. 对变量的非原子访问

更新结构的多个成员或更新大于架构自然字大小的变量（例如，在 16 位计算机上更新 32 位变量）是非原子操作的示例。如果这些操作中断，则会导致数据丢失或损坏。

#### 2. 函数可重入性

如果某个函数可以安全地从多个任务调用，或从任务和中断中都可以调用，则该函数是可重入函数。可重入函数可从多个执行线程中访问，不会损坏数据或逻辑操作，因此可重入函数是线程安全的。每个任务都维护自己的堆栈和自己的一组处理器（硬件）寄存器值。如果除了堆栈中存储的数据或寄存器中保存的数据之外，函数不访问其他数据，则该函数是可重入函数，是线程安全的。下面是可重入函数的示例。

```
/* A parameter is passed into the function. This will either be passed on the stack, or
in a processor register. Either way is safe because each task or interrupt that calls
the function maintains its own stack and its own set of register values, so each task or
interrupt that calls the function will have its own copy of lVar1. */
long lAddOneHundred( long lVar1 )

{
    /* This function scope variable will also be allocated to the stack or a register,
depending on the compiler and optimization level. Each task or interrupt that calls this
function will have its own copy of lVar2. */

    long lVar2;
    lVar2 = lVar1 + 100;
    return lVar2;
}
```

下面是不可重入函数的示例。

```
/* In this case lVar1 is a global variable, so every task that calls lNonsenseFunction will
access the same single copy of the variable. */

long lVar1;

long lNonsenseFunction( void )
{
    /* lState is static, so is not allocated on the stack. Each task that calls this
function will access the same single copy of the variable. */

    static long lState = 0;
    long lReturn;
```

```
switch( lState )  
{  
    case 0 : lReturn = lVar1 + 10;  
    lState = 1;  
    break;  
    case 1 : lReturn = lVar1 + 20;  
    lState = 0;  
    break;  
}  
}
```

## 互斥

为了始终确保数据一致性，可使用互斥方法来管理在任务之间或在任务与中断之间共享的资源的访问。目标是确保在任务开始访问不是可重入、线程安全的共享资源时，该任务对该资源进行独占式访问，直到资源已返回一致状态。

FreeRTOS 提供了几项可用于实现互斥的功能，但最好的互斥方法是，只要可行，在设计应用程序时尽量不共享资源，每个资源仅从单个任务中访问。

## 关键部分和暂停计划程序

### 基本关键部分

基本关键部分是前后分别由 taskENTER\_CRITICAL() 和 taskEXIT\_CRITICAL() 宏调用包围起来的代码区域。关键部分也称为关键区域。

taskENTER\_CRITICAL() 和 taskEXIT\_CRITICAL() 不接受任何参数，也不返回值。（与函数类似的宏并不像真正的函数那样返回值，但将宏视为函数是最简单的。）

下面介绍宏的使用，其中关键部分用来保护对寄存器的访问。

```
/* Ensure access to the PORTA register cannot be interrupted by placing it within a  
critical section. Enter the critical section. */  
  
taskENTER_CRITICAL();  
  
/* A switch to another task cannot occur between the call to taskENTER_CRITICAL() and the  
call to taskEXIT_CRITICAL(). Interrupts might still execute on FreeRTOS ports that allow  
interrupt nesting, but only interrupts whose logical priority is above the value assigned  
to the configMAX_SYSCALL_INTERRUPT_PRIORITY constant. Those interrupts are not permitted  
to call FreeRTOS API functions. */  
  
PORTA |= 0x01;  
  
/* Access to PORTA has finished, so it is safe to exit the critical section. */
```

```
taskEXIT_CRITICAL();
```

几个示例使用名为 vPrintString() 的函数向标准输出写入字符串，标准输出是使用 FreeRTOS Windows 端口时的终端窗口。vPrintString() 可从许多不同的任务中调用，因此，在理论上，它的实现可以使用关键部分来保护对标准输出的访问，如下所示。

```
void vPrintString( const char *pcString )  
{  
    /* Write the string to stdout, using a critical section as a crude method of mutual  
    exclusion. */  
  
    taskENTER_CRITICAL();  
  
    {  
  
        printf( "%s", pcString );  
  
        fflush( stdout );  
  
    }  
  
    taskEXIT_CRITICAL();  
}
```

以这种方式实现的关键部分是提供互斥的极其粗略的方法。其工作原理是完全禁用中断或根据 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 设置的中断优先级禁用中断，具体取决于所用的 FreeRTOS 端口。抢先上下文切换只能在中断内发生，因此只要中断保持禁用状态，调用 taskENTER\_CRITICAL() 的任务肯定会保持“运行”状态，直至退出关键部分。

基本关键部分必须保持简短。否则，它们将对中断响应时间产生不利影响。taskENTER\_CRITICAL() 的每个调用都必须与 taskEXIT\_CRITICAL() 的调用成对使用。因此，标准输出（计算机将其输出数据写入的 stdout 或流）不应使用关键部分保护（如前面的代码所示），因为写入终端是耗时相对较长的操作。本节中的示例将探索替代解决方案。

因为内核会保留嵌套深度的计数，所以关键部分可以安全地进行嵌套。仅当嵌套深度返回到零（即对前面的每个 taskENTER\_CRITICAL() 调用均已执行 taskEXIT\_CRITICAL() 调用时），才会退出关键部分。

对于运行 FreeRTOS 的处理器，任务要改变其中断启用状态的唯一合法方式是调用 taskENTER\_CRITICAL() 和 taskEXIT\_CRITICAL()。通过其他任何方式改变中断启用状态均会使宏的嵌套计数失效。

taskENTER\_CRITICAL() 和 taskEXIT\_CRITICAL() 不以“FromISR”结尾，因此不能从中断服务例程中调用。taskENTER\_CRITICAL\_FROM\_ISR() 是 taskENTER\_CRITICAL() 的中断安全版本。taskEXIT\_CRITICAL\_FROM\_ISR() 是 taskEXIT\_CRITICAL() 的中断安全版本。中断安全版本仅适用于允许中断嵌套的 FreeRTOS 端口。不允许中断嵌套的端口不能使用这种版本。

taskENTER\_CRITICAL\_FROM\_ISR() 返回一个值，该值必须传递给相匹配的 taskEXIT\_CRITICAL\_FROM\_ISR() 调用，如下所示。

```
void vAnInterruptServiceRoutine( void )  
{  
  
    /* Declare a variable in which the return value from taskENTER_CRITICAL_FROM_ISR() will  
    be saved. */  
  
    UBaseType_t uxSavedInterruptStatus;  
  
    /* This part of the ISR can be interrupted by any higher priority interrupt. */
```

```
/* Use taskENTER_CRITICAL_FROM_ISR() to protect a region of this ISR. Save the value
returned from taskENTER_CRITICAL_FROM_ISR() so it can be passed into the matching call to
taskEXIT_CRITICAL_FROM_ISR(). */

uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

/* This part of the ISR is between the call to taskENTER_CRITICAL_FROM_ISR() and
taskEXIT_CRITICAL_FROM_ISR(), so can only be interrupted by interrupts that have a
priority above that set by the configMAX_SYSCALL_INTERRUPT_PRIORITY constant. */

/* Exit the critical section again by calling taskEXIT_CRITICAL_FROM_ISR(), passing in
the value returned by the matching call to taskENTER_CRITICAL_FROM_ISR(). */

taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );

/* This part of the ISR can be interrupted by any higher priority interrupt. */

}
```

如果执行进入然后退出关键部分的代码比执行受关键部分保护的代码需要更多处理时间，就是非常浪费的。基本关键部分可以极快进入、极快退出，始终具有确定性，因此，如果受保护的代码区域很短，最适合使用基本关键部分。

## 暂停 ( 或锁定 ) 计划程序

通过暂停计划程序也可以创建关键部分。暂停计划程序有时也称为锁定计划程序。

基本关键部分保护某个代码区域，使其不被其他任务或中断访问。因为中断保持启用状态，所以通过暂停计划程序实现的关键部分只能保护某个代码区域，使其不被其他任务访问。

因太长而无法简单通过禁用中断来实现的关键部分，可改为通过暂停计划程序来实现。但是，在计划程序暂停时，中断活动会使恢复（或取消暂停）计划程序成为耗时相对较长的操作。在每种情况下考虑使用相应的最佳方法。

## vTaskSuspendAll() API 函数

下面是 vTaskSuspendAll() API 函数原型。

```
void vTaskSuspendAll( void );
```

计划程序是通过调用 vTaskSuspendAll() 暂停的。暂停计划程序可防止进行上下文切换，但会使中断保持启用状态。如果中断在计划程序暂停时请求上下文切换，该请求将会挂起，在计划程序恢复（取消暂停）后才能执行。

计划程序暂停时不能调用 FreeRTOS API 函数。

## xTaskResumeAll() API 函数

下面是 xTaskResumeAll() API 函数原型。

```
BaseType_t xTaskResumeAll( void );
```

计划程序是通过调用 `xTaskResumeAll()` 恢复 ( 取消暂停 ) 的。

下表列出了 `xTaskResumeAll()` 的返回值。

返回的值	描述
返回的值	在计划程序暂停时请求的上下文切换将会挂起，在计划程序恢复后才能执行。如果在 <code>xTaskResumeAll()</code> 返回之前执行挂起的上下文切换，将返回 <code>pdTRUE</code> 。否则，返回 <code>pdFALSE</code> 。

因为内核会保留嵌套深度的计数，所以嵌套调用 `vTaskSuspendAll()` 和 `xTaskResumeAll()` 是安全的。仅当嵌套深度返回到零（即对前面的每个 `vTaskSuspendAll()` 调用均已执行 `xTaskResumeAll()` 调用时），才会恢复计划程序。

下面的代码介绍 `vPrintString()` 的实现，该实现暂停计划程序以保护对终端输出的访问。

```
void vPrintString( const char *pcString )
{
    /* Write the string to stdout, suspending the scheduler as a method of mutual
       exclusion. */

    vTaskSuspendScheduler();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    xTaskResumeScheduler();
}
```

## 互斥锁 ( 和二进制信号灯 )

互斥锁 ( 互斥 ) 是一种特殊类型的二进制信号灯，用于控制对两个或多个任务共享的资源的访问。要使互斥锁变为可用，必须在 `FreeRTOSConfig.h` 中将 `configUSE_MUTEXES` 设置为 1。

在互斥情况下使用互斥锁时，可将其视为与共享资源关联的令牌。任务要合法地访问资源，必须先成功地获取令牌（成为令牌持有者）。当令牌持有者完成对资源的访问时，必须释放回令牌。仅当令牌返回后，其他任务才能成功地获取令牌，然后安全地访问同一共享资源。除非任务持有令牌，否则不允许任务访问共享资源。

尽管互斥锁和二进制信号灯有许多共同的特征，但互斥锁用于互斥的情况与二进制信号灯用于同步的情况完全不同。主要差别在于获取信号灯后对信号灯的处理：

- 用于互斥的信号灯始终必须返回。
- 用于同步的信号灯通常被丢弃，不会返回。

这种机制纯粹通过应用程序编写的规则起作用。没有任何原因导致任务无法随时访问资源，但每个任务都同意不这样做，除非它能够成为互斥锁持有者。

## xSemaphoreCreateMutex() API 函数

FreeRTOS V9.0.0 还包含 xSemaphoreCreateMutexStatic() 函数，该函数分配在编译时静态创建互斥锁所需的内存。互斥锁是一种信号灯。所有不同类型的 FreeRTOS 信号灯的句柄都存储在类型为 SemaphoreHandle\_t 的变量中。

互斥锁必须先创建，然后才能使用。要创建互斥锁类型的信号灯，请使用 xSemaphoreCreateMutex() API 函数。

下面是 xSemaphoreCreateMutex() API 函数原型。

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

下表列出了 xSemaphoreCreateMutex() 返回值。

## 将 vPrintString() 重新编写为使用信号灯（示例 20）

此示例创建 vPrintString() 的新版本，它名为 prvNewPrintString()，然后从多个任务中调用这一新函数。prvNewPrintString() 的功能与 vPrintString() 相同，但是使用互斥锁而不是通过锁定计划程序来控制对标准输出的访问。

下面是 prvNewPrintString() 的实现。

```
static void prvNewPrintString( const char *pcString )
{
    /* The mutex is created before the scheduler is started, so already exists by the time
       this task executes. Attempt to take the mutex, blocking indefinitely to wait for the mutex
       if it is not available right away. The call to xSemaphoreTake() will only return when the
       mutex has been successfully obtained, so there is no need to check the function return
       value. If any other delay period was used, then the code must check that xSemaphoreTake()
       returns pdTRUE before accessing the shared resource (which in this case is standard out).
       Indefinite timeouts are not recommended for production code. */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* The following line will only execute after the mutex has been successfully
           obtained. Standard out can be accessed freely now because only one task can have the mutex
           at any one time. */
        printf( "%s", pcString );
        fflush( stdout );
        /* The mutex MUST be given back! */
    }
    xSemaphoreGive( xMutex );
}
```

prvPrintTask() 实现的任务的两个实例重复调用 prvNewPrintString()。在每次调用之间使用随机延迟时间。任务参数用于向每个任务实例传递一个唯一的字符串。下面是 prvPrintTask() 的实现。

```
static void prvPrintTask( void *pvParameters )
```

```
{  
  
    char *pcStringToPrint;  
  
    const TickType_t xMaxBlockTimeTicks = 0x20;  
  
    /* Two instances of this task are created. The string printed by the task is passed  
    into the task using the task's parameter. The parameter is cast to the required type. */  
  
    pcStringToPrint = ( char * ) pvParameters;  
  
    for( ;; )  
  
    {  
  
        /* Print out the string using the newly defined function. */  
  
        prvNewPrintString( pcStringToPrint );  
  
        /* Wait a pseudo random time. Note that rand() is not necessarily reentrant,  
        but in this case it does not really matter because the code does not care what value is  
        returned. In a more secure application, a version of rand() that is known to be reentrant  
        should be used or calls to rand() should be protected using a critical section. */  
  
        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );  
  
    }  
  
}
```

和通常情况一样，main() 仅创建互斥锁和任务，然后启动计划程序。

prvPrintTask() 的两个实例是以不同优先级创建的，因此优先级较低的任务有时会被优先级较高的任务抢先执行。因为互斥锁用于确保每个任务都可对终端进行互斥访问，所以即使发生抢先，显示的字符串仍是正确的，不会损坏。通过缩短任务处于“被阻止”状态的最长时间（由 xMaxBlockTimeTicks 常量设置），可以提高抢先频率。

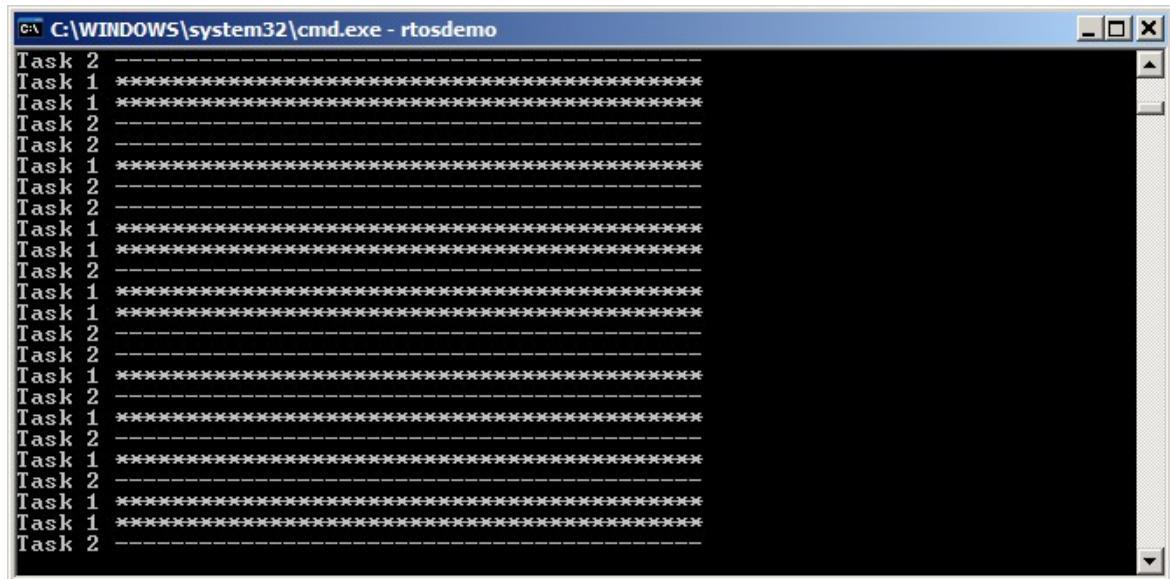
对 FreeRTOS Windows 端口使用示例 20 时的注意事项：

- 调用 printf() 会生成 Windows 系统调用。Windows 系统调用不受 FreeRTOS 控制，会引入不稳定性。
- Windows 系统调用的执行方式意味着极少出现损坏的字符串，即使不使用互斥锁也是如此。

```
int main( void )  
  
{  
  
    /* Before a semaphore is used it must be explicitly created. In this example, a mutex  
    type semaphore is created. */  
  
    xMutex = xSemaphoreCreateMutex();  
  
    /* Check that the semaphore was created successfully before creating the tasks. */  
  
    if( xMutex != NULL )  
  
    {  
  
        /* Create two instances of the tasks that write to stdout. The string they write  
        is passed in to the task as the task's parameter. The tasks are created at different  
        priorities so some preemption will occur. */  
  
    }  
  
}
```

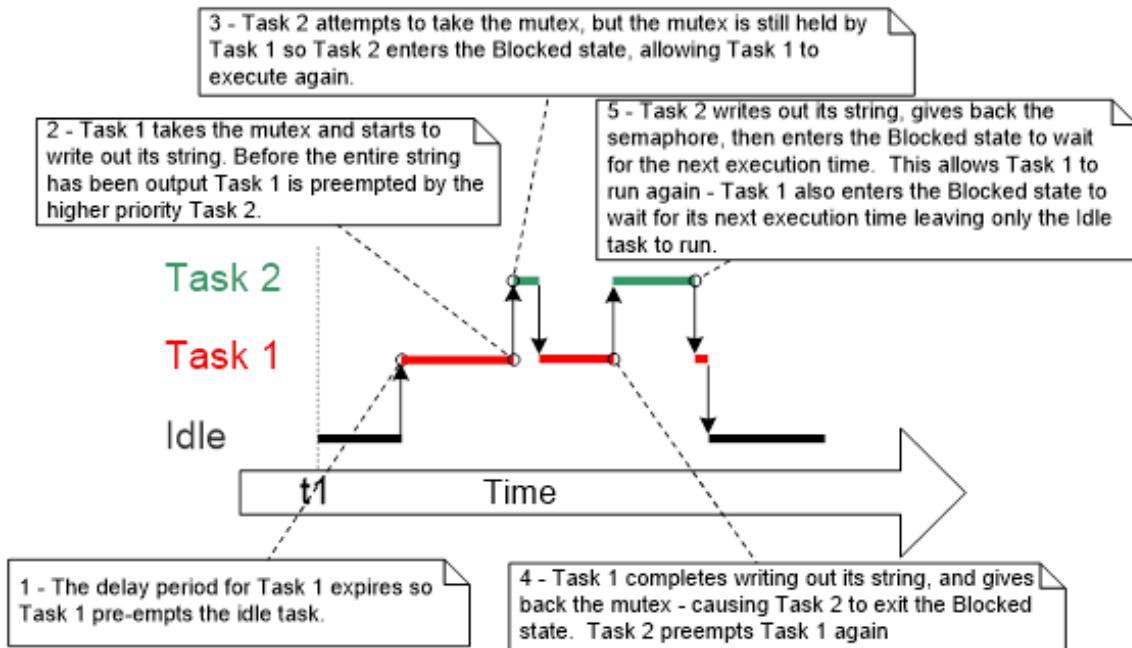
```
xTaskCreate( prvPrintTask, "Print1", 1000, "Task 1  
*****\r\n", 1, NULL );  
  
xTaskCreate( prvPrintTask, "Print2", 1000, "Task 2  
-----\r\n", 2, NULL );  
  
/* Start the scheduler so the created tasks start executing. */  
  
vTaskStartScheduler();  
  
}  
  
/* If all is well, then main() will never reach here because the scheduler will  
now be running the tasks. If main() does reach here, then it is likely that there was  
insufficient heap memory available for the idle task to be created. */  
  
for( ;; );  
  
}
```

下面是输出。



```
Task 2 ---  
Task 1 *****  
Task 1 *****  
Task 2 ---  
Task 2 ---  
Task 1 *****  
Task 2 ---  
Task 2 ---  
Task 1 *****  
Task 1 *****  
Task 2 ---  
Task 2 ---  
Task 1 *****  
Task 2 ---  
Task 2 ---  
Task 1 *****  
Task 2 ---  
Task 1 *****  
Task 2 ---  
Task 1 *****
```

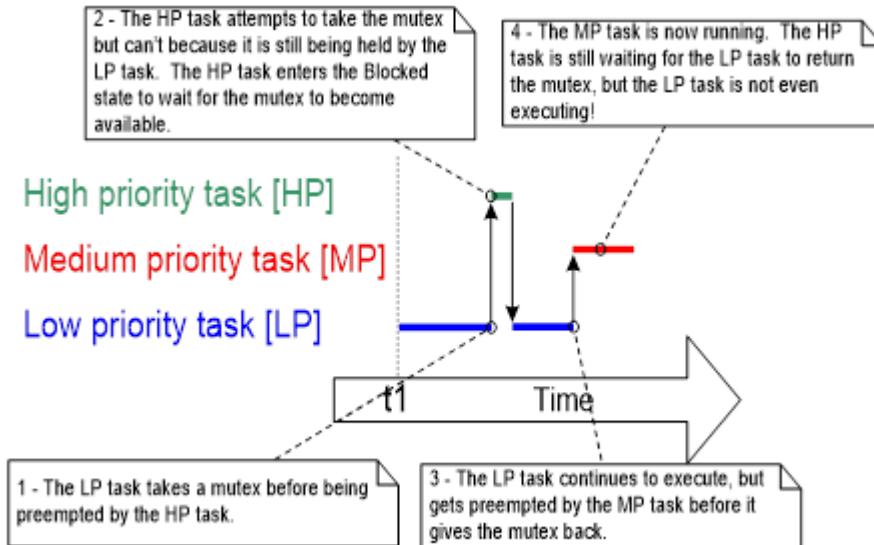
下图是可能的执行顺序。



如预期的一样，终端显示的字符串没有损坏。随机排序是任务使用随机延迟时段的结果。

## 优先级反转

下图说明使用互斥锁实现互斥的潜在缺陷之一。该执行顺序显示，优先级较高的 Task 2 必须等待优先级较低的 Task 1 放弃对互斥锁的控制。优先级较高的任务以这种方式被优先级较低的任务延迟，这称为优先级反转。如果中等优先级的任务在高优先级任务等待信号灯时开始执行，会加重这种不良行为。结果是，高优先级任务等待低优先级任务，低优先级任务甚至无法执行。下图说明了最糟糕的情况。

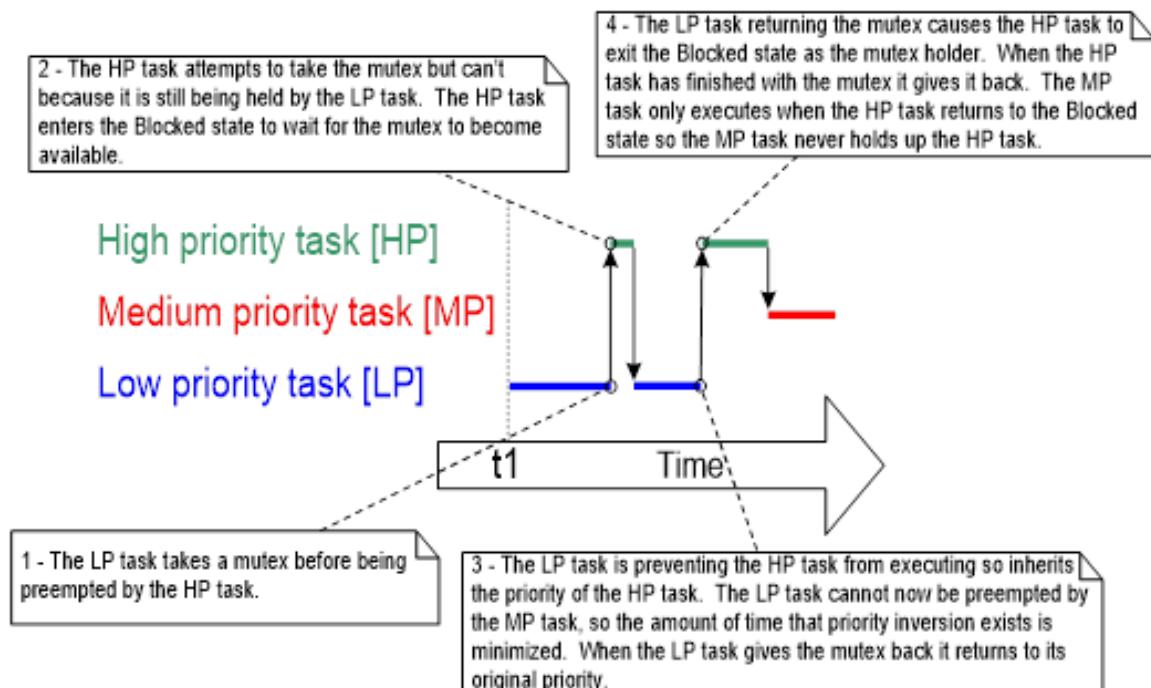


优先级反转是一个严重问题，但在小型嵌入式系统中，通过在设计系统时考虑资源的访问方式，可以避免这一问题。

## 优先级继承

FreeRTOS 互斥锁和二进制信号灯非常相似。差别在于，互斥锁包含基本的优先级继承机制，而二进制信号灯不包括。优先级继承是一种可以尽量减少优先级反转负面影响的方案。它不解决优先级反转问题，但通过确保反转始终具有时效性，从而减少其影响。但是，优先级继承会使系统时间分析复杂化。它不适合用来实现正确的系统操作。

优先级继承的工作方式是将互斥锁持有者的优先级暂时提高至尝试获取相同互斥锁的最高优先级任务的优先级。持有互斥锁的低优先级任务继承等待互斥锁的任务的优先级。下图说明，当互斥锁持有者释放回互斥锁时，其优先级自动重置为其原始值。



优先级继承功能会影响使用互斥锁的任务的优先级。因此，不得从中断服务例程中使用互斥锁。

## 死锁（或抱死）

死锁 或抱死 是将互斥锁用于互斥的另一个潜在缺陷。

当两个任务因为都在等待对方持有的资源而无法继续时，即发生死锁。考虑以下情况：任务 A 和任务 B 都需要获取互斥锁 X 和互斥锁 Y 来执行操作。

1. 任务 A 执行并成功获取互斥锁 X。
2. 任务 A 被任务 B 抢先执行。
3. 任务 B 成功获取互斥锁 Y，然后尝试获取互斥锁 X，但互斥锁 X 已被任务 A 持有，因此对任务 B 不可用。任务 B 选择进入“被阻止”状态，等待互斥锁 X 被释放。
4. 任务 A 继续执行。它尝试获取互斥锁 Y，但互斥锁 Y 已被任务 B 持有，因此对任务 A 不可用。任务 A 选择进入“被阻止”状态，等待互斥锁 Y 被释放。

任务 A 在等待任务 B 持有的互斥锁，而任务 B 在等待任务 A 持有的互斥锁。这两个任务都无法继续，因此发生死锁。

与优先级反转一样，避免死锁的最佳方法是在设计系统时确保不会发生死锁。任务无限期地等待（无超时）获取互斥锁通常是糟糕的做法。而应改用比等待互斥锁的预期最长时间略长的超时。如果在该超时内无法获取互斥锁表明存在设计错误，可能会导致死锁。

如果系统设计人员足够了解整个应用程序，则可以确定并删除可能发生死锁的区域，因此死锁在小型嵌入式系统中并不是个大问题。

## 递归互斥锁

任务自身也可能发生死锁。如果任务尝试多次获取互斥锁而不先返回该互斥锁，就会出现这种情况。考虑以下情况：

1. 任务成功获取互斥锁。
2. 在持有互斥锁时，任务调用库函数。
3. 库函数的实现尝试获取相同的互斥锁，并进入“被阻止”状态，等待互斥锁变为可用。

该任务处于“被阻止”状态，等待互斥锁返回，但该任务已经是互斥锁持有者。任务处于“被阻止”状态等待自身，因此发生死锁。

通过使用递归互斥锁代替标准互斥锁可以避免这类死锁。同一任务可以多次获取递归互斥锁。仅在对之前获取递归互斥锁的每次调用都执行释放递归互斥锁的调用后，才会返回互斥锁。

标准互斥锁和递归互斥锁的创建和使用方式相似：

- 标准互斥锁是使用 `xSemaphoreCreateMutex()` 创建的。递归互斥锁是使用 `xSemaphoreCreateRecursiveMutex()` 创建的。这两个 API 函数具有相同的原型。
- 标准互斥锁是使用 `xSemaphoreTake()` 获取的。递归互斥锁是使用 `xSemaphoreTakeRecursive()` 获取的。这两个 API 函数具有相同的原型。
- 标准互斥锁是使用 `xSemaphoreGive()` 释放的。递归互斥锁是使用 `xSemaphoreGiveRecursive()` 释放的。这两个 API 函数具有相同的原型。

下面的代码说明如何创建和使用递归互斥锁。

```
/* Recursive mutexes are variables of type SemaphoreHandle_t. */

SemaphoreHandle_t xRecursiveMutex;

/* The implementation of a task that creates and uses a recursive mutex. */

void vTaskFunction( void *pvParameters )

{

    const TickType_t xMaxBlock20ms = pdMS_TO_TICKS( 20 );

    /* Before a recursive mutex is used it must be explicitly created. */
    xRecursiveMutex = xSemaphoreCreateRecursiveMutex();

    /* Check the semaphore was created successfully. configASSERT() is described in section
    11.2. */

    configASSERT( xRecursiveMutex );

    /* As per most tasks, this task is implemented as an infinite loop. */
}
```

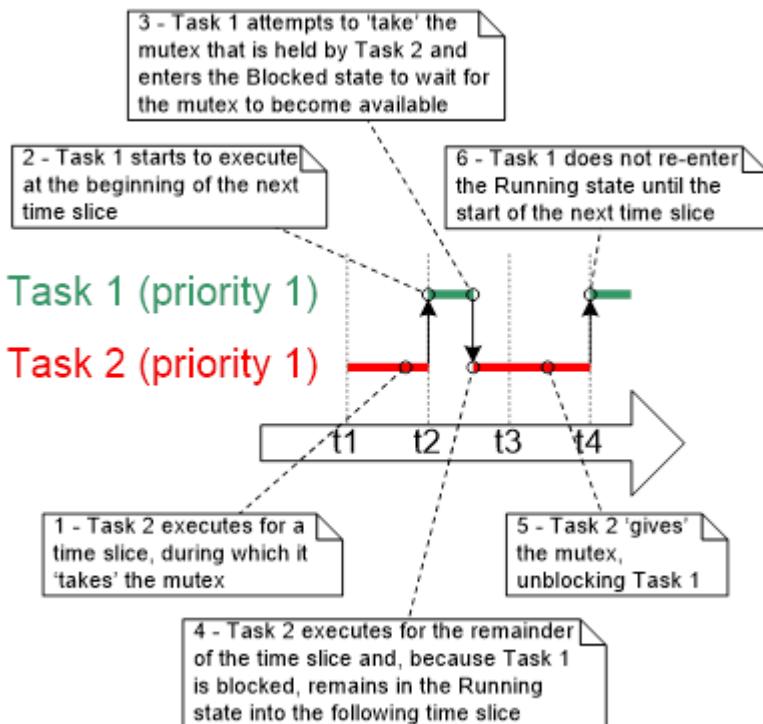
```
for( ; ; )  
{  
    /* ... */  
  
    /* Take the recursive mutex. */  
  
    if( xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms ) == pdPASS )  
    {  
  
        /* The recursive mutex was successfully obtained. The task can now access the  
        resource the mutex is protecting. At this point the recursive call count (which is the  
        number of nested calls to xSemaphoreTakeRecursive()) is 1 because the recursive mutex has  
        only been taken once. */  
  
        /* While it already holds the recursive mutex, the task takes the mutex  
        again. In a real application, this is only likely to occur inside a subfunction called  
        by this task because there is no practical reason to knowingly take the same mutex  
        more than once. The calling task is already the mutex holder, so the second call to  
        xSemaphoreTakeRecursive() does nothing more than increment the recursive call count to 2.  
        */  
  
        xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms );  
  
        /* ... */  
  
        /* The task returns the mutex after it has finished accessing the resource the  
        mutex is protecting. At this point the recursive call count is 2, so the first call to  
        xSemaphoreGiveRecursive() does not return the mutex. Instead, it simply decrements the  
        recursive call count back to 1. */  
  
        xSemaphoreGiveRecursive( xRecursiveMutex );  
  
        /* The next call to xSemaphoreGiveRecursive() decrements the recursive call  
        count to 0, so this time the recursive mutex is returned.*/  
  
        xSemaphoreGiveRecursive( xRecursiveMutex );  
  
        /* Now one call to xSemaphoreGiveRecursive() has been executed for every  
        proceeding call to xSemaphoreTakeRecursive(), so the task is no longer the mutex holder.  
        */  
    }  
}  
}
```

## 互斥锁和任务计划

如果优先级不同的两个任务使用相同的互斥锁，FreeRTOS 计划策略会清楚确定这两个任务的执行顺序。可以运行的优先级最高的任务将选为进入“运行”状态的任务。例如，如果高优先级任务处于“被阻止”状态，以等待低优先级任务持有的互斥锁，则只要低优先级任务返回互斥锁，高优先级任务就会在低优先级任务之前抢先执行。然后，高优先级任务将变为互斥锁持有者。[优先级继承 \(p. 151\)](#)部分已介绍了这种情况。

当任务的优先级相同时，通常会错误地假定任务的执行顺序。如果 Task 1 和 Task 2 的优先级相同，Task 1 处于“被阻止”状态，等待 Task 2 持有的互斥锁，则当 Task 2 释放互斥锁时，Task 1 不会在 Task 2 之前抢先执行。而 Task 2 将保持“运行”状态。Task 1 只是从“被阻止”状态变为“准备就绪”状态。

在下图中，垂直线段标记发生时钟中断的时间。



可以看到，FreeRTOS 计划程序不会在互斥锁可用后立即使 Task 1 成为处于“运行”状态的任务，原因如下：

1. Task 1 和 Task 2 的优先级相同，因此，除非 Task 2 进入“被阻止”状态，否则在发生下一个时钟中断之前，不会切换到 Task 1（假设在 FreeRTOSConfig.h 中 configUSE\_TIME\_SLICING 已设置为 1）。
2. 如果任务在紧密循环中使用互斥锁，并且每当该任务释放互斥锁时均发生上下文切换，则该任务保持“运行”状态的时间很短。如果两个或多个任务在紧密循环中使用同一互斥锁，则在任务之间快速切换会浪费处理器时间。

如果在紧密循环中有多个任务使用一个互斥锁，并且使用该互斥锁的任务具有相同的优先级，则需确保任务的处理时间大致相同。上图是当以下代码中所示的任务的两个实例在创建时具有相同优先级时可能出现的执行顺序。

```
/* The implementation of a task that uses a mutex in a tight loop. The task creates a text string in a local buffer, and then writes the string to a display. Access to the display is protected by a mutex. */

void vATask( void *pvParameter )
{
    extern SemaphoreHandle_t xMutex;

    char cTextBuffer[ 128 ];

    for( ;; )

    {
        /* Generate the text string. This is a fast operation. */
    }
}
```

```

vGenerateTextInALocalBuffer( cTextBuffer );

/* Obtain the mutex that is protecting access to the display. */

xSemaphoreTake( xMutex, portMAX_DELAY );

/* Write the generated text to the display. This is a slow operation. */

vCopyTextToFrameBuffer( cTextBuffer );

/* The text has been written to the display, so return the mutex. */

xSemaphoreGive( xMutex );

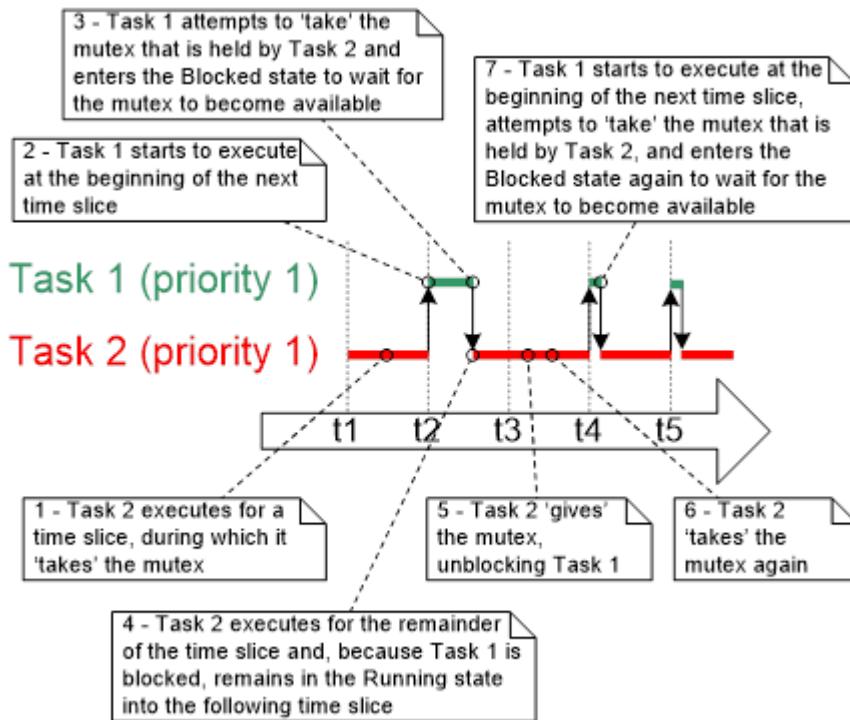
}

}

```

代码中的注释指明，创建字符串是快速操作，而更新显示是慢速操作。因此，如果在互斥锁被持有时更新显示，任务将在其大部分运行时间内持有互斥锁。

在下图中，垂直线段标记发生时钟中断的时间。



此图的步骤 7 中，Task 1 重新进入“被阻止”状态。这发生在 xSemaphoreTake() API 函数内。

在时间片开始与 Task 2 不是互斥锁持有者的短暂停时间段之一重合之前，Task 1 无法获取互斥锁。

通过在 xSemaphoreGive() 调用后添加 taskYIELD() 调用，可以避免这种情况。下面的代码对此进行说明，在任务持有互斥锁时如果时钟计数更改，则调用 taskYIELD()。该代码可确保在循环中使用互斥锁的任务会得到更等量的处理时间，同时确保不会因在任务之间切换过快而浪费处理时间。

```

void vFunction( void *pvParameter )

{

```

```
extern SemaphoreHandle_t xMutex;

char cTextBuffer[ 128 ];

TickType_t xTimeAtWhichMutexWasTaken;

for( ;; )

{

    /* Generate the text string. This is a fast operation. */

    vGenerateTextInALocalBuffer( cTextBuffer );

    /* Obtain the mutex that is protecting access to the display. */

    xSemaphoreTake( xMutex, portMAX_DELAY );

    /* Record the time at which the mutex was taken. */

    xTimeAtWhichMutexWasTaken = xTaskGetTickCount();

    /* Write the generated text to the display. This is a slow operation. */

    vCopyTextToFrameBuffer( cTextBuffer );

    /* The text has been written to the display, so return the mutex. */

    xSemaphoreGive( xMutex );

    /* If taskYIELD() was called on each iteration, then this task would only ever
       remain in the Running state for a short period of time, and processing time would be
       wasted by rapidly switching between tasks. Therefore, only call taskYIELD() if the tick
       count changed while the mutex was held. */

    if( xTaskGetTickCount() != xTimeAtWhichMutexWasTaken )

    {

        taskYIELD();

    }

}

}
```

## 网关守卫任务

网关守卫任务提供了实现互斥的明确方法，并且没有优先级反转或死锁的风险。

网关守卫任务是对资源拥有单独所有权的任务。只有网关守卫任务才能直接访问资源。需要访问该资源的任何其他任务只能使用网关守卫服务间接访问资源。

## 将 vPrintString() 重新编写为使用网关守卫任务（示例 21）

此示例提供 vPrintString() 的另一个替代实现。这一次，网关守卫任务用于管理对标准输出的访问。当任务需要向标准输出写入消息时，它不直接调用 print 函数。而是向网关守卫发送消息。

网关守卫任务使用 FreeRTOS 队列对标准输出的访问执行序列化。任务的内部实现不必考虑互斥，因为这是唯一允许直接访问标准输出的任务。

网关守卫任务的大部分时间处于“被阻止”状态，等待消息到达队列。当消息到达时，网关守卫只将消息写入标准输出，然后返回到“被阻止”状态，等待下一条消息。

中断可以发送到队列，因此，中断服务例程也可以安全地使用网关守卫服务将消息写入终端。在此示例中，使用时钟挂钩函数每 200 个时钟周期输出一条消息。

时钟挂钩（或时钟回调）是内核在每个时钟中断期间调用的函数。要使用时钟挂钩函数，请执行以下操作：

1. 在 FreeRTOSConfig.h 中，将 configUSE\_TICK\_HOOK 设置为 1。
2. 使用此处显示的确切函数名称和原型提供挂钩函数的实现。

```
void vApplicationTickHook( void );
```

时钟挂钩函数在时钟中断的上下文中执行，因此必须非常短。它们只能使用适量的堆栈空间，并且不能调用不以“FromISR()”结尾的任何 FreeRTOS API 函数。

下面是网关守卫任务的实现。计划程序将始终在时钟挂钩函数后立即执行，因此从时钟挂钩调用的中断安全 FreeRTOS API 函数不需要使用 pxHigherPriorityTaskWoken 参数，该参数可设置为 NULL。

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
    char *pcMessageToPrint;

    /* This is the only task that is allowed to write to standard out. Any other task
       wanting to write a string to the output does not access standard out directly, but
       instead sends the string to this task. Because this is the only task that accesses
       standard out, there are no mutual exclusion or serialization issues to consider within the
       implementation of the task itself. */

    for( ;; )
    {
        /* Wait for a message to arrive. An indefinite block time is specified so there is
           no need to check the return value. The function will return only when a message has been
           successfully received. */

        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

        /* Output the received string. */

        printf( "%s", pcMessageToPrint );

        fflush( stdout );

        /* Loop back to wait for the next message. */
    }
}
```

下面是用于写入队列的任务。和之前一样，创建任务的两个单独实例，并使用任务参数将任务写入队列的字符串传递到任务中。

```
static void prvPrintTask( void *pvParameters )
```

```
{  
    int iIndexToString;  
  
    const TickType_t xMaxBlockTimeTicks = 0x20;  
  
    /* Two instances of this task are created. The task parameter is used to pass an index  
    into an array of strings into the task. Cast this to the required type. */  
  
    iIndexToString = ( int ) pvParameters;  
  
    for( ;; )  
  
    {  
  
        /* Print out the string, not directly, but by passing a pointer to the string to  
        the gatekeeper task through a queue. The queue is created before the scheduler is started  
        so will already exist by the time this task executes for the first time. A block time is  
        not specified because there should always be space in the queue. */  
  
        xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[ iIndexToString ] ), 0 );  
  
        /* Wait a pseudo random time. Note that rand() is not necessarily reentrant, but in  
        this case it does not really matter because the code does not care what value is returned.  
        In a more secure application, a version of rand() that is known to be reentrant should be  
        used or calls to rand() should be protected using a critical section. */  
  
        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );  
  
    }  
}
```

时钟挂钩函数对函数的调用次数计数，每当计数达到 200 时都向网关守卫任务发送消息。时钟挂钩写入队列前部，任务写入队列后部，这只是为了进行演示。下面是时钟挂钩实现。

```
void vApplicationTickHook( void )  
{  
    static int iCount = 0;  
  
    /* Print out a message every 200 ticks. The message is not written out directly, but  
    sent to the gatekeeper task. */  
  
    iCount++;  
  
    if( iCount >= 200 )  
  
    {  
  
        /* Because xQueueSendToFrontFromISR() is being called from the tick hook, it is not  
        necessary to use the xHigherPriorityTaskWoken parameter (the third parameter), and the  
        parameter is set to NULL. */  
  
        xQueueSendToFrontFromISR( xPrintQueue, &( pcStringsToPrint[ 2 ] ), NULL );  
  
        /* Reset the count ready to print out the string again in 200 ticks time. */  
  
        iCount = 0;  
  
    }  
}
```

和通常情况一样，main() 创建运行示例所需的队列和任务，然后启动计划程序。下面是 main() 的实现。

```
/* Define the strings that the tasks and interrupt will print out via the gatekeeper. */

static char *pcStringsToPrint[] =
{
    "Task 1 *****\r\n", "Task 2\r\n-----\r\n", "Message printed from the tick
hook interrupt #####\r\n"
};

/*-----*/
/* Declare a variable of type QueueHandle_t. The queue is used to send messages from the
print tasks and the tick interrupt to the gatekeeper task. */

QueueHandle_t xPrintQueue;
/*-----*/
int main( void )
{
    /* Before a queue is used it must be explicitly created. The queue is created to hold a
maximum of 5 character pointers. */

    xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

    /* Check the queue was created successfully. */
    if( xPrintQueue != NULL )
    {
        /* Create two instances of the tasks that send messages to the gatekeeper. The
index to the string the task uses is passed to the task through the task parameter (the
4th parameter to xTaskCreate()). The tasks are created at different priorities, so the
higher priority task will occasionally preempt the lower priority task. */

        xTaskCreate( prvPrintTask, "Print1", 1000, ( void * ) 0, 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

        /* Create the gatekeeper task. This is the only task that is permitted to directly
access standard out. */

        xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL, 0, NULL
    );
    }

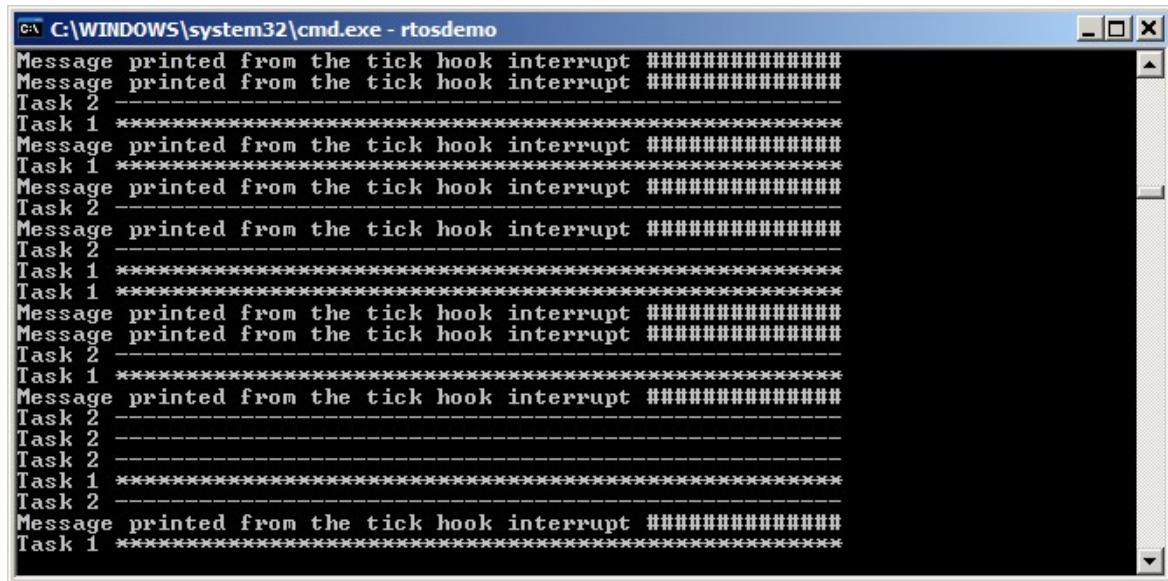
    /* Start the scheduler so the created tasks start executing. */

    vTaskStartScheduler();
}

/* If all is well, then main() will never reach here because the scheduler will
now be running the tasks. If main() does reach here, then it is likely that there was
insufficient heap memory available for the idle task to be created.*/
for( ;; );
```

}

下面是输出的内容。可以看到，源自任务的字符串和源自中断的字符串都正确输出，没有任何损坏。



```
Message printed from the tick hook interrupt #####
Message printed from the tick hook interrupt #####
Task 2 -----
Task 1 ****
Message printed from the tick hook interrupt #####
Task 1 ****
Message printed from the tick hook interrupt #####
Task 2 -----
Task 1 ****
Message printed from the tick hook interrupt #####
Task 2 -----
Task 1 ****
Message printed from the tick hook interrupt #####
Message printed from the tick hook interrupt #####
Task 2 -----
Task 1 ****
Message printed from the tick hook interrupt #####
Task 2 -----
Task 2 -----
Task 2 -----
Task 1 ****
Task 2 -----
Message printed from the tick hook interrupt #####
Task 1 ****
```

为网关守卫任务分配的优先级比输出任务的优先级低，因此发送给网关守卫的消息将保留在队列中，直到两个输出任务都处于“被阻止”状态。在某些情况下，应该为网关守卫分配较高的优先级，以便立即处理消息。这种做法的代价是网关守卫会延迟较低优先级任务，直至它完成对受保护资源的访问。

# 事件组

本节内容：

- 事件组的实际用途。
- 事件组相对于其他 FreeRTOS 功能的优缺点。
- 如何设置事件组中的位。
- 如何在“被阻止”状态中等待在事件组中设置位。
- 如何使用事件组来同步一组任务。

事件组是另一个 FreeRTOS 功能，允许将事件传递给任务与队列和信号灯不同，事件组：

- 允许一个任务在“被阻止”状态等待一个或多个事件的组合发生。
- 当事件发生时，取消阻止等待相同事件或事件组合的所有任务。

由于事件组具有这些独特的属性，因此可用来同步多个任务、将事件广播到多个任务、允许任务在“被阻止”状态等待一组事件中的任一事件发生，以及任务在“被阻止”状态等待多个操作完成。

事件组还可减少应用程序使用的 RAM，因为该功能通常可以将许多二进制信号灯替换为单个事件组。

事件组功能是可选的。要包括事件组功能，请将 FreeRTOS 源文件 event\_groups.c 构建为项目的一部分。

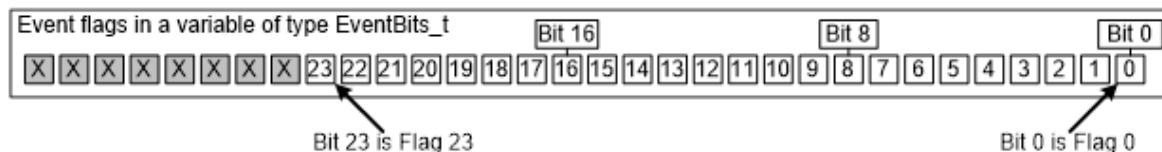
## 事件组的特性

### 事件组、事件标记和事件位

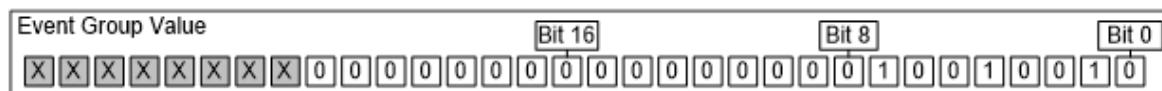
事件标记是一个布尔值（1 或 0），用于指示是否已有事件发生。事件组是一组事件标记。

事件标记只能为 1 或 0，这样，就可以在单个位中存储其状态，并且可以将一个事件组中的所有事件标记的状态存储在单个变量中。事件组中每个事件标记的状态由类型为 EventBits\_t 的变量中的单个位表示。出于此原因，事件标记也称为事件位。如果某个位在 EventBits\_t 变量中设置为 1，则发生了由该位表示的事件。如果某个位在 EventBits\_t 变量中设置为 0，则尚未发生由该位表示的事件。

下图显示如何将各个事件标记映射到类型为 EventBits\_t 的变量中的各个位。



例如，如果事件组的值为 0x92（二进制 1001 0010），则只设置了事件位 1、4 和 7，因此，只发生了由位 1、4 和 7 表示的事件。下图显示一个类型为 EventBits\_t 的变量，此变量已设置事件位 1、4 和 7，而所有其他事件位均为空，该变量为事件组提供的值为 0x92。



应用程序编写者可定义事件组中各个位的含义。例如，应用程序编写者可能会创建一个事件组，然后：

- 将事件组中的位 0 定义为表示已从网络收到一条消息。
- 将事件组中的位 1 定义为表示已准备就绪可向网络发送一条消息。
- 将事件组中的位 2 定义为表示中止当前网络连接。

## 关于 EventBits\_t 数据类型的更多信息

事件组中事件位的数量取决于 FreeRTOSConfig.h 中的 configUSE\_16\_BIT\_TICKS 编译时间配置常量。此常量配置用于保留 RTOS 计时计数的类型，因此它似乎与事件组功能无关。此常量对 EventBits\_t 类型的影响是 FreeRTOS 内部实现的结果；它是很有用的，因为仅当 FreeRTOS 在处理 16 位类型可以比处理 32 位类型更高效的架构上执行时，configUSE\_16\_BIT\_TICKS 才应设置为 1。

- 如果 configUSE\_16\_BIT\_TICKS 设置为 1，则每个事件组包含 8 个可用的事件位。
- 如果 configUSE\_16\_BIT\_TICKS 设置为 0，则每个事件组包含 24 个可用的事件位。

## 由多个任务访问

事件组本身是可以由知道其存在的任何任务或 ISR 访问的对象。任意数量的任务可以在同一事件组中设置位，并且任意数量的任务可以从同一事件组读取位。

## 使用事件组的实用示例

FreeRTOS+TCP TCP/IP 堆栈的实现提供了一个实用实例，用于演示如何使用事件组同时简化设计和将资源占用降至最低。

TCP 套接字必须响应许多不同的事件，包括接受事件、绑定事件、读取事件以及关闭事件。套接字在任何给定时间预期会收到的事件取决于套接字的状态。例如，如果套接字已创建但尚未绑定到一个地址，则它可能预期收到绑定事件而不是读取事件。（如果它没有地址，将无法读取数据。）

FreeRTOS+TCP 套接字的状态保留在一个名为 FreeRTOS\_Socket\_t 的结构中。该结构包含一个事件组，该事件组为套接字必须处理的每个事件定义一个事件位。阻止以等待某个事件或一组事件的 FreeRTOS+TCP API 调用只阻止此事件组。

事件组还包含一个“中止”位，这允许 TCP 连接中止，而无论套接字此时正在等待哪个事件。

## 使用事件组管理事件

### xEventGroupCreate() API 函数

FreeRTOS V9.0.0 还包含 xEventGroupCreateStatic() 函数，该函数分配在编译时静态创建事件组所需的内存。事件组必须先显式创建，然后才能使用。

使用类型为 EventGroupHandle\_t 的变量引用事件组。xEventGroupCreate() API 函数用于创建事件组。它将返回 EventGroupHandle\_t 来引用它创建的事件组。

下面是 xEventGroupCreate() API 函数原型。

```
EventGroupHandle_t xEventGroupCreate( void );
```

下表列出了 xEventGroupCreate() 返回值。

参数名称	描述
------	----

返回值	<p>如果返回 NULL，则表示无法创建事件组，因为没有足够的堆内存可供 FreeRTOS 分配事件组数据结构。有关更多信息，请参阅<a href="#">堆内存管理 (p. 12)</a>。</p> <p>如果返回非 NULL 值，表示事件组已成功创建。返回的值应存储为所创建的事件组的句柄。</p>
-----	---

## xEventGroupSetBits() API 函数

xEventGroupSetBits() API 函数在事件组中设置一个或多个位。它通常用于向任务通知由所设置的一个或多个位表示的事件已发生。

注意：请勿从中断服务例程中调用 xEventGroupSetBits()。而应使用中断安全版本 xEventGroupSetBitsFromISR()。

下面是 xEventGroupSetBits() API 函数原型。

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

下表列出了 xEventGroupSetBits() 参数和返回值。

参数名称	描述
xEventGroup	要在其中设置位的事件组的句柄。事件组句柄将已从用来创建事件组的 xEventGroupCreate() 调用返回。
uxBitsToSet	指定在事件组中将一个或多个事件位设置为 1 的掩码。事件组的值由按位 ORing 将事件组的现有值更新为在 uxBitsToSet 中传递的值。  例如，将 uxBitsToSet 设置为 0x04 (二进制 0100) 将导致设置事件组中的事件位 3 (如果尚未设置)，同时让事件组中的所有其他事件位保持不变。
返回的值	调用 xEventGroupSetBits() 时返回的事件组的值。返回的值不一定会设置由 uxBitsToSet 指定的位，因为这些位可能已由其他任务再次清除。

## xEventGroupSetBitsFromISR() API 函数

xEventGroupSetBitsFromISR() 是 xEventGroupSetBits() 的中断安全版本。

提供信号灯是一个确定性操作，因为事先知道提供信号灯可能导致最多一个任务离开“被阻止”状态。设置事件组中的位并不是确定性操作，因为当在某个事件组中设置位时，事先不知道有多少个任务将离开“被阻止”状态。

FreeRTOS 设计和实现标准不允许在中断服务例程内部或在禁用中断时执行非确定性操作。因此，xEventGroupSetBitsFromISR() 不会在中断服务例程内部直接设置事件位，而是将操作委托给 RTOS 守护程序任务。

下面是 xEventGroupSetBitsFromISR() API 函数原型。

```
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup, const EventBits_t
uxBitsToSet, BaseType_t *pxHigherPriorityTaskWoken );
```

下表列出了 xEventGroupSetBitsFromISR() 参数和返回值。

#### xEventGroup

要在其中设置位的事件组的句柄。事件组句柄将已从用来创建事件组的 xEventGroupCreate() 调用返回。

#### uxBitsToSet

指定在事件组中将一个或多个事件位设置为 1 的位掩码。事件组的值由按位 ORing 将事件组的现有值更新为传递给 uxBitsToSet 的值。例如，将 uxBitsToSet 设置为 0x05 (二进制 0101) 将导致设置事件组中的事件位 3 和事件位 0 (如果尚未设置)，同时让事件组中的所有其他事件位保持不变。

#### pxHigherPriorityTaskWoken

xEventGroupSetBitsFromISR() 不会在中断服务例程内部直接设置事件位，而是通过在计时器命令队列上发送一个命令，将操作委托给 RTOS 守护程序任务。如果守护程序任务处于“被阻止”状态以等待数据在计时器命令队列中变为可用，则写入计数器命令队列将导致守护程序任务离开“被阻止”状态。如果守护程序任务的优先级高于当前正在执行的任务（被中断的任务）的优先级，则 xEventGroupSetBitsFromISR() 会在内部将 pxHigherPriorityTaskWoken 设置为 pdTRUE。

如果 xEventGroupSetBitsFromISR() 将此值设置为 pdTRUE，则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守护程序任务，因为守护程序任务将是优先级最高的“准备就绪”状态任务。

可能的返回值有两个：

仅当数据成功发送到计时器命令队列时，才返回 pdPASS。

如果因为计时器命令队列已满，导致“设置位”命令无法写入此队列，则返回 pdFALSE。

## xEventGroupWaitBits() API 函数

xEventGroupWaitBits() API 函数允许任务读取事件组的值，并（可选）让任务在“被阻止”状态等待事件组中的一个或多个事件位被设置（如果尚未设置）。

下面是 xEventGroupWaitBits() API 函数原型。

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

取消阻止条件是计划程序用来确定任务是否将进入“被阻止”状态以及何时将离开“被阻止”状态的条件。取消阻止条件由 uxBitsToWaitFor 和 xWaitForAllBits 参数值的组合指定：

- uxBitsToWaitFor，指定事件组中要测试的事件位。
- xWaitForAllBits，指定是使用按位 OR 测试还是按位 AND 测试。

如果在调用 xEventGroupWaitBits() 时满足任务的取消阻止条件，则任务将不会进入“被阻止”状态。

下表提供了一些将导致任务进入或退出“被阻止”状态的条件的示例。它仅显示事件组和 uxBitsToWaitFor 值的最低有效的四个二进制位。这两个值的其他位均被视为零。

现有事件组值	uxBitsToWaitFor 值	xWaitForAllBits 值	产生的行为
0000	0101	pdFALSE	发出调用的任务将进入“被阻止”状态，因为在

			事件组中位 0 或位 2 均未设置；当在事件组中设置了位 0 或位 2 时，任务将离开“被阻止”状态。
0100	0101	pdTRUE	发出调用的任务将进入“被阻止”状态，因为在事件组中位 0 和位 2 未同时设置；当在事件组中同时设置了位 0 和位 2 时，任务将离开“被阻止”状态。
0100	0110	pdFALSE	发出调用的任务将不会进入“被阻止”状态，因为 xWaitForAllBits 为 pdFALSE，并且已在事件组中设置了由 uxBitsToWaitFor 指定的两个位之一。
0100	0110	pdTRUE	发出调用的任务将进入“被阻止”状态，因为 xWaitForAllBits 为 pdTRUE，并且只在事件组中设置了由 uxBitsToWaitFor 指定的两个位之一。当在事件组中同时设置了位 2 和位 3 时，任务将离开“被阻止”状态。

发出调用的任务使用 uxBitsToWaitFor 参数指定要测试的位。发出调用的任务在其取消阻止条件得到满足后，最可能需要将这些位清回为零。可以使用 xEventGroupClearBits() API 函数清除事件位，但在以下情况下，使用此函数手动清除事件位将导致应用程序代码中出现争用情况：

- 多个任务使用同一个事件组。
- 位是通过不同的任务或 ISR 在事件组中设置的。

提供 xClearOnExit 参数以避免这些潜在的争用情况。如果 xClearOnExit 设置为 pdTRUE，事件位的测试和清除对发出调用的任务而言是一个原子操作（其他任务或中断无法进行中断）。

下表列出了 xEventGroupWaitBits() 参数和返回值。

参数名称	描述
xEventGroup	包含正读取的事件位的事件组的句柄。事件组句柄将已从用来创建事件组的 xEventGroupCreate() 调用返回。
uxBitsToWaitFor	指定事件组中要测试的一个或多个事件位的位掩码。  例如，如果发出调用的任务希望等待事件组中的事件位 0 和/或事件位 2 变为已设置状态，则将

	uxBitsToWaitFor 设置为 0x05 (二进制 0101)。有关进一步的示例，请参阅表 45。
xClearOnExit	<p>如果发出调用的任务的取消阻止条件得到满足，并且 xClearOnExit 设置为 pdTRUE，则由 uxBitsToWaitFor 指定的事件位将在事件组中先清回为 0，然后发出调用的任务才会退出 xEventGroupWaitBits() API 函数。</p> <p>如果 xClearOnExit 设置为 pdFALSE，则 xEventGroupWaitBits() API 函数不会修改事件组中事件位的状态。</p>
xWaitForAllBits	<p>uxBitsToWaitFor 参数指定事件组中要测试的事件位。xWaitForAllBits 指定应从“被阻止”状态删除发出调用的任务的条件：是在设置了由 uxBitsToWaitFor 参数指定的一个或多个事件位时删除，还是仅当设置了由 uxBitsToWaitFor 参数指定的所有事件位时才删除。</p> <p>如果 xWaitForAllBits 设置为 pdFALSE，则进入“被阻止”状态等待其取消阻止条件得到满足的任务在以下情况下将离开“被阻止”状态：当由 uxBitsToWaitFor 指定的任何位已得到设置时（或者由 xTicksToWait 参数指定的超时到期时）。</p> <p>如果 xWaitForAllBits 设置为 pdTRUE，则进入“被阻止”状态等待其取消阻止条件得到满足的任务在以下情况下将离开“被阻止”状态：仅当 uxBitsToWaitFor 指定的所有位都已得到设置时（或者由 xTicksToWait 参数指定的超时到期时）。</p> <p>有关示例，请参阅上表。</p>
xTicksToWait	<p>任务保留在“被阻止”状态以等待其取消阻止条件得到满足的最大时间量。</p> <p>如果 xTicksToWait 为零或在调用 xEventGroupWaitBits() 时取消阻止条件已经得到满足，则 xEventGroupWaitBits() 将立即返回。</p> <p>阻止时间以计时周期指定，因此它表示的绝对时间取决于计时频率。宏 pdMS_TO_TICKS() 可用于将毫秒为单位指定的时间转换为以计时周期指定的时间。</p> <p>如果 FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 设置为 1，将 xTicksToWait 设置为 portMAX_DELAY 将导致任务无限期等待（无超时）。</p>

返回的值

如果由于发出调用的任务的取消阻止条件得到满足而导致 xEventGroupWaitBits() 返回，则返回的值是发出调用的任务的取消阻止条件得到满足时事件组的值（当 xClearOnExit 为 pdTRUE 时，在任何位被自动清除之前）。在这种情况下，返回的值也将满足取消阻止条件。

如果因为由 xTicksToWait 参数指定的阻止时间过期而导致 xEventGroupWaitBits() 返回，则返回的值是阻止时间过期时事件组的值。在这种情况下，返回的值不满足取消阻止条件。

## 试用事件组 (示例 22 )

此示例演示如何：

- 创建事件组。
- 通过 ISR 设置事件组中的位。
- 通过任务设置事件组中的位。
- 阻止事件组。

首先通过将 xWaitForAllBits 设置为 pdFALSE 来执行此示例，然后通过将 xWaitForAllBits 设置为 pdTRUE 执行此示例，以演示 xEventGroupWaitBits() xWaitForAllBits 参数的作用。

通过任务设置事件位 0 和事件位 1。事件位 2 通过 ISR 设置。使用 #define 语句为这三个位提供描述性名称，如下所示。

```
/* Definitions for the event bits in the event group. */

#define mainFIRST_TASK_BIT ( 1UL << 0UL ) /* Event bit 0, which is set by a task. */

#define mainSECOND_TASK_BIT ( 1UL << 1UL ) /* Event bit 1, which is set by a task. */

#define mainISR_BIT ( 1UL << 2UL ) /* Event bit 2, which is set by an ISR. */
```

以下代码显示如何实现设置事件位 0 和事件位 1 的任务。它位于一个循环中，反复设置一个位，然后设置另一个位，每个 xEventGroupSetBits() 调用之间的延迟为 200 毫秒。在将每个位设置为允许在控制台中看到执行顺序之前，将输出一个字符串。

```
static void vEventBitSettingTask( void *pvParameters )

{
    const TickType_t xDelay200ms = pdMS_TO_TICKS( 200UL ), xDontBlock = 0;
    for( ;; )
    {
        /* Delay for a short while before starting the next loop. */
        vTaskDelay( xDelay200ms );
        /* Print out a message to say event bit 0 is about to be set by the task, and then
         * set event bit 0. */
        vPrintString( "Bit setting task -\t about to set bit 0.\r\n" );
    }
}
```

```
xEventGroupSetBits( xEventGroup, mainFIRST_TASK_BIT );

/* Delay for a short while before setting the other bit. */

vTaskDelay( xDelay200ms );

/* Print out a message to say event bit 1 is about to be set by the task, and then
set event bit 1. */

vPrintString( "Bit setting task -\t about to set bit 1.\r\n" );

xEventGroupSetBits( xEventGroup, mainSECOND_TASK_BIT );

}

}
```

以下代码显示如何实现设置事件组中位 2 的中断服务例程。再一次，在将此位设置为允许在控制台中看到执行顺序之前，将输出一个字符串。在这种情况下，因为不应在中断服务例程中直接执行控制台输出，所以使用 xTimerPendFunctionCallFromISR() 在 RTOS 守护程序任务的上下文中执行输出。

如前面示例所示，中断服务例程由一个简单的用于强制软件中断的定期任务触发。在本示例中，每 500 毫秒生成一次中断。

```
static uint32_t ulEventBitSettingISR( void )

{

    /* The string is not printed within the interrupt service routine, but is instead
    sent to the RTOS daemon task for printing. It is therefore declared static to ensure the
    compiler does not allocate the string on the stack of the ISR because the ISR's stack
    frame will not exist when the string is printed from the daemon task. */

    static const char *pcString = "Bit setting ISR -\t about to set bit 2.\r\n";

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Print out a message to say bit 2 is about to be set. Messages cannot be printed from
    an ISR, so defer the actual output to the RTOS daemon task by pending a function call to
    run in the context of the RTOS daemon task. */

    xTimerPendFunctionCallFromISR( vPrintStringFromDaemonTask, ( void * ) pcString, 0,
&xHigherPriorityTaskWoken );

    /* Set bit 2 in the event group. */

    xEventGroupSetBitsFromISR( xEventGroup, mainISR_BIT, &xHigherPriorityTaskWoken );

    /* xTimerPendFunctionCallFromISR() and xEventGroupSetBitsFromISR() both write to the
    timer command queue, and both used the same xHigherPriorityTaskWoken variable. If writing
    to the timer command queue resulted in the RTOS daemon task leaving the Blocked state,
    and if the priority of the RTOS daemon task is higher than the priority of the currently
    executing task (the task this interrupt interrupted), then xHigherPriorityTaskWoken
    will have been set to pdTRUE. xHigherPriorityTaskWoken is used as the parameter
    to portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken equals pdTRUE, then calling
    portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
    pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
    portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
    this function does not explicitly return a value. */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

以下代码显示如何实现调用 xEventGroupWaitBits() 以阻止事件组的任务。任务为事件组中设置的每个位输出一个字符串。

xEventGroupWaitBits() xClearOnExit 参数设置为 pdTRUE，因此导致 xEventGroupWaitBits() 调用返回的一个或多个事件位将在 xEventGroupWaitBits() 返回之前被自动清除。

```
static void vEventBitReadingTask( void *pvParameters )  
{  
  
    EventBits_t xEventGroupValue;  
  
    const EventBits_t xBitsToWaitFor = ( mainFIRST_TASK_BIT | mainSECOND_TASK_BIT |  
    mainISR_BIT );  
  
    for( ;; )  
  
    {  
  
        /* Block to wait for event bits to become set within the event group. */  
  
        xEventGroupValue = xEventGroupWaitBits( /* The event group to read. */  
        xEventGroup, /* Bits to test. */ xBitsToWaitFor, /* Clear bits on exit if the unblock  
        condition is met. */ pdTRUE, /* Don't wait for all bits. This parameter is set to pdTRUE  
        for the second execution. */ pdFALSE, /* Don't time out. */ portMAX_DELAY );  
  
        /* Print a message for each bit that was set. */  
  
        if( ( xEventGroupValue & mainFIRST_TASK_BIT ) != 0 )  
  
        {  
  
            vPrintString( "Bit reading task -\t Event bit 0 was set\r\n" );  
  
        }  
  
        if( ( xEventGroupValue & mainSECOND_TASK_BIT ) != 0 )  
  
        {  
  
            vPrintString( "Bit reading task -\t Event bit 1 was set\r\n" );  
  
        }  
  
        if( ( xEventGroupValue & mainISR_BIT ) != 0 )  
  
        {  
  
            vPrintString( "Bit reading task -\t Event bit 2 was set\r\n" );  
  
        }  
    }  
}
```

main() 函数在启动计划程序之前创建事件组和任务。下面的代码说明其实现过程。从事件组中进行读取的任务的优先级高于写入事件组的任务的优先级，可确保当每次满足读取任务的取消阻止条件时，读取任务优先于写入任务。

```
int main( void )  
{
```

```
/* Before an event group can be used it must first be created. */

xEventGroup = xEventGroupCreate();

/* Create the task that sets event bits in the event group. */

xTaskCreate( vEventBitSettingTask, "Bit Setter", 1000, NULL, 1, NULL );

/* Create the task that waits for event bits to get set in the event group. */

xTaskCreate( vEventBitReadingTask, "Bit Reader", 1000, NULL, 2, NULL );

/* Create the task that is used to periodically generate a software interrupt. */

xTaskCreate( vInterruptGenerator, "Int Gen", 1000, NULL, 3, NULL );

/* Install the handler for the software interrupt. The syntax required to do this is
depends on the FreeRTOS port being used. The syntax shown here can only be used with the
FreeRTOS Windows port, where such interrupts are only simulated. */

vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulEventBitSettingISR );

/* Start the scheduler so the created tasks start executing. */

vTaskStartScheduler();

/* The following line should never be reached. */

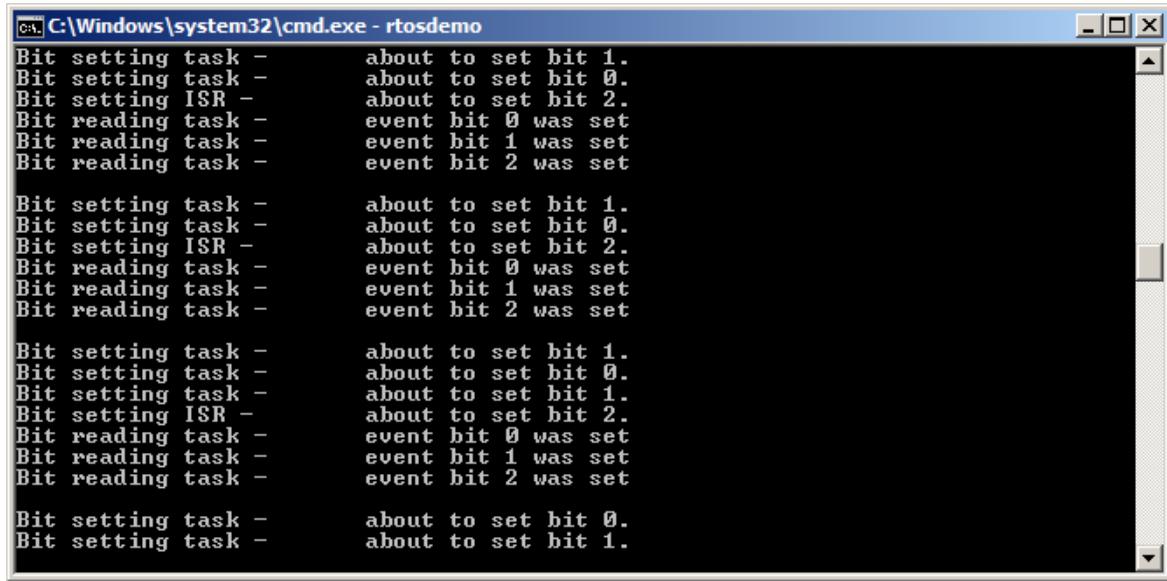
for( ; );

return 0;
}
```

下面是当 xEventGroupWaitBits() xWaitForAllBits 参数设置为 pdFALSE 时执行示例 22 所生成的输出。您将看到，因为 xEventGroupWaitBits() 调用中的 xWaitForAllBits 参数设置为 pdFALSE，所以，从事件组中进行读取的任务将离开“被阻止”状态，并在每次设置任何事件位时立即执行。

```
Bit setting task - about to set bit 1.
Bit reading task - event bit 1 was set
Bit setting task - about to set bit 0.
Bit reading task - event bit 0 was set
Bit setting task - about to set bit 1.
Bit reading task - event bit 1 was set
Bit setting ISR - about to set bit 2.
Bit reading task - event bit 2 was set
Bit setting task - about to set bit 0.
Bit reading task - event bit 0 was set
Bit setting task - about to set bit 1.
Bit reading task - event bit 1 was set
Bit setting ISR - about to set bit 2.
Bit reading task - event bit 2 was set
Bit setting task - about to set bit 0.
Bit reading task - event bit 0 was set
```

下面是当 xEventGroupWaitBits() xWaitForAllBits 参数设置为 pdTRUE 时执行此代码所生成的输出。您将看到，因为 xWaitForAllBits 参数已设置为 pdTRUE，从事件组中进行读取的任务仅当所有三个事件位都已设置后才会离开“被阻止”状态。



```
C:\Windows\system32\cmd.exe - rtosdemo
Bit setting task -      about to set bit 1.
Bit setting task -      about to set bit 0.
Bit setting ISR -      about to set bit 2.
Bit reading task -     event bit 0 was set
Bit reading task -     event bit 1 was set
Bit reading task -     event bit 2 was set

Bit setting task -      about to set bit 1.
Bit setting task -      about to set bit 0.
Bit setting ISR -      about to set bit 2.
Bit reading task -     event bit 0 was set
Bit reading task -     event bit 1 was set
Bit reading task -     event bit 2 was set

Bit setting task -      about to set bit 1.
Bit setting task -      about to set bit 0.
Bit setting task -      about to set bit 1.
Bit setting ISR -      about to set bit 2.
Bit reading task -     event bit 0 was set
Bit reading task -     event bit 1 was set
Bit reading task -     event bit 2 was set

Bit setting task -      about to set bit 0.
Bit setting task -      about to set bit 1.
```

## 使用事件组同步任务

有时，应用程序的设计需要将两个或更多任务彼此同步。例如，设想这样一个设计：任务 A 接收事件，然后将此事件需要的一些处理工作委派给三个其他任务，即任务 B、任务 C 和任务 D。如果在任务 B、C 和 D 全部完成处理前一个事件之前，任务 A 无法收到另一个事件，则所有这四个任务将需要彼此同步。每个任务的同步点将在该任务完成其处理之后，在每个其他任务完成同样处理之前无法继续。任务 A 仅当所有四个任务已达到其同步点后，才能收到另一个事件。

您可以在 FreeRTOS+TCP 演示项目中找到一个不太抽象的示例，它演示了需要进行这种类型的任务同步的情况。此演示在两个任务之间共享一个 TCP 套接字。一个任务将数据发送到套接字，另一个任务从同一套接字接收数据。（这是目前唯一一种可在任务之间共享单个 FreeRTOS+TCP 套接字的方法。）任一个任务关闭 TCP 套接字都是不安全的，除非它确定另一个任务不会再次尝试访问套接字。如果这两个任务的任何一个要关闭此套接字，则它必须向另一个任务通知其意图并等待后一个任务停止使用套接字，然后继续。

在这个场景中，将数据发送到套接字的任务希望关闭套接字，因为只有两个任务需要彼此同步，所以不是什么大问题。可以很容易地看出，如果还有其他任务执行的处理也依赖于打开的套接字，则此场景将变得更复杂且需要更多任务加入同步。

```
void SocketTxTask( void *pvParameters )
{
    xSocket_t xSocket;
    uint32_t ulTxCount = 0UL;
    for( ;; )
    {
        /* Create a new socket. This task will send to this socket, and another task
        will receive from this socket. */

        xSocket = FreeRTOS_socket( ... );
        /* Connect the socket. */
```

```
    FreeRTOS_connect( xSocket, ... );

    /* Use a queue to send the socket to the task that receives data. */
    xQueueSend( xSocketPassingQueue, &xSocket, portMAX_DELAY );

    /* Send 1000 messages to the socket before closing the socket. */
    for( ulTxCount = 0; ulTxCount < 1000; ulTxCount++ )

    {

        if( FreeRTOS_send( xSocket, ... ) < 0 )

        {

            /* Unexpected error - exit the loop, after which the socket
            will be closed. */

            break;

        }

    }

    /* Let the Rx task know the Tx task wants to close the socket. */

    TxTaskWantsToCloseSocket();

    /* This is the Tx task's synchronization point. The Tx task waits here
    for the Rx task to reach its synchronization point. The Rx task will only reach its
    synchronization point when it is no longer using the socket, and the socket can be closed
    safely. */

    xEventGroupSync( ... );

    /* Neither task is using the socket. Shut down the connection, and then close
    the socket. */

    FreeRTOS_shutdown( xSocket, ... );

    WaitForSocketToDisconnect();

    FreeRTOS_closesocket( xSocket );

}

}

/*-----*/
void SocketRxTask( void *pvParameters )
{
    xSocket_t xSocket;

    for( ;; )

    {

        /* Wait to receive a socket that was created and connected by the Tx task. */

        xQueueReceive( xSocketPassingQueue, &xSocket, portMAX_DELAY );
    }
}
```

```
    /* Keep receiving from the socket until the Tx task wants to close the socket.  
 */  
  
    while( TxTaskWantsToCloseSocket() == pdFALSE )  
  
    {  
  
        /* Receive then process data. */  
  
        FreeRTOS_recv( xSocket, ... );  
  
        ProcessReceivedData();  
  
    }  
  
    /* This is the Rx task's synchronization point. It reaches here only when it is  
 no longer using the socket, and it is therefore safe for the Tx task to close the socket.  
 */  
  
    xEventGroupSync( ... );  
  
}
```

上述伪代码显示了两个任务，它们彼此同步，以确保共享的 TCP 套接字在套接字被关闭之前不再由任一任务使用。

事件组可用于创建同步点：

- 必须参与同步的每个任务都会被分配一个在事件组内唯一的事件位。
- 每个任务在到达同步点时设置其自己的事件位。
- 设置自己的事件位后，每个任务会阻止事件组以等待表示所有其他同步任务的事件位也被设置。

在这种情况下，无法使用 xEventGroupSetBits() 和 xEventGroupWaitBits() API 函数。如果使用这些函数，则设置位的操作（以指示任务已到达其同步点）和测试位的操作（以确定其他同步任务是否已到达同步点）将作为两个单独的操作执行。要了解为何这会导致出现问题，请设想这样一个场景：任务 A、任务 B 和任务 C 尝试使用事件组进行同步，

1. 任务 A 和任务 B 已经达到同步点，因此在事件组中设置了其事件位。它们处于“被阻止”状态以等待任务 C 的事件位被设置。
2. 任务 C 到达同步点并使用 xEventGroupSetBits() 在事件组中设置其位。只要设置了任务 C 的位，任务 A 和任务 B 就会离开“被阻止”状态并清除所有三个事件位。
3. 然后，任务 C 调用 xEventGroupWaitBits() 以等待所有三个事件位被设置，但此时所有三个事件位已被清除，任务 A 和任务 B 已离开其各自的同步点，因此同步已失败。

要成功使用事件组创建一个同步点，必须将设置事件位和后续测试事件位作为一个无法中断的操作执行。提供 xEventGroupSync() API 函数的目的就在此。

## xEventGroupSync() API 函数

提供 xEventGroupSync() 的目的是允许两个或多个任务使用事件组彼此进行同步。该函数允许任务设置事件组中的一个或多个事件位，然后等待同一个事件组中的事件位组合得到设置，这两个步骤将作为一个无法中断的操作。

xEventGroupSync() uxBitsToWaitFor 参数指定发出调用的任务的取消阻止条件。如果 xEventGroupSync() 由于取消阻止条件已得到满足而返回，则 uxBitsToWaitFor 指定的事件位将在 xEventGroupSync() 返回之前清回为零。

下面是 xEventGroupSync() API 函数原型。

```
EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet,
    const EventBits_t uxBitsToWaitFor, TickType_t xTicksToWait );
```

下表列出了 xEventGroupSync() 参数和返回值。

参数名称	描述
xEventGroup	要在其中设置并测试事件位的事件组的句柄。事件组句柄将已从用来创建事件组的 xEventGroupCreate() 调用返回。
uxBitsToSet	指定在事件组中将一个或多个事件位设置为 1 的位掩码。事件组的值由按位 ORing 将事件组的现有值更新为在 uxBitsToSet 中传递的值。  例如，将 uxBitsToSet 设置为 0x04 (二进制 0100) 将导致设置事件位 3 (如果尚未设置)，同时让事件组中的所有其他事件位保持不变。
uxBitsToWaitFor	指定事件组中要测试的一个或多个事件位的位掩码。  例如，如果发出调用的任务希望等待事件组中的事件位 0、1 和 2 变为已设置状态，则将 uxBitsToWaitFor 设置为 0x07 (二进制 111)。
xTicksToWait	任务保留在“被阻止”状态以等待其取消阻止条件得到满足的最大时间量。  如果 xTicksToWait 为零或在调用 xEventGroupSync() 时取消阻止条件已经得到满足，则 xEventGroupSync() 将立即返回。  阻止时间以计时周期指定，因此它表示的绝对时间取决于计时频率。宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为以计时周期指定的时间。  如果 FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 设置为 1，将 xTicksToWait 设置为 portMAX_DELAY 将导致任务无限期等待 (无超时)。
返回的值	如果由于发出调用的任务的取消阻止条件得到满足而导致 xEventGroupSync() 返回，则返回的值是发出调用的任务的取消阻止条件得到满足时事件组的值 (在任何位被自动清回为零之前)。在这种情况下，返回的值也将满足发出调用的任务的取消阻止条件。  如果因为由 xTicksToWait 参数指定的阻止时间过期而导致 xEventGroupSync() 返回，则返回的值是阻止时间过期时事件组的值。在这种情况下，返回的值将不满足发出调用的任务的取消阻止条件。

## 同步任务 (示例 23 )

下面的代码演示如何同步单个任务实现的三个实例。任务参数用于将任务在调用 xEventGroupSync() 时设置的事件位传递到每个实例。

任务将在调用 xEventGroupSync() 之前输出一条消息，然后在调用 xEventGroupSync() 返回后再次输出一条消息。每条消息都包含一个时间戳。这样，就可以在生成的输出中观察到执行顺序。使用一个伪随机延迟防止所有任务同时到达同步点。

```
static void vSyncingTask( void *pvParameters )

{
    const TickType_t xMaxDelay = pdMS_TO_TICKS( 4000UL );
    const TickType_t xMinDelay = pdMS_TO_TICKS( 200UL );
    TickType_t xDelayTime;
    EventBits_t uxThisTasksSyncBit;

    const EventBits_t uxAllSyncBits = ( mainFIRST_TASK_BIT | mainSECOND_TASK_BIT |
    mainTHIRD_TASK_BIT );

    /* Three instances of this task are created - each task uses a different event bit in
     * the synchronization. The event bit to use is passed into each task instance using the task
     * parameter. Store it in the uxThisTasksSyncBit variable. */
    uxThisTasksSyncBit = ( EventBits_t ) pvParameters;

    for( ;; )
    {
        /* Simulate this task taking some time to perform an action by delaying for a
         * pseudo-random time. This prevents all three instances of this task from reaching the
         * synchronization point at the same time, and so allows the example's behavior to be
         * observed more easily. */

        xDelayTime = ( rand() % xMaxDelay ) + xMinDelay;

        vTaskDelay( xDelayTime );

        /* Print out a message to show this task has reached its synchronization point.
         * pcTaskGetTaskName() is an API function that returns the name assigned to the task when the
         * task was created. */
        vPrintTwoStrings( pcTaskGetTaskName( NULL ), "reached sync point" );

        /* Wait for all the tasks to have reached their respective synchronization points.
        */

        xEventGroupSync( /* The event group used to synchronize. */ xEventGroup, /*/
        The bit set by this task to indicate it has reached the synchronization point. */
        uxThisTasksSyncBit, /* The bits to wait for, one bit for each task taking part in the
        synchronization. */ uxAllSyncBits, /* Wait indefinitely for all three tasks to reach the
        synchronization point. */ portMAX_DELAY );

        /* Print out a message to show this task has passed its synchronization point. As
        an indefinite delay was used the following line will only be executed after all the tasks
        reached their respective synchronization points. */

        vPrintTwoStrings( pcTaskGetTaskName( NULL ), "exited sync point" );
    }
}
```

```
    }
}
```

main() 函数创建事件组和所有三个任务，然后启动计划程序。下面的代码说明其实现过程。

```
/* Definitions for the event bits in the event group. */

#define mainFIRST_TASK_BIT ( 1UL << 0UL ) /* Event bit 0, set by the first task. */
#define mainSECOND_TASK_BIT( 1UL << 1UL ) /* Event bit 1, set by the second task. */
#define mainTHIRD_TASK_BIT ( 1UL << 2UL ) /* Event bit 2, set by the third task. */

/* Declare the event group used to synchronize the three tasks. */

EventGroupHandle_t xEventGroup;

int main( void )

{
    /* Before an event group can be used it must first be created. */
    xEventGroup = xEventGroupCreate();

    /* Create three instances of the task. Each task is given a different name, which
    is later printed out to give a visual indication of which task is executing. The event bit
    to use when the task reaches its synchronization point is passed into the task using the
    task parameter. */

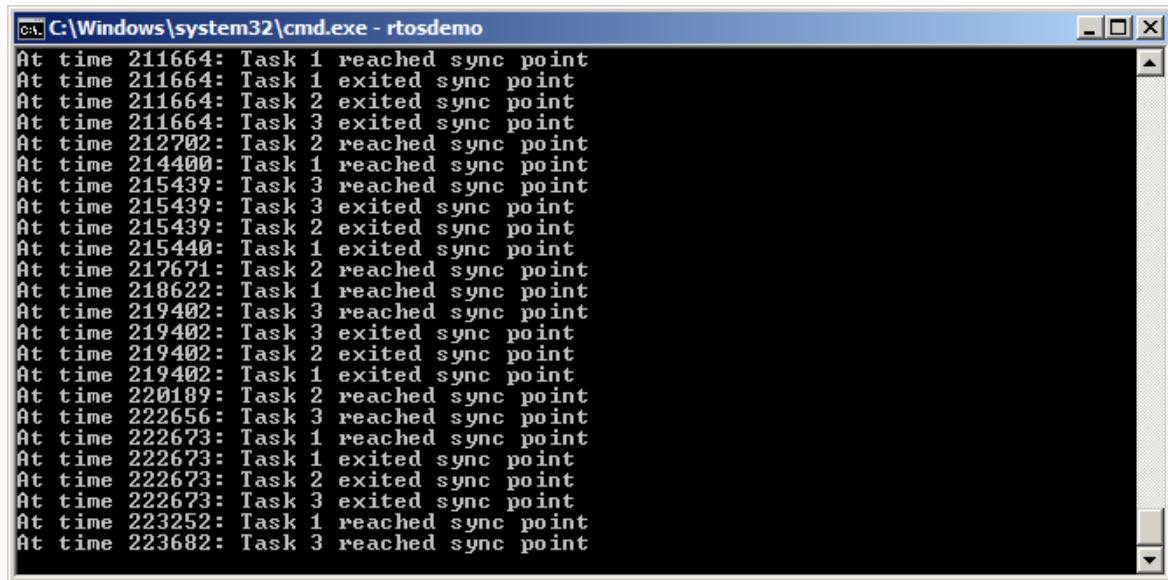
    xTaskCreate( vSyncingTask, "Task 1", 1000, mainFIRST_TASK_BIT, 1, NULL );
    xTaskCreate( vSyncingTask, "Task 2", 1000, mainSECOND_TASK_BIT, 1, NULL );
    xTaskCreate( vSyncingTask, "Task 3", 1000, mainTHIRD_TASK_BIT, 1, NULL );

    /* Start the scheduler so the created tasks start executing. */

    vTaskStartScheduler();

    /* As always, the following line should never be reached. */
    for( ;; );
    return 0;
}
```

执行示例 23 时生成的输出如下所示。您将看到，即使每个任务在不同的（伪随机）时间到达同步点，它们也会同时退出同步点（即最后一个任务到达同步点的时间）。图中显示在 FreeRTOS Windows 端口中运行的示例，但此示例未提供真正的实时行为（尤其是使用 Windows 系统调用输出到控制台时）。因此，会有一些计时变化。



The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe - rtosdemo'. The window contains a log of synchronization events for three tasks (Task 1, Task 2, Task 3) over time. The log entries are as follows:

```
At time 211664: Task 1 reached sync point
At time 211664: Task 1 exited sync point
At time 211664: Task 2 exited sync point
At time 211664: Task 3 exited sync point
At time 212702: Task 2 reached sync point
At time 214400: Task 1 reached sync point
At time 215439: Task 3 reached sync point
At time 215439: Task 3 exited sync point
At time 215439: Task 2 exited sync point
At time 215440: Task 1 exited sync point
At time 217671: Task 2 reached sync point
At time 218622: Task 1 reached sync point
At time 219402: Task 3 reached sync point
At time 219402: Task 3 exited sync point
At time 219402: Task 2 exited sync point
At time 219402: Task 1 exited sync point
At time 220189: Task 2 reached sync point
At time 222656: Task 3 reached sync point
At time 222673: Task 1 reached sync point
At time 222673: Task 1 exited sync point
At time 222673: Task 2 exited sync point
At time 222673: Task 3 exited sync point
At time 223252: Task 1 reached sync point
At time 223682: Task 3 reached sync point
```

# 任务通知

本节内容：

- 任务的通知状态和通知值。
- 如何以及何时使用任务通知代替通信对象，如信号灯。
- 使用任务通知代替通信对象的优势。

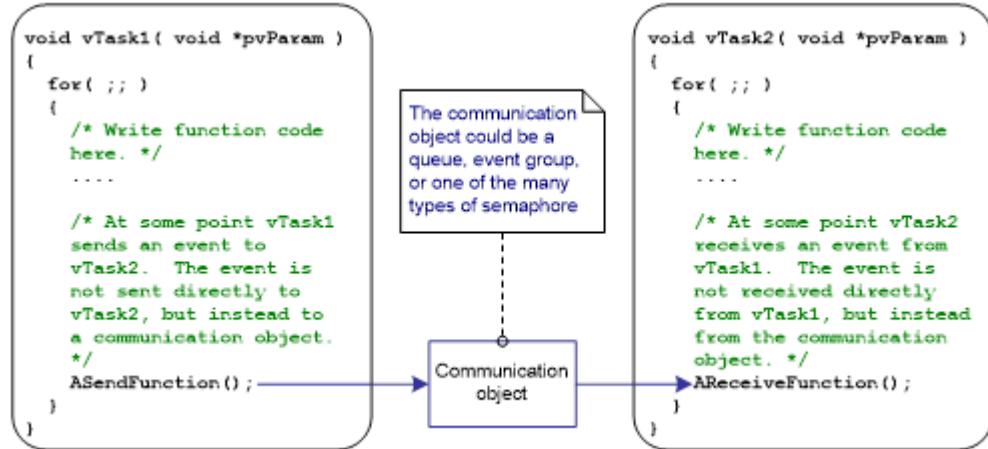
使用 FreeRTOS 的应用程序以一组独立任务的形式构成，这些自主任务必须相互通信，才能共同提供有用的系统功能。

## 通过中间对象进行通信

目前，任务互相通信所用的方法要求创建队列、事件组和不同类型的信号灯等通信对象。

使用通信对象时，事件和数据不直接发送到接收任务或接收 ISR，而是发送到通信对象。同样，任务和 ISR 是从通信对象而不是直接从发送事件或数据的任务或 ISR 接收事件和数据。

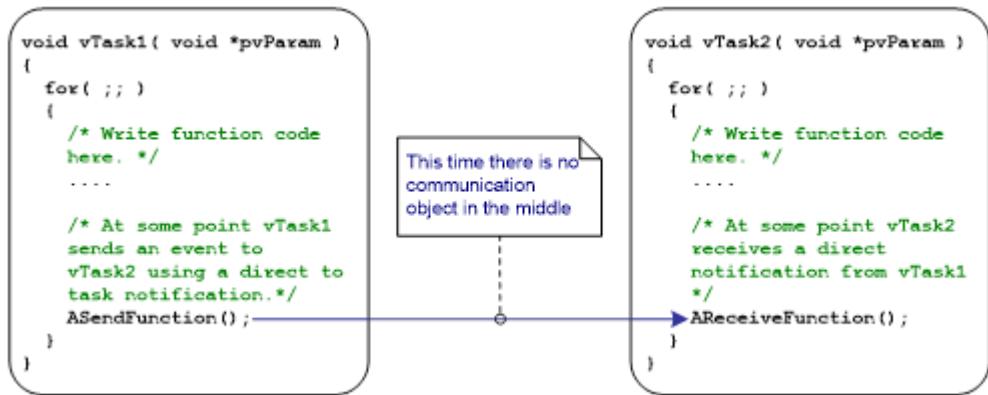
下图说明在任务之间发送事件的通信对象。



## 任务通知：直接到任务通信

通过任务通知，无需单独的通信对象，任务就可以与其他任务进行交互，以及与 ISR 同步。通过使用任务通知，任务或 ISR 可以直接向接收任务发送事件。

下图说明用于直接在任务之间发送事件的任务通知。



任务通知功能是可选的。要包含任务通知功能，请在 FreeRTOSConfig.h 中将 configUSE\_TASK\_NOTIFICATIONS 设置为 1。

当 configUSE\_TASK\_NOTIFICATIONS 设置为 1 时，每个任务都有一个通知状态（待处理或未在等待处理）和一个通知值（一个 32 位无符号整数）。当任务收到通知时，其通知状态将设置为待处理。当任务读取其通知值时，其通知状态将设置为未在等待处理。

任务可在“被阻止”状态中等待其通知状态变为待处理，还可以选择指定超时。

## 任务通知的优势和限制

使用任务通知向任务发送事件或数据要比使用队列、信号灯或事件组快得多。

同样，使用任务通知向任务发送事件或数据所需的 RAM 也比使用队列、信号灯或事件组少得多。这是因为每个通信对象（队列、信号灯或事件组）都必须先创建，然后才能使用，而启用任务通知功能具有固定开销，每个任务只需 8 字节的 RAM。

## 任务通知的限制

在以下情形中不能使用任务通知：

- 向 ISR 发送事件或数据。

通信对象可供 ISR 和任务相互发送事件和数据。

任务通知可用于从 ISR 向任务发送事件和数据，但不能用于从任务向 ISR 发送事件或数据。

- 启用多个接收任务。

通信对象可由任何知其句柄（可能是队列句柄、信号灯句柄或事件组句柄）的任务或 ISR 访问。任意数量的任务和 ISR 可处理发送到任何给定通信对象的事件或数据。

任务通知直接发送给接收任务，因此只能由接收通知的任务来处理。这在多数情况下并不是限制，因为尽管通常有多个任务和 ISR 向相同的通信对象发送事件或数据，但是很少有多个任务和 ISR 接收来自同一通信对象的事件或数据。

- 缓冲多个数据项。

队列是一次可保存多个数据项的通信对象。已发送到队列但尚未从队列中接收的数据将在队列对象中缓冲。

任务通知通过更新接收任务的通知值向任务发送数据。任务的通知值一次只能保存一个值。

- 广播到多个任务。

事件组是可用于一次向多个任务发送事件的通信对象。

任务通知直接发送给接收任务，因此只能由接收任务来处理。

- 在“被阻止”状态中等待发送完成。

如果通信对象暂时处于无法向其中写入更多数据或事件的状态（例如，当队列已满、无法向该队列发送更多数据时），尝试向该对象写入内容的任务可以选择进入“被阻止”状态，以等待其写入操作完成。

如果一个任务尝试向已有待处理通知的另一个任务发送任务通知，则发送任务无法在“被阻止”状态中等待接收任务重置其通知状态。在大多数使用任务通知的情况下，这极少成为限制。

## 使用任务通知

任务通知是非常强大的功能，常常可用来代替二进制信号灯、计数信号灯、事件组，有时甚至可以代替队列。

## 任务通知 API 选项

可以使用 xTaskNotify() API 函数发送任务通知，使用 xTaskNotifyWait() API 函数接收任务通知。

但在大多数情况下，并不需要 xTaskNotify() 和 xTaskNotifyWait() API 函数提供的灵活性。更简单的函数就足够了。xTaskNotifyGive() API 函数可替代 xTaskNotify()，它更简单、但灵活性稍差。ulTaskNotifyTake() API 函数可替代 xTaskNotifyWait()，它更简单、但灵活性稍差。

## xTaskNotifyGive() API 函数

xTaskNotifyGive() 直接向任务发送通知并使接收任务的通知值递增（加 1）。如果接收任务的通知状态还不是待处理，则调用 xTaskNotifyGive() 会将其设置为待处理。

通过 xTaskNotifyGive() API 函数，任务通知可用作更快速的二进制信号灯或计数信号灯轻型替代方案。下面是 xTaskNotifyGive() API 函数原型。

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

下表列出了 xTaskNotifyGive() 参数和返回值。

参数名称/返回的值	描述
xTaskToNotify	向其发送通知的任务的句柄。有关获取任务句柄的信息，请参阅 xTaskCreate() API 函数的 pxCreatedTask 参数。
返回的值	xTaskNotifyGive() 是调用 xTaskNotify() 的宏。通过设置该宏传入 xTaskNotify() 的参数，使 pdPASS 成为唯一可能的返回值。本节稍后将介绍 xTaskNotify()。

## vTaskNotifyGiveFromISR() API 函数

vTaskNotifyGiveFromISR() 是可在中断服务例程中使用的 xTaskNotifyGive() 版本。下面是 vTaskNotifyGiveFromISR() API 函数原型。

```
void vTaskNotifyGiveFromISR( TaskHandle_t xTaskToNotify, BaseType_t *pxHigherPriorityTaskWoken );
```

下表列出了 vTaskNotifyGiveFromISR() 参数和返回值。

参数名称/返回的值	描述
xTaskToNotify	向其发送通知的任务的句柄。有关获取任务句柄的信息，请参阅 xTaskCreate() API 函数的 pxCreatedTask 参数。
pxHigherPriorityTaskWoken	<p>如果将通知发送到的任务正在“被阻止”状态中等待接收通知，则发送通知将导致该任务离开“被阻止”状态。</p> <p>如果调用 vTaskNotifyGiveFromISR() 会导致任务离开“被阻止”状态，并且取消阻止的任务的优先级高于当前正在执行的任务（被中断的任务）的优先级，那么 vTaskNotifyGiveFromISR() 会在内部 <b>&lt;problematic&gt;*&lt;/problematic&gt;</b> 将 pxHigherPriorityTaskWoken 设置为 pdTRUE。</p> <p>如果 vTaskNotifyGiveFromISR() 将此值设置为 pdTRUE，则应在退出中断之前执行上下文切换。这将确保中断直接返回优先级最高的“准备就绪”状态的任务。</p> <p>与所有中断安全 API 函数一样，必须先将 pxHigherPriorityTaskWoken 参数设置为 pdFALSE，然后才能使用。</p>

## ulTaskNotifyTake() API 函数

ulTaskNotifyTake() 允许任务在“被阻止”状态中等待其通知值大于零，并使任务的通知值递减（减 1）或将其清空，然后再返回值。

通过 ulTaskNotifyTake() API 函数，任务通知可用作更快速的二进制信号灯或计数信号灯轻型替代方案。下面是 ulTaskNotifyTake() API 函数原型。

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t xTicksToWait );
```

下表列出了 ulTaskNotifyTake() 参数和返回值。

参数名称/返回的值	描述
xClearCountOnExit	如果将 xClearCountOnExit 设置为 pdTRUE，则会在 ulTaskNotifyTake() 的调用返回之前将调用任务的通知值清空为零。
xTicksToWait	调用任务应保持“被阻止”状态以等待其通知值大于零的最长时间。

	<p>阻止时间以计时周期指定，因此它表示的绝对时间取决于计时频率。宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为计时周期数。</p> <p>如果 FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 设置为 1，将 xTicksToWait 设置为 portMAX_DELAY 将导致任务无限期等待（无超时）。</p>
返回的值	<p>返回的值是在清空为零或递减之前的调用任务通知值，由 xClearCountOnExit 参数的值指定。</p> <p>如果指定了阻止时间（xTicksToWait 不为零）并且返回值不为零，则可能是在阻止时间到期之前，调用任务置于“被阻止”状态以等待其通知值大于零，并且其通知值已更新。</p> <p>如果指定了阻止时间（xTicksToWait 不为零）并且返回值为零，则调用任务置于“被阻止”状态以等待其通知值大于零，但在此之前指定的阻止时间已到期。</p>

## 使用任务通知代替信号灯的方法 1 (示例 24 )

示例 16 使用二进制信号灯从中断服务例程中取消阻止任务，从而有效地使任务与中断同步。此示例复制示例 16 的功能，但使用“直接到任务”通知代替二进制信号灯。

下面的代码说明与中断同步的任务实现。在示例 16 中使用的 xSemaphoreTake() 调用已被 ulTaskNotifyTake() 调用取代。

ulTaskNotifyTake() xClearCountOnExit 参数设置为 pdTRUE，这会使得接收任务的通知值在 ulTaskNotifyTake() 返回之前清空为零。因此，需要处理在每次调用 ulTaskNotifyTake() 之间已经可用的所有事件。在示例 16 中，因为使用了二进制信号灯，所以必须从硬件中确定待处理事件的数量，只是这种方法并非始终可行。在此示例中，是从 ulTaskNotifyTake() 中返回待处理事件数。

在两次调用 ulTaskNotifyTake 之间发生的中断事件锁定在任务的通知值中，如果调用任务已有待处理任务，则对 ulTaskNotifyTake() 的调用将立即返回。

```
/* The rate at which the periodic task generates software interrupts.*/
const TickType_t xInterruptFrequency = pdMS_TO_TICKS( 500UL );

static void vHandlerTask( void *pvParameters )

{
    /* xMaxExpectedBlockTime is set to be a little longer than the maximum expected time
       between events. */

    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS( 10 );
    uint32_t ulEventsToProcess;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {

```

```
/* Wait to receive a notification sent directly to this task from the interrupt
service routine. */

ulEventsToProcess = ulTaskNotifyTake( pdTRUE, xMaxExpectedBlockTime );

if( ulEventsToProcess != 0 )

{

    /* To get here at least one event must have occurred. Loop here until all
the pending events have been processed (in this case, just print out a message for each
event). */

    while( ulEventsToProcess > 0 )

    {

        vPrintString( "Handler task - Processing event.\r\n" );
        ulEventsToProcess--;
    }

}

else

{

    /* If this part of the function is reached, then an interrupt did not arrive
within the expected time. In a real application, it might be necessary to perform some
error recovery operations. */

}

}

}
```

用于生成软件中断的周期性任务在生成中断之前和之后输出一条消息。这样，可以在输出中观察到执行顺序。

下面的代码示例说明该中断处理程序。这与向受委托执行中断处理的任务直接发送通知没有什么区别。

```
static uint32_t ulExampleInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
will get set to pdTRUE inside the interrupt-safe API function if a context switch is
required. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Send a notification directly to the task to which interrupt processing is being
deferred. */

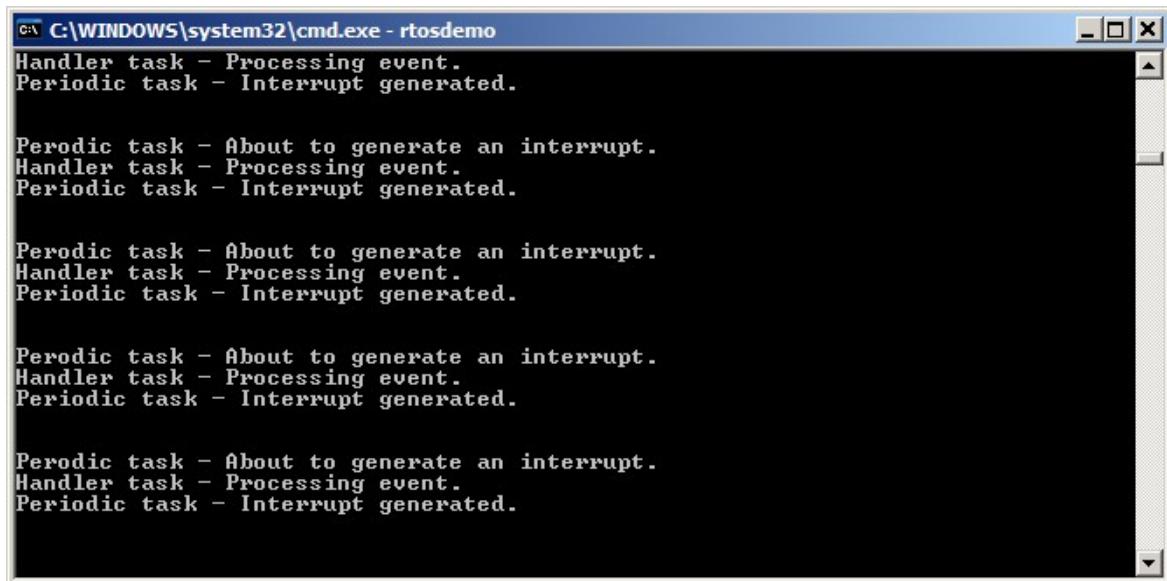
    vTaskNotifyGiveFromISR( /* The handle of the task to which the notification is
being sent. The handle was saved when the task was created. */ xHandlerTask, /*
xHigherPriorityTaskWoken is used in the usual way. */ &xHigherPriorityTaskWoken );
```

```
/* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
xHigherPriorityTaskWoken was set to pdTRUE inside vTaskNotifyGiveFromISR(), then calling
portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
this function does not explicitly return a value. */

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

执行代码时生成的输出如下所示。



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe - rtosdemo'. The window contains the following text output:

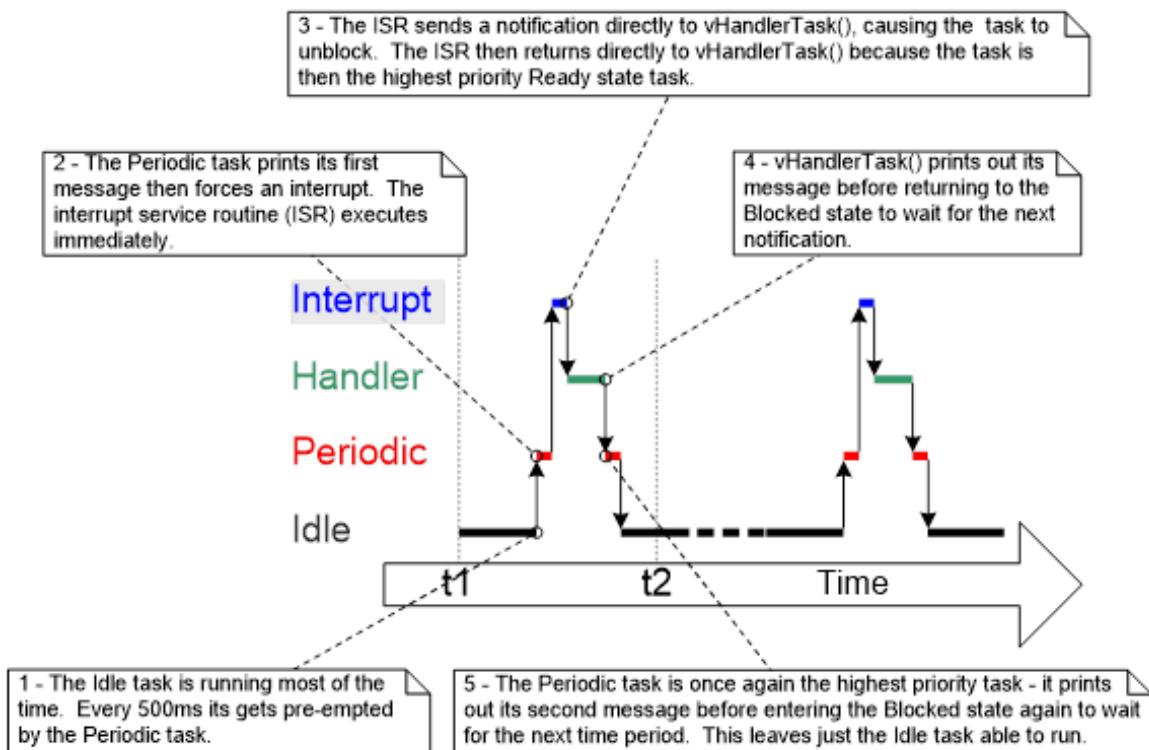
```
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.
```

正如预期的那样，该输出与执行示例 16 时生成的输出相同。只要生成中断，vHandlerTask() 就进入“运行”状态，因此任务输出将周期性任务的输出分隔开了。下面是执行顺序。



## 使用任务通知代替信号灯的方法 2 (示例 25 )

在示例 24 中，ulTaskNotifyTake() xClearOnExit 参数设置为 pdTRUE。示例 25 对示例 24 稍做修改，说明在将 ulTaskNotifyTake() xClearOnExit 参数设置为 pdFALSE 时的行为。

当 xClearOnExit 为 pdFALSE 时，调用 ulTaskNotifyTake() 只会将调用任务的通知值递减（减 1），而不将其清空为零。因此，通知计数是已发生的事件数与已处理的事件数之间的差值。这样可通过两种方式来简化 vHandlerTask() 的结构：

1. 等待处理的事件数保存在通知值中，因此不需要将其保存在本地变量中。
2. 在每次调用 ulTaskNotifyTake() 之间只需处理一个事件。

下面是 vHandlerTask() 的实现。

```
static void vHandlerTask( void *pvParameters )
{
    /* xMaxExpectedBlockTime is set to be a little longer than the maximum expected time
       between events. */

    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS( 10 );

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {

```

```
/* Wait to receive a notification sent directly to this task from the interrupt
service routine. The xClearCountOnExit parameter is now pdFALSE, so the task's
notification value will be decremented by ulTaskNotifyTake() and not cleared to zero. */

if( ulTaskNotifyTake( pdFALSE, xMaxExpectedBlockTime ) != 0 )

{

    /* To get here, an event must have occurred. Process the event (in this case,
just print out a message). */

    vPrintString( "Handler task - Processing event.\r\n" );

}

else

{

    /* If this part of the function is reached, then an interrupt did not arrive
within the expected time. In a real application, it might be necessary to perform some
error recovery operations. */

}

}

}

}
```

为了便于说明，还对中断服务例程进行了修改，以针对每个中断发送多个任务通知，这样便于模拟高频发生的多个中断。下面是中断服务例程的实现。

```
static uint32_t ulExampleInterruptHandler(void)

{

    BaseType_t xHigherPriorityTaskWoken;

    xHigherPriorityTaskWoken = pdFALSE;

    /* Send a notification to the handler task multiple times. The first give will unblock
the task. The following gives are to demonstrate that the receiving task's notification
value is being used to count (latch) events, allowing the task to process each event in
turn. */

    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

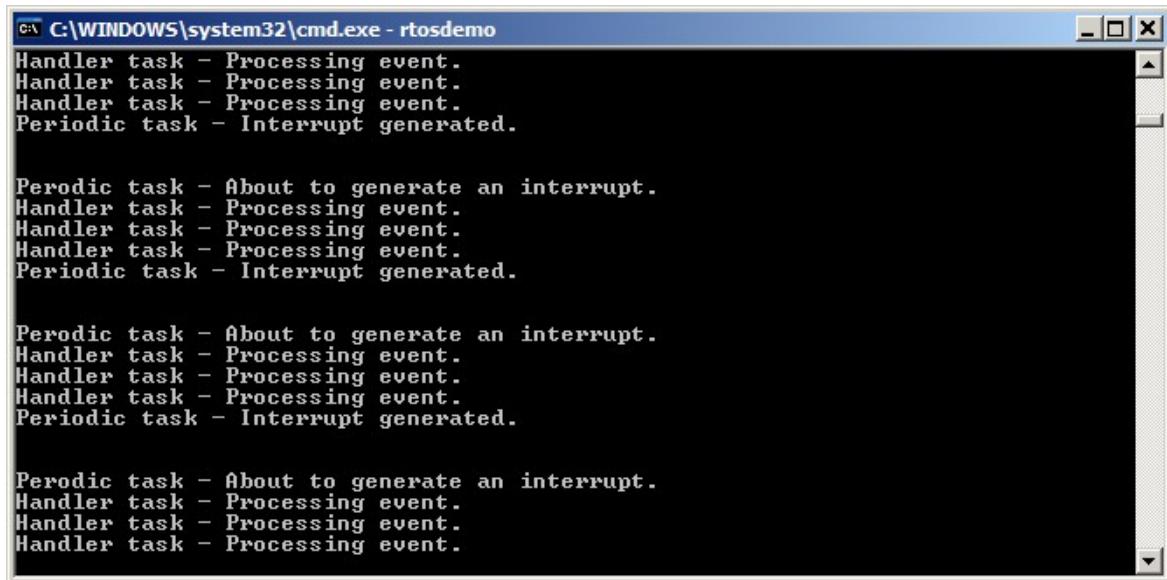
    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

}
```

下面是输出的内容。可以看到，每次生成中断时，vHandlerTask() 会处理全部三个事件。



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
```

## xTaskNotify() 和 xTaskNotifyFromISR() API 函数

xTaskNotify() 是功能更强大的 xTaskNotifyGive() 版本，可用于通过以下任何方式更新接收任务的通知值：

- 使接收任务的通知值递增（加 1），这种情况下 xTaskNotify() 与 xTaskNotifyGive() 等效。
- 在接收任务的通知值中设置一个或多个位。这样，可将任务的通知值用作更快捷的事件组轻型替代方案。
- 仅当接收任务自上次更新以来已读取其通知值时，才向接收任务的通知值中写入一个全新的数字。这样，任务的通知值可提供与长度为一的队列相似的功能。
- 即使接收任务自上次更新以来尚未读取其通知值，也向接收任务的通知值中写入一个全新的数字。这样，任务的通知值可提供与 xQueueOverwrite() API 函数相似的功能。产生的行为有时称为邮箱。

xTaskNotify() 比 xTaskNotifyGive() 更灵活，功能更强大。使用起来也更复杂一些。

xTaskNotifyFromISR() 是可在中断服务例程中使用的 xTaskNotify() 版本。因此，它有一个额外的 pxHigherPriorityTaskWoken 参数。

如果接收任务的通知状态还不是待处理，则调用 xTaskNotify() 总是会将其设置为待处理。

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction );
```

```
BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction, BaseType_t *pxHigherPriorityTaskWoken );
```

下表列出了 xTaskNotify() 参数和返回值。

参数名称/返回的值	描述
xTaskToNotify	向其发送通知的任务的句柄。有关获取任务句柄的信息，请参阅 xTaskCreate() API 函数的 pxCreatedTask 参数。

ulValue	如何使用 ulValue 取决于 eNotifyAction 值。请参阅表 52。
eNotifyAction	枚举类型，用于指定如何更新接收任务的通知值。请参阅下文。
返回的值	xTaskNotify() 将返回 pdPASS，下面指出的一种情况除外。

下表列出了有效 xTaskNotify() eNotifyAction 参数值及其对接收任务的通知值产生的影响。

eNotifyAction 值	对接收任务产生的影响
eNoAction	接收任务的通知状态设置为待处理，但不更新其通知值。未使用 xTaskNotify() ulValue 参数。通过 eNoAction 操作，任务通知可用作更快速的二进制信号灯轻型替代方案。
eSetBits	对接收任务的通知值和 xTaskNotify() ulValue 参数中传递的值逐位执行或运算。例如，如果 ulValue 设置为 0x01，则将在接收任务的通知值中设置位 0。再如，如果 ulValue 为 0x06（二进制 0110），则将在接收任务的通知值中设置位 1 和位 2。  通过 eSetBits 操作，任务通知可用作更快速的事件组轻型替代方案。
eIncrement	接收任务的通知值将递增。未使用 xTaskNotify() ulValue 参数。  通过 eIncrement 操作，任务通知可用作更快速的二进制或计数信号灯轻型替代方案。它与更简单的 xTaskNotifyGive() API 函数等效。
eSetValueWithoutOverwrite	如果在调用 xTaskNotify() 之前接收任务已有待处理通知，则不执行任何操作，xTaskNotify() 将返回 pdFAIL。  如果在调用 xTaskNotify() 之前接收任务没有待处理通知，接收任务的通知值将设置为 xTaskNotify() ulValue 参数中传递的值。
eSetValueWithOverwrite	无论在调用 xTaskNotify() 之前接收任务是否有待处理通知，接收任务的通知值均设置为 xTaskNotify() ulValue 参数中传递的值。

## xTaskNotifyWait() API 函数

xTaskNotifyWait() 是 ulTaskNotifyTake() 的功能更强大的版本。它允许任务等待调用任务的通知状态变为待处理（如果还不是待处理），还可以选择指定超时。xTaskNotifyWait() 还提供在进入函数和退出函数时在调用任务的通知值中清除位的选项。

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit,
                           uint32_t *pulNotificationValue, TickType_t xTicksToWait );
```

下表列出了 xTaskNotifyWait() 参数和返回值。

参数名称/返回的值	描述
ulBitsToClearOnEntry	<p>如果调用任务在调用 xTaskNotifyWait() 之前没有待处理通知，则进入函数时将在任务的通知值中清除所有在 ulBitsToClearOnEntry 中设置的位。</p> <p>例如，如果 ulBitsToClearOnEntry 为 0x01，则将清除该任务的通知值的位 0。再如，将 ulBitsToClearOnEntry 设置为 0xffffffff (ULONG_MAX) 将清除任务的通知值中的所有位，实际上将值清除为 0。</p>
ulBitsToClearOnExit	<p>如果调用任务因为收到通知或因为在调用 xTaskNotifyWait() 之前已有待处理通知而退出 xTaskNotifyWait()，则在该任务退出 xTaskNotifyWait() 函数之前，将在任务的通知值中清除所有在 ulBitsToClearOnExit 中设置的位。</p> <p>在 pulNotificationValue 中保存任务的通知值后，将清除位  <b>&lt;problematic&gt;*&lt;/problematic&gt;</b>          (请参阅下面的 pulNotificationValue 描述)。</p> <p>例如，如果 ulBitsToClearOnExit 为 0x03，则会在函数退出之前清除任务通知值的位 0 和位 1。</p> <p>将 ulBitsToClearOnExit 设置为 0xffffffff (ULONG_MAX) 将清除任务的通知值中的所有位，实际上将值清除为 0。</p>
pulNotificationValue	<p>用于传出任务的通知值。复制到 pulNotificationValue 中的  <b>&lt;problematic&gt;*&lt;/problematic&gt;</b>          值是在因 ulBitsToClearOnExit 设置而清除任何位之前的任务通知值。</p> <p>pulNotificationValue 是可选参数，如果不需，可将其设置为 NULL。</p>
xTicksToWait	<p>调用任务应保持“被阻止”状态以等待其通知状态变为待处理的最长时间。</p> <p>阻止时间以计时周期指定，因此它表示的绝对时间取决于计时频率。宏 pdMS_TO_TICKS() 可用于将以毫秒为单位指定的时间转换为以时钟周期指定的时间。</p> <p>如果 FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 设置为 1，将 xTicksToWait 设置为 portMAX_DELAY 将导致任务无限期等待 (无超时)。</p>
返回的值	<p>可能的返回值有两个：</p> <ol style="list-style-type: none"> <li>pdTRUE</li> </ol>

此值指示 , xTaskNotifyWait() 因为收到通知或因为调用任务在调用 xTaskNotifyWait() 之前已有待处理通知而返回。

如果指定了阻止时间 ( xTicksToWait 不为零 ) , 则调用任务可能置于“被阻止”状态以等待其通知状态变为待处理 , 但在阻止时间到期之前其通知状态已设置为待处理。

### 1. pdFALSE

此值指示在调用任务未收到任务通知的情况下 , xTaskNotifyWait() 返回。

如果 xTicksToWait 不为零 , 则调用任务会保持“被阻止”状态以等待其通知状态变为待处理 , 但指定的阻止时间在此之前就已到期。

## 外围设备驱动程序中使用的任务通知 : UART 示例

外围设备驱动程序库提供可对硬件接口执行常见操作的函数。这些库的外围设备包括通用异步接收器和发送器 (UART)、串行外围设备接口 (SPI) 端口、模数转换器 (ADC) 和以太网端口。这些库提供的函数通常包括用于初始化外围设备、向外围设备发送数据和接收来自外围设备的数据的函数。

对外围设备执行的部分操作需要很长时间才能完成。其中包括高精度 ADC 转换和在 UART 中传输大型数据包。在这些情况下 , 驱动程序库函数可实现为轮询 ( 重复读取 ) 外围设备的状态寄存器 , 以确定操作何时完成。但是 , 这种方式的轮询总是会造成浪费 , 因为它占用处理器的全部时间 , 而不执行任何有成效的处理。在多任务处理系统中 , 这种浪费的代价特别高昂 , 其中轮询外围设备的任务可能会使要执行有成效处理的低优先级任务无法执行。

为避免处理时间的浪费 , 可感知 RTOS 的高效设备驱动程序应由中断驱动 , 并向启动长时间操作的任务提供在“被阻止”状态中等待操作完成的选项。这样 , 在执行长时间操作的任务处于“被阻止”状态时 , 较低优先级的任务便可执行 , 并且其他所有任务都不能使用处理时间 ( 除非它们可以富有成效地使用 ) 。

可感知 RTOS 的驱动程序库的常见做法是 , 使用二进制信号灯将任务置于“被阻止”状态。下面的伪代码说明这一方法 , 概要介绍在 UART 端口上传输数据的可感知 RTOS 的库函数。在下面的代码列表中 :

- xUART 是用于描述 UART 外围设备和保存状态信息的结构。该结构的 xTxSemaphore 成员是 SemaphoreHandle\_t 类型的变量。此处假设已创建信号灯。
- xUART\_Send() 函数不包括任何互斥逻辑。如果有多个任务要使用 xUART\_Send() 函数 , 应用程序编写者必须在应用程序自身内管理互斥。例如 , 任务可能需要在调用 xUART\_Send() 之前获取互斥锁。
- xSemaphoreTake() API 函数用于在启动 UART 传输之后将调用任务置于“被阻止”状态。
- xSemaphoreGiveFromISR() API 函数用于在传输完成后 ( 即 UART 外围设备的传输端中断服务例程执行之时 ) 使任务离开“被阻止”状态。

```
/* Driver library function to send data to a UART. */

 BaseType_t xUART_Send( xUART *pxUARTInstance, uint8_t *pucDataSource, size_t uxLength )

{
    BaseType_t xReturn;
```

```
/* Ensure the UART's transmit semaphore is not already available by attempting to take
the semaphore without a timeout. */

xSemaphoreTake( pxUARTInstance->xTxSemaphore, 0 );

/* Start the transmission. */

UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

/* Block on the semaphore to wait for the transmission to complete. If the semaphore is
obtained, then xReturn will get set to pdPASS. If the semaphore take operation times out,
then xReturn will get set to pdFAIL. If the interrupt occurs between UART_low_level_send()
being called and xSemaphoreTake() being called, then the event will be latched in the
binary semaphore, and the call to xSemaphoreTake() will return immediately. */

xReturn = xSemaphoreTake( pxUARTInstance->xTxSemaphore, pxUARTInstance->xTxTimeout );

return xReturn;
}

/*-----*/
/* The service routine for the UART's transmit end interrupt, which executes after the last
byte has been sent to the UART. */

void xUART_TransmitEndISR( xUART *pxUARTInstance )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Clear the interrupt. */

    UART_low_level_interrupt_clear( pxUARTInstance );

    /* Give the Tx semaphore to signal the end of the transmission. If a task is Blocked
waiting for the semaphore, then the task will be removed from the Blocked state. */

    xSemaphoreGiveFromISR( pxUARTInstance->xTxSemaphore, &xHigherPriorityTaskWoken );
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

这段代码中说明的方法切实可行并且很常用，但它存在一些缺点：

- 该库使用多个信号灯，因此会增加它的 RAM 占用空间。
- 信号灯必须先创建，然后才能使用，因此使用信号灯的库只有在进行显式初始化之后才能使用。
- 信号灯是适用于各种使用案例的通用对象。信号灯包括的逻辑允许任意数量的任务在“被阻止”状态中等待信号灯变为可用，还允许（以确定性方式）选择在信号灯变为可用时使哪个任务离开“被阻止”状态。执行该逻辑需要有限时间。在此代码所示的情形中，在任意给定时间不能有多个任务等待信号灯，因此这种处理开销是不必要的。

下面的代码说明如何通过使用任务通知代替二进制信号灯来避免这些缺点。

注意：如果库使用任务通知，则该库的文档必须明确说明调用库函数会更改调用任务的通知状态和值。

在以下代码中：

- xUART 结构的 xTxSemaphore 成员已经替换为 xTaskToNotify 成员。xTaskToNotify 是 TaskHandle\_t 类型的变量，用于保存正在等待 UART 操作完成的任务的句柄。

- `xTaskGetCurrentTaskHandle()` FreeRTOS API 函数用于获取处于“运行”状态的任务的句柄。
- 该库不创建任何 FreeRTOS 对象，因此不会产生 RAM 开销，也不需要显式初始化。
- 任务通知将直接发送给正在等待 UART 操作完成的任务，因此不会执行任何不必要的逻辑。

从任务和中断服务例程中均可访问 `xUART` 结构的 `xTaskToNotify` 成员，因此需要考虑处理器如何更新其值：

- 如果通过单个内存写入操作更新 `xTaskToNotify`，则可在关键部分之外更新，完全如以下代码所示。如果 `xTaskToNotify` 是 32 位变量（`TaskHandle_t` 是 32 位类型），运行 FreeRTOS 的处理器是 32 位处理器，即是这种情况。
- 如果需要多个内存写入操作才能更新 `xTaskToNotify`，则只能在关键部分内更新 `xTaskToNotify`。否则，中断服务例程可能会在 `xTaskToNotify` 处于不一致状态时访问它。如果 `xTaskToNotify` 是 32 位变量，运行 FreeRTOS 的处理器是 16 位处理器，即是这种情况，因为需要两个 16 位内存写入操作才能更新全部 32 位。

在 FreeRTOS 实现内部，`TaskHandle_t` 是一个指针，因此 `sizeof( TaskHandle_t )` 始终等于 `sizeof( void * )`。

```
/* Driver library function to send data to a UART. */

 BaseType_t xUART_Send( xUART *pxUARTInstance, uint8_t *pucDataSource, size_t uxLength )
{
    BaseType_t xReturn;

    /* Save the handle of the task that called this function. The book text contains notes
     * as to whether the following line needs to be protected by a critical section or not. */
    pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();

    /* Ensure the calling task does not already have a notification pending by calling
     * ulTaskNotifyTake() with the xClearCountOnExit parameter set to pdTRUE, and a block time of
     * 0 (don't block). */
    ulTaskNotifyTake( pdTRUE, 0 );

    /* Start the transmission. */

    UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

    /* Block until notified that the transmission is complete. If the notification is
     * received, then xReturn will be set to 1 because the ISR will have incremented this task's
     * notification value to 1 (pdTRUE). If the operation times out, then xReturn will be 0
     * (pdFALSE) because this task's notification value will not have been changed since it was
     * cleared to 0 above. If the ISR executes between the calls to UART_low_level_send()
     * and the call to ulTaskNotifyTake(), then the event will be latched in the task's notification
     * value, and the call to ulTaskNotifyTake() will return immediately.*/
    xReturn = ( BaseType_t ) ulTaskNotifyTake( pdTRUE, pxUARTInstance->xTxTimeout );

    return xReturn;
}

/*-----*/
/* The ISR that executes after the last byte has been sent to the UART. */

void xUART_TransmitEndISR( xUART *pxUARTInstance )
```

```
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* This function should not execute unless there is a task waiting to be notified.
    Test this condition with an assert. This step is not strictly necessary, but will aid
    debugging. configASSERT() is described in Developer Support.*/

    configASSERT( pxUARTInstance->xTaskToNotify != NULL );

    /* Clear the interrupt. */

    UART_low_level_interrupt_clear( pxUARTInstance );

    /* Send a notification directly to the task that called xUART_Send(). If the task
    is Blocked waiting for the notification, then the task will be removed from the Blocked
    state. */

    vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify, &xHigherPriorityTaskWoken );

    /* Now there are no tasks waiting to be notified. Set the xTaskToNotify member of the
    xUART structure back to NULL. This step is not strictly necessary, but will aid debugging.
    */

    pxUARTInstance->xTaskToNotify = NULL;

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

如以下伪代码所示，任务通知还可以替代接收函数中的信号灯，这段伪代码概要介绍在 UART 端口上接收数据的可感知 RTOS 的库函数。

- xUART\_Receive() 函数不包括任何互斥逻辑。如果有多个任务要使用 xUART\_Receive() 函数，应用程序编写者必须在应用程序自身内管理互斥。例如，任务可能需要在调用 xUART\_Receive() 之前获取互斥锁。
- UART 的接收中断服务例程将 UART 接收的字符放入 RAM 缓冲区。xUART\_Receive() 函数从 RAM 缓冲区返回字符。
- xUART\_Receive() uxWantedBytes 参数用于指定要接收的字符的数量。如果 RAM 缓冲区尚未包含所请求数量的字符，调用任务将置于“被阻止”状态，以等待缓冲区中的字符数量增加的通知。while() 循环用于重复此序列，直到接收缓冲区包含所请求数量的字符或发生超时。
- 调用任务可以多次进入“被阻止”状态。因此，阻止时间的调整应计入自调用 xUART\_Receive() 后已过去的时间。调整可确保在 xUART\_Receive() 内花费的总时间不超过 xUART 结构的 xRxTimeout 成员所指定的阻止时间。阻止时间通过 FreeRTOS vTaskSetTimeOutState() 和 xTaskCheckForTimeOut() 帮助程序函数来调整。

```
/* Driver library function to receive data from a UART. */

size_t xUART_Receive( xUART *pxUARTInstance, uint8_t *pucBuffer, size_t uxWantedBytes )
{
    size_t uxReceived = 0;

    TickType_t xTicksToWait;

    TimeOut_t xTimeOut;

    /* Record the time at which this function was entered. */

    vTaskSetTimeOutState( &xTimeOut );
```

```
/* xTicksToWait is the timeout value. It is initially set to the maximum receive
timeout for this UART instance. */

xTicksToWait = pxUARTInstance->xRxTimeout;

/* Save the handle of the task that called this function. */

pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();

/* Loop until the buffer contains the wanted number of bytes or a timeout occurs. */

while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )

{

    /* Look for a timeout, adjusting xTicksToWait to account for the time spent in this
function so far. */

    if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )

    {

        /* Timed out before the wanted number of bytes were available, exit the loop.
*/
        break;

    }

    /* The receive buffer does not yet contain the required amount of bytes. Wait for
a maximum of xTicksToWait ticks to be notified that the receive interrupt service routine
has placed more data into the buffer. It does not matter if the calling task already had a
notification pending when it called this function. If it did, it would just iterate around
this while loop one extra time. */

    ulTaskNotifyTake( pdTRUE, xTicksToWait );

}

/* No tasks are waiting for receive notifications, so set xTaskToNotify back to NULL.
*/
pxUARTInstance->xTaskToNotify = NULL;

/* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The actual
number of bytes read (which might be less than uxWantedBytes) is returned. */

uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );

return uxReceived;

}

/*-----*/
/* The interrupt service routine for the UART's receive interrupt */

void xUART_ReceiveISR( xUART *pxUARTInstance )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Copy received data into this UART's receive buffer and clear the interrupt. */
}
```

```
UART_low_level_receive( pxUARTInstance );

/* If a task is waiting to be notified of the new data, then notify it now. */

if( pxUARTInstance->xTaskToNotify != NULL )

{

    vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify,
&xHigherPriorityTaskWoken );

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}

}
```

## 外围设备驱动程序中使用的任务通知 : ADC 示例

上一节说明了如何使用 vTaskNotifyGiveFromISR() 从中断向任务发送任务通知。vTaskNotifyGiveFromISR() 是一个使用起来很简单的函数，但其功能有限。它只能以无值事件的形式发送任务通知。它无法发送数据。本节说明如何使用 xTaskNotifyFromISR() 通过任务通知事件发送数据。下面的伪代码说明这一方法，概要介绍适用于模数转换器 (ADC) 的可感知 RTOS 的中断服务例程。

- 假定 ADC 转换至少每 50 毫秒启动一次。
- ADC\_ConversionEndISR() 是用于 ADC 的转换端中断的中断服务例程，每当有新 ADC 值可用时便执行该中断。
- vADCTask() 实现的任务处理 ADC 生成的每个值。假设创建该任务时，该任务的句柄存储在 xADCTaskToNotify 中。
- ADC\_ConversionEndISR() 使用 xTaskNotifyFromISR() ( 将 eAction 参数设置为 eSetValueWithoutOverwrite ) 向 vADCTask() 任务发送任务通知并将 ADC 转换的结果写入该任务的通知值。
- vADCTask() 任务使用 xTaskNotifyWait() 等待有新 ADC 值可用的通知并从其通知值中检索 ADC 转换的结果。

```
/* A task that uses an ADC. */

void vADCTask( void *pvParameters )

{

    uint32_t ulADCValue;

    BaseType_t xResult;

    /* The rate at which ADC conversions are triggered. */

    const TickType_t xADCConversionFrequency = pdMS_TO_TICKS( 50 );

    for( ;; )

    {

        /* Wait for the next ADC conversion result. */

        xResult = xTaskNotifyWait( /* The new ADC value will overwrite the old value, so
there is no need to clear any bits before waiting for the new notification value. */ 0, /* Future ADC values will overwrite the existing value, so there is no need to clear any bits
before exiting xTaskNotifyWait(). */ 0, /* The address of the variable into which the
```

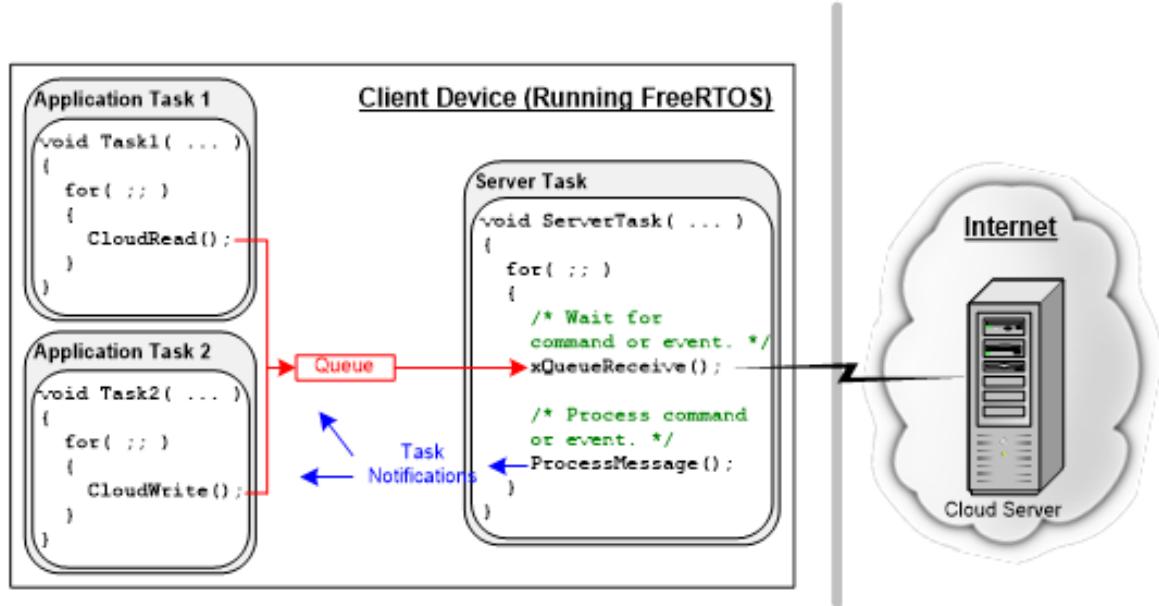
```
task's notification value (which holds the latest ADC conversion result) will be copied.  
/* &ulADCValue, /* A new ADC value should be received every xADCConversionFrequency ticks.  
/* xADCConversionFrequency * 2 );  
  
    if( xResult == pdPASS )  
  
    {  
  
        /* A new ADC value was received. Process it now. */  
  
        ProcessADCResult( ulADCValue );  
  
    }  
  
    else  
  
    {  
  
        /* The call to xTaskNotifyWait() did not return within the expected time.  
Something must be wrong with the input that triggers the ADC conversion or with the ADC  
itself. Handle the error here. */  
  
    }  
  
}  
  
}/*-----*/  
  
/* The interrupt service routine that executes each time an ADC conversion completes. */  
  
void ADC_ConversionEndISR( xADC *pxADCInstance )  
  
{  
  
    uint32_t ulConversionResult;  
  
    BaseType_t xHigherPriorityTaskWoken = pdFALSE, xResult;  
  
    /* Read the new ADC value and clear the interrupt. */  
  
    ulConversionResult = ADC_low_level_read( pxADCInstance );  
  
    /* Send a notification, and the ADC conversion result, directly to vADCTask(). */  
  
    xResult = xTaskNotifyFromISR( xADCTaskToNotify, /* xTaskToNotify parameter. */  
                                ulConversionResult, /* ulValue parameter. */ eSetValueWithoutOverwrite, /* eAction  
parameter. */ &xHigherPriorityTaskWoken );  
  
    /* If the call to xTaskNotifyFromISR() returns pdFAIL then the task is not keeping  
up with the rate at which ADC values are being generated. configASSERT() is described in  
section 11.2.*/  
  
    configASSERT( xResult == pdPASS );  
  
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );  
}
```

## 直接在应用程序中使用的任务通知

本节说明如何在包含以下功能的假想应用程序中使用任务通知：

1. 应用程序通过较慢的互联网连接向远程数据服务器（云服务器）发送数据并从该服务器中请求数据。
2. 从云服务器中请求数据后，请求任务必须在“被阻止”状态中等待接收请求的数据。
3. 在向云服务器发送数据后，发送任务必须在“被阻止”状态中等待云服务器已正确接收数据的确认。

下面是软件设计。



- 处理到云服务器的多个互联网连接的复杂性封装在单个 FreeRTOS 任务中。该任务在 FreeRTOS 应用程序中充当前代理服务器，称为服务器任务。
- 应用程序任务通过调用 CloudRead() 从云服务器中读取数据。CloudRead() 不直接与云服务器通信。而是向队列中的服务器任务发送读取请求，并以任务通知的形式接收从服务器任务请求的数据。
- 应用程序任务通过调用 CloudWrite() 向云服务器写入数据。CloudWrite() 不直接与云服务器通信。而是向队列中的服务器任务发送写入请求，并以任务通知的形式从服务器任务接收写入操作结果。

以下代码说明由 CloudRead() 和 CloudWrite() 函数发送至服务器任务的结构。

```
typedef enum CloudOperations
{
    eRead, /* Send data to the cloud server. */
    eWrite /* Receive data from the cloud server. */
} Operation_t;

typedef struct CloudCommand
{
    Operation_t eOperation; /* The operation to perform (read or write). */
    uint32_t ulDataID; /* Identifies the data being read or written. */
    uint32_t ulDataValue; /* Only used when writing data to the cloud server. */
    TaskHandle_t xTaskToNotify; /* The handle of the task performing the operation. */
}
```

```
} CloudCommand_t;
```

下面是 CloudRead() 的伪代码。该函数将其请求发送到服务器任务，然后调用 xTaskNotifyWait() 以在“被阻止”状态中等待，直至收到请求的数据可用的通知。

```
/* ulDataID identifies the data to read. pulValue holds the address of the variable into
   which the data received from the cloud server is to be written. */

BaseType_t CloudRead( uint32_t ulDataID, uint32_t *pulValue )

{

    CloudCommand_t xRequest;

    BaseType_t xReturn;

    /* Set the CloudCommand_t structure members to be correct for this read request. */

    xRequest.eOperation = eRead; /* This is a request to read data. */

    xRequest.ulDataID = ulDataID; /* A code that identifies the data to read. */

    xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Handle of the calling task. */

    /* Ensure there are no notifications already pending by reading the notification value
       with a block time of 0, and then send the structure to the server task. */

    xTaskNotifyWait( 0, 0, NULL, 0 );

    xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );

    /* Wait for a notification from the server task. The server task writes the value
       received from the cloud server directly into this task's notification value, so there
       is no need to clear any bits in the notification value on entry to or exit from the
       xTaskNotifyWait() function. The received value is written to *pulValue, so pulValue is
       passed as the address to which the notification value is written. */

    xReturn = xTaskNotifyWait( 0, /* No bits cleared on entry. */ 0, /* No bits to clear
        on exit. */ pulValue, /* Notification value into *pulValue. */ pdMS_TO_TICKS( 250 ) );
    /* Wait a maximum of 250ms. */
    /* If xReturn is pdPASS, then the value was obtained. If xReturn is pdFAIL,
       then the request timed out. */

    return xReturn;
}
```

下面的伪代码说明服务器任务如何管理读取请求。当云服务器接收到数据后，服务器任务取消阻止应用程序任务，通过将 eAction 参数设置为 eSetValueWithOverwrite 调用 xTaskNotify()，将接收到的数据发送至应用程序任务。

这是简化情形，因为它假定 GetCloudData() 不必等待获取来自云服务器的值。

```
void ServerTask( void *pvParameters )

{

    CloudCommand_t xCommand;

    uint32_t ulReceivedValue;

    for( ;; )
```

```
{  
    /* Wait for the next CloudCommand_t structure to be received from a task. */  
  
    xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );  
  
    switch( xCommand.eOperation ) /* Was it a read or write request? */  
    {  
  
        case eRead:  
  
            /* Obtain the requested data item from the remote cloud server. */  
  
            ulReceivedValue = GetCloudData( xCommand.ulDataID );  
  
            /* Call xTaskNotify() to send both a notification and the value received from  
            the cloud server to the task that made the request. The handle of the task is obtained  
            from the CloudCommand_t structure. */  
  
            xTaskNotify( xCommand.xTaskToNotify, /* The task's handle is in the structure.  
            */ ulReceivedValue, /* Cloud data sent as notification value. */ eSetValueWithOverwrite );  
  
            break;  
  
            /* Other switch cases go here. */  
  
    }  
}
```

下面是 CloudWrite() 的伪代码。为了便于说明，CloudWrite() 返回一个按位状态代码，其中的每一位都有独特含义。顶部的 #define 语句是四个示例状态位。

该任务清除四个状态位、将其请求发送至服务器任务，然后调用 xTaskNotifyWait() 以在“被阻止”状态中等待状态通知。

```
/* Status bits used by the cloud write operation. */  
  
#define SEND_SUCCESSFUL_BIT ( 0x01 << 0 )  
  
#define OPERATION_TIMED_OUT_BIT ( 0x01 << 1 )  
  
#define NO_INTERNET_CONNECTION_BIT ( 0x01 << 2 )  
  
#define CANNOT_LOCATE_CLOUD_SERVER_BIT ( 0x01 << 3 )  
  
/* A mask that has the four status bits set. */  
  
#define CLOUD_WRITE_STATUS_BIT_MASK ( SEND_SUCCESSFUL_BIT \  
OPERATION_TIMED_OUT_BIT \  
NO_INTERNET_CONNECTION_BIT \  
CANNOT_LOCATE_CLOUD_SERVER_BIT )  
  
uint32_t CloudWrite( uint32_t ulDataID, uint32_t ulDataValue )  
{
```

```
CloudCommand_t xRequest;

uint32_t ulNotificationValue;

/* Set the CloudCommand_t structure members to be correct for this write request. */

xRequest.eOperation = eWrite; /* This is a request to write data. */

xRequest.ulDataID = ulDataID; /* A code that identifies the data being written. */

xRequest.ulDataValue = ulDataValue; /* Value of the data written to the cloud server.
*/
xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Handle of the calling task. */

/* Clear the three status bits relevant to the write operation by
calling xTaskNotifyWait() with the ulBitsToClearOnExit parameter set to
CLOUD_WRITE_STATUS_BIT_MASK, and a block time of 0. The current notification value is not
required, so the pulNotificationValue parameter is set to NULL. */

xTaskNotifyWait( 0, CLOUD_WRITE_STATUS_BIT_MASK, NULL, 0 );

/* Send the request to the server task. */

xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );

/* Wait for a notification from the server task. The server task writes a bitwise
status code into this task's notification value, which is written to ulNotificationValue.
*/
xTaskNotifyWait( 0, /* No bits cleared on entry. */ CLOUD_WRITE_STATUS_BIT_MASK, /
* Clear relevant bits to 0 on exit. */ &ulNotificationValue, /* Notified value. */
pdMS_TO_TICKS( 250 ) ); /* Wait a maximum of 250ms. */ /* Return the status code to the
calling task. */ return ( ulNotificationValue & CLOUD_WRITE_STATUS_BIT_MASK );
```

下面的伪代码说明服务器任务如何管理写入请求。当数据发送到云服务器后，服务器任务取消阻止应用程序任务，通过将 eAction 参数设置为 eSetBits 调用 xTaskNotify()，向应用程序任务发送按位状态代码。在接收任务的通知值中，只有 CLOUD\_WRITE\_STATUS\_BIT\_MASK 常量定义的位是可以更改的，因此接收任务可将其通知值中的其他位用于其他用途。

这是简化情形，因为它假定 SetCloudData() 不必等待获取来自远程云服务器的确认。

```
void ServerTask( void *pvParameters )

{
    CloudCommand_t xCommand;
    uint32_t ulBitwiseStatusCode;
    for( ;; )
    {
        /* Wait for the next message. */

        xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );

        /* Was it a read or write request? */
        switch( xCommand.eOperation )
        {
```

```
    case eWrite:

        /* Send the data to the remote cloud server. SetCloudData() returns a bitwise
        status code that only uses the bits defined by the CLOUD_WRITE_STATUS_BIT_MASK definition
        (shown in the preceding code). */

        ulBitwiseStatusCode = SetCloudData( xCommand.ulDataID, xCommand.ulDataValue );

        /* Send a notification to the task that made the write request. The eSetBits
        action is used so any status bits set in ulBitwiseStatusCode will be set in the
        notification value of the task being notified. All the other bits remain unchanged. The
        handle of the task is obtained from the CloudCommand_t structure. */

        xTaskNotify( xCommand.xTaskToNotify, /* The task's handle is in the structure.
        */ ulBitwiseStatusCode, /* Cloud data sent as notification value. */ eSetBits );

        break;

        /* Other switch cases go here. */

    }

}

}
```

# 开发人员支持

本节介绍的功能通过以下方式最大限度地提高效率：

- 深入了解应用程序的行为。
- 重点突出优化机会。
- 在错误发生的时间点捕获错误。

## configASSERT()

在 C 语言中，宏 assert() 用于验证由程序所做的断言（假设）。断言以 C 语言表达式的形式编写。如果表达式的计算结果为 false (0)，则该断言被视为失败。例如，以下代码用于测试断言“指针 pxMyPointer 不为 NULL”。

```
/* Test the assertion that pxMyPointer is not NULL */  
assert( pxMyPointer != NULL );
```

应用程序编写者通过提供 assert() 宏的实现，来指定断言失败时要执行的操作。

FreeRTOS 源代码不调用 assert()，因为 assert() 不可用于所有用来对 FreeRTOS 进行编译的编译器。相反，FreeRTOS 源代码包含大量对名为 configASSERT() 的宏的调用，此宏可以由应用程序编写者在 FreeRTOSConfig.h 中定义，其行为与标准的 C 语言 assert() 完全类似。

失败的断言必须被视为严重错误。请勿尝试跳过断言已失败的行执行。

使用 configASSERT() 可立即捕获和确定很多最常见的错误来源，从而提高效率。我们强烈建议您在开发或调试 FreeRTOS 应用程序时定义 configASSERT()。

定义 configASSERT() 将帮助您进行运行时调试，但它也将增加应用程序代码大小，因此会降低其执行速度。如果未提供 configASSERT() 的定义，则将使用默认的空定义，而所有对 configASSERT() 的调用都将由 C 语言预处理器完全删除。

## 示例 configASSERT() 定义

当应用程序在调试程序的控制下执行时，以下代码中所示的 configASSERT() 定义很有用。它将在断言失败的任何行上暂停执行，因此，当调试会话暂停时，调试程序将显示断言失败的行。

```
/* Disable interrupts so the tick interrupt stops executing, and then sit in a loop so  
execution does not move past the line that failed the assertion. If the hardware supports  
a debug break instruction, then the debug break instruction can be used in place of the  
for() loop. */  
  
#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS();  
for(;;) }
```

当应用程序未在调试程序的控制下执行时，configASSERT() 的定义很有用。它会输出或以其他方式记录断言失败的源代码行。您可以通过使用标准 C \_\_FILE\_\_ 宏获取源文件的名称，使用标准 C \_\_LINE\_\_ 宏获取源文件中的行号，以确定断言失败的行。

此代码显示的 configASSERT() 定义用于记录断言失败的源代码行。

## Tracealyzer

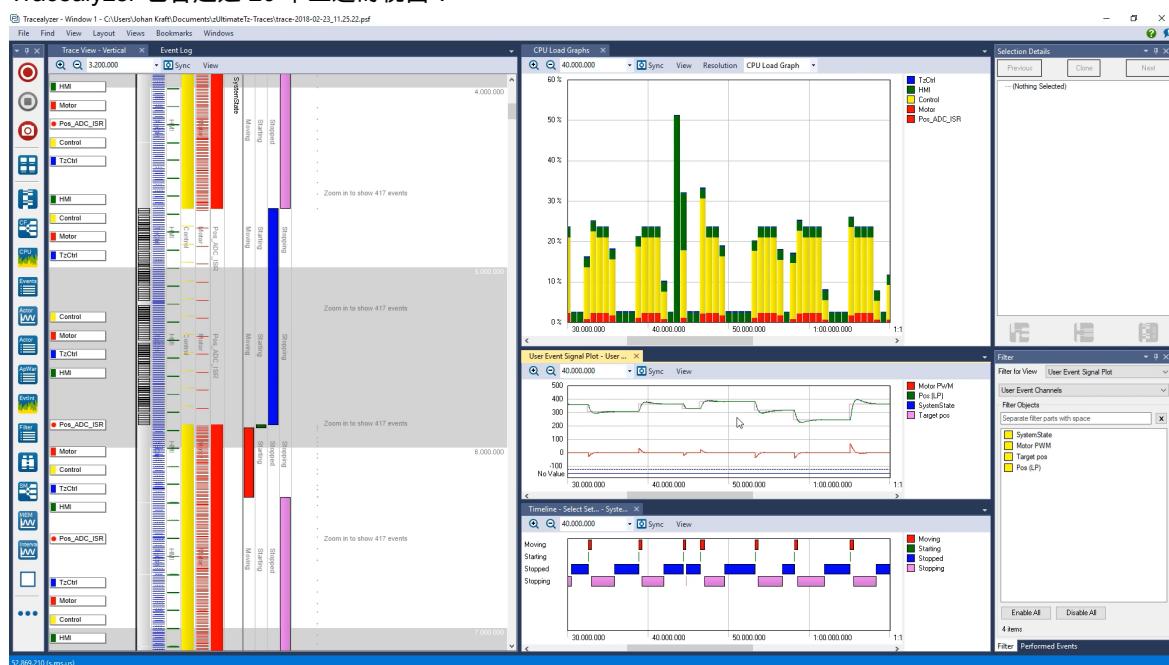
Tracealyzer 是我们的合作伙伴公司 Percepio 提供的运行时诊断和优化工具。

Tracealyzer 可捕获有价值的动态行为信息，并将其表示为互连的图形视图。该工具还可以显示多个同步视图。

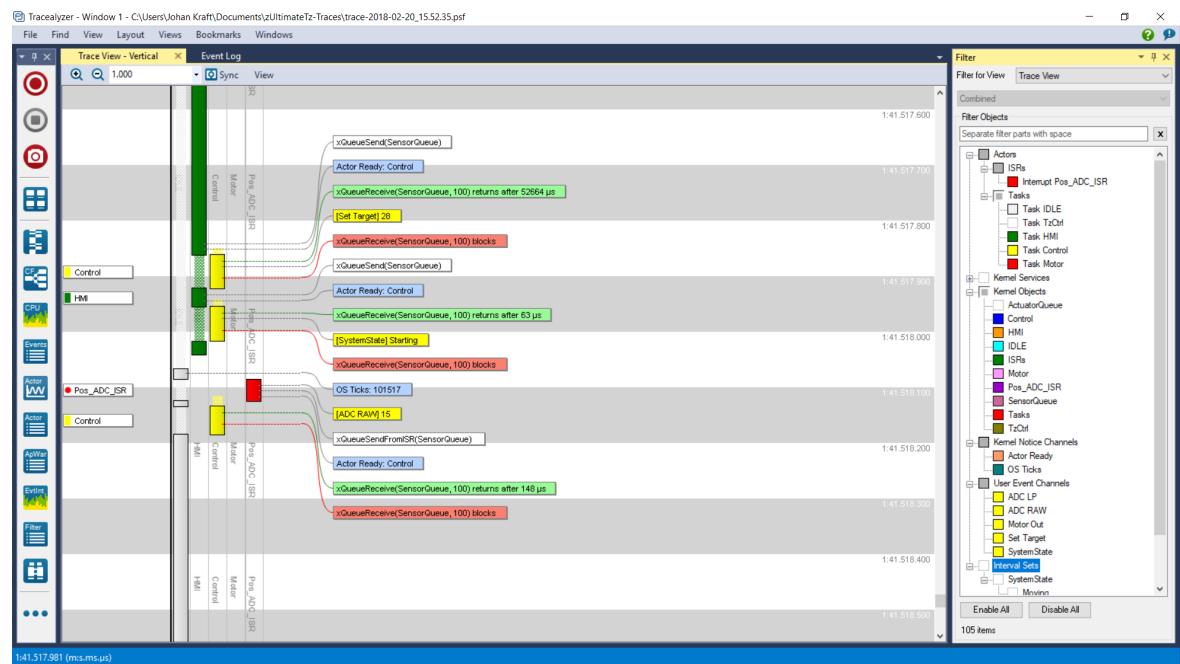
当分析、故障排除或只是优化 FreeRTOS 应用程序时，捕获的信息很有用。

Tracealyzer 可与传统的调试程序一起使用。它以更高级别、基于时间的视角为调试程序的视图提供补充。

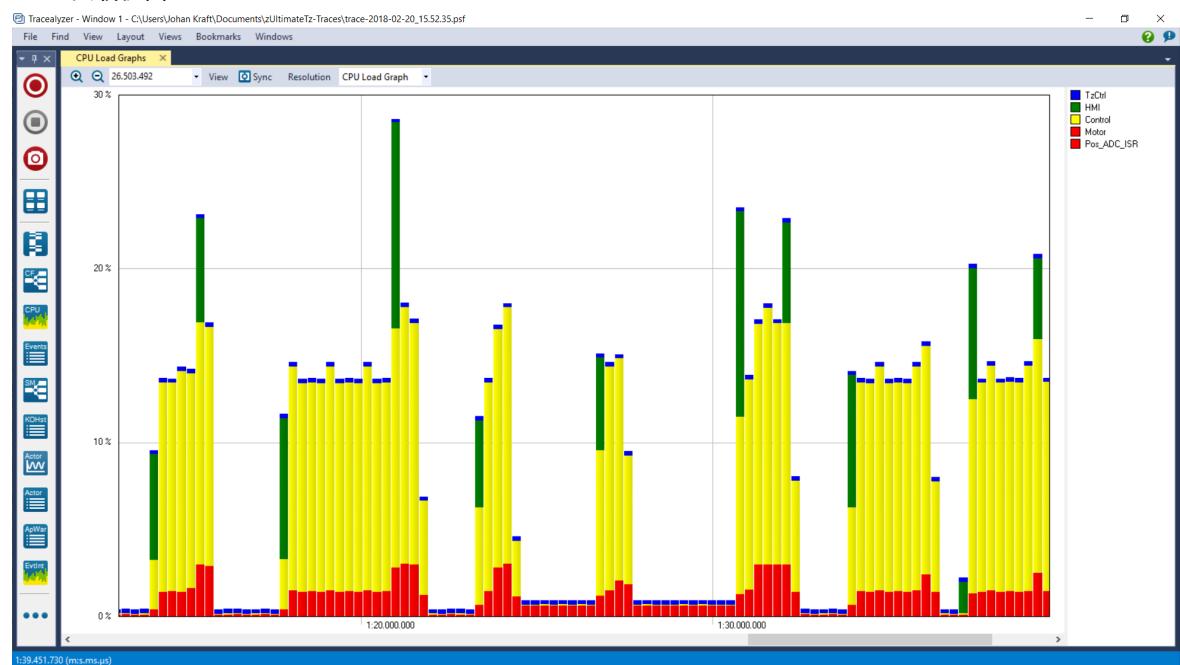
Tracealyzer 包含超过 20 个互连的视图：



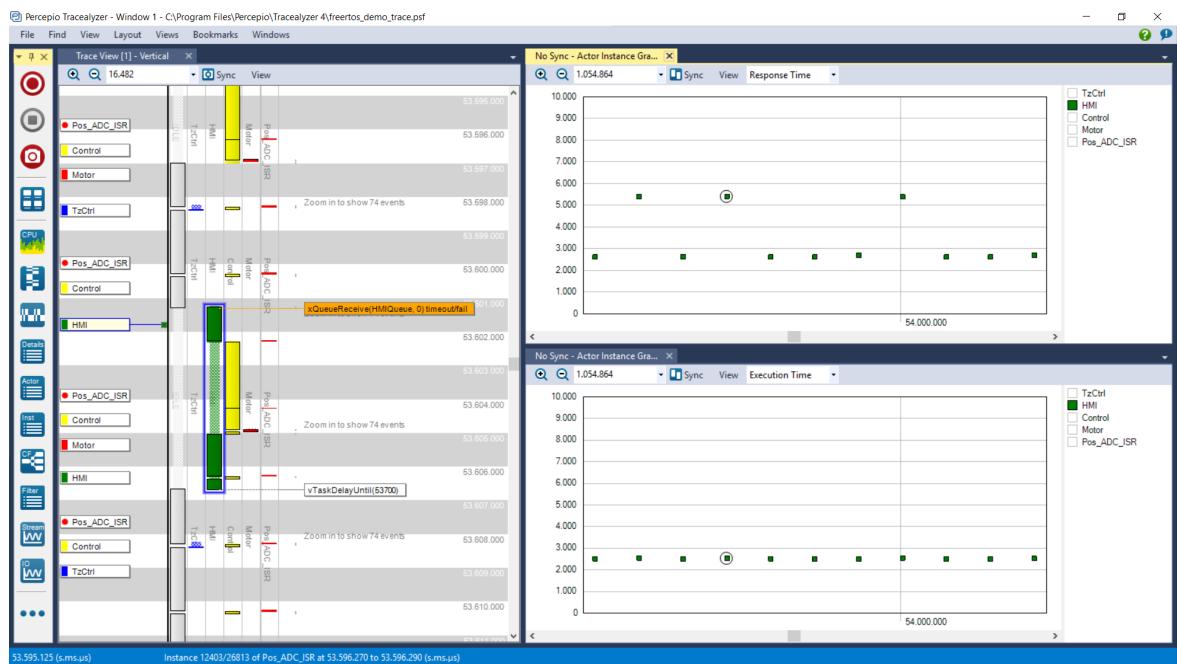
垂直跟踪视图：



CPU 负载视图 :



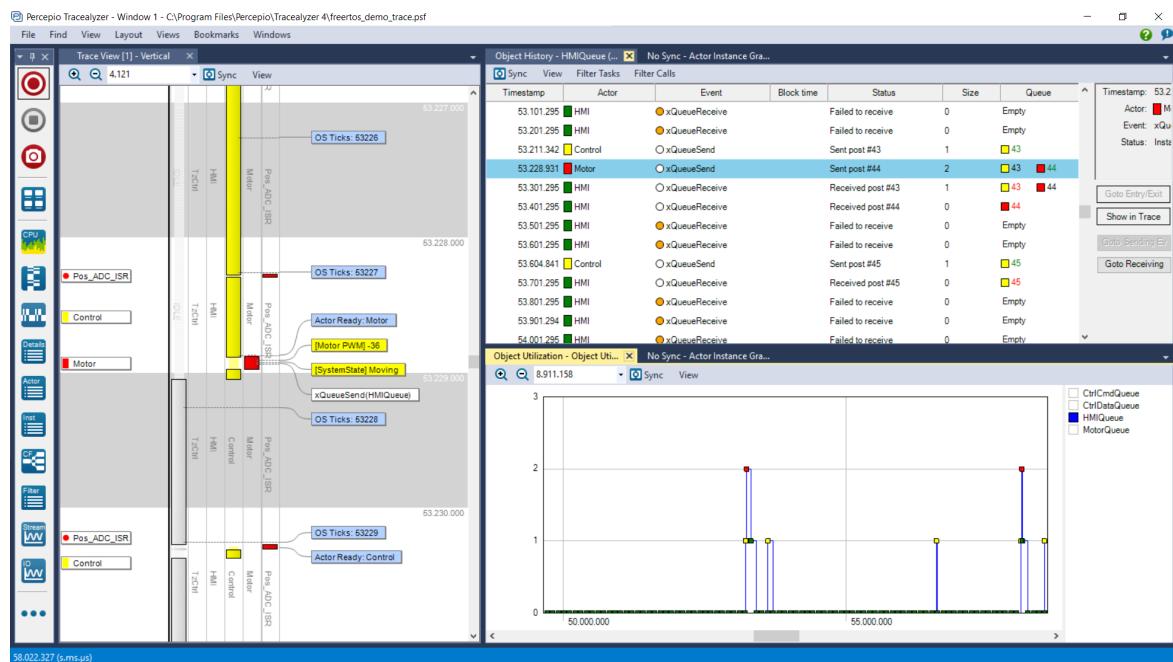
响应时间视图 :



用户事件视图 :



对象历史记录视图 :



## 调试相关的挂钩（回调）函数

如果定义 malloc 失败的挂钩，则可确保在尝试创建任务、队列、信号灯或事件组失败时，应用程序开发人员立即得到通知。有关 malloc 失败挂钩（或回调）的更多信息，请参阅[堆内存管理 \(p. 12\)](#)。

如果定义堆栈溢出挂钩，则可确保当任务所使用的堆栈量超过分配给任务的堆栈空间时，应用程序开发人员得到通知。有关堆栈溢出挂钩的更多信息，请参阅“故障排除”的[堆栈溢出 \(p. 217\)](#)部分。

## 查看运行时和任务状态信息

### 任务运行时统计数据

任务运行时统计数据提供有关每个任务收到的处理时间量的信息。任务的运行时是自应用程序启动后任务已处于“正在运行”状态的总时间。

运行时统计数据的作用是在项目的开发阶段用作分析和调试帮助。它们提供的信息仅当用作运行时统计数据时钟的计数器溢出时才有效。收集运行时统计数据将增加任务上下文切换时间。

要获取二进制文件运行时统计数据信息，请调用 `uxTaskGetSystemState()` API 函数。要获取运行时统计数据信息作为人类可读格式的 ASCII 表，请调用 `vTaskGetRunTimeStats()` 帮助程序函数。

### 运行时统计数据时钟

运行时统计数据需要测量计时周期的小数部分。因此，RTOS 计时计数不用作运行时统计数据时钟。而时钟是由应用程序代码提供。我们建议您让运行时统计数据时钟的频率比计时中断的频率快 10 到 100 倍。运行时统计数据时钟的速度越快，统计数据就越精确，但时间值也会越早溢出。

理想情况下，时间值将由自由运行的 32 位外围设备计时器/计数器生成，而读取其值时不需要其他处理开销。如果可用的外围设备和时钟速度无法实现该方法，则备用但效率不太高的方法包括：

1. 将外围设备配置为以所需的运行时统计数据时钟频率生成定期中断，然后将所生成的中断数计数用作运行时统计数据时钟。

如果定期中断仅用于提供运行时统计数据时钟的目的，则此方法非常低效。但是，如果应用程序已经以合适的频率使用定期中断，则将所生成的中断数计数添加到现有的中断服务例程会很简单高效。

2. 通过将自由运行的 16 位外围设备计数器的当前值用作为 32 位值的最低有效 16 位，并将计时器溢出的次数用作 32 位值的最高有效 16 位，以生成此 32 位值。

您可以通过稍微复杂的操作来生成运行时统计数据时钟：将 RTOS 计时计数与 ARM Cortex-M SysTick 计时器的当前值相结合。FreeRTOS 下载内容中的某些演示项目向您演示了如何执行此操作。

## 将应用程序配置为收集运行时统计数据

下表列出了收集任务运行时统计数据所需的宏。这些宏的前缀均为“port”，因为它们最初设计为包含在 RTOS 端口层中。在 FreeRTOSConfig.h 中定义它们则更实用。

宏	描述
configGENERATE_RUN_TIME_STATS	在 FreeRTOSConfig.h 中，此宏必须设置为 1。当此宏设置为 1 时，计划程序将在适当的时间调用此表中详细介绍的其他宏。
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	必须提供此宏，以初始化用于提供运行时统计数据时钟的外围设备。
portGET_RUN_TIME_COUNTER_VALUE()，或	必须提供这两个宏之一，以便返回当前运行时统计数据时钟值。这是应用程序自首次启动后一直保持运行状态的总时间（采用运行时统计数据时钟单位）。
portALT_GET_RUN_TIME_COUNTER_VALUE(Time)	如果使用第一个宏，则它必须定义为通过求值获得当前时钟值。如果使用第二个宏，则它必须定义为将“Time”参数设置为当前时钟值。

## uxTaskGetSystemState() API 函数

uxTaskGetSystemState() 为在 FreeRTOS 计划程序控制下的每个任务提供状态信息的快照。信息作为 TaskStatus\_t 结构的数组提供，在该数组中，每个任务都有一个索引。下面是 uxTaskGetSystemState() API 函数原型。

下表列出了 uxTaskGetSystemState() 参数和返回值。

参数名称	描述
pxTaskStatusArray	指向 TaskStatus_t 结构数组的指针。  此数组对于每个任务必须包含至少一个 TaskStatus_t 结构。可以使用 uxTaskGetNumberOfTasks() API 函数来确定任务数。  TaskStatus_t 结构如以下代码所示。下表介绍了 TaskStatus_t 结构的成员。

uxArraySize	pxTaskStatusArray 参数所指向的数组的大小。 大小指定为数组中的索引数量（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，则 <b>&lt;problematic&gt;*&lt;/problematic&gt;</b> uxTaskGetSystemState() 将 pulTotalRunTime 设置为自目标启动后的总运行时（由应用程序提供的运行时统计数据时钟定义）。  pulTotalRunTime 是可选的，如果总运行时不是必需的，则可设置为 NULL。
返回的值	返回已由 uxTaskGetSystemState() 填充的 TaskStatus_t 结构的数量。  返回的值应等于由 uxTaskGetNumberOfTasks() API 函数返回的数量，但如果 uxArraySize 参数中传递的值太小，则将为零。

以下是 TaskStatus\_t 结构。

```
typedef struct xTASK_STATUS
{
    TaskHandle_t xHandle;
    const char *pcTaskName;
    UBaseType_t xTaskNumber;
    eTaskState eCurrentState;
    UBaseType_t uxCurrentPriority;
    UBaseType_t uxBasePriority;
    uint32_t ulRunTimeCounter;
    uint16_t usStackHighWaterMark;
} TaskStatus_t;
```

下表列出了 TaskStatus\_t 结构的成员。

参数名称/返回的值	描述
xHandle	与结构中的信息相关的任务的句柄。
pcTaskName	任务的人类可读的文本名称。
xTaskNumber	每个任务都有唯一的 xTaskNumber 值。
	如果应用程序在运行时创建和删除任务，则任务可能与之前删除的任务具有相同的句柄。提供

	xTaskNumber 可使应用程序代码和内核感知调试程序能够区分仍有效的任务和与有效任务具有相同句柄的已删除任务。
eCurrentState	保留任务的状态的枚举类型。eCurrentState 可以是以下值之一：eRunning、eReady、eBlocked、eSuspended、eDeleted。  任务只会在一个非常短的期间内被报告为处于 eDeleted 状态，这个期间从任务被 vTaskDelete() 调用删除的时间开始，到闲置任务释放已分配给该已删除任务的内部数据结构和堆栈的内存时间结束。之后，该任务将不再以任何方式存在，对使用其句柄所做的任何尝试都将是无效的。
uxCurrentPriority	在调用 uxTaskGetSystemState() 时运行任务所采用的优先级。仅当根据 <a href="#">资源管理 (p. 141)</a> 中介绍的优先级继承机制临时向任务分配更高优先级时，uxCurrentPriority 才会高于应用程序编写者分配给任务的优先级。
uxBasePriority	应用程序编写者分配给任务的优先级。仅当在 FreeRTOSConfig.h 中将 configUSE_MUTEXES 设置为 1 时，uxBasePriority 才有效。
ulRunTimeCounter	任务自创建后所使用的总运行时。总运行时作为一个绝对时间提供，该时间使用应用程序编写者为收集运行时统计数据而提供的时钟。仅当在 FreeRTOSConfig.h 中将 configGENERATE_RUN_TIME_STATS 设置为 1 时，ulRunTimeCounter 才有效。
usStackHighWaterMark	任务的堆栈高水位。这是自任务创建后为任务保留的最小堆栈空间量。它指示任务接近溢出其堆栈的程度。此值越接近零，则任务越接近溢出其堆栈。usStackHighWaterMark 以字节为单位指定。

## vTaskList() 帮助程序函数

vTaskList() 提供的任务状态信息与 uxTaskGetSystemState() 提供的信息类似，但它以人类可读的 ASCII 表提供信息，而不是二进制值的数组。

vTaskList() 是一个对处理器要求非常高的函数。它会让计划程序暂停一段较长的时间。因此，我们建议您仅出于调试目的使用此函数，而不是在生产实时系统中使用它。

如果在 FreeRTOSConfig.h 中同时将 configUSE\_TRACE\_FACILITY 和 configUSE\_STATS\_FORMATTING\_FUNCTIONS 设置为 1，则 vTaskList() 可用。

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

下表中列出了 vTaskList() 参数。

参数名称	描述

pcWriteBuffer

指向一个字符缓冲区的指针，将向此字符缓冲区中写入格式化和人类可读的表。该缓冲区必须足够大以容纳整个表。不执行边界检查。

下面是 vTaskList() 生成的输出。

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

在输出中：

- 每行均提供有关单个任务的信息。
- 第一列是任务的名称。
- 第二列是任务的状态，其中“R”表示 Ready（准备就绪），“B”表示 Blocked（被阻止），“S”表示 Suspended（已暂停），而“D”表示任务已被删除。任务只会在非常短的期间内被报告为处于已删除状态，该期间从任务被 vTaskDelete() 调用删除的时间开始，到闲置任务释放已分配给该已删除任务的内部数据结构和堆栈的内存的时间结束。之后，该任务将不再以任何方式存在，对使用其句柄所做的任何尝试都将是无效的。
- 第三列是任务的优先级。
- 第四列是任务的堆栈“高水位”。有关 TaskStatus\_t 结构成员，请参阅该表中 usStackHighWaterMark 的描述。
- 第五列是分配给任务的唯一编号。有关 TaskStatus\_t 结构成员，请参阅该表中 xTaskNumber 的描述。

## vTaskGetRunTimeStats() 帮助程序函数

vTaskGetRunTimeStats() 将收集的运行时统计数据的格式设置为人类可读的 ASCII 表。

vTaskGetRunTimeStats() 是一个对处理器要求非常高的函数。它会让计划程序暂停一段较长的时间。因此，我们建议您仅出于调试目的使用此函数，而不在生产实时系统中使用它。

当在 FreeRTOSConfig.h 中同时将 configGENERATE\_RUN\_TIME\_STATS 和 configUSE\_STATS\_FORMATTING\_FUNCTIONS 设置为 1 时，vTaskGetRunTimeStats() 可用。

下面是 vTaskGetRunTimeStats() API 函数原型。

```
void vTaskGetRunTimeStats( signed char *pcWriteBuffer );
```

下表中列出了 vTaskGetRunTimeStats() 参数。

参数名称	描述
pcWriteBuffer	指向一个字符缓冲区的指针 , 将向此字符缓冲区中写入格式化和人类可读的表。该缓冲区必须足够大以容纳整个表。不执行边界检查。

下面是 vTaskGetRunTimeStats() 生成的输出。

P01SEM1	994	<1%
P01SEM2	23248	1%
GenQ	194479	16%
MuLow	3690	<1%
Rec3	229450	18%
CNT1	242720	19%
PeekL	94	<1%
CNT_INC	165	<1%
CNT2	243166	20%
SUSP_RX	243192	20%
IDLE	55	<1%

在输出中 :

- 每行均提供有关单个任务的信息。
- 第一列是任务名称。
- 第二列是任务处于“正在运行”状态的时间（绝对值）。有关 TaskStatus\_t 结构成员，请参阅该表中 ulRunTimeCounter 的描述。
- 第三列是相对于启动目标后的总时间，任务处于“正在运行”状态的时间百分比。显示的百分比时间的总和通常低于预期的 100%，因为收集和计算统计数据时使用的整数计算向下舍入到最接近的整数值。

## 生成和显示运行时统计数据 , 一个工作示例

本示例使用假定的 16 位计时器生成 32 位运行时统计数据时钟。计数器配置为在每次 16 位值达到其最大值时生成中断，从而有效地创建溢出中断。中断服务例程对溢出发生的次数进行计数。

创建 32 位值时，将溢出发生次数计数用作 32 位值的两个最高有效字节，并将当前 16 位计数器值用作 32 位值的两个最低有效字节。下面是中断服务例程的伪代码。

```
void TimerOverflowInterruptHandler( void )
{
    /* Just count the number of interrupts. */
    ulOverflowCount++;

    /* Clear the interrupt. */
    ClearTimerInterrupt();
}
```

```
}
```

下面是启用收集运行时统计数据的代码。

```
/* Set configGENERATE_RUN_TIME_STATS to 1 to enable collection of runtime
statistics. When this is done, both portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and
portGET_RUN_TIME_COUNTER_VALUE() or portALT_GET_RUN_TIME_COUNTER_VALUE(x) must also be
defined. */

#define configGENERATE_RUN_TIME_STATS 1

/* portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() is defined to call the function that sets up
the hypothetical 16-bit timer. (The function's implementation is not shown.) */

void vSetupTimerForRunTimeStats( void );

#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()

vSetupTimerForRunTimeStats()

/* portALT_GET_RUN_TIME_COUNTER_VALUE() is defined to set its parameter to the current
runtime counter/time value. The returned time value is 32-bits long, and is formed by
shifting the count of 16-bit timer overflows into the top two bytes of a 32-bit number,
and then bitwise ORing the result with the current 16-bit counter value. */

#define portALT_GET_RUN_TIME_COUNTER_VALUE( ulCountValue ) \\\
{

    extern volatile unsigned long ulOverflowCount;

    /* Disconnect the clock from the counter so it does not change while its value is being
used. */

    PauseTimer();

    /* The number of overflows is shifted into the most significant two bytes of the
returned 32-bit value. */

    ulCountValue = ( ulOverflowCount << 16UL );

    /* The current counter value is used as the least significant two bytes of the returned
32-bit value. */

    ulCountValue |= ( unsigned long ) ReadTimerCount();

    /* Reconnect the clock to the counter. */

    ResumeTimer(); \\

}
```

下面这个任务每 5 秒输出收集的运行时统计数据。

```
/* For clarity, calls to fflush() have been omitted from this codelist. */

static void prvStatsTask( void *pvParameters )
```

```
{  
    TickType_t xLastExecutionTime;  
  
    /* The buffer used to hold the formatted runtime statistics text needs to be quite  
    large. It is therefore declared static to ensure it is not allocated on the task stack.  
    This makes this function non-reentrant. */  
  
    static signed char cStringBuffer[ 512 ];  
  
    /* The task will run every 5 seconds. */  
  
    const TickType_t xBlockPeriod = pdMS_TO_TICKS( 5000 );  
  
    /* Initialize xLastExecutionTime to the current time. This is the only time this  
    variable needs to be written to explicitly. Afterward, it is updated internally within the  
    vTaskDelayUntil() API function. */  
  
    xLastExecutionTime = xTaskGetTickCount();  
  
    /* As per most tasks, this task is implemented in an infinite loop. */  
  
    for( ;; )  
  
    {  
  
        /* Wait until it is time to run this task again. */  
  
        vTaskDelayUntil( &xLastExecutionTime, xBlockPeriod );  
  
        /* Generate a text table from the runtime stats. This must fit into the  
        cStringBuffer array. */  
  
        vTaskGetRunTimeStats( cStringBuffer );  
  
        /* Print out column headings for the runtime stats table. */  
  
        printf( "\nTask\t\tAbs\t\t%\n" );  
  
        printf( "-----\n" );  
  
        /* Print out the runtime stats themselves. The table of data contains multiple  
        lines, so the vPrintMultipleLines() function is called instead of calling printf()  
        directly. vPrintMultipleLines() simply calls printf() on each line individually, to ensure  
        the line buffering works as expected. */  
  
        vPrintMultipleLines( cStringBuffer );  
  
    }  
}
```

## 跟踪挂钩宏

跟踪宏已放在 FreeRTOS 源代码中的关键点处。默认情况下，宏是空的，因此它们不会生成任何代码且没有运行时开销。通过覆盖默认的空实现，应用程序编写者可以：

- 将代码插入到 FreeRTOS 中，而无需修改 FreeRTOS 源文件。
- 通过目标硬件上可用的任何方式输出详细的执行顺序信息。跟踪宏出现在 FreeRTOS 源代码中足够多的位置，以允许使用它们创建完整、详细的计划程序活动跟踪和分析日志。

## 可用的跟踪挂钩宏

下表列出了对应用程序编写者最有用的宏。

此表中的许多描述指的是一个名为 pxCurrentTCB 的变量。pxCurrentTCB 是一个 FreeRTOS 私有变量，用于存放处于运行状态的任务的句柄。它可用于从 FreeRTOS/Source/tasks.c 源文件中调用的任何宏。

宏	描述
traceTASK_INCREMENT_TICK(xTickCount)	在计时中断期间调用（计时计数递增之后）。xTickCount 参数将新的计时计数值传递到宏。
traceTASK_SWITCHED_OUT()	在选择要运行的新任务之前调用。此时，pxCurrentTCB 包含要离开“正在运行”状态的任务的句柄。
traceTASK_SWITCHED_IN()	在选择要运行的任务之后调用。此时，pxCurrentTCB 包含要进入“正在运行”状态的任务的句柄。
traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue)	在尝试从空队列读取或者尝试“获取”空的信号灯或互斥后，在当前正在执行的任务进入“被阻止”状态前调用。pxQueue 参数将目标队列或信号灯的句柄传递到宏。
traceBLOCKING_ON_QUEUE_SEND(pxQueue)	在尝试写入已满的队列后，在当前正在执行的任务进入“被阻止”状态前调用。pxQueue 参数将目标队列的句柄传递到宏。
traceQUEUE_SEND(pxQueue)	当队列发送或信号灯“释放”取得成功时，从 xQueueSend()、xQueueSendToFront()、xQueueSendToBack() 或任何信号灯“释放”函数中调用。pxQueue 参数将目标队列或信号灯的句柄传递到宏。
traceQUEUE_SEND_FAILED(pxQueue)	当队列发送或信号灯“释放”操作失败时，从 xQueueSend()、xQueueSendToFront()、xQueueSendToBack() 或任何信号灯“释放”函数中调用。如果队列已满并在任何指定的阻止时间段内保持已满状态，队列发送或信号灯“释放”将失败。pxQueue 参数将目标队列或信号灯的句柄传递到宏。
traceQUEUE_RECEIVE(pxQueue)	当队列接收或信号灯“获取”取得成功时，从 xQueueReceive() 或任何信号灯“获取”函数中调用。pxQueue 参数将目标队列或信号灯的句柄传递到宏。
traceQUEUE_RECEIVE_FAILED(pxQueue)	当队列接收或信号灯接收操作失败时，从 xQueueReceive() 或任何信号灯“获取”函数中调用。如果队列或信号灯为空并在任何指定的阻止时间段内保持为空状态，队列接收或信号灯“获取”操作将失败。pxQueue 参数将目标队列或信号灯的句柄传递到宏。
traceQUEUE_SEND_FROM_ISR(pxQueue)	当发送操作成功时，从 xQueueSendFromISR() 中调用。pxQueue 参数将目标队列的句柄传递到宏。

traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)	当发送操作失败时，从 xQueueSendFromISR() 中调用。如果队列已满，发送操作将失败。pxQueue 参数将目标队列的句柄传递到宏。
traceQUEUE_RECEIVE_FROM_ISR(pxQueue)	当接收操作成功时，从 xQueueReceiveFromISR() 中调用。pxQueue 参数将目标队列的句柄传递到宏。
traceQUEUE_RECEIVE_FROM_ISR_FAILED(pxQueue)	当接收操作由于队列已为空而失败时，从 xQueueReceiveFromISR() 中调用。pxQueue 参数将目标队列的句柄传递到宏。
traceTASK_DELAY_UNTIL()	就在发出调用的任务进入“被阻止”状态前从 vTaskDelayUntil() 中调用。
traceTASK_DELAY()	就在发出调用的任务进入“被阻止”状态前从 vTaskDelay() 中调用。

## 定义跟踪挂钩宏

每个跟踪宏都有默认的空定义。您可以通过在 FreeRTOSConfig.h 中提供新的宏定义来覆盖默认定义。如果跟踪宏定义变长或变复杂，则可以在新的标题文件中实现它们，然后将文件本身包含在 FreeRTOSConfig.h 中。

根据软件工程最佳实践，FreeRTOS 遵守严格的数据隐藏政策。跟踪宏可让用户代码添加到 FreeRTOS 源文件中，因此，跟踪宏可见的数据类型将不同于应用程序代码可见的数据类型：

- 在 FreeRTOS/Source/tasks.c 源文件内部，任务句柄是一个指针，指向描述任务的数据结构。这是任务的任务控制块 (TCB)。在 FreeRTOS/Source/tasks.c 源文件外部，任务句柄是一个指向 void 的指针。
- 在 FreeRTOS/Source/queue.c 源文件内部，任务句柄是一个指针，指向描述队列的数据结构。在 FreeRTOS/Source/queue.c 源文件外部，队列句柄是一个指向 void 的指针。

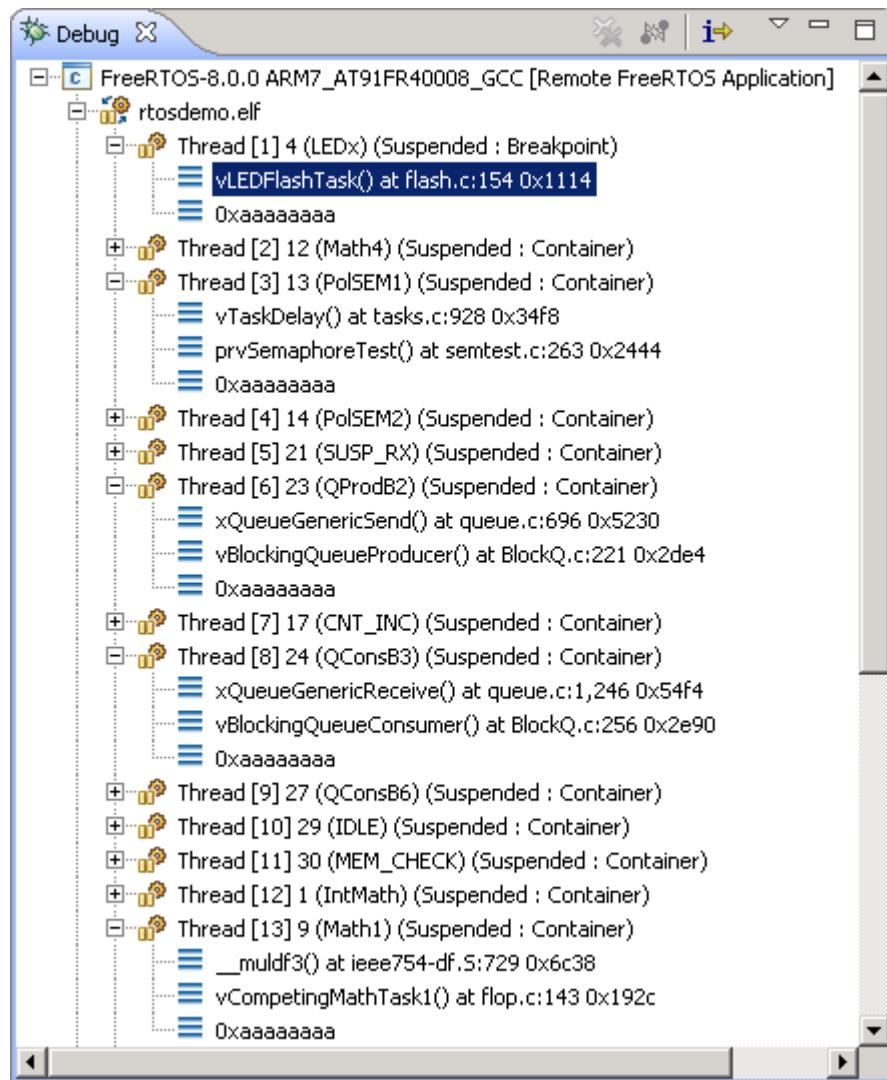
如果通过跟踪宏直接访问通常为私有的 FreeRTOS 数据结构，则务必小心。私有数据结构在 FreeRTOS 版本之间可能会发生变化。

## FreeRTOS 感知调试程序插件

提供一些 FreeRTOS 感知的插件可用于以下 IDE。此列表并不详尽。

- Eclipse (StateViewer)
- Eclipse (ThreadSpy)
- IAR
- ARM DS-5
- Atollic TrueStudio
- Microchip MPLAB
- iSYSTEM WinIDEA

下图显示了 Code Confidence Ltd. 提供的 FreeRTOS ThreadSpy Eclipse 插件。



# 疑难解答

## 本章简介和范围

本节介绍新 FreeRTOS 用户最常遇到的问题，具体为：

- 不正确的中断优先级分配
- 堆栈溢出
- 不当使用 printf()。

使用 configASSERT() 可立即捕获和确定很多最常见的错误来源，从而提高效率。我们强烈建议您在开发或调试 FreeRTOS 应用程序时定义 configASSERT()。有关 configASSERT() 的更多信息，请参阅[开发人员支持 \(p. 202\)](#)。

## 中断优先级

注意：这是引起支持请求的首要原因。在大多数端口中，定义 configASSERT() 会立即捕获错误。

如果使用的 FreeRTOS 端口支持中断嵌套，并且中断的服务例程使用 FreeRTOS API，则必须将中断优先级设置为 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 或更低，如[中断管理 \(p. 108\)](#)中所述。如果不正确设置优先级，关键部分将会低效，进而导致间歇性故障。

如果在符合以下条件的处理器上运行 FreeRTOS，请注意：

- 中断优先级默认具有可能的最高优先级，例如在某些 ARM Cortex 处理器上。在这些处理器上，使用 FreeRTOS API 的中断的优先级不能保持未初始化状态。
- 高数值优先级数字表示低逻辑中断优先级，这看起来有违直觉。同样，这种情况存在于 ARM Cortex 处理器等处理器上。
- 例如，在这类处理器上，优先级为 4 的中断可以中断以优先级 5 执行的中断。因此，如果 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 设置为 5，则只能为使用该 FreeRTOS API 的所有中断分配数值上大于或等于 5 的优先级。在这种情况下，中断优先级 5 或 6 有效，但中断优先级 3 肯定无效。
- 不同的库实现需要以不同的方式指定中断优先级。同样，这与面向 ARM Cortex 处理器的库尤其相关，在这种情况下，中断优先级先位移，然后再写入硬件寄存器。有些库自己执行位移，另一些库需要在优先级传入库函数之前执行位移。
- 在相同架构的不同实现中，实现的中断优先级位的数量不相同。例如，某个制造商的 Cortex-M 处理器可能实现 3 个优先级位，另一个制造商的 Cortex-M 处理器可能实现 4 个优先级位。
- 定义中断优先级的位可以拆分为定义抢先优先级的位和定义子优先级的位。请务必分配所有位以指定抢先优先级，以便不使用子优先级。

在某些 FreeRTOS 端口中，configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 有另一个名称 configMAX\_API\_CALL\_INTERRUPT\_PRIORITY。

## 堆栈溢出

堆栈溢出是引起支持请求的第二重要原因。FreeRTOS 提供了几种功能来帮助捕获和调试与堆栈相关的问题。（这些功能在 FreeRTOS Windows 端口上不可用。）

## uxTaskGetStackHighWaterMark() API 函数

每个任务都维护自己的堆栈，堆栈的总大小是在任务创建时指定的。uxTaskGetStackHighWaterMark() 用于查询任务接近溢出分配给它的堆栈空间的程度。此值称为堆栈的高水位。

下面是 uxTaskGetStackHighWaterMark() API 函数原型。

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

下表列出了 uxTaskGetStackHighWaterMark() 参数和返回值。

参数名称/返回的值	描述
xTask	要查询其堆栈高水位的任务（主体任务）的句柄。 有关获取任务句柄的信息，请参阅 xTaskCreate() API 函数的 pxCreatedTask 参数。任务可以通过传递 NULL 而不是有效任务句柄来查询其自己的堆栈 高水位。
返回的值	任务使用的堆栈的大小随着任务 的执行和中断的处理而增大和缩 小。uxTaskGetStackHighWaterMark() 返回自任务 开始执行以来可用的最小剩余堆栈空间大小。这是 堆栈使用率位于最大（即最深）值时仍未使用的 堆栈的大小。高水位越接近于零，任务就越近于溢出 其堆栈。

## 运行时堆栈检查概述

FreeRTOS 包含两种可选的运行时堆栈检查机制。它们由 FreeRTOSConfig.h 中的 configCHECK\_FOR\_STACK\_OVERFLOW 编译时间配置常量控制。这两种方法都会增加执行上下文切换所需的时间。

堆栈溢出挂钩（即堆栈溢出回调）是内核在检测到堆栈溢出时调用的函数。要使用堆栈溢出挂钩函数，请执行以下操作：

1. 在 FreeRTOSConfig.h 中将 configCHECK\_FOR\_STACK\_OVERFLOW 设置为 1 或 2。
2. 使用下面的函数名称和原型提供挂钩函数的实现。

```
void vApplicationStackOverflowHook( TaskHandle_t *pxTask, signed char*pcTaskName );
```

提供堆栈溢出挂钩是为了更轻松地捕获和调试堆栈错误，但如果发生堆栈溢出，实际上没有办法恢复。此函数的参数将溢出其堆栈的任务的句柄和名称传递给挂钩函数。

堆栈溢出挂钩从中断上下文中调用。

有些微控制器在检测到不正确的内存访问时会生成故障异常。在内核有机会调用堆栈溢出挂钩函数之前，可能会触发故障。

## 运行时堆栈检查方法 1

方法 1 是快速执行方法，但可能会错过上下文切换之间发生的堆栈溢出。此方法在 configCHECK\_FOR\_STACK\_OVERFLOW 设置为 1 时使用。

每次换出时，任务的整个执行上下文都会保存到其堆栈上。堆栈使用率可能在此时达到峰值。当 configCHECK\_FOR\_STACK\_OVERFLOW 设置为 1 时，内核在保存上下文后检查堆栈指针是否保持在有效堆栈空间内。如果发现堆栈指针超出其有效范围，则调用堆栈溢出挂钩。

## 运行时堆栈检查方法 2

方法 2 执行额外检查。此方法在 configCHECK\_FOR\_STACK\_OVERFLOW 设置为 2 时使用。

任务创建时，会以已知模式填充其堆栈。方法 2 测试任务堆栈空间的最后 20 个有效字节以验证此模式未被覆盖。如果这 20 个字节中有任一字节从其预期值发生了更改，则调用堆栈溢出挂钩函数。

方法 2 的执行速度没有方法 1 快，但也是相对快的，因为只测试 20 个字节。它几乎能捕获所有堆栈溢出。

## 不当使用 printf() 和 sprintf()

不当使用 printf() 是一种常见的错误原因。如果没有意识到这一点，应用程序开发人员有时会添加更多 printf() 调用以帮助调试，在此过程中反而加剧了问题。

很多跨编译器供应商会提供一个适用于小型嵌入式系统的 printf() 实现。即使这样，该实现也可能不是线程安全的，可能不适合在中断服务例程内部使用，并且根据输出定向，可能需要相对较长的时间执行。

如果专为小型嵌入式系统设计的 printf() 实现不可用，请谨慎使用泛型 printf() 实现，因为：

- 只包含一个 printf() 或 sprintf() 调用会大幅增加应用程序的可执行文件的大小。
- printf() 和 sprintf() 可能调用 malloc()，如果使用 heap\_3 以外的内存分配方案，这是无效的。有关更多信息，请参阅[示例内存分配方案 \(p. 13\)](#)。
- printf() 和 sprintf() 需要的堆栈可能比其他方法需要的堆栈大很多倍。

## Printf-stdarg.c

很多 FreeRTOS 演示项目使用一个名为 printf-stdarg.c 的文件，它提供最小并且具有堆栈效率的 sprintf() 实现，可以替代标准库版本使用。在大多数情况下，这样可以为调用 sprintf() 和相关函数的每个任务分配小得多的堆栈。

printf-stdarg.c 还提供一种机制，可以将 printf() 输出逐字符定向到某个端口。虽然较慢，但是这可以进一步降低堆栈使用率。

并非所有 printf-stdarg.c 副本都实现 snprintf()。不实现 snprintf() 的副本直接忽略缓冲区大小参数，因为它们直接映射到 sprintf()。

Printf-stdarg.c 是开源的，但属于第三方，因此独立于 FreeRTOS 提供许可证。许可条款包含在源文件顶部。

## 其他常见错误

本节介绍其他常见错误、其可能的原因及其解决方案。

### 症状：向演示添加简单任务导致演示崩溃

创建任务需要从堆获取内存。很多演示应用程序项目将堆大小设置为刚好足够创建演示任务，这样，在任务创建后，剩余的堆将不足以添加更多任务、队列、事件组或信号灯。

空闲任务和 RTOS 守护程序任务是在调用 vTaskStartScheduler() 时自动创建的。仅当剩余堆内存不足以创建这些任务时，vTaskStartScheduler() 才会返回。在 vTaskStartScheduler() 调用后包含一个 null 循环 [ for(;;) ] 可使此错误更易于调试。

要添加更多任务，可以增加堆大小或删除一些现有演示任务。有关更多信息，请参阅[示例内存分配方案 \(p. 13\)](#)。

## 症状：在中断中使用 API 函数导致应用程序崩溃

除非 API 函数的名称以“...FromISR()”结尾，否则请勿在中断服务例程中使用 API 函数。具体而言，除非使用中断安全宏，否则不要在中断中创建关键部分。有关更多信息，请参阅[中断管理 \(p. 108\)](#)。

在支持中断嵌套的 FreeRTOS 端口中，不要在已分配了高于 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 的中断优先级的中断中使用 API 函数。有关更多信息，请参阅[中断嵌套 \(p. 137\)](#)。

## 症状：应用程序有时在中断服务例程中崩溃

首先检查中断是否导致堆栈溢出。某些端口只在任务中检查堆栈溢出，而不是在中断中检查。

中断的定义和使用方式因端口和编译器而异。因此，其次检查中断服务例程中使用的语法、宏和调用惯例，看它们是否与为所使用的端口提供的文档页面上的描述完全一致，是否与随端口提供的演示应用程序中所示的完全一致。

如果运行应用程序的处理器使用低数值优先级数字表示高逻辑优先级，请为分配给每个中断任务的优先级考虑到这一点，因为这看起来有违直觉。如果运行应用程序的处理器将每个中断的优先级默认为可能的最大优先级，请确保不要将每个中断的优先级保留为其默认值。有关更多信息，请参阅[中断嵌套 \(p. 137\)](#)。

## 症状：计划程序在尝试启动第一个任务时崩溃

请确保安装了 FreeRTOS 中断处理程序。有关示例，请参阅为端口提供的演示应用程序。

某些处理器必须处于特权模式，才能启动计划程序。实现这一点的最简单方法是在 C 启动代码中调用 main() 前将处理器置于特权模式。

## 症状：中断被意外保持为禁用或关键部分嵌套不正确

如果在启动计划程序前调用 FreeRTOS API 函数，会有意将中断保持为禁用。直到第一个任务开始执行，才会重新启用。这可以保护系统不会因中断在系统初始化期间、计划程序启动之前、计划程序处于不一致状态时尝试使用 FreeRTOS API 函数而崩溃。

使用 taskENTER\_CRITICAL() 和 taskEXIT\_CRITICAL() 调用更改微控制器中断启用位或优先级标志。请勿使用任何其他方法。这些宏保留其调用嵌套深度计数以确保仅当调用嵌套完全展开到零时才再次启用中断。请注意，某些库函数可能会启用和禁用中断。

## 症状：应用程序甚至在计划程序启动之前就崩溃

不要允许可能导致上下文切换的中断服务例程在计划程序启动前执行。这同样适用于尝试与 FreeRTOS 对象（如队列或信号灯）进行数据收发的任何中断服务例程。计划程序启动后，上下文切换才能发生。

在计划程序启动前，很多 API 函数不能调用。最好将 API 使用限于对象（如任务、队列和信号灯）的创建，而不要在 vTaskStartScheduler() 调用之前使用这些对象。

## 症状：在计划程序暂停时调用 API 函数或从关键部分内 部调用 API 函数导致应用程序崩溃

调用 vTaskSuspendAll() 可暂停计划程序，调用 xTaskResumeAll() 可恢复（取消暂停）计划程序。调用 taskENTER\_CRITICAL() 可进入关键部分，调用 taskEXIT\_CRITICAL() 可退出关键部分。

请勿从关键部分内部调用 API 函数或在计划程序暂停时调用 API 函数。