**FreeRTOS™内核**[**市场领先的**](https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf)**，事实上的标准和跨平台**[**RTOS**](https://www.freertos.org/about-RTOS.html)**内核**

FreeRTOS与全球领先的芯片公司合作开发了18年，它是用于微控制器和小型微处理器的市场领先的实时操作系统（RTOS）。FreeRTOS根据麻省理工学院的开源许可免费分发，其中包括一个内核和一组不断增长的库，适用于所有行业。FreeRTOS每170秒下载一次，其重点是可靠性，可访问性和易用性。  

|  |
| --- |
| **你知道吗？**   * FreeRTOS每170秒下载一次（平均而言，在2019年期间）。 * **自2011年以来**，**FreeRTOS在所有**[**EETimes嵌入式市场**](https://www.embedded.com/electronics-blogs/embedded-market-surveys/4458724/2017-Embedded-Market-Survey)**调查中均位居同类产品**之首，这是第一年。 * 与商业选择相比，FreeRTOS提供了***更低的项目风险***和***更低的成本，***因为：   + 它[完全支持](https://forums.freertos.org/)并记录在案。   + 大多数人从未与我们联系就将产品推向市场，但是他们完全可以放心，他们可以随时选择使用[完全赔偿的商业许可](https://www.freertos.org/a00114.html#commercial)（在专门的支持下）。 * 一些FreeRTOS接口[从不完全禁止中断](https://www.freertos.org/a00110.html#kernel_priority)。 * 为了严格的质量控制目的，并消除所有IP所有权的歧义，将[FreeRTOS官方代码与社区捐助分开](https://www.freertos.org/differences-between-officially-supported-and-contributed-FreeRTOS-code.html)。 * FreeRTOS具有无滴答模式，可[直接支持低功耗应用](https://www.freertos.org/low-power-tickless-rtos.html)。 * FreeRTOS的设计简单易用：仅需要3个所有RTOS端口共有的源文件，以及一个特定于微控制器的源文件，并且其API的设计也非常简单直观。 * RL78端口可在不到4K字节的RAM中创建13个任务，2个队列和4个软件计时器！ |

**为什么选择FreeRTOS？**

*“在这一点上，可以肯定地说，FreeRTOS经过的“同行评审”要比地球上任何其他RTOS都要多。我已经在多个项目中使用了它-其中一个是使用64个以上处理器的多处理器环境，并且需要可靠地运行数月。RTOS核心表现良好。试一下FreeRTOS。”*–约翰·韦斯特摩兰

**FreeRTOS提供了世界上最好的：** FreeRTOS是真正的免费和[受支持的](https://forums.freertos.org/)，即使用于商业应用中也是如此。该[FreeRTOS的开源MIT许可](https://www.freertos.org/a00114.html)并不要求你暴露你的专有IP。您可以使用FreeRTOS将产品推向市场，而无需与我们联系，更不用说支付任何费用，成千上万的人这样做。如果您希望在任何时候获得额外的备份，或者您的法律团队需要额外的书面担保或赔偿，则可以[使用简单的低成本商业升级途径](https://www.freertos.org/a00114.html#commercial)。您可以选择在任何时候选择采用商业路线，这使您高枕无忧。

[FreeRTOS](https://www.freertos.org/RTOS.html)是您下一个应用程序的理想选择的一些原因-FreeRTOS…

* 为许多不同的体系结构和开发工具提供单一且独立的解决方案。
* 众所周知是可靠的。SafeRTOS姊妹项目开展的活动确保了信心。
* 是[功能丰富](https://www.freertos.org/features.html)且仍处于持续活跃发展。
* 具有最小的ROM，RAM和处理开销。通常，RTOS内核二进制映像的大小在6K到12K字节之间。
* 非常简单– RTOS内核的核心仅包含在[3个C文件中](https://www.freertos.org/a00017.html)。.zip文件下载中包含的许多文件中的大多数仅与众多演示应用程序有关。
* 真正免费用于商业应用程序（有关详细信息，请参阅[许可条件](https://www.freertos.org/a00114.html)）。
* 我们的合作伙伴[WITTENSTEIN高完整性系统](https://www.highintegritysystems.com/)以OPEN **RTOS**的形式提供了商业许可，专业支持和移植服务。
* 具有向[Safe **RTOS**](https://www.highintegritysystems.com/)的迁移途径，其中包括针对医疗，汽车和工业领域的认证。
* 拥有庞大且不断增长的用户基础。
* 包含每个端口的预配置示例。无需弄清楚如何设置项目-只需下载并编译！
* 拥有一个出色的，受监控的，积极的免费[支持论坛](https://forums.freertos.org/)。
* 确保需要时可提供商业支持。
* 提供足够的文档。
* 具有很好的可扩展性，简单易用。
* FreeRTOS为eCOS，嵌入式Linux（或Real Linux）甚至uCLinux不适合，不合适或不可用的应用程序提供了更小，更轻松的实时处理替代方案。

**编码标准，测试和样式指南**

在本页：

* [编码标准](https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html)
* [测验](https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html#Testing)
* [命名约定](https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html#NamingConventions)
* [资料类型](https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html#DataTypes)
* [时尚指南](https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html#StyleGuide)

**编码标准/符合MISRA**

核心FreeRTOS源文件（所有端口共有的文件，而不是端口层通用的文件）符合[MISRA](http://www.misra.org.uk/)编码标准准则。使用[pc-lint](https://www.gimpel.com/html/misra.htm)和[链接的lint配置文件](https://www.freertos.org/lint_configuration.zip)检查是否[合格](https://www.freertos.org/lint_configuration.zip)。由于该标准的页面长很多，并且可以从MISRA处以很小的费用购买，因此这里没有重复所有规则。

下面列出了与MISRA标准的差异：

* 两个API函数具有多个出口点。由于关键效率的原因，在这两种情况下允许出现偏差。
* 创建任务时，源代码会操纵内存地址以定位分配给所创建任务的堆栈的起始地址和结束地址。该代码必须适用于已移植FreeRTOS的所有体系结构-包括具有8、16、20、24和32位总线的体系结构。这不可避免地需要一些指针算法。使用指针算术时，将以编程方式检查算术结果的正确性。
* 跟踪宏默认情况下为空，因此不会生成任何代码。因此，使用伪宏定义执行MISRA合规性检查。
* 视情况而定，逐行关闭MISRA规则（也就是说，遵守规则被认为为深度嵌入式系统创建的代码不正确，而不是认真遵守的规则）。每次出现此类情况时，都使用特殊的pc-lint MISRA注释标记语法来说明理由。

FreeRTOS使用许多不同的编译器构建，其中一些比其他的更高级。因此，FreeRTOS不会使用C99标准或C99标准之后引入C语言的任何功能或语法。一个例外是使用stdint.h头文件。该FreeRTOS的/来源/包括目录包含一个名为stdint.readme可以改名stdint.h提供必要建立FreeRTOS的最低stdint类型定义-要你的编译器不提供其自身。

**测验**

本节描述了对通用代码（所有FreeRTOS内核端口构建的[位于FreeRTOS / Source目录中](https://www.freertos.org/a00017.html)的代码）执行的测试，以及对可移植层代码（位于FreeRTOS / Source子目录中的代码）执行的测试。/ portable目录）。

* **通用代码**

标准的演示/测试文件试图提供“分支”测试覆盖率（在大多数情况下，实际上达到了[“条件”覆盖率，](https://en.wikipedia.org/wiki/Code_coverage)因为内核的编码风格专门为此目的而使条件保持简单），从而测试确保了“真”和“假”行使每个决定的路径。如果“ else”路径为空，则通过在每个“ if（）”条件的“ else”路径中将mtCOVERAGE\_TEST\_MARKER（）宏定义为NOP（无操作）指令来使用GCOV测量“分支”覆盖率。mtCOVERAGE\_TEST\_MARKER（）仅在测量测试覆盖率时定义–通常它是一个空宏，不会生成任何代码。

* **端口层**

端口层代码使用“ reg test”任务进行测试，对于支持中断嵌套的端口，使用“ interrupt queue”任务进行测试。

“ reg test”任务创建多个（通常是两个）任务，这些任务首先用已知值填充所有CPU寄存器，然后随着其他测试连续执行（浸泡测试），不断检查每个寄存器是否保持其预期的已知值。每个reg测试任务都使用唯一的值。

“中断队列”任务对嵌套至少三个深度的不同优先级的中断执行测试。宏用于将人为的延迟插入代码中的相关点，以确保实现所需的测试覆盖率。

**值得注意的是，这些测试的彻底性导致多次发现硅中的错误。**

**命名约定**

RTOS内核和演示应用程序源代码使用以下约定：

* 变数
  + *uint32\_t*类型的变量以*ul开头*，其中“ u”表示“无符号”，而“ l”表示“ long”。
  + *uint16\_t*类型的变量以*us*为前缀，其中“ u”表示“无符号”，而“ s”表示“短”。
  + *uint8\_t*类型的变量的前缀为*uc*，其中“ u”表示“无符号”，而“ c”表示“ char”。
  + 非stdint类型的变量以*x*为前缀。示例包括BaseType\_t和TickType\_t，它们是体系结构的自然或最有效类型的便携式层定义的typedef，以及分别用于保存RTOS滴答计数的类型。
  + 非stdint类型的无符号变量有一个附加前缀*u*。例如，类型为UBaseType\_t（无符号BaseType\_t）的变量以*ux*为前缀。
  + *size\_t*类型的变量也带有*x*前缀。
  + 枚举变量以*e*开头
  + 指针有一个附加的前缀*p*，例如，指向uint16\_t的指针将具有前缀*pus*。
  + 与MISRA指南一致，不合格的标准*char*类型仅允许保留ASCII字符，并且前缀为*c*。
  + 与MISRA指南一致，*char \**类型的变量仅允许保存指向ASCII字符串的指针，并以*pc*为前缀。
* 职能
  + 文件范围静态（专用）功能以*prv*为前缀。
  + 根据为变量定义的约定，API函数以其返回类型为前缀，并为*void*添加前缀*v*。
  + API函数名称以其定义文件的名称开头。例如，v Task Delete在task.c中定义，并且具有空返回类型。
* 巨集
  + 宏以定义它们的文件为前缀。前缀为小写。例如，在FreeRTOSConfig.h中定义了USE\_PREEMPTION配置。
  + 除前缀外，宏均以大写形式编写，并使用下划线分隔单词。

**资料类型**

仅使用stdint.h类型和RTOS自己的typedef，但以下情况除外：

* 烧焦

与MISRA指南一致，允许使用非限定字符类型，但仅当它们用于保存ASCII字符时才允许。

* 字符\*

与MISRA指南一致，允许使用不合格的字符指针，但仅当它们用于指向ASCII字符串时才允许使用。当使用期望使用char \*参数的标准库函数时，这消除了消除良性编译器警告的需要，尤其是考虑到某些编译器将默认的不合格char类型签名，而其他编译器将默认的不合格char类型签名。

每个端口定义了四种类型。这些是：

* TickType\_t

如果configUSE\_16\_BIT\_TICKS设置为非零（true），则TickType\_t被定义为无符号16位类型。如果configUSE\_16\_BIT\_TICKS设置为零（false），则TickType\_t被定义为无符号32位类型。有关完整信息，请参见API文档的[自定义](https://www.freertos.org/a00110.html)部分。

32位体系结构应始终将configUSE\_16\_BIT\_TICKS设置为0。

* BaseType\_t

这被定义为架构最有效，最自然的类型。例如，在32位体系结构上，BaseType\_t将被定义为32位类型。在16位体系结构上，BaseType\_t将被定义为16位类型。如果将BaseType\_t定义为char，则必须格外小心，以确保将带符号的char用于函数返回值，该值可以为负，以指示错误。

* UBaseType\_t

无符号的BaseType\_t。

* StackType\_t

定义为体系结构用于堆栈中存储的项目的类型。在正常情况下，在16位体系结构上为16位类型，在32位体系结构上为32位类型，尽管有一些例外。由FreeRTOS内部使用。

**编码风格**

* 缩进

制表符用于缩进。一个制表符等于四个空格。

* 注释

注释永远不会超过第80列，除非它们遵循并描述了一个参数。

不使用C ++样式双斜杠（//）注释。

* 布局

FreeRTOS源代码布局旨在尽可能易于查看和阅读。下面的代码段首先显示文件布局，然后显示C代码格式。

**/\* Library includes come first… \*/**

**#include <stdlib.h>**

**/\* …followed by FreeRTOS includes… \*/**

**#include “FreeRTOS.h”**

**/\* …followed by other includes. \*/**

**#include “HardwareSpecifics.h”**

**/\* #defines comes next, bracketed where possible. \*/**

**#define A\_DEFINITION ( 1 )**

**/\***

**\* Static (file private) function prototypes appear next, with comments**

**\* in this style – with each line starting with a ‘\*’.**

**\*/**

**static void prvAFunction( uint32\_t ulParameter );**

**/\* File scope variables are the last thing before the function definitions.**

**Comments for variables are in this style (without each line starting with**

**a ‘\*’). \*/**

**static BaseType\_t xMyVariable;**

**/\* The following separate is used after the closing bracket of each function,**

**with a blank line following before the start of the next function definition. \*/**

**/\*———————————————————–\*/**

**void vAFunction( void )**

**{**

**/\* Function definition goes here – note the separator after the closing**

**curly bracket. \*/**

**}**

**/\*———————————————————–\*/**

**static UBaseType\_t prvNextFunction( void )**

**{**

**/\* Function definition goes here. \*/**

**}**

**/\*———————————————————–\*/**

**File Layout**

**/\* Function names are always written on a single line, including the return**

**type. As always, there is no space before the opening parenthesis. There**

**is a space after an opening parenthesis. There is a space before a closing**

**parenthesis. There is a space after each comma. Parameters are given**

**verbose, descriptive names (unlike this example!). The opening and closing**

**curly brackets appear on their own lines, lined up underneath each other. \*/**

**void vAnExampleFunction( long lParameter1, unsigned short usParameter2 )**

**{**

**/\* Variable declarations are not indented. \*/**

**uint8\_t ucByte;**

**/\* Code is indented. Curly brackets are always on their own lines**

**and lined up underneath each other. \*/**

**for( ucByte = 0U; ucByte < fileBUFFER\_LENGTH; ucByte++ )**

**{**

**/\* Indent again. \*/**

**}**

**}**

**/\* For, while, do and if constructs follow a similar pattern. There is no**

**space before the opening parenthesis. There is a space after an opening**

**parenthesis. There is a space before a closing parenthesis. There is a**

**space after each semicolon (if there are any). There are spaces before and**

**after each operator. No reliance is placed on operator precedence –**

**parenthesis are always used to make precedence explicit. Magic numbers,**

**other than zero, are always replaced with a constant or #defined constant.**

**The opening and closing curly brackets appear on their own lines. \*/**

**for( ucByte = 0U; ucByte < fileBUFFER\_LENGTH; ucByte++ )**

**{**

**}**

**while( ucByte < fileBUFFER\_LENGTH )**

**{**

**}**

**/\* There must be no reliance on operator precedence – every condition in a**

**multi-condition decision must uniquely be bracketed, as must all**

**sub-expressions. \*/**

**if( ( ucByte < fileBUFFER\_LENGTH ) && ( ucByte != 0U ) )**

**{**

**/\* Example of no reliance on operator precedence! \*/**

**ulResult = ( ( ulValue1 + ulValue2 ) – ulValue3 ) \* ulValue4;**

**}**

**/\* Conditional compilations are laid out and indented as per any**

**other code. \*/**

**#if( configUSE\_TRACE\_FACILITY == 1 )**

**{**

**/\* Add a counter into the TCB for tracing only. \*/**

**pxNewTCB->uxTCBNumber = uxTaskNumber;**

**}**

**#endif**

**A space is placed after an opening square bracket, and before a closing**

**square bracket.**

**ucBuffer[ 0 ] = 0U;**

**ucBuffer[ fileBUFFER\_LENGTH – 1U ] = 0U;**

**Formatting of C Constructs**

## 实施质量管理

|  |
| --- |
| **价值主张** |
| 严格配置管理下的高质量C源代码 |
| 安全关键版本可确保可靠性 |
| 跨平台支持可确保时间投入 |
| 教程和培训工程师的培训 |
| 所有受支持端口的预配置示例项目 |
| 免费支持，其报价优于某些商业替代产品 |
| 庞大且不断增长的用户群和社区 |
| *放心*–随时可以选择低成本商业选择 |
| **=较低的总拥有成本，无风险和引人注目的解决方案** |

|  |  |
| --- | --- |
| **RTOS技术亮点** | |
| 优先调度选项 | 易于使用的消息传递 |
| 合作调度选项 | 带时间切片的轮循 |
| 快速任务通知 | 具有优先级继承的互斥量 |
| 6K至12K ROM占用空间 | 递归互斥 |
| 可配置/可扩展 | 二进制和计数信号量 |
| 与芯片和编译器无关 | 高效的软件计时器 |
| 一些端口永远不会完全禁止中断 | 易于使用的API |

FreeRTOS受到非常严格的质量管理，不仅在[软件编码标准和外观方面](https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html)，而且在实施方面。例如：

* FreeRTOS**绝不**会从关键部分或中断内部执行不确定的操作，例如遍历链表。
* 我们特别为高效的[软件计时器](https://www.freertos.org/RTOS-software-timer.html)实现而感到自豪，因为**除非计时器真正需要维修，否则它不会占用任何CPU时间**。软件计时器不包含需要递减为零的变量。
* 同样，阻止（挂起）的任务列表也不需要费时的定期服务。
* [直接到任务的通知](https://www.freertos.org/RTOS-task-notifications.html)允许快速的任务发信号，几乎没有RAM开销，并且可以用于大多数任务间和中断到任务发信号的情况。
* 所述[FreeRTOS的队列使用模型](https://www.freertos.org/Embedded-RTOS-Queues.html)设法简化与灵活性结合起来（在一个很小的代码大小） -这通常是相互排斥的属性。
* FreeRTOS队列是基础原语，在其上构建了其他通信和同步原语。获得的代码重用大大减少了总体代码大小，进而**有助于测试并确保鲁棒性**。

此外，[TÜV南德意志集团](http://www.tuev-sued.com/)认证的SIL 3个[SAFERTOS实时内核](https://www.freertos.org/FreeRTOS-Plus/Safety_Critical_Certified/SafeRTOS.html)最初是从衍生FreeRTOS操作系统，并经历了最严格的分析和测试过程-它的结果反馈到FreeRTOS的代码库（当共性仍然存在）。

**“官方支持”和“贡献”的FreeRTOS代码  
[**[**RTOS端口**](https://www.freertos.org/a00090.html)**]**

每个受支持的体系结构**和**编译器组合都被视为一个单独的FreeRTOS端口。

一个[演示应用程序](https://www.freertos.org/a00090.html)是运行在一个特定的硬件平台上的特定端口的项目。

FreeRTOS端口的微控制器体系结构特定部分称为端口层。每个端口层都是“官方支持”或“贡献”的。本页说明了两者之间的区别。

**官方支持的港口**

官方支持的端口：

* 包含在主要的FreeRTOS .zip文件发行版中（针对FreeRTOS发行版6.0.4及更高版本，已将提供的端口移至 [FreeRTOS Interactive](http://interactive.freertos.org/)站点）。
* 包括至少一个FreeRTOS.org网站上记录的演示应用程序。

演示应用程序是一个针对特定开发板并使用特定工具链的预配置项目。它旨在允许即使是新用户也可以“开箱即用”地使用，并且包括有关如何使用FreeRTOS API的许多示例。一些官方端口包括多个此类演示项目，以向用户提供硬件和构建环境的选择。

* 具有已知来源，消除了对其知识产权所有权的怀疑。反过来，这使得商业许可和支持合同（可选）由OpenRTOS品牌的WITTENSTEIN高完整性系统提供。
* 已由Amazon Web Services Inc或完全值得信赖的合作伙伴进行书面和/或全面检查和测试。
* 通常，在发布FreeRTOS核心源代码的新版本或相关构建工具的新版本时，进行维护和更新。
* 通常可以在免费访问和监视的[支持论坛](https://forums.freertos.org/)上获得 [支持](https://forums.freertos.org/)。

**贡献端口**

贡献端口：

* 由FreeRTOS社区用户提供，而不是直接由Amazon Web Services提供。
* 可从[FreeRTOS Interactive](http://interactive.freertos.org/)网站免费下载。
* 只能在标准的开源FreeRTOS许可下提供。不提供商业许可证来提供贡献代码。
* 由贡献者自己记录。因此，所提供文件的数量和质量在所提供的软件包之间有所不同。
* 通常，Amazon Web Services无法直接支持。

**FreeRTOS内核开发人员文档**

* [**任务**](https://www.freertos.org/taskandcr.html)
* [**队列，互斥量，信号量…**](https://www.freertos.org/Inter-Task-Communication.html)
* [**直接到任务通知**](https://www.freertos.org/RTOS-task-notifications.html)
* [**流和消息缓冲区**](https://www.freertos.org/RTOS-stream-message-buffers.html)
* [**软件计时器**](https://www.freertos.org/RTOS-software-timer.html)
* [**活动组（或“标志”）**](https://www.freertos.org/FreeRTOS-Event-Groups.html)
* [**源代码组织**](https://www.freertos.org/a00017.html)
* [**FreeRTOSConfig.h**](https://www.freertos.org/a00110.html)
* [**静态与动态内存**](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html)
* [**堆内存管理**](https://www.freertos.org/a00111.html)
* [**堆栈溢出保护**](https://www.freertos.org/Stacks-and-stack-overflow-checking.html)
* [**创建一个新项目**](https://www.freertos.org/Creating-a-new-FreeRTOS-project.html)
* [**联合例程**](https://www.freertos.org/kernel/coroutines.html)（已弃用）
* **任务和协程  
  [**[**入门**](https://www.freertos.org/static_menu.html#getting_started)**]**
* 有关基本的多任务处理概念的介绍，请参见[FreeRTOS的工作](https://www.freertos.org/implementation/index.html)原理部分。
* 该[任务](https://www.freertos.org/a00015.html)和[协程](https://www.freertos.org/croutine.html)文档页面提供信息，以判断何时协程可能适合和可能不适合使用。下面是一个简短的摘要。请注意，可以仅使用任务，仅使用协程或两者的组合来设计应用程序–但是，任务和协程使用不同的API函数，因此不能使用队列（或信号量）将数据从任务传递到例行程序，反之亦然。
* 协程实际上仅用于具有严格RAM约束的超小型处理器。
* **“任务”的特征**
* **简介**：使用RTOS的实时应用程序可以构造为一组独立的任务。每个任务都在其自己的上下文中执行，而不会偶然依赖系统中的其他任务或RTOS调度程序本身。在任何时间点，应用程序中只有一个任务可以执行，实时RTOS调度程序负责确定这应该是哪个任务。因此，RTOS调度程序可以在应用程序执行时重复启动和停止每个任务（将每个任务交换进出）。由于任务不了解RTOS调度程序的活动，因此实时RTOS调度程序负责确保交换任务时的处理器上下文（寄存器值，堆栈内容等）与同一任务时的情况完全相同被换了。为了实现这一目标，每个任务都提供了自己的堆栈。有关更多信息，请[参见FreeRTOS的工作原理](https://www.freertos.org/implementation/index.html)部分。

### “协程”的特征

**注意：**协程实现在RAM非常小的设备上使用，但如今在实际应用中很少使用。因此，虽然没有计划从代码中删除协程，但也没有计划进一步开发它们。

协程在概念上与任务相似，但是具有以下基本差异（在[协程文档页面上有详细介绍](https://www.freertos.org/croutine.html)）：

1. **堆栈使用**

应用程序中的所有协程共享一个堆栈。与使用任务编写的类似应用程序相比，这大大减少了所需的RAM数量。

1. **调度和优先级**

RTOS的协程相对于其他协程使用优先协作调度，但是可以包含在使用抢占式任务调度的应用程序中。

1. **宏实施**

通过一组宏提供协程实现。

1. **使用限制**

减少RAM使用的代价是对如何构造协程的严格限制。

# 任务 [[有关任务的更多信息...](https://www.freertos.org/a00015.html) ]

该[FreeRTOS的教程书籍](https://www.freertos.org/Documentation/RTOS_book.html) 提供关于任务和他们的行为的其他详细信息。

### 任务状态

任务可以以下列状态之一存在：

* **运行**

实际执行任务时，它处于“正在运行”状态。当前正在占用处理器。如果运行RTOS的处理器仅具有一个内核，则在任何给定时间只能有一个任务处于“运行”状态。

* **就绪**

就绪任务是那些能够执行的任务（它们不处于“阻塞”或“挂起”状态），但是由于当前具有不同优先级或更高优先级的其他任务已经处于“运行”态而无法执行。

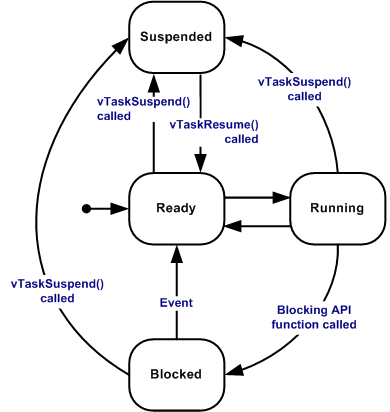
* **阻塞**

如果任务当前正在等待临时事件或外部事件，则称该任务处于“阻塞”状态。例如，如果任务调用vTaskDelay（），它将阻塞（置于“阻塞”状态），直到延迟时间段到期为止–这是一个时间事件。任务也可以阻塞以等待队列，信号量，事件组，通知或信号量事件。处于“阻塞”状态的任务通常有一个“超时”时间段，在此之后，该任务将超时并被解除阻塞，即使未发生正在等待任务的事件。

处于“阻塞”状态的任务不使用任何处理器时间，并且不能被选择进入“运行”状态。

* **挂起**

与处于“阻塞”状态的任务一样，处于“挂起”状态的任务无法选择进入“运行”状态，但是处于“挂起”状态的任务没有超时。相反，仅在分别通过vTaskSuspend（）和xTaskResume（）API调用明确命令这样做时，任务才进入或退出Suspended状态。



**有效的任务状态转换**

### 任务优先级

每个任务的优先级从0到（configMAX\_PRIORITIES – 1），其中configMAX\_PRIORITIES在FreeRTOSConfig.h中定义。

如果使用中的移植接口实现了接口优化的任务选择机制，该机制使用了``计数前导零''类型的指令（用于单个指令中的任务选择）并且FreeRTOSConfig.h中的configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION设置为1，那么configMAX\_PRIORITIES不能大于32。在所有其他情况下，configMAX\_PRIORITIES可以在合理范围内取任何值-但由于RAM的原因，应将效率保持在实际必要的最小值。

低优先级数字表示低优先级任务。该[空闲任务](https://www.freertos.org/RTOS-idle-task.html)的优先级为零（tskIDLE\_PRIORITY）。

FreeRTOS调度程序可确保始终向“就绪”或“运行”[状态的](https://www.freertos.org/RTOS-task-states.html)任务分配处理器（CPU）时间，而不是优先级较低的任务也处于“就绪”状态的任务。换句话说，处于“运行”状态的任务始终是能够运行的最高优先级任务。

任意数量的任务可以共享相同的优先级。如果未定义configUSE\_TIME\_SLICING，或者configUSE\_TIME\_SLICING设置为1，则具有相同优先级的就绪状态任务将使用时间片轮循调度方案共享可用的处理时间。

### 创建任务

任务应具有以下结构：

**void vATaskFunction( void \*pvParameters )**

**{**

**for( ;; )**

**{**

**-- Task application code here. --**

**}**

**/\* Tasks must not attempt to return from their implementing**

**function or otherwise exit. In newer FreeRTOS port**

**attempting to do so will result in an** [**configASSERT()**](https://www.freertos.org/a00110.html#configASSERT) **being**

**called if it is defined. If it is necessary for a task to**

**exit then have the task call vTaskDelete( NULL ) to ensure**

**its exit is clean. \*/**

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

**}**

TaskFunction\_t类型定义为一个返回void并以void指针为其唯一参数的函数。实现任务的所有功能都应属于此类。该参数可用于将任何类型的信息传递到任务中-一些[标准的演示应用程序任务](https://www.freertos.org/a00102.html)对此进行了 [演示](https://www.freertos.org/a00102.html)。

任务函数永远都不应返回，因此通常实现为连续循环。同样，请参见RTOS演示应用程序以获取大量示例。

通过调用[xTaskCreate](https://www.freertos.org/a00125.html)（）或[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)创建任务，并通过调用[vTaskDelete](https://www.freertos.org/a00126.html)（）删除任务。

[[返回顶部](https://www.freertos.org/implementing-a-FreeRTOS-task.html#page_top)]

### 任务创建宏

可以*选择*使用portTASK\_FUNCTION和portTASK\_FUNCTION\_PROTO宏定义任务功能。提供这些宏是为了允许将编译器特定的语法分别添加到函数定义和原型中。除非在文档中特别说明所使用的端口（当前仅是PIC18 fedC端口），否则不需要使用它们。

上面显示的函数的原型可以写成：

**void vATaskFunction（void \* pvParameters）;**

或是

**portTASK\_FUNCTION\_PROTO（vATaskFunction，pvParameters）;**

同样，上面的函数可以等效地写为：

**portTASK\_FUNCTION( vATaskFunction, pvParameters )**

**{**

**for( ;; )**

**{**

**-- Task application code here. --**

**}**

**}**

# 协程 [[有关协程的更多信息](https://www.freertos.org/croutine.html)]

### 协程的状态

协程仅适用于具有严格RAM限制的超小型处理器，通常不会在32位微控制器上使用。

协程可以以下列状态之一存在：

* **运行**

当协程实际上正在执行时，它被称为处于运行状态。当前正在使用处理器。

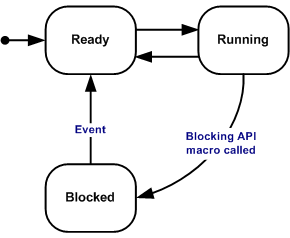
* **就绪**

就绪的协程是那些能够执行（它们未被阻塞）但当前未执行的例程。协程可能处于“就绪”状态，因为：

* 1. 具有相同或更高优先级的另一个协程已处于“运行”状态，或者
  2. 任务处于“运行”状态–仅在应用程序同时使用任务和协程的情况下才如此。
* **阻塞**

如果协程当前正在等待时间或外部事件，则该协程被称为处于Blocked状态。例如，如果协程调用crDELAY（），它将阻塞（置于“阻塞”状态），直到延迟时间到期为止–这是一个时间事件。阻塞的协程不可用于调度。

当前没有与“任务挂起”状态等效的协程状态。



**有效的协同程序状态转换**

### 创建协程

协程应具有以下结构：

**void vACoRoutineFunction( CoRoutineHandle\_t xHandle,**

**UBaseType\_t uxIndex )**

**{**

**crSTART( xHandle );**

**for( ;; )**

**{**

**-- Co-routine application code here. --**

**}**

**crEND();**

**}**

crCOROUTINE\_CODE类型定义为一个返回void并以CoRoutineHandle\_t和一个索引作为其参数的函数。实现协程的所有函数原型都应如此（上面已演示）。

通过调用xCoRoutineCreate（）创建协程。

注意事项：

* 所有协程函数**必须**以对crSTART（）的调用开始。
* 所有协程函数**必须**以对crEND（）的调用结尾。
* 协例程函数永远不应该返回，因此通常实现为连续循环。
* 可以从一个共同例程函数创建许多共同例程。提供uxIndex参数作为区分此类协程的一种方式。

### 协程优先级

每个协程的优先级分配为0到（configMAX\_CO\_ROUTINE\_PRIORITIES – 1）。configMAX\_CO\_ROUTINE\_PRIORITIES是在FreeRTOSConfig.h中定义的，可以在每个应用程序的基础上进行设置。

低优先级数字表示低优先级协程。

协程的优先级仅与其他协程有关。如果您在同一应用程序中混合使用任务和协程，则任务将始终优先于协程。

### 调度协程

通过重复调用[vCoRoutineSchedule（）](https://www.freertos.org/crschedule.html)来调度协程。调用vCoRoutineSchedule（）的最佳位置是从 [空闲任务hook中](https://www.freertos.org/RTOS-idle-task.html)。即使您的应用程序仅使用协程，也是如此，因为在启动调度程序时仍将自动创建空闲任务。 [请参阅后面的示例](https://www.freertos.org/scheduling-coroutines-with-tasks.html)。

### 混合使用任务和协程

从空闲任务中调度协程可以使任务和协程轻松混合在同一应用程序中。完成此操作后，协程将仅在没有比执行的空闲任务高的优先级的任务时执行。 [请参阅后面的示例](https://www.freertos.org/scheduling-coroutines-with-tasks.html)。

### 局限性

与同等任务相比，协程具有较低的RAM使用率的好处是以牺牲一些对协程的使用方式为代价的。协程比任务更具限制性和复杂性。

* **共享堆栈**

当协程阻塞时，不维护该协程的堆栈。这意味着在堆栈上分配的变量很可能会丢失其值。为了克服这个问题，必须将必须在阻塞调用中保持其值的变量声明为静态变量。例如：

**void vACoRoutineFunction( CoRoutineHandle\_t xHandle,**

**UBaseType\_t uxIndex )**

**{**

**static char c = 'a';**

**// Co-routines must start with a call to crSTART().**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// If we set c to equal 'b' here ...**

**c = 'b';**

**// ... then make a blocking call ...**

**crDELAY( xHandle, 10 );**

**// ... c will only be guaranteed to still**

**// equal 'b' here if it is declared static**

**// (as it is here).**

**}**

**// Co-routines must end with a call to crEND().**

**crEND();**

**}**

共享堆栈的另一个结果是，可能导致协程阻塞的API函数的调用只能从该协程函数本身进行，*而不*能从该协程调用的函数内部进行。例如：

**void vACoRoutineFunction( CoRoutineHandle\_t xHandle, UBaseType\_t uxIndex )**

**{**

**// Co-routines must start with a call to crSTART().**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// It is fine to make a blocking call here,**

**crDELAY( xHandle, 10 );**

**// but a blocking call cannot be made from within**

**// vACalledFunction().**

**vACalledFunction();**

**}**

**// Co-routines must end with a call to crEND().**

**crEND();**

**}**

**void vACalledFunction( void )**

**{**

**// Cannot make a blocking call here!**

**}**

* **使用switch语句**

FreeRTOS下载中包含的默认协程实现不允许在switch语句中进行阻塞调用。例如：

* **void vACoRoutineFunction( CoRoutineHandle\_t xHandle, UBaseType\_t uxIndex )**
* **{**
* **// Co-routines must start with a call to crSTART().**
* **crSTART( xHandle );**
* **for( ;; )**
* **{**
* **// It is fine to make a blocking call here,**
* **crDELAY( xHandle, 10 );**
* **switch( aVariable )**
* **{**
* **case 1 : // Cannot make a blocking call here!**
* **break;**
* **default: // Or here!**
* **}**
* **}**
* **// Co-routines must end with a call to crEND().**
* **crEND();**
* **}**

### 快速例程示例

这个简单的示例演示了协程的用法。

1. **创建一个简单的例程来使LED闪烁**

以下代码定义了一个非常简单的协程，该例程不执行任何操作，但会定期使LED闪烁。

**void vFlashCoRoutine( CoRoutineHandle\_t xHandle,**

**UBaseType\_t uxIndex )**

**{**

**// Co-routines must start with a call to crSTART().**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// Delay for a fixed period.**

**crDELAY( xHandle, 10 );**

**// Flash an LED.**

**vParTestToggleLED( 0 );**

**}**

**// Co-routines must end with a call to crEND().**

**crEND();**

**}**

而已！

1. **创建协程**

通过重复调用vCoRoutineSchedule（）来调度协程。最好的方法是在空闲任务中编写一个空闲任务挂钩。首先确保在FreeRTOSConfig.h中将configUSE\_IDLE\_HOOK设置为1 。然后将空闲任务挂钩写为：

**void vApplicationIdleHook( void )**

**{**

**vCoRoutineSchedule( void );**

**}**

或者，如果空闲任务未执行任何其他功能，则从循环内调用vCoRoutineSchedule（）会更加高效：

**void vApplicationIdleHook( void )**

**{**

**for( ;; )**

**{**

**vCoRoutineSchedule( void );**

**}**

**}**

1. **创建协程并启动RTOS调度程序**

可以在main（）中创建协程。

**#include "task.h"**

**#include "croutine.h"**

**#define PRIORITY\_0 0**

**void main( void )**

**{**

**// In this case the index is not used and is passed**

**// in as 0.**

**xCoRoutineCreate( vFlashCoRoutine, PRIORITY\_0, 0 );**

**// NOTE: Tasks can also be created here!**

**// Start the RTOS scheduler.**

**vTaskStartScheduler();**

**}**

1. **扩展示例：使用index参数**

现在假设我们要从同一函数创建8个这样的协例程。每个协程将以不同的速率闪烁不同的LED。index参数可用于从协程函数本身内部区分协程。

这次，我们将创建8个协程，并向每个例程传递不同的索引。

**#include "task.h"**

**#include "croutine.h"**

**#define PRIORITY\_0 0**

**#define NUM\_COROUTINES 8**

**void main( void )**

**{**

**int i;**

**for( i = 0; i < NUM\_COROUTINES; i++ )**

**{**

**// This time i is passed in as the index.**

**xCoRoutineCreate( vFlashCoRoutine, PRIORITY\_0, i );**

**}**

**// NOTE: Tasks can also be created here!**

**// Start the RTOS scheduler.**

**vTaskStartScheduler();**

**}**

协程功能也得到了扩展，因此每个例程都使用不同的LED和闪烁速率。

**const int iFlashRates[ NUM\_COROUTINES ] = { 10, 20, 30, 40, 50, 60, 70, 80 };**

**const int iLEDToFlash[ NUM\_COROUTINES ] = { 0, 1, 2, 3, 4, 5, 6, 7 }**

**void vFlashCoRoutine( CoRoutineHandle\_t xHandle, UBaseType\_t uxIndex )**

**{**

**// Co-routines must start with a call to crSTART().**

**crSTART( xHandle );**

**for( ;; )**

**{**

**// Delay for a fixed period. uxIndex is used to index into**

**// the iFlashRates. As each co-routine was created with**

**// a different index value each will delay for a different**

**// period.**

**crDELAY( xHandle, iFlashRate[ uxIndex ] );**

**// Flash an LED. Again uxIndex is used as an array index,**

**// this time to locate the LED that should be toggled.**

**vParTestToggleLED( iLEDToFlash[ uxIndex ] );**

**}**

**// Co-routines must end with a call to crEND().**

**crEND();**

**}**

### FreeRTOS演示应用示例

下载中包含两个文件，这些文件演示了如何将协程与队列一起使用：

1. [**crflash.c**](https://www.freertos.org/a00013.html#crflash)

这在功能上等同于[标准演示](https://www.freertos.org/a00102.html)文件flash.c， 但使用协程而不是任务。另外，仅出于演示目的，而不是直接从协程内部切换LED（按照上面的快速示例），应切换的LED数量在队列中传递给更高优先级的协程。

1. [**钩子**](https://www.freertos.org/a00013.html#crhook)

演示将数据从中断传递到协程。勾钩函数用作数据源。

在[PC](https://www.freertos.org/a00100.html)和老的一个[ARM Cortex-M3的](https://www.freertos.org/portcortexkeil.html) 演示程序已预先配置为使用这些样本协程文件，可以作为参考。所有其他演示应用程序都配置为仅使用任务，但可以按照以下步骤轻松转换为演示协程。这将用crflash.c实现的功能替换了flash.c内部实现的功能：

1. 在FreeRTOSConfig.h中，将configUSE\_CO\_ROUTINES和configUSE\_IDLE\_HOOK设置为1。
2. 在IDE项目或项目makefile中（取决于所使用的演示项目）：
   1. 替换引用文件FreeRTOS操作系统/演示/通用/最小/ flash.c与FreeRTOS操作系统/演示/通用/最小/ crflash.c。
   2. 将文件FreeRTOS / Source / croutine.c添加到构建中。
3. 在main.c中：
   1. 包括头文件croutine.h，其中包含协程序宏和函数原型。
   2. 更换列入flash.h与crflash.h。
   3. 删除对创建闪存任务vStartLEDFlashTasks（）...的函数的调用。
   4. …并将其替换为创建闪存协程vStartFlashCoRoutines（n）的函数，其中n是应创建的协程的数量。每个协程以不同的速率闪烁不同的LED。
   5. 添加一个空闲挂钩函数，该函数将协程安排为：

**void vApplicationIdleHook( void )**

**{**

**vCoRoutineSchedule( void );**

**}**

如果main（）已经包含一个空闲钩子，则只需将对vCoRoutineSchedule（）的调用添加到现有钩子函数中。

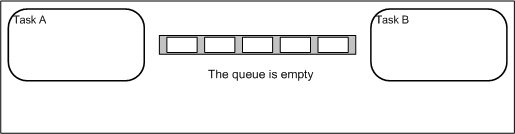
1. 用Flash例程代替Flash任务意味着至少要少分配两个堆栈，因此可以留出较少的堆空间供RTOS调度程序使用。如果您的项目没有足够的RAM来在构建中包含croutine.c，则只需在FreeRTOSConfig.h中将portTOTAL\_HEAP\_SPACE的定义减少（2 \* portMINIMAL\_STACK\_SIZE）。

# FreeRTOS队列 [[任务间通信和同步](https://www.freertos.org/Inter-Task-Communication.html)]

### FreeRTOS队列

[另请参见[阻塞等待多个RTOS对象](https://www.freertos.org/Pend-on-multiple-rtos-objects.html)]

队列是任务间通信的主要形式。它们可用于在任务之间以及中断和任务之间发送消息。在大多数情况下，它们也用作线程安全的FIFO（先进先出）缓冲区，新数据将发送到队列的后面，尽管数据也可以发送到前面。

  
写入队列和从队列读取。在此示例中，创建队列以容纳5个项目，并且队列永远不会变满。

### 用户模型：最简单，最大灵活性。。。

FreeRTOS队列使用模型设法将简单性与灵活性结合在一起-这两者通常是互斥的。消息通过复制通过队列发送，这意味着数据（可以是指向较大缓冲区的指针）本身被复制到队列中，而不是队列中始终仅存储对数据的引用。这是最好的方法，因为：

* 可以将已包含在C变量中的小消息（整数，小结构等）直接发送到队列中。无需为消息分配缓冲区，然后将变量复制到已分配的缓冲区中。同样，可以将消息从队列直接读取到C变量中。

此外，以这种方式发送到队列允许发送任务立即覆盖已发送到队列的变量或缓冲区，即使已发送的消息保留在队列中也是如此。由于变量中包含的数据已复制到队列中，因此变量本身可以重复使用。不需要发送消息的任务和接收消息的任务来同意哪个任务拥有消息，以及哪个任务负责在不再需要消息时释放消息。

* 使用通过复制传递数据的队列不会阻止队列用于通过引用传递数据。当复制到队列中的消息大小超出队列容纳的大小时，请在队列中使用指针，仅将指向消息的指针复制到队列中。这正是 [FreeRTOS + UDP](https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_UDP/FreeRTOS_Plus_UDP.html) 实现在FreeRTOS IP堆栈周围传递大型网络缓冲区的方式。
* 内核完全负责分配用作队列存储区的内存。
* 大小可变的消息可以通过定义队列来发送，以容纳包含指向队列消息的成员和另一个保留队列消息的大小的成员的结构。
* 通过定义队列来保存一个结构，该结构具有一个保存消息类型的成员和另一个保存消息数据的成员（或指向该消息的指针），单个队列可用于接收不同的消息类型以及来自多个位置的消息。消息数据如何解释数据取决于消息类型。这正是管理[FreeRTOS + UDP](https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_UDP/FreeRTOS_Plus_UDP.html) IP堆栈的任务 能够使用单个队列接收ARP计时器事件，从以太网硬件接收到的数据包，从应用程序接收到的数据包，网络中断事件等的通知的方式。
* 该实现自然适合在受存储器保护的环境中使用。受限于受保护的内存区域的任务可以将数据传递至受限于不同的受保护的内存区域的任务，因为通过调用队列发送功能调用RTOS将提高微控制器特权级别。队列存储区域只能由RTOS访问（具有完全特权）。
* 提供了一个单独的API，供在中断内部使用。将用于RTOS任务的API与用于中断服务程序的API分开意味着，RTOS API函数的实现不会在每次执行时都带来检查其调用上下文的开销。在大多数情况下，与其他RTOS产品相比，使用单独的中断API意味着对于最终用户而言，创建RTOS感知中断服务例程更为简单。
* 在任何方面，API都更加简单。

该[FreeRTOS的教程书](https://www.freertos.org/Documentation/RTOS_book.html) 提供了队列，二值信号量，互斥，计数信号量和递归信号量的其他信息，以及简单的一组随附示例项目工作的例子。

### 阻塞队列

队列API函数允许指定阻塞时间。

当任务尝试从空队列中读取数据时，该任务将被置于“阻塞”状态（因此不会消耗任何CPU时间，其他任务可以运行），直到队列上有可用数据或阻塞时间到为止。

当任务尝试写入已满队列时，该任务将被置于“阻塞”状态（因此不会消耗任何CPU时间，其他任务可以运行），直到队列中有可用空间或阻塞时间到为止。

如果同一队列上有多个任务阻塞，则优先级最高的任务将是首先取消阻塞的任务。

有关与队列相关的API函数的列表，请参见用户文档的“[队列管理”](https://www.freertos.org/a00018.html)部分。在FreeRTOS / Demo / Common / Minimal目录中搜索文件将显示其用法的多个示例。

请注意，中断中必须使用以 “ FromISR”结尾的API函数。

# FreeRTOS二值信号量 [[任务间通信和同步](https://www.freertos.org/Inter-Task-Communication.html)]

[另请参见[阻止多个RTOS对象](https://www.freertos.org/Pend-on-multiple-rtos-objects.html)]

该[FreeRTOS的教程书](https://www.freertos.org/Documentation/RTOS_book.html) 提供了队列，二值信号量，互斥，计数信号量和递归信号量的其他信息，以及简单的一组随附示例项目工作的例子。 

### FreeRTOS二值信号量

|  |
| --- |
| [**提示：“任务通知”可以在许多情况下提供轻量级的二值信号量替代方案**](https://www.freertos.org/RTOS_Task_Notification_As_Binary_Semaphore.html) |

二值信号量用于互斥和同步目的。

二值信号量和互斥量非常相似，但有一些细微的区别：互斥量包括优先级继承机制，二值信号量则没有。这使得二值信号量是实现同步（任务之间或任务之间和中断之间）的更好选择，而互斥量则是实现简单互斥的更好选择。该[描述](https://www.freertos.org/Real-time-embedded-RTOS-mutexes.html)的互斥量是如何被用作一个互斥机制的二进制信号也同样成立。本小节仅描述使用二值信号量进行同步。

信号量API函数允许指定阻塞时间。阻塞时间表示在尝试“获取”信号量时，如果信号量不立即可用，则任务应进入“阻塞”状态的最大系统“心跳”数。如果多个任务因等待同一个信号量被阻塞，则优先级最高的任务将是下次信号量可用时被最先解除阻塞的任务。

将二值信号量视为只能容纳一项的队列。因此，队列只能为空或已满（因此为二值）。使用队列的任务和中断并不关心队列的内容，它们只想知道队列是空还是已满。（例如）可以利用该机制来使任务与中断同步。

考虑使用任务为外围设备服务的情况。轮询外围设备会浪费CPU资源，并阻止其他任务执行。因此，最好将任务的大部分时间都花在“阻塞”状态（允许其他任务执行），并且仅在实际上有事要做时才执行。这是通过使用二值信号量来实现的，即在尝试“获取”信号量时让任务为Block。然后为外设编写一个中断例程，当外设请求服务时，该例程仅“发出”信号量。任务始终“获取”信号量（从队列中读取以使队列为空），但从不“给予”信号量。中断总是“给予”信号量（写入队列以使其满载），但从不获取它。提供的源代码[xSemaphoreGiveFromISR（）](https://www.freertos.org/a00124.html)文档页面应使此内容更清晰。另请参阅[RTOS任务通知](https://www.freertos.org/RTOS-task-notifications.html)，在某些情况下可以将其用作更快，更轻量的二值信号量替代方法。

任务优先级可用于确保外围设备及时获得服务-有效地生成“延迟中断”方案。（请注意，FreeRTOS还具有[内置的延迟中断机制](https://www.freertos.org/xTimerPendFunctionCallFromISR.html)）。另一种方法是使用队列代替信号量。完成此操作后，中断例程可以捕获与外围事件关联的数据，并将其发送到任务队列中。当数据在队列中可用时，任务将解除阻塞，从队列中检索数据，然后执行所需的任何数据处理。第二种方案允许中断保持尽可能短的时间，而所有后期处理都在任务内进行。

有关与[信号量](https://www.freertos.org/a00113.html)相关的API函数的列表，请参见用户文档的“[信号量/ Mutexes”](https://www.freertos.org/a00113.html)部分。在FreeRTOS / Demo / Common / Minimal目录中搜索文件将显示其用法的多个示例。请注意，中断不得使用不以“ FromISR”结尾的API函数。

  
使用信号量将任务与中断同步。中断只会“发出”信号量，而任务只会“接收”信号量。

# FreeRTOS计数信号量 [[任务间通信和同步](https://www.freertos.org/Inter-Task-Communication.html)]

[另请参见[阻止多个RTOS对象](https://www.freertos.org/Pend-on-multiple-rtos-objects.html)]

该[FreeRTOS的教程书](https://www.freertos.org/Documentation/RTOS_book.html) 提供了队列，二值信号量，互斥，计数信号量和递归信号量的其他信息，以及简单的一组随附示例项目工作的例子。 

### FreeRTOS计数信号量

|  |
| --- |
| [**提示：在许多情况下，“任务通知”可以为代替计数信号量提供轻巧的选择**](https://www.freertos.org/RTOS_Task_Notification_As_Counting_Semaphore.html) |

正如可以将二值信号量视为长度为1的队列一样，将计数信号量视为长度大于1的队列。同样，信号量的用户对队列中存储的数据不感兴趣–只是队列是否为空。

计数信号量通常用于两件事：

1. 计数事件。

在这种使用情况下，事件处理程序将在每次事件发生时“给予”信号量（增加信号量计数值），而处理程序任务将在每次处理事件时“获取”信号量（减小信号量计数值）。因此，计数值是已发生的事件数与已处理的事件数之间的差。在这种情况下，希望在创建信号量时将计数值设为零。

1. 资源管理。

在这种使用情况下，计数值指示可用资源的数量。要获得对资源的控制，任务必须首先获得一个信号量-减少信号量计数值。当计数值达到零时，将没有可用资源。当任务使用资源完成时，它将“给予”信号量–增加信号量计数值。在这种情况下，希望在创建信号量时使计数值等于最大计数值。

有关与[信号量](https://www.freertos.org/a00113.html)相关的API函数的列表，请参见用户文档的“[信号量/ Mutexes”](https://www.freertos.org/a00113.html)部分。在FreeRTOS / Demo / Common / Minimal目录中搜索文件将显示其用法的多个示例。请注意，中断中必须使用以“ FromISR”结尾的API函数。

# FreeRTOS互斥量 [[任务间通信和同步](https://www.freertos.org/Inter-Task-Communication.html)]

[另请参见[阻止多个RTOS对象](https://www.freertos.org/Pend-on-multiple-rtos-objects.html)]

该[FreeRTOS的教程书](https://www.freertos.org/Documentation/RTOS_book.html) 提供了队列，二值信号量，互斥量，计数信号量和递归互斥信号量的其他信息，以及简单的一组随附示例项目工作的例子。 

### FreeRTOS互斥量

互斥量是包括优先级继承机制的[二值信号量](https://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html)。二值信号量是实现同步（任务之间或任务与中断之间）的更好选择，而互斥锁是实现简单互斥（因此称为“ MUT”和“ EX”）的更好选择。

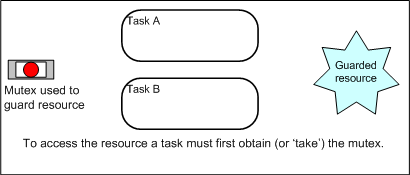
当互斥用于互斥时，互斥量的作用就像令牌一样，用于保护资源。当任务希望访问资源时，它必须首先获取（“获取”）令牌。完成资源使用后，它必须“返还”令牌-允许其他任务有机会访问相同的资源。

互斥量使用相同的信号量访问API函数，因此也允许指定阻塞时间。阻塞时间表示如果互斥量不立即可用，则在尝试“获取”互斥量时任务应进入“阻塞”状态的最大“心跳”数。但是，与二值信号量不同，互斥量具有优先级继承。这意味着，如果高优先级任务在尝试获取当前由低优先级任务持有的互斥锁（令牌）时阻塞，则持有令牌的任务的优先级会暂时提高到阻塞任务的优先级。该机制旨在确保将较高优先级的任务在尽可能短的时间内保持在阻塞状态，从而最大程度地减少已经发生的“优先级倒置”。

优先级继承不能根除优先级倒置！它只是在某些情况下将其影响最小化。硬实时应用程序的设计应确保首先不会发生优先级倒置。

不应从中断中使用互斥对象，因为：

* 它们包括优先级继承机制，该机制仅在指定互斥量并从任务中获取互斥量时才有意义，而不是中断。
* 中断不能阻塞以等待由互斥锁保护的资源变为可用。

  
使用互斥量来保护对共享资源的访问。

# FreeRTOS递归互斥量 [[任务间通信和同步](https://www.freertos.org/Inter-Task-Communication.html)]

[另请参见[阻止多个RTOS对象](https://www.freertos.org/Pend-on-multiple-rtos-objects.html)]

该[FreeRTOS的教程书](https://www.freertos.org/Documentation/RTOS_book.html) 提供了队列，二值信号量，互斥，计数信号量和递归信号量的其他信息，以及简单的一组随附示例项目工作的例子。

### FreeRTOS递归互斥量

所有者可以反复“递归”使用递归使用的互斥锁。在所有者为每个成功的xSemaphoreTakeRecursive（）请求调用xSemaphoreGiveRecursive（）之前，互斥量不再可用。例如，如果一个任务成功“获取”了相同的互斥量5次，则该互斥量将无法用于任何其他任务，直到它也将“互斥量”恰好“给予”了5次。

这种类型的信号量使用优先级继承机制，因此一旦不再需要该信号量的任务，就必须“取回”该信号量。

互斥类型信号量不能在中断服务程序中使用。

不应从中断中使用互斥对象，因为：

* 它们包括优先级继承机制，该机制仅在指定互斥量并从任务中获取互斥量时才有意义，而不是中断。
* 中断不能阻塞以等待由互斥锁保护的资源变为可用。

# RTOS任务通知 [[任务间通信和同步](https://www.freertos.org/Inter-Task-Communication.html)]

|  |
| --- |
| [**从FreeRTOS V8.2.0开始可用，   从V10.4.0开始，每个任务支持多个通知**](https://www.freertos.org/upgrading-to-FreeRTOS-V8.html) |

### 描述

[另请参阅博客文章[“使用FreeRTOS通知减少RAM占用空间并加快执行速度”](https://www.freertos.org/2020/09/decrease-ram-footprint-and-accelerate-execution-with-freertos-notifications.html) ]

每个RTOS任务都有一系列*任务通知*。每个任务通知都有一个可以为“待处理”或“不待处理”的*通知状态*，以及一个32位*通知值*。常量[configTASK\_NOTIFICATION\_ARRAY\_ENTRIES](https://www.freertos.org/a00110.html#configTASK_NOTIFICATION_ARRAY_ENTRIES)设置任务通知数组中的索引数。在FreeRTOS V10.4.0之前的版本中，任务只有一个任务通知，而没有一系列通知。

*直接任务通知*是经由中间对象的事件直接发送到一个任务，而不是间接的任务，诸如队列，事件组或信号量。将直接任务通知发送到任务会将目标任务通知的状态设置为“待处理”。就像任务可以被中间对象（例如信号量）阻塞以等待该信号量可用一样，任务也可以被任务通知阻塞以等待该通知的状态变为挂起状态。

向任务发送直接任务通知也可以选择 [通过以下其中一种方式更新目标通知的值](https://www.freertos.org/RTOS-task-notifications.html#uses)：

* 无论接收任务是否已读取正在覆盖的值，都将覆盖该值。
* 覆盖值，但仅当接收任务已读取要覆盖的值时。
* 在值中设置一位或多位。
* 增加值（加1）。

调用[xTaskNotifyWait（）/ xTaskNotifyWaitIndexed（）](https://www.freertos.org/xTaskNotifyWait.html) 读取通知值将清除该通知的状态为“未挂起”。还可以通过调用[xTaskNotifyStateClear（）/ xTaskNotifyStateClearIndexed（）](https://www.freertos.org/xTaskNotifyStateClear.html)将通知状态显式设置为“未挂起” 。

**注意：**数组中的每个通知都是独立运行的–一项任务一次只能在数组中的一个通知上阻塞，并且不会被发送给任何其他数组索引的通知所解除阻塞。

默认情况下，RTOS任务通知功能处于启用状态，并且可以通过在[FreeRTOSConfig.h](https://www.freertos.org/a00110.html)[中将configUSE\_TASK\_NOTIFICATIONS](https://www.freertos.org/a00110.html#configUSE_TASK_NOTIFICATIONS)设置为0来将其排除在构建之外（每个任务每个数组索引节省8个字节） 。

**重要说明：** FreeRTOS[流和消息缓冲区](https://www.freertos.org/RTOS-stream-message-buffers.html) 在数组索引0处使用任务通知。如果要在对Stream或Message Buffer API函数的调用中维护任务通知的状态，请在数组索引大于0处使用任务通知。

### 性能优势和使用限制

任务通知的灵活性使它们可以在需要创建单独的[队列](https://www.freertos.org/Embedded-RTOS-Queues.html)， [二值信号量](https://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html)， [计数信号量](https://www.freertos.org/Real-time-embedded-RTOS-Counting-Semaphores.html) 或[事件组的情况下使用](https://www.freertos.org/FreeRTOS-Event-Groups.html)。与使用中间对象（例如二值信号量）来解除阻塞任务相比，**使用**直接通知来解除阻塞RTOS任务的**速度快了45％ \*，**并 **使用了更少的RAM**。可以预期，这些性能优势需要一些用例限制：

1. 仅当只有一个任务可以作为事件的接收者时，才可以使用RTOS任务通知。但是，在大多数实际使用情况下都可以满足此条件，例如中断使执行任务处理的任务中断时，该任务将处理该中断接收到的数据。
2. 仅在使用RTOS任务通知代替队列的情况下：接收任务可以在“阻塞”状态下等待通知（因此不占用任何CPU时间），而发送任务不能在“阻塞”状态下等待消息。如果发送无法立即完成，则发送结束。

### 用例

通知是使用[xTaskNotifyIndexed（）](https://www.freertos.org/xTaskNotify.html) 和[xTaskNotifyGiveIndexed（）](https://www.freertos.org/xTaskNotifyGive.html) API函数（及其[等效](https://www.freertos.org/RTOS-task-notification-API.html)的[中断安全](https://www.freertos.org/RTOS-task-notification-API.html)函数 ）发送的，并保持待处理状态，直到接收到的RTOS任务调用[xTaskNotifyWaitIndexed（）](https://www.freertos.org/xTaskNotifyWait.html)或 [ulTask​​NotifyTakeIndexed（）](https://www.freertos.org/ulTaskNotifyTake.html) API函数中的任何一个 [为止](https://www.freertos.org/xTaskNotifyWait.html)。这些API函数中的每一个都有一个不带“索引”前缀的等效项。非“索引”版本始终在数组索引0处的任务通知上运行。例如，xTaskNotifyGive（TargetTask）等效于xTaskNotifyGiveIndexed（TargetTask，0） –两者都在处理任务TargetTask所引用任务的索引0处递增任务通知。

#### 例子

* [使用RTOS任务通知作为轻量级二值信号量](https://www.freertos.org/RTOS_Task_Notification_As_Binary_Semaphore.html)
* [使用RTOS任务通知作为轻量级计数信号量](https://www.freertos.org/RTOS_Task_Notification_As_Counting_Semaphore.html)
* [使用RTOS任务通知作为轻量级事件组](https://www.freertos.org/RTOS_Task_Notification_As_Event_Group.html)
* [使用RTOS任务通知作为轻量级邮箱](https://www.freertos.org/RTOS_Task_Notification_As_Mailbox.html)

**\*** *使用FreeRTOS V8.1.2中的二值信号量实现进行测量，并在优化-O2的情况下使用GCC进行编译，并且未定义configASSERT（）。使用FreeRTOS V8.2.0及更高版本中改进的二值信号量实现仍可实现35％的改进。*

# RTOS任务通知 用作轻量级二值信号量

相关页面：

* [RTOS任务通知](https://www.freertos.org/RTOS-task-notifications.html)
* [使用RTOS任务通知作为轻量级计数信号量](https://www.freertos.org/RTOS_Task_Notification_As_Counting_Semaphore.html)
* [使用RTOS任务通知作为轻量级事件组](https://www.freertos.org/RTOS_Task_Notification_As_Event_Group.html)
* [使用RTOS任务通知作为轻量级邮箱](https://www.freertos.org/RTOS_Task_Notification_As_Mailbox.html)

与**使用**二值信号量来解除阻塞任务相比，**使用** 直接通知来解除阻塞RTOS任务**要快45％，**并且 **使用更少的RAM**。此页面演示了如何完成此操作。

二值信号量是最大计数为1的信号量，因此是“二值”名称。任务只能“获取”信号量（如果可用），并且信号量仅在其计数为1时可用。

当使用任务通知代替二值信号量时，将使用接收任务的 [通知值](https://www.freertos.org/RTOS-task-notifications.html)代替二值信号量的计数值，并使用[ulTask​​NotifyTake（）](https://www.freertos.org/ulTaskNotifyTake.html) （或ulTask​​NotifyTakeIndexed（））API函数代替信号量的xSemaphoreTake（ ）API函数。ulTask​​NotifyTake（）函数的xClearOnExit参数设置为pdTRUE，因此每次收到通知时，计数值都将返回零，这是模拟二值信号量。

同样，使用[xTaskNotifyGive（）](https://www.freertos.org/xTaskNotifyGive.html)（或xTaskNotifyGiveIndexed（））或 [vTaskNotifyGiveFromISR（）](https://www.freertos.org/vTaskNotifyGiveFromISR.html)（或vTaskNotifyGiveIndexedFromISR（））函数代替信号量的xSemaphoreGive（）和xSemaphoreGiveFromISR（）函数。

请参见下面的示例。

**/\* This is an example of a transmit function in a generic**

**peripheral driver. An RTOS task calls the transmit function,**

**then waits in the Blocked state (so not using an CPU time)**

**until it is notified that the transmission is complete. The**

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

**is used to notify the task. \*/**

**/\* Stores the handle of the task that will be notified when the**

**transmission is complete. \*/**

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

**/\* The index within the target task's array of task notifications**

**to use. \*/**

**const UBaseType\_t xArrayIndex = 1;**

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

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

**{**

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

**is in progress. A mutex can be used to guard access to the**

**peripheral if necessary. \*/**

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

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

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

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

**transmission is complete. \*/**

**vStartTransmit( pcData, xDatalength );**

**}**

**/\*-----------------------------------------------------------\*/**

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

**void vTransmitEndISR( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

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

**a transmission was in progress. \*/**

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

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

[**vTaskNotifyGiveIndexedFromISR**](https://www.freertos.org/vTaskNotifyGiveFromISR.html)**( xTaskToNotify,**

**xArrayIndex,**

**&xHigherPriorityTaskWoken );**

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

**to notify. \*/**

**xTaskToNotify = NULL;**

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

**context switch should be performed to ensure the interrupt**

**returns directly to the highest priority task. The macro used**

**for this purpose is dependent on the port in use and may be**

**called portEND\_SWITCHING\_ISR(). \*/**

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

**}**

**/\*-----------------------------------------------------------\*/**

**/\* The task that initiates the transmission, then enters the**

**Blocked state (so not consuming any CPU time) to wait for it**

**to complete. \*/**

**void vAFunctionCalledFromATask( uint8\_t ucDataToTransmit,**

**size\_t xDataLength )**

**{**

**uint32\_t ulNotificationValue;**

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

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

**StartTransmission( ucDataToTransmit, xDataLength );**

**/\* Wait to be notified that the transmission is complete. Note**

**the first parameter is pdTRUE, which has the effect of clearing**

**the task's notification value back to 0, making the notification**

**value act like a binary (rather than a counting) semaphore. \*/**

**ulNotificationValue =** [**ulTaskNotifyTakeIndexed**](https://www.freertos.org/ulTaskNotifyTake.html)**( xArrayIndex,**

**pdTRUE,**

**xMaxBlockTime );**

**if( ulNotificationValue == 1 )**

**{**

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

**}**

**else**

**{**

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

**}**

**}**

# RTOS任务通知 用作轻量级计数信号量

相关页面：

* [RTOS任务通知](https://www.freertos.org/RTOS-task-notifications.html)
* [使用RTOS任务通知作为轻量级二值信号量](https://www.freertos.org/RTOS_Task_Notification_As_Binary_Semaphore.html)
* [使用RTOS任务通知作为轻量级事件组](https://www.freertos.org/RTOS_Task_Notification_As_Event_Group.html)
* [使用RTOS任务通知作为轻量级邮箱](https://www.freertos.org/RTOS_Task_Notification_As_Mailbox.html)

与通过信号量 取消任务相比，使用直接通知取消阻止RTOS任务的**速度快45％，**并且 **使用的RAM更少**。此页面演示了如何完成此操作。

计数信号量是一种信号量，它的计数值可以为零到创建信号量时所设置的最大值。任务能“获取”信号量（如果可用），而信号量仅在其计数大于零时才可用。

当使用任务通知代替计数信号量时，将使用接收任务的 [通知值](https://www.freertos.org/RTOS-task-notifications.html)代替计数信号量的计数值，并且使用[ulTask​​NotifyTake（）](https://www.freertos.org/ulTaskNotifyTake.html)（或ulTask​​NotifyTakeIndexed（））API函数代替信号量的xSemaphoreTake（ ）API函数。ulTask​​NotifyTake（）函数的xClearOnExit参数设置为pdFALSE，因此每次接收通知时，计数值仅递减（而不是清除）–模拟计数信号量。

同样，使用[xTaskNotifyGive（）](https://www.freertos.org/xTaskNotifyGive.html)（或xTaskNotifyGiveIndexed（））或 [vTaskNotifyGiveFromISR（）](https://www.freertos.org/vTaskNotifyGiveFromISR.html)（或vTaskNotifyGiveIndexedFromISR（））函数代替信号量的xSemaphoreGive（）和xSemaphoreGiveFromISR（）函数。

下面的第一个示例将接收任务的通知值用作计数信号量。第二个示例提供了更加实用和有效的实现。

**范例1：**

**/\* An interrupt handler that does not process interrupts directly,**

**but instead defers processing to a high priority RTOS task. The**

**ISR uses RTOS task notifications to both unblock the RTOS task**

**and increment the RTOS task's notification value. \*/**

**void vANInterruptHandler( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken;**

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

**prvClearInterruptSource();**

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

**If calling vTaskNotifyGiveFromISR() unblocks the handling**

**task, and the priority of the handling task is higher than**

**the priority of the currently running task, then**

**xHigherPriorityTaskWoken will be automatically set to pdTRUE. \*/**

**xHigherPriorityTaskWoken = pdFALSE;**

**/\* Unblock the handling task so the task can perform**

**any processing necessitated by the interrupt. xHandlingTask**

**is the task's handle, which was obtained when the task was**

[**created**](https://www.freertos.org/a00125.html)**. vTaskNotifyGiveFromISR() also increments**

**the receiving task's notification value. \*/**

**vTaskNotifyGiveFromISR( xHandlingTask, &xHigherPriorityTaskWoken );**

**/\* Force a context switch if xHigherPriorityTaskWoken is now**

**set to pdTRUE. The macro used to do this is dependent on**

**the port and may be called portEND\_SWITCHING\_ISR. \*/**

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

**}**

**/\*-----------------------------------------------------------\*/**

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

**needs servicing. \*/**

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

**{**

**BaseType\_t xEvent;**

**const TickType\_t xBlockTime = pdMS\_TO\_TICS( 500 );**

**uint32\_t ulNotifiedValue;**

**for( ;; )**

**{**

**/\* Block to wait for a notification. Here the RTOS**

**task notification is being used as a counting semaphore.**

**The task's notification value is incremented each time**

**the ISR calls vTaskNotifyGiveFromISR(), and decremented**

**each time the RTOS task calls ulTaskNotifyTake() - so in**

**effect holds a count of the number of outstanding interrupts.**

**The first parameter is set to pdFALSE, so the notification**

**value is only decremented and not cleared to zero, and one**

**deferred interrupt event is processed at a time. See**

**example 2 below for a more pragmatic approach. \*/**

**ulNotifiedValue = ulTaskNotifyTake( pdFALSE,**

**xBlockTime );**

**if( ulNotifiedValue > 0 )**

**{**

**/\* Perform any processing necessitated by the interrupt. \*/**

**xEvent = xQueryPeripheral();**

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

**{**

**vProcessPeripheralEvent( xEvent );**

**}**

**}**

**else**

**{**

**/\* Did not receive a notification within the expected**

**time. \*/**

**vCheckForErrorConditions();**

**}**

**}**

**}**

**范例2：**

此示例显示了针对RTOS任务的更为实用和有效的实现。在此实现中，从ulTask​​NotifyTake（）返回的值用于知道必须处理多少未完成的ISR事件，从而允许每次调用ulTask​​NotifyTake（）时将RTOS任务的通知计数清零。假定中断服务程序（ISR）如上面的示例1所示。

**/\* The index within the target task's array of task notifications**

**to use. \*/**

**const UBaseType\_t xArrayIndex = 0;**

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

**needs servicing. \*/**

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

**{**

**BaseType\_t xEvent;**

**const TickType\_t xBlockTime = pdMS\_TO\_TICS( 500 );**

**uint32\_t ulNotifiedValue;**

**for( ;; )**

**{**

**/\* As before, block to wait for a notification form the ISR.**

**This time however the first parameter is set to pdTRUE,**

**clearing the task's notification value to 0, meaning each**

**outstanding outstanding deferred interrupt event must be**

**processed before ulTaskNotifyTake() is called again. \*/**

**ulNotifiedValue = ulTaskNotifyTakeIndexed( xArrayIndex,**

**pdTRUE,**

**xBlockTime );**

**if( ulNotifiedValue == 0 )**

**{**

**/\* Did not receive a notification within the expected**

**time. \*/**

**vCheckForErrorConditions();**

**}**

**else**

**{**

**/\* ulNotifiedValue holds a count of the number of**

**outstanding interrupts. Process each in turn. \*/**

**while( ulNotifiedValue > 0 )**

**{**

**xEvent = xQueryPeripheral();**

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

**{**

**vProcessPeripheralEvent( xEvent );**

**ulNotifiedValue--;**

**}**

**else**

**{**

**break;**

**}**

**}**

**}**

**}**

**}**

# RTOS任务通知 用作轻量级事件组

相关页面：

* [RTOS任务通知](https://www.freertos.org/RTOS-task-notifications.html)
* [使用RTOS任务通知作为轻量级二值信号量](https://www.freertos.org/RTOS_Task_Notification_As_Binary_Semaphore.html)
* [使用RTOS任务通知作为轻量级计数信号量](https://www.freertos.org/RTOS_Task_Notification_As_Counting_Semaphore.html)
* [使用RTOS任务通知作为轻量级邮箱](https://www.freertos.org/RTOS_Task_Notification_As_Mailbox.html)

一个[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)是一组二进制标志（或bit）的，以其中的每一个应用程序写入器可以分配的含义。RTOS任务可以进入“阻塞”状态，以等待组中的一个或多个标志变为可用状态。处于“阻塞”状态时，RTOS任务不会消耗任何CPU时间。

当使用任务通知代替事件组时，将使用接收任务的 [通知值](https://www.freertos.org/RTOS-task-notifications.html)代替事件组，将接收任务的通知值内的位用作事件标志，并使用[xTaskNotifyWait（）](https://www.freertos.org/xTaskNotifyWait.html) API函数事件组的xEventGroupWaitBits（）API函数的名称。

同样，分别使用[xTaskNotify（）](https://www.freertos.org/xTaskNotify.html)和 [xTaskNotifyFromISR（）](https://www.freertos.org/xTaskNotifyFromISR.html) API函数（将其eAction参数设置为eSetBits）来设置位，分别代替xEventGroupSetBits（）和xEventGroupSetBitsFromISR（）函数。

与xEventGroupSetBitsFromISR（）相比，xTaskNotifyFromISR（）具有**显着的性能优势**，因为xTaskNotifyFromISR（）完全在ISR中执行，而xEventGroupSetBitsFromISR（）必须将某些处理交给RTOS守护程序任务。

与使用事件组不同，接收任务不能指定仅在同时激活多个位组合时才希望退出“阻塞”状态。取而代之的是，当任何位变为活动状态时，任务将被解除阻塞，并且必须自己测试位组合。

请参阅以下示例：

**/\* This example demonstrates a single RTOS task being used to process**

**events that originate from two separate interrupt service routines -**

**a transmit interrupt and a receive interrupt. Many peripherals will**

**use the same handler for both, in which case the peripheral's**

**interrupt status register can simply be bitwise ORed with the**

**receiving task's notification value.**

**First bits are defined to represent each interrupt source. \*/**

**#define TX\_BIT 0x01**

**#define RX\_BIT 0x02**

**/\* The handle of the task that will receive notifications from the**

**interrupts. The handle was obtained when the [task](https://www.freertos.org/a00125.html)**

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

**static TaskHandle\_t xHandlingTask;**

**/\*-----------------------------------------------------------\*/**

**/\* The implementation of the transmit interrupt service routine. \*/**

**void vTxISR( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

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

**prvClearInterrupt();**

**/\* Notify the task that the transmission is complete by setting the TX\_BIT**

**in the task's notification value. \*/**

**xTaskNotifyFromISR( xHandlingTask,**

**TX\_BIT,**

**eSetBits,**

**&xHigherPriorityTaskWoken );**

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

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

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

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

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

**}**

**/\*-----------------------------------------------------------\*/**

**/\* The implementation of the receive interrupt service routine is identical**

**except for the bit that gets set in the receiving task's notification value. \*/**

**void vRxISR( void )**

**{**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

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

**prvClearInterrupt();**

**/\* Notify the task that the reception is complete by setting the RX\_BIT**

**in the task's notification value. \*/**

**xTaskNotifyFromISR( xHandlingTask,**

**RX\_BIT,**

**eSetBits,**

**&xHigherPriorityTaskWoken );**

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

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

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

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

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

**}**

**/\*-----------------------------------------------------------\*/**

**/\* The implementation of the task that is notified by the interrupt service**

**routines. \*/**

**static void prvHandlingTask( void \*pvParameter )**

**{**

**const TickType\_t xMaxBlockTime = pdMS\_TO\_TICKS( 500 );**

**BaseType\_t xResult;**

**for( ;; )**

**{**

**/\* Wait to be notified of an interrupt. \*/**

**xResult = xTaskNotifyWait( pdFALSE, /\* Don't clear bits on entry. \*/**

**ULONG\_MAX, /\* Clear all bits on exit. \*/**

**&ulNotifiedValue, /\* Stores the notified value. \*/**

**xMaxBlockTime );**

**if( xResult == pdPASS )**

**{**

**/\* A notification was received. See which bits were set. \*/**

**if( ( ulNotifiedValue & TX\_BIT ) != 0 )**

**{**

**/\* The TX ISR has set a bit. \*/**

**prvProcessTx();**

**}**

**if( ( ulNotifiedValue & RX\_BIT ) != 0 )**

**{**

**/\* The RX ISR has set a bit. \*/**

**prvProcessRx();**

**}**

**}**

**else**

**{**

**/\* Did not receive a notification within the expected time. \*/**

**prvCheckForErrors();**

**}**

**}**

**}**

# RTOS任务通知 用作轻量级邮箱

相关页面：

* [RTOS任务通知](https://www.freertos.org/RTOS-task-notifications.html)
* [使用RTOS任务通知作为轻量级二值信号量](https://www.freertos.org/RTOS_Task_Notification_As_Binary_Semaphore.html)
* [使用RTOS任务通知作为轻量级计数信号量](https://www.freertos.org/RTOS_Task_Notification_As_Counting_Semaphore.html)
* [使用RTOS任务通知作为轻量级事件组](https://www.freertos.org/RTOS_Task_Notification_As_Event_Group.html)

RTOS任务通知可用于将数据发送到任务，但是比RTOS队列所能实现的限制要多得多，因为：

1. 只能发送32位值
2. 该值被保存为接收任务的[通知值](https://www.freertos.org/RTOS-task-notifications.html)，并且在任何一次只能有一个通知值

因此，短语“轻量邮箱”优先于“轻量队列”。任务的通知值是邮箱值。

使用[xTaskNotify（）](https://www.freertos.org/xTaskNotify.html)（或xTaskNotifyIndexed（））和[xTaskNotifyFromISR（）](https://www.freertos.org/xTaskNotifyFromISR.html)（或xTaskNotifyIndexedFromISR（））API函数将数据发送到任务，[并将](https://www.freertos.org/xTaskNotify.html)其eAction参数设置为eSetValueWithOverwrite或eSetValueWithoutOverwrite。如果eAction设置为eSetValueWithOverwrite，则即使接收任务已经有待处理的通知，接收任务的通知值也会更新。如果eAction设置为eSetValueWithoutOverwrite，则仅当接收任务尚未有待处理的通知时，才更新接收任务的通知值-因为更新通知值将覆盖接收任务处理之前的值。

任务可以使用[xTaskNotifyWait（）](https://www.freertos.org/xTaskNotifyWait.html) （或xTaskNotifyWaitIndexed（））读取自己的通知值。

有关示例，请参阅相关API函数的文档。

# RTOS流和消息缓冲区 [[任务间通信和同步](https://www.freertos.org/Inter-Task-Communication.html)]

|  |
| --- |
| [**可从FreeRTOS V10.0.0获得**](https://www.freertos.org/upgrading-to-FreeRTOS-V8.html) |

## 介绍

流缓冲区是[RTOS任务](https://www.freertos.org/a00015.html)到RTOS任务，是对任务通信原语的中断。与大多数其他FreeRTOS通信原语不同，它们针对单读取器单写入器方案进行了优化，例如将数据从中断服务例程传递到任务，或从一个微控制器内核传递到双核CPU上的另一个内核。数据通过复制传递–发送者将数据复制到缓冲区中，并通过读取将其复制出缓冲区。

流缓冲区传递连续的字节流。消息缓冲区传递可变大小但不连续的消息。消息缓冲区使用流缓冲区进行数据传输。

**重要说明**：在FreeRTOS对象中，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）是唯一的，它假定只有一个任务或中断将写入缓冲区（写程序），并且只有一个任务或中断将从缓冲区（读取器）读取。对于作者和读者来说，不同的任务或中断是安全的，但是与其他FreeRTOS对象不同，拥有多个不同的作者或多个读者是不安全的。如果存在多个不同的编写器，则应用程序编写器必须将每个调用都置于编写API函数（例如[xStreamBufferSend（）](https://www.freertos.org/xStreamBufferSend.html)）在临界区中，并使用0的发送块时间。同样，如果要有多个不同的读取器，则应用程序编写者必须[将对](https://www.freertos.org/xStreamBufferReceive.html)读取API函数（例如[xStreamBufferReceive（）](https://www.freertos.org/xStreamBufferReceive.html)）的每次调用放在临界区中并使用接收块时间为0。

### 进一步阅读

以下页面更详细地描述了流缓冲区和消息缓冲区，并提供了它们分别用于实现对任务的中断以及对处理器核心与处理器核心之间的通信的示例。

[有关流缓冲区的](https://www.freertos.org/RTOS-stream-buffer-example.html)  
[更多信息](https://www.freertos.org/RTOS-message-buffer-example.html) [...](https://www.freertos.org/RTOS-stream-buffer-example.html)[有关消息缓冲区的更多信息...](https://www.freertos.org/RTOS-message-buffer-example.html)  
[有关使用消息缓冲区进行核心到核心通信的博客...](https://www.freertos.org/articles/001_simple_freertos_core_to_core_communication/simple_freertos_core_to_core_communication_AMP.html)

# RTOS流缓冲区 [[流缓冲区和消息缓冲区](https://www.freertos.org/RTOS-stream-message-buffers.html)]

|  |
| --- |
| [**可从FreeRTOS V10.0.0获得**](https://www.freertos.org/upgrading-to-FreeRTOS-V8.html) |

在本页面：

1. [介绍](https://www.freertos.org/RTOS-stream-buffer-example.html#introduction)
2. [入门](https://www.freertos.org/RTOS-stream-buffer-example.html#getting_started)
3. [阻止读取和触发电平](https://www.freertos.org/RTOS-stream-buffer-example.html#blocking_reads)
4. [阻止写入](https://www.freertos.org/RTOS-stream-buffer-example.html#blocking_writes)
5. [发送和接收完整的宏（供多核使用）](https://www.freertos.org/RTOS-stream-buffer-example.html#macros)

## 介绍

流缓冲区允许将字节流从中断服务例程传递到任务，或从一个任务传递到另一任务。字节流可以具有任意长度，并且不一定具有开头或结尾。可以一次性写入任意数量的字节，并且可以一次性读取任意数量的字节。数据通过复制传递–发送者将数据复制到缓冲区中，并通过读取将其复制出缓冲区。

与大多数其他FreeRTOS通信原语不同，流缓冲区针对单读取器单写入器场景进行了优化，例如将数据从中断服务例程传递到任务，或从一个微控制器内核传递到双核CPU上的另一个内核。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用流缓冲区功能。

流缓冲区实现将[直接用于任务通知](https://www.freertos.org/RTOS-task-notifications.html)。因此，调用将调用任务置于“阻塞”状态的流缓冲区API函数可以更改调用任务的通知状态和值。

**重要说明**：在FreeRTOS对象中，流缓冲区实现（[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-example.html) 实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）是唯一的，它假定只有一个任务或中断将写入缓冲区（写程序），并且只有一个任务或中断将从缓冲区（读取器）读取。对于作者和读者来说，不同的任务或中断是安全的，但是与其他FreeRTOS对象不同，拥有多个不同的作者或多个读者是不安全的。如果存在多个不同的编写器，则应用程序编写器必须将每个调用都置于编写API函数（例如[xStreamBufferSend（）](https://www.freertos.org/xStreamBufferSend.html)）在临界区中，并使用0的发送块时间。同样，如果要有多个不同的读取器，则应用程序编写者必须[将对](https://www.freertos.org/xStreamBufferReceive.html)读取API函数（例如[xStreamBufferReceive（）](https://www.freertos.org/xStreamBufferReceive.html)）的每次调用放在临界区中并使用接收块时间为0。

### 入门

所述**FreeRTOS的/演示/普通/最小/ StreamBufferInterrupt.c**源文件提供了如何使用流缓冲器从中断服务例程的数据传递到一个任务重注释的示例。

有关与[流缓冲区](https://www.freertos.org/RTOS-stream-buffer-API.html)相关的API函数的列表，请参见用户文档的[流缓冲区部分](https://www.freertos.org/RTOS-stream-buffer-API.html)，在许多情况下，包括演示所使用功能的代码段。

## 阻止读取和触发电平

[xStreamBufferReceive（）](https://www.freertos.org/xStreamBufferReceive.html)用于从RTOS任务从流缓冲区中读取数据。 [xStreamBufferReceiveFromISR（）](https://www.freertos.org/xStreamBufferReceiveFromISR.html)用来从中断服务例程（ISR）从流缓冲区中读取数据。

xStreamBufferReceive（）允许指定块时间。如果在任务使用xStreamBufferReceive（）从碰巧为空的流缓冲区中读取时指定了非零的块时间，则该任务将被置于“阻塞”状态（因此它不会消耗任何CPU时间，并且其他任务可以运行）直到指定数量的数据在流缓冲区中变得可用，或者块时间到期为止。从阻塞状态中删除等待数据的任务之前，流缓冲区中必须存在的数据量称为流缓冲区的触发级别。例如：

* 如果在读取触发级别为1的空流缓冲区时阻止了任务，则当将单个字节写入缓冲区或任务的阻止时间到期时，任务将被取消阻止。
* 如果在读取触发器级别为10的空流缓冲区时阻止了任务，则直到该流缓冲区包含至少10个字节或任务的阻止时间到期后，该任务才会被取消阻止。

如果读取任务的阻止时间在达到触发级别之前已过期，则该任务仍将接收，但是实际上有许多字节可用。

**笔记：**

* 将触发级别设置为0是无效的。尝试将触发级别设置为0将导致使用触发级别1。
* 指定大于流缓冲区大小的触发级别也是无效的。

流缓冲区的触发级别最初是[在创建](https://www.freertos.org/xStreamBufferCreate.html)流缓冲区时设置的 ，然后可以使用[xStreamBufferSetTriggerLevel（）](https://www.freertos.org/xStreamBufferSetTriggerLevel.html) API函数进行更改。

## 阻止写入

[xStreamBufferSend（）](https://www.freertos.org/xStreamBufferSend.html)）用于将数据从RTOS任务发送到流缓冲区。 [xStreamBufferSendFromISR（）](https://www.freertos.org/xStreamBufferSendFromISR.html)用来将数据从中断服务程序（ISR）发送到流缓冲区。

如果在任务使用xStreamBufferSend（）写入恰好已满的流缓冲区时指定了非零的块时间，则该任务将被置于“阻塞”状态（这样就不会占用任何CPU时间，其他任务也可以运行）直到流缓冲区中的任何空间变为可用，或者块时间到期为止。

## 发送完整的和接收完整的宏

[另请参阅[有关将消息缓冲区用于双核-核心到核心通信](https://www.freertos.org/articles/001_simple_freertos_core_to_core_communication/simple_freertos_core_to_core_communication_AMP.html)的[博客](https://www.freertos.org/articles/001_simple_freertos_core_to_core_communication/simple_freertos_core_to_core_communication_AMP.html)。]

### sbSEND\_COMPLETED（）（和sbSEND\_COMPLETED\_FROM\_ISR（））

sbSEND\_COMPLETED（）是在将数据写入流缓冲区时调用的宏（在FreeRTOS API函数内部）。它采用一个参数，该参数是已更新的流缓冲区的句柄。

sbSEND\_COMPLETED（）检查流缓冲区上是否有等待任务等待数据，如果有，则将该任务从“阻止”状态中删除。

通过在FreeRTOSConfig.h中提供他们自己的sbSEND\_COMPLETED（）实现，应用程序编写者可以更改此默认行为。当使用流缓冲区在多核处理器上的内核之间传递数据时，这很有用。在那种情况下，可以实现sbSEND\_COMPLETED（）在另一个CPU内核中生成中断，然后该中断的服务例程可以使用xStreamBufferSendCompletedFromISR（）API函数来检查并在必要时取消阻止正在等待数据的任务。该**FreeRTOS的/演示/通用/最小/ MessageBufferAMP.c**源文件提供正是场景的大量注释的例子。

### sbRECEIVE\_COMPLETED（）（和sbRECEIVE\_COMPLETED\_FROM\_ISR（））

sbRECEIVE\_COMPLETED（）是sbSEND\_COMPLETED（）的接收等效项。从流缓冲区读取数据时（在FreeRTOS API函数内部）将调用它。宏检查流缓冲区上是否有任务被阻塞，以等待缓冲区中的可用空间；如果是，则将该任务从“阻塞”状态中删除。与sbSEND\_COMPLETED（）一样，通过在FreeRTOSConfig.h中提供替代实现，sbRECEIVE\_COMPLETED（）的默认行为

# RTOS消息缓冲区 [[流缓冲区和消息缓冲区](https://www.freertos.org/RTOS-stream-message-buffers.html)]

|  |
| --- |
| [**可从FreeRTOS V10.0.0获得**](https://www.freertos.org/upgrading-to-FreeRTOS-V8.html) |

在本页面：

1. [介绍](https://www.freertos.org/RTOS-message-buffer-example.html#introduction)
2. [入门](https://www.freertos.org/RTOS-message-buffer-example.html#getting_started)
3. [调整消息缓冲区的大小](https://www.freertos.org/RTOS-message-buffer-example.html#sizing)
4. [阻止读写](https://www.freertos.org/RTOS-message-buffer-example.html#blocking_writes)
5. [发送和接收完整的宏（供多核使用）](https://www.freertos.org/RTOS-message-buffer-example.html#macros)

## 介绍

消息缓冲区允许将可变长度的离散消息从中断服务例程传递到任务，或从一个任务传递到另一任务。例如，长度为10、20和123字节的消息都可以写入和读取相同的消息缓冲区。与使用[流缓冲区不同](https://www.freertos.org/RTOS-stream-buffer-example.html)，10字节的消息只能作为10字节的消息读出，而不能作为单个字节读出。消息缓冲区建立在流缓冲区之上（也就是说，它们使用流缓冲区实现）。

数据通过副本传递到消息缓冲区中-发送方将数据复制到缓冲区中，并通过读取将其复制出缓冲区。

另请参阅[configMESSAGE\_BUFFER\_LENGTH\_TYPE](https://www.freertos.org/a00110.html#configMESSAGE_BUFFER_LENGTH_TYPE) 配置参数的定义 。

与大多数其他FreeRTOS通信原语不同，流缓冲区以及消息缓冲区针对单读取器单写入器场景进行了优化，例如将数据从中断服务例程传递到任务，或从一个微控制器内核传递到双核CPU上的另一个内核。

通过 在构建中包含FreeRTOS / source / stream\_buffer.c源文件来启用消息缓冲区功能。

消息缓冲区实现将[直接用于任务通知](https://www.freertos.org/RTOS-task-notifications.html)。因此，调用将呼叫任务置于“已阻止”状态的消息缓冲区API函数可以更改呼叫任务的通知状态和值。

**重要说明**：在FreeRTOS对象中，流缓冲区实现（消息缓冲区实现也是如此，因为消息缓冲区建立在流缓冲区的顶部）是唯一的，它假定只有一个任务或中断将写入缓冲区（写程序），并且只有一个任务或中断将从缓冲区（读取器）读取。对于作者和读者来说，不同的任务或中断是安全的，但是与其他FreeRTOS对象不同，拥有多个不同的作者或多个读者是不安全的。如果存在多个不同的编写器，则应用程序编写器必须将每个调用都置于编写API函数（例如[xStreamBufferSend（）](https://www.freertos.org/xStreamBufferSend.html)）在临界区中，并使用0的发送块时间。同样，如果要有多个不同的读取器，则应用程序编写者必须[将对](https://www.freertos.org/xStreamBufferReceive.html)读取API函数（例如[xStreamBufferReceive（）](https://www.freertos.org/xStreamBufferReceive.html)）的每次调用放在临界区中并使用接收块时间为0。

### 入门

所述**FreeRTOS的/演示/普通/最小/ MessageBufferAMP.c**源文件提供了如何使用一个消息缓冲器从一个MCU芯通过可变长度的数据到另一个上的多芯MCU重注释的示例。那是相当高级的场景，但是在较简单的单核场景中，用于创建消息，从消息缓冲区发送消息和从消息缓冲区接收消息的机制是相同的，与演示中不同，该示例不需要覆盖[sbSEND\_COMPLETE（）](https://www.freertos.org/RTOS-stream-buffer-example.html#macros)宏。

有关与[消息缓冲区](https://www.freertos.org/RTOS-message-buffer-API.html)相关的API函数的列表，请参见用户文档的[消息缓冲区部分](https://www.freertos.org/RTOS-message-buffer-API.html)，在许多情况下，包括演示所使用功能的代码段。

## 调整消息缓冲区的大小

为了使消息缓冲区能够处理大小可变的消息，每条消息的长度都在消息本身之前写入消息缓冲区中（这在FreeRTOS API函数内部发生）。长度存储在变量中，变量的类型由FreeRTOSConfig.h中的configMESSAGE\_BUFFER\_LENGTH\_TYPE常量设置。如果未定义，configMESSAGE\_BUFFER\_LENGTH\_TYPE的默认类型为size\_t。在32位体系结构上，size\_t通常为4字节。因此，作为示例，当configMESSAGE\_BUFFER\_LENGTH为4个字节时，将10个字节的消息写入消息缓冲区实际上将消耗14个字节的缓冲区空间。同样，将100字节的消息写入消息缓冲区实际上将存储104字节的缓冲区空间。

## 阻止读写

[xMessageBufferReceive（）](https://www.freertos.org/xMessageBufferReceive.html)）用于从RTOS任务的消息缓冲区中读取数据。 [xMessageBufferReceiveFromISR（）](https://www.freertos.org/xMessageBufferReceiveFromISR.html)用来从中断服务程序（ISR）的消息缓冲区中读取数据。 [xMessageBufferSend（）](https://www.freertos.org/xMessageBufferSend.html)）用于将数据从RTOS任务发送到消息缓冲区。 [xMessageBufferSendFromISR（）](https://www.freertos.org/xMessageBufferSendFromISR.html)）用于将数据从中断服务例程（ISR）发送到消息缓冲区。

如果在任务使用xMessageBufferReceive（）从碰巧为空的消息缓冲区中读取时指定了非零的阻止时间，则该任务将被置于“阻止”状态（因此它不会消耗任何CPU时间，并且其他任务可以运行）直到消息缓冲区中的数据可用或阻止时间到期为止。

如果在任务使用xMessageBufferSend（）写入恰好已满的消息缓冲区时指定了非零的阻止时间，则该任务将被置于“阻止”状态（因此它不消耗任何CPU时间，其他任务可以运行）直到消息缓冲区中的任何一个空间可用或阻止时间到期为止。

## 发送完整的和接收完整的宏

由于消息缓冲区建立在流缓冲区上，因此sbSEND\_COMPLETE（）和sbRECEIVE\_COMPLETE（）宏的行为与[描述流缓冲区](https://www.freertos.org/RTOS-stream-buffer-example.html)的[页面所描述的](https://www.freertos.org/RTOS-stream-buffer-example.html)完全相同 。

# 软件计时器

在此页面上：

* [简而言之，软件计时器](https://www.freertos.org/RTOS-software-timer.html#software-timers-in-a-nutshell)
* [软件计时器实现中的效率注意事项](https://www.freertos.org/RTOS-software-timer.html#efficiency-considerations-in-software-timer-implementations)
* [有关编写计时器回调函数的重要信息](https://www.freertos.org/RTOS-software-timer.html#important-information-on-writing-timer-callback-functions)
* [计时器服务/守护程序任务以及计时器命令队列](https://www.freertos.org/RTOS-software-timer-service-daemon-task.html)
* [配置应用程序以使用软件计时器](https://www.freertos.org/Configuring-a-real-time-RTOS-application-to-use-software-timers.html)
* [一键式计时器与自动重装计时器](https://www.freertos.org/One-shot-Vs-auto-reload-real-time-software-timers.html)
* [重置软件计时器](https://www.freertos.org/Resetting-an-RTOS-software-timer.html)
* [火](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html)

### 简而言之，软件计时器

软件计时器（或仅是“计时器”）允许将来在设定的时间执行功能。计时器执行的功能称为计时器的回调函数。计时器启动到执行回调函数之间的时间称为计时器的时间段。简而言之，当计时器的时间段到期时，将执行计时器的回调函数。

注意，必须先显式创建一个软件计时器，然后才能使用它。

### 软件计时器实现中的效率注意事项

软件计时器功能易于实现，但难以有效实现。FreeRTOS实现不会从中断上下文执行计时器回调函数，不会消耗**任何**处理时间，除非计时器实际上已过期，不会在滴答中断中增加任何处理开销，并且在禁用中断时不会遍历任何链接列表结构。

计时器服务任务（主要）利用了现有的FreeRTOS功能，允许将计时器功能添加到应用程序中，而对应用程序可执行二进制文件的大小影响最小。

### 有关编写计时器回调函数的重要信息

计时器回调函数在计时器服务任务的上下文中执行。因此，**重要的**是定时器回调函数从来没有试图阻止。例如，在访问队列或信号量时，计时器回调函数不得调用vTaskDelay（），vTaskDelayUntil（）或指定非零的块时间。

# 软件计时器 [[有关软件计时器的更多信息…](https://www.freertos.org/RTOS-software-timer.html) ]

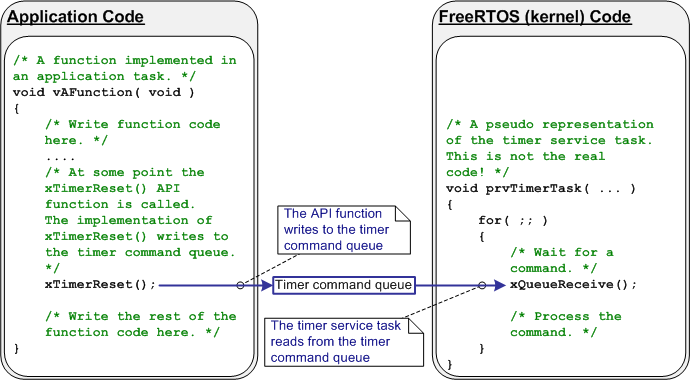
另请参见“ [RTOS守护进程任务启动挂钩”](https://www.freertos.org/a00016.html#DaemonTaskStartupHook)功能。

### 计时器服务/守护程序任务以及计时器命令队列

计时器功能是可选的，不是FreeRTOS内核的一部分。相反，它由计时器服务（或守护程序）任务提供。

FreeRTOS提供了一组与计时器相关的API函数。其中许多功能使用标准的FreeRTOS队列将命令发送到计时器服务任务。为此目的使用的队列称为“定时器命令队列”。“定时器命令队列”是FreeRTOS定时器实现专用的，不能直接访问。

下图演示了这种情况。左侧的代码代表一个功能，该功能是用户应用程序的一部分，并从作为同一用户应用程序的一部分创建的任务中调用。右侧的代码表示计时器服务任务的实现。计时器命令队列是应用程序任务和计时器服务任务之间的链接。在这种演示情况下，将从应用程序代码中调用xTimerReset（）API函数。这导致将重置命令发送到计时器命令队列，以由计时器服务任务进行处理。应用程序代码仅调用xTimerReset（）API函数–它不能（也不能）直接访问计时器命令队列。

  
应用程序代码，FreeRTOS  
计时器API，计时器命令队列和计时器服务任务的上下文。

# 软件计时器 [[有关软件计时器的更多信息…](https://www.freertos.org/RTOS-software-timer.html) ]

### 配置应用程序以使用软件计时器

要使[FreeRTOS软件计时器API](https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html) 在应用程序中可用，只需：

1. 将FreeRTOS / Source / timers.c源文件添加到您的项目中，然后
2. 在应用程序FreeRTOSConfig.h头文件中的下表中定义常量。

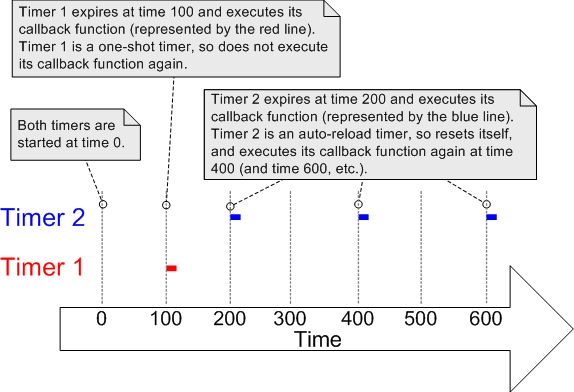
|  |  |
| --- | --- |
| **不变** | **描述** |
| configUSE\_TIMERS | 设置为1以包括计时器功能。当configUSE\_TIMERS设置为1时，将在RTOS调度程序启动时自动创建计时器服务任务。 |
| configTIMER\_TASK\_PRIORITY | 设置计时器服务任务的优先级。与所有任务一样，计时器服务任务可以以0到（configMAX\_PRIORITIES – 1）之间的任意优先级运行。  需要仔细选择该值以满足应用程序的要求。例如，如果将计时器服务任务设置为系统中的最高优先级任务，则发送到计时器服务任务的命令（调用计时器API函数时）和过期的计时器都将立即得到处理。相反，如果为计时器服务任务赋予了较低的优先级，则直到计时器服务任务是能够运行的最高优先级任务，才会处理发送到计时器服务任务和过期计时器的命令。但是，在这里值得注意的是，计时器的到期时间是相对于发送命令的时间而不是相对于处理命令的时间而计算的。 |
| configTIMER\_QUEUE\_LENGTH | 这将设置计时器命令队列可以一次保存的未处理命令的最大数量。  计时器命令队列可能填满的原因包括：   * 在启动RTOS调度程序之前，因此在创建计时器服务任务之前，请进行多个计时器API函数调用。 * 从中断服务程序（ISR）进行多个（中断安全的）定时器API函数调用。 * 从一个优先级高于计时器服务任务优先级的任务中进行多个计时器API函数调用。 |
| configTIMER\_TASK\_STACK\_DEPTH | 设置分配给计时器服务任务的堆栈大小（用字而不是字节）。  计时器回调函数在计时器服务任务的上下文中执行。因此，计时器服务任务的堆栈要求取决于计时器回调函数的堆栈要求。 |

# 软件计时器 [[有关软件计时器的更多信息…](https://www.freertos.org/RTOS-software-timer.html) ]

### 一键式计时器与自动重装计时器

计时器有两种类型，一击式计时器和自动重装计时器。一旦启动，一次性计时器将仅执行一次其回调函数。可以手动重新启动它，但不会自动重新启动自身。相反，一旦启动，自动重载计时器将在每次执行其回调函数后自动重新启动自身，从而导致定期执行回调。

下图的时间轴说明了单次定时器和自动重载定时器之间的行为差​​异。在此图中，定时器1是周期等于100的单触发定时器，定时器2是周期等于200的自动重载定时器。

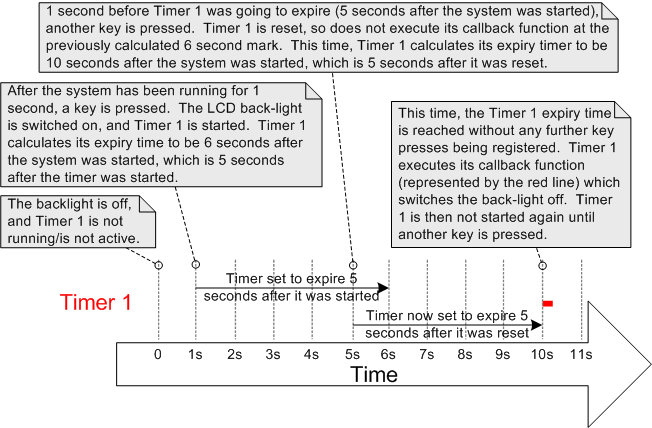
  
一击式计时器和自动重装计时器的行为

# 软件计时器 [[有关软件计时器的更多信息…](https://www.freertos.org/RTOS-software-timer.html) ]

### 重置软件计时器

可以重新设置已经开始运行的计时器。重置计时器会导致计时器重新计算其到期时间，因此到期时间相对于重置计时器的时间，而不是相对于计时器最初启动的时间。下图演示了此行为，其中计时器1是单触发计时器，其周期等于5秒。

在所示的示例中，假定按下某个键时该应用程序将打开LCD背光，并且背光将保持点亮状态，直到5秒钟过去而没有任何键被按下。这5秒钟过去后，计时器1用于关闭LCD背光灯。

  
重置计时器时的计时器行为

# 事件位（或*标志*）和事件组 [[任务间通信和同步](https://www.freertos.org/Inter-Task-Communication.html)]

|  |
| --- |
| [**提示：在许多情况下，“任务通知”可以为事件组提供轻巧的选择**](https://www.freertos.org/RTOS_Task_Notification_As_Event_Group.html) |

在本页面：

* [事件位（事件标志）](https://www.freertos.org/FreeRTOS-Event-Groups.html#event_bits)
* [活动组](https://www.freertos.org/FreeRTOS-Event-Groups.html#event_groups)
* [事件组和事件位数据类型](https://www.freertos.org/FreeRTOS-Event-Groups.html#data_types)
* [事件组RTOS API函数](https://www.freertos.org/FreeRTOS-Event-Groups.html#API)
* [实施事件组时，RTOS必须克服的挑战](https://www.freertos.org/FreeRTOS-Event-Groups.html#challenges)
* [范例程式码](https://www.freertos.org/FreeRTOS-Event-Groups.html#examples)

### 事件位（事件标志）

事件位用于指示事件是否发生。事件位通常称为事件*标志*。例如，一个应用程序可以：

* 设置为1时，定义一个位（或标志），表示“已接收到一条消息并准备好进行处理”；设置为0时，定义为“没有等待处理的消息”。
* 设置为1时，定义一个位（或标志），表示“应用程序已将准备发送到网络的消息排队”，当它设置为“没有队列准备发送到网络的消息”时，设置为0。
* 定义一个位（或标志），当它设置为1时，表示“是时候将心跳消息发送到网络上”，而当它设置为0时，则表示“还没有时间发送另一个心跳消息”。

### 活动组

事件组是一组事件位。事件组中的各个事件位由位号引用。扩展上面提供的示例：

* 表示“已接收到一条消息并准备进行处理”的事件位可能是事件组中的位号0。
* 表示“应用程序已将准备发送到网络的消息排队了”的事件位可能是同一事件组中的第1位。
* 表示“到了将心跳消息发送到网络的时候了”的事件位可能是同一事件组中的第2位。

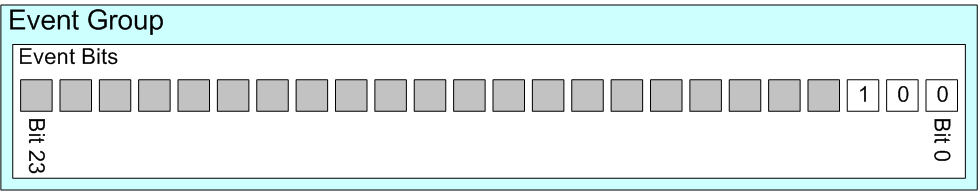
### 事件组和事件位数据类型

事件组由EventGroupHandle\_t类型的变量引用。

如果将[configUSE\_16\_BIT\_TICKS](https://www.freertos.org/a00110.html#configUSE_16_BIT_TICKS) 设置为1，则事件组中存储的位数（或标志）为8；如果将 [configUSE\_16\_BIT\_TICKS](https://www.freertos.org/a00110.html#configUSE_16_BIT_TICKS)设置为0，则事件为24。对configUSE\_16\_BIT\_TICKS的依赖性源于内部实现的线程本地存储所使用的数据类型。任务。

事件组中的所有事件位都存储在EventBits\_t类型的单个无符号变量中。事件位0存储在位位置0，事件位1存储在位位置1，依此类推。

下图显示了一个24位事件组，该组使用三个位来保存已描述的三个示例事件。在图像中，仅事件位2被设置。

  
**包含24个事件位的事件组，其中只有三个在使用中**

### 事件组RTOS API函数

提供了事件组[API功能，](https://www.freertos.org/event-groups-API.html)这些[功能](https://www.freertos.org/event-groups-API.html)使任务可以在事件组中[设置](https://www.freertos.org/xEventGroupSetBits.html)一个或多个事件位，在事件组中 [清除](https://www.freertos.org/xEventGroupClearBits.html)一个或多个事件位，并[暂挂](https://www.freertos.org/xEventGroupWaitBits.html)（进入“阻止”状态，因此任务不会占用任何处理时间）以等待一组一个或多个事件位在事件组中被置位。

事件组还可以用于[同步](https://www.freertos.org/xEventGroupSync.html)任务，从而创建通常称为“约会”的任务。任务同步点是应用程序代码中的一个位置，在该位置，任务将在“阻塞”状态下等待（不占用任何CPU时间），直到参与同步的所有其他任务也都到达其同步点为止。

### 实施事件组时，RTOS必须克服的挑战

实施事件组时，RTOS必须克服的两个主要挑战是：

1. 避免在用户的应用程序中创建竞争条件：

在以下情况下，事件组实现将在应用程序中创建竞争条件：

* + 目前尚不清楚谁负责清除单个位（或标志）。
  + 尚不清楚何时清除一点。
  + 尚不清楚在任务退出测试该位值的API函数时是否设置了位或清除了位（此后可能是另一个任务或中断更改了位的状态）。

FreeRTOS事件组的实现通过包含内置的智能功能来确保设置，测试和清除位显得原子化，从而消除了潜在的竞争条件。线程本地存储和对API函数返回值的谨慎使用使这成为可能。

1. 避免不确定性：

事件组概念隐含着不确定的行为，因为它不知道事件组上有多少个任务被阻止，因此，当设置了事件位时，也不知道需要测试多少条件或取消阻止任务。

该FreeRTOS的质量标准并**没有**允许同时中断被禁止，以进行非确定性的行为，或中断服务程序中。为确保在设置事件位时不违反这些严格的质量标准：

* + RTOS调度程序的锁定机制用于确保从RTOS任务设置事件位时，中断保持启用状态。
  + 当试图从中断服务程序中设置事件位时，使用集中式延迟中断机制来延迟将位设置为任务的操作。

### 范例程式码

[API](https://www.freertos.org/event-groups-API.html) 文档 中提供了示例代码段，并且在EventGroupsDemo.c 标准演示任务集中提供了完整的示例（源文件位于FreeRTOS 主[下载](https://www.freertos.org/a00104.html)的FreeRTOS / Demo / Common / Minimal目录中）。

# 来源组织

每个RTOS端口都带有一个预先配置的演示应用程序，该应用程序已经构建了必要的RTOS源文件，并包括了必要的RTOS头文件。强烈建议将提供的演示用作所有基于FreeRTOS的新应用程序的基础。提供此页面以帮助查找和理解所提供的项目。

### 基本目录结构

FreeRTOS下载包括每个处理器端口和每个演示应用程序的源代码。将所有端口放在一次下载中可以大大简化分发，但是文件数量似乎令人生畏。但是目录结构非常简单，FreeRTOS实时内核仅包含***在3个文件中***（如果需要[软件计时器](https://www.freertos.org/RTOS-software-timer.html)，[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)或协程功能，则需要其他文件）。

从顶部开始，下载内容被分为三个子目录。FreeRTOS，FreeRTOS-Labs和FreeRTOS-Plus。这些显示如下：

**+ -FreeRTOS-Labs包含FreeRTOS-Labs**

**|**

**+ -FreeRTOS-Plus包含FreeRTOS** [**+**](https://www.freertos.org/FreeRTOS-Plus/index.html)**组件和演示项目。**

**|**

**+ -FreeRTOS包含FreeRTOS实时内核源代码**

**文件和演示项目**

FreeRTOS-Labs和FreeRTOS-Plus目录树包含多个描述其内容的自述文件。FreeRTOS-Labs库具有全部功能，但是正在进行优化或重构以提高内存使用率，模块化，文档，演示可用性或测试范围。

### FreeRTOS内核目录结构

FreeRTOS内核的核心源文件和演示项目包含在两个子目录中，如下所示：

**FreeRTOS**

**|**

**+ -Demo包含演示应用程序项目。**

**|**

**+ -Source包含实时内核源代码。**

核心RTOS代码包含在三个文件，这是所谓的被称为tasks.c，queue.c和list.c。这三个文件位于FreeRTOS / Source目录中。同一目录包含两个名为timers.c和croutine.c的可选文件，它们分别实现软件计时器和协程功能。

每个受支持的处理器体系结构都需要少量体系结构特定的RTOS代码。这是RTOS可移植层，位于FreeRTOS / Source / Portable / [compiler] / [architecture]子目录中，其中[compiler]和[architecture]是用于创建端口的编译器，其上的体系结构该端口分别运行。

由于在[内存管理页面上所述](https://www.freertos.org/a00111.html)的原因，样本堆分配方案也位于可移植层中。各种样本heap\_x.c文件位于FreeRTOS / Source / portable / MemMang目录中。

便携式层目录的示例：

* 如果将TriCore 1782端口与GCC编译器一起使用：

TriCore特定文件（port.c）在FreeRTOS / Source / Portable / GCC / TriCore\_1782目录中。除了FreeRTOS / Source / Portable / MemMang之外，所有其他FreeRTOS / Source / Portable子目录都可以忽略或删除。

* 如果将Renesas RX600端口与IAR编译器一起使用：

RX600专用文件（port.c）在FreeRTOS / Source / Portable / IAR / RX600目录中。除了FreeRTOS / Source / Portable / MemMang之外，所有其他FreeRTOS / Source / Portable子目录都可以忽略或删除。

* 对于所有端口，依此类推...

所述的结构FreeRTOS的/源目录下面被示出。

**FreeRTOS**

**|**

**+ -Source核心FreeRTOS内核文件**

**|**

**+ -include核心FreeRTOS内核头文件**

**|**

**+-便携式处理器专用代码。**

**|**

**+-编译器x编译器x支持的所有端口**

**+-编译器y编译器y支持的所有端口**

**+ -MemMang示例堆实现**

FreeRTOS下载还包含用于每种处理器体系结构和编译器端口的演示应用程序。演示应用程序的大部分代码对所有端口通用，并包含在FreeRTOS / Demo / Common / Minimal目录中（位于FreeRTOS / Demo / Common / Full目录中的代码是旧代码，仅由PC端口使用） 。

其余的FreeRTOS / Demo子目录包含用于构建单个演示应用程序的预配置项目。目录被命名以指示它们所关联的端口。每个RTOS端口还[具有其自己的网页](https://www.freertos.org/a00090.html)，该[网页](https://www.freertos.org/a00090.html)详细描述了可在其中找到该端口的演示应用程序的目录。

演示目录示例：

* 如果构建针对Infineon TriBoard硬件的TriCore GCC演示应用程序：

TriCore演示应用程序项目文件位于FreeRTOS / Demo / TriCore\_TC1782\_TriBoard\_GCC目录中。FreeRTOS / Demo目录（Common目录除外）中包含的所有其他子目录都可以忽略或删除。

* 如果构建针对RX62N RDK硬件的瑞萨RX6000 IAR演示应用程序：

IAR工作区文件位于FreeRTOS / Demo / RX600\_RX62N-RDK\_IAR目录中。FreeRTOS / Demo目录（Common目录除外）中包含的所有其他子目录都可以忽略或删除。

* 对于所有端口，依此类推...

所述的结构FreeRTOS的/演示目录下面被示出。

**FreeRTOS**

**|**

**+-演示**

**|**

**+ -Common所有演示使用的演示应用程序文件。**

**+ -Dir x演示应用程序为端口x构建文件**

**+ -Dir y演示应用程序为y端口构建文件**

### 创建自己的应用程序

**[**[**创建新的FreeRTOS应用程序**](https://www.freertos.org/Creating-a-new-FreeRTOS-project.html)**页面上提供了更多详细信息]**

提供了预先配置的演示应用程序，以确保项目已经存在，其中包含正确的RTOS内核源文件和正确的编译器选项集，因此可以以最少的用户工作量进行构建。因此，强烈建议通过修改现有的预配置演示应用程序来创建新应用程序。为此，可以先构建一个现有的演示应用程序以确保可以进行完整构建，然后用您自己的应用程序源文件增量替换FreeRTOS / Demo目录中项目中包含的文件。

# 定制 [[配置](https://www.freertos.org/a00109.html)]

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

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

这是典型的FreeRTOSConfig.h定义，后面是每个参数的说明：

**#ifndef FREERTOS\_CONFIG\_H**

**#define FREERTOS\_CONFIG\_H**

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

**your application. \*/**

**#include "something.h"**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 'config'参数

### configUSE\_PREEMPTION

设置为1以使用抢占式RTOS调度程序，或设置为0以使用协作式RTOS调度程序。

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

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

通用接口：

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

特定接口：

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

### configUSE\_TICKLESS\_IDLE

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

### configUSE\_IDLE\_HOOK

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

### configUSE\_MALLOC\_FAILED\_HOOK

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

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

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

**void vApplicationMallocFailedHook（void）;**

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

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

**void vApplicationDaemonTaskStartupHook（void）;**

### configUSE\_TICK\_HOOK

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

### configCPU\_CLOCK\_HZ

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

### configSYSTICK\_CLOCK\_HZ

仅用于ARM Cortex-M端口的可选参数。

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

### configTICK\_RATE\_HZ

RTOS滴答中断的频率。

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

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

### configMAX\_PRIORITIES

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

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

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

### configMINIMAL\_STACK\_SIZE

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

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

### configMAX\_TASK\_NAME\_LEN

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

### configUSE\_TRACE\_FACILITY

如果您希望包括其他数据结构和功能以帮助执行可视化和跟踪，请设置为1。

### configUSE\_STATS\_FORMATTING\_FUNCTIONS

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

### configUSE\_16\_BIT\_TICKS

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

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

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

### configIDLE\_SHOULD\_YIELD

此参数控制处于空闲优先级的任务的行为。它仅在以下情况下有效：

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

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

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



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

可以通过以下方式避免这种情况：

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

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

### configUSE\_TASK\_NOTIFICATIONS

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

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

当直接任务通知包含在构建中时，每个任务会占用8个字节的额外RAM。

### configTASK\_NOTIFICATION\_ARRAY\_ENTRIES

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

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

### configUSE\_MUTEXES

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

### configUSE\_RECURSIVE\_MUTEXES

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

### configUSE\_COUNTING\_SEMAPHORES

设置为1可在构建中包括计数信号量，或设置为0可在构建中忽略计数信号量。

### configUSE\_ALTERNATIVE\_API

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

### configCHECK\_FOR\_STACK\_OVERFLOW

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

### configQUEUE\_REGISTRY\_SIZE

队列注册表具有两个目的，这两个目的都与RTOS内核感知的调试相关联：

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

除非使用的是RTOS内核感知调试器，否则队列注册表没有任何用途。

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

### configUSE\_QUEUE\_SETS

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

### configUSE\_TIME\_SLICING

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

### configUSE\_NEWLIB\_REENTRANT

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

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

### configENABLE\_BACKWARD\_COMPATIBILITY

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

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

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

### configSTACK\_DEPTH\_TYPE

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

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

### configMESSAGE\_BUFFER\_LENGTH\_TYPE

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

### configSUPPORT\_STATIC\_ALLOCATION

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

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

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

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

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

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

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

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

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

**uint32\_t \*pulIdleTaskStackSize )**

**{**

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

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

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

**static StaticTask\_t xIdleTaskTCB;**

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

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

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

**\*ppxIdleTaskTCBBuffer = &xIdleTaskTCB;**

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

**\*ppxIdleTaskStackBuffer = uxIdleTaskStack;**

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

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

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

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

**}**

**/\*———————————————————–\*/**

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

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

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

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

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

**uint32\_t \*pulTimerTaskStackSize )**

**{**

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

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

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

**static StaticTask\_t xTimerTaskTCB;**

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

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

**task’s state will be stored. \*/**

**\*ppxTimerTaskTCBBuffer = &xTimerTaskTCB;**

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

**\*ppxTimerTaskStackBuffer = uxTimerTaskStack;**

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

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

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

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

**}**

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

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

**is set to 1.**

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

### configSUPPORT\_DYNAMIC\_ALLOCATION

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

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

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

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

### configTOTAL\_HEAP\_SIZE

FreeRTOS堆中可用的RAM总量。

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

### configAPPLICATION\_ALLOCATED\_HEAP

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

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

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

### configGENERATE\_RUN\_TIME\_STATS

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

### configUSE\_CO\_ROUTINES

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

### configMAX\_CO\_ROUTINE\_PRIORITIES

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

### configUSE\_TIMERS

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

### configTIMER\_TASK\_PRIORITY

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

### configTIMER\_QUEUE\_LENGTH

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

### configTIMER\_TASK\_STACK\_DEPTH

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

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

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

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

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

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

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

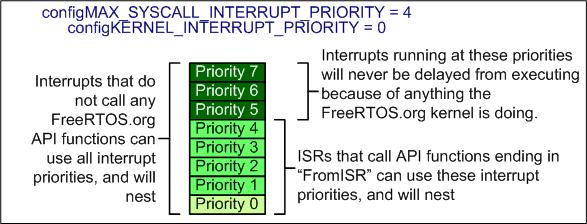
对于仅实现configKERNEL\_INTERRUPT\_PRIORITY的端口，  
configKERNEL\_INTERRUPT\_PRIORITY设置RTOS内核本身使用的中断优先级。调用API函数的中断也必须以此优先级执行。不调用API函数的中断可以以更高的优先级执行，因此不会因RTOS内核活动（在硬件本身的限制内）而延迟执行。

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

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

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

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

  
中断优先级配置示例

这些配置参数允许非常灵活的中断处理：

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

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

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

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

### configASSERT

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**{**

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

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

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

**debug\_printf( pcMessage );**

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

**portRESET\_PRIVILEGE( xRunningPrivileged );**

**}**

该技术只能在开发过程中使用，而不能在部署过程中使用，因为它会绕过内存保护。

### configTOTAL\_MPU\_REGIONS

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

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

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

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

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

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

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

## 包含参数

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

每个宏采用以下形式：

**INCLUDE\_FunctionName**

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

**#define INCLUDE\_vTaskDelete 1**

要从构建中排除vTaskDelete（），请使用：

**#define INCLUDE\_vTaskDelete 0**

# 静态与动态内存分配

### 介绍

V9.0.0之前的FreeRTOS版本从[特殊的FreeRTOS堆中](https://www.freertos.org/a00111.html)分配下面列出的RTOS对象使用的内存。FreeRTOS V9.0.0及更高版本使应用程序编写者可以自己提供内存，从而可以选择创建以下对象，而无需动态分配任何内存：

* 任务
* 软件计时器
* s列
* 活动组
* 二值信号量
* 计数信号量
* 递归信号量
* 互斥量

使用静态或动态内存分配是更可取的取决于应用程序以及应用程序编写者的偏好。两种方法都有优点和缺点，并且两种方法都可以在同一RTOS应用程序中使用。

[位于主要FreeRTOS的FreeRTOS / Source / WIN32-MSVC-Static-Allocation-Only 目录](https://www.freertos.org/freertos-static-allocation-demo.html)中的 简单Win32示例 演示了如何创建FreeRTOS应用程序而无需 在项目中包含任何[FreeRTOS堆](https://www.freertos.org/a00111.html)实现。

### 使用动态分配的RAM创建RTOS对象

动态创建RTOS对象的好处是更加简单，并有可能最大程度地减少应用程序的最大RAM使用量：

* 创建对象时，需要较少的功能参数。
* 内存分配在RTOS API函数内自动进行。
* 应用程序编写者不需要自己关心分配内存的问题。
* 如果删除了RTOS对象使用的RAM，则可以重新使用该对象，从而有可能减少应用程序的最大RAM占用空间。
* 提供RTOS API函数以返回有关堆使用情况的信息，从而可以优化堆大小。
* 可以选择使用的内存分配方案来最佳地适合应用程序，例如，对于安全性至关重要的应用程序来说，heap\_1.c为简化和确定性通常是必需的； heap\_4.c为碎片保护； heap\_5.c在多个RAM区域中拆分堆；或者应用程序编写者自己提供的一种分配方案。

如果[configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html#configSUPPORT_DYNAMIC_ALLOCATION)设置为1或未定义，以下API函数将 使用动态分配的RAM创建RTOS对象：

* [xTaskCreate（）](https://www.freertos.org/a00125.html)
* [xQueueCreate（）](https://www.freertos.org/a00116.html)
* [xTimerCreate（）](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)
* [xEventGroupCreate（）](https://www.freertos.org/xEventGroupCreate.html)
* [xSemaphoreCreateBinary（）](https://www.freertos.org/xSemaphoreCreateBinary.html)
* [xSemaphoreCreateCounting（）](https://www.freertos.org/CreateCounting.html)
* [xSemaphoreCreateMutex（）](https://www.freertos.org/CreateMutex.html)
* [xSemaphoreCreateRecursiveMutex（）](https://www.freertos.org/xSemaphoreCreateRecursiveMutex.html)

### 使用静态分配的RAM创建RTOS对象

使用静态分配的RAM创建RTOS对象的好处是为应用程序编写器提供了更多控制权：

* 可以将RTOS对象放置在特定的内存位置。
* 最大RAM占用空间可以在链接时而不是运行时确定。
* 应用程序编写者无需关心内存分配失败的适当处理。
* 它允许RTOS用于根本不允许任何动态内存分配的应用程序（尽管FreeRTOS包括可以克服大多数反对意见的分配方案）。

下列API函数（如果[configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html#configSUPPORT_STATIC_ALLOCATION)设置为1时可用） 允许使用应用程序[编写](https://www.freertos.org/a00110.html#configSUPPORT_STATIC_ALLOCATION)器提供的内存创建RTOS对象。为了提供内存，应用程序编写者只需要声明一个适当的对象类型的变量，然后将该变量的地址传递给RTOS API函数即可。提供了[StaticAllocation.c](https://sourceforge.net/p/freertos/code/HEAD/tree/trunk/FreeRTOS/Demo/Common/Minimal/StaticAllocation.c) 标准的演示/测试任务来演示如何使用这些功能：

* [xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)
* [xQueueCreateStatic（）](https://www.freertos.org/xQueueCreateStatic.html)
* [xTimerCreateStatic（）](https://www.freertos.org/xTimerCreateStatic.html)
* [xEventGroupCreateStatic（）](https://www.freertos.org/xEventGroupCreateStatic.html)
* [xSemaphoreCreateBinaryStatic（）](https://www.freertos.org/xSemaphoreCreateBinaryStatic.html)
* [xSemaphoreCreateCountingStatic（）](https://www.freertos.org/xSemaphoreCreateCountingStatic.html)
* [xSemaphoreCreateMutexStatic（）](https://www.freertos.org/xSemaphoreCreateMutexStatic.html)
* [xSemaphoreCreateRecursiveMutexStatic（）](https://www.freertos.org/xSemaphoreCreateRecursiveMutexStatic.html)

# 内存管理

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

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

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

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

**…**经常需要替代的内存分配实现。

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

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

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

## RTOS源代码下载中包含的内存分配实现

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

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

以下内容：

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

笔记：

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

### heap\_1.c

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

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

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

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

heap\_1实现：

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

### 堆\_2.c

 现在首选使用heap\_2，因为首选使用heap\_4。

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

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

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

此实现：

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

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

### heap\_3.c

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

此实现：

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

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

### heap\_4.c

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

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

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

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

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

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

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

**typedef struct xHeapStats**

**{**

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

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

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

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

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

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

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

**} HeapStats\_t;**

heap\_4：

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

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

### heap\_5.c

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

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

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

**typedef struct HeapRegion**

**{**

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

**uint8\_t \*pucStartAddress;**

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

**size\_t xSizeInBytes;**

**} HeapRegion\_t;**

**The HeapRegion\_t type definition**

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

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

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

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

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

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

**{**

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

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

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

**};**

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

**vPortDefineHeapRegions( xHeapRegions );**

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

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

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

# 堆栈使用情况和堆栈溢出检查

### 堆栈使用

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

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

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

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

**signed char \*pcTaskName );**

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

堆栈溢出检查会带来上下文切换开销，因此仅在开发或测试阶段才建议使用它。

### 堆栈溢出检测-方法1

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

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

### 堆栈溢出检测–方法2

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

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

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

# 创建一个新的FreeRTOS项目

## 介绍

FreeRTOS的设计简单易用：仅需要3个所有RTOS端口共有的源文件，以及一个特定于微控制器的源文件，并且其API的设计也非常简单直观。

FreeRTOS已移植到许多不同的微控制器体系结构和许多不同的编译器。每个官方端口都带有一个官方演示，该演示（至少在创建时）在未进行任何修改的硬件平台上编译和执行。

提供了演示项目，以确保新用户可以在最快的时间内以最少的麻烦开始使用FreeRTOS。

FreeRTOS支持的每种体系结构都用于许多不同的微控制器，这意味着FreeRTOS可以与数千种不同的微控制器零件号一起使用。将这个数字乘以受支持的编译器的数量，然后再乘以市场上不断增加的入门工具包和评估板的数量，很明显，尽管我们已尽最大努力，但我们只能提供官方演示与可能的组合中的一小部分完全匹配的项目。

**始终建议通过**[**从提供的预配置演示之一开始然后进行改编**](https://www.freertos.org/porting-a-freertos-demo-to-different-hardware.html)**来创建新的FreeRTOS项目 。这样做可以确保新项目包括所有必需的源文件和头文件，并安装必要的中断服务例程，而项目创建者无需付出任何努力。**

一些FreeRTOS用户还想知道如何通过改编现有项目的方式来创建FreeRTOS项目。下面记录了执行此操作的过程。

## FreeRTOS项目的剖析

FreeRTOS应用程序将像非RTOS应用程序一样启动并执行，直到调用[vTaskStartScheduler（）](https://www.freertos.org/a00132.html)为止。通常从应用程序的main（）函数调用vTaskStartScheduler（）。RTOS仅在调用vTaskStartScheduler（）之后才控制执行顺序。

它被**强烈建议**，以确保代码被所选择的目标尝试使用任何RTOS功能之前正确执行的（正确的启动代码，正确的接头配置等）。

### 源文件

FreeRTOS是作为标准C源文件提供的，它与项目中的所有其他C文件一起构建。FreeRTOS源文件以zip文件分发。该[RTOS源代码的组织](https://www.freertos.org/a00017.html)页面描述中的zip文件中的文件的结构。

至少，以下源文件必须包含在您的项目中：

* FreeRTOS /源/tasks.c
* FreeRTOS /源/queue.c
* FreeRTOS /源/ list.c
* FreeRTOS /源/可移植/ [编译器] / [体系结构] /port.c。
* FreeRTOS / Source / portable / MemMang / heap\_x.c [其中'x'是1，2，3，4或5](https://www.freertos.org/a00111.html)。

如果包含port.c文件的目录也包含汇编语言文件，那么也必须使用汇编语言文件。

### 可选的源文件

如果需要[软件计时器](https://www.freertos.org/RTOS-software-timer.html)功能，则将FreeRTOS / Source / timers.c添加到项目中。

如果需要[事件组](https://www.freertos.org/FreeRTOS-Event-Groups.html)功能，则将FreeRTOS / Source / event\_groups.c添加到项目中。

如果您需要[流缓冲区或消息缓冲区](https://www.freertos.org/RTOS-stream-message-buffers.html)功能，则将FreeRTOS / Source / stream\_buffer.c添加到您的项目中。

如果需要协程功能，则将FreeRTOS / Source / croutine.c添加到您的项目中（请注意，不推荐使用协程，不建议将其用于新设计）。

### 头文件

以下目录必须位于编译器的include路径中（必须告诉编译器在这些目录中搜索头文件）：

* FreeRTOS /来源/包含
* FreeRTOS /源/可移植/ [编译器] / [体系结构]。
* 无论哪个目录包含要使用的FreeRTOSConfig.h文件，请参阅下面的“配置文件”段落。

根据端口的不同，汇编程序的include路径中可能还必须包含相同的目录。

### 配置文件

每个项目还需要一个名为[FreeRTOSConfig.h](https://www.freertos.org/a00110.html)的文件。FreeRTOSConfig.h使RTOS内核适合正在构建的应用程序。因此，它特定于应用程序而不是RTOS，并且应位于应用程序目录中，而不是RTOS内核源代码目录之一中。

如果您的项目中包含heap\_1，heap\_2，heap\_4或heap\_5，则FreeRTOSConfig.h定义configTOTAL\_HEAP\_SIZE将确定FreeRTOS堆的尺寸。如果configTOTAL\_HEAP\_SIZE设置得太高，您的应用程序将不会链接。

FreeRTOSConfig.h定义configMINIMAL\_STACK\_SIZE设置空闲任务使用的堆栈大小。如果configMINIMAL\_STACK\_SIZE设置得太低，则空闲任务将产生堆栈溢出。建议从为相同微控制器架构提供的官方FreeRTOS演示中复制configMINIMAL\_STACK\_SIZE设置。FreeRTOS演示项目存储在FreeRTOS / Demo目录的子目录中。请注意，某些演示项目较旧，因此不包含所有可用的配置选项。

### 中断向量

**[Cortex-M用户：有关安装中断处理**[**程序的**](https://www.freertos.org/FAQHelp.html)**信息，请参见“**[**我创建的应用程序已编译，但无法运行**](https://www.freertos.org/FAQHelp.html)**”常见问题解答。**

每个RTOS端口都使用一个计时器来生成定期的滴答中断。许多端口使用其他中断来管理上下文切换。RTOS端口所需的中断由提供的RTOS端口源文件提供服务。

用于安装RTOS端口提供的中断处理程序的方法取决于所使用的端口和编译器。请参考所使用的​​端口的官方演示应用程序，并在需要时进行复制。另请参阅为官方演示应用程序[提供](https://www.freertos.org/a00090.html)的 [文档页面](https://www.freertos.org/a00090.html)。

# FreeRTOS内核辅助文档

* [**空闲任务**](https://www.freertos.org/RTOS-idle-task.html)
* [**挂钩（回调）**](https://www.freertos.org/a00016.html)
* [**Windows模拟器**](https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html)
* [**内存保护单元支持**](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html)
* [**线程本地存储指针**](https://www.freertos.org/thread-local-storage-pointers.html)
* [**阻止多个对象**](https://www.freertos.org/Pend-on-multiple-rtos-objects.html)
* [**延迟中断处理**](https://www.freertos.org/deferred_interrupt_processing.html)
* [**低功耗支持**](https://www.freertos.org/low-power-tickless-rtos.html)
* [**跟踪功能**](https://www.freertos.org/rtos-trace-macros.html)
* [**运行时间统计**](https://www.freertos.org/rtos-run-time-stats.html)
* [**移植指南**](https://www.freertos.org/FreeRTOS-porting-guide.html)
* [**Posix / Linux模拟器**](https://www.freertos.org/FreeRTOS-simulator-for-Linux.html)
* [**旧版跟踪设施**](https://www.freertos.org/a00086.html)
* [**FreeRTOS概念**](https://www.freertos.org/implementation/main.html)

# 任务 [[有关任务的更多信息...](https://www.freertos.org/a00015.html) ]

该[FreeRTOS的教程书籍](https://www.freertos.org/Documentation/RTOS_book.html) 提供关于任务和他们的行为的其他详细信息。

### 空闲任务

启动RTOS计划程序时，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。它以最低的[优先级](https://www.freertos.org/RTOS-task-priority.html)创建，以确保在就绪[状态下](https://www.freertos.org/RTOS-task-states.html)具有较高优先级的应用程序任务时，不占用任何CPU时间。

空闲任务负责释放[RTOS分配](https://www.freertos.org/a00111.html)给已删除任务的内存。因此，对于使用[vTaskDelete（）](https://www.freertos.org/a00126.html)函数的应用程序来说，重要的是确保空闲任务不会因处理时间而短缺。空闲任务没有其他活动功能，因此在所有其他条件下都可以合理地饿死微控制器的时间。

应用程序任务可以共享空闲任务优先级（tskIDLE\_PRIORITY）。有关如何配置此行为的信息，请参阅configIDLE\_SHOULD\_YIELD[配置参数](https://www.freertos.org/a00110.html)。

### 空闲任务挂钩

空闲任务挂钩是在空闲任务的每个循环期间调用的函数。如果希望应用程序功能以空闲优先级运行，则有两个选项：

1. 在空闲任务挂钩中实现功能。

必须始终至少准备好一项任务。因此，挂钩函数必须不要调用任何可能导致空闲任务阻塞的API函数（例如vTaskDelay（）或具有阻塞时间的[队列或信号量](https://www.freertos.org/Inter-Task-Communication.html)函数）。协例程可以在挂钩函数中阻塞是可以的。

1. 创建一个空闲优先级任务以实现功能。

这是一种更灵活的解决方案，但具有更高的RAM使用开销。

有关使用空闲挂钩的更多信息，请参见[嵌入式软件应用程序设计](https://www.freertos.org/tutorial/index.html)部分。

要创建一个空闲钩子：

1. 在[FreeRTOSConfig.h](https://www.freertos.org/a00110.html)中将configUSE\_IDLE\_HOOK设置为1 。
2. 定义一个具有以下名称和原型的函数：

vApplicationIdleHook（void）;

通常使用空闲挂钩功能将微控制器CPU置于省电模式。

# 挂钩功能

### 空闲挂钩功能

空闲任务可以选择调用应用程序定义的挂钩（或回调）函数–空闲挂钩。空闲任务以最低的优先级运行，因此仅当没有更高优先级的任务能够运行时，才会执行这种空闲挂钩函数。这使空闲挂钩功能成为使处理器进入低功耗状态的理想场所–每当不执行任何处理时，便自动节省了电源。

只有在FreeRTOSConfig.h中将configUSE\_IDLE\_HOOK设置为1时，才会调用空闲挂钩。设置此选项后，应用程序必须为hook函数提供以下原型：

**vApplicationIdleHook（void）;**

只要空闲任务正在运行，就会重复调用空闲钩子。 至关重要的是，空闲挂钩函数不会调用任何可能导致其阻塞的API函数。 另外，如果应用程序使用vTaskDelete（）API函数，则必须允许空闲任务挂钩定期返回（这是因为空闲任务负责清理由RTOS内核分配给任务的资源）已被删除）。

### 勾号功能

滴答中断可以选择调用应用程序定义的挂钩（或回调）函数–滴答钩。勾钩提供了实现计时器功能的便利位置。

只有在FreeRTOSConfig.h中将configUSE\_TICK\_HOOK设置为1时，刻度钩才会被调用。设置此选项后，应用程序必须为hook函数提供以下原型：

**vApplicationTickHook（void）;**

vApplicationTickHook（）是从ISR内执行的，因此必须非常短，不占用太多堆栈并且不调用任何以“ FromISR”或“ FROM\_ISR”结尾的API函数。

有关如何使用刻度钩的示例，请参见[演示应用程序文件crhook.c](https://www.freertos.org/a00013.html#crhook)。

### Malloc失败的挂钩函数

由[heap\_1.c，heap\_2.c，heap\_3.c，heap\_4.c和heap\_5.c](https://www.freertos.org/a00111.html)实现的内存分配方案可以选择包含malloc（）故障挂钩（或回调）函数，可以将其配置为在pvPortMalloc（）时被调用曾经返回NULL。

定义malloc（）故障挂钩将有助于识别由堆内存不足引起的问题，尤其是在API函数内对pvPortMalloc（）的调用失败时。

只有在FreeRTOSConfig.h中将configUSE\_MALLOC\_FAILED\_HOOK设置为1时，才会调用malloc失败的挂钩。设置此选项后，应用程序必须为hook函数提供以下原型：

**vApplicationMallocFailedHook（void）;**

### 堆栈溢出挂钩功能

有关详细信息，请参见[堆栈溢出保护](https://www.freertos.org/Stacks-and-stack-overflow-checking.html)页面。

### 守护程序任务启动挂钩

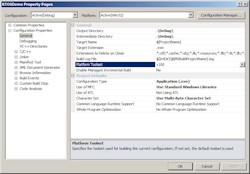
RTOS守护程序任务与[计时器服务任务](https://www.freertos.org/RTOS-software-timer-service-daemon-task.html)相同。有时将其称为守护程序任务，因为该任务现在不仅用于维护计时器，还用于更多任务。

如果在FreeRTOSConfig.h中将configUSE\_DAEMON\_TASK\_STARTUP\_HOOK设置为1，则将在Daemon Task首次开始执行时立即调用Daemon Task Startup Hook。如果应用程序包含可从调度程序启动后执行的初始化代码中受益的初始化代码，则这将很有用，该代码允许初始化代码利用RTOS功能。

如果configUSE\_DAEMON\_TASK\_STARTUP\_HOOK设置为1，则应用程序编写器必须使用以下名称提供原型的Daemon Task启动挂钩函数的实现。

**vApplicationDaemonTaskStartupHook（void）;**

# 用于Visual Studio或Eclipse和MingW的 FreeRTOS Windows端口 [ [RTOS端口](https://www.freertos.org/a00090.html)]

[](https://www.freertos.org/fr-content-src/uploads/2018/07/platform_toolset.jpg)

**注意！**FreeRTOS发行版中的Visual Studio项目是在不同的时间创建的，因此使用[Visual Studio for C / C ++工具的](https://visualstudio.microsoft.com/vs/community/)各种不同的 [免费版本](https://visualstudio.microsoft.com/vs/community/)。通常，您不需要使用与创建项目时完全相同的Visual Studio版本，并且如果版本不匹配，Visual Studio将提供有关重新定位项目的说明。但是，在存在向后兼容性问题的情况下，可能有必要将Visual Studio版本更新为最新版本。

### 前言–适用于初学者

如果您是FreeRTOS的 [新手](https://www.freertos.org/simple-freertos-demos.html) ，那么在查看该页面之前，建议先阅读“[简单FreeRTOS项目入门”](https://www.freertos.org/simple-freertos-demos.html)文档（该文档还介绍了如何使用FreeRTOS Windows端口）。

### 介绍

本页显示了适用于FreeRTOS的Windows端口层，该层已使用[Visual Studio Community Edition](https://visualstudio.microsoft.com/vs/express/) 和[Eclipse IDE（针对](http://www.eclipse.org/downloads/)基于[MingW](http://www.mingw.org/) GCC的编译器的[C和C ++开发人员）](http://www.eclipse.org/downloads/) 进行了开发和测试。为两个工具链都提供了演示项目。这两个工具链也是免费的，尽管如果要将Visual Studio用于评估目的以外的其他任何用途，则必须注册Visual Studio。

该端口是在运行32位Windows XP的双核Intel处理器上开发的，现在在运行64位Windows 10的四核Intel处理器上进行维护（尽管该项目创建的是32位二进制文​​件）。

### *使用Windows FreeRTOS端口的注意事项*

*使用此RTOS端口之前，请阅读以下所有内容。*

1. [工作原理](https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html#PrincipalsOfOperation)
2. [使用端口之前的注意事项](https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html#NotesForUsers)
3. [源代码组织](https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html#SourceCodeOrg)
4. [使用Eclipse / MingW（GCC）演示](https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html#UsingEclipseMingW)
5. [演示应用](https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html#DemoApp)
6. [定义和使用模拟中断服务程序](https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html#SimulatedISRs)

## 工作原理

### 运行任务的线程

Windows端口层为FreeRTOS应用程序创建的每个FreeRTOS任务创建一个低优先级的Windows线程。然后，所有低优先级Windows线程都保持挂起状态，而不是运行由FreeRTOS调度程序选择为处于“运行”状态的FreeRTOS任务的Windows线程。这样，FreeRTOS调度程序将根据其调度策略选择运行哪个低优先级Windows线程。所有其他低优先级Windows线程无法运行，因为它们已挂起。

当任务进入和退出运行状态时，在微控制器上运行的FreeRTOS端口必须执行复杂的上下文切换，以保存和恢复微控制器上下文（寄存器等）。相反，Windows模拟器层仅需暂停和恢复Windows线程，因为它们代表的任务进入和离开“运行”状态。实际的上下文切换留给Windows。

### 模拟滴答中断

滴答中断的生成由高优先级Windows线程模拟，该线程将定期抢占正在运行任务的低优先级线程。Windows系统时钟限制了可达到的报价速度，在正常的FreeRTOS术语中，该速度很慢且精度非常低。因此，不可能获得真实的实时行为。

### 模拟中断处理

模拟中断处理由更高优先级的第二个Windows线程执行，由于其优先级，该线程还可以抢占正在运行FreeRTOS任务的低优先级线程。模拟中断处理的线程将等待，直到系统中的另一个线程通知它有一个未决的中断。例如，模拟滴答中断的产生的线程会设置一个中断挂起位，然后通知模拟正在处理的中断的Windows线程将一个中断挂起。然后，模拟中断处理线程将执行并查看所有可能的中断挂起位-执行任何模拟中断处理并根据需要清除中断挂起位。

## 使用模拟器前的注意事项

### 嵌入式工程师进行Windows编程

在使用提供的Windows项目之前，请注意，我（Windows模拟器的身份）是嵌入式程序员，不是Windows程序员。该实施可能是幼稚的。 我们将不胜感激地收到那些对Windows编程更了解的人士对当前实现提供的任何[反馈](https://www.freertos.org/RTOS-contact-and-support.html)。

### 在Windows上删除任务

删除FreeRTOS任务后，Windows端口将终止负责运行该任务的线程。但是，在Windows下，终止另一个线程中的线程不会导致终止线程正在使用的资源返回到系统。这意味着在运行时，可以调用FreeRTOS vTaskDelete（）API函数的次数受到限制。该限制非常高（成千上万），但是确实阻止了标准演示[“死亡”](https://www.freertos.org/a00102.html#death) 任务在演示任务开始报告错误之前无限期执行。

### 在主机Windows计算机的CPU上加载

运行FreeRTOS应用程序时，主机Windows计算机的CPU上的负载将非常高。由于只使用了低优先级的线程，因此响应性不会受到太大影响，但是CPU核心温度将升高，并且CPU冷却风扇将相应地响应。

如果您以某种方式担心计算机处理高温的能力，那么我建议您使用实用程序，该实用程序提供有关当前CPU核心温度以及当前温度与最高温度的接近程度的信息。您的特定CPU的温度等级。我个人 为此目的使用了免费的 [Core Temp](http://www.alcpu.com/CoreTemp/)实用程序。

## 源代码组织

### Eclipse和MingW（GCC）

FreeRTOS模拟器演示应用程序的Eclipse项目位于FreeRTOS主下载的FreeRTOS / Demo / WIN32-MingW目录中。需要将其导入到Eclipse工作区中才能构建项目。

### 视觉工作室

用于FreeRTOS模拟器演示应用程序的Visual Studio解决方案称为WIN32.sln，位于FreeRTOS 主下载的FreeRTOS / Demo / WIN32-MSVN目录中。

## 使用Eclipse和MingW（GCC）演示

### 获取编译器

该[MingW平台](http://www.mingw.org/)编译工具不包括作为Eclipse发行的一部分，必须单独下载。

### 将FreeRTOS模拟器项目导入Eclipse工作区

要将FreeRTOS模拟器项目导入Eclipse：

1. 启动Eclipse IDE，然后转到Eclipse Workbench。
2. 从Eclipse的“文件”菜单中选择“导入”。将出现一个对话框。
3. 在对话框中，选择“常规| 现有项目进入工作区”。将出现另一个对话框，使您可以导航到并选择一个根目录。
4. 选择FreeRTOS / Demo / WIN32-MingW 作为目录–这将显示一个名为RTOSDemo的项目，该项目应该被导入。

## 演示应用

### 功能性

常量mainCREATE\_SIMPLE\_BLINKY\_DEMO\_ONLY（在main.c顶部已#defined定义）用于在简单的Blinky风格演示和更全面的测试和演示应用程序之间进行切换，如以下两节所述。

### mainCREATE\_SIMPLE\_BLINKY\_DEMO\_ONLY设置为1时的功能

如果mainCREATE\_SIMPLE\_BLINKY\_DEMO\_ONLY设置为1，则main（）将调用main\_blinky（），该方法在main\_blinky.c中实现。

main\_blinky（）创建了一个非常简单的演示，其中包含两个任务和一个队列。一个任务通过队列将值100重复发送给另一任务。接收任务每次接收到队列中的值时都会打印出一条消息。

### mainCREATE\_SIMPLE\_BLINKY\_DEMO\_ONLY设置为0时的功能

如果mainCREATE\_SIMPLE\_BLINKY\_DEMO\_ONLY设置为0，则main（）将调用main\_full（），该方法在main\_full.c中实现。

main\_full（）创建的演示非常全面。它创建的任务主要由[标准演示任务组成](https://www.freertos.org/a00102.html)–除了测试端口并演示如何使用FreeRTOS API之外，它们不执行任何特定功能。

### main\_full（）创建的“检查”任务

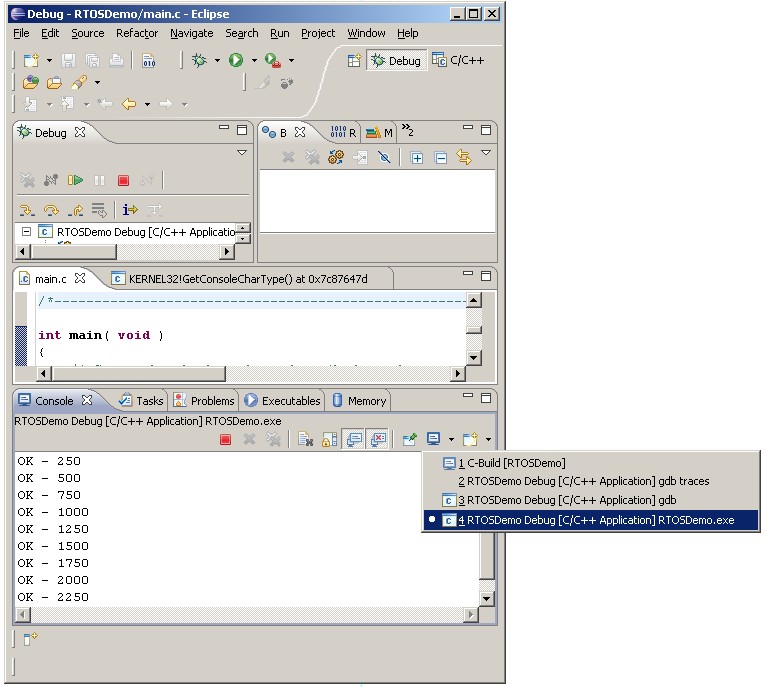
完整的演示除了标准演示任务之外，还会创建一个“检查”任务。这仅每（模拟）五秒钟执行一次，但是具有最高优先级以确保获得处理时间。其主要功能是检查所有标准演示任务是否仍在运行。

检查任务维护一个状态字符串，每次执行时都会输出到控制台。如果所有标准演示任务都正常运行，则该字符串将打印出“ OK”和当前的滴答计数。如果检测到错误，则该字符串将打印出一条消息，指出该错误是在哪个任务中报告的。

### 查看控制台输出

eclipse项目会将字符串输出到集成控制台。要查看这些字符串，必须使用可从小的计算机监视器图标速度按钮访问的下拉列表选择“ RTOSDemo.exe”控制台-如下图所示。

Visual Studio控制台输出将出现在命令提示符窗口中。

  
在Eclipse调试会话期间选择“ RTOSDemo.exe”控制台

## 定义和使用模拟中断服务程序

### 定义模拟中断服务程序的处理程序

中断服务例程必须具有以下原型：

**无符号长ulInterruptName（void）;**

其中“ ulInterruptName”可以是任何适当的函数名称。

如果执行例程应导致上下文切换，则中断函数必须返回pdTRUE。否则，中断函数应返回pdFALSE。

### 安装模拟中断服务程序的处理程序

可以使用在Win32端口层中定义的vPortSetInterruptHandler（）函数来安装模拟中断服务例程的处理程序 。它具有如下所示的原型：

**void vPortSetInterruptHandler（unsigned long ulInterruptNumber，unsigned long（\* pvHandler）（void））;**

ulInterruptNumber的值必须在3到31之间（包括3和31），并且在应用程序内必须是唯一的（意味着在任何应用程序中总共可以定义29个模拟中断）。模拟器本身使用数字0到2（含0和2）。

pvHandler应该指向要安装的中断号的处理函数。

### 触发模拟中断服务程序

可以将中断设置为挂起，并在适当时通过调用vPortGenerateSimulatedInterrupt（）函数来执行，该 函数也定义为Win32端口层的一部分。它具有如下所示的原型：

**void vPortGenerateSimulatedInterrupt（unsigned long ulInterruptNumber）;**

ulInterruptNumber是要设置为挂起的中断的编号，它对应于vPortSetInterruptHandler（）的ulInterruptNumber参数。

### 安装和触发中断的示例

模拟器本身使用三个中断，一个中断用于任务产生，一个中断用于模拟滴答，一个中断用于终止正在执行FreeRTOS任务的Windows线程，此任务已被删除。作为一个简单的示例，下面显示了yield中断的代码。

中断函数除了请求上下文切换外不执行其他操作，因此仅返回pdTRUE。它使用以下代码定义：

**静态无符号长prvProcessYieldInterrupt（void）**

**{**

**/ \*这里没有任何处理，此中断仅用于导致**

**上下文切换，因此它仅返回pdTRUE。\* /**

**返回pdTRUE;**

**}**

然后使用以下调用安装模拟的中断处理程序函数，其中portINTERRUPT\_YIELD定义为2：

**vPortSetInterruptHandler（portINTERRUPT\_YIELD，prvProcessYieldInterrupt）;**

每当调用taskYIELD（）/ portYIELD（）时，都应执行此中断，因此portYIELD（）的Win32端口版本定义为：

**＃定义portYIELD（）vPortGenerateSimulatedInterrupt（portINTERRUPT\_YIELD）**

# 内存保护单元（MPU）支持

|  |
| --- |
| [**有关将FreeRTOS-MPU项目更新到FreeRTOS V10.4.0的信息，请参见“升级”部分。**](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#upgrading-to-FreeRTOS-10.4.0) |

在本页面：

* 介绍
  + [MPU的成本和收益](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#mpu_benefits)
  + [MPU策略](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#mpu_strategies)
  + [例子](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#mpu_examples)
  + [将MPU项目升级到FreeRTOS V10.4.0](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#upgrading-to-FreeRTOS-10.4.0)
* FreeRTOS-MPU细节
  + [FreeRTOS-MPU功能](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#mpu_features)
  + [创建任务](https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html#creating_tasks)

## 介绍

FreeRTOS在ARMv7-M（Cortex-M3，Cortex-M4和Cortex-M7微控制器）和ARMv8-M（Cortex-M23和Cortex-M33微控制器）内核上提供了官方的内存保护单元（MPU）支持：

* 有两个用于ARMv7-M内核的FreeRTOS端口，一个包含MPU支持，一个不包含。
* 对于ARMv8-M内核，只有一个FreeRTOS端口，因为[MPU支持是一种编译时选项](https://www.freertos.org/2020/04/using-freertos-on-armv8-m-microcontrollers.html)。

第三方维护通往其他微控制器内核的MPU端口。 

#### MPU的成本和收益

FreeRTOS MPU端口通过首先使任务能够以特权或非特权模式运行，其次限制了对诸如RAM，可执行代码，外围设备和内存之类资源的访问，从而使微控制器应用程序更强大，更安全，从而超出了任务堆栈的限制。例如，防止代码从RAM中执行就可以带来巨大的好处，因为这样做可以防止许多攻击媒介，例如缓冲区溢出漏洞或加载到RAM中的恶意代码的执行。

使用MPU必然会使应用程序设计更加复杂，因为首先必须确定MPU的内存区域限制并将其描述给RTOS，其次，MPU会限制应用程序任务可以执行和不能执行的操作。

#### MPU策略

创建将每个任务限制在其自己的内存区域中的应用程序可能是最安全的，但设计和实现也是最复杂的。通常，最好使用MPU创建一个伪进程和线程模型-允许线程组共享一个内存空间。例如，创建一个可供信任的第一方代码访问的存储空间，以及一个仅不受信任的第三方代码访问的存储空间。

#### 其他例子

LPC17xx版的FreeRTOS电子书[包含一章关于FreeRTOS-MPU的使用](https://www.freertos.org/MPU_Chapter.pdf)，尽管其中包含的信息有些过时。

将[在ARMv8-M微控制器使用FreeRTOS的](https://www.freertos.org/2020/04/using-freertos-on-armv8-m-microcontrollers.html)博客中提到了如何在ARMv8-M微控制器采用MPU。

FreeRTOS / Demo / CORTEX\_MPU\_Simulator\_Keil\_GCC目录中 的演示项目使用Keil uVision构建和模拟GCC项目，提供了一个不需要任何特定硬件的有效示例。其他FreeRTOS-MPU示例项目包括[Nuvoton NuMaker-PFM-M2351演示](https://www.freertos.org/RTOS-Cortex-M23-NuMaker-PFM-M2351-Keil.html) 和[NXP LPCXpresso55S69演示](https://www.freertos.org/RTOS-Cortex-M33-LPC55S69-MCUXpresso-GCC.html)演示等。

[在FreeRTOS V9.0.0发行之前，位于FreeRTOS / Demo / CORTEX\_MPU\_LPC1768\_GCC\_RedSuite和FreeRTOS / Demo / CORTEX\_MPU\_LM3Sxxxx\_Rowley目录中的FreeRTOS-MPU演示项目已退役]

#### 升级信息

FreeRTOS MPU端口已更新，以响应最终用户的反馈。本节描述了升级到FreeRTOS V10.4.0或更高版本，以及在此之前升级到FreeRTOS V10.3.0或更高版本所必需的更改。

**FreeRTOS版本10.4.0中的更改：**

* FreeRTOS V10.4.0引入了一个新的变量\_\_privileged\_functions\_start\_\_ 来指示特权代码的开始位置。它需要以与先前存在的链接器变量（例如[\_\_privileged\_functions\_end\_\_）](https://www.freertos.org/MPU_Chapter.pdf)相同的方式从链接器脚本中导出。

如果您在较早创建的项目中遇到未解析符号\_\_privileged\_functions\_start\_\_的链接器错误，则需要导出变量\_\_privileged\_functions\_start\_\_，其值等于\_\_FLASH\_segment\_start\_\_。

* 现在，堆内存位于特权部分中，因此，未特权的任务无法调用[pvPortMalloc（）](https://www.freertos.org/a00111.html)或[vPortFree（）](https://www.freertos.org/a00111.html)。
* [xTaskCreate（）](https://www.freertos.org/a00125.html)不再可用于创建非特权任务。请改用[xTaskCreateRestricted（）](https://www.freertos.org/xTaskCreateRestricted.html)。
* 现在，用于ARM Cortex-M4微控制器的FreeRTOS-MPU端口支持具有16个MPU区域的微控制器。要使用16个区域 ，请在[FreeRTOSConfig.h](https://www.freertos.org/a00110.html)中将configTOTAL\_MPU\_REGIONS设置为16 。
* 通过在[FreeRTOSConfig.h中](https://www.freertos.org/a00110.html)分别定义configTEX\_S\_C\_B\_FLASH和configTEX\_S\_C\_B\_FLASH和configTEX\_S\_C\_B\_SRAM，应用程序编写者现在可以覆盖TEX的默认值，覆盖Flash和RAM的MPU区域的Shareable（S），Cacheable（C）和Bufferable（B）位 。

**FreeRTOS版本10.3.0中的更改：**

* 现在可以防止源自内核代码外部的任何特权升级（输入中断后由硬件本身执行的升级除外）。为此，在[FreeRTOSConfig.h](https://www.freertos.org/a00110.html)中将configENFORCE\_SYSTEM\_CALLS\_FROM\_KERNEL\_ONLY设置 为1 ，并将链接器变量[\_\_syscalls\_flash\_start\_\_](https://github.com/FreeRTOS/FreeRTOS/blob/V10.3.1/FreeRTOS/Demo/CORTEX_MPU_STM32L4_Discovery_GCC_IAR_Keil/Projects/GCC/STM32L475VGTX_FLASH.ld#L113)和 [\_\_syscalls\_flash\_end\_\_分别](https://github.com/FreeRTOS/FreeRTOS/blob/V10.3.1/FreeRTOS/Demo/CORTEX_MPU_STM32L4_Discovery_GCC_IAR_Keil/Projects/GCC/STM32L475VGTX_FLASH.ld#L115)定义为系统调用内存的开始和结束地址。

## FreeRTOS-MPU细节

#### FreeRTOS-MPU功能

* 与标准ARM Cortex-M3和Cortex-M4F端口兼容。
* 可以创建任务以在特权模式或非特权模式下运行。非特权任务只能访问它们自己的堆栈以及最多三个用户可定义的内存区域（每个任务三个）。用户定义的内存区域在创建任务时分配给任务，并且可以根据需要在运行时重新配置。
* 用户可定义的存储区域可以单独设置参数。例如，可以将某些区域设置为只读，而将其他区域设置为不可执行（在ARM术语中，从不执行，或仅执行XN），等等。
* 没有特权的任务之间没有共享数据内存，但是没有特权的任务可以使用标准队列和信号量机制将消息彼此传递。可以使用用户可定义的内存区域来显式创建共享内存区域，但是不建议这样做。
* 特权模式任务可以将自己设置为非特权模式，但是一旦进入非特权模式，就无法将其自身设置回特权模式。
* FreeRTOS API位于Flash的区域中，该区域仅在微控制器处于特权模式时才能访问（调用API函数会导致临时切换到特权模式）。
* 由RTOS内核维护的数据（FreeRTOS源文件专用的所有非堆栈数据）位于RAM区域中，该区域仅在微控制器处于特权模式时才能访问。
* 仅在微控制器处于特权模式时才能访问系统外围设备。标准外围设备（UART等）可通过任何代码访问，但可以使用用户可定义的存储区进行显式保护。

#### 创建任务

FreeRTOS-MPU端口可以具有两种类型的任务：

1. **特权任务：**特权任务可以访问整个内存映射。可以使用[xTaskCreate（）](https://www.freertos.org/a00125.html) 或[xTaskCreateRestricted（）](https://www.freertos.org/xTaskCreateRestricted.html) API函数创建特权任务。
2. **非特权任务：**非特权任务只能访问其堆栈。此外，它可以被授予访问多达三个用户可定义的内存区域（每个任务三个）的权限。非特权任务只能使用[xTaskCreateRestricted（）](https://www.freertos.org/xTaskCreateRestricted.html) API创建 。 请注意，不得使用xTaskCreate（）API创建非特权任务。

如果任务要使用MPU，则必须提供以下附加信息：

* 任务堆栈的地址。
* 最多三个用户定义的内存区域的开始，大小和访问参数。

因此，创建任务所需的参数总数很大。为了使创建MPU感知任务更容易，FreeRTOS-MPU使用了一个名为[xTaskCreateRestricted（）](https://www.freertos.org/xTaskCreateRestricted.html)的API函数 。这允许在一个const结构中定义除一个参数外的所有参数，并将该结构作为一个参数传递（通过引用）到xTaskCreateRestricted（）中。

特权模式任务可以调用 [portSWITCH\_TO\_USER\_MODE（）](https://www.freertos.org/portSWITCH_TO_USER_MODE.html)将其自身设置[为非](https://www.freertos.org/portSWITCH_TO_USER_MODE.html)特权模式。在非特权模式下运行的任务无法将自己设置为特权模式。

可以使用[vTaskAllocateMPURegions（）](https://www.freertos.org/vTaskAllocateMPURegions.html)更改分配给任务的内存区域 。有关更多信息，请参见xTaskCreateRestricted（）和vTaskAllocateMPURegions（）API文档。

# 线程本地存储指针

### 介绍

线程本地存储（或TLS）允许应用程序编写器将值存储在任务的控制块内，从而使值特定于任务本身（本地），并允许每个任务具有其自己的唯一值。

线程本地存储最常用于存储单线程程序否则将存储在全局变量中的值。例如，许多库都包含一个名为errno的全局变量。如果库函数将错误条件返回给调用函数，则调用函数可以检查errno值以确定错误是什么。在单线程应用程序中，将errno声明为全局变量就足够了，但是在多线程应用程序中，每个线程（任务）必须具有自己唯一的errno值-否则一个任务可能会读取打算用于另一任务的errno值。

### 线程本地存储指针

FreeRTOS通过使用线程本地存储指针为应用程序编写者提供了灵活的线程本地存储机制。

所述[configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS](https://www.freertos.org/a00110.html#configNUM_THREAD_LOCAL_STORAGE_POINTERS) 编译时间的配置恒定尺寸的每个任务阵列空指针的（无效\*）。所述 [vTaskSetThreadLocalStoragePointer（）](https://www.freertos.org/vTaskSetThreadLocalStoragePointer.html) API函数是用来设置空隙的指针数组中的值，并且 [pvTaskGetThreadLocalStoragePointer（）](https://www.freertos.org/pvTaskGetThreadLocalStoragePointer.html) API函数是用来读取从空隙的指针数组的值。

### 线程局部整数

大小小于或等于void指针大小的值可以直接存储在线程本地存储指针数组中。例如，如果sizeof（void \*）为4，则可以使用简单的强制类型转换将32位值存储在void指针变量中，以避免编译器警告。但是，如果sizeof（void \*）为2，则只能直接存储16位值。

**uint32\_t ulVariable;**

**/\* Write the 32-bit 0x12345678 value directly into index 1 of the thread**

**local storage array. Passing NULL as the task handle has the effect of writing**

**to the calling task’s thread local storage array. \*/**

**vTaskSetThreadLocalStoragePointer( NULL, /\* Task handle. \*/**

**1, /\* Index into the array. \*/**

**( void \* ) 0x12345678 );**

**/\* Store the value of the 32-bit variable ulVariable to index 0 of the calling**

**task’s thread local storage array. \*/**

**ulVariable = ERROR\_CODE;**

**vTaskSetThreadLocalStoragePointer( NULL, /\* Task handle. \*/**

**0, /\* Index into the array. \*/**

**( void \* ) ulVariable );**

**/\* Read the value stored in index 5 of the calling task’s thread local storage**

**array into ulVariable. \*/**

**ulVariable = ( uint32\_t ) pvTaskGetThreadLocalStoragePointer( NULL, 5 );**

**Storing and retrieving 32-bit values directly from an index in the thread local storage array**

### 线程局部结构

前面的示例将值直接存储到线程本地存储数组中。下一个示例演示如何使用数组中的值作为内存中其他地方存在的结构的指针。

**typedef struct**

**{**

**uint32\_t ulValue1;**

**uint32\_t ulValue2;**

**} xExampleStruct;**

**xExampleStruct \*pxStruct;**

**/\* Create a structure for use by this task. \*/**

**pxStruct = pvPortMalloc( sizeof( xExampleStruct ) );**

**/\* Set the structure members. \*/**

**pxStruct->ulValue1 = 0;**

**pxStruct->ulValue2 = 1;**

**/\* Store a pointer to the structure in index 0 of the calling task’s thread**

**local storage array. \*/**

**vTaskSetThreadLocalStoragePointer( NULL, /\* Task handle. \*/**

**0, /\* Index into the array. \*/**

**( void \* ) pxStruct );**

**/\* Locate the structure used by the calling task by reading its location from**

**index 0 of the calling task’s thread local storage array. \*/**

**pxStruct = ( xExampleStruct \* ) pvTaskGetThreadLocalStoragePointer( NULL, 0 );**

**Storing a pointer to a structure in the calling task’s thread local storage array**

# 阻塞等待多个RTOS对象

### 队列集简介

[队列集](https://www.freertos.org/RTOS-queue-sets.html)是FreeRTOS的一项功能，当同时从多个队列和/或信号量接收时，RTOS任务可以阻塞（挂起）任务。队列和信号量被分为几组，然后，任务而不是在单个队列或信号量上阻塞，而是在集合上阻塞。

**注意：**如果您将FreeRTOS与旧版代码的第三方集成在一起，有时有时需要阻塞（挂起）多个队列，但是不受这种限制的设计通常可以使用替代方法以更有效的方式实现相同的功能。[在此页底部记录的](https://www.freertos.org/Pend-on-multiple-rtos-objects.html#alternative_design_pattern)设计模式 。

### 使用队列集

队列集以与select（）API函数和相关函数类似的方式使用，它们是标准Berkeley套接字网络API的一部分。

队列集可以包含队列和信号量，它们一起称为队列集成员。可以采用队列句柄或信号量句柄的API函数参数和返回值使用QueueSetMemberHandle\_t类型。通常，可以将QueueHandle\_t和SemaphoreHandle\_t类型的变量隐式转换为QueueSetMemberHandle\_t参数或返回值，而不会生成编译器警告（通常不需要显式转换为QueueSetMemberHandle\_t类型或从该类型强制转换）。

|  |  |
| --- | --- |
| **创建队列集** | 在使用队列集之前，必须使用[xQueueCreateSet（）](https://www.freertos.org/xQueueCreateSet.html) API函数创建它 。创建队列集后，将通过类型为QueueSetHandle\_t的变量引用该队列集。 |
| **将成员添加到队列集** | 所述[xQueueAddToSet（）](https://www.freertos.org/xQueueAddToSet.html) API函数用于队列或旗语添加到队列集合。 |
| **阻塞（挂起）队列集** | 所述[xQueueSelectFromSet（）](https://www.freertos.org/xQueueSelectFromSet.html) API函数用于测试任何成员集是否准备好用于读- ，其中读出装置“接收”当构件是一个队列，“取”当构件是一个信号量。  就像使用[xQueueReceive（）](https://www.freertos.org/a00118.html) 和[xSemaphoreTake（）](https://www.freertos.org/a00122.html) API函数时一样，xQueueSelectFromSet（）允许调用任务有选择地阻塞，直到队列集的成员准备好读取为止。  如果对xQueueSelectFromSet（）的调用超时，则返回NULL。否则，xQueueSelectFromSet（）返回准备读取的队列集成员的句柄，从而允许调用任务立即调用xQueueReceive（）或xSemaphoreTake（）（分别在队列句柄或信号量句柄上），并确保操作成功。 |

### 源代码示例

所述[xQueueCreateSet（）](https://www.freertos.org/xQueueCreateSet.html) API函数文档页面包括源代码的例子。

名为QueueSet.c的标准演示/测试文件（位于主要FreeRTOS zip文件下载的 FreeRTOS / Demo / Common / Minimal /目录中）包含一个完整的示例。

# 使用队列集的替代方法

除非有特定的集成问题需要在多个队列上进行阻塞，否则通常可以使用单个队列以更低的代码大小，RAM大小和运行时间开销实现相同的功能。FreeRTOS + UDP实现提供了一个方便的示例，说明了如何完成此操作，以下各小节对此进行了描述。

### UDP / IP堆栈：问题定义

管理FreeRTOS + UDP堆栈的任务是事件驱动的。有多个事件源。某些事件没有任何与之关联的数据。某些事件具有与之关联的可变数量的数据。活动包括：

* 接收帧的以太网硬件。帧包含大量可变数据。
* 以太网硬件完成帧的传输，释放网络和DMA缓冲区。
* 发送数据包的应用程序任务。数据包可能包含大量可变的数据。
* 各种软件计时器，包括ARP计时器。计时器事件不与任何数据关联。

### UDP / IP堆栈：解决方案

UDP / IP堆栈*可以*为每个事件源使用不同的队列，然后使用设置为立即阻塞所有队列的队列。相反，UDP / IP堆栈：

1. 定义一个结构，该结构包含一个用于保存事件类型的成员，以及另一个用于保存与事件关联的数据（或指向数据的指针）的成员。
2. 使用创建的单个队列来保存定义的结构。每个事件源都发布到同一队列。

结构定义如下所示。

**typedef struct IP\_TASK\_COMMANDS**

**{**

**eIPEvent\_t eEventType; /\* Tells the receiving task what the event is. \*/**

**void \*pvData; /\* Holds or points to any data associated with the event. \*/**

**} xIPStackEvent\_t;**

如何使用此结构的示例：

* ARP计时器到期时，它将事件发送到队列，且eEventType设置为eARPTimerEvent（枚举类型）。ARP计时器事件不与任何数据关联，因此未设置pvData。
* 当以太网驱动程序接收到帧时，它将事件发送到队列，其中eEventType设置为eEthernetRxEvent，而pvData设置为指向帧缓冲区。
* 等等。

UDP / IP任务使用一个简单的循环处理事件：

**/\* The variable used to receive from the queue. \*/**

**xIPStackEvent\_t xReceivedEvent;**

**for( ;; )**

**{**

**/\* Wait until there is something to do. \*/**

**xQueueReceive( xNetworkEventQueue, &xReceivedEvent, portMAX\_DELAY );**

**/\* Perform a different action for each event type. \*/**

**switch( xReceivedEvent.eEventType )**

**{**

**case eNetworkDownEvent :**

**prvProcessNetworkDownEvent();**

**break;**

**case eEthernetRxEvent :**

**prvProcessEthernetFrame( xReceivedEvent.pvData );**

**break;**

**case eARPTimerEvent :**

**prvAgeARPCache();**

**break;**

**case eStackTxEvent :**

**prvProcessGeneratedPacket( xReceivedEvent.pvData );**

**break;**

**case eDHCPEvent:**

**vDHCPProcess();**

**break;**

**default :**

**/\* Should not get here. \*/**

**break;**

**}**

**}**

# 延迟中断处理

### 什么是延迟中断处理？

在FreeRTOS中，延迟中断处理程序是指由中断服务程序（ISR）解除阻塞（触发）的RTOS任务，因此，中断所需的处理可以在解除阻塞的任务中执行，而不是直接在ISR中执行。该机制与标准中断处理不同，在标准中断处理中，所有处理都在ISR中执行，因为大部分处理都推迟到ISR退出之后：

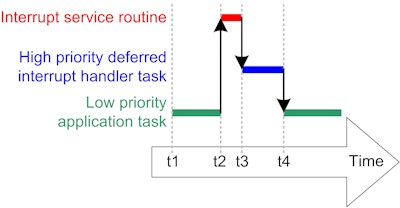
* **标准ISR处理**

标准的ISR处理通常包括记录中断原因，清除中断，然后在ISR本身内部执行中断所需的任何处理。

* **延迟中断处理**

延迟的中断处理通常将涉及记录中断原因并在ISR内清除中断，然后取消对RTOS任务的阻塞，以便中断所需的处理可以由未阻塞的任务执行，而不是在ISR内执行。

如果将要延迟处理的任务分配给足够高的优先级，那么ISR将直接返回到未阻塞的任务（中断将中断一个任务，然后返回到另一个任务），从而导致中断请求所需的所有处理。中断在时间上连续执行（没有间隔），就像所有处理都在ISR本身中执行一样。可以在下图中看到，其中所有中断处理都发生在时间t2和t4之间，即使部分处理是由任务执行的。

  
**延迟  
处理任务具有较高优先级 时，延迟中断处理执行顺序**

参考上图：

* 1. 在时间t2：低优先级任务被中断抢占。
  2. 在时间t3：ISR直接返回到在ISR内部不受阻碍的任务。大部分中断处理是在无阻塞任务中执行的。
  3. 在时间t4：未被ISR阻塞的任务返回“阻塞”状态以等待下一个中断，从而使优先级较低的应用程序任务得以继续执行。

### 何时使用延迟中断处理

大多数嵌入式工程师将努力使ISR内部花费的时间最小化（以最小化系统中的抖动，使相同或较低优先级的其他中断能够执行，最大化中断响应性等），以及推迟中断处理的技术。一项任务提供了一种方便的方法来实现这一目标。但是，首先解除阻塞然后再切换到RTOS任务本身的机制需要有限的时间，因此通常，只有在以下情况下，应用程序才能从推迟中断处理中受益：

* 需要执行冗长的操作，或者
* 使用完整的RTOS API（而不是仅使用ISR安全的API）或从中受益，或者
* 需要在合理的范围内执行不确定的操作。

### 将中断处理推迟到任务的技术

将中断推迟到任务的方法分为两类：

1. **集中式延迟中断处理**

之所以称为集中式延迟中断处理，是因为使用此方法的每个中断都在同一RTOS守护程序任务的上下文中执行。RTOS守护程序任务由FreeRTOS创建，也称为 [计时器服务任务](https://www.freertos.org/RTOS-software-timer-service-daemon-task.html)。

要将中断处理推迟到RTOS守护程序任务，请在对[xTimerPendFunctionCallFromISR（）](https://www.freertos.org/xTimerPendFunctionCallFromISR.html) API函数的调用[中将](https://www.freertos.org/xTimerPendFunctionCallFromISR.html)指向中断处理函数的指针作为xFunctionToPend参数[传递](https://www.freertos.org/xTimerPendFunctionCallFromISR.html)。有关工作示例，请参见xTimerPendFunctionCallFromISR（）文档页面。

集中式延迟中断处理的优点包括最小的资源使用，因为每个延迟中断处理程序都使用相同的任务。

集中式延迟中断处理的缺点包括：

* + 所有延迟的中断处理函数均在同一RTOS守护程序任务的上下文中执行，因此以相同的RTOS任务优先级执行。
  + xTimerPendFunctionCallFromISR（）通过计时器命令队列将指向延迟的中断处理函数的指针发送到RTOS守护程序任务。因此，RTOS守护程序任务按在队列中接收的顺序处理功能，而不必按中断优先级顺序处理。
  + 写入计时器命令队列，然后从中读取，会增加额外的延迟。

1. **应用控制的延迟中断处理**

之所以称为应用程序控制的延迟中断处理，是因为使用此方法的每个中断都在应用程序编写器创建的任务的上下文中执行。有关 工作示例，请参见将 [RTOS任务通知用作轻量计数信号量](https://www.freertos.org/RTOS_Task_Notification_As_Counting_Semaphore.html)文档页面。

应用程序控制的延迟中断处理的优点包括：

* + 减少延迟（功能指针不通过队列传递）。
  + 为每个处理RTOS任务的延迟中断分配不同优先级的能力–允许延迟中断任务的相对优先级匹配其各自中断的相对优先级。

应用程序控制的延迟中断处理的缺点包括资源消耗较大，因为通常需要更多的任务。

# 低功耗支持无 ick怠速模式

[**另请参见**[**ARM Cortex-M MCU的低功耗功能**](https://www.freertos.org/low-power-ARM-cortex-rtos.html)]  
[**另请参见**[**SAM4L**](https://www.freertos.org/Atmel_SAM4L-EK_Low_Power_Tick-less_RTOS_Demo.html)**，**[**RX100**](https://www.freertos.org/RX100_RSK_Low_Power_Tick-less_RTOS_Demo.html)**，**[**STM32L**](https://www.freertos.org/STM32L-discovery-low-power-tickless-RTOS-demo.html)**，**[**CEC1302**](https://www.freertos.org/Microchip_CEC1302_ARM_Cortex-M4F_Low_Power_Demo.html)**和**[**EFM32**](https://www.freertos.org/EFM32-Giant-Gecko-Pearl-Gecko-tickless-RTOS-demo.html)**MCU的Tickless演示**]

## 省电介绍

通常，通过使用[空闲任务挂钩](https://www.freertos.org/a00016.html)将微控制器置于低功耗状态，来减少运行FreeRTOS的微控制器的功耗。通过这种简单的方法可以实现的节能效果受到定期退出然后重新进入低功耗状态以处理滴答中断的限制。此外，如果滴答声中断的频率过高，则除了最轻的省电模式外，每个滴答声进入低功耗状态然后退出低功耗状态所消耗的能量和时间将超过所有潜在的节电增益。

FreeRTOS无滴答空闲模式在空闲期间（没有能够执行的应用程序任务的时间段）停止定期滴答中断，然后在滴答中断重新启动时对RTOS滴答计数值进行校正调整。

停止滴答中断可以使微控制器保持深度节能状态，直到发生中断，或者是RTOS内核将任务转换为就绪状态的时候。

## portSUPPRESS\_TICKS\_AND\_SLEEP（）宏

**portSUPPRESS\_TICKS\_AND\_SLEEP（xExpectedIdleTime）**

通过在FreeRTOSConfig.h中将configUSE\_TICKLESS\_IDLE定义为1来启用内置的无滴答空闲功能（对于支持此功能的端口）。通过在FreeRTOSConfig.h中将configUSE\_TICKLESS\_IDLE定义为2，可以为任何FreeRTOS端口（包括那些包含内置实现的端口）提供用户定义的无滴答空闲功能。

启用无滴答空闲功能后，当满足以下两个条件时，内核将调用portSUPPRESS\_TICKS\_AND\_SLEEP（）宏：

1. 空闲任务是唯一能够运行的任务，因为所有应用程序任务都处于“阻塞”状态或“挂起”状态。
2. 在内核将应用程序任务从“已阻止”状态转换到“已阻止”状态之前，至少还要经过*n个*完整的滴答周期，其中 *n*由FreeRTOSConfig.h中的configEXPECTED\_IDLE\_TIME\_BEFORE\_SLEEP定义设置。

portSUPPRESS\_TICKS\_AND\_SLEEP（）的单个参数的值等于将任务移入“就绪”状态之前的滴答周期总数。因此，参数值是微控制器在滴答中断停止（被抑制）时可以安全地保持在深度睡眠状态的时间。

**注意**：如果从portSUPPRESS\_TICKS\_AND\_SLEEP（）内部调用[eTaskConfirmSleepModeStatus（）](https://www.freertos.org/eTaskConfirmSleepModeStatus.html)返回eNoTasksWaitingTimeout，则微控制器可以无限期保持深度睡眠状态。当满足以下条件时，eTaskConfirmSleepModeStatus（）将仅返回eNoTasksWaitingTimeout：

1. 由于不使用软件计时器，因此调度程序不会在将来的任何时候执行计时器回调函数。
2. 所有应用程序任务都处于“挂起”状态，或者处于具有无限超时（portMAX\_DELAY的超时值）的“阻塞”状态，因此调度程序不会由于将来在任何固定时间将任务从“阻塞”状态移出而导致。

为了避免竞争情况，RTOS调度程序在调用portSUPPRESS\_TICKS\_AND\_SLEEP（）之前挂起，并在portSUPPRESS\_TICKS\_AND\_SLEEP（）完成后恢复。这样可确保应用程序任务无法在微控制器退出低功耗状态与portSUPPRESS\_TICKS\_AND\_SLEEP（）完成执行之间执行。此外，portSUPPRESS\_TICKS\_AND\_SLEEP（）函数必须在停止的滴答声源和微控制器进入睡眠状态之间创建一个小的临界区。应从此关键部分调用eTaskConfirmSleepModeStatus（）。

现在，所有GCC，IAR和Keil ARM Cortex-M3和ARM Cortex-M4端口都提供默认的portSUPPRESS\_TICKS\_AND\_SLEEP（）实现。“ [ARM Cortex-M MCU](https://www.freertos.org/low-power-ARM-cortex-rtos.html)的[低功耗功能”](https://www.freertos.org/low-power-ARM-cortex-rtos.html) 页面上提供了有关使用ARM Cortex-M实施的重要信息 。

随着时间的推移，默认实现将添加到其他FreeRTOS端口。同时，下面描述的钩子可用于向任何端口添加无滴答功能。

## 实施portSUPPRESS\_TICKS\_AND\_SLEEP（）

如果正在使用的FreeRTOS端口未提供portSUPPRESS\_TICKS\_AND\_SLEEP（）的默认实现，则应用程序编写者可以通过在FreeRTOSConfig.h中定义portSUPPRESS\_TICKS\_AND\_SLEEP（）来提供自己的实现。

如果正在使用的FreeRTOS端口确实提供了portSUPPRESS\_TICKS\_AND\_SLEEP（）的默认实现，则应用程序编写者可以通过在FreeRTOSConfig.h中定义portSUPPRESS\_TICKS\_AND\_SLEEP（）来覆盖默认实现。

以下源代码是应用程序编写器可能如何实现portSUPPRESS\_TICKS\_AND\_SLEEP（）的示例。该示例是基本示例，将介绍内核维护的时间与日历时间之间的一些偏差。FreeRTOS的官方版本试图通过提供更复杂的实现来消除任何滑动（尽可能）。

在示例中显示的函数调用中，只有 [vTaskStepTick（）](https://www.freertos.org/vTaskStepTick.html)和[eTaskConfirmSleepModeStatus（）](https://www.freertos.org/eTaskConfirmSleepModeStatus.html) 是FreeRTOS API的一部分。其他功能特定于正在使用的硬件上可用的时钟和省电模式，因此必须由应用程序编写器提供。

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

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

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

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

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

**{**

**unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;**

**eSleepModeStatus eSleepStatus;**

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

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

**ulLowPowerTimeBeforeSleep = ulGetExternalTime();**

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

**prvStopTickInterruptTimer();**

**/\* Enter a critical section that will not effect interrupts bringing the MCU**

**out of sleep mode. \*/**

**disable\_interrupts();**

**/\* Ensure it is still ok to enter the sleep mode. \*/**

**eSleepStatus = eTaskConfirmSleepModeStatus();**

**if( eSleepStatus == eAbortSleep )**

**{**

**/\* A task has been moved out of the Blocked state since this macro was**

**executed, or a context siwth is being held pending. Do not enter a**

**sleep state. Restart the tick and exit the critical section. \*/**

**prvStartTickInterruptTimer();**

**enable\_interrupts();**

**}**

**else**

**{**

**if( eSleepStatus == eNoTasksWaitingTimeout )**

**{**

**/\* It is not necessary to configure an interrupt to bring the**

**microcontroller out of its low power state at a fixed time in the**

**future. \*/**

**prvSleep();**

**}**

**else**

**{**

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

**power state at the time the kernel next needs to execute. The**

**interrupt must be generated from a source that remains operational**

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

**vSetWakeTimeInterrupt( xExpectedIdleTime );**

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

**prvSleep();**

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

**state for, which will be less than xExpectedIdleTime if the**

**microcontroller was brought out of low power mode by an interrupt**

**other than that configured by the vSetWakeTimeInterrupt() call.**

**Note that the scheduler is suspended before**

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

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

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

**ulLowPowerTimeAfterSleep = ulGetExternalTime();**

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

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

**vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );**

**}**

**/\* Exit the critical section - it might be possible to do this immediately**

**after the prvSleep() calls. \*/**

**enable\_interrupts();**

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

**prvStartTickInterruptTimer();**

**}**

**}**

**用户定义的portSUPPRESS\_TICKS\_AND\_SLEEP（）的示例实现**

# 跟踪挂钩宏

|  |
| --- |
| [也可以使用：**FreeRTOS + Trace**，这是FreeRTOS的第三方跟踪工具-FreeRTOS +生态系统的一部分](https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_Trace/FreeRTOS_Plus_Trace.html) |

### 描述

跟踪挂钩宏是一项非常强大的功能，可让您收集有关嵌入式应用程序行为方式的数据。

FreeRTOS源代码中的主要关注点包含空宏，应用程序可以重新定义空宏，以提供特定于应用程序的跟踪工具。应用程序仅需要实现那些特别令人感兴趣的宏，而未使用的宏保持为空，因此不会影响应用程序的时序。

### 例子

下面是如何使用这些宏的一些示例：

* 设置数字输出以指示正在执行的任务–允许使用逻辑分析仪查看和记录任务执行的顺序和时间。
* 同样-将模拟输出设置为代表正在执行任务的电压-允许使用示波器查看和记录任务执行顺序和时间。
* 记录任务执行序列，任务定时，RTOS内核事件和API调用以进行离线分析。
* 将RTOS内核事件集成到第三方调试器中。

#### 例子1

FreeRTOS任务标签功能为通过数字或模拟输出设置日志记录提供了一种简单的机制。例如，可以将标签值设置为该任务唯一的电压。然后可以定义traceSWITCHED\_IN（）宏，以将模拟输出简单地设置为与要接通的任务关联的值。例如：

**/\* First task sets its tag value to 1. \*/**

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

**{**

**/\* This task is going to be represented by a voltage scale of 1. \*/**

**vTaskSetApplicationTaskTag( NULL, ( void \* ) 1 );**

**for( ;; )**

**{**

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

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\* Second task sets its tag value to 2. \*/**

**void vTask2( void \*pvParameters )**

**{**

**/\* This task is going to be represented by a voltage scale of 2. \*/**

**vTaskSetApplicationTaskTag( NULL, ( void \* ) 2 );**

**for( ;; )**

**{**

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

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\* Define the traceTASK\_SWITCHED\_IN() macro to output the voltage associated**

**with the task being selected to run on port 0. \*/**

**#define traceTASK\_SWITCHED\_IN() vSetAnalogueOutput( 0, (int)pxCurrentTCB->pxTaskTag )**

#### 例子2

API调用日志记录可用于记录发生上下文切换的原因。RTOS内核调用日志记录可用于记录任务执行的顺序。例如：

**/\* traceBLOCKING\_ON\_QUEUE\_RECEIVE() is just one of the macros that can be used to**

**record why a context switch is about to occur. \*/**

**#define traceBLOCKING\_ON\_QUEUE\_RECEIVE(xQueue)**

**ulSwitchReason = reasonBLOCKING\_ON\_QUEUE\_READ;**

**/\* log\_event() is an application defined function that logs which tasks ran when,**

**and why. \*/**

**#define traceTASK\_SWITCHED\_OUT()**

**log\_event( pxCurrentTCB, ulSwitchReason );**

### 定义

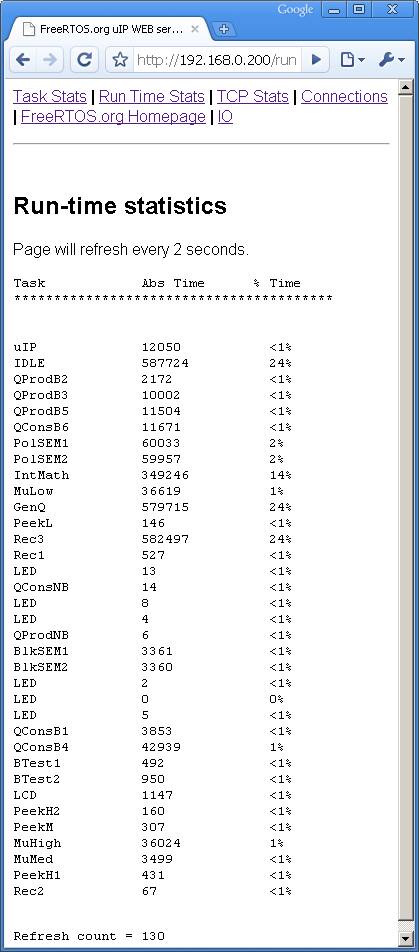
从中断（尤其是滴答中断）内部调用的宏必须快速执行并且不占用过多的堆栈空间。设置变量，写入跟踪寄存器或输出到端口都是可以接受的。尝试将fprintf（）将数据记录到慢速磁盘上将不起作用！

宏定义必须在包含FreeRTOS.h之前发生。定义跟踪宏的最简单位置是FreeRTOSConfig.h的底部，或者是位于FreeRTOSConfig.h底部的单独的头文件中。

下表描述了可用的宏。宏参数用于指示与记录的事件相关联的任务，队列，信号量或互斥量。

|  |  |
| --- | --- |
| **宏定义** | **描述** |
| traceTASK\_INCREMENT\_TICK（xTickCount） | 在滴答中断期间调用。 |
| traceTASK\_SWITCHED\_OUT（） | 在选择要运行的新任务之前调用。此时，pxCurrentTCB包含即将离开运行状态的任务的句柄。 |
| traceTASK\_SWITCHED\_IN（） | 在选择要运行的任务后调用。此时，pxCurrentTCB包含即将进入运行状态的任务的句柄。 |
| traceMOVED\_TASK\_TO\_READY\_STATE（xTask） | 当任务转换为就绪状态时调用。 |
| traceBLOCKING\_ON\_QUEUE\_RECEIVE（xQueue） | 表示在尝试从空队列中读取或试图“获取”空信号量或互斥锁之后，当前正在执行的任务将被阻塞。 |
| traceBLOCKING\_ON\_QUEUE\_SEND（xQueue） | 表示尝试写入完整队列后，当前正在执行的任务即将阻塞。 |
| traceGIVE\_MUTEX\_RECURSIVE（xMutex） | 从xSemaphoreGiveRecursive（）内部调用。 |
| traceGIVE\_MUTEX\_RECURSIVE\_FAILED（xMutex） | 从xSemaphoreGiveRecursive（）内部调用。 |
| traceQUEUE\_CREATE（pxNewQueue） | 如果成功创建队列，则从xQueueCreate（）或xQueueCreateStatic（）内部调用。 |
| traceQUEUE\_CREATE\_FAILED（） | 如果由于可用的堆内存不足而未能成功创建队列，则从xQueueCreate（）或xQueueCreateStatic（）内部调用。 |
| traceCREATE\_MUTEX（pxNewMutex） | 如果成功创建了互斥锁，则从xSemaphoreCreateMutex（）中调用。 |
| traceCREATE\_MUTEX\_FAILED（） | 如果由于可用的堆内存不足而无法成功创建互斥锁，则从xSemaphoreCreateMutex（）中调用。 |
| traceGIVE\_MUTEX\_RECURSIVE（xMutex） | 如果互斥成功完成，则从xSemaphoreGiveRecursive（）内部调用。 |
| traceGIVE\_MUTEX\_RECURSIVE\_FAILED（xMutex） | 如果未成功给出互斥锁，则从xSemaphoreGiveRecursive（）中调用，因为调用任务不是互斥锁的所有者。 |
| traceTAKE\_MUTEX\_RECURSIVE（xMutex） | 从xQueueTakeMutexRecursive（）内部调用。 |
| traceCREATE\_COUNTING\_SEMAPHORE（） | 如果成功创建了信号量，则从xSemaphoreCreateCounting（）中调用。 |
| traceCREATE\_COUNTING\_SEMAPHORE\_FAILED（） | 如果由于可用的堆内存不足而未能成功创建信号量，则从xSemaphoreCreateCounting（）中调用。 |
| traceQUEUE\_SEND（xQueue） | 当队列发送成功时，从xQueueSend（），xQueueSendToFront（），xQueueSendToBack（）或任何信号量'give'函数中调用。 |
| traceQUEUE\_SEND\_FAILED（xQueue） | 当由于队列已满（在指定的任何块时间之后）而导致队列发送操作失败时，从xQueueSend（），xQueueSendToFront（），xQueueSendToBack（）或任何信号量'give'函数中调用。 |
| traceQUEUE\_RECEIVE（xQueue） | 当队列接收成功时，从xQueueReceive（）或任何信号量“ take”函数中调用。 |
| traceQUEUE\_RECEIVE\_FAILED（xQueue） | 当由于队列为空（在指定的任何块时间之后）而导致队列接收操作失败时，从xQueueReceive（）或任何信号量“获取”函数中调用。 |
| traceQUEUE\_PEEK（xQueue） | 从xQueuePeek（）中调用 |
| traceQUEUE\_SEND\_FROM\_ISR（xQueue） | 发送操作成功时，从xQueueSendFromISR（）中调用。 |
| traceQUEUE\_SEND\_FROM\_ISR\_FAILED（xQueue） | 当发送操作由于队列已满而失败时，从xQueueSendFromISR（）中调用。 |
| traceQUEUE\_RECEIVE\_FROM\_ISR（xQueue） | 接收操作成功时，从xQueueReceiveFromISR（）中调用。 |
| traceQUEUE\_RECEIVE\_FROM\_ISR\_FAILED（xQueue） | 当由于队列已为空而导致接收操作失败时，从xQueueReceiveFromISR（）中调用。 |
| traceQUEUE\_DELETE（xQueue） | 从vQueueDelete（）中调用。 |
| traceTASK\_CREATE（xTask） | 成功创建任务时，从xTaskCreate（）（或[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)）内部调用。 |
| traceTASK\_CREATE\_FAILED（pxNewTCB） | 当由于可用的堆空间不足而未能成功创建任务时，从xTaskCreate（）（或[xTaskCreateStatic（）](https://www.freertos.org/xTaskCreateStatic.html)）内部调用。 |
| traceTASK\_DELETE（xTask） | 从vTaskDelete（）中调用。 |
| traceTASK\_DELAY\_UNTIL（） | 从vTaskDelayUntil（）内部调用。 |
| traceTASK\_DELAY（） | 从vTaskDelay（）内部调用。 |
| traceTASK\_PRIORITY\_SET（xTask，uxNewPriority） | 从vTaskPrioritySet（）中调用。 |
| traceTASK\_SUSPEND（xTask） | 从vTaskSuspend（）内部调用。 |
| traceTASK\_RESUME（xTask） | 从vTaskResume（）内部调用。 |
| traceTASK\_RESUME\_FROM\_ISR（xTask） | 从xTaskResumeFromISR（）内部调用。 |
| traceTIMER\_COMMAND\_RECEIVED（pxTimer，xCommandID，xCommandValue） | 在实际处理命令之前，每次在计时器服务任务中收到命令时都会调用该命令。 |
| traceTIMER\_COMMAND\_SEND（pxTimer，xCommandID，xOptionalValue，xStatus） | 从任何将命令发送到计时器服务任务的API函数中调用，例如xTimerReset（），xTimerStop（）等。如果未成功将命令发送到计时器命令队列，则xStatus将为pdFAIL。 |
| traceTIMER\_CREATE（pxNewTimer） | 如果成功创建了计时器，则从xTimerCreate（）中调用。 |
| traceTIMER\_CREATE\_FAILED（） | 如果由于可用的堆内存不足而未能成功创建计时器，请从xTimerCreate（）中调用。 |
| traceTIMER\_EXPIRED（pxTimer） | 在软件计时器到期时调用，在执行计时器回调之前调用。 |

# 运行时间统计

点击放大  
[](https://www.freertos.org/fr-content-src/uploads/2018/07/rtos-run-time-stats.jpg)

### 描述

FreeRTOS可以选择收集有关每个任务已使用的处理时间的信息。所述[vTaskGetRunTimeStats（）](https://www.freertos.org/a00021.html#vTaskGetRunTimeStats)然后API函数可用于以表格的形式呈现该信息，在右侧，如图所示。

每个任务都有两个值：

1. 绝对时间（绝对时间）

这是任务实际执行的总“时间”（任务一直处于“运行”状态的总时间）。用户可以根据自己的应用选择合适的时基。

1. 时间百分比（时间百分比）

这显示了基本相同的信息，但显示为总处理时间的百分比，而不是绝对时间。

### 配置和用法

需要三个宏。这些可以在FreeRTOSConfig.h中定义。

1. configGENERATE\_RUN\_TIME\_STATS

通过将configGENERATE\_RUN\_TIME\_STATS定义为1来启用运行时统计信息的收集。一旦设置了该值，则还必须定义其他两个宏才能成功编译。

1. portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS（）

运行时统计信息时基需要比滴答中断具有更高的分辨率–否则，统计信息可能太不准确而无法真正有用。建议使时基比滴答中断快10至100倍。时基越快，统计信息将越准确-但计时器值也会越早溢出。

如果将configGENERATE\_RUN\_TIME\_STATS定义为1，则RTOS内核将在启动时自动调用portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS（）（从vTaskStartScheduler（）API函数内部调用）。旨在使应用程序设计者使用宏来配置合适的时基。下面提供了一些示例。

1. portGET\_RUN\_TIME\_COUNTER\_VALUE（）

该宏应仅返回当前的“时间”，如portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS（）所配置。再次在下面提供一些示例。

该[vTaskGetRunTimeStats（）](https://www.freertos.org/a00021.html#vTaskGetRunTimeStats) API函数用于检索收集的统计数据。

### 例子

[在许多其他应用中] [LPC17xx](https://www.freertos.org/portLM3Sxxxx_Eclipse.html)[LPCXpresso](https://www.freertos.org/Free-RTOS-For-NXP-Cortex-M3-LPC17xx.html) 和[LM3Sxxxx Eclipse](https://www.freertos.org/portLM3Sxxxx_Eclipse.html) Web服务器演示应用程序已配置为生成运行时统计信息。

LM3Sxxxx示例

LM3Sxxxx Eclipse演示应用程序已经包含20KHz计时器测试。更新了中断处理程序，使其在每次执行时仅递增一个名为ulHighFrequencyTimerTicks的变量。portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS（）只需将此变量设置为0，portGET\_RUN\_TIME\_COUNTER\_VALUE（）返回其值。为了实现这一点，在FreeRTOSConfig.h中添加了以下几行：

**extern volatile unsigned long ulHighFrequencyTimerTicks;**

**/\* ulHighFrequencyTimerTicks is already being incremented at 20KHz. Just set**

**its value back to 0. \*/**

**#define portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() ( ulHighFrequencyTimerTicks = 0UL )**

**#define portGET\_RUN\_TIME\_COUNTER\_VALUE() ulHighFrequencyTimerTicks**

LPC17xx示例

LPC17xx演示应用程序不包括高频中断测试，因此使用portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS（）将定时器0外设配置为生成时基。portGET\_RUN\_TIME\_COUNTER\_VALUE（）仅返回当前计时器0的计数器值。这是使用以下函数和宏实现的。

**/\* Defined in main.c. \*/**

**void vConfigureTimerForRunTimeStats( void )**

**{**

**const unsigned long TCR\_COUNT\_RESET = 2,**

**CTCR\_CTM\_TIMER = 0x00,**

**TCR\_COUNT\_ENABLE = 0x01;**

**/\* Power up and feed the timer with a clock. \*/**

**PCONP |= 0x02UL;**

**PCLKSEL0 = (PCLKSEL0 & (~(0x3<<2))) | (0x01 << 2);**

**/\* Reset Timer 0 \*/**

**T0TCR = TCR\_COUNT\_RESET;**

**/\* Just count up. \*/**

**T0CTCR = CTCR\_CTM\_TIMER;**

**/\* Prescale to a frequency that is good enough to get a decent resolution,**

**but not too fast so as to overflow all the time. \*/**

**T0PR = ( configCPU\_CLOCK\_HZ / 10000UL ) - 1UL;**

**/\* Start the counter. \*/**

**T0TCR = TCR\_COUNT\_ENABLE;**

**}**

**/\* Defined in FreeRTOSConfig.h. \*/**

**extern void vConfigureTimerForRunTimeStats( void );**

**#define portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() vConfigureTimerForRunTimeStats()**

**#define portGET\_RUN\_TIME\_COUNTER\_VALUE() T0TC**

# 创建一个新的FreeRTOS端口

**如果使用许多现有端口和演示应用程序之一，则无需阅读或理解此页面！**

将FreeRTOS移植到一个完全不同且尚不支持的微控制器并不是一件容易的事。此页面的目的是描述启动新端口所需的整理房间的前提条件。

每个端口都具有一定的唯一性，并且非常取决于所使用的处理器和工具，因此此页面无法提供有关端口详细信息的详细信息。但是[，](https://www.freertos.org/RTOS_ports.html)还有 [许多其他FreeRTOS端口已经存在](https://www.freertos.org/RTOS_ports.html)，建议将其用作参考。

在同一个处理器系列中进行移植是一项更为直接的任务-例如，从一个基于ARM7的设备移植到另一个。如果这是您的目标，那么详细说明[如何修改现有演示应用程序](https://www.freertos.org/porting-a-freertos-demo-to-different-hardware.html)的文档页面 将是一个不错的起点。

### 设置目录结构

FreeRTOS内核源代码通常包含在所有端口共有的3个源文件（如果使用协程，则为4个）和一个或两个“ RT”内核文件，这些文件可将RTOS内核定制为特定的体系结构。

建议的步骤：

1. 下载最新版本的FreeRTOS源代码。
2. 将文件解压缩到一个方便的位置，注意保持目录结构。
3. 熟悉[源代码的组织](https://www.freertos.org/a00017.html)和目录结构。
4. 创建一个目录，其中将包含[体系结构]端口的“端口”文件。按照链接中概述的约定，目录的格式应为：FreeRTOS / Source / portable / [编译器名称] / [处理器名称]。例如，如果您使用的是GCC编译器，则可以在现有FreeRTOS / Source / portable / GCC目录之外创建一个[architecture]目录。
5. 将空的port.c和portmacro.h文件复制到您刚刚创建的目录中。这些文件应仅包含需要实现的函数和宏的存根。有关此类功能和宏的列表，请参阅现有的port.c和portmacro.h文件。您可以通过简单地删除函数和宏主体，从这些现有文件之一创建存根文件。
6. 如果要移植的微控制器上的堆栈从高内存向下扩展到低内存，则将portmacro.h中的portSTACK\_GROWTH设置为-1，否则将portSTACK\_GROWTH设置为1。
7. 创建一个目录，其中将包含[architecture]端口的[演示应用程序](https://www.freertos.org/a00102.html)文件。再次遵循约定，其格式应为FreeRTOS / Demo / [architecture\_compiler]或类似形式。
8. 将现有的FreeRTOSConfig.h和main.c文件复制到刚刚创建的目录中。同样，应将它们编辑为存根文件。
9. 看一下FreeRTOSConfig.h文件。它包含一些需要为您选择的硬件进行设置的宏。
10. 从刚刚创建的目录创建一个目录，并将其命名为ParTest（可能是FreeRTOS / Demo / [architecture\_compiler] / ParTest）。将一个ParTest.c存根文件复制到此目录中。
11. ParTest.c包含以下三个简单函数：
    1. 设置一些可以使几个LED闪烁的GPIO，
    2. 设置或清除特定的LED，并且
    3. 切换LED的状态。

这三个功能需要为您的开发板实施。使LED输出正常工作将有助于其余所需的工作。看看其他演示项目中包含的许多现有ParTest.c文件作为示例（ParTest名称是PARallel端口TEST的历史异常），以及[描述如何修改现有演示应用程序](https://www.freertos.org/porting-a-freertos-demo-to-different-hardware.html)以获取编写和测试信息的页面。 ParTest.c函数。

### 创建一个项目

现在，所有必需的文件都就位了，您需要创建一个可以成功构建它们的项目（或makefile）。显然，它们只包含存根，因此不会做任何事情，但是一旦它们构建了存根，就可以用工作功能逐步替换它们。

该项目将需要包含以下文件：

* 来源/tasks.c
* 源/Queue.c
* 源/List.c
* 源/便携式/ [编译器名称] / [处理器名称] /port.c
* 源/便携式[/MemMang/heap\_1.c（或heap\_2.c或heap\_3.c或heap\_4.c）](https://www.freertos.org/a00111.html)
* 演示/ [处理器名称] /main.c
* 演示/ [处理器名称] /ParTest/ParTest.c

以下目录必须位于包含路径中–请使用Demo / [Process name]目录中的相对路径–不能是绝对路径：

* 演示/通用（即../通用）
* 演示/ [处理器名称]
* 来源/包含
* 源/便携式/ [编译器名称] / [处理器名称]

### 实施存根

现在困难一点。项目编译完成后，便需要实施可移植层存根。建议pxPortInitialiseStack（）是要实现的第一个函数。要实现pxPortInitialiseStack（），您必须首先确定您的任务上下文堆栈框架结构，该结构非常依赖于体系结构。

### 获得帮助

不要忘记，[WITTENSTEIN高完整性系统](http://www.openrtos.com/)可提供全面的移植和测试服务！

# Posix/Linux Simulator Demo for FreeRTOS using GCC [[RTOS Ports](https://www.freertos.org/a00090.html)]

**Note:** Due to recent updates to enable its use on Mac computers and in the Windows Subsystem for Linux (WSL) the Linux/POSIX port in FreeRTOS V10.4.0 is considered to be a release candidate only. Please provide feedback in the [FreeRTOS support forum](https://forums.freertos.org/).

The FreeRTOS port documented on this page allows FreeRTOS to run on Linux just as the [FreeRTOS Windows port](https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html) (often referred to as the FreeRTOS Windows simulator) has always allowed FreeRTOS to run on Windows. The port was contributed by David Vrabel, and inspired by William Davy’s original 2008 Linux port.

The implementation of the port layer uses POSIX threading, so the port is also referred to as the POSIX port. It should not be confused with the [FreeRTOS+POSIX library](https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_POSIX/index.html) – they do completely different things. FreeRTOS+POSIX provides a POSIX threading wrapper for the native FreeRTOS API, whereas the implementation of the Linux port uses the POSIX threading API provided by the host operating system.

Just like the Windows port, the FreeRTOS Linux port provides a convenient environment in which you can experiment with FreeRTOS and develop FreeRTOS applications intended for later porting to real embedded hardware – but it will not exhibit real-time behaviour.

### IMPORTANT! Notes on using the Posix/Linux Simulator Demo for FreeRTOS

*Please read all the following points before using the simulator demo.*

* [Source code organisation](https://www.freertos.org/FreeRTOS-simulator-for-Linux.html#SourceCodeOrg)
* [The FreeRTOS Linux port demo application](https://www.freertos.org/FreeRTOS-simulator-for-Linux.html#SimulatorApp)
* [Building and running the Posix/Linux Simulator Demo for FreeRTOS](https://www.freertos.org/FreeRTOS-simulator-for-Linux.html#build_instructions_1)
* [GDB Debugging Tips](https://www.freertos.org/FreeRTOS-simulator-for-Linux.html#gdb_debugging_tips)
* [Port-Layer design methodology justification](https://www.freertos.org/FreeRTOS-simulator-for-Linux.html#port_layer_design)
* [Known issues](https://www.freertos.org/FreeRTOS-simulator-for-Linux.html#known_issues)

Also see the FAQ [My application does not run, what could be wrong?](https://www.freertos.org/FAQHelp.html).

## Source Code Organization

The [FreeRTOS zip file download](https://www.freertos.org/a00104.html) contains the source code for all the FreeRTOS ports and demo applications – so it contains many more files than are required to build and run the pre-configured demo that uses the FreeRTOS Linux port. See the [Source Code Organization](https://www.freertos.org/a00017.html) page for information on the zip file’s directory structure.

* The makefile that builds the demo project documented on this page is contained in the [FreeRTOS/Demo/POSIX\_GCC](https://github.com/FreeRTOS/FreeRTOS/tree/master/FreeRTOS/Demo/Posix_GCC) directory.
* The RTOS port layer for Linux (POSIX) is located in the [FreeRTOS/Source/portable/ThirdParty/GCC/Posix](https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/master/portable/ThirdParty/GCC/Posix) directory.

## The Posix/Linux Simulator Demo

### Functionality

The constant mainSELECTED\_APPLICATION, which is #defined at the top of main.c, is used to switch between a simple Blinky style demo, a more comprehensive test and demo application, and a tcp echo client application, as described in the next three sections.

Blinky demo functionality

If mainSELECTED\_APPLICATION is set to BLINKY\_DEMO, then main() will call main\_blinky(), which is implemented in main\_blinky.c. main\_blinky() creates a very simple demo that includes two tasks, a software timer, and a queue. One task repeatedly sends the value 100 at a frequency of 200 milliseconds to the other task through the queue, while the timer sends the value 200 every 2000ms to the same queue. The receiving task prints out a message each time it receives either of the values from the queue.

Comprehensive demo functionality

If mainSELECTED\_APPLICATION is set to FULL\_DEMO, then main() will call main\_full(), which is implemented in main\_full.c. The demo created by main\_full() consists mainly of the standard demo tasks which don’t perform any particular functionality other than testing the port and demonstrating how the FreeRTOS API can be used.

The full demo includes a ‘check’ that executes every (simulated) ten seconds, but has the highest priority to ensure it gets processing time. Its main function is to check all the standard demo tasks are still operational. The check task maintains a status string that is output to the console each time it executes. If all the standard demo tasks are running without error, then the string contains “OK” and the current tick count. If an error has been detected, then the string contains a message that indicates which task reported the error.

Echo client functionality

If mainSELECTED\_APPLICATION is set to ECHO\_CLIENT\_DEMO, then main() will call main\_tcp\_echo\_client\_tasks(), which is is implemented in main\_networking.c.

The TCP echo demo uses the [FreeRTOS+TCP TCP/IP stack](https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/index.html) to connect and communicate with a [standard TCP echo server](https://en.wikipedia.org/wiki/Echo_Protocol) on TCP port 7. The FreeRTOS+TCP network interface for Linux uses libpcap to access the network.

To configure the TCP/IP stack for use with the demo:

* Follow the instructions under the [Software Setup #1, Software Setup #2, and Software Setup #4 sections](https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/examples_FreeRTOS_simulator.html#static-dynamic) on the page that describes using FreeRTOS+TCP on Windows hosts (the steps under Software Setup #3 are not required).
* Set the constants configECHO\_SERVER\_ADDR0 to configECHO\_SERVER\_ADDR3 to the address of the echo server in FreeRTOSConfig.h.

It is common for TCP port 7 (the standard echo port) to be blocked by firewalls. If this is the case then change the port number used by both the FreeRTOS application and the echo server to a high but valid number, such as 5200. The port number used by the FreeRTOS application is set by the echoECHO\_PORT constant in [TCPEchoClient\_SingleTasks.c](https://github.com/FreeRTOS/FreeRTOS/blob/master/FreeRTOS/Demo/Posix_GCC/TCPEchoClient_SingleTasks.c). If you create a TCP echo server using the nc command in Linux then the port number is set using the -l switch:

**$ sudo nc -l 7**

#### Network troubleshooting

[ARP](https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/ARP.html) responses may not get sent if the echo server is running on the same computer as the FreeRTOS demo, resulting in the demo not being able to connect to the echo server. If this is an issue then run the echo server on a different computer.

## Building the Posix/Linux Simulator Demo

* Pre-requisites (the output below shows the versions used during testing):
  1. gcc
  2. **$ gcc --version**

**gcc (GCC) 9.2.0**

* 1. make
  2. **$ make --version**
  3. **GNU Make 3.81**

**Copyright (C) 2006 Free Software Foundation, Inc.**

* 1. [libpcap](https://www.tcpdump.org/) (for networking support)

**$ version: libpcap-devel-1.5.3-11.x86\_64**

To install on ubuntu run

**$ sudo apt-get install libpcap-dev**

To install on rpm based system run

**$ sudo yum install libpcap-devel**

or

**$ sudo dnf install libpcap-devel**

To install on MacOS run

**$ brew install libpcap**

Alternatively, it is possible to install from [source](https://github.com/the-tcpdump-group/libpcap) follow the instructions at [INSTALL.md](https://github.com/the-tcpdump-group/libpcap/blob/master/INSTALL.md)

* Building the source code:
  1. Navigate to the Demo Directory at: [Demo source](https://github.com/FreeRTOS/FreeRTOS/tree/master/FreeRTOS/Demo/Posix_GCC)

**$ cd FreeRTOS/Demo/Posix\_GCC/**

* 1. To build run:

**$ make**

* 1. To clean run:

**$ make clean**

* Running the Demo (check the instructions above on how to choose between the available demos)
  1. Navigate to the newly created “build” directory

**$ cd build**

* 1. Run the demo (blinky and full)

**$ ./posix\_demo**

* 1. Run the Networking demo:

Run an echo server on a different machine

**$ sudo nc -l 7**

Run on your machine

**$ sudo ./posix\_demo**

## GDB Debugging Tips

This section assumes that you have installed and are familiar with [gdb](https://www.gnu.org/software/gdb/). You can find the gdb documentation [here](https://www.gnu.org/software/gdb/documentation/).

The port layer uses two process signals: SIGUSR1, and SIGALRM. If a pthread is not waiting on the signal, then GDB will pause the process when it receives the signal. GDB must be told to ignore (and not print) the signal SIGUSR1 because it is received asynchronously by each thread. In GDB, enter

**$ handle SIGUSR1 nostop noprint pass**

to ensure that debugging is not interrupted by the signals. See

**$ man signal**

for more information.

Alternatively, create a file in your home directory called .gdbinit and place the following two lines in it:

**handle SIGUSR1 nostop noignore noprint**

**handle SIGALRM nostop noignore noprint**

When you add these two lines to the .gdbinit file, it tells GDB to not break on those signals.

There are three different timers available for use as the System tick: ITIMER\_REAL, ITIMER\_VIRTUAL and ITIMER\_PROF. The default timer is ITIMER\_VIRTUAL because it only counts when the process is executing in user space, so it will stop when a break-point is hit. ITIMER\_PROF is equivalent to ITIMER\_VIRTUAL but it includes the time spent executing system calls. ITIMER\_REAL continues counting even when the process is not executing at all, so it represents real-time. ITIMER\_REAL is the only usable option because the other timers don't tick unless the process is actually running. For that reason, if nanosleep is called in the IDLE task hook, the time reported by the non-real timers will hardly ever increase.

## Port-Layer Design Methodology Justification

A simple implementation of a FreeRTOS Simulator would simply wrap the platform native threads, and all calls to switch Task contexts would call the OS suspend and resume thread API. This simulator uses the Posix condition variables and Signals to control the execution of the underlying Posix threads. Signals can be delivered to the threads asynchronously, so that they interrupt the execution of the target thread, while suspended threads wait on condition variables to resume.

Typically, when designing a multi-threaded process, we use multiple threads to allow for concurrent execution and to implement a degree of non-blocking on IO tasks. This simulator does not use the threads to achieve concurrent execution, but only to store the context of the execution. Signals, mutexes and condition variables are used to synchronize context switching, but ultimately, the decision to change the context is driven by the FreeRTOS scheduler.

When a new Task is created, a pthread is created as the context for the execution of that Task. The pthread immediately suspends itself, and returns the execution to the creator. When a pthread is suspended, it is waiting in a call to pthread\_cond\_wait, which is blocked until it receives a resume signalpthread\_cond\_signal.

FreeRTOS Tasks can be switched in two ways, co-operatively by calling taskYIELD() or pre-emptively as part of the System Tick. In this simulator, the Task contexts are switched by resuming the next task context (decided by the FreeRTOS Scheduler) and suspending the current context (with a brief handshake between the two).

The System tick is generated using an ITIMER and the signal is delivered (only to) the currently executing pthread. The System Tick Signal Handler increments the tick and selects the next context. It resumes that thread and sends a signal to itself to suspend. The suspend is only processed when the System Tick Signal Handler exits, because signals are queued.

## Known Issues

pthread\_create and pthread\_exit/cancel are system intensive calls which can rapidly saturate the processing time.

If you call system and library functions that block (printf), this could cause the whole process to halt. If it is necessary to make system calls, all signals on that thread must be masked, then re-allowed after the system call finishes execution. Additional threads could be created to simulate interrupts, but signals should be masked in those threads as well, so that they do not receive signals and so be scheduled to execute by the FreeRTOS scheduler and become part of the regular FreeRTOS tasks.

To prevent the process from stealing all of the Idle execution time of the Host OS, use nano\_sleep(). It doesn't use any signals in its implementation but will abort from the sleep/suspend process immediately to service a signal. Therefore, the best way to use it is to set a sleep time longer than a FreeRTOS execution time-slice and call it from the Idle task so that the process suspends until the next tick.

# 旧版跟踪实用程序

[该页面记录了旧版跟踪实用程序-在FreeRTOS V7.1.0中已被删除-用户可能会发现更新，更易于使用的较新的[跟踪挂钩宏](https://www.freertos.org/rtos-trace-macros.html)。

跟踪可视化实用程序允许检查RTOS活动。

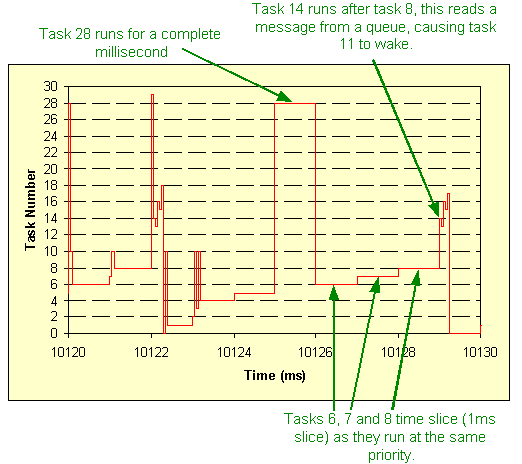
它记录任务分配给微控制器处理时间的顺序。

要使用该实用程序，在编译应用程序时，必须在FreeRTOSConfig.h中将宏configUSE\_TRACE\_FACILITY定义为1 。有关更多信息，请参见RTOS API文档中的配置部分。

跟踪通过调用vTaskStartTrace（）开始，并通过调用ulTask​​EndTrace（）结束。如果缓冲区已满，它将自动结束。

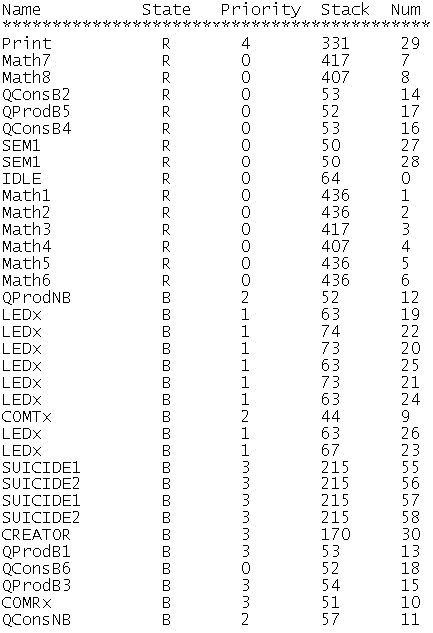
可以将完成的跟踪缓冲区存储到磁盘以供脱机检查。DOS / Windows实用程序tracecon.exe将存储的缓冲区转换为制表符分隔的文本文件。然后可以将其打开并在电子表格应用程序中进行检查。

以下是从AMD 186演示应用程序收集的10毫秒示例输出。x轴显示时间的流逝，y轴显示正在运行的任务的数量。



每个任务在创建时都会自动分配一个编号。vTaskList（）可用于获取分配给每个任务的编号以及一些其他有用的信息。演示应用程序期间vTaskList（）返回的信息如下所示，其中：

* 名称–是创建任务时给它的名称。请注意，该演示应用程序创建了多个任务的多个实例。
* 状态–显示任务的状态。可以是“ B”锁定，“ R” eady，“ S”挂起或“ D”删除。
* 优先级–是任务创建时赋予的优先级。
* 堆栈–显示任务堆栈的高水位线。这是任务生命周期内可用的最小可用堆栈量。
* Num –是自动分配给任务的数字。



在此示例中，可以看到任务6、7、8和14都以优先级0运行。因此，它们在自己与其他优先级0任务（包括空闲任务）之间的时间片。任务14从队列中读取一条消息（请参阅演示应用程序中的BlockQ.c）。这样可以释放队列中的空间。任务13被阻止，等待空间可用，因此现在唤醒，发布消息，然后再次阻止。

**注意：**在当前的实现中，跟踪的时间分辨率等于滴答率。上下文切换的发生频率可能​​高于系统刻度（例如，如果任务阻止了）。发生这种情况时，跟踪将显示发生了上下文切换，并将准确显示上下文切换的顺序。但是，无法准确记录在系统滴答之间发生的上下文切换的时间。通过使用自由运行的计时器，可以轻松地修改端口以提供更高分辨率的时间戳。

# RTOS实施

本节描述了FreeRTOS实现的一部分。

如果您执行以下操作，这些页面将很有帮助：

* 希望修改FreeRTOS源代码。
* 将实时内核移植到另一个微控制器或原型板。
* 是使用RTOS的新手，希望获得有关其操作和实现的更多信息。

从左侧的菜单框中可以访问两章：

1. *–基本原理和实时操作系统概念*

它包含有关多任务处理和基本实时概念的背景信息，仅供初学者使用。

1. *– RTOS实施*

这从下至上解释了实时内核上下文切换源代码。

FreeRTOS实时内核已被移植到许多不同的微控制器架构中。由于以下原因，本示例选择了Atmel AVR端口：

* + [AVR](http://www.microchip.com/wwwproducts/en/atmega32)架构的简单性。
  + 免费使用[WinAVR（GCC）开发工具](http://winavr.sourceforge.net/)。
  + [STK500原型板的](http://www.microchip.com/developmenttools/productdetails.aspx?partno=atstk500)低成本

本节最后详细介绍了一个完整的上下文切换。

# RTOS基础知识

## 页数

* [**多任务**](https://www.freertos.org/implementation/a00004.html)
* [**排程**](https://www.freertos.org/implementation/a00005.html)
* [**上下文切换**](https://www.freertos.org/implementation/a00006.html)
* [**实时应用**](https://www.freertos.org/implementation/a00007.html)
* [**实时调度**](https://www.freertos.org/implementation/a00008.html)

## 详细说明

本节简要介绍了实时和多任务处理概念。在阅读第2节之前，必须先理解这些内容。

# 多任务 [ [RTOS基础知识](https://www.freertos.org/implementation/a00002.html)]

该**内核**是在操作系统内的核心部件。诸如Linux之类的操作系统采用允许用户看似同时访问计算机的内核。多个用户显然可以同时执行多个程序。

每个执行程序都是在操作系统控制下的**任务**（或线程）。如果操作系统可以这种方式执行多个任务，则称其为**多任务处理**。

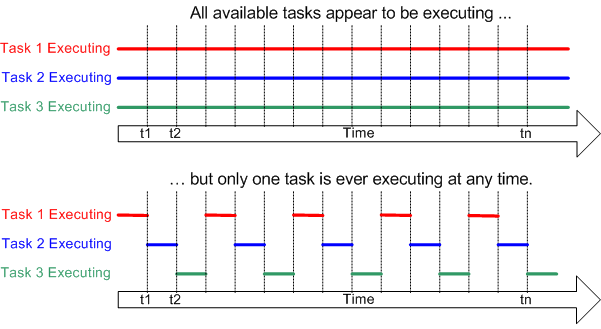
使用多任务操作系统可以简化复杂软件应用程序的设计：

* 操作系统的多任务和任务间通信功能允许将复杂的应用程序划分为一组更小且更易于管理的任务。
* 分区可以简化软件测试，团队内部的工作分解以及代码重用。
* 复杂的时序和排序细节可以从应用程序代码中删除，并成为操作系统的责任。

另请参见常见问题解答“[为什么要使用RTOS？](https://www.freertos.org/FAQWhat.html#WhyUseRTOS)”。

**多任务与并发**

传统处理器一次只能执行一个任务-但是通过在任务之间快速切换，多任务操作系统可以使其**看起来**好像每个任务正在同时执行。下图描述了这三个任务相对于时间的执行模式。任务名称用颜色编码并写在左侧。时间从左到右移动，彩色线显示在任何特定时间正在执行的任务。上面的图展示了感知到的并发执行模式，下面的图展示了实际的多任务执行模式。

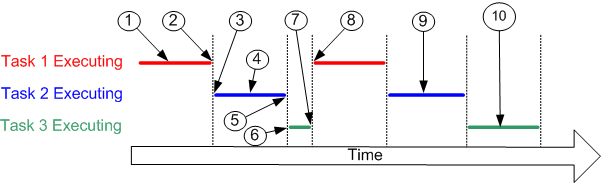


# 计划 [ [RTOS基础知识](https://www.freertos.org/implementation/a00002.html)]

该**调度**负责决定哪一个任务应该在任何特定时间被执行内核的一部分。内核可以在任务生命周期中挂起许多任务，然后再继续执行任务。

该**调度策略**是使用由调度来决定在任何时间点，可以执行哪个任务的算法。（非实时）多用户系统的策略很可能会允许每个任务在处理器时间上“公平”地分配一部分。稍后将描述实时/嵌入式系统中使用的策略。

除了被内核非自愿挂起外，任务还可以选择自行挂起。如果它想要延迟（**睡眠**）一段固定的时间，或者等待（**阻止**）资源变得可用（例如，串行端口）或发生事件（例如，按下按键），它将执行此操作。阻塞或休眠的任务无法执行，并且不会分配任何处理时间。

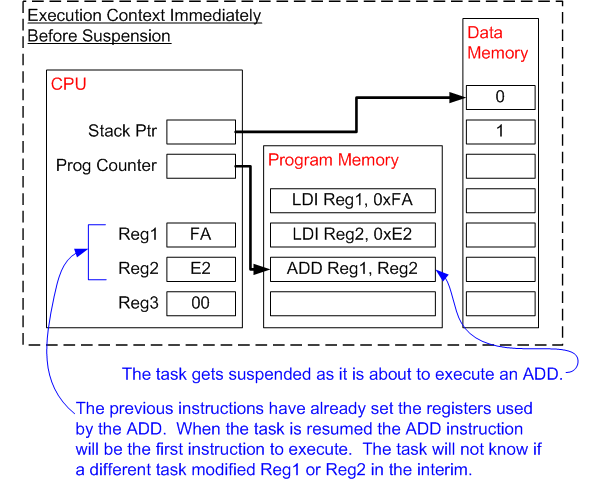


参考上图中的数字：

* 在（1），任务1正在执行。
* 在（2）处，内核挂起（交换）任务1…
* …，在（3）处继续执行任务2。
* 在执行任务2（4）时，它锁定处理器外围设备以进行其自己的独占访问。
* 在（5），内核挂起任务2…
* …，在（6）处继续执行任务3。
* 任务3尝试访问同一处理器外围设备，发现它被锁定了，任务3无法继续执行，因此在（7）处将其自身挂起。
* 在（8），内核恢复任务1。
* 等等。
* 下次执行任务2（9）时，它以处理器外围设备结束并对其进行解锁。
* 下次执行任务3（10）时，它发现它现在可以访问处理器外围设备，并且这次执行直到被内核挂起为止。

# 上下文切换 [ [RTOS基础知识](https://www.freertos.org/implementation/a00002.html)]

在执行任务时，它利用处理器/微控制器寄存器并像访问其他程序一样访问RAM和ROM。这些资源（处理器寄存器，堆栈等）一起构成任务执行**上下文**。



任务是一段顺序的代码–它不知道内核何时将其挂起（交换或切出）或恢复（交换或切入），甚至不知道何时发生。考虑一个示例，该示例紧接在执行对两个处理器寄存器中包含的值求和的指令之前被挂起。当任务挂起时，其他任务将执行，并且可能会修改处理器寄存器的值。恢复后，任务将不知道处理器寄存器已被更改-如果它使用修改后的值，则求和将导致错误的值。

为防止此类错误，至关重要的是，恢复任务后的上下文必须与暂停前的上下文相同。操作系统内核负责确保是这种情况–并且通过在任务挂起时保存任务的上下文来做到这一点。恢复任务后，操作系统内核会在执行任务之前恢复其保存的上下文。保存挂起的任务的上下文并恢复正在恢复的任务的上下文的过程称为**上下文切换**。

# 实时应用 [ [RTOS基础知识](https://www.freertos.org/implementation/a00002.html)]

实时操作系统（**RTOSes**）使用这些相同的原理实现多任务处理-但它们的目标与非实时系统的目标有很大不同。不同的目标反映在调度策略中。实时/嵌入式系统旨在提供对现实事件的及时响应。现实世界中发生的事件可以有一个截止时间，在这个截止时间之前，实时/嵌入式系统必须做出响应，并且RTOS调度策略必须确保这些截止时间得到满足。

为了实现此目标，软件工程师必须首先为每个任务分配优先级。然后，RTOS的调度策略是简单地确保能够执行的最高优先级任务是给定处理时间的任务。如果它们准备同时运行，则可能需要在优先级相同的任务之间“公平地”共享处理时间。

**例：**

最基本的示例是结合了键盘和LCD的实时系统。用户必须在合理的时间内获得每次按键的视觉反馈–如果用户看不到在此时间内按键被接受，则软件产品充其量是笨拙的。如果最长可接受时间是100ms，则0到100ms之间的任何响应都是可以接受的。此功能可以实现为具有以下结构的自主任务：

**void vKeyHandlerTask( void \*pvParameters )**

**{**

**// Key handling is a continuous process and as such the task**

**// is implemented using an infinite loop (as most real time**

**// tasks are).**

**for( ;; )**

**{**

**[Suspend waiting for a key press]**

**[Process the key press]**

**}**

**}**

现在假设实时系统还执行依赖于数字滤波输入的控制功能。必须对输入进行采样，滤波并每2ms执行一次控制周期。为了使滤波器正确工作，样本的时间规律性必须精确到0.5ms。此功能可以实现为具有以下结构的自主任务：

**void vControlTask( void \*pvParameters )**

**{**

**for( ;; )**

**{**

**[Suspend waiting for 2ms since the start of the previous**

**cycle]**

**[Sample the input]**

**[Filter the sampled input]**

**[Perform control algorithm]**

**[Output result]**

**}**

**}**

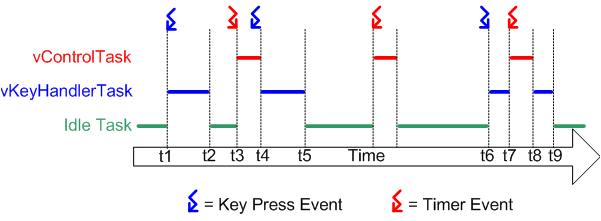
软件工程师必须为控制任务分配最高优先级，例如：

1. 控制任务的截止日期比密钥处理任务的截止日期更严格。
2. 对于控制任务而言，错过最后期限的后果要比对于密钥处理程序任务而言更大。

下一页演示了实时操作系统如何计划这些任务。

# 实时调度 [ [RTOS基础知识](https://www.freertos.org/implementation/a00002.html)]

下图演示了实时操作系统如何调度上一页中定义的任务。RTOS本身已经创建了一个任务-**空闲**任务-仅在没有其他任务可以执行时才执行。RTOS空闲任务始终处于能够执行的状态。



参考上图：

* 在开始时，我们的两个任务都无法运行– vControlTask​​等待正确的时间开始一个新的控制周期，而vKeyHandlerTask等待按键被按下。处理器时间分配给RTOS空闲任务。
* 在时间t1，发生按键。vKeyHandlerTask现在可以执行–它的优先级高于RTOS空闲任务，因此具有处理器时间。
* 在时间t2，vKeyHandlerTask已完成对键的处理并更新了LCD。在按下另一个键之前，它无法继续，因此将其自身挂起，然后再次恢复RTOS空闲任务。
* 在时间t3，计时器事件指示到了执行下一个控制周期的时间。vControlTask​​现在可以执行，并且优先级最高的任务会立即安排在处理器时间中。
* 在时间t3和t4之间，当vControlTask​​仍在执行时，会发生按键。vKeyHandlerTask现在可以执行，但是由于它的优先级比vControlTask​​低，因此没有安排任何处理器时间。
* 在t4时刻，vControlTask​​完成对控制周期的处理，并且直到下一个计时器事件才能重新启动–它会自行挂起。vKeyHandlerTask现在是可以运行的具有最高优先级的任务，因此安排处理器时间来处理上一次按键。
* 在t5处，按键已被处理，并且vKeyHandlerTask暂停自身以等待下一个按键事件。同样，我们两个任务都无法执行，RTOS空闲任务是计划的处理器时间。
* 在t5和t6之间，将处理计时器事件，但不会再发生按键操作。
* 下一次按键发生在时间t6，但是在vKeyHandlerTask完成按键处理之前，会发生计时器事件。现在，两个任务都可以执行。由于vControlTask​​具有更高的优先级，因此vKeyHandlerTask在完成密钥处理之前将被挂起，并且vControlTask​​是计划的处理器时间。
* 在t8时刻，vControlTask​​完成对控制周期的处理，并暂停自身以等待下一个周期。vKeyHandlerTask还是可以运行的优先级最高的任务，因此安排了处理器时间，因此可以完成按键处理。

# RTOS实施

## 页数

* [**建筑模块**](https://www.freertos.org/implementation/a00009.html)
* [**详细的例子**](https://www.freertos.org/implementation/a00018.html)

## 详细说明

本节从下至上描述了RTOS上下文切换源代码。以FreeRTOS Atmel AVR微控制器端口为例。本节最后详细介绍了一个完整的上下文切换。

# 构件 [ [RTOS实施](https://www.freertos.org/implementation/a00003.html)]

## 页数

* [**开发工具**](https://www.freertos.org/implementation/a00010.html)
* [**RTOS刻度**](https://www.freertos.org/implementation/a00011.html)
* [**WinAVR信号属性**](https://www.freertos.org/implementation/a00012.html)
* [**WinAVR裸属性**](https://www.freertos.org/implementation/a00013.html)
* [**FreeRTOS勾号代码**](https://www.freertos.org/implementation/a00014.html)
* [**AVR上下文**](https://www.freertos.org/implementation/a00015.html)
* [**保存上下文**](https://www.freertos.org/implementation/a00016.html)
* [**恢复上下文**](https://www.freertos.org/implementation/a00017.html)

# C开发工具 [ [RTOS实施构建块](https://www.freertos.org/implementation/a00009.html)]

FreeRTOS的目标是使其简单易懂。为此，大多数RTOS源代码都是用C而不是汇编器编写的。

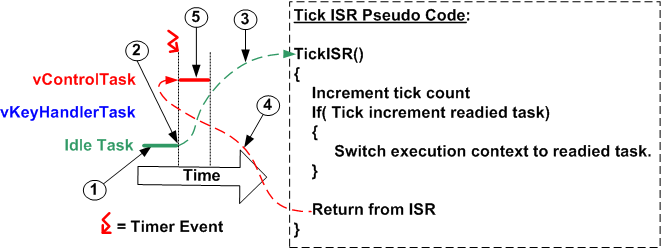
此处提供的示例使用[WinAVR开发工具](http://sourceforge.net/projects/winavr)。WinAVR是基于GCC的免费Windows到AVR交叉编译器。

# RTOS标记 [ [RTOS实施构建块](https://www.freertos.org/implementation/a00009.html)]

休眠时，RTOS任务将指定一个时间，在此之后需要“唤醒”。阻止时，RTOS任务可以指定希望等待的最长时间。

FreeRTOS实时内核使用**刻度**计数变量来测量时间。计时器中断（RTOS**滴答中断**）以严格的时间精度来增加滴答计数，从而使实时内核能够以选定的计时器中断频率的分辨率来测量时间。

每次滴答计数增加时，实时内核必须检查以查看是否现在应该解除阻塞或唤醒任务。在滴答ISR期间唤醒或取消阻止的任务的优先级可能会高于被中断任务的优先级。在这种情况下，滴答ISR应该返回到新唤醒/未阻止的任务，从而有效地中断了一项任务，但又返回到另一项任务。如下所示：



参考上图中的数字：

* 在（1），RTOS空闲任务正在执行。
* 在（2）处发生RTOS滴答，控制权转移到滴答ISR（3）。
* RTOS滴答ISR使vControlTask​​准备就绪，并且由于vControlTask​​的优先级高于RTOS空闲任务，因此将上下文切换到vControlTask​​的上下文。
* 由于执行上下文现在是vControlTask​​的执行上下文，因此退出ISR（4）将控制权返回给vControlTask​​，vControlTask​​开始执行（5）。

以这种方式发生的上下文切换被称为**抢占式**，因为被中断的任务被抢占而不会自动暂停。

FreeRTOS的AVR端口使用定时器1上的比较匹配事件来生成RTOS滴答。以下页面描述了如何使用WinAVR开发工具来实现RTOS滴答ISR。

# GCC信号属性 [ [RTOS实施构建块](https://www.freertos.org/implementation/a00009.html)]

该[GCC开发工具](http://gcc.gnu.org/)允许写入在C.甲中断比较匹配事件AVR计时器1周可以使用以下句法来编写。

**void SIG\_OUTPUT\_COMPARE1A( void ) \_\_attribute\_\_ ( ( signal ) );**

**void SIG\_OUTPUT\_COMPARE1A( void )**

**{**

**/\* ISR C code for RTOS tick. \*/**

**vPortYieldFromTick();**

**}**

的*“\_\_attribute\_\_ ( ( signal ) )”*上的函数原型INFORMS指令编译器的功能是在编译器输出的两个重要的变化一个ISR和结果。

1. “信号”属性可确保在ISR退出时将在ISR期间修改的每个处理器寄存器恢复到其原始值。这是必需的，因为编译器无法对何时执行中断做出任何假设，因此无法优化哪些处理器寄存器需要保存，哪些不需要。
2. “信号”属性还强制使用“从中断返回”指令（RETI）代替原本会使用的“返回”指令（RET）。AVR微控制器在输入ISR时会禁用中断，并且需要RETI指令才能在退出时重新启用它们。

编译器输出的代码：

**;void SIG\_OUTPUT\_COMPARE1A( void )**

**;{**

**; ---------------------------------------**

**; CODE GENERATED BY THE COMPILER TO SAVE**

**; THE REGISTERS THAT GET ALTERED BY THE**

**; APPLICATION CODE DURING THE ISR.**

**PUSH R1**

**PUSH R0**

**IN R0,0x3F**

**PUSH R0**

**CLR R1**

**PUSH R18**

**PUSH R19**

**PUSH R20**

**PUSH R21**

**PUSH R22**

**PUSH R23**

**PUSH R24**

**PUSH R25**

**PUSH R26**

**PUSH R27**

**PUSH R30**

**PUSH R31**

**; ---------------------------------------**

**; CODE GENERATED BY THE COMPILER FROM THE**

**; APPLICATION C CODE.**

**;vPortYieldFromTick();**

**CALL 0x0000029B ;Call subroutine**

**;}**

**; ---------------------------------------**

**; CODE GENERATED BY THE COMPILER TO**

**; RESTORE THE REGISTERS PREVIOUSLY**

**; SAVED.**

**POP R31**

**POP R30**

**POP R27**

**POP R26**

**POP R25**

**POP R24**

**POP R23**

**POP R22**

**POP R21**

**POP R20**

**POP R19**

**POP R18**

**POP R0**

**OUT 0x3F,R0**

**POP R0**

**POP R1**

**RETI**

**; ---------------------------------------**

# GCC Naked属性 [ [RTOS实施构建块](https://www.freertos.org/implementation/a00009.html)]

上一节展示了如何使用“signal”属性在C语言中编写ISR，以及这如何导致部分执行上下文被自动保存（仅保存由ISR修改的处理器寄存器）。但是，执行上下文切换需要保存整个上下文。

应用程序代码可以在进入ISR时显式保存所有处理器寄存器，但是这样做将导致某些处理器寄存器被保存两次-一次由编译器生成的代码保存，然后由应用程序代码再次保存。这是不希望的，可以通过使用除“signal”属性之外的*“naked”*属性来避免。

**void SIG\_OUTPUT\_COMPARE1A( void ) \_\_attribute\_\_ ( ( signal, naked ) );**

**void SIG\_OUTPUT\_COMPARE1A( void )**

**{**

**/\* ISR C code for RTOS tick. \*/**

**vPortYieldFromTick();**

**}**

“ naked”属性可防止编译器生成任何函数入口或出口代码。现在，编译代码将产生更简单的输出：

**;void SIG\_OUTPUT\_COMPARE1A( void )**

**;{**

**; ---------------------------------------**

**; NO COMPILER GENERATED CODE HERE TO SAVE**

**; THE REGISTERS THAT GET ALTERED BY THE**

**; ISR.**

**; ---------------------------------------**

**; CODE GENERATED BY THE COMPILER FROM THE**

**; APPLICATION C CODE.**

**;vTaskIncrementTick();**

**CALL 0x0000029B ;Call subroutine**

**; ---------------------------------------**

**; NO COMPILER GENERATED CODE HERE TO RESTORE**

**; THE REGISTERS OR RETURN FROM THE ISR.**

**; ---------------------------------------**

**;}**

使用'naked'属性时，编译器不会生成*任何*函数入口或出口代码，因此现在必须显式添加。RTOS宏portSAVE\_CONTEXT（）和portRESTORE\_CONTEXT（）分别保存和还原整个执行上下文。

**void SIG\_OUTPUT\_COMPARE1A( void ) \_\_attribute\_\_ ( ( signal, naked ) );**

**void SIG\_OUTPUT\_COMPARE1A( void )**

**{**

**/\* Macro that explicitly saves the execution**

**context. \*/**

**portSAVE\_CONTEXT();**

**/\* ISR C code for RTOS tick. \*/**

**vPortYieldFromTick();**

**/\* Macro that explicitly restores the**

**execution context. \*/**

**portRESTORE\_CONTEXT();**

**/\* The return from interrupt call must also**

**be explicitly added. \*/**

**asm volatile ( "reti" );**

**}**

“naked”属性使应用程序代码可以完全控制何时以及如何保存AVR上下文。如果应用程序代码在输入ISR时保存了整个上下文，则无需在执行上下文切换之前再次保存它，因此没有一个处理器寄存器被保存两次。

# FreeRTOS标记代码 [ [RTOS实施构建块](https://www.freertos.org/implementation/a00009.html)]

FreeRTOS AVR端口使用的实际源代码与前一页中显示的示例略有不同。vPortYieldFromTick（）本身被实现为“裸”函数，并且上下文在vPortYieldFromTick（）中保存和恢复。由于实现了非抢先的上下文切换（任务自身阻塞），因此可以通过这种方式完成此操作，此处不再赘述。

因此，RTOS刻度的FreeRTOS实现是（有关更多详细信息，请参见源代码片段中的注释）：

**void SIG\_OUTPUT\_COMPARE1A( void ) \_\_attribute\_\_ ( ( signal, naked ) );**

**void vPortYieldFromTick( void ) \_\_attribute\_\_ ( ( naked ) );**

**/\*--------------------------------------------------\*/**

**/\* Interrupt service routine for the RTOS tick. \*/**

**void SIG\_OUTPUT\_COMPARE1A( void )**

**{**

**/\* Call the tick function. \*/**

**vPortYieldFromTick();**

**/\* Return from the interrupt. If a context**

**switch has occurred this will return to a**

**different task. \*/**

**asm volatile ( "reti" );**

**}**

**/\*--------------------------------------------------\*/**

**void vPortYieldFromTick( void )**

**{**

**/\* This is a naked function so the context**

**is saved. \*/**

**portSAVE\_CONTEXT();**

**/\* Increment the tick count and check to see**

**if the new tick value has caused a delay**

**period to expire. This function call can**

**cause a task to become ready to run. \*/**

**vTaskIncrementTick();**

**/\* See if a context switch is required.**

**Switch to the context of a task made ready**

**to run by vTaskIncrementTick() if it has a**

**priority higher than the interrupted task. \*/**

**vTaskSwitchContext();**

**/\* Restore the context. If a context switch**

**has occurred this will restore the context of**

**the task being resumed. \*/**

**portRESTORE\_CONTEXT();**

**/\* Return from this naked function. \*/**

**asm volatile ( "ret" );**

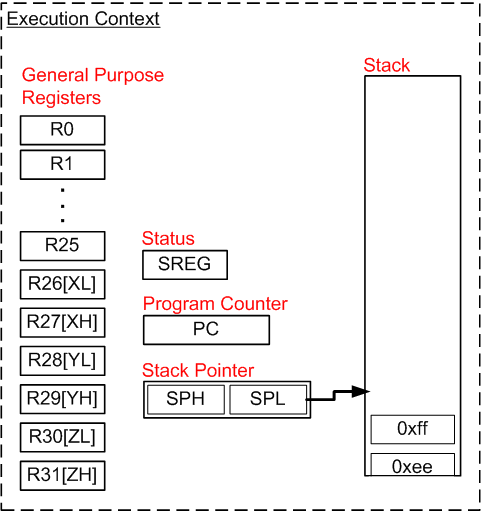
**}**

**/\*--------------------------------------------------\*/**

# AVR上下文 [ [RTOS实施构建块](https://www.freertos.org/implementation/a00009.html)]

上下文切换要求保存整个执行上下文。在AVR微控制器上，上下文包括：

* 32个通用处理器寄存器。gcc开发工具假定寄存器R1设置为零。
* 状态寄存器。状态寄存器的值会影响指令执行，并且必须在上下文切换中保留。
* 程序计数器。恢复后，任务必须从即将挂起之前要执行的指令继续执行。
* 两个堆栈指针寄存器。



# Saving the RTOS Task Context [[RTOS Implementation Building Blocks](https://www.freertos.org/implementation/a00009.html)]

Each real time task has its own stack memory area so the context can be saved by simply pushing processor registers onto the task stack. Saving the AVR context is one place where assembly code is unavoidable.

portSAVE\_CONTEXT() is implemented as a macro, the source code for which is given below:

**#define portSAVE\_CONTEXT()**

**asm volatile (**

**"push r0 nt" (1)**

**"in r0, \_\_SREG\_\_ nt" (2)**

**"cli nt" (3)**

**"push r0 nt" (4)**

**"push r1 nt" (5)**

**"clr r1 nt" (6)**

**"push r2 nt" (7)**

**"push r3 nt"**

**"push r4 nt"**

**"push r5 nt"**

**:**

**:**

**:**

**"push r30 nt"**

**"push r31 nt"**

**"lds r26, pxCurrentTCB nt" (8)**

**"lds r27, pxCurrentTCB + 1 nt" (9)**

**"in r0, \_\_SP\_L\_\_ nt" (10)**

**"st x+, r0 nt" (11)**

**"in r0, \_\_SP\_H\_\_ nt" (12)**

**"st x+, r0 nt" (13)**

**);**

参考上面的源代码：

* 保存状态寄存器时，将首先保存处理器寄存器R0，因为它必须被保存为原始值。
* 状态寄存器被移到R0（2）中，因此可以保存到堆栈（4）中。
* 处理器中断被禁用（3）。如果仅从ISR内部调用portSAVE\_CONTEXT（），则无需显式禁用中断，因为AVR已经这样做。由于portSAVE\_CONTEXT（）宏也在中断服务例程之外使用（任务挂起时），因此必须尽早明确清除中断。
* 编译器从ISR C源代码生成的代码假定R1设置为零。R1的原始值在清除（1）之前被保存（5）。
* 在（7）和（8）之间，所有其余处理器寄存器均按数字顺序保存。
* 现在，挂起的任务堆栈包含任务执行上下文的副本。内核存储任务堆栈指针，以便在恢复任务时可以检索和还原上下文。X处理器寄存器加载了堆栈指针要保存到的地址（8和9）。
* 堆栈指针被保存，首先是低字节（10和11），然后是高半字节（12和13）。

# 恢复上下文 [ [RTOS实施构建块](https://www.freertos.org/implementation/a00009.html)]

RTOS宏portRESTORE\_CONTEXT（）与portSAVE\_CONTEXT（）相反。要恢复的任务的上下文先前已存储在任务堆栈中。实时内核检索任务的堆栈指针，然后将POP的上下文返回到正确的处理器寄存器中。

**#define portRESTORE\_CONTEXT()**

**asm volatile (**

**"lds r26, pxCurrentTCB nt" (1)**

**"lds r27, pxCurrentTCB + 1 nt" (2)**

**"ld r28, x+ nt"**

**"out \_\_SP\_L\_\_, r28 nt" (3)**

**"ld r29, x+ nt"**

**"out \_\_SP\_H\_\_, r29 nt" (4)**

**"pop r31 nt"**

**"pop r30 nt"**

**:**

**:**

**:**

**"pop r1 nt"**

**"pop r0 nt" (5)**

**"out \_\_SREG\_\_, r0 nt" (6)**

**"pop r0 nt" (7)**

**);**

参考上面的代码：

* FreeRTOS pxCurrentTCB变量保存可从中检索任务堆栈指针的地址。它被加载到X寄存器（1和2）中。
* 将要恢复的任务的堆栈指针加载到AVR堆栈指针中，首先是低字节（3），然后是高半字节（4）。
* 然后以相反的数字顺序从堆栈弹出处理器寄存器，直到R1。
* 状态寄存器存储在寄存器R1和R0之间的堆栈中，因此在R0（7）之前被恢复（6）。

# 详细示例 [ [RTOS实施](https://www.freertos.org/implementation/a00003.html)]

## 页数

* [**放在一起**](https://www.freertos.org/implementation/a00019.html)
* [**第1步**](https://www.freertos.org/implementation/a00020.html)
* [**第2步**](https://www.freertos.org/implementation/a00021.html)
* [**第三步**](https://www.freertos.org/implementation/a00022.html)
* [**步骤4**](https://www.freertos.org/implementation/a00023.html)
* [**第5步**](https://www.freertos.org/implementation/a00024.html)
* [**第6步**](https://www.freertos.org/implementation/a00025.html)
* [**步骤7**](https://www.freertos.org/implementation/a00026.html)

## 详细说明

第2部分的最后一部分显示了如何使用这些构造块和源代码模块在AVR微控制器上实现上下文切换。该示例分七个步骤演示了从优先级较低的任务TaskA切换到优先级较高的任务TaskB的过程。

源代码与WinAVR开发工具兼容。

# 全部放在一起 [[详细示例](https://www.freertos.org/implementation/a00018.html)]

第2部分的最后一部分显示了如何使用这些构造块和源代码模块在AVR微控制器上实现RTOS上下文切换。该示例分七个步骤演示了从优先级较低的任务TaskA切换到优先级较高的任务TaskB的过程。

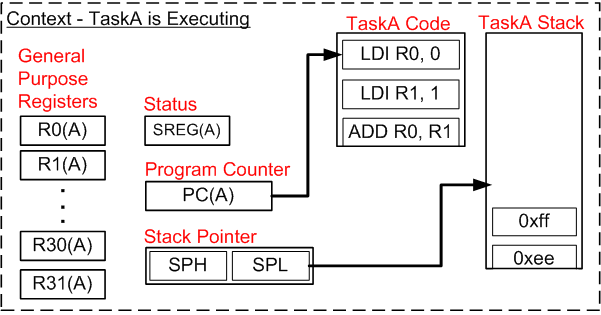
源代码与WinAVR C开发工具兼容。

# RTOS上下文切换–步骤1 [[详细示例](https://www.freertos.org/implementation/a00018.html)]

## 在RTOS滴答中断之前

本示例从TaskA执行开始。TaskB之前已被挂起，因此其上下文已经存储在TaskB堆栈中。

TaskA具有下图所示的上下文。

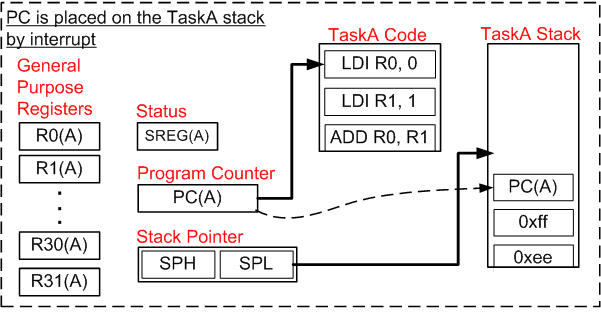


每个寄存器中的（A）标签表明该寄存器包含任务A上下文的正确值。

# RTOS上下文切换–步骤2 [[详细示例](https://www.freertos.org/implementation/a00018.html)]

## 发生RTOS滴答中断

就像TaskA即将执行LDI指令一样，发生RTOS滴答。发生中断时，AVR微控制器会在跳转到RTOS滴答ISR的开始之前自动将当前程序计数器（PC）放入堆栈。



# RTOS上下文切换–步骤3 [[详细示例](https://www.freertos.org/implementation/a00018.html)]

## RTOS滴答中断执行

ISR源代码如下。注释已删除，以方便阅读，但可以在上一页中查看。

**/\* Interrupt service routine for the RTOS tick. \*/**

**void SIG\_OUTPUT\_COMPARE1A( void )**

**{**

**vPortYieldFromTick();**

**asm volatile ( "reti" );**

**}**

**/\*--------------------------------------------------\*/**

**void vPortYieldFromTick( void )**

**{**

**portSAVE\_CONTEXT();**

**vTaskIncrementTick();**

**vTaskSwitchContext();**

**portRESTORE\_CONTEXT();**

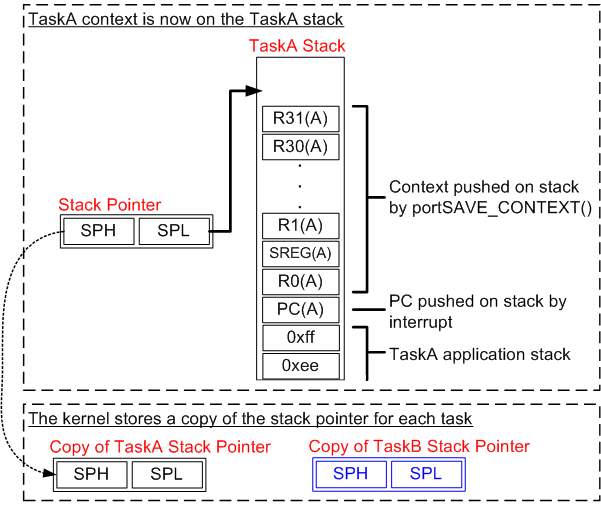
**asm volatile ( "ret" );**

**}**

**/\*--------------------------------------------------\*/**

SIG\_OUTPUT\_COMPARE1A（）是裸函数，因此第一条指令是对vPortYieldFromTick（）的调用。vPortYieldFromTick（）也是一个裸函数，因此通过调用portSAVE\_CONTEXT（）显式保存AVR执行上下文。

portSAVE\_CONTEXT（）将整个AVR执行上下文推送到TaskA堆栈中，从而得到如下所示的堆栈。TaskA的堆栈指针现在指向其自身上下文的顶部。portSAVE\_CONTEXT（）通过存储堆栈指针的副本来完成。实时内核已经具有TaskB堆栈指针的副本–是在上次TaskB挂起时获取的。



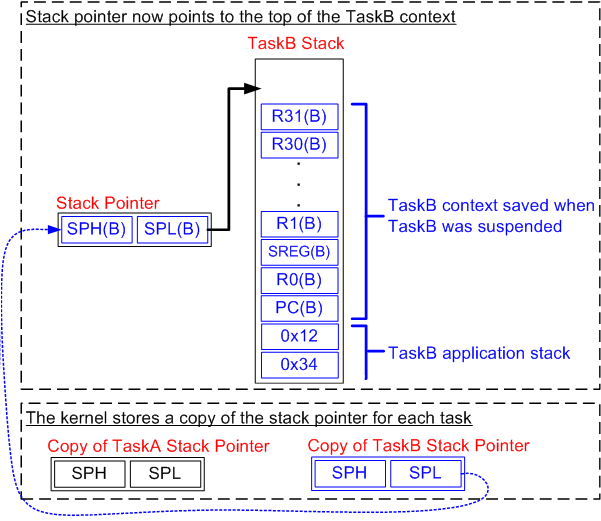
# RTOS上下文切换–步骤4 [[详细示例](https://www.freertos.org/implementation/a00018.html)]

## 增加刻度线计数

保存TaskA上下文后，将执行RTOS函数vTaskIncrementTick（）。出于本示例的目的，假定滴答计数的增加已导致TaskB准备运行。TaskB具有比TaskA更高的优先级，因此当ISR完成时，vTaskSwitchContext（）选择TaskB作为要给予处理时间的任务。

# RTOS上下文切换–步骤5 [[详细示例](https://www.freertos.org/implementation/a00018.html)]

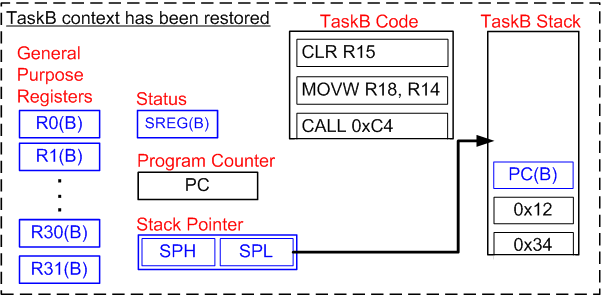
## 检索TaskB堆栈指针



必须还原TaskB上下文。RTOS宏portRESTORE\_CONTEXT要做的第一件事是从TaskB挂起时获取的副本中检索TaskB堆栈指针。TaskB堆栈指针已加载到处理器堆栈指针中，因此现在AVR堆栈指向TaskB上下文的顶部。

# RTOS上下文切换–步骤6 [[详细示例](https://www.freertos.org/implementation/a00018.html)]

## 恢复TaskB上下文



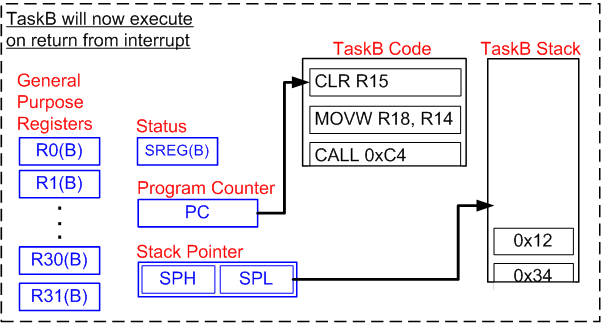
portRESTORE\_CONTEXT（）通过将TaskB上下文从其堆栈还原到适当的处理器寄存器中来完成。

只有程序计数器保留在堆栈上。

# RTOS上下文切换–步骤7 [[详细示例](https://www.freertos.org/implementation/a00018.html)]

## RTOS刻度线退出

vPortYieldFromTick（）返回到SIG\_OUTPUT\_COMPARE1A（），其中最后一条指令是从中断（RETI）返回。RETI指令假定堆栈上的下一个值是发生中断时放置在堆栈上的返回地址。



当RTOS滴答中断开始时，AVR会自动将TaskA返回地址放到堆栈中-下**一条**要在**TaskA中**执行的指令的地址。RTOS滴答处理程序更改了堆栈指针，因此它现在指向**TaskB**堆栈。因此，RETI指令从堆栈返回的返回地址POP'实际上是**TaskB**在挂起之前将要执行的指令的地址。

RTOS滴答中断中断了**TaskA**，但返回到**TaskB** –上下文切换完成！

# 在小型嵌入式系统中使用FreeRTOS进行实时应用设计的教程

**如果您是直接来这里的，那么先开始：**

1. **查看**[**主页**](https://www.freertos.org/index.html)**以将其置于上下文中**
2. **查看**[**什么是FreeRTOS**](https://www.freertos.org/RTOS.html)**页面以查看FreeRTOS适合的位置**

**<<< |**[**>>>**](https://www.freertos.org/tutorial/solution1.html)

**提示：**使用**<<<**和**>>>**箭头浏览此部分。

## 介绍

本节提供了有关编写在内存受限的微控制器上使用RTOS的应用程序的教程。它不是FreeRTOS特定的教程。如果您正在寻找特定的FreeRTOS教程，或者是有关在嵌入式系统中使用RTOS的更完整的教程，那么 [FreeRTOS书籍](https://www.freertos.org/Documentation/RTOS_book.html) 将是更有价值的资源。

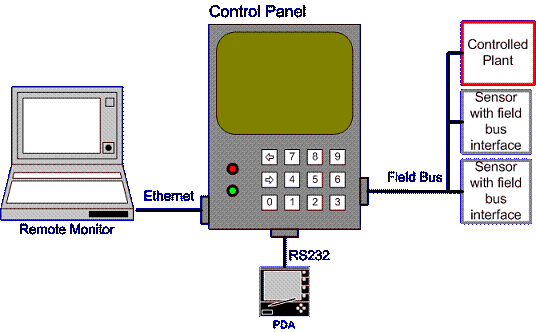
该网站的这一部分为假设的嵌入式实时应用程序提供了四种对比设计解决方案。判断每种解决方案是否适用于具有不同RAM，ROM和处理能力的嵌入式计算机。此外，还评估了每个设计的简单性和相应的可维护性。

这并不是要提供所有可能设计的详尽列表，而是有关FreeRTOS实时内核使用方式的指南。

应该注意的是，本节是几年前编写的，当时FreeRTOS主要用于非常小的微控制器。从那时起，在更大的微控制器上使用FreeRTOS变得越来越普遍，而这些微控制器不受它们提供的ROM和RAM的限制。

**注意：**自FreeRTOS V4.0.0引入以来，这些页面尚未更新。V4.0.0引入了协程的概念，它将为此处提供的解决方案提供不同而新颖的解决方案。该[任务和协程](https://www.freertos.org/taskandcr.html)文档提供了进一步的信息。

## [假设的]应用

  
**系统上下文**[不按比例缩放]。

该应用程序将在嵌入式单板计算机上执行，该计算机必须控制工厂，同时维护本地和远程用户界面。

上面已描述，该系统包括：

1. 控制终端内的嵌入式计算机。
2. 两个现场总线联网传感器。
3. 被控制的工厂（可以是任何东西，电动机，加热器等）。它连接在同一现场总线网络上。
4. 使用通用IO扫描的矩阵键盘。
5. 两个LED指示灯。
6. LCD显示器。
7. 远程监视计算机可以连接到的嵌入式Web服务器。
8. 与PDA上运行的配置实用程序的RS232接口。

## 顶级软件要求

在这里，我们对排序和时序要求感兴趣，而不是确切的功能要求。

### 工厂控制

每个控制周期应执行以下顺序：

1. 在现场总线上发送帧以从联网的传感器请求数据。
2. 等待接收来自两个传感器的数据。
3. 执行控制算法。
4. 向工厂发送命令。

嵌入式计算机的控制功能应每10ms精确地发送一个请求，并且结果命令应在该请求的5ms内发送。控制算法依赖于精确的时序，因此满足这些时序要求至关重要。

### 本地操作员界面[键盘和LCD]

操作员可以使用键盘和LCD来选择，查看和修改系统数据。在控制工厂时，操作员界面应起作用。

为确保不会遗漏按键，键盘至少应每15毫秒扫描一次。LCD应该在按键按下后的50ms内更新。

### LED

LED将用于指示系统状态。绿色LED闪烁应指示系统正在按预期运行。红色LED闪烁应指示故障状态。

正确的LED指示灯应每秒闪烁一次。该闪烁速率应保持在50ms以内。

### RS232 PDA接口

PDA RS232接口应能够查看和访问与本地操作员接口相同的数据，并且应用相同的时序约束-减少任何数据传输时间。

### TCP / IP接口

嵌入式Web服务器应在一秒钟内处理HTTP请求。

## 应用组件

假设系统的时序要求可以分为三类：

1. **严格的时间安排**–工厂控制

控制功能有非常严格的时序要求，因为它必须每10ms执行一次。

1. **灵活的计时**– LED

虽然LED输出具有最大和最小时间限制，但在较大的时序范围内它们可以工作。

1. **仅截止日期的计时**–人机界面

这包括键盘，LCD，RS232和TCP / IP以太网通信。

人机界面功能具有不同类型的时序要求，因为仅指定了最大限制。例如，必须至少每10ms扫描一次键盘，但可接受10ms以下的任何速率。

# 解决方案＃1 为什么使用RTOS内核？

[**<<<**](https://www.freertos.org/tutorial/index.html)**|**[**>>>**](https://www.freertos.org/tutorial/solution2.html)

另请参阅常见问题解答项目“[为什么使用RTOS？](https://www.freertos.org/FAQWhat.html#WhyUseRTOS)”。

## 概要

无需使用RTOS内核即可生成许多应用程序，并且此页面介绍了可能采用的方法。

即使这种情况下的应用程序对于这种方法来说可能太复杂了，但仍包含该页面以突出潜在的问题并与以下基于RTOS的软件设计形成对比。

## 实作

该解决方案使用传统的无限循环方法，由此应用程序的每个组件都由一个执行完毕的函数表示。

理想情况下，将使用硬件计时器来安排时间紧迫的工厂控制功能。但是，必须等待数据的到来和执行复杂的计算，使得控制功能不适合在中断服务程序中执行。

### 经营理念

可以修改在无限循环内调用组件的频率和顺序，以引入一些优先级。在下面的示例中提供了几个这样的测序方法。

### 调度程序配置

不使用RTOS调度程序。

### 结论

简单的循环方法非常适合小型应用程序和具有灵活时序要求的应用程序，但是如果扩展到大型系统，则可能变得复杂，难以分析且难以维护。

## 例

此示例是先前介绍的假设应用程序的部分实现。

### 工厂控制功能

控制功能可以由以下伪代码表示：

**void PlantControlCycle( void )**

**{**

**TransmitRequest();**

**WaitForFirstSensorResponse();**

**if( Got data from first sensor )**

**{**

**WaitForSecondSensorResponse();**

**if( Got data from second sensor )**

**{**

**PerformControlAlgorithm();**

**TransmitResults();**

**}**

**}**

**}**

**人机界面功能**

这包括键盘，LCD，RS232通信和嵌入式Web服务器。

以下伪代码表示用于控制这些接口的简单无限循环结构。

**int main( void )**

**{**

**Initialise();**

**for( ;; )**

**{**

**ScanKeypad();**

**UpdateLCD();**

**ProcessRS232Characters();**

**ProcessHTTPRequests();**

**}**

**// Should never get here.**

**return 0;**

**}**

这假定了两件事：首先，通信IO由中断服务例程缓冲，因此外围设备不需要轮询。其次，循环内的各个函数调用执行得足够快，可以满足所有最大时序要求。

## 调度工厂控制功能

控制功能的长度意味着不能简单地从10ms定时器中断中调用它。

将其添加到无限循环将需要引入一些时间控制。例如 … ：

**// Flag used to mark the time at which a**

**// control cycle should start (mutual exclusion**

**// issues being ignored for this example).**

**int TimerExpired;**

**// Service routine for a timer interrupt. This**

**// is configured to execute every 10ms.**

**void TimerInterrupt( void )**

**{**

**TimerExpired = true;**

**}**

**// Main() still contains the infinite loop -**

**// within which a call to the plant control**

**// function has been added.**

**int main( void )**

**{**

**Initialise();**

**for( ;; )**

**{**

**// Spin until it is time for the next**

**// cycle.**

**if( TimerExpired )**

**{**

**PlantControlCycle();**

**TimerExpired = false;**

**ScanKeypad();**

**UpdateLCD();**

**// The LEDs could use a count of**

**// the number of interrupts, or a**

**// different timer.**

**ProcessLEDs();**

**// Comms buffers must be large**

**// enough to hold 10ms worth of**

**// data.**

**ProcessRS232Characters();**

**ProcessHTTPRequests();**

**}**

**// The processor can be put to sleep**

**// here provided it is woken by any**

**// interrupt.**

**}**

**// Should never get here.**

**return 0;**

**}**

……但这不是可接受的解决方案：

* 现场总线上的延迟或故障会导致工厂控制功能的执行时间增加。接口功能的时序要求很可能会被违反。
* 在每个周期执行所有功能也可能导致违反控制周期时序。
* 执行时间的抖动可能会导致错过周期。例如，当没有收到HTTP请求时，ProcessHTTPRequests（）的执行时间可以忽略不计，但是在为页面提供服务时，执行时间 很长。
* 它不是非常可维护的-它依赖于在最大时间内执行的每个功能。
* 每个周期仅对通信缓冲区进行一次服务，因此其长度必须大于原来的长度。

## 替代结构

可以确定两个因素，这些因素限制了目前为止描述的简单循环结构的适用性。

1. 每个函数调用的长度

允许每个功能全部执行需要很长时间。可以通过将每个功能拆分为多个状态来防止此情况。每个调用仅执行一种状态。以控制功能为例：

**// Define the states for the control cycle function.**

**typdef enum eCONTROL\_STATES**

**{**

**eStart, // Start new cycle.**

**eWait1, // Wait for the first sensor response.**

**eWait2 // Wait for the second sensor response.**

**} eControlStates;**

**void PlantControlCycle( void )**

**{**

**static eControlState eState = eStart;**

**switch( eState )**

**{**

**case eStart :**

**TransmitRequest();**

**eState = eWait1;**

**break;**

**case eWait1;**

**if( Got data from first sensor )**

**{**

**eState = eWait2;**

**}**

**// How are time outs to be handled?**

**break;**

**case eWait2;**

**if( Got data from first sensor )**

**{**

**PerformControlAlgorithm();**

**TransmitResults();**

**eState = eStart;**

**}**

**// How are time outs to be handled?**

**break;**

**}**

**}**

1. 现在，此功能在结构上更加复杂，并带来了进一步的调度问题。随着额外状态的添加，例如处理超时和错误情况，代码本身将变得更加难以理解。
2. 计时器的粒度

较短的计时器间隔将提供更大的灵活性。

将控制功能实现为状态机（这样做可以缩短每个调用的时间）可以允许从计时器中断中调用它。计时器间隔必须足够短，以确保以满足其计时要求的频率调用该函数。此选项充满时间和维护问题。

另外，可以修改无限循环解决方案以在每个循环上调用不同的函数-高优先级控制函数的调用频率更高：

**int main( void )**

**{**

**int Counter = -1;**

**Initialise();**

**// Each function is implemented as a state**

**// machine so is guaranteed to execute**

**// quickly - but must be called often.**

**// Note the timer frequency has been raised.**

**for( ;; )**

**{**

**if( TimerExpired )**

**{**

**Counter++;**

**switch( Counter )**

**{**

**case 0 : ControlCycle();**

**ScanKeypad();**

**break;**

**case 1 : UpdateLCD();**

**break;**

**case 2 : ControlCycle();**

**ProcessRS232Characters();**

**break;**

**case 3 : ProcessHTTPRequests();**

**// Go back to start**

**Counter = -1;**

**break;**

**}**

**TimerExpired = false;**

**}**

**}**

**// Should never get here.**

**return 0;**

**}**

可以通过事件计数器引入更多的智能，从而仅在发生需要服务的事件时才调用较低优先级的功能：

**for( ;; )**

**{**

**if( TimerExpired )**

**{**

**Counter++;**

**// Process the control cycle every other loop.**

**switch( Counter )**

**{**

**case 0 : ControlCycle();**

**break;**

**case 1 : Counter = -1;**

**break;**

**}**

**// Process just one of the other functions. Only process**

**// a function if there is something to do. EventStatus()**

**// checks for events since the last iteration.**

**switch( EventStatus() )**

**{**

**case EVENT\_KEY : ScanKeypad();**

**UpdateLCD();**

**break;**

**case EVENT\_232 : ProcessRS232Characters();**

**break;**

**case EVENT\_TCP : ProcessHTTPRequests();**

**break;**

**}**

**TimerExpired = false;**

**}**

**}**

以这种方式处理事件将减少浪费的CPU周期，但是设计仍然会在控制周期执行的频率上出现抖动。

# 解决方案2 ：完全抢占式系统

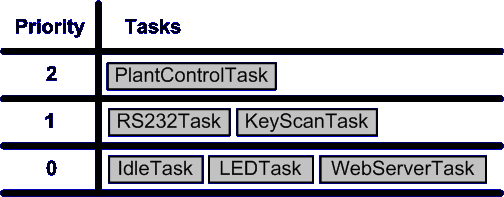
[**<<<**](https://www.freertos.org/tutorial/solution1.html)**|**[**>>>**](https://www.freertos.org/tutorial/solution3.html)

## 概要

这是传统的抢先式多任务解决方案。它充分利用了RTOS服务，而无需考虑由此产生的内存和处理器开销。将所需功能简单地划分为许多自治任务。

## 实作

为系统的每个部分创建一个单独的任务，可以将其识别为能够独立存在或具有特定的计时要求。

  
**解决方案2的任务和优先级**

任务将阻塞，直到某个事件表明需要进行处理为止。事件可以是外部事件（例如，按下某个键），也可以是内部事件（例如，计时器到期）。这种事件驱动的方法意味着不会浪费CPU时间来轮询未发生的事件。

根据任务的时间要求将优先级分配给任务。时序要求越严格，优先级越高（并非所有优先级分配评估都这么简单）。

### 经营理念

能够执行（未被阻止）的最高优先级任务是RTOS保证获得处理器时间的任务。如果更高优先级的任务可用，内核将立即挂起正在执行的任务。

这种调度是自动进行的，在应用程序源代码中没有任何明确的知识，结构或命令。但是，应用程序设计人员有责任确保为任务分配适当的优先级。

当没有任务能够执行时，空闲任务将执行。空闲任务可以选择将处理器置于省电模式。

### 调度程序配置

调度程序配置为抢先操作。内核滴答频率应设置为提供所需时间粒度的最低值。

### 结论

如果可以使用RAM和处理能力，这将是一个很好的解决方案。将应用程序划分为任务并为每个任务分配优先级需要仔细考虑。

## 例

此示例是先前介绍的假设应用程序的部分实现。使用FreeRTOS API。

### 工厂控制任务

此任务实现所有控制功能。它具有严格的时序要求，因此在系统内具有最高的优先级：

**#define CYCLE\_RATE\_MS 10**

**#define MAX\_COMMS\_DELAY 2**

**void PlantControlTask( void \*pvParameters )**

**{**

**TickType\_t xLastWakeTime;**

**DataType Data1, Data2;**

**InitialiseTheQueue();**

**// A**

**xLastWakeTime = xTaskGetTickCount();**

**// B**

**for( ;; )**

**{**

**// C**

**vTaskDelayUntil( &xLastWakeTime, CYCLE\_RATE\_MS );**

**// Request data from the sensors.**

**TransmitRequest();**

**// D**

**if( xQueueReceive( xFieldBusQueue, &Data1, MAX\_COMMS\_DELAY ) )**

**{**

**// E**

**if( xQueueReceive( xFieldBusQueue, &Data2, MAX\_COMMS\_DELAY ) )**

**{**

**PerformControlAlgorithm();**

**TransmitResults();**

**}**

**}**

**}**

**// Will never get here!**

**}**

参考上面代码片段中的标签：

1. xLastWakeTime已初始化。该变量与vTaskDelayUntil（） API函数一起使用，以控制执行控制函数的频率。
2. 此功能作为自主任务执行，因此切勿退出。
3. vTaskDelayUntil（）告诉内核，该任务应在xLastWakeTime中存储的时间之后恰好10ms开始执行。在此时间之前，控制任务将阻止。由于这是系统中最高优先级的任务，因此可以保证在正确的时间再次开始执行。它将抢占正在运行的任何低优先级任务。
4. 从联网传感器请求数据到接收到数据之间有一段有限的时间。到达现场总线的数据通过中断服务例程放置在xFieldBusQueue中，因此控制任务可以在队列上进行阻塞调用以等待数据可用。与以前一样，由于它是系统中优先级最高的任务，因此可以保证在有数据可用时立即继续执行。
5. 作为“ D”，等待来自第二个传感器的数据。

xQueueReceive（）的返回值为0表示在指定的块周期内没有数据到达。这是任务必须处理的错误条件。为了简单起见，已省略了此错误处理功能和其他错误处理功能。

**嵌入式Web服务器任务**

嵌入式Web服务器任务可以由以下伪代码表示。这仅在数据可用时利用处理器时间，但是将花费可变且相对较长的时间才能完成。因此，它具有较低的优先级，以防止对工厂控制，RS232或键盘扫描任务的时间产生不利影响。

**void WebServerTask( void \*pvParameters )**

**{**

**DataTypeA Data;**

**for( ;; )**

**{**

**// Block until data arrives. xEthernetQueue is filled by the**

**// Ethernet interrupt service routine.**

**if( xQueueReceive( xEthernetQueue, &Data, MAX\_DELAY ) )**

**{**

**ProcessHTTPData( Data );**

**}**

**}**

**}**

**RS232接口**

这在结构上与嵌入式Web服务器任务非常相似。优先考虑确保它不会对工厂控制任务的时间产生不利影响。

**void RS232Task( void \*pvParameters )**

**{**

**DataTypeB Data;**

**for( ;; )**

**{**

**// Block until data arrives. xRS232Queue is filled by the**

**// RS232 interrupt service routine.**

**if( xQueueReceive( xRS232Queue, &Data, MAX\_DELAY ) )**

**{**

**ProcessSerialCharacters( Data );**

**}**

**}**

**}**

**键盘扫描任务**

这是一个简单的周期性任务。它的优先级与RS232任务相似，因此具有中等优先级。

设置的循环时间比指定的极限快得多。这是因为这样的事实，即它可能不会在请求后立即获得处理器时间，一旦执行可能会被工厂控制任务抢占。

**#define DELAY\_PERIOD 4**

**void KeyScanTask( void \*pvParmeters )**

**{**

**char Key;**

**TickType\_t xLastWakeTime;**

**xLastWakeTime = xTaskGetTickCount();**

**for( ;; )**

**{**

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

**vTaskDelayUntil( &xLastWakeTime, DELAY\_PERIOD );**

**// Scan the keyboard.**

**if( KeyPressed( &Key ) )**

**{**

**UpdateDisplay( Key );**

**}**

**}**

**}**

如果整个系统的时间安排可以使其成为最低优先级的任务，则可以完全删除对vTaskDelayUntil（）的调用。每当所有较高优先级的任务被阻止时，按键扫描功能将连续执行-有效地代替了空闲任务。

**LED任务**

这是所有任务中最简单的。

**#define DELAY\_PERIOD 1000**

**void LEDTask( void \*pvParmeters )**

**{**

**TickType\_t xLastWakeTime;**

**xLastWakeTime = xTaskGetTickCount();**

**for( ;; )**

**{**

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

**vTaskDelayUntil( &xLastWakeTime, DELAY\_PERIOD );**

**// Flash the appropriate LED.**

**if( SystemIsHealthy() )**

**{**

**FlashLED( GREEN );**

**}**

**else**

**{**

**FlashLED( RED );**

**}**

**}**

**}**

# 解决方案＃3 减少RAM使用

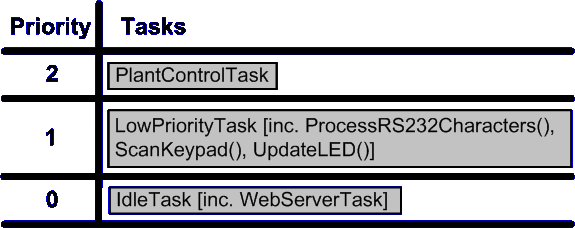
[**<<<**](https://www.freertos.org/tutorial/solution2.html)**|**[**>>>**](https://www.freertos.org/tutorial/solution4.html)

**注意：**自FreeRTOS V4.0.0引入以来，这些页面尚未更新。V4.0.0引入了协程的概念，它将为此处提供的解决方案提供不同而新颖的解决方案。该[任务和协程](https://www.freertos.org/taskandcr.html)文档提供了进一步的信息。

## 概要

解决方案2充分利用了RTOS。这样可以产生一个干净的设计，但是只能在具有足够RAM和处理资源的嵌入式计算机上使用。解决方案3尝试通过将功能分区更改为任务来减少RAM使用。

## 实作

  
**解决方案3的任务和优先级**

[前面](https://www.freertos.org/tutorial/index.html#grouping) 我们已经[看到了](https://www.freertos.org/tutorial/index.html#grouping)如何将假设应用程序的时序要求分为三类：

1. **严格的时间安排**–工厂控制

和以前一样，创建高优先级任务来服务关键控制功能。

1. **仅截止日期的计时**–人机界面

解决方案3将RS232，按键扫描和LED功能分组为一个中等优先级的任务。

由于先前所述的原因，期望嵌入式Web服务器任务以较低的优先级进行操作。与其创建专门用于Web服务器的任务，不如执行空闲任务挂钩以将Web服务器功能添加到该空闲任务。必须编写Web服务器以确保它永远不会阻塞！

1. **灵活的计时**– LED

如果RAM非常昂贵，LED功能太简单，无法保证自己的任务。出于演示的原因，此示例将LED功能包括在单个中优先级任务中。当然，它可以通过多种方式实现（例如，通过外设定时器实现）。

空闲任务以外的任务将阻塞，直到一个事件表明需要进行处理为止。事件可以是外部事件（例如，按下某个键），也可以是内部事件（例如，计时器到期）。

### 经营理念

将功能分组为中等优先级任务比解决方案1中显示的无限循环实现具有三个重要优势：

1. 队列的使用允许中等优先级的任务被阻塞，直到事件导致数据可用为止–然后立即跳转到相关功能以处理该事件。与无限循环实现相反，无限循环实现仅在循环循环到适当的处理程序后才处理事件，因此可以防止浪费处理器周期。
2. 实时内核的使用消除了在应用程序源代码中明确考虑时间紧迫任务的调度的要求。
3. 从循环中删除嵌入式Web服务器功能使执行时间更加可预测。

另外，已被分组为一个任务的功能是从以前共享了相同优先级的几个任务中获取的（禁止LED功能）。无论是单个任务还是多个任务，执行此优先级的代码的频率都不会改变。

作为最高优先级的任务，工厂控制任务可以确保在需要时分配处理时间。如有必要，它将抢占低优先级和中优先级的任务。每当两个应用程序任务都被阻止时，将执行空闲任务。空闲任务可以选择将处理器置于省电模式。

### 调度程序配置

调度程序配置为抢先操作。内核滴答频率应设置为提供所需时间粒度的最低值。

### 结论

对于RAM有限的系统，这可能是一个很好的解决方案，但它仍然占用大量处理器。应该检查系统内的备用容量，以备将来扩展。

## 例

此示例是先前介绍的假设应用程序的部分实现。使用FreeRTOS API。

### 工厂控制任务

工厂控制任务与[解决方案＃2](https://www.freertos.org/tutorial/solution2.html#pcf)中[描述的](https://www.freertos.org/tutorial/solution2.html#pcf)任务相同。

### 嵌入式Web服务器

这只是一个从空闲任务中调用并运行到完成的函数。

### 中优先级任务

中优先级任务可以由以下伪代码表示。

**#define DELAY\_PERIOD 4**

**#define FLASH\_RATE 1000**

**void MediumPriorityTask( void \*pvParameters )**

**{**

**xQueueItem Data;**

**TickType\_t FlashTime;**

**InitialiseQueue();**

**FlashTime = xTaskGetTickCount();**

**for( ;; )**

**{**

**do**

**{**

**// A**

**if( xQueueReceive( xCommsQueue, &Data, DELAY\_PERIOD ) )**

**{**

**ProcessRS232Characters( Data.Value );**

**}**

**// B**

**} while ( uxQueueMessagesWaiting( xCommsQueue ) );**

**// C**

**if( ScanKeypad() )**

**{**

**UpdateLCD();**

**}**

**// D**

**if( ( xTaskGetTickCount() - FlashTime ) >= FLASH\_RATE )**

**{**

**FlashTime = xTaskGetTickCount();**

**UpdateLED();**

**}**

**}**

**// Should never get here.**

**return 0;**

**}**

参考上面代码片段中的标签：

1. 该任务首先阻止等待通信事件。阻止时间相对较短。
2. 执行do-while循环，直到队列中没有数据为止。如果数据到达的速度太快而导致队列完全空了，则必须修改此实现。
3. 队列中的所有数据都已排空，或者在指定的阻塞时间内没有数据到达。阻塞等待数据所花费的最大时间足够短，以确保足够频繁地扫描键盘，以满足指定的时序约束。
4. 检查是否到了闪烁LED的时间。该线路执行的频率会有些抖动，但是LED时序要求足够灵活，可以通过此实现方式来满足。

# 解决方案＃4 减少处理器开销

[**<<<**](https://www.freertos.org/tutorial/solution3.html)**| >>>**

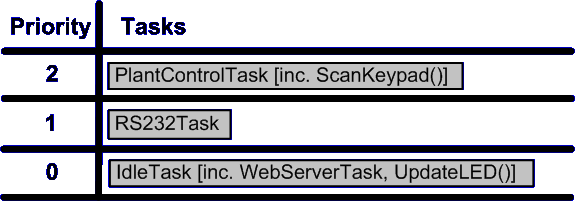
**注意：**自FreeRTOS V4.0.0引入以来，这些页面尚未更新。V4.0.0引入了协程的概念，它将为此处提供的解决方案提供不同而新颖的解决方案。该[任务和协程](https://www.freertos.org/taskandcr.html)文档提供了进一步的信息。

## 概要

解决方案2显示了如何通过充分利用RTOS功能来生成干净的应用程序。解决方案＃3显示了如何将其应用于RAM资源有限的嵌入式计算机。解决方案＃4进行了进一步修改，目的是减少RTOS处理开销。

通过配置内核以进行协作调度，然后从事件中断服务例程中执行上下文切换，可以创建混合调度算法（完全抢占或完全协作）。

## 实作

  
**解决方案4的任务和优先级**

关键的工厂控制功能再次由高优先级任务实现，但是使用协作调度程序需要对其实现进行更改。以前，使用vTaskDelayUntil（） API函数来维护计时。使用抢占式调度程序时，为控制任务分配最高优先级，以确保它在指定的时间开始执行。现在正在使用协作调度程序–因此，仅当从应用程序源代码中明确请求时，任务切换才会发生，因此会丢失保证的时间。

解决方案4使用来自外设计时器的中断来确保以控制任务所需的确切频率请求上下文切换。调度程序确保每个请求的上下文切换都导致切换到能够运行的最高优先级任务。

键盘扫描功能还需要常规的处理器时间，因此它也可以在计时器中断触发的任务中执行。此任务的时间安排可以轻松评估；错误情况给出了控制功能的最坏情况处理时间–当联网传感器没有数据传来导致控制功能超时时，错误情况将给出处理时间。键盘扫描功能的执行时间基本上是固定的。因此，我们可以肯定的是，以这种方式链接它们的功能绝不会导致控制周期频率发生抖动-或更糟的是会错过控制周期。

RS232任务将由RS232中断服务程序安排。

LED功能的灵活计时要求意味着它可能可以在空闲任务挂钩中加入嵌入式Web服务器任务。如果这还不够，那么也可以将其移至高优先级任务。

### 经营理念

协作调度程序仅在明确请求一个上下文切换时才执行。这大大减少了RTOS带来的处理器开销[除了空闲任务不再能够使处理器进入节能模式的事实？]。空闲任务（包括嵌入式Web服务器功能）将在没有任何不必要的内核中断的情况下执行。

来自RS232或定时器外设的中断将导致上下文切换，并且仅在必要时才进行上下文切换。这样，RS232任务仍将抢占空闲任务，而工厂控制任务本身仍可以抢占–保持优先的系统功能。

### 调度程序配置

调度程序配置为协同操作。内核滴答仅用于维护实时滴答值。

### 结论

RTOS内核的功能可以用很少的开销使用，即使在处理器和内存受限而无法完全抢占解决方案的系统上，也可以简化设计。

## 例

此示例是先前介绍的假设应用程序的部分实现。使用FreeRTOS API。

### 高优先级任务

高优先级任务由定期中断服务例程“给予”的信号量触发：

**void vTimerInterrupt( void )**

**{**

**// 'Give' the semaphore. This will wake the high priority task.**

**xSemaphoreGiveFromISR( xTimingSemaphore );**

**// The high priority task will now be able to execute but as**

**// the cooperative scheduler is being used it will not start**

**// to execute until we explicitly cause a context switch.**

**taskYIELD();**

**}**

请注意，用于强制从ISR内部进行上下文切换的语法对于不同的端口是不同的。不要直接复制此示例，而应查看正在使用的端口的文档。

高优先级任务包含工厂控制和键盘功能。 首先调用PlantControlCycle（）以确保其时序一致。

**void HighPriorityTaskTask( void \*pvParameters )**

**{**

**// Start by obtaining the semaphore.**

**xSemaphoreTake( xSemaphore, DONT\_BLOCK );**

**for( ;; )**

**{**

**// Another call to take the semaphore will now fail until**

**// the timer interrupt has called xSemaphoreGiveFromISR().**

**// We use a very long block time as the timing is controlled**

**// by the frequency of the timer.**

**if( xSemaphoreTake( xSemaphore, VERY\_LONG\_TIME ) == pdTRUE )**

**{**

**// We unblocked because the semaphore became available.**

**// It must be time to execute the control algorithm.**

**PlantControlCycle();**

**// Followed by the keyscan.**

**if( KeyPressed( &Key ) )**

**{**

**UpdateDisplay( Key );**

**}**

**}**

**// Now we go back and block again until the next timer interrupt.**

**}**

**}**

**RS232任务**

RS232任务只是在队列中阻塞，等待数据到达。RS232中断服务程序必须将数据发布到队列中-使任务准备就绪可以运行-然后强制进行上下文切换。该机制按照上面给出的计时器中断伪代码进行。

因此，RS232任务可以由以下伪代码表示：

**void vRS232Task( void \*pvParameters )**

**{**

**DataType Data;**

**for( ;; )**

**{**

**if( cQueueReceive( xRS232Queue, &Data, MAX\_DELAY ) )**

**{**

**ProcessRS232Data( Data );**

**}**

**}**

**}**

**嵌入式Web服务器和LED功能**

其余的系统功能位于空闲任务挂钩中。这只是一个空闲任务的每个周期调用的函数。

**void IdleTaskHook( void )**

**{**

**static TickType\_t LastFlashTime = 0;**

**ProcessHTTPRequests();**

**// Check the tick count value to see if it is time to flash the LED**

**// again.**

**if( ( xTaskGetTickCount() - LastFlashTime ) > FLASH\_RATE )**

**{**

**UpdateLED();**

**// Remember the time now so we know when the next flash is due.**

**LastFlashTime = xTaskGetTickCount();**

**}**

**}**

# 在Eclipse中导入和构建演示项目

本页提供有关如何使用Eclipse导入和构建作为Eclipse项目提供的许多FreeRTOS演示应用程序的常规信息。该[演示应用程序特定文档](https://www.freertos.org/a00090.html)页面提供更多的信息针对特定的演示，包括FreeRTOS的目录结构中的项目的位置。下面的演示基于Windows环境下的基本Eclipse Embedded CDT版本。

**注：**特定于供应商的Eclipse发行版可能看起来与用于创建以下屏幕快照的基本Eclipse版本不同。

## 先决条件：

1. 安装Eclipse发行版

基本的Eclipse版本Eclipse Embedded CDT（C / C ++开发工具）可以在[此处](https://projects.eclipse.org/projects/iot.embed-cdt/downloads)下载 。

**注意：**大多数特定于供应商的基于Eclipse的IDE都包含在发行版中的工具链，并为您自动透明地设置路径。请检查您的特定演示页面，该页面链接到[演示应用程序特定文档](https://www.freertos.org/a00090.html)。如果特定于供应商的IDE提供了工具链和构建工具，则可以跳过以下步骤。

1. 安装GCC工具链

请参阅[演示应用程序特定的文档](https://www.freertos.org/a00090.html)页面，以获取演示所需的特定工具链的信息。

1. 安装构建工具（Windows）

POSIX平台通常包含make在系统发行版中，或者它们可能需要您安装一些开发人员软件包（例如，在macOS上，您需要安装Xcode命令行工具）。按照本[指南中](https://gnu-mcu-eclipse.github.io/install/)的“ Windows Build Tools”部分，安装Make生成工具。

**注意：**如果您安装了另一个环境（例如MinGW或Cygwin），并且想要使用该环境提供的Make工具，请确保make程序名为“ make.exe”。同样，对于MinGW，将“ mingw32-make.exe”重命名（或创建副本并重命名）为“ make.exe”。

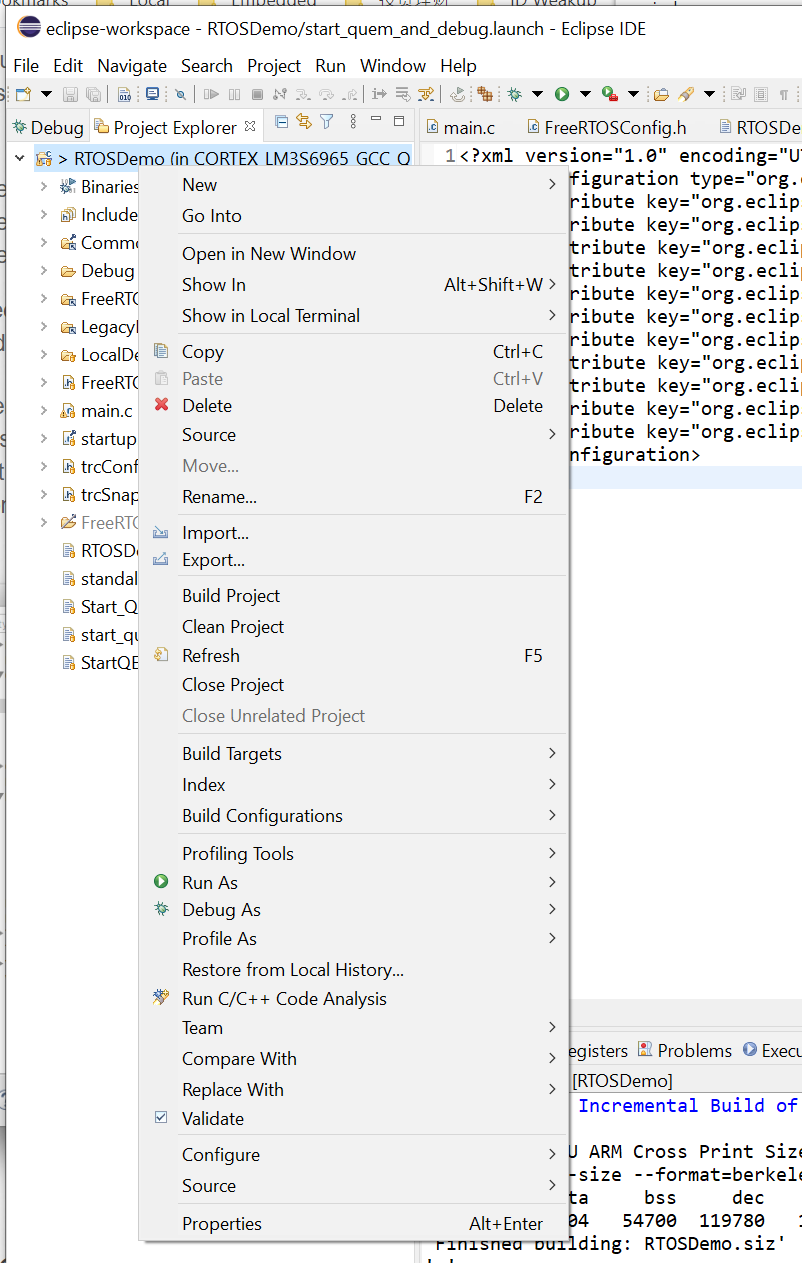
1. 管理PATH环境变量

基本的Eclipse版本依靠设置环境变量PATH来访问工具链二进制文件。该PATH可以设置从：

* + 系统全局PATH设置或每个用户PATH设置。
  + Eclipse工作区所有项目的通用设置。
  + 项目的构建配置。

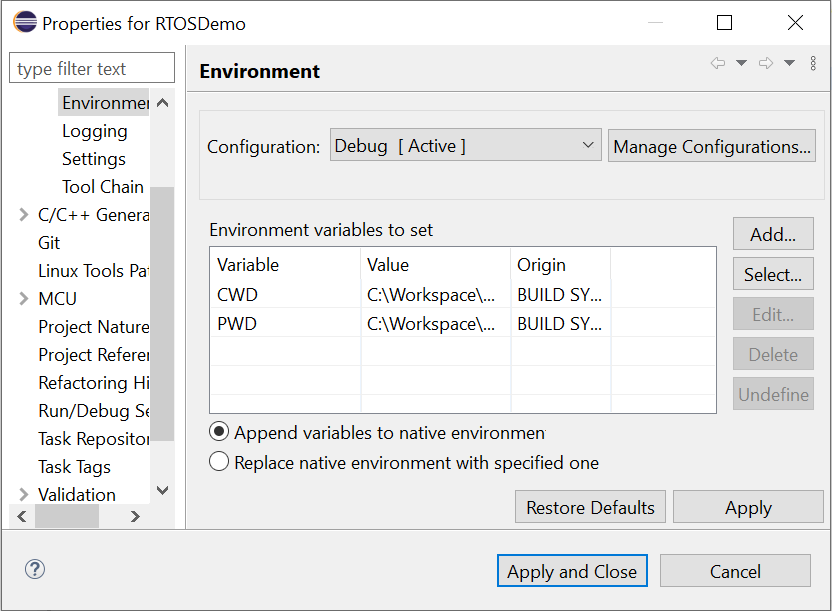
强烈建议您PATH从项目的构建配置中进行设置，以避免在使用计算机上可能安装的多个工具链时发生任何冲突。要PATH 在Eclipse中设置项目的变量，请执行以下操作：

* + 在“项目资源管理器”中右键单击演示项目，然后选择“属性”。



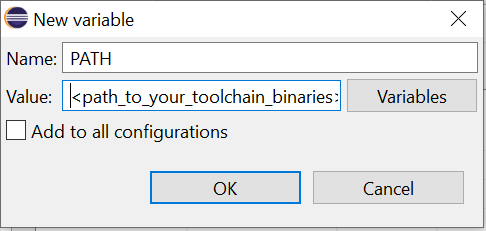
打开项目属性

* + 在弹出窗口的左侧，选择“ C / C ++ build –> Environment”，然后单击右侧的“ Add…”按钮添加一个新变量。



项目属性

* + 输入“ PATH”作为变量名，并使用工具链和构建工具二进制文件文件夹的路径设置值。

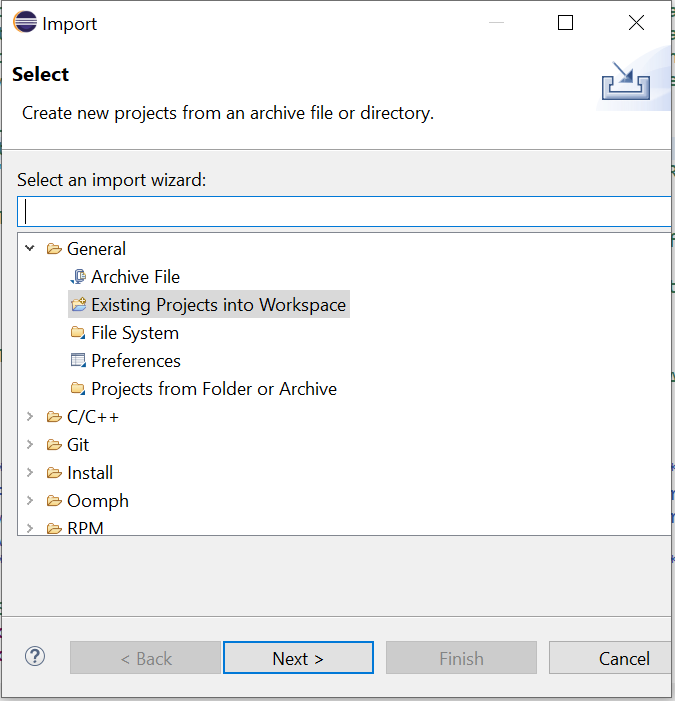


添加PATH变量

## 导入并构建演示项目：

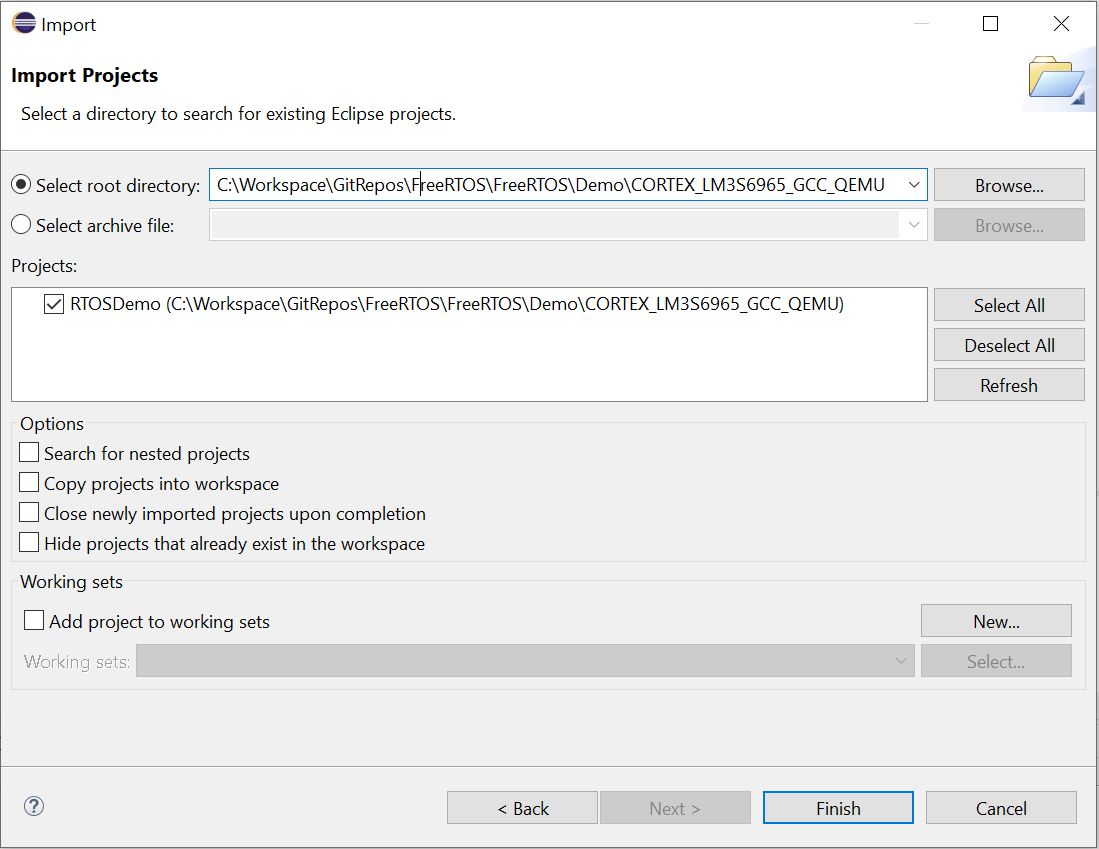
**重要！**FreeRTOS Eclipse项目使用到源文件的相对路径，因此，如果您的目录结构与FreeRTOS官方zip文件发行版中使用的目录结构不同，则不会构建。将项目导入Eclipse工作区时，请确保未选中“将项目复制到工作区”复选框。

1. 启动Eclipse，然后选择一个现有的或在出现提示时创建一个新的工作区。
2. 从Eclipse的“文件”菜单中选择“导入...”。导入对话框将打开。
3. 在“导入”对话框中，选择“常规->现有项目到工作区”。导入项目对话框将打开。



将现有项目导入工作空间

1. 在“导入项目”对话框中，导航到并选择FreeRTOS/Demo/<YOUR\_PROJECT> 目录，并确保未选中“将项目复制到工作区”复选框。



在“导入项目”对话框中选择目录和项目。

1. 在“导入项目”对话框的“项目”窗口中，选择RTOSDemo项目，然后选择“完成”。
2. 从Eclipse的“项目”菜单中选择“全部构建”。确保项目的构建没有任何错误或警告。