# 1[为什么选择FreeRTOS？](http://blog.csdn.net/zhzht19861011/article/details/49819109)

### 1.为什么学习RTOS？

         作为基于ARM7、Cortex-M3硬件开发的[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)工程师，我一直反对使用RTOS。不仅因为不恰当的使用RTOS会给项目带来额外的稳定性风险，更重要的是我认为绝大多数基于ARM7、Cortex-M3硬件的项目，还没复杂到使用RTOS的地步，使用状态机就足够了。

         对于现代的微处理器，特别是资源相对丰富ARM7、Cortex-M3硬件来说，RTOS占用的硬件资源已经越来越可以忽略。所以在当今环境下，我们无需担心RTOS会拖累性能。相反，RTOS提供的事件驱动型设计方式，使得RTOS只是在处理实际任务时才会运行，这能够更合理的利用CPU。在实际项目中，如果程序等待一个超时事件，传统的无RTOS情况下，要么在原地一直等待而不能执行其它任务，要么使用复杂（相对RTOS提供的任务机制而言）的状态机机制。如果使用RTOS，则可以很方便的将当前任务阻塞在该事件下，然后自动去执行别的任务，这显然更方便，并且可以高效的利用CPU。处理这类事件，是我使用RTOS的最大动力，但考虑到系统的稳定性，我不得不再三权衡RTOS可能带来的一些弊端：

1. 大多数RTOS代码都具有一定规模，任何代码都可能带来BUG，何况是代码具有一定规模的RTOS，因此引入RTOS的同时也可能会引入该RTOS的BUG，这些RTOS本身的BUG一旦被触发，影响可能是是灾难性的。
2. 熟练的使用RTOS是一项技能，需要专业的知识储备和长期的经验积累。不将RTOS分析透彻，很容易为项目埋下错误。典型的，像中断优先级、任务堆栈分配、可重入等，都是更容易出错的地方。
3. RTOS的优先级嵌套使得任务执行顺序、执行时序更难分析，甚至变成不可能。任务嵌套对所需的最大堆栈RAM大小估计也变得困难。这对于很多对安全有严格要求的场合是不可想象的。
4. RTOS应该用于任务复杂的场合，以至于对任务调度的需求可以抵消RTOS所带来的稳定性影响，但大部分的应用并非复杂到需要RTOS。

         以上原因是我拒绝在实际项目中使用RTOS的理由，但是否使用RTOS跟是否学习RTOS完全是两码事。我认为任何嵌入式软件设计人员都应该至少学习一种RTOS，不仅是需要掌握RTOS背后的[**操作系统**](http://lib.csdn.net/base/operatingsystem)原理、学习RTOS的编程方式，更是为将来做准备。

         即便我认为现在的物联网有点言过其实，但我依然看好物联网的发展前景。随着物联网的发展，未来的嵌入式产品必然更为复杂、连接性更强以及需要更丰富的用户界面。当处理这些任务时，一个好的RTOS就变得不可缺少了。

         书到用时方恨少，我希望自己永远不会有这种感觉。所以从现在起，我要开始深入一个RTOS，探索它背后的原理，掌握其编程方法，避免其缺陷和陷阱，并将它安全的用在将来的项目中。

### 2.为什么选用FreeRTOS？

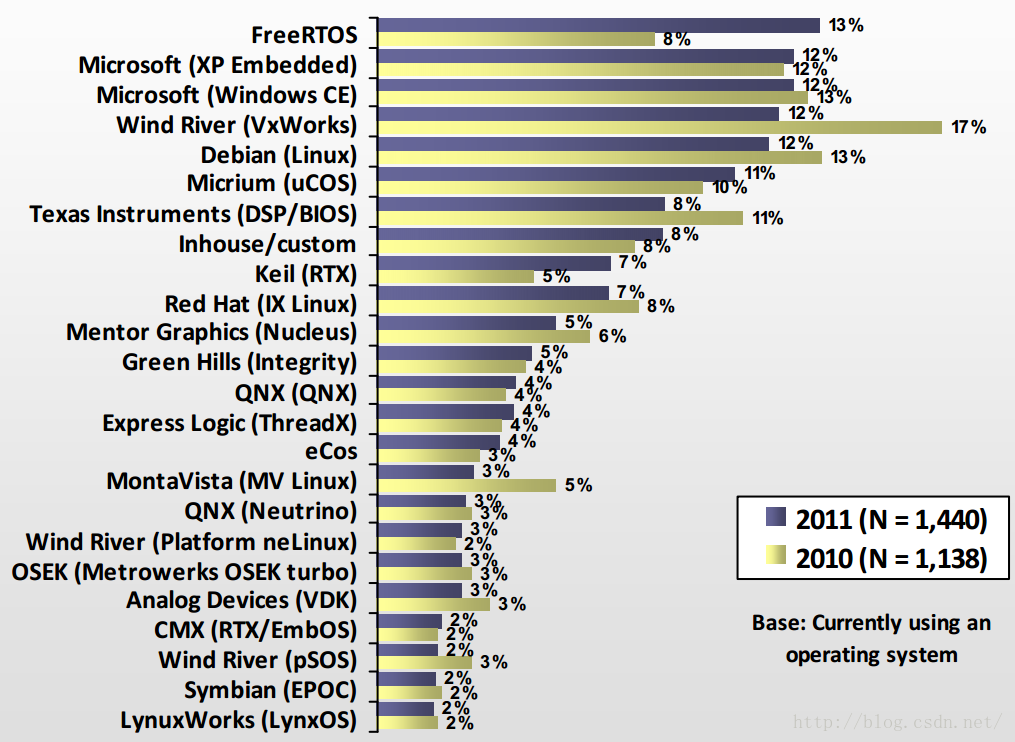
         对比了许多RTOS，最终选择FreeRTOS，原因是多方面的：

1. SafeRTOS便是基于FreeRTOS而来，前者是经过安全认证的RTOS，因此对于FreeRTOS的安全性也有了信心。
2. 大量开发者使用，并保持高速增长趋势。2011、2012、2013、2014、2015连续5年的EEtimes杂志嵌入式系统市场报告显示，FreeRTOS在RTOS内核使用榜和RTOS内核计划使用榜上都名列前茅。更多的人使用可以促进发现BUG，增强稳定性。
3. 简单。内核只有3个.c文件，全部围绕着任务调度，没有任何其它干扰，便于理解学习。而且，我根本不需要其它繁多的功能，只要任务调度就够了。
4. 文档齐全。在FreeRTOS官方网站上，可以找到所有你需要的资料。
5. 免费、开放源码。完全可以免费用于商业产品，开放源码更便于学习操作系统原理、从全局掌握FreeRTOS运行机理、以及对操作系统进行深度裁剪以适应自己的硬件。

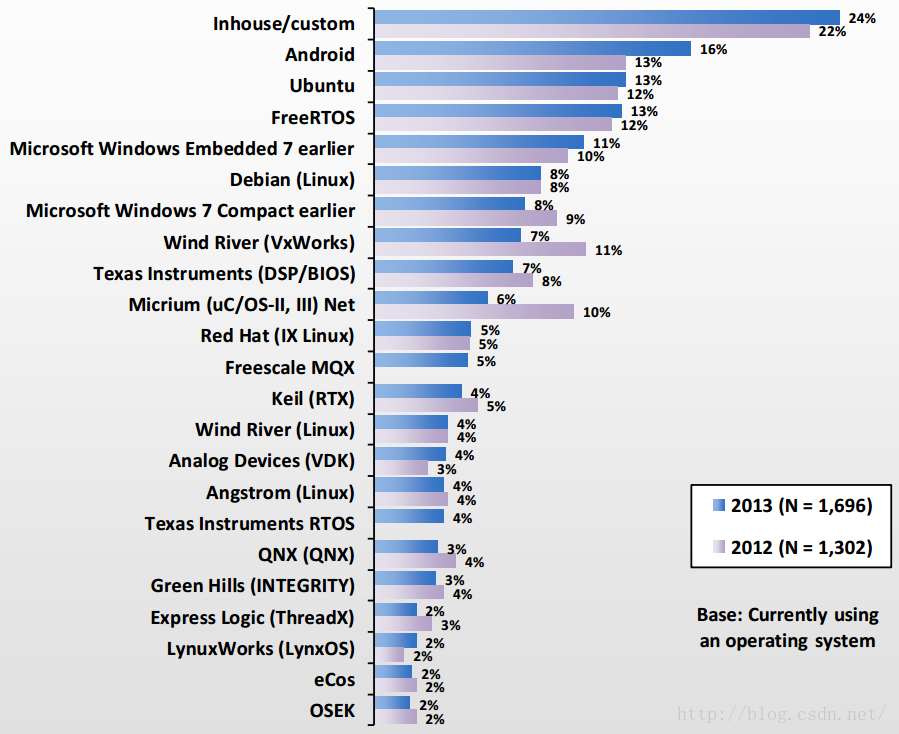
         学习的资料来源主要是FreeRTOS的官方网站（[www.freertos.org](http://www.freertos.org/)）和源代码。FreeRTOS的创始人RichardBarry编写了大量的移植代码和配套文档，我只不过是沿着Richard Barry铺好的路前进，所以，这没什么困难的。

         最后，感谢RichardBarry的付出，感谢Richard Barry的无私开源精神！

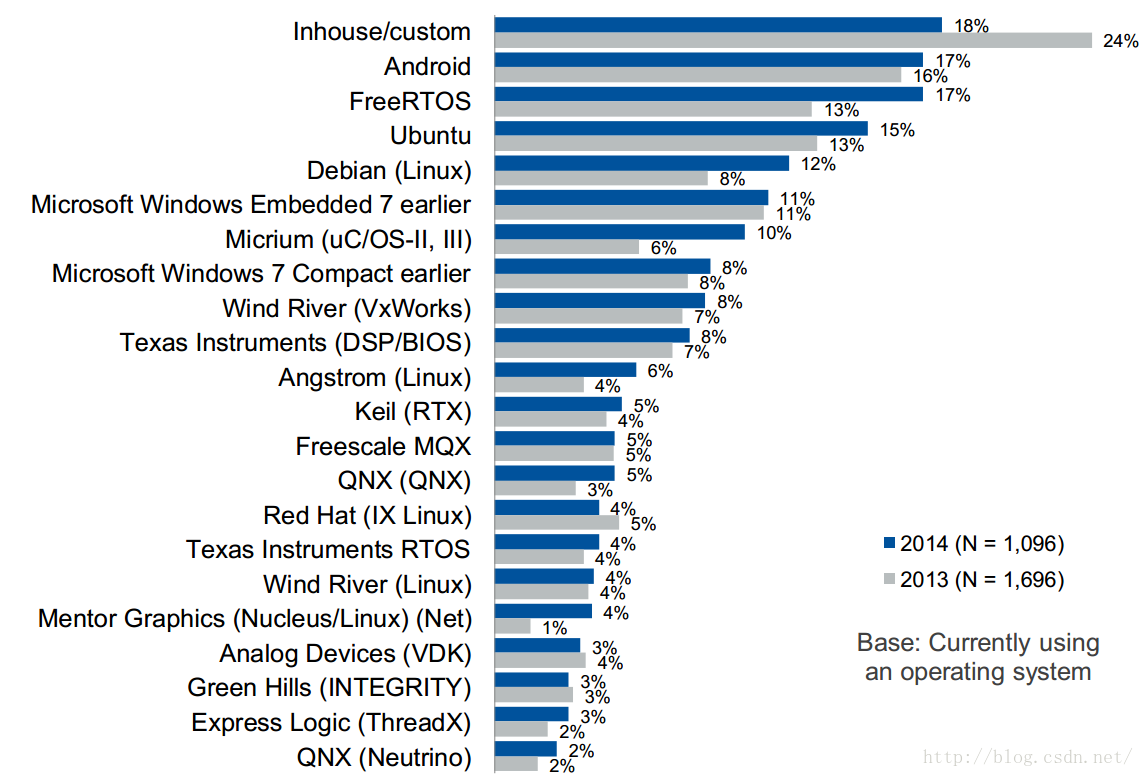
附录1:2010~2014年EEtimes杂志嵌入式市场调查报告有关RTOS使用榜截图



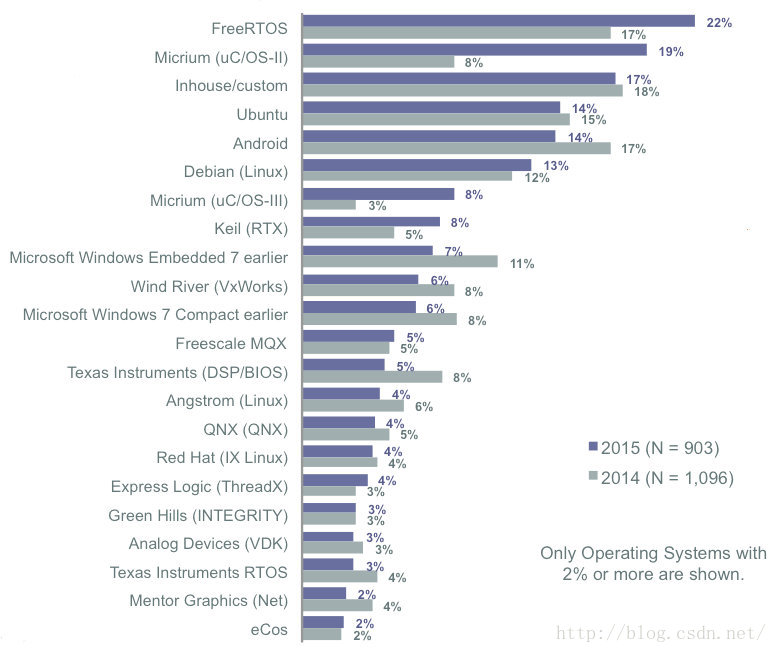
2010和2011年RTOS使用榜



2012和2013年RTOS使用榜



2013年和2014年RTOS使用榜



2014年和2015年RTOS使用榜

# 2 [FreeRTOS入门指南](http://blog.csdn.net/zhzht19861011/article/details/49819309)

FreeRTOS可以被移植到很多不同[**架构**](http://lib.csdn.net/base/architecture)的处理器和编译器。每一个RTOS移植都附带一个已经配置好的演示例程，可以方便快速启动开发。更好的是，每个演示例程都附带一个说明网页，提供如何定位RTOS演示工程源代码、如何编译演示例程、如何配置硬件平台的全部信息。

      演示例程说明网页还提供基本的RTOS移植细节信息，包括如何编写FreeRTOS兼容的中断服务例程，不同架构的中断处理会稍有不同。

      通过下面的简单说明，可以在几分钟内运行RTOS。

**1.查找相关文档页**

       FreeRTOS具有详细的开发说明文档，可以在其官方网站上查看。首先打开官方网站，目前的网站地址是：[http://www.freertos.org](http://www.freertos.org/)。在首页左侧的导航栏中，展开"Supported Devices & Demos"菜单项，单击"OfficiallySupported Demos"链接，去查看FreeRTOS支持的微控制器制造商列表。单击微控制器制造商名称，进入具体的制造商文档页面列表。

**2.获取RTOS源代码**

       到FreeRTOS官方网站下载源码，下载包包含RTOS内核源码和官方移植演示工程。解压缩后放到合适的目录下。(如果你不想访问慢吞吞的国外网站，我在CSDN做了一个镜像，可以 [点击此处](http://blog.csdn.net/zhzht19861011/article/details/51525665) ，这篇文章中有最新的大部分FreeRTOS源码包下载链接)

       每一个RTOS移植包都附带有预先配置好的演示例程 ，已经创建好了所有必须的RTOS源文件并包含了必须的RTOS头文件。推荐在提供的演示例程的基础上进行自己的FreeRTOS应用编程。

**3.FreeRTOS源码目录结构**

       FreeRTOS下载包中包含每个处理器移植和演示例程的源码。将所有移植包放入一个下载文件中大大简化了分类处理，但是下载包中的文件数量也多的惊人！无论如何，目录结构还是非常简单的，并且FreeRTOS实时内核仅仅只有3个文件（如果需要，还有一些附加文件，比如软件定时器、事件组以及协程）。

      下载包目录包含两个子目录：FreeRTOS和FreeRTOS-Plus。如下所示：

* FreeRTOS-Plus          包含FreeRTOS+组件和演示例程；
* FreeRTOS                   包含FreeRTOS实时内核源文件和演示例程。

      FreeRTOS-Plus目录树包含多个自述文件（Readme）。接下来本文只描述FreeRTOS内核的核心源文件和演示例程，它们又被分成两个主要的子目录，如下所示：

1. FreeRTOS
2. |+-- Demo     包含演示例程工程；
3. |+-- Source   包含实时内核源文件。

      RTOS代码的核心包含在三个文件中：tasks.c、queue.c、list.c。这三个文件位于FreeRTOS/Source目录。在该目录下还包含三个可选的文件：timers.c、event\_groups.c、croutine.c，分别实现软件定时、事件组和协程功能。

      FreeRTOS/Source目录结构如下所示：

1. FreeRTOS
2. | +-- Source  FreeRTOS内核代码文件
3. |   |+-- include          FreeRTOS内核代码头文件
4. |   |+-- Portable         处理器特定代码
5. |   |  |+--Compiler x     支持编译器x的所有移植包
6. |   |  |+--Compiler y     支持编译器y的所有移植包
7. |   |  |+--MemMang        内存堆实现范例

      每个支持的处理器架构需要一小段与处理器架构相关的RTOS代码。这个是RTOS移植层，它位于FreeRTOS/Source/Portable/[相应编译器]/[相应CPU架构]子目录。

      对于FreeRTOS，堆栈设计也属于移植层。FreeRTOS/Source/portable/MemMang目录下heap\_x.c文件给出了多种堆栈方案，后续文章将会详细介绍堆栈操作。

      移植层目录举例：

* 如果在GCC编译器下使用TriCore1782：TriCore特定文件（port.c）位于FreeRTOS/Source/Portable/GCC/TriCore\_1782目录下。FreeRTOS/Source/Portable 子目录下的所有文件，除了FreeRTOS/Source/Portable/MemMang目录外都可以忽略或删除。
* 如果在IAR编译器下使用Renesas RX600：RX600特定文件（port.c）位于FreeRTOS/Source/Portable/IAR/RX600目录下。FreeRTOS/Source/Portable 子目录下的所有文件，除了FreeRTOS/Source/Portable/MemMang目录外都可以忽略或删除。

      FreeRTOS下载包中还包含各种处理器架构和编译器的演示例程。大多数的演示例程代码对所有移植都是通用的，位于FreeRTOS/Demo/Common/Minimal目录。FreeRTOS/Demo/Common/Full目录下的是历史遗留代码，仅用于PC。

      FreeRTOS/Demo目录结构如下所示：

1. FreeRTOS
2. |+-- Demo
3. |  |+-- Common          所有例程都可以使用的演示例程文件
4. |  |+-- Dir x           用于x平台的演示例程工程文件
5. |  |+-- Dir y           用于y平台的演示例程工程文件

      FreeRTOS/Demo目录下剩余的子目录包含预先配置好的工程，可以用于构建个人演示例程。子目录的命名与移植平台和编译器相关。每一个RTOS移植包都有自己的说明文档。

      演示例程目录举例：

* 如果以英飞凌TriBoard开发板硬件构建TriCoreGCC演示例程：TriCore演示例程工程文件位于FreeRTOS/Demo/TriCore\_TC1782\_TriBoard\_GCC目录。目录FreeRTOS/Demo下的所有子目录（Common目录除外）都可以忽略或删掉。
* 如果以RX62N硬件构建Renesas RX600 IAR演示例程：IAR工程文件位于FreeRTOS/Demo/RX600\_RX62N-RDK\_IAR目录。目录FreeRTOS/Demo下的所有子目录（Common目录除外）都可以忽略或删掉。

**4.编译工程**

      根据上一节FreeRTOS源码目录结构说明的RTOS演示工程的所在的位置，打开并编译演示工程。

**5.运行演示例程**

      演示例程附带的说明网页会介绍如何配置硬件、下载程序和执行演示例程。说明网页还会提供演示例程的功能信息，这样你就可以判断演示例程执行是否正确。

# 3 [FreeRTOS移植指南](http://blog.csdn.net/zhzht19861011/article/details/49961843)

FreeRTOS下载包中已经包含很多演示例程- 每一个例程都是针对于：

1. 特定的微控制器；
2. 特定的开发工具（编译器、调试器等等）；
3. 特定的硬件平台（样机或评估板）。

      可以在官方网站首页左侧的树形菜单 'Supported Devices' 中找到这些例程介绍。

      可惜的是不可能为所有微控制器、编译器和评估板提供演示例程。因此，官方提供的演示例程可能不完全符合你正在使用的开发平台。本章描述如何通过修改或合并官方提供的演示例程，来满足自己的开发平台需求（包括微处理器和编译器）。

      修改一个现有的评估板例程，使之运行到另一个同类评估板上，通常是比较简单的，稍微复杂些的是跨编译器移植。本文介绍这两情况下的修改，只是对相似的平台有效。然而，将FreeRTOS移植到一个全新的平台、未支持的处理器[**架构**](http://lib.csdn.net/base/architecture)，并不是件简单的事情。本文不讨论如何将FreeRTOS移植到一个全新平台。

### 1.修改例程使之运行到不同评估板

      本节描述如何通过修改一个官方提供的演示例程，使之运行到另一个评估板，这里两个评估板使用同系列微处理器，使用相同编译器。在这个例子中，将运行于SAM7S-EK硬件开发板上的IAR SAM7S演示例程，修改使之运行到Olimex SAM7-P64开发板。（注：两块开发板都是使用ATMEL公司的ARM7微处理器，前者使用AT91SAM7S256，后者使用AT91SAM7S64）

#### 1.1初始编译

      作为修改练习的起点，被修改的演示例程是要能使用的。因此，在未做任何修改之前，首先检查下载的演示例程能否被正确的编译。绝大多数情况下，演示例程编译后是没有任何错误和警告的。

      关于演示例程所在目录，参考[《FreeRTOS系列第2篇---FreeRTOS入门指南》](http://blog.csdn.net/zhzht19861011/article/details/49819309)一文的第三节。

#### 1.2修改LED IO端口

       LED灯是用来指示演示例程运行的最简单方法，因此点亮新硬件平台上的LED灯通常是最容易的。

      两个不同评估板上的LED连接到相同的IO端口通常是不太可能的，因此，一些小幅度修改是必须的。

      在partest.c文件中的vParTestInitialise() 函数包含IO端口的模式和方向配置。在main.c文件中的prvSetupHardware()函数包含更多的硬件初始化（比如，使能IO外设的时钟模块），可能需要根据不同的使用进行一些修改。

      根据目标评估板的硬件，在上面两个函数中做必要的修改，然后写一段简单程序，来检查硬件LED是否完好。这个简单程序不使用FreeRTOS，只是为了确保硬件LED能够正常工作。因此，注释掉之前的main()函数，使用下面的例子代替：

1. int main( void )
2. {
3. volatile unsigned long ul;   /\* 禁止编译器优化此变量 \*/
4. /\* 初始化LED IO为输出-注：prvSetupHardware()也可能会被调用\*/
5. vParTestInitialise();
6. /\*不断开启LED \*/
7. for( ;; )
8. {
9. /\* 我们暂时不使用RTOS，这里只是使用一个非常粗糙的延时\*/
10. for( ul = 0; ul < 0xfffff; ul++ )
11. {
12. }
13. /\* 开启4个LED \*/
14. vParTestToggleLED( 0 );
15. vParTestToggleLED( 1 );
16. vParTestToggleLED( 2 );
17. vParTestToggleLED( 3 );
18. }
19. return 0;
20. }

#### 1.3 RTOS调度器简介

      一旦确定硬件LED可以正常工作，就可以恢复原来的main()函数。

      作为入门级的多任务应用程序应该尽量的简单，LED闪烁[**测试**](http://lib.csdn.net/base/softwaretest)程序常常担任这样的角色，可以堪比经典的“Hello Wold”。这个任务几乎在所有演示例程中都能看到，在main()函数中调用vStartLEDFlashTasks() （使用协程版本时调用vStartFlashCoRoutines()）来实现。如果你使用的演示例程main()函数中并没有调用vStartLEDFlashTasks()（或vStartFlashCoRoutines()），那么需要你将FreeRTOS/Demo/Common/Minimal/Flash.c文件添加到你的工程，并在main()函数手动的增加vStartLEDFlashTasks()函数。

      除了调用vStartLEDFlashTasks()外，注释掉所有用于启动一个或多个演示任务的函数。最后的main()函数仅调用三个函数：prvSetupHardware()、vStartLEDFlashTasks()和vTaskStartScheduler()。例如（基于典型的main()函数）：

1. int main( void )
2. {
3. /\* 设置用于演示的微控制器硬件 \*/
4. prvSetupHardware();
5. /\* 留下这个函数 \*/
6. vCreateFlashTasks();
7. /\* 所有的其它创建任务的函数统统注释掉
8. vCreatePollQTasks();
9. vCreateComTestTasks();
10. //等等…
11. xTaskCreate( vCheckTask,"check", STACK\_SIZE, NULL, TASK\_PRIORITY, NULL );
12. \*/
13. /\*启动RTOS调度器. \*/
14. vTaskStartScheduler();
15. /\* 永远不会执行到这里! \*/
16. return 0;
17. }

      这是一个非常简单的应用程序，正确执行后，LED0~2（包括2）或分别按照不同的频率闪烁。

#### 1.4 收尾工作

      一旦简单的LED闪烁例程正确执行后，你可以恢复之前注释掉的所有的演示任务。

      以下要点需牢记：

* 如果你使用的演示例程最初并没有调用vTaskCreateFlashTasks()函数，而是手动的增加了这个函数，那么应该再手动的删除掉这个函数。主要有两个方面的原因：第一是这个LED闪烁任务用到的IO可能也被演示例程的其它任务使用，第二是演示例程可能已经占用了所有的RAM，已经没有空余RAM用来增加新的任务。
* 标准的“通讯测试（comtest）”（如果演示例程中有的话）任务使用到微控制器的一个UART外设。检测硬件是可用的。
* 有些外设不进行修改就想用于任何不同的硬件或接口是不可能的，比如LCD。

### 2.合并或修改官方演示工程

      本节主要描述如何修改一个现存的工程或者按照需求合并两个现存的工程。比如，你希望使用GCC编译器创建一个STR9演示工程（demo project），并且你下载的FreeRTOS软件包中并没有GCC版本的STR9演示例程，但是FreeRTOS下载包中有IAR版本的STR9演示例程和GCC版本的STR75x演示例程。则可以通过这两个现存的工程来创       建GCC版本的STR9演示工程。可以有两种方式完成：

      使用GCC版本的STR75x演示工程，修改使之符合指定的微处理器（STR9评估板上的微处理器）。

      使用GCC创建一个新的工程。从IAR版本的STR9演示工程中获取文件和配置信息，使之符合GCC编译器需求。

#### 2.1识别用于特定微控制器的FreeRTOS内核文件

      对于一个特定平台，大多数（不是全部）硬件接口代码包含在一个叫做FreeRTOS/source/portable/[编译器]/[微控制器/port.c的文件中，和它对应的头文件是FreeRTOS/source/portable/[编译器]/[微控制器]/portmacro.h。

     对于一些编译器来说，port.c和portmacro.h就是所需要的全部硬件接口代码。另一些还需要一些汇编文件，这些文件叫做portasm.s或者portasm.asm。

     最后，仅对于ARM7 GCC移植，同样存在一个类似的硬件接口文件：portISR.c，portISR.c是从port.c中分离出来的，这些代码必须在ARM模式下编译，port.c中剩余的代码既可以在ARM模式下编译，也可在THUMB模式下编译。

#### 2.2识别用于特定编译器的文件

      编译器可以为[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)系统提供某些特定的[**C语言**](http://lib.csdn.net/base/c)扩展。比如某个特定关键字可以标识出一个函数是中断处理服务函数。

      扩展的C语言部分，是不属于标准C语言规范的。因此，编译器与编译器之间是有差别的。FreeRTOS的文件中就包含类似的非标准C语言语法，在文件夹FreeRTOS/source/portable中（上文中提到的特定微控制器硬件接口代码也在这个文件中）。此外，一些演示例程会使用到中断服务程序，这些中断服务程序并不属于FreeRTOS的一部分，并且如何定义和使用这些中断服务程序也是编译器所特定的。

#### 2.3硬件底层文件

      C启动文件和链接脚本都属于处理器和编译器特定的。不推荐尝试从无到有的创建这些文件，应该到FreeRTOS演示工程中寻找一个合适的来修改。

      要特别小心ARM7启动文件。它必须将IRQ中断服务程序入口地址配置到快速中断处理向量表或者普通中断向量表中。这两种情况，演示工程都提供了例子。

      链接脚本必须正确的描述当前使用处理器的内存映射。

#### 2.4工程设置

      每一个工程通常都会定义一些宏，这些预处理宏定义了一些要被编译的特定的硬件接口代码。要包含portmacro.h文件才能识别这些宏。比如，当使用GCC编译MegaAVR硬件接口代码时，宏GCC\_MEGA\_AVR必须被定义；当使用IAR编译MegaAVR硬件接口代码时，宏IAR\_MEGA\_AVR必须被定义等等。参考演示例程工程以及FreeRTOS/source/include/portable.h文件可以查找当前工程定义了那些宏。如果预处理宏未定义，那么portmacro.h文件所在目录的路径必须被包含到预处理器的搜索路径中。

      其它的编译器设置，比如优化选项，也是很关键的。可以参考提供的演示工程。

具有IDE的编译器通常具有目标微控制器选项并将它作为工程设置的一部分，所以新的工程也必须适应新的目标微控制器，同样的，如果使用到makefile文件，则makefile文件也必须更新以符合新的目标微控制器。

#### 2.5配置系统节拍时钟中断

      调用函数prvSetupTimerInterrupt()来配置系统节拍中断，这个函数可以在以下路径的文件中找到：FreeRTOS/source/portable/[compiler]/[microcontroller]/port.c

#### 2.6 RAM和ROM的使用

      FreeRTOS内存管理一章中描述了FreeRTOS如何使用RAM，并且描述了RAM是如何分配给RTOS内核的。

      如果你要将演示例程移植到一个RAM稍小的微处理器上，那么你可能需要减少configTOTAL\_HEAP\_SIZE的值（位于FreeRTOSConfig.h），并且减少演示例程的任务个数。可以通过简单的注释掉不需要的任务来实现。

      如果你要将演示例程移植到一个ROM较小的微处理器中，那么你可能需要减少应用例程的文件数目，他们位于FreeRTOS/Demo/common文件夹目录下。同时你还要删除main函数中对他们的调用函数。

注：可能你是通过搜索引擎找到这篇文章，满怀希望的点进来，以为能解决自己移植的所有问题，但是看完后却发现本文站的角度太高，并不是特别适合对移植一无所知的你。先别急着以为本文是标题党而点踩，可能有一篇文章适合你，这篇文章以Cortex-M3硬件平台为例，详细的介绍移植过程，请[点击这里](http://blog.csdn.net/zhzht19861011/article/details/50072033)。

# 4 [FreeRTOS编码标准及风格指南](http://blog.csdn.net/zhzht19861011/article/details/50057531)

**1.编码标准**

      FreeRTOS的核心源代码遵从MISRA编码标准指南。这个标准篇幅稍长，你可以在MISRA官方网站花少量钱买到，这里不再复制任何标准。

      FreeRTOS源代码不符合MISRA标准的项目如下所示：

* 有两个API函数有多个返回点。MISRA编码标准强制规定：一个函数在其结尾应该有单一的返回点。
* 指针算数运算，在创建任务时，为了兼容8、16、20、24、32位总线，不可避免的使用了指针算数运算。MISRA编码标准强制规定：指针的算术运算只能用在指向数组或数组元素的指针上。
* 默认情况下，跟踪宏为空语句，因此不符合MISRA的规定。MISRA编码标准强制规定：预处理指令在句法上应该是有意义的。

      FreeRTOS可以在很多不同编译器中编译，其中的一些编译器比同类有更高级特性。因为这个原因，FreeRTOS不使用任何非[**C语言**](http://lib.csdn.net/base/c)标准的特性或语法。一个例外情况是头文件stdint.h。在文件夹FreeRTOS/Source/include下包含一个叫做stdint.readme的文件，如果你的编译器不提供stdint类型定义，可以将stdint.readme文件重命名为stdint.h。

**2.命名规则**

      RTOS内核和演示例程源代码使用以下规则：

      1> 变量

* uint32\_t类型的变量使用前缀ul，这里’u’表示’unsigned’，’l’表示’long’
* uint16\_t类型的变量使用前缀us，这里’u’表示’unsigned’，’s’表示’short’
* uint8\_t类型的变量使用前缀uc，这里’u’表示’unsigned’，’c’表示’char’
* 非stdint类型的变量使用前缀x，比如基本的Type\_t和TickType\_t类型，这些类型在移植层定义，定义成符合处理器架构的最高效类型；
* 非stdint类型的无符号变量使用前缀ux，比如UbaseType\_t（unsigned BaseType\_t）
* size\_t类型的变量使用前缀x；
* 枚举类型变量使用前缀e
* 指针类型变量在类型基础上附加前缀p，比如指向uint16\_t的指针变量前缀为pus
* 与MISRA指南一致，char类型变量仅被允许保存ASCII字符，前缀为c
* 与MISRA指南一致，char \*类型变量仅允许指向ASCII字符串，前缀为pc

      2> 函数

* 在文件作用域范围的函数前缀为prv
* API函数的前缀为它们的返回类型，当返回为空时，前缀为v
* API函数名字起始部分为该函数所在的文件名。比如vTaskDelete函数定义在tasks.c，并且该函数返回空。

      3> 宏

* 宏的名字起始部分为该宏定义所在的文件名的一部分。比如configUSE\_PREEMPTION定义在FreeRTOSConfig.h文件中。
* 除了前缀，宏剩下的字母全部为大写，两个单词间用下划线（’\_’）隔开。

**3. 数据类型**

      只有stdint.h和RTOS自己定义的数据类型可以使用，但也有例外情况，如下所示：

* char：与MISRA编码标准指南一致，char类型变量仅被允许保存ASCII字符
* char \*：与MISRA编码标准指南一致，char \*类型变量仅允许指向ASCII字符串。当标准库函数期望一个char \*参数时，这样做可以消除一些编译器警告；特别是考虑到有些编译器将char类型当做signed类型，还有些编译器将char类型当做unsigned类型。

      有三种类型会在移植层定义，它们是：

* TickType\_t：如果configUSE\_16\_BIT\_TICKS为非零（条件为真），TickType\_t定义为无符号16位类型。如果configUSE\_16\_BIT\_TICKS为零（条件为假），TickType\_t定义为无符号32位类型。注：32位架构的微处理器应设置configUSE\_16\_BIT\_TICKS为零。
* BaseType\_t：定义为微处理器架构效率最高的数据类型。比如，在32位架构处理器上，BaseType\_t应该定义为32位类型。在16位架构处理器上，BaseType\_t应该定义为16位类型。如果BaseType\_t定义为char，对于函数返回值一定要确保使用的是signed char，否则可能造成负数错误。
* UbaseType\_t：这是一个无符号BaseType\_t类型

3.4风格指南

* 缩进：缩进使用制表符，一个制表符等于4个空格。
* 注释：注释单行不超过80列，特殊情况除外。不使用C++风格的双斜线（//）注释
* 布局：FreeRTOS的源代码被设计成尽可能的易于查看和阅读。下面的代码片中，第一部分展示文件布局，第二部分展示C代码设计格式。

1. /\* 首先在这里包含库文件... \*/
2. #include <stdlib.h>
4. /\* ...然后是FreeRTOS的头文件... \*/
5. #include "FreeRTOS.h"
7. /\* ...紧接着包含其它头文件. \*/
8. #include "HardwareSpecifics.h"
10. /\* 随后是#defines, 在合理的位置添加括号. \*/
11. #define A\_DEFINITION    ( 1 )
13. /\*
14. \* 随后是Static (文件内部的)函数原型,
15. \* 如果注释有多行，参照本条注释风格---每一行都以’\*’起始.
16. \*/
17. static void prvAFunction( uint32\_t ulParameter );
19. /\* 文件作用域变量（本文件内部使用）紧随其后，要在函数体定义之前. \*/
20. static BaseType\_t xMyVariable.
22. /\* 每一个函数的结束都有一行破折号，破折号与下面的第一个函数之间留一行空白。\*/
24. /\*-----------------------------------------------------------\*/
26. void vAFunction( void )
27. {
28. /\* 函数体在此定义，注意要用大括号括住 \*/
29. }
30. /\*-----------------------------------------------------------\*/
32. static UBaseType\_t prvNextFunction( void )
33. {
34. /\* 函数体在此定义. \*/
35. }
36. /\*-----------------------------------------------------------\*/
38. /\*
39. \* 函数名字总是占一行，包括返回类型。 左括号之前没有空格左括号之后有一个空格，
40. \* 每个参数后面有一个空格参数的命名应该具有一定的描述性.
41. \*/
42. void vAnExampleFunction( long lParameter1, unsigned short usParameter2 )
43. {
44. /\* 变量声明没有缩进. \*/
45. uint8\_t ucByte;
47. /\* 代码要对齐.  大括号占独自一行. \*/
48. for( ucByte = 0U; ucByte < fileBUFFER\_LENGTH; ucByte++ )
49. {
50. /\* 这里再次缩进. \*/
51. }
52. }
54. /\*
55. \* for、while、do、if结构具有相似的模式。这些关键字和左括号之间没有空格。
56. \* 左括号之后有一个空格，右括号前面也有一个空格，每个分号后面有一个空格。
57. \* 每个运算符的前后各一个空格。使用圆括号明确运算符的优先级。不允许有0
58. \* 以外的数字（魔鬼数）出现，必要时将这些数字换成能表示出数字含义的常量或
59. \* 宏定义。
60. \*/
61. for( ucByte = 0U; ucByte < fileBUFFER\_LENGTH; ucByte++ )
62. {
63. }
65. while( ucByte < fileBUFFER\_LENGTH )
66. {
67. }
69. /\*
70. \* 由于运算符优先级的复杂性，我们不能相信自己对运算符优先级时刻保持警惕
71. \* 并能正确的使用，因此对于多个表达式运算时，使用括号明确优先级顺序
72. \*/
73. if( ( ucByte < fileBUFFER\_LENGTH ) && ( ucByte != 0U ) )
74. {
75. ulResult = ( ( ulValue1 + ulValue2 ) - ulValue3 ) \* ulValue4;
76. }
78. /\* 条件表达式也要像其它代码那样对齐。 \*/
79. #if( configUSE\_TRACE\_FACILITY == 1 )
80. {
81. /\* 向TCB增加一个用于跟踪的计数器. \*/
82. pxNewTCB->uxTCBNumber = uxTaskNumber;
83. }
84. #endif
86. /\*方括号前后各留一个空格\*/
87. ucBuffer[ 0 ] = 0U;
88. ucBuffer[ fileBUFFER\_LENGTH - 1U ] = 0U;

# 5 [FreeRTOS在Cortex-M3上的移植](http://blog.csdn.net/zhzht19861011/article/details/50072033)

### 1. FreeRTOS下载包的文件结构

      在FreeRTOS官方网站可以下载到最新版的FreeRTOS包，我这里使用的是V8.2.3版本。

      下载包内的总文件数量多的令人生畏，但文件结构却很简洁。[**《FreeRTOS入门指南》**](http://blog.csdn.net/zhzht19861011/article/details/49819309)一文的第3节详细描述了下载包文件结构，我们这里只是简单提一下。

      下载包根目录下包含两个子目录：FreeRTOS和FreeRTOS-Plus。其中，FreeRTOS-Plus文件夹中包含一些FreeRTOS+组件和演示例程（组件大都收费），我们不对这个文件夹下的内容多做了解，重点说一下FreeRTOS文件夹。

      FreeRTOS文件夹下包含两个子目录：Demo和Source。其中，Demo包含演示例程的工程文件，Source包含实时[**操作系统**](http://lib.csdn.net/base/operatingsystem)源代码文件。

      FreeRTOS实时操作系统内核仅包含三个必要文件，此外还有三个可选文件。RTOS核心代码位于三个源文件中，分别是tasks.c、queue.c和list.c。这三个文件位于FreeRTOS/Source目录下，在同一目录下还有3个可选的文件，叫做timers.c、event\_groups.c和croutine.c，分别用于软件定时器、事件组和协程。

      对于支持的处理器[**架构**](http://lib.csdn.net/base/architecture)，RTOS需要一些与处理器架构相关的代码。可以称之为RTOS硬件接口层，它们位于FreeRTOS/Source/Portable/[相应编译器]/[相应处理器架构]文件夹下。我们这次要移植到Cortex-M3微控制，使用Keil MDK编译器，所以需要的RTOS硬件接口代码位于：FreeRTOS\Source\portable\RVDS\ARM\_CM3文件夹下。

      堆栈分配也是属于硬件接口层（移植层），在FreeRTOS/Source/portable/MemMang文件夹下具有各种类型的堆栈分配方案。这里我们使用heap\_1.c提供的堆栈分配方案。关于FreeRTOS的内存管理，后续[**《FreeRTOS内存管理》**](http://blog.csdn.net/zhzht19861011/article/details/50248421)一文中会详细介绍FreeRTOS内存管理的特性和用法，[《FreeRTOS内存管理分析》](http://blog.csdn.net/zhzht19861011/article/details/51606068)一文会从源码级别分析FreeRTOS内存管理的具体实现，这里不用多纠结，你也可以快速的浏览一下这两篇文章，里面或许有许多不懂的，但不要着急，先放过它们。

      FreeRTOS文件夹下的Demo文件夹中还包括各种演示例程，涉及多种架构的处理器以及多种编译器。FreeRTOS/Demo/Common/Minimal文件夹下的演示例程代码中，绝大部分对所有移植硬件接口都是适用的。FreeRTOS/Demo/Common/Full文件夹下的代码属于历史遗留代码，仅用于PC移植层。

### 2. 移植前的一些准备

* 一块具有Cortex-M3微处理器的硬件板子，并且保证板子可以正常运转。
* 下载FreeRTOS程序包（[《FreeRTOS历史版本更新记录》](http://blog.csdn.net/zhzht19861011/article/details/51525665)一文中有下载地址，这是我在CSDN下载频道做的镜像文件。如果你能忍受下载网速慢，也可以去官方网站下载。）
* 下载CMSIS-M3，其实主要是需要里面的core\_cm3.h文件（可以去ARM官方下载，如果你安装了keil 5或比较新的Keil 4 MDK编译器，在目录：Keil\ARM\CMSIS文件夹下也可以找到）

### 3.移植过程

#### 3.1 添加RTOS核心代码

         将tasks.c、queue.c和list.c这三个内核代码加入工程，将port.c和heap\_1.c这两个与处理器相关代码加入工程。port.c位于FreeRTOS\Source\portable\RVDS\ARM\_CM3文件夹下，heap\_1.c位于FreeRTOS/Source/portable/MemMang文件夹下。

#### 3.2 添加头文件路径

1. ...\FreeRTOS\Source\portable\RVDS\ARM\_CM3
2. …\FreeRTOS\Source\include

#### 3.3 编写FreeRTOSConfig.h文件

      对于刚接触FreeRTOS的用户来说，最简单方法是找一个类似的Demo工程，复制该工程下的FreeRTOSConfig.h文件，在这个基础上进行修改。详细的配置说明将在后续[**《FreeRTOS内核配置说明》**](http://blog.csdn.net/zhzht19861011/article/details/50134883)一文中给出，这里依然不必纠结。

#### 3.4 编写一些钩子函数

      如果你在FreeRTOSConfig.h中设置了configUSE\_TICK\_HOOK=1，则必须编写voidvApplicationTickHook( void )函数。该函数利用时间片中断，可以很方便的实现一个定时器功能。详见后续文章[《FreeRTOS内核配置说明》](http://blog.csdn.net/zhzht19861011/article/details/50134883)有关宏configUSE\_TICK\_HOOK一节。

      如果你在FreeRTOSConfig.h中设置了configCHECK\_FOR\_STACK\_OVERFLOW=1或=2，则必须编写voidvApplicationStackOverflowHook( xTaskHandle pxTask, signed char \*pcTaskName )函数，该函数用于检测堆栈溢出，详见后续文章《FreeRTOS内核配置说明》有关宏configCHECK\_FOR\_STACK\_OVERFLOW一节。

#### 3.5 检查硬件

      为了验证你的硬件板子是否可靠的工作，首先编写一个小程序片，比如闪烁一个LED灯或者发送一个字符等等，我们这里使用UART发送一个字符。代码如下所示（假设你已经配置好了启动代码，并正确配置了UART）：

1. #include"task.h"
2. #include"queue.h"
3. #include"list.h"
4. #include"portable.h"
5. #include"debug.h"
7. int main(void)
8. {
9. init\_rtos\_debug();           //初始化调试串口
10. MAP\_UARTCharPut('A');        //发送一个字符
11. while(1);
12. }

      如果硬件可以正常发送字符，说明硬件以及启动代码OK，可以进行下一步。

#### 3.6 挂接中断

      在Cortex-M3硬件下，FreeRTOS使用SysTick作为系统节拍时钟，使用SVC和PendSVC进行上下文切换。异常中断服务代码位于port.c文件中，FreeRTOS的作者已经为各种架构的CPU写好了这些代码，可以直接拿来用，需要用户做的，仅仅是将这些异常中断入口地址挂接到启动代码中。

      在startup.s中，**使用IMPORT关键字声明要挂接的异常中断服务函数名**，然后将：

1. DCD     SVC\_Handler            换成：   DCD     vPortSVCHandler
2. DCD     PendSV\_Handler         换成：   DCD     xPortPendSVHandler
3. DCD     SysTick\_Handler        换成：   DCD     xPortSysTickHandler

#### 3.7 建立第一个任务Task

      在步骤3.5中，我们为了[**测试**](http://lib.csdn.net/base/softwaretest)硬件是是否能够工作，编写了一个发送字符的小函数，这里我们将把这个小函数作为我们第一个任务要执行的主要代码：每隔1秒钟，发送一个字符。代码如下所示：

1. voidvTask(void \*pvParameters)
2. {
3. while(1)
4. {
5. MAP\_UARTCharPut(0x31);
6. vTaskDelay(1000/portTICK\_RATE\_MS);
7. }
8. }

      FreeRTOS的任务以及编写格式将在后续文章[**《FreeRTOS任务概述》**](http://blog.csdn.net/zhzht19861011/article/details/50312443)一文中详述，这里只是一个很简单的任务，先有有大体印象。这里面有一个API函数vTaskDelay()，这个函数用于延时，具体用法将在后续文章[**《FreeRTOS任务控制》**](http://blog.csdn.net/zhzht19861011/article/details/50454591)一文中详细介绍。这里不必在意太多的未知情况，因为后面会一点点将这些未知空间探索一遍的。

#### 3.8 设置节拍时钟

      这里我们使用SysTick定时器作为系统的节拍时钟，设定每隔10ms产生一次节拍中断。由于FreeRTOS对移植做了非常多的工作，以至于我们只需要在FreeRTOSConfig.h中配置好以下两个宏定义即可：

1. configCPU\_CLOCK\_HZ     (/\*你的硬件平台CPU系统时钟，Fcclk\*/)
2. configTICK\_RATE\_HZ       ((portTickType)100)

      第一个宏定义CPU系统时钟，也就是CPU执行时的频率。第二个宏定义FreeRTOS的时间片频率，这里定义为100，表明RTOS一秒钟可以切换100次任务，也就是每个时间片为10ms。

      在prot.c中，函数vPortSetupTimerInterrupt()设置节拍时钟。该函数根据上面的两个宏定义的参数，计算SysTick定时器的重装载数值寄存器，然后设置SysTick定时器的控制及状态寄存器，设置如下：使用内核时钟源、使能中断、使能SysTick定时器。另外，函数vPortSetupTimerInterrupt()由函数vTaskStartScheduler()调用，这个函数用于启动调度器。

#### 3.9设置中断优先级相关宏

      这里特别重要，因为涉及到中断优先级和中断嵌套。这里先给出基于Cortex-M3硬件（lpc177x\_8x系列微控制器）的一个配置例子，在FreeRTOSConfig.h中：

1. #ifdef \_\_NVIC\_PRIO\_BITS
2. #defineconfigPRIO\_BITS   \_\_NVIC\_PRIO\_BITS
3. #else
4. #defineconfigPRIO\_BITS   5  /\*lpc177x\_8x微处理器使用优先级寄存器的5位\*/
5. #endif
7. /\*设置内核使用的中断优先级\*/
8. #define configKERNEL\_INTERRUPT\_PRIORITY      ( 31 << (8 - configPRIO\_BITS) )
9. /\*定义RTOS可以屏蔽的最大中断优先级，大于这个优先级的中断，不受RTOS控制\*/
10. #defineconfigMAX\_SYSCALL\_INTERRUPT\_PRIORITY   ( 5<< (8 - configPRIO\_BITS) )

      后续文章[**《FreeRTOS内核配置说明》**](http://blog.csdn.net/zhzht19861011/article/details/50134883)会详细介绍这些宏的含义，对于Cortex-M内核，后续文章[**《Cortex-M内核使用FreeRTOS特别注意事项》**](http://blog.csdn.net/zhzht19861011/article/details/50135449)一文，会讲述这些宏与硬件的联系，那个时候你一定会清楚这些宏所定义的数字会对你的硬件产生什么影响的。现在，我们只需要知道他们很重要就足够了，没人能一口吃成胖子。

#### 3.10 设置其它宏

      还需要在FreeRTOSConfig.h设置一些必要的宏，这些宏如下所示：

1. #define configUSE\_PREEMPTION 1            //配置为1使用抢占式内核,配置为0使用时间片
2. #define configUSE\_IDLE\_HOOK  0            //设置为1使用空闲钩子；设置为0不使用空闲钩子
3. #define configMAX\_PRIORITIES      ( 5 )   //应用程序任务中可用优先级数目
4. #define configUSE\_TICK\_HOOK       0       //就设置为1使用时间片钩子，设置为0不使用
5. #define configMINIMAL\_STACK\_SIZE      ( ( unsigned short ) 80 )     //最小空闲堆栈
6. #define configTOTAL\_HEAP\_SIZE         ( ( size\_t ) ( 5 \* 1024 ) )   //内核总共可用RAM

#### 3.11 创建任务

      调用FreeRTOS提供的API函数来创建任务，代码如下所示：

1. xTaskCreate(vTask,"Task1",50,NULL,1,NULL);

      关于详细的创建任务API函数，会在后续文章[**《FreeRTOS任务创建和删除》**](http://blog.csdn.net/zhzht19861011/article/details/50371956)一文中介绍。

#### 3.12 开启调度器

      调用FreeRTOS提供的API函数来启动调度器，代码如下所示：

1. vTaskStartScheduler();

      关于详细的开启调度器API函数，会在后续文章[**《FreeRTOS内核控制》**](http://blog.csdn.net/zhzht19861011/article/details/50553099)一文中介绍。

      此时的main函数代码如下所示：

1. int main(void)
2. {
3. init\_rtos\_debug();                             //初始化调试串口
5. xTaskCreate(vTask,"Task1",50,NULL,1,NULL);
6. vTaskStartScheduler();
7. while(1);
8. }

### 4. 小结

      到这里，一个最基本的FreeRTOS应用程序就已经运行起来，将硬件板子接到PC的RS232串口，可以观察到每隔一秒钟，板子都会向PC发送一个指定的字符。

      回头看一下移植过程，FreeRTOS移植到Cortex-M3硬件是多么的简单，这一方面归功于FreeRTOS的设计师已经为移植做了大量工作，同时，新一代的Cortex-M3硬件也为操作系统增加了一些列便利特性，比如SysTick定时器和全新的中断及异常。

      但是移植成功也只是万里长征的第一步，因为这只是最简单的应用。我们还不清除FreeRTOS背后的机理、调度[**算法**](http://lib.csdn.net/base/datastructure)的面貌、甚至连信号量也都没有涉及。就本文的移植过程来看，我们也刻意忽略了很多细节，比如FreeRTOSConfig.h文件中的宏都有什么意义？改动后对RTOS有何影响？比如FreeRTOS任务API的细节、调度API的细节，再比如FreeRTOS的内存如何分配？如何进行堆栈溢出检查等等。

      所以，先不要沾沾自喜，曲折的道路还远没到来呢。

      接下来的很多篇文章会围绕这个最简单的移植例程做详细的讲解，要把本篇文章中刻意隐藏的细节一一拿出来。这要一直持续到我们介绍队列、信号量、互斥量等通讯机制为止。

# 6 [FreeRTOS内核配置说明](http://blog.csdn.net/zhzht19861011/article/details/50134883)

FreeRTOS内核是高度可定制的，使用配置文件FreeRTOSConfig.h进行定制。每个FreeRTOS应用都必须包含这个头文件，用户根据实际应用来裁剪定制FreeRTOS内核。这个配置文件是针对用户程序的，而非内核，因此配置文件一般放在应用程序目录下，不要放在RTOS内核源码目录下。

       在下载的FreeRTOS文件包中，每个演示例程都有一个FreeRTOSConfig.h文件。有些例程的配置文件是比较旧的版本，可能不会包含所有有效选项。如果没有在配置文件中指定某个选项，那么RTOS内核会使用默认值。典型的FreeRTOSConfig.h配置文件定义如下所示，随后会说明里面的每一个参数。

1. #ifndef FREERTOS\_CONFIG\_H
2. #define FREERTOS\_CONFIG\_H
4. /\*Here is a good place to include header files that are required across
5. yourapplication. \*/
6. #include "something.h"
8. #define configUSE\_PREEMPTION                    1
9. #define configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 0
10. #define configUSE\_TICKLESS\_IDLE                 0
11. #define configCPU\_CLOCK\_HZ                      60000000
12. #define configTICK\_RATE\_HZ                      250
13. #define configMAX\_PRIORITIES                    5
14. #define configMINIMAL\_STACK\_SIZE                128
15. #define configTOTAL\_HEAP\_SIZE                   10240
16. #define configMAX\_TASK\_NAME\_LEN                 16
17. #define configUSE\_16\_BIT\_TICKS                  0
18. #define configIDLE\_SHOULD\_YIELD                 1
19. #define configUSE\_TASK\_NOTIFICATIONS            1
20. #define configUSE\_MUTEXES                       0
21. #define configUSE\_RECURSIVE\_MUTEXES             0
22. #define configUSE\_COUNTING\_SEMAPHORES           0
23. #define configUSE\_ALTERNATIVE\_API               0/\* Deprecated! \*/
24. #define configQUEUE\_REGISTRY\_SIZE               10
25. #define configUSE\_QUEUE\_SETS                    0
26. #define configUSE\_TIME\_SLICING                  0
27. #define configUSE\_NEWLIB\_REENTRANT              0
28. #define configENABLE\_BACKWARD\_COMPATIBILITY     0
29. #define configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS 5
31. /\*Hook function related definitions. \*/
32. #define configUSE\_IDLE\_HOOK                     0
33. #define configUSE\_TICK\_HOOK                     0
34. #define configCHECK\_FOR\_STACK\_OVERFLOW          0
35. #define configUSE\_MALLOC\_FAILED\_HOOK            0
37. /\*Run time and task stats gathering related definitions. \*/
38. #define configGENERATE\_RUN\_TIME\_STATS           0
39. #define configUSE\_TRACE\_FACILITY                0
40. #define configUSE\_STATS\_FORMATTING\_FUNCTIONS    0
42. /\*Co-routine related definitions. \*/
43. #define configUSE\_CO\_ROUTINES                   0
44. #define configMAX\_CO\_ROUTINE\_PRIORITIES         1
46. /\*Software timer related definitions. \*/
47. #define configUSE\_TIMERS                        1
48. #define configTIMER\_TASK\_PRIORITY               3
49. #define configTIMER\_QUEUE\_LENGTH                10
50. #define configTIMER\_TASK\_STACK\_DEPTH            configMINIMAL\_STACK\_SIZE
52. /\*Interrupt nesting behaviour configuration. \*/
53. #define configKERNEL\_INTERRUPT\_PRIORITY        [dependent of processor]
54. #define configMAX\_SYSCALL\_INTERRUPT\_PRIORITY   [dependent on processor and application]
55. #define configMAX\_API\_CALL\_INTERRUPT\_PRIORITY  [dependent on processor and application]
57. /\*Define to trap errors during development. \*/
58. #define configASSERT( ( x ) )     if( ( x ) == 0) vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )
60. /\*FreeRTOS MPU specific definitions. \*/
61. #define configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS 0
63. /\*Optional functions - most linkers will remove unused functions anyway. \*/
64. #define INCLUDE\_vTaskPrioritySet                1
65. #define INCLUDE\_uxTaskPriorityGet               1
66. #define INCLUDE\_vTaskDelete                     1
67. #define INCLUDE\_vTaskSuspend                    1
68. #define INCLUDE\_xResumeFromISR                  1
69. #define INCLUDE\_vTaskDelayUntil                 1
70. #define INCLUDE\_vTaskDelay                      1
71. #define INCLUDE\_xTaskGetSchedulerState          1
72. #define INCLUDE\_xTaskGetCurrentTaskHandle       1
73. #define INCLUDE\_uxTaskGetStackHighWaterMark     0
74. #define INCLUDE\_xTaskGetIdleTaskHandle          0
75. #define INCLUDE\_xTimerGetTimerDaemonTaskHandle  0
76. #define INCLUDE\_pcTaskGetTaskName               0
77. #define INCLUDE\_eTaskGetState                   0
78. #define INCLUDE\_xEventGroupSetBitFromISR        1
79. #define INCLUDE\_xTimerPendFunctionCall          0
81. /\* Aheader file that defines trace macro can be included here. \*/
83. #end if/\* FREERTOS\_CONFIG\_H\*/

**1.configUSE\_PREEMPTION**

       为1时RTOS使用抢占式调度器，为0时RTOS使用协作式调度器（时间片）。

      注：在多任务管理机制上，[**操作系统**](http://lib.csdn.net/base/operatingsystem)可以分为抢占式和协作式两种。协作式操作系统是任务主动释放CPU后，切换到下一个任务。任务切换的时机完全取决于正在运行的任务。

**2.configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION**

       某些运行FreeRTOS的硬件有两种方法选择下一个要执行的任务：通用方法和特定于硬件的方法（以下简称“特殊方法”）。

       通用方法：

* configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION设置为0或者硬件不支持这种特殊方法。
* 可以用于所有FreeRTOS支持的硬件。
* 完全用C实现，效率略低于特殊方法。
* 不强制要求限制最大可用优先级数目

       特殊方法：

* 并非所有硬件都支持。
* 必须将configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION设置为1。
* 依赖一个或多个特定架构的汇编指令（一般是类似计算前导零[CLZ]指令）。
* 比通用方法更高效。
* 一般强制限定最大可用优先级数目为32。

**3.configUSE\_TICKLESS\_IDLE**

       设置configUSE\_TICKLESS\_IDLE为1使能低功耗tickless模式，为0保持系统节拍（tick）中断一直运行。

       通常情况下，FreeRTOS回调空闲任务钩子函数（需要设计者自己实现），在空闲任务钩子函数中设置微处理器进入低功耗模式来达到省电的目的。因为系统要响应系统节拍中断事件，因此使用这种方法会周期性的退出、再进入低功耗状态。如果系统节拍中断频率过快，则大部分电能和CPU时间会消耗在进入和退出低功耗状态上。

       FreeRTOS的tickless空闲模式会在空闲周期时停止周期性系统节拍中断。停止周期性系统节拍中断可以使微控制器长时间处于低功耗模式。移植层需要配置外部唤醒中断，当唤醒事件到来时，将微控制器从低功耗模式唤醒。微控制器唤醒后，会重新使能系统节拍中断。由于微控制器在进入低功耗后，系统节拍计数器是停止的，但我们又需要知道这段时间能折算成多少次系统节拍中断周期，这就需要有一个不受低功耗影响的外部时钟源，即微处理器处于低功耗模式时它也在计时的，这样在重启系统节拍中断时就可以根据这个外部计时器计算出一个调整值并写入RTOS 系统节拍计数器变量中。

**4.configUSE\_IDLE\_HOOK**

       设置为1使用空闲钩子（Idle Hook类似于回调函数），0忽略空闲钩子。

       当RTOS调度器开始工作后，为了保证至少有一个任务在运行，空闲任务被自动创建，占用最低优先级（0优先级）。对于已经删除的RTOS任务，空闲任务可以释放分配给它们的堆栈内存。因此，在应用中应该注意，使用vTaskDelete()函数时要确保空闲任务获得一定的处理器时间。除此之外，空闲任务没有其它特殊功能，因此可以任意的剥夺空闲任务的处理器时间。

       应用程序也可能和空闲任务共享同个优先级。

       空闲任务钩子是一个函数，这个函数由用户来实现，RTOS规定了函数的名字和参数，这个函数在每个空闲任务周期都会被调用。

       要创建一个空闲钩子：

1. 设置FreeRTOSConfig.h 文件中的configUSE\_IDLE\_HOOK 为1；
2. 定义一个函数，函数名和参数如下所示：
3. void vApplicationIdleHook(void );

       这个钩子函数不可以调用会引起空闲任务阻塞的API函数（例如：vTaskDelay()、带有阻塞时间的队列和信号量函数），在钩子函数内部使用协程是被允许的。

       使用空闲钩子函数设置CPU进入省电模式是很常见的。

**5.configUSE\_MALLOC\_FAILED\_HOOK**

       每当一个任务、队列、信号量被创建时，内核使用一个名为pvPortMalloc()的函数来从堆中分配内存。官方的下载包中包含5个简单内存分配策略，分别保存在源文件heap\_1.c、heap\_2.c、heap\_3.c、heap\_4.c、heap\_5.c中。        仅当使用这五个简单策略之一时，宏configUSE\_MALLOC\_FAILED\_HOOK才有意义。

       如果定义并正确配置malloc()失败钩子函数，则这个函数会在pvPortMalloc()函数返回NULL时被调用。只有FreeRTOS在响应内存分配请求时发现堆内存不足才会返回NULL。

       如果宏configUSE\_MALLOC\_FAILED\_HOOK设置为1，那么必须定义一个malloc()失败钩子函数，如果宏configUSE\_MALLOC\_FAILED\_HOOK设置为0，malloc()失败钩子函数不会被调用，即便已经定义了这个函数。malloc()失败钩子函数的函数名和原型必须如下所示：

1. void vApplicationMallocFailedHook( void);

**6.configUSE\_TICK\_HOOK**

       设置为1使用时间片钩子（Tick Hook），0忽略时间片钩子。

       注：时间片钩子函数（Tick Hook Function）

       时间片中断可以周期性的调用一个被称为钩子函数（回调函数）的应用程序。时间片钩子函数可以很方便的实现一个定时器功能。

       只有在FreeRTOSConfig.h中的configUSE\_TICK\_HOOK设置成1时才可以使用时间片钩子。一旦此值设置成1，就要定义钩子函数，函数名和参数如下所示：

1. void vApplicationTickHook( void );

       vApplicationTickHook()函数在中断服务程序中执行，因此这个函数必须非常短小，不能大量使用堆栈，不能调用以”FromISR" 或 "FROM\_ISR”结尾的API函数。

       在FreeRTOSVx.x.x\FreeRTOS\Demo\Common\Minimal文件夹下的crhook.c文件中有使用时间片钩子函数的例程。

**7.configCPU\_CLOCK\_HZ**

       写入实际的CPU内核时钟频率，也就是CPU指令执行频率，通常称为Fcclk。配置此值是为了正确的配置系统节拍中断周期。

**8.configTICK\_RATE\_HZ**

       RTOS 系统节拍中断的频率。即一秒中断的次数，每次中断RTOS都会进行任务调度。

系统节拍中断用来测量时间，因此，越高的测量频率意味着可测到越高的分辨率时间。但是，高的系统节拍中断频率也意味着RTOS内核占用更多的CPU时间，因此会降低效率。RTOS演示例程都是使用系统节拍中断频率为1000HZ，这是为了[**测试**](http://lib.csdn.net/base/softwaretest)RTOS内核，比实际使用的要高。（实际使用时不用这么高的系统节拍中断频率）

       多个任务可以共享一个优先级，RTOS调度器为相同优先级的任务分享CPU时间，在每一个RTOS 系统节拍中断到来时进行任务切换。高的系统节拍中断频率会降低分配给每一个任务的“时间片”持续时间。

**9.configMAX\_PRIORITIES**

       配置应用程序有效的优先级数目。任何数量的任务都可以共享一个优先级，使用协程可以单独的给与它们优先权。见configMAX\_CO\_ROUTINE\_PRIORITIES。

       在RTOS内核中，每个有效优先级都会消耗一定量的RAM，因此这个值不要超过你的应用实际需要的优先级数目。

注：任务优先级

       每一个任务都会被分配一个优先级，优先级值从0~ （configMAX\_PRIORITIES - 1）之间。低优先级数表示低优先级任务。空闲任务的优先级为0（tskIDLE\_PRIORITY），因此它是最低优先级任务。

       FreeRTOS调度器将确保处于就绪状态（Ready）或运行状态（Running）的高优先级任务比同样处于就绪状态的低优先级任务优先获取处理器时间。换句话说，处于运行状态的任务永远是高优先级任务。

       处于就绪状态的相同优先级任务使用时间片调度机制共享处理器时间。

**10.configMINIMAL\_STACK\_SIZE**

       定义空闲任务使用的堆栈大小。通常此值不应小于对应处理器演示例程文件FreeRTOSConfig.h中定义的数值。

       就像xTaskCreate()函数的堆栈大小参数一样，堆栈大小不是以字节为单位而是以字为单位的，比如在32位[**架构**](http://lib.csdn.net/base/architecture)下，栈大小为100表示栈内存占用400字节的空间。

**11.configTOTAL\_HEAP\_SIZE**

       RTOS内核总计可用的有效的RAM大小。仅在你使用官方下载包中附带的内存分配策略时，才有可能用到此值。每当创建任务、队列、互斥量、软件定时器或信号量时，RTOS内核会为此分配RAM，这里的RAM都属于configTOTAL\_HEAP\_SIZE指定的内存区。后续的内存配置会详细讲到官方给出的内存分配策略。

**12.configMAX\_TASK\_NAME\_LEN**

       调用任务函数时，需要设置描述任务信息的字符串，这个宏用来定义该字符串的最大长度。这里定义的长度包括字符串结束符’\0’。

**13.configUSE\_TRACE\_FACILITY**

       设置成1表示启动可视化跟踪调试，会激活一些附加的结构体成员和函数。

**14.configUSE\_STATS\_FORMATTING\_FUNCTIONS （V7.5.0新增）**

       设置宏configUSE\_TRACE\_FACILITY和configUSE\_STATS\_FORMATTING\_FUNCTIONS为1会编译vTaskList()和vTaskGetRunTimeStats()函数。如果将这两个宏任意一个设置为0，上述两个函数不会被编译。

**15.configUSE\_16\_BIT\_TICKS**

       定义系统节拍计数器的变量类型，即定义portTickType是表示16位变量还是32位变量。

       定义configUSE\_16\_BIT\_TICKS为1意味着portTickType代表16位无符号整形，定义configUSE\_16\_BIT\_TICKS为0意味着portTickType代表32位无符号整形。

       使用16位类型可以大大提高8位和16位架构微处理器的性能，但这也限制了最大时钟计数为65535个’Tick’。因此，如果Tick频率为250HZ（4MS中断一次），对于任务最大延时或阻塞时间，16位计数器是262秒，而32位是17179869秒。

**16.configIDLE\_SHOULD\_YIELD**

       这个参数控制任务在空闲优先级中的行为。仅在满足下列条件后，才会起作用。

1. 使用抢占式内核调度
2. 用户任务使用空闲优先级。

         通过时间片共享同一个优先级的多个任务，如果共享的优先级大于空闲优先级，并假设没有更高优先级任务，这些任务应该获得相同的处理器时间。

         但如果共享空闲优先级时，情况会稍微有些不同。当configIDLE\_SHOULD\_YIELD为1时，其它共享空闲优先级的用户任务就绪时，空闲任务立刻让出CPU，用户任务运行，这样确保了能最快响应用户任务。处于这种模式下也会有不良效果（取决于你的程序需要），描述如下：

http://img.blog.csdn.net/20151201205910254

       图中描述了四个处于空闲优先级的任务，任务A、B和C是用户任务，任务I是空闲任务。上下文切换周期性的发生在T0、T1…T6时刻。当用户任务运行时，空闲任务立刻让出CPU，但是，空闲任务已经消耗了当前时间片中的一定时间。这样的结果就是空闲任务I和用户任务A共享一个时间片。用户任务B和用户任务C因此获得了比用户任务A更多的处理器时间。

       可以通过下面方法避免：

* 如果合适的话，将处于空闲优先级的各单独的任务放置到空闲钩子函数中；
* 创建的用户任务优先级大于空闲优先级；
* 设置IDLE\_SHOULD\_YIELD为0；

       设置configIDLE\_SHOULD\_YIELD为0将阻止空闲任务为用户任务让出CPU，直到空闲任务的时间片结束。这确保所有处在空闲优先级的任务分配到相同多的处理器时间，但是，这是以分配给空闲任务更高比例的处理器时间为代价的。

**17.configUSE\_TASK\_NOTIFICATIONS（V8.2.0新增）**

       设置宏configUSE\_TASK\_NOTIFICATIONS为1（或不定义宏configUSE\_TASK\_NOTIFICATIONS）将会开启任务通知功能，有关的API函数也会被编译。设置宏configUSE\_TASK\_NOTIFICATIONS为0则关闭任务通知功能，相关API函数也不会被编译。默认这个功能是开启的。开启后，每个任务多增加8字节RAM。

       这是个很有用的特性，一大亮点。

       每个RTOS任务具有一个32位的通知值，RTOS任务通知相当于直接向任务发送一个事件，接收到通知的任务可以解除任务的阻塞状态（因等待任务通知而进入阻塞状态）。相对于以前必须分别创建队列、二进制信号量、计数信号量或事件组的情况，使用任务通知显然更灵活。更好的是，相比于使用信号量解除任务阻塞，使用任务通知可以快45%（使用GCC编译器，-o2优化级别）。

**18.configUSE\_MUTEXES**

       设置为1表示使用互斥量，设置成0表示忽略互斥量。读者应该了解在FreeRTOS中互斥量和二进制信号量的区别。

       关于互斥量和二进制信号量简单说：

* 互斥型信号量必须是同一个任务申请，同一个任务释放，其他任务释放无效。
* 二进制信号量，一个任务申请成功后，可以由另一个任务释放。
* 互斥型信号量是二进制信号量的子集

**19.configUSE\_RECURSIVE\_MUTEXES**

       设置成1表示使用递归互斥量，设置成0表示不使用。

**20.configUSE\_COUNTING\_SEMAPHORES**

       设置成1表示使用计数信号量，设置成0表示不使用。

**21.configUSE\_ALTERNATIVE\_API**

       设置成1表示使用“替代”队列函数（'alternative' queue functions），设置成0不使用。替代API在queue.h头文件中有详细描述。

**注：“替代”队列函数已经被弃用，在新的设计中不要使用它！**

**22.configCHECK\_FOR\_STACK\_OVERFLOW**

       每个任务维护自己的栈空间，任务创建时会自动分配任务需要的占内存，分配内存大小由创建任务函数（xTaskCreate()）的一个参数指定。堆栈溢出是设备运行不稳定的最常见原因，因此FreeeRTOS提供了两个可选机制用来辅助检测和改正堆栈溢出。配置宏configCHECK\_FOR\_STACK\_OVERFLOW为不同的常量来使用不同堆栈溢出检测机制。

       注意，这个选项仅适用于内存映射未分段的微处理器架构。并且，在RTOS检测到堆栈溢出发生之前，一些处理器可能先产生故障（fault）或异常（exception）来反映堆栈使用的恶化。如果宏configCHECK\_FOR\_STACK\_OVERFLOW没有设置成0，用户必须提供一个栈溢出钩子函数，这个钩子函数的函数名和参数必须如下所示：

1. void vApplicationStackOverflowHook(TaskHandle\_t xTask, signed char \*pcTaskName );

参数xTask和pcTaskName为堆栈溢出任务的句柄和名字。请注意，如果溢出非常严重，这两个参数信息也可能是错误的！在这种情况下，可以直接检查pxCurrentTCb变量。

       推荐仅在开发或测试阶段使用栈溢出检查，因为堆栈溢出检测会增大上下文切换开销。

       任务切换出去后，该任务的上下文环境被保存到自己的堆栈空间，这时很可能堆栈的使用量达到了最大（最深）值。在这个时候，RTOS内核会检测堆栈指针是否还指向有效的堆栈空间。如果堆栈指针指向了有效堆栈空间之外的地方，堆栈溢出钩子函数会被调用。

       这个方法速度很快，但是不能检测到所有堆栈溢出情况（比如，堆栈溢出没有发生在上下文切换时）。设置configCHECK\_FOR\_STACK\_OVERFLOW为1会使用这种方法。

       当堆栈首次创建时，在它的堆栈区中填充一些已知值（标记）。当任务切换时，RTOS内核会检测堆栈最后的16个字节，确保标记数据没有被覆盖。如果这16个字节有任何一个被改变，则调用堆栈溢出钩子函数。

       这个方法比第一种方法要慢，但也相当快了。它能有效捕捉堆栈溢出事件（即使堆栈溢出没有发生在上下文切换时），但是理论上它也不能百分百的捕捉到所有堆栈溢出（比如堆栈溢出的值和标记值相同，当然，这种情况发生的概率极小）。

       使用这个方法需要设置configCHECK\_FOR\_STACK\_OVERFLOW为2.

**23.configQUEUE\_REGISTRY\_SIZE**

       队列记录有两个目的，都涉及到RTOS内核的调试：

1. 它允许在调试GUI中使用一个队列的文本名称来简单识别队列；
2. 包含调试器需要的每一个记录队列和信号量定位信息；

       除了进行内核调试外，队列记录没有其它任何目的。

       configQUEUE\_REGISTRY\_SIZE定义可以记录的队列和信号量的最大数目。如果你想使用RTOS内核调试器查看队列和信号量信息，则必须先将这些队列和信号量进行注册，只有注册后的队列和信号量才可以使用RTOS内核调试器查看。查看API参考手册中的vQueueAddToRegistry() 和vQueueUnregisterQueue()函数获取更多信息。

**24.configUSE\_QUEUE\_SETS**

       设置成1使能队列集功能（可以阻塞、挂起到多个队列和信号量），设置成0取消队列集功能。

**25.configUSE\_TIME\_SLICING（V7.5.0新增）**

       默认情况下（宏configUSE\_TIME\_SLICING未定义或者宏configUSE\_TIME\_SLICING设置为1），FreeRTOS使用基于时间片的优先级抢占式调度器。这意味着RTOS调度器总是运行处于最高优先级的就绪任务，在每个RTOS 系统节拍中断时在相同优先级的多个任务间进行任务切换。如果宏configUSE\_TIME\_SLICING设置为0，RTOS调度器仍然总是运行处于最高优先级的就绪任务，但是当RTOS 系统节拍中断发生时，相同优先级的多个任务之间不再进行任务切换。

**26.configUSE\_NEWLIB\_REENTRANT（V7.5.0新增）**

       如果宏configUSE\_NEWLIB\_REENTRANT设置为1，每一个创建的任务会分配一个newlib（一个[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)C库）reent结构。

**27.configENABLE\_BACKWARD\_COMPATIBILITY**

       头文件FreeRTOS.h包含一系列#define宏定义，用来映射版本V8.0.0和V8.0.0之前版本的数据类型名字。这些宏可以确保RTOS内核升级到V8.0.0或以上版本时，之前的应用代码不用做任何修改。在FreeRTOSConfig.h文件中设置宏configENABLE\_BACKWARD\_COMPATIBILITY为0会去掉这些宏定义，并且需要用户确认升级之前的应用没有用到这些名字。

**28.configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS**

       设置每个任务的线程本地存储指针数组大小。

       线程本地存储允许应用程序在任务的控制块中存储一些值，每个任务都有自己独立的储存空间，宏configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS指定每个任务线程本地存储指针数组的大小。API函数vTaskSetThreadLocalStoragePointer()用于向指针数组中写入值，API函数pvTaskGetThreadLocalStoragePointer()用于从指针数组中读取值。

       比如，许多库函数都包含一个叫做errno的全局变量。某些库函数使用errno返回库函数错误信息，应用程序检查这个全局变量来确定发生了那些错误。在单线程程序中，将errno定义成全局变量是可以的，但是在多线程应用中，每个线程（任务）必须具有自己独有的errno值，否则，一个任务可能会读取到另一个任务的errno值。

       FreeRTOS提供了一个灵活的机制，使得应用程序可以使用线程本地存储指针来读写线程本地存储。具体参见后续文章《FreeRTOS系列第12篇---FreeRTOS任务应用函数》。

**29.configGENERATE\_RUN\_TIME\_STATS**

       设置宏configGENERATE\_RUN\_TIME\_STATS为1使能运行时间统计功能。一旦设置为1，则下面两个宏必须被定义：

1. portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS()：用户程序需要提供一个基准时钟函数，函数完成初始化基准时钟功能，这个函数要被define到宏portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS()上。这是因为运行时间统计需要一个比系统节拍中断频率还要高分辨率的基准定时器，否则，统计可能不精确。基准定时器中断频率要比统节拍中断快10~100倍。基准定时器中断频率越快，统计越精准，但能统计的运行时间也越短（比如，基准定时器10ms中断一次，8位无符号整形变量可以计到2.55秒，但如果是1秒中断一次，8位无符号整形变量可以统计到255秒）。
2. portGET\_RUN\_TIME\_COUNTER\_VALUE()：用户程序需要提供一个返回基准时钟当前“时间”的函数，这个函数要被define到宏portGET\_RUN\_TIME\_COUNTER\_VALUE()上。

       举一个例子，假如我们配置了一个定时器，每500us中断一次。在定时器中断服务例程中简单的使长整形变量ulHighFrequencyTimerTicks自增。那么上面提到两个宏定义如下（可以在FreeRTOSConfig.h中添加）：

1. extern volatile unsigned longulHighFrequencyTimerTicks;
2. #define portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() ( ulHighFrequencyTimerTicks = 0UL )
3. #define portGET\_RUN\_TIME\_COUNTER\_VALUE() ulHighFrequencyTimerTicks

**30.configUSE\_CO\_ROUTINES**

       设置成1表示使用协程，0表示不使用协程。如果使用协程，必须在工程中包含croutine.c文件。

       注：协程（Co-routines）主要用于资源发非常受限的嵌入式系统（RAM非常少），通常不会用于32位微处理器。

       在当前嵌入式硬件环境下，不建议使用协程，FreeRTOS的开发者早已经停止开发协程。

**31.configMAX\_CO\_ROUTINE\_PRIORITIES**

       应用程序协程（Co-routines）的有效优先级数目，任何数目的协程都可以共享一个优先级。使用协程可以单独的分配给任务优先级。见configMAX\_PRIORITIES。

**32.configUSE\_TIMERS**

       设置成1使用软件定时器，为0不使用软件定时器功能。详细描述见FreeRTOS software timers 。

**33.configTIMER\_TASK\_PRIORITY**

       设置软件定时器服务/守护进程的优先级。详细描述见FreeRTOS software timers 。

**34.configTIMER\_QUEUE\_LENGTH**

       设置软件定时器命令队列的长度。详细描述见FreeRTOS software timers。

**35.configTIMER\_TASK\_STACK\_DEPTH**

       设置软件定时器服务/守护进程任务的堆栈深度，详细描述见FreeRTOS software timers 。

**36.configKERNEL\_INTERRUPT\_PRIORITY、configMAX\_SYSCALL\_INTERRUPT\_PRIORITY和configMAX\_API\_CALL\_INTERRUPT\_PRIORITY**

这是移植和应用FreeRTOS出错最多的地方，所以需要打起精神仔细读懂。

       Cortex-M3、PIC24、dsPIC、PIC32、SuperH和RX600硬件设备需要设置宏configKERNEL\_INTERRUPT\_PRIORITY；PIC32、RX600和Cortex-M硬件设备需要设置宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY。

       configMAX\_SYSCALL\_INTERRUPT\_PRIORITY和configMAX\_API\_CALL\_INTERRUPT\_PRIORITY，这两个宏是等价的，后者是前者的新名字，用于更新的移植层代码。

       注意下面的描述中，在中断服务例程中仅可以调用以“FromISR”结尾的API函数。

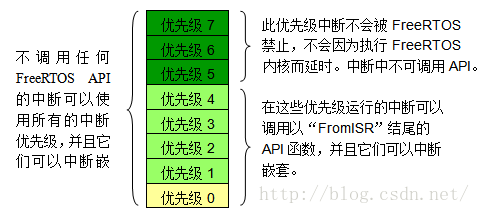
* **仅需要**设置configKERNEL\_INTERRUPT\_PRIORITY的硬件设备（也就是宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY不会用到）：configKERNEL\_INTERRUPT\_PRIORITY用来设置RTOS内核自己的中断优先级。调用API函数的中断必须运行在这个优先级；不调用API函数的中断，可以运行在更高的优先级，所以这些中断不会被因RTOS内核活动而延时。
* configKERNEL\_INTERRUPT\_PRIORITY和configMAX\_SYSCALL\_INTERRUPT\_PRIORITY**都需要**设置的硬件设备：configKERNEL\_INTERRUPT\_PRIORITY用来设置RTOS内核自己的中断优先级。因为RTOS内核中断不允许抢占用户使用的中断，因此这个宏一般定义为硬件最低优先级。configMAX\_SYSCALL\_INTERRUPT\_PRIORITY用来设置可以在中断服务程序中安全调用FreeRTOS API函数的最高中断优先级。优先级小于等于这个宏所代表的优先级时，程序可以在中断服务程序中安全的调用FreeRTOS API函数；如果优先级大于这个宏所代表的优先级，表示FreeRTOS无法禁止这个中断，在这个中断服务程序中绝不可以调用任何API函数。

       通过设置configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的优先级级别高于configKERNEL\_INTERRUPT\_PRIORITY可以实现完整的中断嵌套模式。这意味着FreeRTOS内核不能完全禁止中断，即使在临界区。此外，这对于分段内核架构的微处理器是有利的。请注意，当一个新中断发生后，某些微处理器架构会（在硬件上）禁止中断，这意味着从硬件响应中断到FreeRTOS重新使能中断之间的这段短时间内，中断是不可避免的被禁止的。

       不调用API的中断可以运行在比configMAX\_SYSCALL\_INTERRUPT\_PRIORITY高的优先级，这些级别的中断不会被FreeRTOS禁止，因此不会因为执行RTOS内核而被延时。

       例如：假如一个微控制器有8个中断优先级别：0表示最低优先级，7表示最高优先级（Cortex-M3和Cortex-M4内核优先数和优先级别正好与之相反，后续文章会专门介绍它们）。当两个配置选项分别为4和0时，下图描述了每一个优先级别可以和不可做的事件：

* configMAX\_SYSCALL\_INTERRUPT\_PRIORITY=4
* configKERNEL\_INTERRUPT\_PRIORITY=0



       这些配置参数允许非常灵活的中断处理：

       在系统中可以像其它任务一样为中断处理任务分配优先级。这些任务通过一个相应中断唤醒。中断服务例程（ISR）内容应尽可能的精简---仅用于更新数据然后唤醒高优先级任务。ISR退出后，直接运行被唤醒的任务，因此中断处理（根据中断获取的数据来进行的相应处理）在时间上是连续的，就像ISR在完成这些工作。这样做的好处是当中断处理任务执行时，所有中断都可以处在使能状态。

       中断、中断服务例程（ISR）和中断处理任务是三码事：当中断来临时会进入中断服务例程，中断服务例程做必要的数据收集（更新），之后唤醒高优先级任务。这个高优先级任务在中断服务例程结束后立即执行，它可能是其它任务也可能是中断处理任务，如果是中断处理任务，那么就可以根据中断服务例程中收集的数据做相应处理。

       configMAX\_SYSCALL\_INTERRUPT\_PRIORITY接口有着更深一层的意义：在优先级介于RTOS内核中断优先级（等于configKERNEL\_INTERRUPT\_PRIORITY）和configMAX\_SYSCALL\_INTERRUPT\_PRIORITY之间的中断允许全嵌套中断模式并允许调用API函数。大于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的中断优先级绝不会因为执行RTOS内核而延时。

       运行在大于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的优先级中断是不会被RTOS内核所屏蔽的，因此也不受RTOS内核功能影响。这主要用于非常高的实时需求中。比如执行电机转向。但是，这类中断的中断服务例程中绝不可以调用FreeRTOS的API函数。

      为了使用这个方案，应用程序要必须符合以下规则：调用FreeRTOS API函数的任何中断，都必须和RTOS内核处于同一优先级（由宏configKERNEL\_INTERRUPT\_PRIORITY设置），或者小于等于宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY定义的优先级。

**37.configASSERT**

       断言，调试时可以检查传入的参数是否合法。FreeRTOS内核代码的关键点都会调用configASSERT( x )函数，如果参数x为0，则会抛出一个错误。这个错误很可能是传递给FreeRTOS API函数的无效参数引起的。定义configASSERT()有助于调试时发现错误，但是，定义configASSERT()也会增大应用程序代码量，增大运行时间。推荐在开发阶段使用这个断言宏。

       举一个例子，我们想把非法参数所在的文件名和代码行数打印出来，可以先定义一个函数vAssertCalled，该函数有两个参数，分别接收触发configASSERT宏的文件名和该宏所在行，然后通过显示屏或者串口输出。代码如下：

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

       这里\_\_FILE\_\_和\_\_LINE\_\_是大多数编译器预定义的宏，分别表示代码所在的文件名（字符串格式）和行数（整形）。

       这个例子虽然看起来很简单，但由于要把整形\_\_LINE\_\_转换成字符串再显示，在效率和实现上，都不能让人满意。我们可以使用C标准库assert的实现方法，这样函数vAssertCalled只需要接收一个字符串形式的参数（推荐仔细研读下面的代码并理解其中的技巧）：

1. #define STR(x)  VAL(x)
2. #define VAL(x)  #x
3. #define configASSERT(x) ((x)?(void) 0 :xAssertCalld(\_\_FILE\_\_ ":" STR(\_\_LINE\_\_) " " #x"\n"))

       这里稍微讲解一下，由于内置宏\_\_LINE\_\_是整数型的而不是字符串型，把它转化成字符串需要一个额外的处理层。宏STR和和宏VAL正是用来辅助完成这个转化。宏STR用来把整形行号替换掉\_\_LINE\_\_，宏VAL用来把这个整形行号字符串化。忽略宏STR和VAL中的任何一个，只能得到字符串”\_\_LINE\_\_”，这不是我们想要的。

       这里使用三目运算符’?:’来代替参数判断if语句，这样可以接受任何参数或表达式，代码也更紧凑，更重要的是代码优化度更高，因为如果参数恒为真，在编译阶段就可以去掉不必要的输出语句。

**38.INCLUDE Parameters**

       以“INCLUDE”起始的宏允许用户不编译那些应用程序不需要的实时内核组件（函数），这可以确保在你的嵌入式系统中RTOS占用最少的ROM和RAM。

每个宏以这样的形式出现：

       INCLUDE\_FunctionName

      在这里FunctionName表示一个你可以控制是否编译的API函数。如果你想使用该函数，就将这个宏设置成1，如果不想使用，就将这个宏设置成0。比如，对于API函数vTaskDelete()：

1. #define INCLUDE\_vTaskDelete    1

       表示希望使用vTaskDelete()，允许编译器编译该函数

1. #define INCLUDE\_vTaskDelete    0

       表示禁止编译器编译该函数。

# 7 [Cortex-M内核使用FreeRTOS特别注意事项](http://blog.csdn.net/zhzht19861011/article/details/50135449)

在阅读本文之前，有两个定义在FreeRTOSConfig.h中的宏，你必须先明白它们是什么意思，[《FreeRTOS内核配置说明》](http://blog.csdn.net/zhzht19861011/article/details/50134883)一文中，讲解了这两个宏：

* configKERNEL\_INTERRUPT\_PRIORITY
* configMAX\_SYSCALL\_INTERRUPT\_PRIORITY

       FreeRTOS与Cortex-M内核可谓是绝配，以至于让移植和使用FreeRTOS都变得更简单起来。根据FreeRTOS官方反馈，在Cortex-M内核上使用FreeRTOS大多数的问题点是由不正确的优先级设置引起的。这个问题也是在意料之中的，因为尽管Cortex-M内核的中断模式是非常强大的，但对于那些使用传统中断优先级架构的工程师来说，Cortex-M内核中断机制也有点笨拙（或者是说使用比较繁琐），并且违反直觉（这个主要是因为Cortex-M中断优先级数值越大代表的优先级反而越小）。本章打算描述Cortex-M的中断优先级机制，并描述怎样结合RTOS内核使用。

       说明：虽然Cortex-M内核的优先级方案看上去比较复杂，但每一个官方发布的FreeRTOS 接口包（在FreeRTOSV7.2.0\FreeRTOS\Source\portable文件夹中，一般为port.c）内都会有正确配置的演示例程，可以以此为参考。

### ****1.有效优先级****

#### ****1.1Cortex-M 硬件详述****

       首先需要清楚有效优先级的总数，这取决于微控制器制造商怎么使用Cortex内核。所以，并不是所有的Cortex-M内核微处理器都具有相同的中断优先级级别。

       Cortex-M构架自身最多允许256级可编程优先级（优先级配置寄存器最多8位，所以优先级范围从0x00~0xFF），但是绝大多数微控制器制造商只是使用其中的一部分优先级。比如，TI Stellaris Cortex-M3和Cortex-M4微控制器使用优先级配置寄存器的3个位，能提供8级优先级。再比如，NXP LPC17xx Cortex-M3微控制器使用优先级配置寄存器的5个位，能提供32级优先级。

#### ****1.2应用到RTOS****

       RTOS中断嵌套方案将有效的中断优先级分成两组：一组可以通过RTOS临界区屏蔽，另一组不受RTOS影响，永远都是使能的。宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY在FreeRTOSConfig.h中配置，定义两组中断优先级的边界。逻辑优先级高于此值的中断不受RTOS影响。最优值取决于微控制器使用的优先级配置寄存器的位数。

### ****2.与数值相反的优先级值和逻辑优先级设置****

#### ****2.1Cortex-M 硬件详述****

       有必要先解释一下优先级值和逻辑优先级：在Cortex-M内核中，假如有8级优先级，我们说优先级值是0~7，但数值最大的优先级7却代表着最低的逻辑优先级。很多使用传统传统中断优先级[**架构**](http://lib.csdn.net/base/architecture)的工程师会觉得这样比较绕，违反直觉。以下内容提到的优先级要仔细区分是优先级数值还是逻辑优先级。

       接下来需要清楚的是，在Cortex-M内核中，一个中断的优先级数值越低，逻辑优先级却越高。比如，中断优先级为2的中断可以抢占中断优先级为5的中断，但反过来就不行。换句话说，中断优先级2比中断优先级5的优先级更高。

       这是Cortex-M内核最容易让人犯错之处，因为大多数的非Cortex-M内核微控制器的中断优先级表述是与之相反的。

#### ****2.2应用到 RTOS****

       以“FromISR”结尾的FreeRTOS函数是具有中断调用保护的（执行这些函数会进入临界区），但是就算是这些函数，也不可以被逻辑优先级高于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的中断服务函数调用。（宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY定义在头文件FreeRTOSConfig.h中）。因此，任何使用RTOSAPI函数的中断服务例程的中断优先级数值**大于等于**configMAX\_SYSCALL\_INTERRUPT\_PRIORITY宏的值。这样就能保证中断的逻辑优先级等于或低于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY。

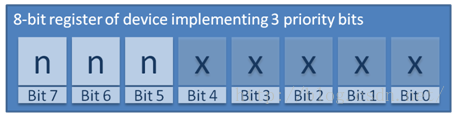
       Cortex中断默认情况下有一个数值为0的优先级。大多数情况下0代表最高级优先级。因此，**绝对不可以在优先级为0的中断服务例程中调用RTOSAPI函数。**

### ****3.Cortex-M 内部优先级概述****

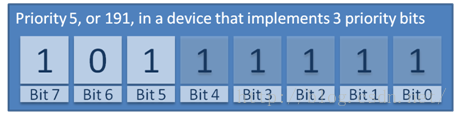
#### ****3.1Cortex-M 硬件详述****

       Cortex-M内核的中断优先级寄存器是以最高位（MSB）对齐的。比如，如果使用了3位来表达优先级，则这3个位位于中断优先级寄存器的bit5、bit6、bit7位。剩余的bit0~bit4可以设置成任何值，但为了兼容，最好将他们设置成1.

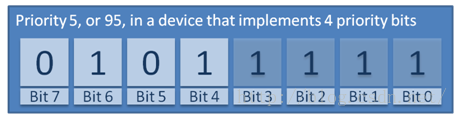
       Cortex-M优先级寄存器最多有8位，如果一个微控制器只使用了其中的3位，那么这3位是以最高位对齐的，见下图：



      某微控制器只使用了优先级寄存器中的3位，下图展示了优先级数值5（二进制101B）是怎样在优先级寄存器中存储的。如果优先级寄存器中未使用的位置1，下图也展示了为什么数值5（二进制0000 0101B）可以看成数值191（二进制1011 1111）的。



      某微控制器只使用了优先级寄存器中的4位，下图展示了优先级数值5（二进制101B）是怎样在优先级寄存器中存储的。如果优先级寄存器中未使用的位置1，下图也展示了为什么数值5（二进制0000 0101B）可以看成数值95（二进制0101 1111）的。



#### ****3.2应用到 RTOS****

      上文中已经描述，那些在中断服务例程中调用RTOS API函数的中断逻辑优先级必须低于或等于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY（低逻辑优先级意味着高优先级数值）。

      CMSIS以及不同的微控制器供应商提供了可以设置某个中断优先级的库函数。一些库函数的参数使用最低位对齐，另一些库函数的参数可能使用最高位对齐，所以，使用时应该查阅库函数的应用手册进行正确设置。

      可以在FreeRTOSConfig.h中设置宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY和configKERNEL\_INTERRUPT\_PRIORITY的值。这两个宏需要根据Cortex-M内核自身的情况进行设置，要**以最高有效位对齐**。比如某微控制器使用中断优先级寄存器中的3位，设置configKERNEL\_INTERRUPT\_PRIORITY的值为5，则代码为：

1. #define     configKERNEL\_INTERRUPT\_PRIORITY  （5<<(8-3)）

       宏configKERNEL\_INTERRUPT\_PRIORITY指定RTOS内核使用的中断优先级，因为RTOS内核不可以抢占用户任务，因此这个宏一般设置为硬件支持的最小优先级。对于Cortex-M硬件，RTOS使用到硬件的PendSV和SysTick硬件中断，在函数xPortStartScheduler()中（该函数在port.c中，由启动调度器函数vTaskStartScheduler()调用），将PendSV和SysTick硬件中断优先级寄存器设置为宏configKERNEL\_INTERRUPT\_PRIORITY指定的值。

       有关代码如下（位于port.c）：

1. /\*PendSV优先级设置寄存器地址为0xe000ed22
2. SysTick优先级设置寄存器地址为0xe000ed23\*/
3. #define portNVIC\_SYSPRI2\_REG     ( \* ( ( volatile uint32\_t \* ) 0xe000ed20 ))
5. #define portNVIC\_PENDSV\_PRI ( ( (uint32\_t)configKERNEL\_INTERRUPT\_PRIORITY ) << 16UL )
6. #define portNVIC\_SYSTICK\_PRI ( ( (uint32\_t)configKERNEL\_INTERRUPT\_PRIORITY ) << 24UL )
7. /\* …. \*/
8. /\*确保PendSV 和SysTick为最低优先级中断 \*/
9. portNVIC\_SYSPRI2\_REG |= portNVIC\_PENDSV\_PRI;
10. portNVIC\_SYSPRI2\_REG |=portNVIC\_SYSTICK\_PRI;

### 4.****临界区****

#### ****4.1Cortex-M 硬件详述****

      RTOS内核使用Cortex-M内核的BASEPRI寄存器来实现临界区（注：BASEPRI为优先级屏蔽寄存器，优先级数值大于或等于该寄存器的中断都会被屏蔽，优先级数值越大，逻辑优先级越低，但是**为零时不屏蔽任何中断**）。这允许RTOS内核可以只屏蔽一部分中断，因此可以提供一个灵活的中断嵌套模式。

      那些需要在中断调用时保护的API函数，FreeRTOS使用寄存器BASEPRI实现中断保护临界区。当进入临界区时，将寄存器BASEPRI的值设置成configMAX\_SYSCALL\_INTERRUPT\_PRIORITY，当退出临界区时，将寄存器BASEPRI的值设置成0。很多Bug反馈都提到，当退出临界区时不应该将寄存器设置成0，应该恢复它之前的状态（之前的状态不一定是0）。但是Cortex-M NVIC决不会允许一个低优先级中断抢占当前正在执行的高优先级中断，不管BASEPRI寄存器中是什么值。与进入临界区前先保存BASEPRI的值，退出临界区再恢复的方法相比，退出临界区时将BASEPRI寄存器设置成0的方法可以获得更快的执行速度。

#### ****4.2应用到RTOS kernel****

      RTOS内核通过写configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的值到BASEPRI寄存器的方法创建临界区。中断优先级0（具有最高的逻辑优先级）不能被BASEPRI寄存器屏蔽，因此，configMAX\_SYSCALL\_INTERRUPT\_PRIORITY绝不可以设置成0。

# 8 [FreeRTOS内存管理](http://blog.csdn.net/zhzht19861011/article/details/50248421)

本文介绍内存管理的基础知识，详细源码分析见《 [**FreeRTOS高级篇7---FreeRTOS内存管理分析**](http://blog.csdn.net/zhzht19861011/article/details/51606068)》

FreeRTOS提供了几个内存堆管理方案，有复杂的也有简单的。其中最简单的管理策略也能满足很多应用的要求，比如对安全要求高的应用，这些应用根本不允许动态内存分配的。

FreeRTOS也允许你自己实现内存堆管理，甚至允许你同时使用两种内存堆管理方案。同时实现两种内存堆允许任务堆栈和其它RTOS对象放置到快速的内部RAM，应用数据放置到低速的外部RAM。

每当创建任务、队列、互斥量、软件定时器、信号量或事件组时，RTOS内核会为它们分配RAM。标准函数库中的malloc()和free()函数有些时候能够用于完成这个任务，但是：

1. 在嵌入式系统中，它们并不总是可以使用的；
2. 它们会占用更多宝贵的代码空间；
3. 它们没有线程保护；
4. 它们不具有确定性（每次调用执行的时间可能会不同）；

      因此，提供一个替代的内存分配方案通常是必要的。

[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)/实时系统具有千差万别的RAM和时间要求，因此一个RAM内存分配[**算法**](http://lib.csdn.net/base/datastructure)可能仅属于一个应用的子集。

      为了避免这个问题，FreeRTOS在移植层保留内存分配API函数。移植层在RTOS核心代码源文件之外（不属于核心源代码），这使得不同的应用程序可以提供适合自己的应用实现。当RTOS内核需要RAM时，调用pvPortMallo()函数来代替malloc()函数。当RAM要被释放时，调用vPortFree()函数来代替free()函数。

      FreeRTOS下载包中提供5种简单的内存分配实现，本文稍后会进行描述。用户可以适当的选择其中的一个，也可以自己设计内存分配策略。

      FreeRTOS提供的内存分配方案分别位于不同的源文件（heap\_1.c、heap\_2.c、heap\_3.c、heap\_4.c、heap\_5.c）之中，源文件位于下载包\FreeRTOS\Source\portable\MemMang文件夹中。其它实现方法可以根据需要增加。如果要使用FreeRTOS提供的内存堆分配方案，选中的源文件必须被正确的包含到工程文件中。

**1.heap\_1.c**

      这是所有实现中最简单的一个。一旦分配内存之后，它甚至不允许释放分配的内存。尽管这样，heap\_1.c还是适用于大部分嵌入式应用程序。这是因为大多数深度嵌入式（deeplyembedded）应用只是在系统启动时创建所有任务、队列、信号量等，并且直到程序结束都会一直使用它们，永远不需要删除。

当需要分配RAM时，这个内存分配方案只是简单的将一个大数组细分出一个子集来。大数组的容量大小通过FreeRTOSConfig.h文件中的configTOTAL\_HEAP\_SIZE宏来设置。

API函数xPortGetFreeHeapSize()返回未分配的堆栈空间总大小，可以通过这个函数返回值对configTOTAL\_HEAP\_SIZE进行合理的设置。

      heap\_1功能简介：

* 用于从不会删除任务、队列、信号量、互斥量等的应用程序（实际上大多数使用FreeRTOS的应用程序都符合这个条件）
* 执行时间是确定的并且不会产生内存碎片
* 实现和分配过程非常简单，需要的内存是从一个静态数组中分配的，意味着这种内存分配通常只是适用于那些不进行动态内存分配的应用。

**2.heap\_2.c**

      和方案1不同，这个方案使用一个最佳匹配算法，它允许释放之前分配的内存块。它不会把相邻的空闲块合成一个更大的块（换句话说，这会造成内存碎片）。

有效的堆栈空间大小由位于FreeRTOSConfig.h文件中的configTOTAL\_HEAP\_SIZE宏来定义。

      API函数xPortGetFreeHeapSize()返回剩下的未分配堆栈空间的大小（可用于优化设置configTOTAL\_HEAP\_SIZE宏的值），但是不能提供未分配内存的碎片细节信息。

      heap\_2功能简介：

* 可以用于重复的分配和删除具有相同堆栈空间的任务、队列、信号量、互斥量等等，并且不考虑内存碎片的应用程序。
* 不能用在分配和释放随机字节堆栈空间的应用程序
  + 如果一个应用程序动态的创建和删除任务，并且分配给任务的堆栈空间总是同样大小，那么大多数情况下heap\_2.c是可以使用的。但是，如果分配给任务的堆栈不总是相等，那么释放的有效内存可能碎片化，形成很多小的内存块。最后会因为没有足够大的连续堆栈空间而造成内存分配失败。在这种情况下，heap\_4.c是一个很好的选择。
  + 如果一个应用程序动态的创建和删除队列，并且在每种情况下队列存储区域（队列存储区域指队列项数目乘以每个队列长度）都是同样的，那么大多数情况下heap\_2.c可以使用。但是，如果队列存储区在每种情况下并不总是相等，那么释放的有效内存可能碎片化，形成很多小的内存块。最后会因为没有足够大的连续堆栈空间而造成内存分配失败。在这种情况下，heap\_4.c是一个很好的选择。
  + 应用程序直接调用pvPortMalloc() 和 vPortFree()函数，而不仅是通过FreeRTOS API间接调用。
* 如果你的应用程序中的队列、任务、信号量、互斥量等等处在一个不可预料的顺序，则可能会导致内存碎片问题，虽然这是小概率事件，但必须牢记。
* 不具有确定性，但是它比标准库中的malloc函数具有高得多的效率。

      heap\_2.c适用于需要动态创建任务的大多数小型实时系统（smallreal time）。

**3.heap\_3.c**

      heap\_3.c简单的包装了标准库中的malloc()和free()函数，包装后的malloc()和free()函数具备线程保护。

      heap\_3.c功能简介：

* 需要链接器设置一个堆栈，并且编译器库提供malloc()和free()函数。
* 不具有确定性
* 可能明显的增大RTOS内核的代码大小

      注：使用heap\_3时，FreeRTOSConfig.h文件中的configTOTAL\_HEAP\_SIZE宏定义没有作用。

**4.heap\_4.c**

      这个方案使用一个最佳匹配算法，但不像方案2那样。它会将相邻的空闲内存块合并成一个更大的块（包含一个合并算法）。

      有效的堆栈空间大小由位于FreeRTOSConfig.h文件中的configTOTAL\_HEAP\_SIZE来定义。

      API函数xPortGetFreeHeapSize()返回剩下的未分配堆栈空间的大小（可用于优化设置configTOTAL\_HEAP\_SIZE宏的值），但是不能提供未分配内存的碎片细节信息。

      heap\_4.c功能简介：

* 可用于重复分配、删除任务、队列、信号量、互斥量等等的应用程序。
* 可以用于分配和释放随机字节内存的情况，并不像heap\_2.c那样产生严重碎片。
* 不具有确定性，但是它比标准库中的malloc函数具有高得多的效率。

      heap\_4.c还特别适用于移植层代码，可以直接使用pvPortMalloc()和 vPortFree()函数来分配和释放内存。

**5.heap\_5.c（V8.1.0新增）**

      这个方案同样实现了heap\_4.c中的合并算法，并且允许堆栈跨越多个非连续的内存区。

      Heap\_5通过调用vPortDefineHeapRegions()函数实现初始化，在该函数执行完成前不允许使用内存分配和释放。创建RTOS对象（任务、队列、信号量等等）会隐含的调用pvPortMalloc()，因此必须注意：使用heap\_5创建任何对象前，要先执行vPortDefineHeapRegions()函数。

      vPortDefineHeapRegions()函数只需要单个参数。该参数是一个HeapRegion\_t结构体类型数组。HeapRegion\_t在portable.h中定义，如下所示：

1. typedef struct HeapRegion
2. {
3. /\* 用于内存堆的内存块起始地址\*/
4. uint8\_t \*pucStartAddress;
6. /\* 内存块大小 \*/
7. size\_t xSizeInBytes;
8. } HeapRegion\_t;

      这个数组必须使用一个NULL指针和0字节元素作为结束，起始地址必须从小到大排列。下面的代码段提供一个例子。MSVCWin32模拟器演示例程使用了heap\_5，因此可以当做一个参考例程。

1. /\* 在内存中为内存堆分配两个内存块.第一个内存块0x10000字节,起始地址为0x80000000,
2. 第二个内存块0xa0000字节,起始地址为0x90000000.起始地址为0x80000000的内存块的
3. 起始地址更低,因此放到了数组的第一个位置.\*/
4. const HeapRegion\_t xHeapRegions[] =
5. {
6. { ( uint8\_t \* ) 0x80000000UL, 0x10000 },
7. { ( uint8\_t \* ) 0x90000000UL, 0xa0000 },
8. { NULL, 0 } /\* 数组结尾. \*/
9. };
11. /\* 向函数vPortDefineHeapRegions()传递数组参数. \*/
12. vPortDefineHeapRegions( xHeapRegions );

# 9 [FreeRTOS任务概述](http://blog.csdn.net/zhzht19861011/article/details/50312443)

### 1. 任务和协程（Co-routines）

      应用程序可以使用任务也可以使用协程，或者两者混合使用，但是任务和协程使用不同的API函数，因此在任务和协程之间不能使用同一个队列或信号量传递数据。

      通常情况下，协程仅用在资源非常少的微处理器中，特别是RAM非常稀缺的情况下。目前协程很少被使用到，因此对于协程FreeRTOS作者既没有把它删除也没有进一步开发。

      所以本系列文章以后不会对协程过多描述，包括其API函数。

#### 1.1任务的特性

      简而言之：使用RTOS的实时应用程序可认为是一系列独立任务的集合。每个任务在自己的环境中运行，不依赖于系统中的其它任务或者RTOS调度器。在任何时刻，只有一个任务得到运行，RTOS调度器决定运行哪个任务。调度器会不断的启动、停止每一个任务，宏观看上去就像整个应用程序都在执行。作为任务，不需要对调度器的活动有所了解，在任务切入切出时保存上下文环境（寄存器值、堆栈内容）是调度器主要的职责。为了实现这点，每个任务都需要有自己的堆栈。当任务切出时，它的执行环境会被保存在该任务的堆栈中，这样当再次运行时，就能从堆栈中正确的恢复上次的运行环境。

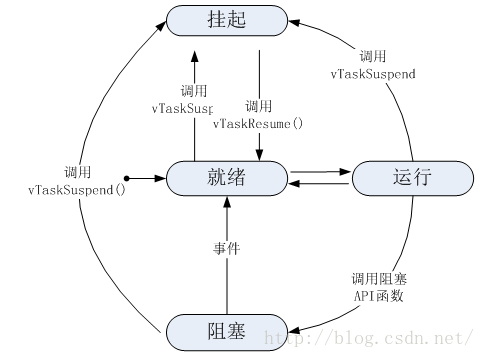
#### 1.2任务概要

* 简单
* 没有使用限制
* 支持完全抢占
* 支持优先级
* 每个任务都有自己的堆栈，消耗RAM较多
* 如果使用抢占，必须小心的考虑可重入问题

### 2. 任务状态

      一个任务可为下面中的一个：

1. 运行：如果一个任务正在执行，那么说这个任务处于运行状态。此时它占用处理器。
2. 就绪：就绪的任务已经具备执行的能力（不同于阻塞和挂起），但是因为有一个同优先级或者更高优先级的任务处于运行状态而还没有真正执行。
3. 阻塞：如果任务当前正在等待某个时序或外部中断，我们就说这个任务处于阻塞状态。比如一个任务调用vTaskDelay()后会阻塞到延时周期到为止。任务也可能阻塞在队列或信号量事件上。进入阻塞状态的任务通常有一个“超时”周期，当事件超时后解除阻塞。
4. 挂起：处于挂起状态的任务同样对调度器无效。仅当明确的分别调用vTaskSuspend() 和xTaskResume() API函数后，任务才会进入或退出挂起状态。不可以指定超时周期事件（不可以通过设定超时事件而退出挂起状态）



### 3.任务优先级

      每个任务都要被指定一个优先级，从0~configMAX\_PRIORITIES，configMAX\_PRIORITIES定义在FreeRTOSConfig.h中。

      如果某[**架构**](http://lib.csdn.net/base/architecture)硬件支持CLZ（或类似）指令（计算前导零的数目，Cortex-M3是支持该指令的，从ARMv6T2才支持这个指令），并且打算在移植层使用这个特性来优化任务调度机制，需要有一些步骤，首先将FreeRTOSConfig.h中configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION设置为1，并且最大优先级数目configMAX\_PRIORITIES不能大于32。除此之外，configMAX\_PRIORITIES可以设置为任意值，但是考虑到configMAX\_PRIORITIES设置越大，RAM消耗也越大，一般设置为满足使用的最小值。

      低优先级数值代表低优先级。空闲任务（idle task）的优先级为0（tskIDLE\_PRIORITY）。

      FreeRTOS调度器确保处于最高优先级的就绪或运行态任务获取处理器，换句话说，处于运行状态的任务，只有其中的最高优先级任务才会运行。

      任何数量的任务可以共享同一个优先级。如果宏configUSE\_TIME\_SLICING未定义或着宏configUSE\_TIME\_SLICING定义为1，处于就绪态的多个相同优先级任务将会以时间片切换的方式共享处理器。

### 4.实现一个任务

      一个任务具有以下结构：

1. void vATaskFunction( voidvoid \*pvParameters )
2. {
3. for( ;; )
4. {
5. /\*-- 应用程序代码放在这里. --\*/
6. }
8. /\* 任务不可以从这个函数返回或退出。在较新的FreeRTOS移植包中，如果
9. 试图从一个任务中返回，将会调用configASSERT()（如果定义的话）。
10. 如果一个任务确实要退出函数，那么这个任务应调用vTaskDelete(NULL)
11. 函数，以便处理一些清理工作。\*/
12. vTaskDelete( NULL );
13. }

**任务函数返回为void，参数只有一个void类型指针**。所有的任务函数都应该是这样。void类型指针可以向任务传递任意类型信息。

      任务函数决不应该返回，因此通常任务函数都是一个死循环。

      任务由xTaskCreate()函数创建，由vTaskDelete()函数删除。

### 5.空闲任务和空闲任务钩子（idle task和Idle Task hook）

#### 5.1空闲任务

      空闲任务是启动RTOS调度器时由内核自动创建的任务，这样可以确保至少有一个任务在运行。空闲任务具有最低任务优先级，这样如果有其它更高优先级的任务进入就绪态就可以立刻让出CPU。

      删除任务后，空闲任务用来释放RTOS分配给被删除任务的内存。因此，在应用中使用vTaskDelete()函数后确保空闲任务能获得处理器时间就很重要了。除此之外，空闲任务没有其它有效功能，所以可以被合理的剥夺处理器时间，并且它的优先级也是最低的。

      应用程序任务共享空闲任务优先级（tskIDLE\_PRIORITY）也是可能的。这种情况如何配置可以参考configIDLE\_SHOULE\_YIELD配置参数类获取更多信息。

#### 5.2空闲任务钩子

      空闲任务钩子是一个函数，每一个空闲任务周期被调用一次。如果你想将任务程序功能运行在空闲优先级上，可以有两种选择：

1. 在一个空闲任务钩子中实现这个功能：因为FreeRTOS必须至少有一个任务处于就绪或运行状态，因此钩子函数不可以调用可能引起空闲任务阻塞的API函数（比如vTaskDelay()或者带有超时事件的队列或信号量函数）。
2. 创建一个具有空闲优先级的任务去实现这个功能：这是个更灵活的解决方案，但是会带来更多RAM开销。

      创建一个空闲钩子步骤如下：

* 在FreeRTOSConfig.h头文件中设置configUSE\_IDLE\_HOOK为1；
* 定义一个函数，名字和参数原型如下所示：

1. void vApplicationIdleHook( void );

         通常，**使用这个空闲钩子函数设置CPU进入低功耗模式**。

# 10 [FreeRTOS任务创建和删除](http://blog.csdn.net/zhzht19861011/article/details/50371956)

在FreeRTOS移植到Cortex-M3硬件平台的文章中，我们已经见过任务创建API，但那篇文章的重点在于如何移植FreeRTOS，本文将重点放在任务的创建和删除API函数上面。

      任务创建和删除API函数位于文件task.c中，需要包含task.h头文件。

### 1.任务创建

#### 1.1函数描述

1. BaseType\_t xTaskCreate(
2. TaskFunction\_t pvTaskCode,
3. const charchar \* const pcName,
4. unsigned short usStackDepth,
5. voidvoid \*pvParameters,
6. UBaseType\_t uxPriority,
7. TaskHandle\_t \* pvCreatedTask
8. );

      创建新的任务并加入任务就绪列表。

      如果使用FreeRTOS-MPU（在官方下载包中，为Cortex-M3内核写了两个移植方案，一个是普通的FreeRTOS移植层，还有一个是FreeRTOS-MPU移植层。后者包含完整的内存保护），那么推荐使用函数xTaskCreateRestricted()来代替xTaskCreate()。在使用FreeRTOS-MPU的情况下，使用xTaskCreate()函数可以创建运行在特权模式或用户模式（见下面对函数参数uxPriority的描述）的任务。当运行在特权模式下，任务可以访问整个内存映射；当处于用户模式下，任务仅能访问自己的堆栈。无论在何种模式下，MPU都不会自动捕获堆栈溢出，因此标准的FreeRTOS堆栈溢出检测机制仍然会被用到。xTaskCreateRestricted()函数具有更大的灵活性。

#### 1.2参数描述

* pvTaskCode：指针，指向任务函数的入口。任务永远不会返回（位于死循环内）。该参数类型TaskFunction\_t定义在文件projdefs.h中，定义为：typedefvoid (\*TaskFunction\_t)( void \* )。
* pcName：任务描述。主要用于调试。字符串的最大长度由宏configMAX\_TASK\_NAME\_LEN指定，该宏位于FreeRTOSConfig.h文件中。
* usStackDepth：指定任务堆栈大小，能够支持的堆栈变量数量，而**不是字节数**。比如，在16位宽度的堆栈下，usStackDepth定义为100，则实际使用200字节堆栈存储空间。堆栈的宽度乘以深度必须不超过size\_t类型所能表示的最大值。比如，size\_t为16位，则可以表示的最大值是65535。
* pvParameters：指针，当任务创建时，作为一个参数传递给任务。
* uxPriority：任务的优先级。具有MPU支持的系统，可以通过置位优先级参数的portPRIVILEGE\_BIT位，随意的在特权（系统）模式下创建任务。比如，创建一个优先级为2的特权任务，参数uxPriority可以设置为( 2 | portPRIVILEGE\_BIT )。
* pvCreatedTask：用于回传一个句柄（ID），创建任务后可以使用这个句柄引用任务。

#### 1.3返回值

      如果任务成功创建并加入就绪列表函数返回pdPASS，否则函数返回错误码，具体参见projdefs.h。

#### 1.4用法举例

1. /\* 创建任务. \*/
2. void vTaskCode( voidvoid \* pvParameters )
3. {
4. for( ;; )
5. {
6. /\* 任务代码放在这里 \*/
7. }
8. }
10. /\* 创建任务函数 \*/
11. void vOtherFunction( void )
12. {
13. static unsigned char ucParameterToPass;
14. xTaskHandlexHandle;
16. /\* 创建任务，存储句柄。注：传递的参数ucParameterToPass必须和任务具有相同的生存周期，
17. 因此这里定义为静态变量。如果它只是一个自动变量，可能不会有太长的生存周期，因为
18. 中断和高优先级任务可能会用到它。 \*/
19. xTaskCreate( vTaskCode, "NAME", STACK\_SIZE,&ucParameterToPass, tskIDLE\_PRIORITY, &xHandle );
21. /\* 使用句柄删除任务. \*/
22. if( xHandle !=NULL )
23. {
24. vTaskDelete( xHandle );
25. }
26. }

### 2.任务删除

#### 2.1 任务描述

      voidvTaskDelete( TaskHandle\_t xTask );

      从RTOS内核管理器中删除一个任务。任务删除后将会从就绪、阻塞、暂停和事件列表中移除。在文件FreeRTOSConfig.h中，必须定义宏INCLUDE\_vTaskDelete 为1，本函数才有效。

      注：被删除的任务，其在任务创建时由内核分配的存储空间，会由空闲任务释放。如果有应用程序调用xTaskDelete()，必须保证空闲任务获取一定的微控制器处理时间。任务代码自己分配的内存是不会自动释放的，因此删除任务前，应该将这些内存释放。

#### 2.2参数描述

* xTask：被删除任务的句柄。为NULL表示删除当前任务。

# 11 [FreeRTOS任务控制](http://blog.csdn.net/zhzht19861011/article/details/50454591)

FreeRTOS任务控制API函数主要实现任务延时、任务挂起、解除任务挂起、任务优先级获取和设置等功能。

### 1.相对延时

#### 1.1函数描述

      void vTaskDelay( portTickTypexTicksToDelay )

      调用vTaskDelay()函数后，任务会进入阻塞状态，持续时间由vTaskDelay()函数的参数xTicksToDelay指定，单位是系统节拍时钟周期。常量portTICK\_RATE\_MS 用来辅助计算真实时间，此值是系统节拍时钟中断的周期，单位是毫秒。在文件FreeRTOSConfig.h中，宏INCLUDE\_vTaskDelay 必须设置成1，此函数才能有效。

      vTaskDelay()指定的延时时间是从调用vTaskDelay()后开始计算的相对时间。比如vTaskDelay(100)，那么从调用vTaskDelay()后，任务进入阻塞状态，经过100个系统时钟节拍周期，任务解除阻塞。因此，vTaskDelay()并不适用与周期性执行任务的场合。此外，其它任务和中断活动，会影响到vTaskDelay()的调用（比如调用前高优先级任务抢占了当前任务），因此会影响任务下一次执行的时间。API函数vTaskDelayUntil()可用于固定频率的延时，它用来延时一个绝对时间。

#### 1.2参数描述

* xTicksToDelay：延时时间总数，单位是系统时钟节拍周期。

#### 1.3用法举例

1. voidvTaskFunction( voidvoid \* pvParameters )
2. {
3. /\* 阻塞500ms. \*/
4. constportTickType xDelay = 500 / portTICK\_RATE\_MS;
6. for( ;; )
7. {
8. /\* 每隔500ms触发一次LED， 触发后进入阻塞状态 \*/
9. vToggleLED();
10. vTaskDelay( xDelay );
11. }
12. }

### 2.绝对延时

#### 2.1函数描述

         void vTaskDelayUntil( TickType\_t \*pxPreviousWakeTime,

                      const TickType\_txTimeIncrement );

      任务延时一个指定的时间。周期性任务可以使用此函数，以确保一个恒定的频率执行。在文件FreeRTOSConfig.h中，宏INCLUDE\_vTaskDelayUntil 必须设置成1，此函数才有效。

      这个函数不同于vTaskDelay()函数的一个重要之处在于：vTaskDelay()指定的延时时间是从调用vTaskDelay()之后（执行完该函数）开始算起的，但是vTaskDelayUntil()指定的延时时间是一个绝对时间。

      调用vTaskDelay()函数后，任务会进入阻塞状态，持续时间由vTaskDelay()函数的参数指定，单位是系统节拍时钟周期。因此vTaskDelay()并不适用于周期性执行任务的场合。因为调用vTaskDelay()到任务解除阻塞的时间不总是固定的并且该任务下一次调用vTaskDelay()函数的时间也不总是固定的（两次执行同一任务的时间间隔本身就不固定，中断或高优先级任务抢占也可能会改变每一次执行时间）。

      vTaskDelay()指定一个从调用vTaskDelay()函数后开始计时，到任务解除阻塞为止的相对时间，而vTaskDelayUntil()指定一个绝对时间，每当时间到达，则解除任务阻塞。

      应当指出的是，如果指定的唤醒时间已经达到，vTaskDelayUntil()立刻返回（不会有阻塞）。因此，使用vTaskDelayUntil()周期性执行的任务，无论任何原因（比如，任务临时进入挂起状态）停止了周期性执行，使得任务少运行了一个或多个执行周期，那么需要重新计算所需要的唤醒时间。这可以通过传递给函数的指针参数pxPreviousWake指向的值与当前系统时钟计数值比较来检测，在大多数情况下，这并不是必须的。

      常量portTICK\_RATE\_MS 用来辅助计算真实时间，此值是系统节拍时钟中断的周期，单位是毫秒。

      当调用vTaskSuspendAll()函数挂起RTOS调度器时，不可以使用此函数。

#### 2.2参数描述

* pxPreviousWakeTime：指针，指向一个变量，该变量保存任务最后一次解除阻塞的时间。第一次使用前，该变量必须初始化为当前时间。之后这个变量会在vTaskDelayUntil()函数内自动更新。
* xTimeIncrement：周期循环时间。当时间等于(\*pxPreviousWakeTime + xTimeIncrement)时，任务解除阻塞。如果不改变参数xTimeIncrement的值，调用该函数的任务会按照固定频率执行。

#### 2.3用法举例

1. //每10次系统节拍执行一次
2. void vTaskFunction( voidvoid \* pvParameters )
3. {
4. static portTickType xLastWakeTime;
5. const portTickType xFrequency = 10;
7. // 使用当前时间初始化变量xLastWakeTime
8. xLastWakeTime = xTaskGetTickCount();
10. for( ;; )
11. {
12. //等待下一个周期
13. vTaskDelayUntil( &xLastWakeTime,xFrequency );
15. // 需要周期性执行代码放在这里
16. }
17. }

### 3.获取任务优先级

#### 3.1函数描述

      UBaseType\_t uxTaskPriorityGet(TaskHandle\_t xTask );

      获取指定任务的优先级。在文件FreeRTOSConfig.h中，宏INCLUDE\_vTaskPriorityGet必须设置成1，此函数才有效。

#### 3.2参数描述

* xTask：任务句柄。NULL表示获取当前任务的优先级。

#### 3.3返回值

      返回指定任务的优先级。

#### 3.4用法举例

1. voidvAFunction( void )
2. {
3. xTaskHandlexHandle;
4. // 创建任务，保存任务句柄
5. xTaskCreate( vTaskCode, "NAME",STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );
6. // ...
7. // 使用句柄获取创建的任务的优先级
8. if( uxTaskPriorityGet( xHandle ) !=tskIDLE\_PRIORITY )
9. {
10. // 任务可以改变自己的优先级
11. }
12. // ...
13. // 当前任务优先级比创建的任务优先级高？
14. if( uxTaskPriorityGet( xHandle ) <uxTaskPriorityGet( NULL ) )
15. {
16. // 当前优先级较高
17. }
18. }

### 4.设置任务优先级

#### 4.1函数描述

      void vTaskPrioritySet( TaskHandle\_txTask,

                       UBaseType\_tuxNewPriority );

      设置指定任务的优先级。如果设置的优先级高于当前运行的任务，在函数返回前会进行一次上下文切换。在文件FreeRTOSConfig.h中，宏INCLUDE\_vTaskPrioritySet 必须设置成1，此函数才有效。

#### 4.2参数描述

* xTask：要设置优先级任务的句柄，为NULL表示设置当前运行的任务。
* uxNewPriority：要设置的新优先级。

#### 4.3用法举例

1. voidvAFunction( void )
2. {
3. xTaskHandlexHandle;
4. // 创建任务，保存任务句柄。
5. xTaskCreate( vTaskCode, "NAME",STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );
6. // ...
7. // 使用句柄来提高创建任务的优先级
8. vTaskPrioritySet( xHandle,tskIDLE\_PRIORITY + 1 );
9. // ...
10. // 使用NULL参数来提高当前任务的优先级，设置成和创建的任务相同。
11. vTaskPrioritySet( NULL, tskIDLE\_PRIORITY +1 );
12. }

### 5.任务挂起

#### 5.1函数描述

      void vTaskSuspend( TaskHandle\_txTaskToSuspend );

      挂起指定任务。被挂起的任务绝不会得到处理器时间，不管该任务具有什么优先级。

      调用vTaskSuspend函数是不会累计的：即使多次调用vTaskSuspend ()函数将一个任务挂起，也只需调用一次vTaskResume ()函数就能使挂起的任务解除挂起状态。在文件FreeRTOSConfig.h中，宏INCLUDE\_vTaskSuspend必须设置成1，此函数才有效。

#### 5.2参数描述

* xTaskToSuspend：要挂起的任务句柄。为NULL表示挂起当前任务。

#### 5.3用法举例

1. voidvAFunction( void )
2. {
3. xTaskHandlexHandle;
4. // 创建任务，保存任务句柄.
5. xTaskCreate( vTaskCode, "NAME",STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );
6. // ...
7. // 使用句柄挂起创建的任务.
8. vTaskSuspend( xHandle );
9. // ...
10. // 任务不再运行，除非其它任务调用了vTaskResume(xHandle )
11. //...
12. // 挂起本任务.
13. vTaskSuspend( NULL );
14. // 除非另一个任务使用handle调用了vTaskResume，否则永远不会执行到这里
15. }

### 6.恢复挂起的任务

#### 6.1函数描述

      void vTaskResume( TaskHandle\_txTaskToResume );

      恢复挂起的任务。

      通过调用一次或多次vTaskSuspend()挂起的任务，可以调用一次vTaskResume ()函数来再次恢复运行。在文件FreeRTOSConfig.h中，宏INCLUDE\_vTaskSuspend必须置1，此函数才有效。

#### 6.2参数描述

* xTaskToResume：要恢复运行的任务句柄。

#### 6.3用法举例

1. voidvAFunction( void )
2. {
3. xTaskHandle xHandle;
4. // 创建任务，保存任务句柄
5. xTaskCreate( vTaskCode, "NAME",STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );
6. // ...
7. // 使用句柄挂起创建的任务
8. vTaskSuspend( xHandle );
9. // ...
10. //任务不再运行，除非其它任务调用了vTaskResume(xHandle )
11. //...
12. // 恢复挂起的任务.
13. vTaskResume( xHandle );
14. // 任务再一次得到处理器时间
15. // 任务优先级与之前相同
16. }

### 7.恢复挂起的任务（在中断服务函数中使用）

#### 7.1函数描述

      BaseType\_t xTaskResumeFromISR(TaskHandle\_t xTaskToResume );

      用于恢复一个挂起的任务，用在ISR中。

      通过调用一次或多次vTaskSuspend()函数而挂起的任务，只需调用一次xTaskResumeFromISR()函数即可恢复运行。

      xTaskResumeFromISR()不可用于任务和中断间的同步，如果中断恰巧在任务被挂起之前到达，这就会导致一次中断丢失（任务还没有挂起，调用xTaskResumeFromISR()函数是没有意义的，只能等下一次中断）。这种情况下，可以使用信号量作为同步机制。在文件FreeRTOSConfig.h中，宏INCLUDE\_vTaskSuspend 和 INCLUDE\_xTaskResumeFromISR 必须设置成1，此函数才有效。

#### 7.2参数描述

* xTaskToResume：要恢复运行的任务句柄。

#### 7.3返回值

      如果恢复任务后需要上下文切换返回pdTRUE，否则返回pdFALSE。由ISR确定是否需要上下文切换。

#### 7.4用法举例

1. xTaskHandlexHandle;               //注意这是一个全局变量
3. void vAFunction( void )
4. {
5. // 创建任务并保存任务句柄
6. xTaskCreate( vTaskCode, "NAME",STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );
8. // ... 剩余代码.
9. }
11. void vTaskCode( voidvoid \*pvParameters )
12. {
13. for( ;; )
14. {
15. // ... 在这里执行一些其它功能
17. // 挂起自己
18. vTaskSuspend( NULL );
20. //直到ISR恢复它之前，任务会一直挂起
21. }
22. }
24. void vAnExampleISR( void )
25. {
26. portBASE\_TYPExYieldRequired;
28. // 恢复被挂起的任务
29. xYieldRequired = xTaskResumeFromISR(xHandle );
31. if( xYieldRequired == pdTRUE )
32. {
33. // 我们应该进行一次上下文切换
34. // 注:  如何做取决于你具体使用，可查看说明文档和例程
35. portYIELD\_FROM\_ISR();
36. }
37. }

# 12 [FreeRTOS任务应用函数](http://blog.csdn.net/zhzht19861011/article/details/50498173)

任务应用函数是一组辅助类函数，一般用于调试信息输出、获取任务句柄、获取任务状态、操作任务标签值等等。

### 1.获取任务系统状态

#### 1.1函数描述

1. UBaseType\_t uxTaskGetSystemState(
2. TaskStatus\_t \* constpxTaskStatusArray,
3. const UBaseType\_tuxArraySize,
4. unsigned longlong \* constpulTotalRunTime );

      该函数向TaskStatus\_t结构体填充相关信息，系统中每一个任务的信息都可以填充到TaskStatus\_t结构体数组中，数组大小由uxArraySize指定。结构体TaskStatus\_t定义如下：

1. typedef struct xTASK\_STATUS
2. {
3. /\* 任务句柄\*/
4. TaskHandle\_t xHandle;
6. /\* 指针，指向任务名\*/
7. const signed charchar \*pcTaskName;
9. /\*任务ID，是一个独一无二的数字\*/
10. UBaseType\_t xTaskNumber;
12. /\*填充结构体时，任务当前的状态（运行、就绪、挂起等等）\*/
13. eTaskState eCurrentState;
15. /\*填充结构体时，任务运行（或继承）的优先级。\*/
16. UBaseType\_t uxCurrentPriority;
18. /\* 当任务因继承而改变优先级时，该变量保存任务最初的优先级。仅当configUSE\_MUTEXES定义为1有效。\*/
19. UBaseType\_t uxBasePriority;
21. /\* 分配给任务的总运行时间。仅当宏configGENERATE\_RUN\_TIME\_STATS为1时有效。\*/
22. unsigned long ulRunTimeCounter;
24. /\* 从任务创建起，堆栈剩余的最小数量，这个值越接近0，堆栈溢出的可能越大。 \*/
25. unsigned short usStackHighWaterMark;
26. }TaskStatus\_t;

      注意，这个函数仅用来调试用，调用此函数会挂起所有任务，直到函数最后才恢复挂起的任务，因此任务可能被挂起很长时间。在文件FreeRTOSConfig.h中，宏configUSE\_TRACE\_FACILITY必须设置为1，此函数才有效。

#### 1.2参数描述

* pxTaskStatusArray：指向TaskStatus\_t类型的结构体数组。这个数组至少要包含1个元素。RTOS控制的任务数量可以使用API函数uxTaskGetNumberOfTasks()获取。
* uxArraySize：参数pxTaskStatusArray指向的数组大小，也就是该数组的索引数目。
* pulTotalRunTime：如果在文件FreeRTOSConfig.h中设置宏configGENERATE\_RUN\_TIME\_STATS为1，则该函数将总运行时间写入\*pulTotalRunTime中。pulTotalRunTime可以设置为NULL，表示忽略总运行时间。

#### 1.3返回值

         被填充的TaskStatus\_t结构体数量。这个值应该等于通过调用API函数uxTaskGetNumberOfTasks()返回的值，但如果传递给uxArraySize参数的值太小，则返回0。

#### 1.4用法举例

1. /\*本例演示如是使用uxTaskGetSystemState()函数来获取运行时间信息，并将其转化为程序员更易识别的字符格式，这些转化后的字符保存到pcWriteBuffer中。\*/
2. void vTaskGetRunTimeStats(signed charchar \*pcWriteBuffer )
3. {
4. TaskStatus\_t\*pxTaskStatusArray;
5. volatileUBaseType\_t uxArraySize, x;
6. unsignedlong ulTotalRunTime, ulStatsAsPercentage;
8. /\* 防御性代码，确保字符串有合理的结束\*/
9. \*pcWriteBuffer = 0x00;
11. /\* 获取任务总数目\*/
12. uxArraySize = uxTaskGetNumberOfTasks ();
14. /\*为每个任务的TaskStatus\_t结构体分配内存，也可以静态的分配一个足够大的数组 \*/
15. pxTaskStatusArray = pvPortMalloc( uxArraySize \* sizeof( TaskStatus\_t ));
17. if(pxTaskStatusArray != NULL )
18. {
19. /\*获取每个任务的状态信息 \*/
20. uxArraySize = uxTaskGetSystemState( pxTaskStatusArray, uxArraySize,&ulTotalRunTime );
22. /\* 百分比计算 \*/
23. ulTotalRunTime /= 100UL;
25. /\* 避免除零错误 \*/
26. if(ulTotalRunTime > 0 )
27. {
28. /\* 将获得的每一个任务状态信息部分的转化为程序员容易识别的字符串格式\*/
29. for( x = 0; x < uxArraySize; x++ )
30. {
31. /\* 计算任务运行时间与总运行时间的百分比。\*/
32. ulStatsAsPercentage = pxTaskStatusArray[ x ].ulRunTimeCounter /ulTotalRunTime;
34. if( ulStatsAsPercentage > 0UL )
35. {
36. sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%\r\n",
37. pxTaskStatusArray[ x ].pcTaskName,
38. pxTaskStatusArray[ x ].ulRunTimeCounter,
39. ulStatsAsPercentage );
40. }
41. else
42. {
43. /\* 任务运行时间不足总运行时间的1%\*/
44. sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r\n",
45. pxTaskStatusArray[ x ].pcTaskName,
46. pxTaskStatusArray[x ].ulRunTimeCounter );
47. }
49. pcWriteBuffer += strlen( ( charchar \* ) pcWriteBuffer );
50. }
51. }
53. /\* 释放之前申请的内存\*/
54. vPortFree( pxTaskStatusArray );
55. }
56. }

### 2.获取当前任务句柄

#### 2.1函数描述

      TaskHandle\_t xTaskGetCurrentTaskHandle(void );

      在文件FreeRTOSConfig.h中，宏INCLUDE\_xTaskGetCurrentTaskHandle必须设置为1，此函数才有效。

#### 2.2返回值

      返回当前任务（调用该函数的任务）的句柄。

### 3.获取空闲任务句柄

#### 3.1函数描述

      TaskHandle\_t xTaskGetIdleTaskHandle(void );

      在文件FreeRTOSConfig.h中，宏INCLUDE\_xTaskGetIdleTaskHandle必须设置为1，此函数才有效。

#### 3.2返回值

      返回空闲任务句柄。空闲任务在RTOS调度器启动时自动创建。

### 4.获取任务堆栈最大使用深度

#### 4.1函数描述

      UBaseType\_t uxTaskGetStackHighWaterMark( TaskHandle\_t xTask );

      任务的堆栈空间会随着任务执行以及中断处理而增长或缩小。该函数可以返回任务启动后的最小剩余堆栈空间。换句话说，可以间接估算出一个任务最多需要多少堆栈空间。在文件FreeRTOSConfig.h中，宏INCLUDE\_uxTaskGetStackHighWaterMark 必须设置成1，此函数才有效。

#### 4.2参数描述

* xTask：任务句柄。NULL表示查看当前任务的堆栈使用情况。

#### 4.3返回值

      返回最小剩余堆栈空间，以字为单位。比如一个32为[**架构**](http://lib.csdn.net/base/architecture)处理器，返回值为1表示有4字节堆栈空间没有使用过。如果返回值为0，则任务很可能已经发生了堆栈溢出。

#### 4.4用法举例

1. void vTask1( voidvoid \* pvParameters )
2. {
3. UBaseType\_tuxHighWaterMark;
5. /\* 入口处检测一次 \*/
6. uxHighWaterMark =uxTaskGetStackHighWaterMark( NULL );
8. for( ;; )
9. {
10. /\* 正常调用函数 \*/
11. vTaskDelay( 1000 );
13. /\* 测量堆栈使用情况 \*/
14. uxHighWaterMark =uxTaskGetStackHighWaterMark( NULL );
15. }
16. }

### 5.获取任务状态

#### 5.1函数描述

      eTaskState eTaskGetState( TaskHandle\_txTask );

      返回一个枚举类型的任务状态值。在文件FreeRTOSConfig.h中，宏INCLUDE\_eTaskGetState必须设置为1，此函数才有效。

#### 5.2参数描述

* xTask：任务句柄

#### 5.3返回值

         下表列出返回值和对应的任务状态。



### 6.获取任务描述内容

#### 6.1函数描述

      char \* pcTaskGetTaskName( TaskHandle\_txTaskToQuery );

      获取任务的描述内容，在文件FreeRTOSConfig.h中，宏INCLUDE\_pcTaskGetTaskName必须设置成1，此函数才有效。

#### 6.2参数描述

* xTaskToQuery：任务的句柄。NULL表示获取当前任务的描述内容指针。

#### 6.3返回值

      一个指针，指向任务描述字符串。

### 7.获取系统节拍次数

#### 7.1函数描述

      volatile TickType\_t xTaskGetTickCount(void );

      这个函数不能在ISR中调用。在ISR中用xTaskGetTickCountFromISR()，原型为volatileTickType\_t xTaskGetTickCountFromISR( void )。

#### 7.2返回值

      返回从vTaskStartScheduler函数调用后的系统时钟节拍次数。

### 8.获取调度器状态

#### 8.1函数描述

      BaseType\_t xTaskGetSchedulerState( void);

      获取调度器当前状态。在文件FreeRTOSConfig.h中，宏INCLUDE\_xTaskGetSchedulerState或configUSE\_TIMERS必须定义为1，此函数才有效。

#### 8.2返回值

      返回值是以下常量之一（定义在task.h）：taskSCHEDULER\_NOT\_STARTED（未启动）、taskSCHEDULER\_RUNNING（正常运行）、taskSCHEDULER\_SUSPENDED（挂起）。

### 9.获取任务总数

#### 9.1函数描述

      UBaseType\_t uxTaskGetNumberOfTasks(void );

      获取RTOS内核当前管理的任务总数。包含所有就绪、阻塞和挂起状态的任务。对于一个删除的任务，如果它的堆栈空间还没有被空闲任务释放掉，则这个被删除的任务也含在计数值中。

#### 9.2返回值

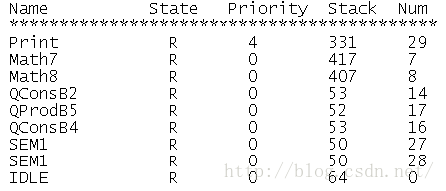
      返回RTOS内核当前管理的任务总数。

### 10.获取所有任务详情

#### 10.1函数描述

      void vTaskList( char \*pcWriteBuffer );

      将每个任务的状态、堆栈使用情况等以字符的形式保存到参数pcWriteBuffer指向的区域。vTaskList()函数调用usTaskGetSystemState()函数，然后将得到的信息格式化为程序员易读的字符形式。输出的内容例子如下图所示，图中State一栏中，B表示阻塞、R表示就绪、D表示删除（等待清除内存）、S表示挂起或阻塞。



      注意，调用这个函数会挂起所有任务，这一过程可能持续较长时间，因此本函数仅在调试时使用。在文件FreeRTOSConfig.h中，宏configUSE\_TRACE\_FACILITY和configUSE\_STATS\_FORMATTING\_FUNCTIONS必须定义为1，此函数才有效。

#### 10.2参数描述

* pcWriteBuffer：任务的信息会写入这个缓冲区，为ASCII表单形式。这个缓冲区要足够大，以容纳生成的报告，每个任务大约需要40个字节。

### 11.获取任务运行时间

#### 11.1函数描述

      void vTaskGetRunTimeStats( char\*pcWriteBuffer );

      这个函数用于统计每个任务的运行时间。要使用这个函数必须满足一些条件，那就是必须有一个用于时间统计的定时器或计数器，这个定时器或计数器的精度要至少大于10倍的系统节拍周期。这个定时器或计数器的配置以及获取定时时间是由两个宏定义实现的，这两个宏一般在文件FreeRTOSConfig.h中定义。配置定时器或计数器的宏为portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS()，获取定时时间的宏为portGET\_RUN\_TIME\_COUNTER\_VALUE。实现了这两个宏定义后，还必须在文件FreeRTOSConfig.h中将宏configGENERATE\_RUN\_TIME\_STATS和configUSE\_STATS\_FORMATTING\_FUNCTIONS设置为1，此API函数才有效。

      这个API函数调用usTaskGetSystemState()函数获取每个任务的状态信息，并把其中的运行时间格式化为程序员易读的字符形式，并将这些信息保存到参数pcWriteBuffer指向的区域。

      注意，调用这个函数会挂起所有任务，这一过程可能持续较长时间，因此本函数仅在调试时使用。

#### 11.2参数描述

* pcWriteBuffer：任务的运行时间信息会写入这个缓冲区，为ASCII表单形式。这个缓冲区要足够大，以容纳生成的报告，每个任务大约需要40个字节。

#### 11.3用法举例

      以lpc17xx系列为控制为例，我们使用定时器0来作为统计基准时钟。

##### 11.3.1使能函数宏

      在文件FreeRTOSConfig.h中，设置宏configGENERATE\_RUN\_TIME\_STATS和configUSE\_STATS\_FORMATTING\_FUNCTIONS为1，

##### 11.3.2定时初始化定时器代码

1. void vConfigureTimerForRunTimeStats( void )
2. {
3. /\* 使能定时器0的外设电源，配置外设时钟 \*/
4. PCONP |= 0x02UL;
5. PCLKSEL0 = (PCLKSEL0& (~(0x3<<2))) | (0x01 << 2);
7. /\* 复位定时器 0 \*/
8. T0TCR = 0x02;
10. /\* 作为计数器 \*/
11. T0CTCR = 0x00;
13. /\* 预分频，设置合适的分辨率即可 \*/
14. T0PR =  ( configCPU\_CLOCK\_HZ / 10000UL ) - 1UL;
16. /\* 启动计数器 \*/
17. T0TCR = 0x01;
18. }

##### 11.3.3定义配置定时器和获取定时时间宏

      在文件FreeRTOSConfig.h中，定义下列代码：

1. extern void vConfigureTimerForRunTimeStats( void );
2. #define portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() vConfigureTimerForRunTimeStats()
3. #defineportGET\_RUN\_TIME\_COUNTER\_VALUE() T0TC

### 12.设置任务标签值

#### 12.1函数描述

1. voidvTaskSetApplicationTaskTag( TaskHandle\_t xTask,
2. TaskHookFunction\_tpxTagValue );

      可以给每个任务分配一个标签值。这个值一般用于应用程序，RTOS内核不会使用。在文件FreeRTOSConfig.h中，宏configUSE\_APPLICATION\_TASK\_TAG必须设置为1，此函数才有效。

#### 12.2参数描述

* xTask：任务句柄。NULL表示当前任务。
* pxTagValue：要分配给任务的标签值。这是一个TaskHookFunction\_t类型的函数指针，但也可以给任务标签分配任意的值。

      注：TaskHookFunction\_t原型定义：typedef BaseType\_t (\*TaskHookFunction\_t)(void \* )

#### 12.3用法举例

1. /\* 在这个例子中，给任务设置一个整形标签值。例子中使用了RTOS跟踪钩子宏。\*/
2. void vATask( voidvoid \*pvParameters )
3. {
4. /\* 为自己的标签分配一个整形值 \*/
5. vTaskSetApplicationTaskTag( NULL, ( voidvoid \* ) 1 );
7. for( ;; )
8. {
9. /\* 任务主体代码 \*/
10. }
11. }
12. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
14. /\*在这个任务中，给任务设置一个函数标签值。首先定义一个回调函数，这个函数必须声明为TaskHookFunction\_t类型。 \*/
15. static BaseType\_t prvExampleTaskHook( voidvoid \* pvParameter )
16. {
17. /\* 这里为用户定义代码 –可能是记录数据、更新任务状态值等。\*/
19. return 0;
20. }
22. /\* 将回调函数设置为任务的标签值。 \*/
23. void vAnotherTask( voidvoid \*pvParameters )
24. {
25. /\* 注册回调函数\*/
26. vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );
28. for( ;; )
29. {
30. /\* 任务主体代码 \*/
31. }
32. }
34. /\* 每当任务切换时，会调用xTaskCallApplicationTaskHook 函数（见14.）。 \*/
35. #define traceTASK\_SWITCHED\_OUT() xTaskCallApplicationTaskHook(pxCurrentTCB,0 )

### 13.获取任务标签值

#### 13.1函数描述

      TaskHookFunction\_txTaskGetApplicationTaskTag( TaskHandle\_t xTask );

      返回分配给任务的标签值。程序员定义标签值，RTOS内核通常不会访问标签值。

      函数仅对高级用户使用。在文件FreeRTOSConfig.h中，宏configUSE\_APPLICATION\_TASK\_TAG必须设置为1，此函数才有效。

#### 13.2参数描述

* xTask：任务句柄。NULL表示当前任务。

#### 13.3返回值

      返回指定任务的标签值。

### 14.执行任务的应用钩子函数

#### 14.1函数描述

1. BaseType\_txTaskCallApplicationTaskHook(
2. TaskHandle\_txTask,
3. void\*pvParameter );

      可以为每个任务分配一个标签值，当这个值是一个TaskHookFunction\_t类型函数指针时，相当于应用程序向任务注册了一个回调函数，而API函数xTaskCallApplicationTaskHook用来调用这个回调函数。

      一般这个函数配合RTOS跟踪钩子宏使用，见12.设置任务标签值一节的用法举例。

#### 14.2参数描述

* xTask：任务句柄。NULL表示当前任务。
* pvParameter：作为参数传递给应用钩子函数

### 15.设置线程本地存储指针

#### 15.1函数描述

1. void vTaskSetThreadLocalStoragePointer(TaskHandle\_t xTaskToSet,
2. BaseType\_t xIndex,
3. void\*pvValue )

      此函数仅用于高级用户。

      线程本地存储允许应用程序在任务的控制块中存储一些值，每个任务都有自己独立的储存空间。

      比如，许多库函数都包含一个叫做errno的全局变量。某些库函数使用errno返回库函数错误信息，应用程序检查这个全局变量来确定发生了那些错误。在单线程程序中，将errno定义成全局变量是可以的，但是在多线程应用中，每个线程（任务）必须具有自己独有的errno值，否则，一个任务可能会读取到另一个任务的errno值。

      FreeRTOS提供了一个灵活的机制，使得应用程序可以使用线程本地存储指针来读写线程本地存储。在文件FreeRTOSConfig.h中，宏configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS指定每个任务线程本地存储指针数组的大小。API函数vTaskSetThreadLocalStoragePointer()用于向指针数组中写入值，API函数pvTaskGetThreadLocalStoragePointer()用于从指针数组中读取值。

#### 15.2参数描述

* xTaskToSet：任务句柄。NULL表示当前任务。
* xIndex：写入到线程本地存储数组的索引号，线程本笃存储数组的大小由宏configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS设定，该宏在文件FreeRTOSConfig.h中。
* pvValue：写入到指定索引地址的数据值

#### 15.3用法举例

     参见16.获取线程本地存储指针一节。

### 16.读取线程本地存储指针

#### 16.1函数描述

1. void\*pvTaskGetThreadLocalStoragePointer(
2. TaskHandle\_txTaskToQuery,
3. BaseType\_txIndex );

      此函数仅用于高级用户。从线程本地存储指针数组中读取值。更详细描述见15.设置线程本地存储指针一节。

#### 16.2参数描写

* xTaskToQuery：任务句柄。NULL表示当前任务。
* xIndex：写入到线程本地存储数组的索引号，线程本笃存储数组的大小由宏configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS设定，该宏在文件FreeRTOSConfig.h中。

#### 16.3返回值

      返回一个指针，这个指针存储在线程本地存储指针数组中，数组索引由参数xIndex指定。

#### 16.4用法举例

##### 16.4.1存储一个整形数

1. uint32\_tulVariable;
3. /\* 向当前任务的线程本地存储数组下标为1的位置写入一个指向32位常量值的指针。\*/
4. vTaskSetThreadLocalStoragePointer(NULL, 1, ( voidvoid \* ) 0x12345678 );
6. /\*向当前任务的线程本地存储数组下标为0的位置写入一个指向32整形值的指针\*/
7. ulVariable= ERROR\_CODE;
8. vTaskSetThreadLocalStoragePointer(NULL, 0, ( voidvoid \* ) ulVariable );
10. /\*从当前任务的线程本地存储数组下标为5的位置读取出一个指针并赋值给32位整形变量。\*/
11. ulVariable= ( uint32\_t ) pvTaskGetThreadLocalStoragePointer( NULL, 5 );

##### 16.4.2存储结构提

1. typedefstruct
2. {
3. uint32\_t ulValue1;
4. uint32\_t ulValue2;
5. }xExampleStruct;
7. xExampleStruct\*pxStruct;
9. /\*为结构体分配内存\*/
10. pxStruct= pvPortMalloc( sizeof( xExampleStruct ) );
12. /\*为结构体成员赋值\*/
13. pxStruct->ulValue1= 0;
14. pxStruct->ulValue2= 1;
16. /\*向当前任务的线程本地存储数组下标为0的位置写入一个指向结构体变量的指针\*/
17. vTaskSetThreadLocalStoragePointer(NULL, 0, ( voidvoid \* ) pxStruct );
19. /\*从当前任务的线程本地存储数组下标为0的位置读取出一个结构体指针\*/
20. pxStruct= ( xExampleStruct \* ) pvTaskGetThreadLocalStoragePointer( NULL, 0 );

### 17.设置超时状态

#### 17.1函数描述

      void vTaskSetTimeOutState( TimeOut\_t \*const pxTimeOut );

      此函数仅用于高级用户，通常与API函数xTaskCheckForTimeOut()共同使用。

      任务因为等待某事件而进入阻塞状态，通常情况下任务会设置一个等待超时周期。如果在等待事件超时，任务会退出阻塞状态。想象一个这样的应用，某任务等待一个事件而进入阻塞状态，但是事件迟迟不发生，超时后任务退出阻塞状态继续执行任务。假如任务等待的事件仍然没有发生，则任务又会阻塞在该事件下。只要任务等待的事件一直不发生，这个任务进入阻塞然后超时退出阻塞，再进入阻塞的循环就会一直存在。是不是可以设定一个总超时时间，只要总阻塞时间大于这个总超时时间，则可以结束这个任务或进行相应记录？freeRTOS提供了两个API函数来完成这个功能，这就是vTaskSetTimeOutState()和xTaskCheckForTimeOut()。

      vTaskSetTimeOutState()函数用于设置初始条件，之后调用xTaskCheckForTimeOut()函数检查任务总阻塞时间是否超过总超时时间，如果没有超过，则调整剩余的超时时间计数器。

#### 17.2参数描述

* pxTimeOut：指向一个结构体的指针，该结构体用来保存确定超时是否发生的必要信息。vTaskSetTimeOutState()函数会设置结构体的成员。

#### 17.3用法举例

      参见18.超时检测。

### 18.超时检测

#### 18.1函数描述

1. BaseType\_t xTaskCheckForTimeOut(TimeOut\_t \* const pxTimeOut,
2. TickType\_t\* const pxTicksToWait );

      此函数仅用于高级用户，通常与API函数vTaskSetTimeOutState共同使用。

      详细描述见17.设置超时状态。

#### 18.2参数描述

* pxTimeOut：指向一个结构体的指针。该结构体保存确定超时是否发生的必要信息。使用API函数vTaskSetTimeOutState初始化该结构体。
* pxTicksToWait：TickType\_t指针，指向的变量保存总超时时间。

#### 18.3返回值

* pdTRUE：总超时发生
* pdFALSE：总超时未发生

#### 18.4用法举例

1. /\* 函数用于从RX缓冲区中接收uxWantedBytes字节数据，RX缓冲区由UART中断填充。如果RX缓冲区没有足够的数据，则任务进入阻塞状态，直到RX缓冲区有足够数据或者发生超时。如果超时后仍然没有足够的数据，则任务会再次进入阻塞状态，xTaskCheckForTimeOut()函数用于重新计算总超时时间以确保总阻塞状态时间不超过MAX\_TIME\_TO\_WAIT。如果总阻塞状态时间大于了总超时时间，则不管RX缓冲区是否有充足数据，都将这些数据读出来。
2. \*/
3. size\_txUART\_Receive( uint8\_t \*pucBuffer, size\_t uxWantedBytes )
4. {
5. size\_t uxReceived = 0;
6. TickType\_t xTicksToWait = MAX\_TIME\_TO\_WAIT;
7. TimeOut\_t xTimeOut;
9. /\* 初始化结构体变量xTimeOut。\*/
10. vTaskSetTimeOutState( &xTimeOut );
12. /\* 无限循环，直到缓冲区包含足够的数据或者阻塞超时发生。\*/
13. while( UART\_bytes\_in\_rx\_buffer(pxUARTInstance ) < uxWantedBytes )
14. {
15. /\* RX缓冲区没有足够多的数据，表示任务已经进入过一次阻塞状态。调用API函数xTaskCheckForTimeOut检查总阻塞时间是否超过总超时时间，如果没有，则调整剩余的总超时时间。\*/
16. if( xTaskCheckForTimeOut( &xTimeOut,&xTicksToWait ) != pdFALSE )
17. {
18. /\* 如果总阻塞时间大于总超时时间，则退出这个循环 \*/
19. break;
20. }
22. /\* 在等待了xTicksToWait个系统节拍周期后，向接收中断发出通知，需要更多数据。
23. \*/
24. ulTaskNotifyTake( pdTRUE, xTicksToWait );
25. }
27. /\*从RX缓冲区读取uxWantedBytes个字节并放到pucBuffer缓冲区。\*/
28. uxReceived = UART\_read\_from\_receive\_buffer(pxUARTInstance,  pucBuffer,  uxWantedBytes );
30. return uxReceived;
31. }

# 13 [FreeRTOS内核控制](http://blog.csdn.net/zhzht19861011/article/details/50553099)

内核控制的一些功能需要移植层提供，为了方便移植，这些API函数用宏来实现，比如上下文切换、进入和退出临界区、禁止和使能可屏蔽中断。内核控制函数还包括启动和停止调度器、挂起和恢复调度器以及用于低功耗模式的调整系统节拍函数。

### 1.强制上下文切换宏

taskYIELD：用于强制上下文切换的宏。在中断服务程序中的等价版本为portYIELD\_FROM\_ISR，这也是个宏，其实现取决于移植层。

      用于上下文切换的实际代码由移植层提供。对于Cortex-M3硬件，这个宏会引起PendSV中断。

### 2.进入临界区宏

taskENTER\_CRITICAL：用于进入临界区的宏。在临界区中不会发生上下文切换。

     进入临界区的实际代码由移植层提供，对于Cortex-M3硬件，先禁止所有RTOS可屏蔽中断，这可以通过向basepri 寄存器写入configMAX\_SYSCALL\_INTERRUPT\_PRIORITY来实现。basepri寄存器被设置成某个值后，所有优先级号大于等于此值的中断都被禁止，但若被设置为0，则不关闭任何中断，0为默认值。然后临界区嵌套计数器增1。

### 3.退出临界区宏

taskEXIT\_CRITICAL：用于退出临界区的宏。

      退出临界区的实际代码有移植层提供，对于Cortex-M3硬件，先将临界区嵌套计数器减1，如果临界区计数器为零，则使能所有RTOS可屏蔽中断，这可以通过向basepri 寄存器写入0来实现。

### 4.禁止可屏蔽中断宏

taskDISABLE\_INTERRUPTS：禁止所有RTOS可屏蔽中断。在调用宏taskENTER\_CRITICAL进入临界区时，也会间接调用该宏禁止所有RTOS可屏蔽中断。

### 5.使能可屏蔽中断宏

taskENABLE\_INTERRUPTS：使能所有RTOS可屏蔽中断。在调用宏taskEXIT\_CRITICAL退出临界区时，也会间接调用该宏使能所有RTOS可屏蔽中断。

### 6.启动调度器

#### 6.1函数描述

      void vTaskStartScheduler( void );

      启动RTOS调度器，之后RTOS内核控制哪个任务执行以及何时执行。

      当调用vTaskStartScheduler()后，空闲任务被自动创建。如果configUSE\_TIMERS被设置为1，定时器后台任务也会被创建。

      如果vTaskStartScheduler()成功执行，则该函数不会返回，直到有任务调用了vTaskEndScheduler()。如果因为RAM不足而无法创建空闲任务，该函数也可能执行失败，并会立刻返回调用处。

### 7.停止调度器

#### 7.1函数描述

      void vTaskEndScheduler( void );

      仅用于x86硬件[**架构**](http://lib.csdn.net/base/architecture)中。

      停止RTOS内核系统节拍时钟。所有创建的任务自动删除并停止多任务调度。

### 8.挂起调度器

#### 8.1函数描述

      void vTaskSuspendAll( void );

      挂起调度器，但不禁止中断。当调度器挂起时，不会进行上下文切换。调度器挂起后，正在执行的任务会一直继续执行，内核不再调度（意味着当前任务不会被切换出去），直到该任务调用了xTaskResumeAll ()函数。

      内核调度器挂起期间，那些可以引起上下文切换的API函数（如vTaskDelayUntil()、xQueueSend()等）决不可使用。

### 9.恢复被挂起的调度器

#### 9.1函数描述

      BaseType\_t xTaskResumeAll( void );

      恢复因调用vTaskSuspendAll()函数而挂起的实时内核调度器。xTaskResumeAll()仅恢复调度器，它不会恢复那些被vTaskSuspend()函数挂起的任务。

#### 9.2返回值

      返回pdTRUE 表示恢复调度器引起了一次上下文切换，否则，返回pdFALSE。

#### 9.3用法举例

1. voidvTask1( voidvoid \* pvParameters )
2. {
3. for( ;; )
4. {
5. /\* 任务代码写在这里 \*/
7. /\* ... \*/
9. /\* 有些时候，某个任务希望可以连续长时间的运行，但这时不能使用taskENTER\_CRITICAL ()/taskEXIT\_CRITICAL ()的方法，这样会屏蔽掉中断，引起中断丢失，包括系统节拍时钟。可以使用vTaskSuspendAll ()停止RTOS内核调度：\*/
10. xTaskSuspendAll ();
12. /\* 执行操作代码放在这里。这样不用进入临界区就可以连续长时间执行了。在这期间，中断仍然会得到响应，RTOS内核系统节拍时钟也会继续保持运作 \*/
14. /\* ... \*/
16. /\* 操作结束，重新启动RTOS内核 。我们想强制进行一次上下文切换，但是如果恢复调度器的时候已经执行了上下文切换，再执行一次是没有意义的，因此会进行一次判断。\*/
17. if( !xTaskResumeAll () )
18. {
19. taskYIELD ();
20. }
21. }
22. }

### 10.调整系统节拍

#### 10.1函数描述

      void vTaskStepTick( TickType\_txTicksToJump );

      如果RTOS使能tickless空闲功能，每当只有空闲任务被执行时，系统节拍时钟中断将会停止，微控制器进入低功耗模式。当微控制器退出低功耗后，系统节拍计数器必须被调整，将进入低功耗的时间弥补上。

      如果FreeRTOS移植文件中定义了宏portSUPPRESS\_TICKS\_AND\_SLEEP()实体，则函数vTaskStepTick用于在这个宏portSUPPRESS\_TICKS\_AND\_SLEEP()实体内部调整系统节拍计数器。函数vTaskStepTick是一个全局函数，所以也可以在宏portSUPPRESS\_TICKS\_AND\_SLEEP()实体中重写该函数。

      在文件FreeRTOSConfig.h中，宏configUSE\_TICKLESS\_IDLE必须设置为1，此函数才有效。

#### 10.2参数描述

* xTickToJump：时间值，单位是系统节拍周期，表示微处理器进入低功耗的时间，函数根据这个值来调整系统节拍计数器的值。

#### 10.3用法举例

1. /\* 首先定义宏portSUPPRESS\_TICKS\_AND\_SLEEP()。宏参数指定要进入低功耗（睡眠）的时间，单位是系统节拍周期。\*/
2. #defineportSUPPRESS\_TICKS\_AND\_SLEEP( xIdleTime ) vApplicationSleep( xIdleTime )
4. /\* 定义被宏portSUPPRESS\_TICKS\_AND\_SLEEP()调用的函数 \*/
5. void vApplicationSleep(TickType\_t xExpectedIdleTime )
6. {
7. unsigned long ulLowPowerTimeBeforeSleep,ulLowPowerTimeAfterSleep;
9. /\* 从时钟源获取当前时间，当微控制器进入低功耗的时候，这个时钟源必须在运行 \*/
10. ulLowPowerTimeBeforeSleep =ulGetExternalTime();
12. /\*停止系统节拍时钟中断。\*/
13. prvStopTickInterruptTimer();
15. /\* 配置一个中断，当指定的睡眠时间达到后，将处理器从低功耗中唤醒。这个中断源必须在微控制器进入低功耗时也可以工作。\*/
16. vSetWakeTimeInterrupt( xExpectedIdleTime );
18. /\*进入低功耗 \*/
19. prvSleep();
21. /\* 确定微控制器进入低功耗模式持续的真正时间。因为其它中断也可能使得微处理器退出低功耗模式。注意：在调用宏portSUPPRESS\_TICKS\_AND\_SLEEP()之前，调度器应该被挂起，portSUPPRESS\_TICKS\_AND\_SLEEP()返回后，再将调度器恢复。因此，这个函数未完成前，不会执行其它任务。\*/
22. ulLowPowerTimeAfterSleep =ulGetExternalTime();
24. /\*调整内核系统节拍计数器。\*/
25. vTaskStepTick( ulLowPowerTimeAfterSleep –ulLowPowerTimeBeforeSleep );
27. /\*重新启动系统节拍时钟中断。\*/
28. prvStartTickInterruptTimer();
29. }

# 14 [FreeRTOS任务通知](http://blog.csdn.net/zhzht19861011/article/details/50578159)

注：本文介绍任务通知的基础知识，详细源码分析见《[**FreeRTOS高级篇8---FreeRTOS任务通知分析**](http://blog.csdn.net/zhzht19861011/article/details/51629039)》

      每个RTOS任务都有一个32位的通知值，任务创建时，这个值被初始化为0。RTOS任务通知相当于直接向任务发送一个事件，接收到通知的任务可以解除阻塞状态，前提是这个阻塞事件是因等待通知而引起的。发送通知的同时，也可以可选的改变接收任务的通知值。

      可以通过下列方法向接收任务更新通知：

* 不覆盖接收任务的通知值
* 覆盖接收任务的通知值
* 设置接收任务通知值的某些位
* 增加接收任务的通知值

      相对于用前必须分别创建队列、二进制信号量、计数信号量或事件组的情况，使用任务通知显然更灵活。更好的是，相比于使用信号量解除任务阻塞，使用任务通知可以快45%、使用更少的RAM（使用GCC编译器，-o2优化级别）。

      使用API函数xTaskNotify()和xTaskNotifyGive()（中断保护等价函数为xTaskNotifyFromISR()和vTaskNotifyGiveFromISR()）发送通知，在接收RTOS任务调用API函数xTaskNotifyWait()或ulTaskNotifyTake()之前，这个通知都被保持着。如果接收RTOS任务已经因为等待通知而进入阻塞状态，则接收到通知后任务解除阻塞并清除通知。

      RTOS任务通知功能默认是使能的，可以通过在文件FreeRTOSConfig.h中设置宏configUSE\_TASK\_NOTIFICATIONS为0来禁止这个功能，禁止后每个任务节省8字节内存。

      虽然RTOS任务通知速度更快并且占用内存更少，但它也有一些限制：

1. 只能有一个任务接收通知事件。
2. 接收通知的任务可以因为等待通知而进入阻塞状态，但是发送通知的任务即便不能立即完成通知发送也不能进入阻塞状态。

### 1.发送通知-方法1

#### 1.1函数描述

1. BaseType\_t xTaskNotify( TaskHandle\_txTaskToNotify,
2. uint32\_t ulValue,
3. eNotifyAction eAction);

      向指定任务发送指定的通知值。如果打算使用RTOS任务通知实现轻量级的二进制或计数信号量，推荐使用API函数xTaskNotifyGive()来代替本函数。

      此函数不可以在中断服务例程中调用，中断保护等价函数为xTaskNotifyFromISR()。

#### 1.2参数描述

* xTaskToNotify：被通知的任务句柄。
* ulValue：通知更新值
* eAction：枚举类型，指明更新通知值的方法

      枚举变量成员以及作用如下表所示。



#### 1.3返回值

参数eAction为eSetValueWithoutOverwrite时，如果被通知任务还没取走上一个通知，又接收到了一个通知，则这次通知值未能更新并返回pdFALSE，否则返回pdPASS。

### 2.发送通知-方法2

#### 2.1函数描述

      BaseType\_t xTaskNotifyGive(TaskHandle\_t xTaskToNotify );

      其实这是一个宏，本质上相当于xTaskNotify( ( xTaskToNotify ), ( 0 ), eIncrement )。可以使用该API函数代替二进制或计数信号量，但速度更快。在这种情况下，应该使用API函数ulTaskNotifyTake()来等待通知，而不应该使用API函数xTaskNotifyWait()。

      此函数不可以在中断服务例程中调用，中断保护等价函数为vTaskNotifyGiveFromISR()。

#### 2.2参数描述

* xTaskToNotify：被通知的任务句柄。

#### 2.3用法举例

1. staticvoid prvTask1( voidvoid \*pvParameters );
2. staticvoid prvTask2( voidvoid \*pvParameters );
4. /\* 保存任务句柄 \*/
5. staticTaskHandle\_t xTask1 = NULL, xTask2 = NULL;
7. /\* 创建两个任务，它们之间反复发送通知，启动RTOS调度器\*/
8. voidmain( void )
9. {
10. xTaskCreate( prvTask1, "Task1",200, NULL, tskIDLE\_PRIORITY, &xTask1 );
11. xTaskCreate( prvTask2, "Task2",200, NULL, tskIDLE\_PRIORITY, &xTask2 );
12. vTaskStartScheduler();
13. }
14. /\*-----------------------------------------------------------\*/
16. staticvoid prvTask1( voidvoid \*pvParameters )
17. {
18. for( ;; )
19. {
20. /\*向prvTask2(),发送通知，使其解除阻塞状态 \*/
21. xTaskNotifyGive( xTask2 );
23. /\* 等待prvTask2()的通知，进入阻塞 \*/
24. ulTaskNotifyTake( pdTRUE, portMAX\_DELAY);
25. }
26. }
27. /\*-----------------------------------------------------------\*/
29. staticvoid prvTask2( voidvoid \*pvParameters )
30. {
31. for( ;; )
32. {
33. /\* 等待prvTask1()的通知，进入阻塞 \*/
34. ulTaskNotifyTake( pdTRUE, portMAX\_DELAY);
36. /\*向prvTask1(),发送通知，使其解除阻塞状态 \*/
37. xTaskNotifyGive( xTask1 );
38. }
39. }

### 3.获取通知

#### 3.1函数描述

1. uint32\_t ulTaskNotifyTake( BaseType\_txClearCountOnExit,
2. TickType\_txTicksToWait );

      ulTaskNotifyTake()是专门为使用更轻量级更快的方法来代替二进制或计数信号量而量身打造的。FreeRTOS获取信号量的API函数为xSemaphoreTake()，可以使用ulTaskNotifyTake()函数等价代替。

      当一个任务使用通知值来实现二进制或计数信号量时，其它任务或者中断要使用API函数xTaskNotifyGive()或者使用参数eAction为eIncrement的API函数xTaskNotify()。推荐使用xTaskNotifyGive()函数（其实是个宏，我们这里把它看作一个API函数）。另外需要注意的是，如果在中断中使用，要使用它们的中断保护等价函数：vTaskNotifyGiveFromISR()和xTaskNotifyFromISR()。

      API函数xTaskNotifyTake()有两种方法处理任务的通知值，一种方法是在函数退出时将通知值清零，这种方法适用于实现二进制信号量；另外一种方法是在函数退出时将通知值减1，这种方法适用于实现计数信号量。

      如果RTOS任务的通知值为0，使用xTaskNotifyTake()可以可选的使任务进入阻塞状态，直到该任务的通知值不为0。进入阻塞的任务不消耗CPU时间。

#### 3.2参数描述

* xClearCountOnExit：如果该参数设置为pdFALSE，则API函数xTaskNotifyTake()退出前，将任务的通知值减1；如果该参数设置为pdTRUE，则API函数xTaskNotifyTake()退出前，将任务通知值清零。
* xTicksToWait：因等待通知而进入阻塞状态的最大时间。时间单位为系统节拍周期。宏pdMS\_TO\_TICKS用于将指定的毫秒时间转化为相应的系统节拍数。

#### 3.3返回值

      返回任务的当前通知值，为0或者为调用API函数xTaskNotifyTake()之前的通知值减1。

### 3.4用法举例

1. /\* 中断处理程序。\*/
2. voidvANInterruptHandler( void )
3. {
4. BaseType\_txHigherPriorityTaskWoken;
6. prvClearInterruptSource();
8. /\* xHigherPriorityTaskWoken必须被初始化为pdFALSE。如果调用vTaskNotifyGiveFromISR()会解除vHandlingTask任务的阻塞状态，并且vHandlingTask任务的优先级高于当前处于运行状态的任务，则xHigherPriorityTaskWoken将会自动被设置为pdTRUE。\*/
9. xHigherPriorityTaskWoken = pdFALSE;
11. /\*向一个任务发送通知，xHandlingTask是该任务的句柄。\*/
12. vTaskNotifyGiveFromISR( xHandlingTask,&xHigherPriorityTaskWoken );
14. /\* 如果xHigherPriorityTaskWoken为pdTRUE，则强制上下文切换。这个宏的实现取决于移植层，可能会调用portEND\_SWITCHING\_ISR \*/
15. portYIELD\_FROM\_ISR(xHigherPriorityTaskWoken );
16. }
17. /\*---------------------------------------------------------------------------------------------------\*/
19. /\* 一个因为等待通知而阻塞的任务。\*/
20. voidvHandlingTask( voidvoid \*pvParameters )
21. {
22. BaseType\_txEvent;
24. for( ;; )
25. {
26. /\*等待通知，无限期阻塞。参数pdTRUE表示函数退出前会清零通知值。这显然是用于替代二进制信号量的用法。需要注意的是，真实的程序一般不会无限期阻塞。\*/
27. ulTaskNotifyTake( pdTRUE, portMAX\_DELAY);
29. /\* 当处理完所有事件后，仍然等待下一个通知\*/
30. do
31. {
32. xEvent = xQueryPeripheral();
34. if( xEvent != NO\_MORE\_EVENTS )
35. {
36. vProcessPeripheralEvent( xEvent);
37. }
39. } while( xEvent != NO\_MORE\_EVENTS );
40. }
41. }

### 4.等待通知

#### 4.1函数描述

1. BaseType\_t xTaskNotifyWait( uint32\_tulBitsToClearOnEntry,
2. uint32\_tulBitsToClearOnExit,
3. uint32\_t\*pulNotificationValue,
4. TickType\_txTicksToWait );

      如果打算使用RTOS任务通知实现轻量级的二进制或计数信号量，推荐使用API函数ulTaskNotifyTake()来代替本函数。

#### 4.2参数描述

* ulBitsToClearOnEntry：在使用通知之前，先将任务的通知值与参数ulBitsToClearOnEntry的按位取反值按位与操作。设置参数ulBitsToClearOnEntry为0xFFFFFFFF(ULONG\_MAX)，表示清零任务通知值。
* ulBitsToClearOnExit：在函数xTaskNotifyWait()退出前，将任务的通知值与参数ulBitsToClearOnExit的按位取反值按位与操作。设置参数ulBitsToClearOnExit为0xFFFFFFFF(ULONG\_MAX)，表示清零任务通知值。
* pulNotificationValue：用于向外回传任务的通知值。这个通知值在参数ulBitsToClearOnExit起作用前将通知值拷贝到\*pulNotificationValue中。如果不需要返回任务的通知值，这里设置成NULL。
* xTicksToWait：因等待通知而进入阻塞状态的最大时间。时间单位为系统节拍周期。宏pdMS\_TO\_TICKS用于将指定的毫秒时间转化为相应的系统节拍数。

#### 4.3返回值

      如果接收到通知，返回pdTRUE，如果API函数xTaskNotifyWait()等待超时，返回pdFALSE。

#### 4.4用法举例

1. /\*这个任务使用任务通知值的位来传递不同的事件，这在某些情况下可以代替事件组。\*/
2. voidvAnEventProcessingTask( voidvoid \*pvParameters )
3. {
4. uint32\_tulNotifiedValue;
6. for( ;; )
7. {
8. /\*等待通知，无限期阻塞（没有超时，所以不用检查函数返回值）。其它任务或者中断设置的通知值中的不同位表示不同的事件。参数0x00表示使用通知前不清除任务的通知值位，参数ULONG\_MAX 表示函数xTaskNotifyWait()退出前将任务通知值设置为0\*/
9. xTaskNotifyWait( 0x00, ULONG\_MAX,&ulNotifiedValue, portMAX\_DELAY );
11. /\*根据通知值处理事件\*/
12. if( ( ulNotifiedValue & 0x01 ) != 0)
13. {
14. prvProcessBit0Event();
15. }
17. if( ( ulNotifiedValue & 0x02 ) != 0)
18. {
19. prvProcessBit1Event();
20. }
22. if( ( ulNotifiedValue & 0x04 ) != 0)
23. {
24. prvProcessBit2Event();
25. }
27. /\* ……\*/
28. }
29. }

### 5.任务通知并查询

#### 5.1函数描述

1. BaseType\_t xTaskNotifyAndQuery(TaskHandle\_t xTaskToNotify,
2. uint32\_tulValue,
3. eNotifyActioneAction,
4. uint32\_t\*pulPreviousNotifyValue );

      此函数与任务通知API函数xTaskNotify()非常像，只不过此函数具有一个附加参数，用来回传任务当前的通知值，然后根据参数ulValue和eAction更新任务的通知值。

      此函数不能在中断服务例程中使用，在中断服务例程中使用xTaskNotifyAndQueryFromISR()函数。

#### 5.2参数描述

* xTaskToNotify：被通知的任务句柄。
* ulValue：通知更新值
* eAction：枚举类型，指明更新通知值的方法，枚举变量成员以及作用见xTaskNotify()一节。
* pulPreviousNotifyValue：回传未被更新的任务通知值。如果不需要回传未被更新的任务通知值，这里设置为NULL，这样就等价于调用xTaskNotify()函数。

#### 5.3返回值

      参数eAction为eSetValueWithoutOverwrite时，如果被通知任务还没取走上一个通知，又接收到了一个通知，则这次通知值未能更新并返回pdFALSE，否则返回pdPASS。

# 15 [使用任务通知实现命令行解释器](http://blog.csdn.net/zhzht19861011/article/details/50654242)

虽然这是介绍FreeRTOS系列的文章，但这篇文章偏重于命令行解释器的实现。这一方面是因为任务通知使用起来非常简单，另一方面也因为对于[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)程序来说，使用命令行解释器来辅助程序调试是非常有用的。程序调试是一门技术，基本上我们需要两种调试手段，一种是可以单步仿真的硬件调试器，另外一种是可以长期监视程序状态的状态输出，可以通过串口、显示屏等等手段输出异常信息或者某些关键点。这里的命令行解释器就属于后者。

     本文实现的命令行解释器具有以下特性：

* 支持十进制参数，识别负号；
* 支持十六进制参数，十六进制以‘0x’开始；
* 命令名长度可定义，默认最大20个字符；
* 参数数目可定义，默认最多8个参数；
* 命令名和参数之间以空格隔开，空格个数任意；
* 整条命令以回车换行符结束；
* 整条命令最大长度可定义，默认64字节，包括回车换行符；
* 如果使用SecureCRT串口工具（推荐），支持该软件的控制字符，比如退格键、左移键、右移键等。

      一个带参数的命令格式如下所示：

**参数名** <参数1> <参数2> … <参数3>[回车换行符]

### 1.编码风格

      FreeRTOS的编码标准及风格见《[FreeRTOS系列第4篇---FreeRTOS编码标准及风格指南](http://blog.csdn.net/zhzht19861011/article/details/50057531)》，但我自己的编码风格跟FreeRTOS并不相同，并且我也不打算改变我当前坚持使用的编码风格。所以在这篇或者以后的文章中可能会在一个程序中看到两种不同的编码风格，对于涉及FreeRTOS的代码，我尽可能使用FreeRTOS建议的编码风格，与FreeRTOS无关的代码，我仍然使用自己的编码风格。我可以保证，两种编码风格决不会影响程序的可读性，编写良好可读性的代码，是我一直注重并坚持的。

### 2.一些准备工作

#### 2.1串口硬件驱动

      命令行解释器使用一个硬件串口，需要外部提供两个串口底层函数：一个是串口初始化函数init\_cmd\_uart()，用于初始化串口波特率、中断等事件；另一个是发送单个字符函数my\_putc()。此外，命令行为串口接收中断服务程序提供函数fill\_rec\_buf()，用于保存接收到的字符，当收到回车换行符后，该函数向命令行分析任务发送通知。

#### 2.2一个类printf函数

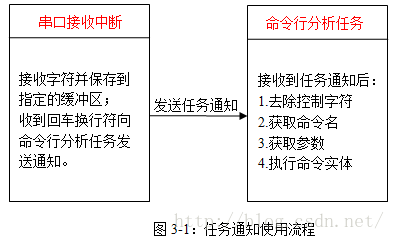
      类printf函数用来格式化输出，我一般用来辅助调试，为了方便的将调试代码从程序中去除，需要将类printf函数进行封装。我的文章《[编写优质嵌入式C程序](http://blog.csdn.net/zhzht19861011/article/details/45508029)》第5.2节给出了一个完整的类printf函数实现和封装代码，最终我们使用到的类printf函数是如下形式的宏：

1. MY\_DEBUGF(CMD\_LINE\_DEBUG,("第%d个参数:%d\n",i+1,arg[i]));

### 3.使用任务通知

      我们将会创建一个任务，用来分析接收到的命令，如果命令有效则调用命令实现函数。这个任务名字为vTaskCmdAnalyze()。串口接收中断用于接收命令，如果接收到回车换行符，则向任务vTaskCmdAnalyze()发送任务通知，表明已经接收到一条完整命令，任务可以去处理了。

       示意框图如图3-1所示。



### 4.数据结构

      命令行解释器程序需要涉及两个[**数据结构**](http://lib.csdn.net/base/datastructure)：一个与命令有关，包括命令的名字、命令的最大参数数目、命令的回调函数类型、命令帮助信息等；另一个与分析命令有关，包括接收命令字符缓冲区、存放参数缓冲区等。

#### 4.1与命令有关的数据结构

       定义如下：

1. typedef struct {
2. char constconst \*cmd\_name;                        //命令字符串
3. int32\_t max\_args;                            //最大参数数目
4. void (\*handle)(int argc,voidvoid \* cmd\_arg);     //命令回调函数
5. char  \*help;                                 //帮助信息
6. }cmd\_list\_struct;

      需要说明一下命令回调函数的参数，argc保存接收到的参数数目，cmd\_arg指向参数缓冲区，目前只支持32位的整形参数，这在绝大多数嵌入式场合是足够的。

#### 4.2与分析命令有关数据结构

       定义如下：

1. #define ARG\_NUM     8          //命令中允许的参数个数
2. #define CMD\_LEN     20         //命令名占用的最大字符长度
3. #define CMD\_BUF\_LEN 60         //命令缓存的最大长度
5. typedef struct {
6. char rec\_buf[CMD\_BUF\_LEN];            //接收命令缓冲区
7. char processed\_buf[CMD\_BUF\_LEN];      //存储加工后的命令(去除控制字符)
8. int32\_t cmd\_arg[ARG\_NUM];             //保存命令的参数
9. }cmd\_analyze\_struct;

      缓冲区的大小使用宏来定义，通过更改相应的宏定义，可以设置整条命令的最大长度、命令参数最大数目等。

### 5.串口接收中断处理函数

      本文使用的串口软件是SecureCRT，在这个软件下敲击的任何键盘字符，都会立刻通过串口硬件发送出去，这与Telnet类似。所以我们无需使用串口的FIFO，每接收到一个字符就产生一次中断。串口中断与硬件关系密切，所以命令行解释器提供了一个与硬件无关的函数fill\_rec\_buf()，每当串口中断接收到一个字符，就以收到的字符为参数调用这个函数。       fill\_rec\_buf()函数主要操作变量cmd\_analyze，变量的声明原型为：

1. cmd\_analyze\_struct cmd\_analyze;

       函数fill\_rec\_buf()的实现代码为：

1. /\*提供给串口中断服务程序，保存串口接收到的单个字符\*/
2. void fill\_rec\_buf(char data)
3. {
4. //接收数据
5. static uint32\_t rec\_count=0;
7. cmd\_analyze.rec\_buf[rec\_count]=data;
8. if(0x0A==cmd\_analyze.rec\_buf[rec\_count] && 0x0D==cmd\_analyze.rec\_buf[rec\_count-1])
9. {
10. BaseType\_t xHigherPriorityTaskWoken = pdFALSE;
11. rec\_count=0;
13. /\*收到一帧数据，向命令行解释器任务发送通知\*/
14. vTaskNotifyGiveFromISR (xCmdAnalyzeHandle,&xHigherPriorityTaskWoken);
16. /\*是否需要强制上下文切换\*/
17. portYIELD\_FROM\_ISR(xHigherPriorityTaskWoken );
18. }
19. else
20. {
21. rec\_count++;
23. /\*防御性代码，防止数组越界\*/
24. if(rec\_count>=CMD\_BUF\_LEN)
25. {
26. rec\_count=0;
27. }
28. }
29. }

### 6.命令行分析任务

      命令行分析任务大部分时间都会因为等待任务通知而处于阻塞状态。当接收到一个通知后，任务首先去除命令行中的无效字符和控制字符，然后找出命令名并分析参数数目、将参数转换成十六进制数并保存到参数缓冲区中，最后检查命令名和参数是否合法，如果合法则调用命令回调函数处理本条命令。

#### 6.1去除无效字符和控制字符

      串口软件SecureCRT支持控制字符。比如在输入一串命令的时候，发现某个字符输入错误，就要使用退格键或者左右移动键定位到错误的位置进行修改。这里的退格键和左右移动键都属于控制字符，比如退格键的键值为0x08、左移键的键值为0x1B0x5B 0x44。我们之前也说过，在软件SecureCRT中输入字符时，每敲击一个字符，该字符立刻通过串口发送给我们的嵌入式设备，也就是所有键值都会按照敲击键盘的顺序存入到接收缓冲区中，但这里面可能有我们不需要的字符，我们首先需要利用控制字符将不需要的字符删除掉。这个工作由函数get\_true\_char\_stream()实现，代码如下所示：

1. /\*\*
2. \* 使用SecureCRT串口收发工具,在发送的字符流中可能带有不需要的字符以及控制字符,
3. \* 比如退格键,左右移动键等等,在使用命令行工具解析字符流之前,需要将这些无用字符以
4. \* 及控制字符去除掉.
5. \* 支持的控制字符有:
6. \*   上移:1B 5B 41
7. \*   下移:1B 5B 42
8. \*   右移:1B 5B 43
9. \*   左移:1B 5B 44
10. \*   回车换行:0D 0A
11. \*  Backspace:08
12. \*  Delete:7F
13. \*/
14. static uint32\_t get\_true\_char\_stream(charchar \*dest,const charchar \*src)
15. {
16. uint32\_t dest\_count=0;
17. uint32\_t src\_count=0;
19. while(src[src\_count]!=0x0D && src[src\_count+1]!=0x0A)
20. {
21. if(isprint(src[src\_count]))
22. {
23. dest[dest\_count++]=src[src\_count++];
24. }
25. else
26. {
27. switch(src[src\_count])
28. {
29. case    0x08:                          //退格键键值
30. {
31. if(dest\_count>0)
32. {
33. dest\_count --;
34. }
35. src\_count ++;
36. }break;
37. case    0x1B:
38. {
39. if(src[src\_count+1]==0x5B)
40. {
41. if(src[src\_count+2]==0x41 || src[src\_count+2]==0x42)
42. {
43. src\_count +=3;              //上移和下移键键值
44. }
45. else if(src[src\_count+2]==0x43)
46. {
47. dest\_count++;               //右移键键值
48. src\_count+=3;
49. }
50. else if(src[src\_count+2]==0x44)
51. {
52. if(dest\_count >0)           //左移键键值
53. {
54. dest\_count --;
55. }
56. src\_count +=3;
57. }
58. else
59. {
60. src\_count +=3;
61. }
62. }
63. else
64. {
65. src\_count ++;
66. }
67. }break;
68. default:
69. {
70. src\_count++;
71. }break;
72. }
73. }
74. }
75. dest[dest\_count++]=src[src\_count++];
76. dest[dest\_count++]=src[src\_count++];
77. return dest\_count;
78. }

#### 6.2参数分析

      接收到的命令中可能带有参数，我们需要知道参数的数目，还需要把字符型的参数转换成整形数并保存到参数缓冲区（这是因为命令回调函数需要这两个参数）。这个工作由函数cmd\_arg\_analyze()实现，代码如下所示：

1. /\*\*
2. \* 命令参数分析函数,以空格作为一个参数结束,支持输入十六进制数(如:0x15),支持输入负数(如-15)
3. \* @param rec\_buf   命令参数缓存区
4. \* @param len       命令的最大可能长度
5. \* @return -1:       参数个数过多,其它:参数个数
6. \*/
7. static int32\_t cmd\_arg\_analyze(charchar \*rec\_buf,unsigned int len)
8. {
9. uint32\_t i;
10. uint32\_t blank\_space\_flag=0;    //空格标志
11. uint32\_t arg\_num=0;             //参数数目
12. uint32\_t index[ARG\_NUM];        //有效参数首个数字的数组索引
14. /\*先做一遍分析,找出参数的数目,以及参数段的首个数字所在rec\_buf数组中的下标\*/
15. for(i=0;i<len;i++)
16. {
17. if(rec\_buf[i]==0x20)        //为空格
18. {
19. blank\_space\_flag=1;
20. continue;
21. }
22. else if(rec\_buf[i]==0x0D)   //换行
23. {
24. break;
25. }
26. else
27. {
28. if(blank\_space\_flag==1)
29. {
30. blank\_space\_flag=0;
31. if(arg\_num < ARG\_NUM)
32. {
33. index[arg\_num]=i;
34. arg\_num++;
35. }
36. else
37. {
38. return -1;      //参数个数太多
39. }
40. }
41. }
42. }
44. for(i=0;i<arg\_num;i++)
45. {
46. cmd\_analyze.cmd\_arg[i]=string\_to\_dec((unsigned charchar \*)(rec\_buf+index[i]),len-index[i]);
47. }
48. return arg\_num;
49. }

      在这个函数cmd\_arg\_analyze()中，调用了字符转整形函数string\_to\_dec()。我们只支持整形参数，这里给出一个字符转整形函数的简单实现，可以识别负号和十六进制的前缀’0x’。在这个函数中调用了三个C库函数，分别是isdigit()、isxdigit()和tolower()，因此需要包含头文件#include <ctype.h>。函数string\_to\_dec()实现代码如下：

1. /\*字符串转10/16进制数\*/
2. static int32\_t string\_to\_dec(uint8\_t \*buf,uint32\_t len)
3. {
4. uint32\_t i=0;
5. uint32\_t base=10;       //基数
6. int32\_t  neg=1;         //表示正负,1=正数
7. int32\_t  result=0;
9. if((buf[0]=='0')&&(buf[1]=='x'))
10. {
11. base=16;
12. neg=1;
13. i=2;
14. }
15. else if(buf[0]=='-')
16. {
17. base=10;
18. neg=-1;
19. i=1;
20. }
21. for(;i<len;i++)
22. {
23. if(buf[i]==0x20 || buf[i]==0x0D)    //为空格
24. {
25. break;
26. }
28. result \*= base;
29. if(isdigit(buf[i]))                 //是否为0~9
30. {
31. result += buf[i]-'0';
32. }
33. else if(isxdigit(buf[i]))           //是否为a~f或者A~F
34. {
35. result+=tolower(buf[i])-87;
36. }
37. else
38. {
39. result += buf[i]-'0';
40. }
41. }
42. result \*= neg;
44. return result ;
45. }

#### 6.3定义命令回调函数

      我们举两个例子：第一个是不带参数的例子，输入命令后，函数返回一个“Helloworld！”字符串；第二个是带参数的例子，我们输入命令和参数后，函数返回每一个参数值。我们在讲数据结构的时候特别提到过命令回调函数的原型，这里要根据这个函数原型来声明命令回调函数。

##### 6.3.1不带参数的命令回调函数举例

1. /\*打印字符串:Hello world!\*/
2. void printf\_hello(int32\_t argc,voidvoid \*cmd\_arg)
3. {
4. MY\_DEBUGF(CMD\_LINE\_DEBUG,("Hello world!\n"));
5. }

##### 6.3.2带参数的命令行回调函数举例

1. /\*打印每个参数\*/
2. void handle\_arg(int32\_t argc,voidvoid \* cmd\_arg)
3. {
4. uint32\_t i;
5. int32\_t  \*arg=(int32\_t \*)cmd\_arg;
7. if(argc==0)
8. {
9. MY\_DEBUGF(CMD\_LINE\_DEBUG,("无参数\n"));
10. }
11. else
12. {
13. for(i=0;i<argc;i++)
14. {
15. MY\_DEBUGF(CMD\_LINE\_DEBUG,("第%d个参数:%d\n",i+1,arg[i]));
16. }
17. }
18. }

#### 6.4定义命令表

      在讲数据结构的时候，我们定义了与命令有关的数据结构。每条命令需要包括命名名、最大参数、命令回调函数、帮助等信息，这里要将每条命令组织成列表的形式。

1. /\*命令表\*/
2. const cmd\_list\_struct cmd\_list[]={
3. /\*   命令    参数数目    处理函数        帮助信息                         \*/
4. {"hello",   0,      printf\_hello,   "hello                      -打印HelloWorld!"},
5. {"arg",     8,      handle\_arg,      "arg<arg1> <arg2> ...      -测试用,打印输入的参数"},
6. };

      如果要定义自己的命令，只需要按照6.3节的格式编写命令回调函数，然后将命令名、参数数目、回调函数和帮助信息按照本节格式加入到命令表中即可。

#### 6.5命令行分析任务实现

      有了上面的基础，命令行分析任务实现起来就非常轻松了，源码如下：

1. /\*命令行分析任务\*/
2. void vTaskCmdAnalyze( voidvoid \*pvParameters )
3. {
4. uint32\_t i;
5. int32\_t rec\_arg\_num;
6. char cmd\_buf[CMD\_LEN];
8. while(1)
9. {
10. uint32\_t rec\_num;
12. ulTaskNotifyTake(pdTRUE,portMAX\_DELAY);
13. rec\_num=get\_true\_char\_stream(cmd\_analyze.processed\_buf,cmd\_analyze.rec\_buf);
15. /\*从接收数据中提取命令\*/
16. for(i=0;i<CMD\_LEN;i++)
17. {
18. if((i>0)&&((cmd\_analyze.processed\_buf[i]==' ')||(cmd\_analyze.processed\_buf[i]==0x0D)))
19. {
20. cmd\_buf[i]='\0';        //字符串结束符
21. break;
22. }
23. else
24. {
25. cmd\_buf[i]=cmd\_analyze.processed\_buf[i];
26. }
27. }
29. rec\_arg\_num=cmd\_arg\_analyze(&cmd\_analyze.processed\_buf[i],rec\_num);
31. for(i=0;i<sizeof(cmd\_list)/sizeof(cmd\_list[0]);i++)
32. {
33. if(!strcmp(cmd\_buf,cmd\_list[i].cmd\_name))       //字符串相等
34. {
35. if(rec\_arg\_num<0 || rec\_arg\_num>cmd\_list[i].max\_args)
36. {
37. MY\_DEBUGF(CMD\_LINE\_DEBUG,("参数数目过多!\n"));
38. }
39. else
40. {
41. cmd\_list[i].handle(rec\_arg\_num,(voidvoid \*)cmd\_analyze.cmd\_arg);
42. }
43. break;
44. }
46. }
47. if(i>=sizeof(cmd\_list)/sizeof(cmd\_list[0]))
48. {
49. MY\_DEBUGF(CMD\_LINE\_DEBUG,("不支持的指令!\n"));
50. }
51. }
52. }

### 7.使用的串口工具

      推荐使用SecureCRT软件，这是我觉得最适合命令行交互的串口工具。此外，这个软件非常强大，除了支持串口，还支持SSH、Telnet等。对于串口，SecureCRT工具还支持文件发送协议：Xmodem、Ymodem和Zmodem。这在使用串口远程升级时很有用，可以用来发送新的程序二进制文件。我曾经使用Ymodem做过远程升级，以后有时间再详细介绍SecureCRT的Ymodem功能细节。

      要用于本文介绍的命令行解释器，要对SecureCRT软件做一些设置。

#### 7.1设置串口参数

      选择Serial功能、设置端口、波特率、校验等，特别要注意的是不要勾选任何流控制选项，如图2-1所示。

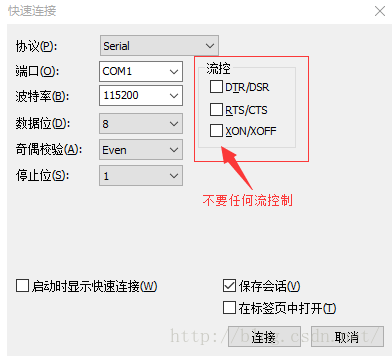


图2-1：设置串口参数

#### 7.2设置新行模式

       依次点击菜单栏的“选项”---“会话选项”，在弹出的“会话选项”界面中，点击左边树形菜单的“终端”---“仿真”---“模式”，在右边的仿真模式区域选中“换行”和“新行模式”，如图2-2所示。

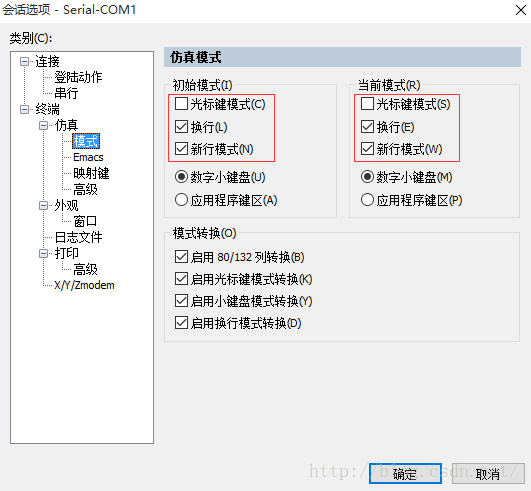


图2-2：设置新行模式

#### 7.3设置本地回显

       依次点击菜单栏的“选项”---“会话选项”，在弹出的“会话选项”界面中，点击左边树形菜单的“终端”---“仿真”---“高级”，在右边的“高级仿真”区域，选中“本地回显”，如图2-3所示。

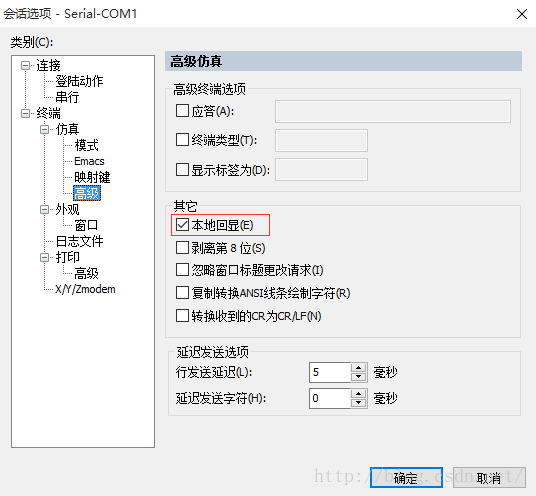


图2-3：设置本地回显

### 8.测试

       我们通过6.3节和6.4接定义了两个命令，第一条命令的名字为”hello”，这是一个无参数命令，直接输出字符串”Hello world！”。第二条命令的名字为”arg”，是一个带参数命令，输出每个参数的值。下面对这两个命令进行[**测试**](http://lib.csdn.net/base/softwaretest)。

#### 8.1无参数命令测试

       设置好SecureCRT软件，输入字符”hello”后，按下回车键，设备会返回字符串”Hello world！”。如图8-1所示。

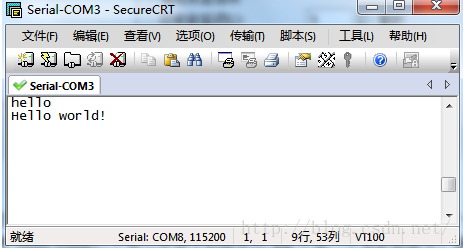
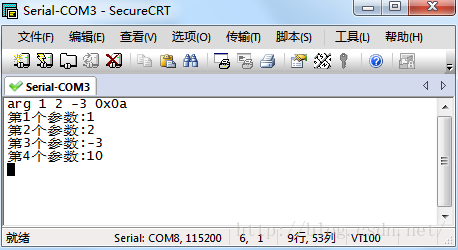


图8-1：无参数命令测试

#### 8.2带参数命令测试

       设置好SecureCRT软件，输入字符”arg 1 2 -3 0x0a”后，按下回车键，设备会返回每个参数值。如图8-2所示。



# 16 [可视化追踪调试](http://blog.csdn.net/zhzht19861011/article/details/50717549)

使用RTOS编程，为每个任务分配多大的堆栈空间就成了一项技术活：分配多了浪费系统资源，分配少了又恐怕会发生堆栈溢出。由于中断和抢占式调度器的存在，我们要估算出一个任务需要多少堆栈是非常困难的，今天我们就介绍一种方法，来获取每个任务的剩余堆栈空间。本文以NXP LPC177x\_8x系列微控制器为例。

      我们将这个功能做成一个命令，添加到《[FreeRTOS系列第15篇---使用任务通知实现命令行解释器](http://blog.csdn.net/zhzht19861011/article/details/50654242)》一文介绍的命令解释列表中。当程序运行一段时间后，我们在SecureCRT软件中输入命令“task”后回车，能看到如图1-1所示的任务信息。这里只有两个任务，其中堆栈一列中的数字，代表对应任务剩余的堆栈空间，单位是StackType\_t类型，这个类型在移植层定义，一般定义为4字节。

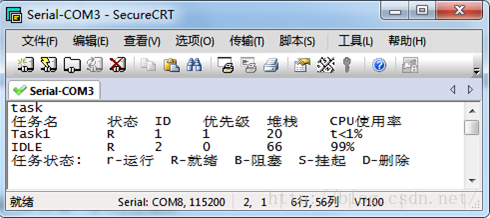


图1-1：任务信息

**1.使能可视化追踪和运行时间统计功能**

      如图1-1所示，要实现堆栈使用量信息以及CPU使用率信息，必须将FreeRTOSConfig.h文件中的两个宏设置为1：

1. #define configUSE\_TRACE\_FACILITY          1
2. #define configGENERATE\_RUN\_TIME\_STATS 1

      第一个宏用来使能可视化追踪功能，第二个宏用来使能运行时间统计功能。如果第二个宏设置为1，则下面两个宏必须被定义：

1. portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS()：用户程序需要提供一个基准时钟函数，函数完成初始化基准时钟功能，这个函数要被define到宏portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS()上。这是因为运行时间统计需要一个比系统节拍中断频率还要高分辨率的基准定时器，否则，统计可能不精确。基准定时器中断频率要比统节拍中断快10~100倍。基准定时器中断频率越快，统计越精准，但能统计的运行时间也越短（比如，基准定时器10ms中断一次，8位无符号整形变量可以计到2.55秒，但如果是1秒中断一次，8位无符号整形变量可以统计到255秒）。
2. portGET\_RUN\_TIME\_COUNTER\_VALUE()：用户程序需要提供一个返回基准时钟当前“时间”的函数，这个函数要被define到宏portGET\_RUN\_TIME\_COUNTER\_VALUE()上。

         我们使用定时器1来产生基准时钟，定时器1初始化函数为：

1. /\*\*
2. \* 初始化计时定时器1,用于OS任务运行时间统计
3. \*/
4. void init\_timer1\_for\_runtime\_state(void)
5. {
6. TIM\_TIMERCFG\_Type Timer0CfgType;
8. Timer0CfgType.PrescaleOption=TIM\_PRESCALE\_USVAL;        //预分频的单位是微秒
9. Timer0CfgType.PrescaleValue=500;                        //预分频后为500微秒,
10. TIM\_Init(LPC\_TIM1,TIM\_TIMER\_MODE,&Timer0CfgType);
12. LPC\_TIM1->TCR=0x01;
13. }

      定时器1被配置成每隔500微秒，TC寄存器值增一。我们将定时器1的 TC寄存器值作为基准时钟当前时间。当TC寄存器值溢出时，大概要经过24.8天，这对于我们这个应用是足够的。

      在FreeRTOSConfig.h中，定义初始化基准定时器宏和获取当前时间宏：

1. extern void init\_timer1\_for\_runtime\_state(void);
2. #define TIMER1\_TC         ( \* ( ( volatile uint32\_t \* )0x40008008 ) )
3. #define portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() init\_timer1\_for\_runtime\_state()
4. #define portGET\_RUN\_TIME\_COUNTER\_VALUE() TIMER1\_TC

**2.获取任务信息并格式化**

      获取每个任务的状态信息使用的是API函数uxTaskGetSystemState()，该函数定义为：

1. UBaseType\_tuxTaskGetSystemState(
2. TaskStatus\_t \* constpxTaskStatusArray,
3. const UBaseType\_tuxArraySize,
4. unsigned longlong \* constpulTotalRunTime );

      函数uxTaskGetSystemState()向TaskStatus\_t结构体填充相关信息，系统中每一个任务的信息都可以填充到TaskStatus\_t结构体数组中，数组大小由uxArraySize指定。结构体TaskStatus\_t定义如下：

1. typedef struct xTASK\_STATUS
2. {
3. /\* 任务句柄\*/
4. TaskHandle\_t xHandle;
6. /\* 指针，指向任务名\*/
7. const signed charchar \*pcTaskName;
9. /\*任务ID，是一个独一无二的数字\*/
10. UBaseType\_t xTaskNumber;
12. /\*填充结构体时，任务当前的状态（运行、就绪、挂起等等）\*/
13. eTaskState eCurrentState;
15. /\*填充结构体时，任务运行（或继承）的优先级。\*/
16. UBaseType\_t uxCurrentPriority;
18. /\* 当任务因继承而改变优先级时，该变量保存任务最初的优先级。仅当configUSE\_MUTEXES定义为1有效。\*/
19. UBaseType\_t uxBasePriority;
21. /\* 分配给任务的总运行时间。仅当宏configGENERATE\_RUN\_TIME\_STATS为1时有效。\*/
22. unsigned long ulRunTimeCounter;
24. /\* 从任务创建起，堆栈剩余的最小数量，这个值越接近0，堆栈溢出的可能越大。 \*/
25. unsigned short usStackHighWaterMark;
26. }TaskStatus\_t;

      注意，这个函数仅用来调试用，调用此函数会挂起所有任务，直到函数结束后才恢复挂起的任务，因此任务可能被挂起很长时间。在文件FreeRTOSConfig.h中，宏configUSE\_TRACE\_FACILITY必须设置为1，此函数才有效。

      由于我们不使用动态内存分配策略，所以实现定义了最大任务个数并预先分配好了存储任务状态信息的数组：

1. #defineMAX\_TASK\_NUM        5
2. TaskStatus\_tpxTaskStatusArray[MAX\_TASK\_NUM];

      正确调用函数uxTaskGetSystemState()后，任务的信息会被放在TaskStatus\_t结构体中，我们需要将这些信息格式化为容易阅读的形式，并共通过串口打印到屏幕。完成这些功能的函数叫做get\_task\_state()，代码如下所示：

1. /\*获取OS任务信息\*/
2. voidget\_task\_state(int32\_t argc,voidvoid \*cmd\_arg)
3. {
4. const chartask\_state[]={'r','R','B','S','D'};
5. volatile UBaseType\_t uxArraySize, x;
6. uint32\_t ulTotalRunTime,ulStatsAsPercentage;
8. /\* 获取任务总数目 \*/
9. uxArraySize = uxTaskGetNumberOfTasks();
10. if(uxArraySize>MAX\_TASK\_NUM)
11. {
12. MY\_DEBUGF(CMD\_LINE\_DEBUG,("当前任务数量过多！\n"));
13. }
15. /\*获取每个任务的状态信息 \*/
16. uxArraySize = uxTaskGetSystemState(pxTaskStatusArray, uxArraySize, &ulTotalRunTime );
18. #if (configGENERATE\_RUN\_TIME\_STATS==1)
20. MY\_DEBUGF(CMD\_LINE\_DEBUG,("任务名      状态  ID    优先级  堆栈    CPU使用率\n"));
22. /\* 避免除零错误 \*/
23. if( ulTotalRunTime > 0 )
24. {
25. /\* 将获得的每一个任务状态信息部分的转化为程序员容易识别的字符串格式 \*/
26. for( x = 0; x < uxArraySize; x++ )
27. {
28. char tmp[128];
30. /\* 计算任务运行时间与总运行时间的百分比。\*/
31. ulStatsAsPercentage =(uint64\_t)(pxTaskStatusArray[ x ].ulRunTimeCounter)\*100 / ulTotalRunTime;
33. if( ulStatsAsPercentage > 0UL )
34. {
36. sprintf(tmp,"%-12s%-6c%-6d%-8d%-8d%d%%",pxTaskStatusArray[ x].pcTaskName,task\_state[pxTaskStatusArray[ x ].eCurrentState],
37. pxTaskStatusArray[ x ].xTaskNumber,pxTaskStatusArray[ x].uxCurrentPriority,
38. pxTaskStatusArray[ x ].usStackHighWaterMark,ulStatsAsPercentage);
39. }
40. else
41. {
42. /\* 任务运行时间不足总运行时间的1%\*/
43. sprintf(tmp,"%-12s%-6c%-6d%-8d%-8dt<1%%",pxTaskStatusArray[x ].pcTaskName,task\_state[pxTaskStatusArray[ x ].eCurrentState],
44. pxTaskStatusArray[ x ].xTaskNumber,pxTaskStatusArray[ x].uxCurrentPriority,
45. pxTaskStatusArray[ x ].usStackHighWaterMark);
46. }
47. MY\_DEBUGF(CMD\_LINE\_DEBUG,("%s\n",tmp));
48. }
49. }
50. MY\_DEBUGF(CMD\_LINE\_DEBUG,("任务状态:   r-运行  R-就绪  B-阻塞  S-挂起  D-删除\n"));
51. #endif //#if (configGENERATE\_RUN\_TIME\_STATS==1)
52. }

**3.添加到命令解释列表**

      在《FreeRTOS系列第15篇---使用任务通知实现命令行解释器》一文我们讲过了命令表，这里只需要将get\_task\_state()函数添加到命令列表中，命令设置为”task”，代码如下所示：

1. /\*命令表\*/
2. const cmd\_list\_structcmd\_list[]={
3. /\*   命令    参数数目    处理函数        帮助信息                                  \*/
4. {"?",       0,     handle\_help,     "?                                  -打印帮助信息"},
5. {"reset",   0,     handle\_reset,    "reset                              -重启控制器"},
6. {"arg",     8,     handle\_arg,      "arg<arg1> <arg2> ...               -测试用,打印输入的参数"},
7. {"hello",   0,     printf\_hello,    "hello                              -打印HelloWorld!"},
8. {"task",    0,     get\_task\_state,  "task                               -获取任务信息"},
9. };

# 17 [FreeRTOS队列](http://blog.csdn.net/zhzht19861011/article/details/50736067)

本文介绍队列的基本知识，详细源码分析见《[FreeRTOS高级篇5---FreeRTOS队列分析](http://blog.csdn.net/zhzht19861011/article/details/51510384)》

**1.FreeRTOS队列**

      队列是主要的任务间通讯方式。可以在任务与任务间、中断和任务间传送信息。大多数情况下，队列用于具有线程保护的FIFO（先进先出）缓冲区：新数据放在队列的后面。当然，数据也可以放在队列的前面，在下一篇讲队列API函数时，会涉及到数据的存放位置。

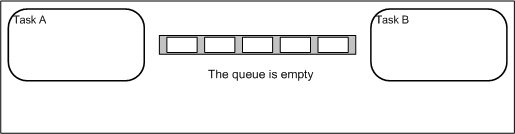


图1-1：读写队列

      图1-1所示的队列中，最多能保存5个项目，并且假设队列永远不会满。任务A使用API函数xQueueSendToBack()向队列发送数据，每次发送一个数据，新入队的数据置于上一次入队数据的后面。任务B使用API函数xQueueReceive()将数据从队列取出，先入队的数据先出队。

**2.使用模型：最简单、最灵活**

      通常情况下，鱼和熊掌是不可兼得的，但FreeRTOS的队列用户模型管理却兼顾简单和灵活。发送到队列的消息是通过拷贝实现的，这意味着队列存储的数据是原数据，而不是原数据的引用。FreeRTOS队列具有以下特性：

* C变量（整形、简单结构体等等）中的简单信息可以直接传送到队列。这样就不需要为信息分配缓存也不需要再进行什么拷贝工作。同样的，信息可以直接从队列读取到C变量中。用直接拷贝的方法入队，可以允许任务立即覆写已经入队的变量或者缓存，实际上队列中已经保存了这些变量或缓冲区携带的信息。因为变量中的数据内容是以拷贝的方式入队的，所以变量自身是允许重复使用的。发送信息的任务和接收信息的任务并不需要就哪个任务拥有信息、哪个任务释放信息（当信息不再使用时）而达成一致。
* 队列是通过拷贝传递数据的，但这并不妨碍队列通过引用来传递数据。当信息的大小到达一个临界点后，逐字节拷贝整个信息是不实际的，可以定义一个指针队列，只拷贝指向消息的指针来代替整个信息拷贝。FreeRTOS+UDP IP栈例程正是使用这种方法向FreeRTOS协议栈传递大量网络数据的。
* 队列内存区域分配由内核完成。
* 变长消息可以通过定义保存一个结构体变量的队列实现，结构体一个成员指向要入队的缓存，另一个成员保存缓存数据的大小。
* 单个队列可以接收不同类型信息，并且信息可以来自不同的位置。通过定义保存一个结构体变量的队列来实现，结构体的一个成员保存信息类型，另一个成员保存信息数据（或者指向信息数据的指针）。数据如何解读取决于信息类型。管理FreeRTOS+UDP IP栈的任务正是使用单个队列接收ARP定时器时间通知、以太网硬件传送来的数据包、从应用层传送来的数据包、网络关闭事件等等。
* 天生适用于那些内存保护（MPU）场合。一个具有内存区域保护的任务可以向另一个具有内存区域保护的任务传递数据，因为调用队列发送函数会引起RTOS提升微控制器特权级别。只有RTOS（具有所有特权）才可以访问队列存储区域。
* 在中断函数中使用独立的API。将RTOS任务API和中断服务例程API分来实现意味着可以避免执行时的上下文调用检查开销，还意味着在大多数情况下，与其它RTOS产品相比，用户创建中断服务例程会更简单。
* API函数很简单。

**3.队列阻塞**

      API函数允许指定阻塞时间。

      每当任务企图从一个空的队列读取数据时，任务会进入阻塞状态（这样任务不会消耗任何CPU时间并且另一个任务可以运行）直到队列中出现有效数据或者阻塞时间到期。

      每当任务企图向一个满的队列写数据时，任务会进入阻塞状态，直到队列中出现有效空间或者阻塞时间到期。

      如果多个任务阻塞在一个队列上，那么最高优先级别的任务会第一个解除阻塞。

      注：中断程序中绝不可以使用不带“FromISR”结尾的API函数！

      总结一下队列的基本用法：

1. 定义一个队列句柄变量，用于保存创建的队列：xQueueHandle xQueue1;
2. 使用API函数xQueueCreate()创建一个队列。
3. 如果希望使用先进先出队列，使用API函数xQueueSend()或xQueueSendToBack()向队列投递队列项。如果希望使用后进先出队列，使用API函数xQueueSendToFront()向队列投递队列项。如果在中断服务程序中，切记使用它们的带中断保护版本。
4. 使用API函数xQueueReceive()从队列读取队列项，如果在中断服务程序中，切记使用它们的带中断保护版本。

      以上使用的API函数将在下一篇文章中介绍。

# 18 [FreeRTOS队列API函数](http://blog.csdn.net/zhzht19861011/article/details/50800538)

FreeRTOS为操作队列提供了非常丰富的API函数，包括队列的创建、删除，灵活的入队和出队方式、带中断保护的入队和出队等等。下面就来详细讲述这些API函数。

### 1.获取队列入队信息数目

#### ****1.1函数描述****

      UBaseType\_t uxQueueMessagesWaiting( QueueHandle\_t xQueue );

      返回队列中存储的信息数目。具有中断保护的版本为uxQueueMessagesWaitingFromISR()，原型为：UBaseType\_t uxQueueMessagesWaitingFromISR( const QueueHandle\_t xQueue )。

#### ****1.2参数描述****

* xQueue：队列句柄

### 2.获取队列的空闲数目

#### ****2.1函数描述****

      UBaseType\_t uxQueueSpacesAvailable( QueueHandle\_t xQueue );

      返回队列的空闲数目。

#### ****2.2参数描述****

* xQueue：队列句柄

### 3.删除队列

#### ****3.1函数描述****

      void vQueueDelete( QueueHandle\_t xQueue );

      删除队列并释放所有分配给队列的内存。

**3.2参数描述**

* xQueue：队列句柄

### 4.复位队列

#### ****4.1函数描述****

      BaseType\_t xQueueReset( QueueHandle\_t xQueue );

      将队列复位到初始状态。

#### ****4.2参数描述****

* xQueue：队列句柄

#### ****4.3返回值****

      FreeRTOSV7.2.0以及以后的版本总是返回pdPASS。

### 5.创建队列

#### ****5.1函数描述****

     QueueHandle\_t xQueueCreate (UBaseType\_t uxQueueLength, UBaseType\_t uxItemSize);

     创建新队列。为新队列分配指定的存储空间并返回队列句柄。

#### ****5.2参数描述****

* usQueueLength：队列项数目
* uxItemSize：每个队列项大小，单位是字节。队列项通过拷贝入队而不是通过引用入队，因此需要队列项的大小。每个队列项的大小必须相同。

#### ****5.3返回值****

      成功创建队列返回队列句柄，否自返回0。

#### ****5.4用法举例****

1. struct AMessage
2. {
3. portCHAR ucMessageID;
4. portCHAR ucData[ 20 ];
5. };
6. void vATask( void\*pvParameters )
7. {
8. xQueueHandle xQueue1, xQueue2;
9. // 创建一个队列，队列能包含10个unsigned long类型的值。
10. xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ));
11. if( xQueue1 ==0 )
12. {
13. // 队列创建失败，不可以使用
14. }
15. // 创建一个队列，队列能包含10个 Amessage结构体指针类型的值。
16. // 这样可以通过传递指针变量来包含大量数据。
17. xQueue2 =xQueueCreate( 10, sizeof( struct AMessage \* ) );
18. if( xQueue2 ==0 )
19. {
20. // 队列创建失败，不可以使用
21. }
22. // ... 任务的其它代码.
23. }

### 6.向队列投递队列项

#### 6.1 函数描述

      BaseType\_t xQueueSend(QueueHandle\_t xQueue,  const void \* pvItemToQueue,  TickType\_t xTicksToWait );

      其实是一个宏，真正被调用的函数是xQueueGenericSend()。定义这个宏是为了向后兼容那些不包含函数xQueueSendToFront()和xQueueSendToBack()宏的FreeRTOS版本。它与xQueueSendToBack()等同。

      这个宏向队列尾部投递一个队列项。项目以拷贝的形式入队，而不是引用形式入队。绝不可以在中断服务例程中调用这个宏，使用带有中断保护的版本xQueueSendFromISR()来完成相同的功能。

6.2参数描述

* xQueue：队列句柄。
* pvItemToQueue：指针，指向要入队的项目。要保存到队列中的项目字节数在队列创建时就已确定。因此要从指针pvItemToQueue指向的区域拷贝到队列存储区域的字节数，也已确定。
* xTicksToWait：如果队列满，任务等待队列空闲的最大时间。如果队列满并且xTicksToWait被设置成0，函数立刻返回。时间单位为系统节拍时钟周期，因此宏portTICK\_PERIOD\_MS可以用来辅助计算真实延时值。如果INCLUDE\_vTaskSuspend设置成1，并且指定延时为portMAX\_DELAY将引起任务无限阻塞（没有超时）。

#### 6.3返回值

      队列项入队成功返回pdTRUE，否则返回errQUEUE\_FULL。

#### 6.4用法举例

1. struct AMessage
2. {
3. portCHAR ucMessageID;
4. portCHAR ucData[ 20 ];
5. }xMessage;
6. unsigned portLONG ulVar = 10UL;
7. void vATask( voidvoid \*pvParameters )
8. {
9. xQueueHandle xQueue1, xQueue2;
10. struct AMessage \*pxMessage;
11. /\*创建一个队列，队列能包含10个unsigned long类型的值。\*/
12. xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );
13. /\* 创建一个队列，队列能包含10个 Amessage结构体指针类型的值。
14. 这样可以通过传递指针变量来包含大量数据。\*/
15. xQueue2 = xQueueCreate( 10, sizeof( struct AMessage \* ) );
16. // ...
17. if( xQueue1 != 0 )
18. {
19. /\*1个unsigned long型数据入队.如果需要等待队列空间变的有效，
20. 会最多等待10个系统节拍周期\*/
21. if( xQueueSend( xQueue1, ( voidvoid \* ) &ulVar, ( portTickType ) 10 ) !=pdPASS )
22. {
23. /\*消息入队失败\*/
24. }
25. }
26. if( xQueue2 != 0 )
27. {
28. /\* 发送一个指向结构体Amessage的对象，如果队列满也不等待 \*/
29. pxMessage = & xMessage;
30. xQueueSend( xQueue2, ( voidvoid \* ) &pxMessage, ( portTickType ) 0 );
31. }
32. //... 任务其余代码.
33. }

### 7.向队列投递队列项（带中断保护）

#### 7.1函数描述

1. BaseType\_t xQueueSendFromISR (QueueHandle\_t xQueue,
2. const voidvoid \*pvItemToQueue,  BaseType\_t \*pxHigherPriorityTaskWoken);

      其实是一个宏，真正被调用的函数是xQueueGenericSendFromISR()。这个宏是xQueueSend()的中断保护版本，用于中断服务程序，等价于xQueueSendToBackFromISR()。

    在中断服务例程中向队列尾部投递一个队列项。

#### 7.2参数描述

* xQueue：队列句柄。
* pvItemToQueue：指针，指向要入队的项目。要保存到队列中的项目字节数在队列创建时就已确定。因此要从指针pvItemToQueue指向的区域拷贝到队列存储区域的字节数，也已确定。
* pxHigherPriorityTaskWoken：如果入队导致一个任务解锁，并且解锁的任务优先级高于当前运行的任务，则该函数将\*pxHigherPriorityTaskWoken设置成pdTRUE。如果xQueueSendFromISR()设置这个值为pdTRUE，则中断退出前需要一次上下文切换。从FreeRTOS V7.3.0起，pxHigherPriorityTaskWoken称为一个可选参数，并可以设置为NULL。

#### 7.3返回值

      列项入队成功返回pdTRUE，否则返回errQUEUE\_FULL。

#### 7.4用法举例

1. void vBufferISR( void )
2. {
3. portCHARcIn;
4. portBASE\_TYPE xHigherPriorityTaskWoken;
6. /\* 初始化，没有唤醒任务\*/
7. xHigherPriorityTaskWoken = pdFALSE;
9. /\* 直到缓冲区为空 \*/
10. do
11. {
12. /\* 从缓冲区获得一个字节数据 \*/
13. cIn = portINPUT\_BYTE( RX\_REGISTER\_ADDRESS );
15. /\* 投递这个数据 \*/
16. xQueueSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );
17. }while( portINPUT\_BYTE( BUFFER\_COUNT ) );
19. /\* 这里缓冲区已空，如果需要进行一个上下文切换\*/
20. /\*根据不同移植平台，这个函数也不同\*/
21. portYIELD\_FROM\_ISR(xHigherPriorityTaskWoken);
22. }

### 8.向队列尾部投递队列项

#### 8.1函数描述

1. BaseType\_t xQueueSendToBack(QueueHandle\_t xQueue,
2. const voidvoid \* pvItemToQueue, TickType\_t xTicksToWait );

      其实是一个宏，真正被调用的函数是xQueueGenericSend()。这个宏等价于xQueueSend()。

      向队列尾投递一个队列项。绝不可以在中断中调用这个宏，可以使用带有中断保护的版本xQueueSendToBackFromISR ()来完成相同功能。

#### 8.2参数描述

      同xQueueSend()。

#### 8.3返回值

      同xQueueSend()。

#### 8.4用法举例

      同xQueueSend()。

### 9.向队列尾部投递队列项（带中断保护）

#### 9.1函数描述

1. BaseType\_t xQueueSendToBackFromISR (QueueHandle\_t xQueue,
2. const voidvoid \*pvItemToQueue, BaseType\_t \*pxHigherPriorityTaskWoken );

      其实是一个宏，真正被调用的函数是xQueueGenericSendFromISR()。这个宏是xQueueSendToBack()的中断保护版本，用于中断服务程序，等价于xQueueSendFromISR()。

    在中断服务例程中向队列尾部投递一个队列项。

#### 9.2参数描述

      同QueueSendFromISR()。

#### 9.3返回值

      同QueueSendFromISR()。

#### 9.4用法举例

      同QueueSendFromISR()。

### 10.向队列首部投递队列项

#### 10.1函数描述

1. BaseType\_t xQueueSendToFront(QueueHandle\_t xQueue,
2. const voidvoid \* pvItemToQueue,TickType\_t xTicksToWait);

      其实是一个宏，真正被调用的函数是xQueueGenericSend()。

      这个宏向队列首部投递一个队列项。绝不可以在中断服务例程中调用这个宏，可以使用带有中断保护的版本xQueueSendToFrontFromISR ()来完成相同功能。

#### 10.2参数描述

* xQueue：队列句柄。
* pvItemToQueue：指针，指向要入队的项目。要保存到队列中的项目字节数在队列创建时就已确定。因此要从指针pvItemToQueue指向的区域拷贝到队列存储区域的字节数，也已确定。
* xTicksToWait：如果队列满，任务等待队列空闲的最大时间。如果队列满并且xTicksToWait被设置成0，函数立刻返回。时间单位为系统节拍时钟周期，因此宏portTICK\_PERIOD\_MS可以用来辅助计算真实延时值。如果INCLUDE\_vTaskSuspend设置成1，并且指定延时为portMAX\_DELAY将引起任务无限阻塞（没有超时）。

#### 10.3返回值

      队列项入队成功返回pdTRUE，否则返回errQUEUE\_FULL。

### 11.向队列首部投递队列项（带中断保护）

#### 11.1函数描述

1. BaseType\_t xQueueSendToFrontFromISR (QueueHandle\_t xQueue,
2. const voidvoid \*pvItemToQueue,BaseType\_t \*pxHigherPriorityTaskWoken);

      其实是一个宏，真正被调用的函数是xQueueGenericSendFromISR()。这个宏是xQueueSendToFront ()的中断保护版本，用于中断服务程序。

#### 11.2参数描述

* xQueue：队列句柄。
* pvItemToQueue：指针，指向要入队的项目。要保存到队列中的项目字节数在队列创建时就已确定。因此要从指针pvItemToQueue指向的区域拷贝到队列存储区域的字节数，也已确定。
* pxHigherPriorityTaskWoken：如果入队导致一个任务解锁，并且解锁的任务优先级高于当前运行的任务，则该函数将\*pxHigherPriorityTaskWoken设置成pdTRUE。如果xQueueSendFromISR()设置这个值为pdTRUE，则中断退出前需要一次上下文切换。从FreeRTOS V7.3.0起，pxHigherPriorityTaskWoken称为一个可选参数，并可以设置为NULL。

#### 11.3返回值

      列项入队成功返回pdTRUE，否则返回errQUEUE\_FULL。

### 12.读取并移除队列项

#### 12.1函数描述

1. BaseType\_t xQueueReceive(QueueHandle\_t xQueue,
2. voidvoid \*pvBuffer,TickType\_t xTicksToWait);

      其实是一个宏，真正被调用的函数是xQueueGenericReceive()。

      这个宏从队列中读取一个队列项并把该队列项从队列中删除。读取队列项是以拷贝的形式完成，而不是以引用的形式，因此必须提供足够大的缓冲区以便容纳队列项。参数pvBuffer指向这个缓冲区。

      绝不可以在中断服务例程中调用这个宏，可以使用使用带有中断保护的版本xQueueReceiveFromISR来完成相同功能。

#### 12.2参数描述

* pxQueue：队列句柄。
* pvBuffer：指向一个缓冲区，用于拷贝接收到的列表项。
* xTicksToWait：要接收的项目队列为空时，允许任务最大阻塞时间。如果设置该参数为0，则表示即队列为空也立即返回。阻塞时间的单位是系统节拍周期，宏portTICK\_RATE\_MS可辅助计算真实阻塞时间。如果INCLUDE\_vTaskSuspend设置成1，并且阻塞时间设置成portMAX\_DELAY，将会引起任务无限阻塞（不会有超时）。

#### 12.3返回值

      成功接收到列表项返回pdTRUE，否则返回pdFALSE。

#### 12.4用法举例

1. struct AMessage
2. {
3. portCHAR ucMessageID;
4. portCHAR ucData[ 20 ];
5. } xMessage;
7. xQueueHandle xQueue;
9. // 创建一个队列并投递一个值
10. void vATask( voidvoid \*pvParameters )
11. {
12. struct AMessage \*pxMessage;
14. // 创建一个队列，队列能包含10个 Amessage结构体指针类型的值。
15. // 这样可以通过传递指针变量来包含大量数据。
16. xQueue =xQueueCreate( 10, sizeof( struct AMessage \* ) );
17. if( xQueue == 0)
18. {
19. // 创建队列失败
20. }
21. // ...
22. // 向队列发送一个指向结构体对象Amessage的指针，如果队列满不等待
23. pxMessage = & xMessage;
24. xQueueSend(xQueue, ( voidvoid \* ) &pxMessage, ( portTickType ) 0 );
25. // ... 其它代码
26. }
28. // 该任务从队列中接收一个队列项
29. voidvADifferentTask( voidvoid \*pvParameters )
30. {
31. struct AMessage \*pxRxedMessage;
33. if( xQueue != 0)
34. {
35. // 从创建的队列中接收一个消息，如果消息无效，最多阻塞10个系统节拍周期
36. if(xQueueReceive( xQueue, &( pxRxedMessage ), ( portTickType ) 10 ) )
37. {
38. // 现在pcRxedMessage 指向由vATask任务投递进来的结构体Amessage变量
39. }
40. }
41. // ... 其它代码
42. }

### 13读取并移除队列项（带中断保护）

#### 13.1函数描述

1. BaseType\_t xQueueReceiveFromISR (QueueHandle\_t xQueue,
2. voidvoid \*pvBuffer, BaseType\_t \*pxHigherPriorityTaskWoken);

      从队列中读取一个队列项并把该队列项从队列中删除。功能与xQueueReceive()相同，用于中断服务函数。

#### 13.2参数描述

* pxQueue：队列句柄。
* pvBuffer：指向一个缓冲区，用于拷贝接收到的列表项。
* pxHigherPriorityTaskWoken：如果入队导致一个任务解锁，并且解锁的任务优先级高于当前运行的任务，则该函数将\*pxHigherPriorityTaskWoken设置成pdTRUE。如果xQueueSendFromISR()设置这个值为pdTRUE，则中断退出前需要一次上下文切换。从FreeRTOS V7.3.0起，pxHigherPriorityTaskWoken称为一个可选参数，并可以设置为NULL。

#### 13.3返回值

      成功接收到列表项返回pdTRUE，否则返回pdFALSE。

#### 13.4用法举例

1. xQueueHandle xQueue;
3. /\* 该函数创建一个队列并投递一些值 \*/
4. voidvAFunction( voidvoid \*pvParameters )
5. {
6. portCHAR cValueToPost;
7. const portTickType xBlockTime = (portTickType )0xff;
9. /\*创建一个队列，可以容纳10个portCHAR型变量 \*/
10. xQueue = xQueueCreate( 10, sizeof( portCHAR ) );
11. if( xQueue == 0 )
12. {
13. /\* 队列创建失败 \*/
14. }
15. /\*…... \*/
16. /\* 投递一些字符，在ISR中使用。如果队列满，任务将会阻塞xBlockTime 个系统节拍周期 \*/
17. cValueToPost = 'a';
18. xQueueSend( xQueue, ( voidvoid \* ) &cValueToPost, xBlockTime );
19. cValueToPost = 'b';
20. xQueueSend( xQueue, ( voidvoid \* ) &cValueToPost, xBlockTime );
21. /\*... 继续投递字符 ... 当队列满时，这个任务会阻塞\*/
22. cValueToPost = 'c';
23. xQueueSend( xQueue, ( voidvoid \* ) &cValueToPost, xBlockTime );
24. }
26. /\* ISR：输出从队列接收到的所有字符 \*/
27. voidvISR\_Routine( void )
28. {
29. portBASE\_TYPE xTaskWokenByReceive = pdFALSE;
30. portCHAR cRxedChar;
32. while( xQueueReceiveFromISR( xQueue, ( voidvoid \*) &cRxedChar, &xTaskWokenByReceive) )
33. {
34. /\* 接收到一个字符串，输出.\*/
35. vOutputCharacter( cRxedChar );
36. /\* 如果从队列移除一个字符串后唤醒了向此队列投递字符的任务，那么参数xTaskWokenByReceive将会设置成pdTRUE，这个循环无论重复多少次，仅会
37. 有一个任务被唤醒。\*/
38. }
39. /\*这里缓冲区已空，如果需要进行一个上下文切换根据不同移植平台，这个函数也不同 \*/
40. portYIELD\_FROM\_ISR(xTaskWokenByReceive);
41. }

### 14.读取但不移除队列项

#### 14.1函数描述

1. BaseType\_t xQueuePeek(QueueHandle\_t xQueue,
2. voidvoid \*pvBuffer, TickType\_t xTicksToWait);

      其实是一个宏，真正被调用的函数是xQueueGenericReceive()。

      这个宏从队列中读取一个队列项，但不会把该队列项从队列中移除。这个宏绝不可以用在中断服务例程中，可以使用使用带有中断保护的版本xQueuePeekFromIS()来完成相同功能。

#### 14.2参数描述

      同xQueueReceive()。

#### 14.3返回值

      同xQueueReceive()。

#### 14.4用法举例

      同xQueueReceive()。

### 15.读取但不移除队列项（带中断保护）

#### 15.1函数描述

      BaseType\_t xQueuePeekFromISR(QueueHandle\_t xQueue, void \*pvBuffer,);

      功能与xQueuePeek()相同，用于中断服务程序。

#### 15.2参数描述

* pxQueue：队列句柄。
* pvBuffer：指向一个缓冲区，用于拷贝接收到的列表项。

#### 15.3返回值

      成功接收到列表项返回pdTRUE，否则返回pdFALSE。

### 16.队列注册

#### 16.1函数描述

      void vQueueAddToRegistry(QueueHandle\_t xQueue, char \*pcQueueName,);

      为队列分配名字并进行注册。

#### 16.2参数描述

* l  xQueue：队列句柄
* l  pcQueueName：分配给队列的名字。这仅是一个有助于调试的字符串。队列注册仅存储指向队列名字符串的指针，因此这个字符串必须是静态的（全局变量活着存储在ROM/Flash中），不可以定义到堆栈中。

      队列注册有两个目的，这两个目的都是为了调试RTOS内核：

1. 它允许队列具有一个相关的文本名字，在GUI调试中可以容易的标识队列；
2. 包含调试器用于定位每一个已经注册的队列和信号量时所需的信息。

       队列注册仅用于调试器。

       宏configQUEUE\_REGISTRY\_SIZE定义了可以注册的队列和信号量的最大数量。仅当你想使用可视化调试内核时，才进行队列和信号量注册。

#### 16.3用法举例

1. void vAFunction( void )
2. {
3. xQueueHandle xQueue;
5. /\*创建一个队列，可以容纳10个char类型数值 \*/
6. xQueue = xQueueCreate( 10, sizeof( portCHAR ) );
8. /\* 我们想可视化调试，所以注册它\*/
9. vQueueAddToRegistry( xQueue, "AMeaningfulName" );
10. }

### 17.解除注册

#### 17.1函数描述

      void vQueueUnregisterQueue(QueueHandle\_t xQueue);

      从队列注册表中移除指定的队列。

17.2参数描述

* xQueue：队列句柄

### 18.查询队列是否为空（仅用于中断服务程序）

#### 18.1函数描述

      BaseType\_t xQueueIsQueueEmptyFromISR( const QueueHandle\_t xQueue );

      查询队列是否为空。这个函数仅用于ISR。

#### 18.2参数描述

* xQueue：队列句柄

#### 18.3返回值

      队列非空返回pdFALSE，其它值表示队列为空。

### 19.查询队列是否满（仅用于中断服务程序）

#### 19.1函数描述

      BaseType\_t xQueueIsQueueFullFromISR( const QueueHandle\_t xQueue );

      查询队列是否满，仅用于ISR。

#### 19.2参数描述

* xQueue：队列句柄

#### 19.3返回值

      队列没有满返回pdFALSE，其它值表示队列满。

### 20.向队列尾部覆盖式投递队列项

#### 20.1函数描述

      BaseType\_t xQueueOverwrite(QueueHandle\_t xQueue, const void \* pvItemToQueue);

      其实是个宏，真正被调用的函数是xQueueGenericSend()。这个宏是xQueueSendToBack()的另一个版本，向队列尾投递一个队列项，如果队列已满，则覆盖之前的队列项。一般用于只有一个队列项的队列中，如果队列的队列项超过1个，使用这个宏会触发一个断言（已经正确定义configASSERT()的情况下）。这个宏绝不可以在中断服务程序中调用，可以使用使用带有中断保护的版本xQueueOverwriteFromISR()来完成相同功能。

#### 20.2参数描述

* xQueue：队列句柄。
* pvItemToQueue：指针，指向要入队的项目。要保存到队列中的项目字节数在队列创建时就已确定。因此要从指针pvItemToQueue指向的区域拷贝到队列存储区域的字节数，也已确定。

#### 20.3返回值

      总是返回pdPASS。

#### 20.4用法举例

1. void vFunction( voidvoid \*pvParameters )
2. {
3. QueueHandle\_t xQueue;
4. unsigned long ulVarToSend, ulValReceived;
6. /\*创建队列，保存一个unsignedlong值。如果一个队列的队列项超过1个，强烈建议不要使用xQueueOverwrite()，如果使用xQueueOverwrite()会触发一个断言（已经正确定义configASSERT()的情况下）。\*/
7. xQueue = xQueueCreate( 1, sizeof( unsigned long ) );
9. /\*使用 xQueueOverwrite().向队列写入10\*/
10. ulVarToSend = 10;
11. xQueueOverwrite( xQueue, &ulVarToSend );
13. /\*从队列读取值，但是不把这个值从队列中删除。\*/
14. ulValReceived = 0;
15. xQueuePeek( xQueue, &ulValReceived, 0 );
17. if( ulValReceived != 10 )
18. {
19. /\* 处理错误\*/
20. }
22. /\*到这里队列仍是满的。使用xQueueOverwrite()覆写队列，写入值100 \*/
23. ulVarToSend = 100;
24. xQueueOverwrite( xQueue, &ulVarToSend );
26. /\* 从队列中读取值\*/
27. xQueueReceive( xQueue, &ulValReceived, 0 );
29. if( ulValReceived != 100 )
30. {
31. /\*处理错误 \*/
32. }
34. /\* ... \*/
35. }

### 21向队列尾部覆盖式投递队列项（带中断保护）

#### 21.1函数描述

1. BaseType\_t xQueueOverwriteFromISR (QueueHandle\_t xQueue, const voidvoid \* pvItemToQueue,
2. BaseType\_t \*pxHigherPriorityTaskWoken);

      其实是个宏，真正被调用的函数是xQueueGenericSendFromISR()。这个宏的功能与xQueueOverwrite()相同，用在中断服务程序中。

#### 21.2参数描述

* xQueue：队列句柄。
* pvItemToQueue：指针，指向要入队的项目。要保存到队列中的项目字节数在队列创建时就已确定。因此要从指针pvItemToQueue指向的区域拷贝到队列存储区域的字节数，也已确定。
* pxHigherPriorityTaskWoken：如果入队导致一个任务解锁，并且解锁的任务优先级高于当前运行的任务，则该函数将\*pxHigherPriorityTaskWoken设置成pdTRUE。如果xQueueSendFromISR()设置这个值为pdTRUE，则中断退出前需要一次上下文切换。从FreeRTOS V7.3.0起，pxHigherPriorityTaskWoken称为一个可选参数，并可以设置为NULL。

#### 21.3返回值

      总是返回pdPASS。

# 19 [FreeRTOS信号量](http://blog.csdn.net/zhzht19861011/article/details/50835613)

本文介绍信号量的基础知识，详细源码分析见《[FreeRTOS高级篇6---FreeRTOS信号量分析](http://blog.csdn.net/zhzht19861011/article/details/51537234)》

**1.信号量简介**

      FreeRTOS的信号量包括二进制信号量、计数信号量、互斥信号量（以后简称互斥量）和递归互斥信号量（以后简称递归互斥量）。

      我们可以把互斥量和递归互斥量看成特殊的信号量。互斥量和信号量在用法上不同：

* 信号量用于同步，任务间或者任务和中断间同步；互斥量用于互锁，用于保护同时只能有一个任务访问的资源，为资源上一把锁。
* 信号量用于同步时，一般是一个任务（或中断）给出信号，另一个任务获取信号；互斥量必须在同一个任务中获取信号、同一个任务给出信号。
* 互斥量具有优先级继承，信号量没有。
* 互斥量不能用在中断服务程序中，信号量可以。
* 创建互斥量和创建信号量的API函数不同，但是共用获取和给出信号API函数；

**2.二进制信号量**

      二进制信号量既可以用于互斥功能也可以用于同步功能。

      二进制信号量和互斥量非常相似，但是有一些细微差别：互斥量包含一个优先级继承机制，二进制信号量则没有这个机制。这使得二进制信号量更好的用于实现同步（任务间或任务和中断间），互斥量更好的用于实现简单互斥。本节仅描述用于同步的二进制信号量。

      信号量API函数允许指定一个阻塞时间。当任务企图获取一个无效信号量时，任务进入阻塞状态，阻塞时间用来确定任务进入阻塞的最大时间，阻塞时间单位为系统节拍周期时间。如果有多个任务阻塞在同一个信号量上，那么当信号量有效时，具有最高优先级别的任务最先解除阻塞。

      可以将二进制信号量看作只有一个项目（item）的队列，因此这个队列只能为空或满（因此称为二进制）。任务和中断使用队列无需关注谁控制队列---只需要知道队列是空还是满。利用这个机制可以在任务和中断之间同步。

      考虑这样一种情况，一个任务用来维护外设。使用轮询的方法会浪费CPU资源并且妨碍其它任务执行。更好的做法是任务的大部分时间处于阻塞状态（允许其它任务执行），直到某些事件发生该任务才执行。可以使用二进制信号量实现这种应用：当任务取信号量时，因为此时尚未发生特定事件，信号量为空，任务会进入阻塞状态；当外设需要维护时，触发一个中断服务例程，该中断服务仅仅给出信号量（向队列写数据）。任务只是取信号，并不需要归还，中断服务只是给信号。

      任务的优先级可以用于确保外设及时获得维护。还可以使用队列来代替二进制信号量。中断例程可以捕获与外设事件相关的数据并将它发往任务的队列。任务发现队列数据有效时解除阻塞，如果需要，则进行数据处理。第二种方案使得中断执行尽可能的短，其它处理过程可以在任务中实现。

      注：中断程序中决不可使用无“FromISR”结尾的API函数。

      注：在大部分应用场合，任务通知都可以代替二进制信号量，并且速度更快、生成的代码更少。



图1-1：中断和任务之间同步---使用信号量

      如图1-1所示，程序开始运行时，信号量无效，因此任务阻塞在这个信号量下。一段时间后，一个中断发生，在中断服务程序中使用API函数xSemaphoreGiveFromISR()给出了一个信号，信号量变得有效。当退出中断服务程序后，执行上下文切换，任务解除阻塞，使用API函数xSemaphoreTake()取走信号量并执行任务。之后信号量变得无效，任务再次进入阻塞。

**3.计数信号量**

      二进制信号量可以被认为是长度为1的队列，计数信号量则可以被认为长度大于1的队列。此外，信号量使用者不必关心存储在队列中的数据，只需关心队列是否为空。

      通常计数信号量用于下面两种事件：

1. 计数事件：在这种场合下，每当事件发生，事件处理程序将给出一个信号（信号量计数值增1），当处理事件时，处理程序会取走信号量（信号量计数值减1）。因此，计数值是事件发生的数量和事件处理的数量差值。在这种情况下，计数信号量在创建时其值为0。
2. 资源管理：这种用法下，计数值表示有效的资源数目。任务必须先获取信号量才能获取资源控制权。当计数值减为零时表示没有的资源。当任务完成后，它会返还信号量---信号量计数值增加。在这种情况下，信号量创建时，计数值等于最大资源数目。

      注：中断程序中决不可使用无“FromISR”结尾的API函数。

      注：在大部分应用场合，任务通知都可以代替计数信号量，并且速度更快、生成的代码更少。

**4.互斥量**

      互斥量是一个包含优先级继承机制的二进制信号量。用于实现同步（任务之间或者任务与中断之间）的话，二进制信号量是更好的选择，互斥量用于简单的互锁。

      用于互锁的互斥量可以充当保护资源的令牌。当一个任务希望访问某个资源时，它必须先获取令牌。当任务使用完资源后，必须还回令牌，以便其它任务可以访问同一资源。

      互斥量和信号量使用相同的API函数，因此互斥量也允许指定一个阻塞时间。阻塞时间单位为系统节拍周期时间，数目表示获取互斥量无效时最多处于阻塞状态的系统节拍周期个数。

      互斥量与二进制信号量最大的不同是：互斥量具有优先级继承机制。也就是说，如果一个互斥量（令牌）正在被一个低优先级任务使用，此时一个高优先级企图获取这个互斥量，高优先级任务会因为得不到互斥量而进入阻塞状态，正在使用互斥量的低优先级任务会临时将自己的优先级提升，提升后的优先级与与进入阻塞状态的高优先级任务相同。这个优先级提升的过程叫做优先级继承。这个机制用于确保高优先级任务进入阻塞状态的时间尽可能短，以及将已经出现的“优先级翻转”影响降低到最小。

      在很多场合中，某个硬件资源只有一个，当低优先级任务占用该资源的时候，即便高优先级任务也只能乖乖的等待低优先级任务释放资源。这里高优先级任务无法运行而低优先级任务可以运行的现象称为“优先级翻转”。

      为什么优先级继承能够降低优先级翻转的影响呢？举个例子，现在有任务A、任务B和任务C，三个任务的优先级顺序为任务C>任务B>任务A。任务A和任务C都要使用某一个硬件资源，并且当前任务A占有该资源。

      先看没有优先级继承的情况：任务C也要使用该资源，但是此时任务A正在使用这个资源，因此任务C进入阻塞，此时三个任务的优先级顺序没有发生变化。在任务C进入阻塞之后，某硬件产生了一次中断，唤醒了一个事件，该事件可以解除任务B的阻塞状态。在中断结束后，因为任务B的优先级是大于任务A的，所以任务B抢占任务A的CPU权限。那么任务C的阻塞时间就至少为：中断处理时间+任务B的运行时间+任务A的运行时间。

      再看有优先级继承的情况：任务C也要使用该资源，但是此时任务A正在使用这个资源，因此任务C进入阻塞，此时由于优先级A会继承任务C的优先级，三个任务的优先级顺序发生了变化，新的优先级顺序为：任务C=任务A>任务B。在任务C进入阻塞之后，某硬件产生了一次中断，唤醒了一个事件，该事件可以解除任务B的阻塞状态。在中断结束后，因为任务A的优先级临时被提高，大于任务B的优先级，所以任务A继续获得CPU权限。任务A完成后，处于高优先级的任务C会接管CPU。所以任务C的阻塞时间为：中断处理时间+任务A的运行时间。看，任务C的阻塞时间变小了，这就是优先级继承的优势。

      优先级继承不能解决优先级反转，只能将这种情况的影响降低到最小。硬实时系统在一开始设计时就要避免优先级反转发生。

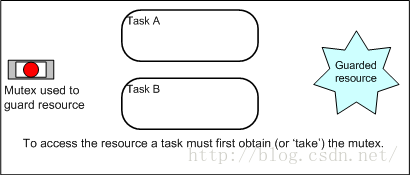


图4-1 互斥量用于保护资源

      如图4-1所示，互斥量用来保护资源。为了访问资源，任务必须先获取互斥量。任务A想获取资源，首先它使用API函数xSemaphoreTake()获取信号量，成功获取到信号量后，任务A就持有了互斥量，可以安全的访问资源。期间任务B开始执行，它也想访问资源，任务B也要先获得信号量，但是信号量此时是无效的，任务B进入阻塞状态。当任务A执行完成后，使用API函数xSemaphoreGive()释放信号量。之后任务B解除阻塞，任务B使用API函数xSemaphoreTake()获取并得到信号量，任务B可以访问资源。

**5.递归互斥量**

      已经获取递归互斥量的任务可以重复获取该递归互斥量。使用xSemaphoreTakeRecursive() 函数成功获取几次递归互斥量，就要使用xSemaphoreGiveRecursive()函数返还几次，在此之前递归互斥量都处于无效状态。比如，某个任务成功获取5次递归互斥量，那么在它没有返还5次该递归互斥量之前，这个互斥量对别的任务无效。

      递归互斥量可以看成带有优先级继承机制的信号量，获取递归互斥量的任务在用完后必须返还。

      互斥量不可以用在中断服务程序中，这是因为：

* 互斥量具有优先级继承机制，只有在任务中获取或给出互斥才有意义。
* 中断不能因为等待互斥量而阻塞。

# 20 [FreeRTOS信号量API函数](http://blog.csdn.net/zhzht19861011/article/details/50920336)

FreeRTOS的信号量包括二进制信号量、计数信号量、互斥信号量（以后简称互斥量）和递归互斥信号量（以后简称递归互斥量）。我们可以把互斥量和递归互斥量看成特殊的信号量。

      信号量API函数实际上都是宏，它使用现有的队列机制。这些宏定义在semphr.h文件中。如果使用信号量或者互斥量，需要包含semphr.h头文件。

      二进制信号量、计数信号量和互斥量信号量的创建API函数是独立的，但是获取和释放API函数都是相同的；递归互斥信号量的创建、获取和释放API函数都是独立的。

### 1创建二进制信号量

#### 1.1函数描述

       SemaphoreHandle\_t  xSemaphoreCreateBinary( void );

       这个函数用于创建一个二进制信号量。二进制信号量要么有效要么无效，这也是为什么叫做二进制的原因。

       新创建的信号量处于无效状态，这意味着使用API函数xSemaphoreTake()获取信号之前，需要先给出信号。

       二进制信号量和互斥量非常相似，但也有细微的区别：互斥量具有优先级继承机制，二进制信号量没有这个机制。这使得二进制信号量更适合用于同步（任务之间或者任务和中断之间），互斥量更适合互锁。

       一旦获得二进制信号量后不需要恢复，一个任务或中断不断的产生信号，而另一个任务不断的取走这个信号，通过这样的方式来实现同步。

       低优先级任务拥有互斥量的时候，如果另一个高优先级任务也企图获取这个信号量，则低优先级任务的优先级会被临时提高，提高到和高优先级任务相同的优先级。这意味着互斥量必须要释放，否则高优先级任务将不能获取这个互斥量，并且那个拥有互斥量的低优先级任务也永远不会被剥夺，这就是[**操作系统**](http://lib.csdn.net/base/operatingsystem)中的优先级翻转。

       互斥量和二进制信号量都是SemaphoreHandle\_t类型，并且可以用于任何具有这类参数的API函数中。

#### 1.1.2返回值

* NULL：创建信号量失败，因为FreeRTOS堆栈不足。
* 其它值：信号量创建成功。这个返回值存储着信号量句柄。

#### 1.1.3用法举例

1. SemaphoreHandle\_t xSemaphore;
3. void vATask( voidvoid \* pvParameters )
4. {
5. /\* 创建信号量 \*/
6. xSemaphore = xSemaphoreCreateBinary();
8. if( xSemaphore == NULL )
9. {
10. /\* 因堆栈不足，信号量创建失败，这里进行失败处理\*/
11. }
12. else
13. {
14. /\* 信号量可以使用。信号量句柄存储在变量xSemahore中。
15. 如果在这里调用API函数xSemahoreTake()来获取信号量，
16. 则必然是失败的，因为创建的信号量初始是无效（空）的。\*/
17. }
18. }

### 2创建计数信号量

#### 2.1函数描述

1. SemaphoreHandle\_t xSemaphoreCreateCounting ( UBaseType\_t uxMaxCount,
2. UBaseType\_t uxInitialCount )

       创建计数信号量，计数信号量通常用于以下两种情况：

1. 事件计数：在这种应用场合，每当事件发生，事件处理程序会“产生”一个信号量（信号量计数值会递增），每当处理任务处理事件，会取走一个信号量（信号量计数值会递减）。因此，事件发生或者事件被处理后，计数值是会变化的。
2. 资源管理：在这种应用场合下，计数值表示有效资源的数目。为了获得资源，任务首先要获得一个信号量---递减信号量计数值。当计数值为0时，表示没有可用的资源。当占有资源的任务完成，它会释放这个资源，相应的信号量计数值会增一。计数值达到初始值（最大值）表示所有资源都可用。

#### 2.2参数描述

* uxMaxCount：最大计数值，当信号到达这个值后，就不再增长了。
* uxInitialCount：创建信号量时的初始值。

#### 2.3返回值

       NULL表示信号量创建失败，否则返回信号量句柄。

#### 2.4用法举例

1. void vATask( voidvoid \* pvParameters )
2. {
3. xSemaphoreHandle xSemaphore;
5. // 必须先创建信号量，才能使用它
6. // 信号量可以计数的最大值为10,计数初始值为0.
7. xSemaphore = xSemaphoreCreateCounting( 10, 0 );
9. if( xSemaphore != NULL )
10. {
11. // 信号量创建成功
12. // 现在可以使用信号量了。
13. }
14. }

### 3创建互斥量

#### 3.1函数描述

       SemaphoreHandle\_t xSemaphoreCreateMutex( void )

       创建互斥量。可以使用API函数xSemaphoreTake()和xSemaphoreGive()访问互斥量，但是绝不可以用xSemaphoreTakeRecursive()和xSemaphoreGiveRecursive()访问。

       二进制信号量和互斥量非常相似，但也有细微的区别：互斥量具有优先级继承机制，二进制信号量没有这个机制。这使得二进制信号量更适合用于同步（任务之间或者任务和中断之间），互斥量更适合互锁。

       一旦获得二进制信号量后不需要恢复，一个任务或中断不断的产生信号，而另一个任务不断的取走这个信号，通过这样的方式来实现同步。

       低优先级任务拥有互斥量的时候，如果另一个高优先级任务也企图获取这个信号量，则低优先级任务的优先级会被临时提高，提高到和高优先级任务相同的优先级。这意味着互斥量必须要释放，否则高优先级任务将不能获取这个互斥量，并且那个拥有互斥量的低优先级任务也永远不会被剥夺，这就是操作系统中的优先级翻转。

       互斥量和二进制信号量都是SemaphoreHandle\_t类型，并且可以用于任何具有这类参数的API函数中。

#### 3.2返回值

       NULL表示信号量创建失败，否则返回信号量句柄。

#### 3.3用法举例

1. xSemaphoreHandle xSemaphore;
3. voidvATask( voidvoid \* pvParameters )
4. {
5. // 互斥量在未创建之前是不可用的
6. xSemaphore = xSemaphoreCreateMutex();
7. if( xSemaphore != NULL )
8. {
9. // 创建成功
10. // 在这里可以使用这个互斥量了
11. }
12. }

### 4创建递归互斥量

#### 4.1函数描述

       SemaphoreHandle\_t xSemaphoreCreateRecursiveMutex( void )

       用于创建递归互斥量。被创建的互斥量可以被API函数xSemaphoreTakeRecursive()和xSemaphoreGiveRecursive()使用，但不可以被API函数xSemaphoreTake()和xSemaphoreGive()使用。

       递归类型的互斥量可以被拥有者重复获取。拥有互斥量的任务必须调用API函数xSemaphoreGiveRecursive()将拥有的递归互斥量全部释放后，该信号量才真正被释放。比如，一个任务成功获取同一个互斥量5次，那么这个任务要将这个互斥量释放5次之后，其它任务才能获取到它。

       递归互斥量具有优先级继承机制，因此任务获得一次信号后必须在使用完后做一个释放操作。

       互斥量类型信号不可以用在中断服务例程中。

#### 4.2返回值

       NULL表示互斥量创建失败，否则返回互斥量句柄。

#### 4.3用法举例

1. xSemaphoreHandle xMutex;
3. void vATask( voidvoid \* pvParameters )
4. {
5. // 互斥量未创建前是不能被使用的
6. xMutex = xSemaphoreCreateRecursiveMutex();
8. if( xMutex != NULL )
9. {
10. // 创建成功
11. // 在这里创建互斥量
12. }
13. }

### 5删除信号量

#### 5.1函数描述

       void vSemaphoreDelete( SemaphoreHandle\_t xSemaphore );

       删除信号量。如果有任务阻塞在这个信号量上，则这个信号量不要删除。

#### 5.2参数描述

       xSemaphore：信号量句柄

### 6获取信号量

#### 6.1函数描述

       xSemaphoreTake(SemaphoreHandle\_t xSemaphore, TickType\_t xTicksToWait)

       获取信号量。信号量必须是通过API函数xSemaphoreCreateBinary()、xSemaphoreCreateCounting()和xSemaphoreCreateMutex()预先创建过的。注意，递归互斥量类型信号量不能使用该函数、不用在中断服务程序中使用该函数。

#### 6.2参数描述

* xSemaphore：信号量句柄
* xTickToWait：信号量无效时，任务最多等待的时间，单位是系统节拍周期个数。使用宏portTICK\_PERIOD\_MS可以辅助将系统节拍个数转化为实际时间（以毫秒为单位）。如果设置为0，表示不是设置等待时间。如果INCLUDE\_vTaskSuspend设置为1，并且参数xTickToWait为portMAX\_DELAY则可以无限等待。

#### 6.3返回值

       成功获取到信号量返回pdTRUE，否则返回pdFALSE。

#### 6.4用法举例

1. SemaphoreHandle\_t xSemaphore = NULL;
3. /\*这个任务创建信号量 \*/
4. void vATask( voidvoid \* pvParameters )
5. {
6. /\*创建互斥型信号量，用于保护共享资源。\*/
7. xSemaphore = xSemaphoreCreateMutex();
8. }
10. /\* 这个任务使用信号量 \*/
11. void vAnotherTask( voidvoid \* pvParameters )
12. {
13. /\* ... 做其它事情. \*/
15. if( xSemaphore != NULL )
16. {
17. /\*如果信号量无效，则最多等待10个系统节拍周期。\*/
18. if( xSemaphoreTake( xSemaphore, ( TickType\_t ) 10 ) == pdTRUE )
19. {
20. /\*到这里我们获取到信号量，现在可以访问共享资源了\*/
22. /\* ... \*/
24. /\* 完成访问共享资源后，必须释放信号量\*/
25. xSemaphoreGive( xSemaphore );
26. }
27. else
28. {
29. /\* 没有获取到信号量，这里处理异常情况。\*/
30. }
31. }
32. }

### 7获取信号量（带中断保护）

#### 7.1函数描述

1. xSemaphoreTakeFromISR(SemaphoreHandle\_t xSemaphore,
2. signedBaseType\_t \*pxHigherPriorityTaskWoken)

       API函数xSemaphoreTake()的另一版本，用于中断服务程序。

#### 7.2参数描述

* xSemaphore：信号量句柄
* pxHigherPriorityTaskWoken：如果\*pxHigherPriorityTaskWoken为pdTRUE，则需要在中断退出前手动进行一次上下文切换。从FreeRTOS V7.3.0开始，该参数为可选参数，并可以设置为NULL。

#### 7.3返回值

       信号量成功获取返回pdTRUE，否则返回pdFALSE。

### 8获取递归互斥量

#### 8.1函数描述

       xSemaphoreTakeRecursive(SemaphoreHandle\_t xMutex, TickType\_t xTicksToWait );

       获取递归互斥信号量。互斥量必须是通过API函数xSemaphoreCreateRecursiveMutex()创建的类型。

       文件FreeRTOSConfig.h中的宏configUSE\_RECURSIVE\_MUTEXES必须设置成1，此函数才有效。

      已经获取递归互斥量的任务可以重复获取该递归互斥量。使用xSemaphoreTakeRecursive() 函数成功获取几次递归互斥量，就要使用xSemaphoreGiveRecursive()函数返还几次，在此之前递归互斥量都处于无效状态。比如，某个任务成功获取5次递归互斥量，那么在它没有返还5次该递归互斥量之前，这个互斥量对别的任务无效。

#### 8.2参数描述

* xMutex：互斥量句柄，必须是使用API函数xSemaphoreCreateRecursiveMutex()返回的。
* xTickToWait：互斥量无效时，任务最多等待的时间，单位是系统节拍周期个数。使用宏portTICK\_PERIOD\_MS可以辅助将系统节拍个数转化为实际时间（以毫秒为单位）。如果设置为0，表示不是设置等待时间。如果任务已经拥有信号量则xSemaphoreTakeRecursive()立即返回，不管xTickToWait是什么值。

#### 8.3返回值

     成功获取递归互斥量返回pdTURE，否则返回pdFALSE。

#### 8.4用法举例

1. SemaphoreHandle\_t xMutex = NULL;
3. // 这个任务创建互斥量
4. void vATask( voidvoid \*pvParameters )
5. {
6. // 这个互斥量用于保护共享资源
7. xMutex =xSemaphoreCreateRecursiveMutex();
8. }
10. //这个任务使用互斥量
11. void vAnotherTask( voidvoid \*pvParameters )
12. {
13. // ... 做其它事情.
15. if( xMutex != NULL )
16. {
17. // 如果互斥量无效，则最多等待10系统时钟节拍周期.
18. if(xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 ) == pdTRUE )
19. {
20. // 到这里我们成功获取互斥量并可以访问共享资源了
22. // ...
23. // 由于某种原因，某些代码需要在一个任务中多次调用API函数
24. // xSemaphoreTakeRecursive()。当然不会像本例中这样连续式
25. //调用，实际代码会有更加复杂的结构
26. xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 );
27. xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 );
29. // 我们获取一个互斥量三次，所以我们要将这个互斥量释放三次
30. //它才会变得有效。再一次说明，实际代码可能会更加复杂。
31. xSemaphoreGiveRecursive( xMutex );
32. xSemaphoreGiveRecursive( xMutex );
33. xSemaphoreGiveRecursive( xMutex );
35. // 到这里，这个共享资源可以被其它任务使用了.
36. }
37. else
38. {
39. // 处理异常情况
40. }
41. }
42. }

### 9释放信号量

#### 9.1函数描述

       xSemaphoreGive(SemaphoreHandle\_t xSemaphore )

       用于释放一个信号量。信号量必须是API函数xSemaphoreCreateBinary()、xSemaphoreCreateCounting()或xSemaphoreCreateMutex() 创建的。必须使用API函数xSemaphoreTake()获取这个信号量。

       这个函数绝不可以在中断服务例程中使用，可以使用带中断保护版本的API函数xSemaphoreGiveFromISR()来实现相同功能。

       这个函数不能用于使用API函数xSemaphoreCreateRecursiveMutex()所创建的递归互斥量。

#### 9.2参数描述

* xSemaphore：信号量句柄。

#### 9.3返回值

       信号量释放成功返回pdTRUE，否则返回pdFALSE。

#### 9.4用法举例

1. SemaphoreHandle\_t xSemaphore = NULL;
3. voidvATask( voidvoid \* pvParameters )
4. {
5. // 创建一个互斥量，用来保护共享资源
6. xSemaphore = xSemaphoreCreateMutex();
8. if( xSemaphore != NULL )
9. {
10. if( xSemaphoreGive( xSemaphore ) != pdTRUE )
11. {
12. //我们希望这个函数调用失败，因为首先要获取互斥量
13. }
14. // 获取信号量，不等待
15. if( xSemaphoreTake( xSemaphore, ( TickType\_t ) 0 ) )
16. {
17. // 现在我们拥有互斥量，可以安全的访问共享资源
18. if( xSemaphoreGive( xSemaphore ) != pdTRUE )
19. {
20. //我们不希望这个函数调用失败，因为我们必须
21. //要释放已获取的互斥量
22. }
23. }
24. }
25. }

### 10释放信号量（带中断保护）

#### 10.1函数描述

1. xSemaphoreGiveFromISR(SemaphoreHandle\_t xSemaphore,
2. signed BaseType\_t \*pxHigherPriorityTaskWoken )

       释放信号量。是API函数xSemaphoreGive()的另个版本，用于中断服务程序。信号量必须是通过API函数xSemaphoreCreateBinary()或xSemaphoreCreateCounting()创建的。这里没有互斥量，是因为互斥量不可以用在中断服务程序中。

#### 10.2参数描述

       xSemaphore：信号量句柄

       pxHigherPriorityTaskWoken：如果\*pxHigherPriorityTaskWoken为pdTRUE，则需要在中断退出前人为的经行一次上下文切换。从FreeRTOS V7.3.0开始，该参数为可选参数，并可以设置为NULL。

#### 10.3返回值

       成功释放信号量返回pdTURE，否则返回errQUEUE\_FULL。

#### 10.4用法举例

1. #define LONG\_TIME 0xffff
2. #define TICKS\_TO\_WAIT    10
4. SemaphoreHandle\_t xSemaphore = NULL;
6. /\* Repetitive task. \*/
7. void vATask( voidvoid \* pvParameters )
8. {
9. /\* 我们使用信号量同步,所以先创建一个二进制信号量.必须确保
10. 在创建这个二进制信号量之前，中断不会访问它。\*/
11. xSemaphore = xSemaphoreCreateBinary();
13. for( ;; )
14. {
15. /\* 我们希望每产生10次定时器中断，任务运行一次。\*/
16. if( xSemaphoreTake( xSemaphore, LONG\_TIME ) == pdTRUE )
17. {
18. /\* 到这里成功获取到信号量\*/
20. ...
22. /\* 我们成功执行完一次，由于这是个死循环，所以任务仍会
23. 阻塞在等待信号量上。信号量由ISR释放。\*/
24. }
25. }
26. }
28. /\* 定时器 ISR \*/
29. void vTimerISR( voidvoid \* pvParameters )
30. {
31. static unsigned char ucLocalTickCount = 0;
32. static signed BaseType\_txHigherPriorityTaskWoken;
34. /\*定时器中断发生 \*/
36. ...执行其它代码
38. /\*需要vATask() 运行吗? \*/
39. xHigherPriorityTaskWoken = pdFALSE;
40. ucLocalTickCount++;
41. if( ucLocalTickCount >= TICKS\_TO\_WAIT )
42. {
43. /\* 释放信号量，解除vATask任务阻塞状态 \*/
44. xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );
46. /\* 复位计数器 \*/
47. ucLocalTickCount = 0;
48. }
50. /\* 如果 xHigherPriorityTaskWoken 表达式为真，需要执行一次上下文切换\*/
51. portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );
52. }

### 11释放递归互斥量

#### 11.1函数描述

       xSemaphoreGiveRecursive(SemaphoreHandle\_t xMutex )

       释放一个递归互斥量。互斥量必须是使用 API函数xSemaphoreCreateRecursiveMutex()创建的。文件FreeRTOSConfig.h中宏configUSE\_RECURSIVE\_MUTEXES必须设置成1本函数才有效。

#### 11.2参数描述

* xMutex：互斥量句柄。必须是函数xSemaphoreCreateRecursiveMutex()返回的值。

#### 11.3返回值

       如果递归互斥量释放成功，返回pdTRUE。

#### 11.4用法举例

       见“8 获取递归互斥量”。

### 12获取互斥量持有任务的句柄

#### 12.1函数描述

       TaskHandle\_t xSemaphoreGetMutexHolder( SemaphoreHandle\_t xMutex );

       返回互斥量持有任务的句柄（如果有的话），互斥量由参数xMutex指定。

       如果调用此函数的任务持有互斥量，那么可以可靠的返回任务句柄，但是如果是别的任务持有互斥量，则不总可靠。

       文件FreeRTOSConfig.h中宏configUSE\_MUTEXES必须设置成1本函数才有效。

#### 12.2参数描述

* xMutex：互斥量句柄

#### 12.3返回值

       返回互斥量持有任务的句柄。如果参数xMutex不是互斥类型信号量或者虽然互斥量有效但这个互斥量不被任何任务持有则返回NULL。

      这是FreeRTOS基础篇的最后一篇博文，到这里我们已经可以移植、熟练使用FreeRTOS了。但如果想知道FreeRTOS背后的运行机制，这些是远远不够的，下面要走的路还会很长。要不要了解FreeRTOS背后运行机制，全凭各位的兴趣，毕竟我们即使不清楚汽车的构造细节，但只要掌握驾驶技巧也可以很好的开车的。使用RTOS也与之相似，只要我们掌握了基础篇的那些知识，我们已经可以很好的使用FreeRTOS了。探索FreeRTOS背后运行的机制，是我们对未知事件的好奇，也是我们相信理解了FreeRTOS运行机制，可以让我们更优雅、更少犯错、更举重若轻的的使用RTOS。

      FreeRTOS高级篇已经开始写了，可以[点击这里](http://blog.csdn.net/zhzht19861011/article/category/6191478)擦看最新的文章列表。