µC/OS-III™

The Real-Time Kernel

User's Manual

屈环宇 译

Micriμm
Press
Weston, FL 33326

译者序

很高兴终于完成了 uC/OS-III 嵌入式系统的翻译，翻译从 2011 年 10 月 15 日开始到 2011 年 11 月 3 日为止，共 20 天时间，平均每天 5 个小时。本想将 uC/OS-III 函数的 API 部分也翻译的，但毕竟考研更甚于爱好，我得为 2013 年 1 月的考研做准备呀~~。

在此，我要感谢：

1、我的导师：乐光学教授。是您经常带我去公司拓展视野，并让我坚定不移地往嵌入式方面发展。

2、我的师傅：张雪强博士。是您无偿提供给我一些开发板，作为回报，帮您的店铺宣传一下 http://as-robot.taobao.com/

3、还有我的亲朋好友们。

特别是今天上午，用了 3 小时终于在 MDK4.0 编译器上将 uC/OS-III 移植到 stm32f103rb 处理器上，并成功运行。移植的步骤，我也会发布到网上的，谢谢大家。

我的 QQ 号码是 522430192，我的邮箱是 522430192@qq.com，希望大家多多联系我，共同学习，共同进步。

：屈环宇

：嘉兴学院

：2011 年 11 月 3 日晚

序言

什么是 uC/OS-III?

uC/OS-III(Micro C OS Three 微型的 C 语言编写的操作系统第 3 版)是一个可升级的，可固化的，基于优先级的实时内核。它对任务的个数无限制。uC/OS-III 是一个第 3 代的系统内核，支持现代的实时内核所期待的大部分功能。例如资源管理，同步，任务间的通信等等。然而，uC/OS-III 提供的特色功能在其它的实时内核中是找不到的，比如说完备的运行时间测量性能，直接地发送信号或者消息到任务，任务可以同时等待多个内核对象等。

为什么命名一个新的版本？

uC/OS 系列，第一代产生于 1992。经过了多年的使用和上千人的反馈，已经产生了很多的进化版本。

uC/OS-III 是这些反馈和经验的总结。在 uC/OS-II 中很少使用的功能已经被删除或者被更新，添加了更高效的功能和服务。其中最有用的功能应该是时间片轮转法（round robin),这个是 uC/OS-II 中不支持的，但是现在已经是 uC/OS-III 的一个功能了。

uC/OS-III 会提供新的功能以更好地适应新出现的处理器。特别的，uC/OS-III 被设计用于 32 位处理器，但是它也能在 16 位或 8 位处理器中很好地工作。

uC/OS-III 的目标

uC/OS-III 最主要的目标是提供一流的实时内核以适应更新很快的嵌入式产品。使用像 uC/OS-III 那样具有雄厚的基础和稳定的框架的商业实时内核，能够帮助设计师们处理日益复杂的嵌入式设计。

这本书中的目标，是为了介绍 uC/OS-III 的内部工作。了解这些会帮助读者实现逻辑上的设计方案，协调统一硬件和软件会让你对整体的设计很有把握。

1、简介

在重要的地方，实时系统凭借其系统性的计算和及时的处理能力工作着。一共有 2 种类型的实时系统：软实时系统和硬实时系统。

软实时系统和硬实时系统的区别在于一旦没有在规定的时间内完成任务所导致后果的严重性。超过时限后所得到的结果即使正确也可能是毫无作用的。

硬实时系统中，运算超时是不允许发生的。在很多情况下，超时会导致巨大的灾难，会威胁人们的生命安全。但是在软实时系统中，超时不会导致严重后果。

实时系统的应用范围很广，但很多实时系统是嵌入式的。一个嵌入式系统是计算机中添加操作系统，但是用户不公认这是个计算机。以下列出嵌入式系统的一些例子

航空航天	**通讯**	**加工控制**
飞行管理系统	路由器	化学工厂
喷射发动机控制	交换机	工厂自动化
武器系统	手机	食品加工
语音	**计算机外围设备**	**机器人**
MP3 播放器	打印机	

放大器和调谐器	扫描仪	
汽车制造业 反锁死制动系统 气候控制 引擎控制 GPS	**家用电器** 空气调节机 恒温器 大型家用电器	**视频** 广播设备 高清电视
办公室自动化 传真机 复印机	**等等**	

实时系统的设计，调试和配置比非实时系统难得多。

1-1 前后台系统

简单的小型系统设计一般是基于前后台的或者无限循环的系统。包含一个无限循环的模块实现需要的操作（后台）。中断处理程序实现异步事件（前台）。前台也叫做中断级，后台也叫作任务级。

临界操作应该在任务级中被执行，不可避免地必须在中断处理程序中执行也要确保是在很短的时间内完成。因为这会导致 ISR 占用更长的时间。通常的，ISR 中使能相关的信息而在后台程序中执行相应的操作。这叫做任务级响应。任务级响应的时间依赖于后台循环一次所需的时间，通常这不是一个固定常量。另外，如果其中的代码稍有改动，那么循环一次所用的时间也将有所变化。

大多数高产量低成本微控制器的应用软件（例如微波炉，电话玩具等）都是基于前后台系统的。

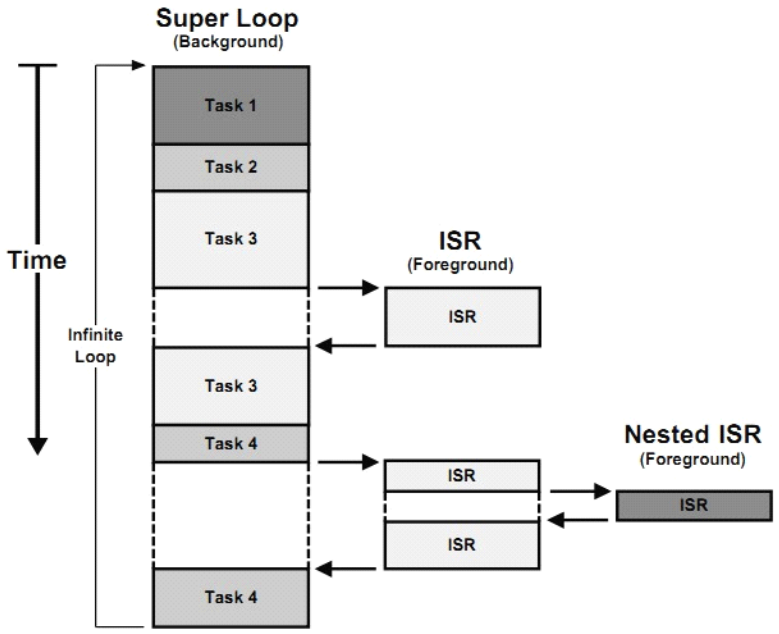


Figure 1-1 **Foreground/Background (SuperLoops) systems**

1-2 实时内核

实时内核是一个能管理 MPU、MCU、DSP 时间和资源的软件。

实时内核的应用包括迅速地响应，可靠地完成工作的各个部分。任务（也叫做线程）是一段简单的程序，运行时完全地占用 CPU。在单 CPU 中，任何时候只有 1 个任务被执行。

内核的责任是管理任务，也做多任务处理。多任务处理的作用是协调和切换多个任务依次享用 CPU。多任务处理最大化 CPU 的功能

同时会让我们感觉是多个 CPU 在同时运行。多任务处理也有利于处理模块化的应用。多任务处理一个最重要的方面在于它允许程序员管理复杂的实时应用。在多任务处理中程序员可以简单的维护和升级产品。

uC/OS-III 是一个抢占式内核，这意味着 uC/OS-III 总是执行最重要的就绪任务，如图 1-2。

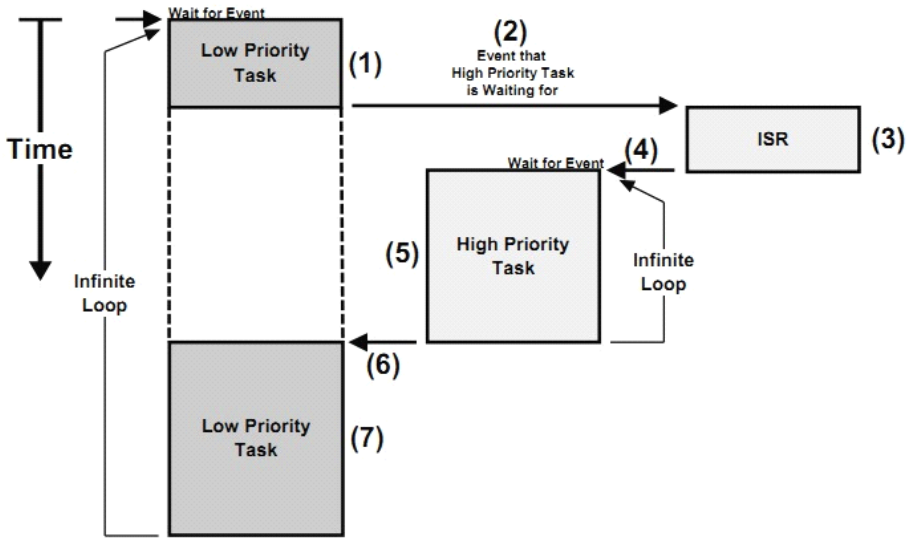


Figure 1-2 **µC/OS-III is a preemptive kernel**

F1-2（1） 一个低优先级的任务正在被执行

F1-2（2）发生一个中断，CPU 转向 ISR

F1-2（3）ISR 响应中断请求设备，但是 ISR 只做非常少的工作。ISR 应该标记或发送消息到一个高优先级的任务，让中断能够快速处理完毕。例如，如果一个中断来自于以太网控制器，ISR 标记任务，在任务级响应以太网控制器。

F1-2（4）当 ISR 执行完毕，uC/OS-III 注意到 ISR 创建的一个更高优先级的任务就绪。uC/OS-III 将不会返回到中断前的任务，它会切换

到这个更高优先级的任务。

F1-2（5）高优先级任务执行必要的处理答复中断请求设备。

F1-2（6）当高优先级任务完成时，返回原任务中断前的代码。

F1-2（7）原任务在它被中断的地方开始执行。

uC/OS-III 内核也负责管理任务间的交流，系统的资源（内存和 I/O）。

系统中加入内核需要额外的支出，因为内核提供服务时需要时间去处理。大多数的额外支出取决于服务的调用频繁度。在一个优秀的设计中，内核占用 CPU 的时间介于 2%到 4%之间。因为 uC/OS-III 是一个软件，添加到目标系统中需要额外的 ROM 和 RAM。

低档的单片机很有可能不支持像 uC/OS-III 那样的实时内核，因为它只有很少的 RAM 可供访问。uC/OS-III 内核需要 1K 到 4K 之间的 RAM，加上每个任务自己所需的堆栈空间。至少有 4K 大小 RAM 的处理器才有可能成功移植 uC/OS-III。

最后，为了更好地使用 CPU，uC/OS-III 提供了大约 70 种常用的服务。当用过像 uC/OS-III 那样的具有实时内核的系统后，你将不会再去使用前后台系统了。

1-3 实时系统（RTOS）

一个实时系统通常包括一个实时内核以及其他高级的服务，例如：文件管理，堆栈协议，图形用户接口等等。大多数服务都是跟 I/O 有

关的。

Micrium 提供了 RTOS 一套完整的组件，包括 uC/FS、uC/TCP-IP、uC/GUI、uC/USB 等。这些组件大部分都可以单独执除了 uC/TCP-IP。实时内核在应用中不是必须的。事实上，用户可以单独选择您的应用所需用的组件。详情和购买请联系 Micrium(www.Micrium.com)。

1-4 uC/OS-III

uC/OS-III 是一个可扩展的，可固化的，抢占式的实时内核，它管理的任务个数不受限制。它是第三代内核，提供了现代实时内核所期望的所有功能包括资源管理、同步、内部任务交流等。uC/OS-III 也提供了很多特性是在其他实时内核中所没有的。比如能在运行时测量运行性能，直接得发送信号或消息给任务，任务能同时等待多个信号量和消息队列。

以下列出 uC/OS-III 的特点：

源代码：uC/OS-III 完全根据 ANSI-C 标准写的。代码的规范是 Micrium 团队的一种文化。虽然很多商业内核供应商提供他们产品的源代码，但是这些产品很有可能是笨重且难以利用的。除非代码严格地遵循标准并且产品有完整的带例子的说明书以展示代码是怎样工作的。通过这本书，你将会对 uC/OS-III 内部的工作情况有一个很深的了解。

应用程序接口（API）：uC/OS-III 是很直观的。如果你熟悉类似的编码规范，你能轻松地知道函数名所对应的服务，以及需要怎样的参

数。例如：指向对象的指针通常是第一个参数，指向错误代码的指针通常是最后一个参数。

抢占式多任务处理：uC/OS-III 是一个抢占式多任务处理内核，因此，uC/OS-III 正在运行的经常是最重要的就绪任务。

时间片轮转调度：uC/OS-III 允许多个任务拥有相同的优先级。当多个相同优先级的任务就绪时，并且这个优先级是目前最高的。uC/OS-III 会分配用户定义的时间片给每个任务去运行。每个任务可以定义不同的时间片。当任务用不完时间片时可以让出 CPU 给另一个任务。

快速响应中断：uC/OS-III 有一些内部的数据结构和变量。uC/OS-III 保护临界段可以通过锁定调度器代替关中断。因此关中断的时间会非常少。这样就使 uC/OS-III 可以响应一些非常快的中断源了。

确定性的：uC/OS-III 的中断响应时间是可确定的，uC/OS-III 提供的大部分服务的执行时间也是可确定的。

可扩展的：根据应用的需求，代码大小可以被调整。编译时通过调整 uC/OS-III 源代码中的大约 40 个#define(见 OS_CFG.H)可以在添加或移除一些功能。uC/OS-III 的服务还提供一些实时检查功能。特别的，uC/OS-III 能检传递的参数是否为 NULL 指针，ISR 是否就绪了任务级服务。参数有允许范围，指定选项都是有用的。检测功能可以被关闭（在编译时）以提供更好的性能和缩减代码大小。实际上，

可扩展的 uC/OS-III 支持更广泛的应用和项目。

易移植的：uC/OS-III 可以被移植到大部分的 CPU 架构中。大部分的支持 uC/OS-II 的器件通过改动就能支持 uC/OS-III。而 uC/OS-II 已经移植到 45 种 CPU 架构中了。

可固化的：uC/OS-III 专为嵌入式系统设计，它可以跟应用程序代码一起被固化。

可实时配置的：uC/OS-III 允许用户在运行时配置内核。特别的，所有的内核对象如任务、堆栈、信号量、事件标志组、消息队列、消息、互斥信号量、内存分区、软件定时器等都是在运行时分配的,以免在编译时的过度分配。

任务数无限制：uC/OS-III 对任务数量无限制。实际上，任务的数量限制于处理器能提供的内存大小。每一个任务需要有自己的堆栈空间，uC/OS-III 在运行时监控任务堆栈的生长。uC/OS-III 对任务的大小无限制，

优先级数无限制：uC/OS-III 对优先级的数量无限制。然而，配置 uC/OS-III 的优先级在 32 到 256 之间已经满足大多数的应用了。

内核对象数无限制：uC/OS-III 支持任何数量的任务、信号量、互斥信号量、事件标志组、消息队列、软件定时器、内存分区。用户在运行时分配所有的内核对象。

服务：uC/OS-III 提供了高档实时内核所需要的所有功能，例如任务管理、时间管理、信号量、事件标志组、互斥信号量、消息队列、

软件定时器、内存分区等。

互斥信号量（Mutexes）：互斥信号量用于资源管理。它是一个内置优先级的特殊类型信号量，用于消除优先级反转。互斥信号量可以被嵌套，因此，任务可申请同一个互斥信号量多达 250 次。当然，互斥信号量的占有者需要释放同等次数。

嵌套的任务停止：uC/OS-III 允许任务停止自身或者停止另外的任务。停止一个任务意味着这个任务将不再执行直到被其他的任务恢复。停止可以被嵌套到 250 级。换句话说，一个任务可以停止另外的任务多达 250 次。当然，这个任务必须被恢复同等次数才有资格再次获得 CPU。

软件定时器：可以定义任意数量的一次性的、周期性的、或者两者兼有的定时器。定时器是倒计时的，执行用户定义的行为一直到计数减为 0。每一个定时器可以有自己的行为，如果一个定时器是周期性的，计数减为 0 时会自动重装计数值并执行用户定义的行为。

挂起多个对象：uC/OS-III 允许任务等待多个事件的发生。特别的，任务可以同时等待多个信号量和消息队列被提交。等待中的任务在事件发生的时候被唤醒。

任务信号量：uC/OS-III 允许 ISR 或者任务直接地发送信号量给其它任务。这样就避免了必须产生一个中间级内核对象如一个信号量或者事件标志组只为了标记一个任务。提高了内核性能。

任务消息：uC/OS-III 允许 ISR 或者任务直接发送消息到另一个任

务。这样就避免产生一个消息队列，提高了内核性能。

任务寄存器：每一个任务可以拥有用户可定义的任务寄存器，不同于 CPU 寄存器。

错误检测：uC/OS-III 能检测指针是否为 NULL、在 ISR 中调用的任务级服务是否允许、参数在允许范围内、配置选项的有效性、函数的执行结果等。每一个 uC/OS-III 的 API 函数返回一个对应于函数调用结果的错误代号。

内置的性能测量：uC/OS-III 有内置性能测量功能。能测量每一个任务的执行时间,每个任务的堆栈使用情况，任务的执行次数，CPU 的使用情况，ISR 到任务的切换时间,任务到任务的切换时间，列表中的峰值数，关中断、锁调度器平均时间等。

可优化：uC/OS-III 被设计于能够根据 CPU 的架构被优化。uC/OS-III 所用的大部分数据类型能够被改变，以更好地适应 CPU 固有的字大小。优先级调度法则可以通过编写一些汇编语言而获益于一些特殊的指令如位设置、位清除、计数清零指令（CLZ）,find-first-one(FF1)指令。

死锁预防：uC/OS-III 中所有的挂起服务都可以有时间限制，预防死锁。

任务级的时基处理：uC/OS-III 有时基任务，时基 ISR 触发时基任务。uC/OS-III 使用了哈希列表结构，可以大大减少处理延时和任务超时所产生的开支。

用户可定义的钩子函数：uC/OS-III 允许程序员定义 hook 函数，hook 函数被 uC/OS-III 调用。hook 函数允许用户扩展 uC/OS-III 的功能。有的 hook 函数在任务切换的时候被调用，有的在任务创建的时候被调用，有的在任务删除的时候被调用。

时间戳：为了测量时间，uC/OS-III 需要一个 16 位或者 32 位的时时间戳计数器。这个计数器值可以在运行时被读取以测量时间。例如：当 ISR 提交消息到任务时，时间戳计数器自动读取并保存作为消息。当接收者接收到这条消息，时间戳被提供在消息内。通过读取现在的时间戳，消息的响应时间可以被确定。

嵌入的内核调试器：这个功能允许内核调试器查看 uC/OS-III 的变量和数据结构通过一个用户定义的通道。（但是只能在调试器遇到断点的时候查看）。uC/OS-III 内核也支持 uC/Probe（探针）在运行时显示信息。

对象名称：每个 uC/OS-III 的内核对象有一个相关联的名字。这样就能很容易的识别出对象所指定的作用。分配一个 ASCII 码的名字给任务、信号量、互斥信号量、事件标志组、消息队列、内存块、软件定时器。对象的名字长度没有限制，但是必须以空字符结束。

1-5 uC/OS,uC/OS-II,uC/OS-III 的性能对比

表 1.1 列出了 uC/OS 的演变，比较每个版本的区别。

功能	**uC/OS**	**uC/OS-II**	**uC/OS-III**
诞生年份	1992	1998	2009
书	有	有	有
提供源代码	是	是	是
抢占式多任务	是	是	是
最大任务数	64	256	无限制
每个优先级的任务数	1	1	无限制
时间片轮转	否	否	是
信号量	是	是	是
互斥信号量	否	是	是（可嵌套的）
事件标志	否	是	是
消息邮箱	是	是	否（不需要了）
消息队列	是	是	是
固定大小的内存管理	否	是	是

不通过信号量标记一个任务	否	否	是
不通过消息队列发消息给任务	否	否	是
软件定时器	否	是	是
任务停止/恢复	否	是	是（可嵌套的）
死锁预防	是	是	是
可扩展的	是	是	是
代码段需求	3K 到 8K	6K 到 26K	6K 到 20K
数据段需求	1K+	1K+	1K+
可固化	是	是	是
在运行时配置	否	否	是
编译时配置	是	是	是
每个对象命名	否	是	是
挂起多个对象	否	是	是
任务寄存器	否	是	是
嵌入的测量功能	否	有限制	大量的
用户可定义的 hook 函数	否	是	是
时间戳	否	否	是
嵌入的内核调试	否	是	是
汇编可优化	否	否	是

任务级的时基定时器处理	否	否	是
提供的服务	~20	~90	~70
MISRA-C:1998	否	是（除了 10 个规则）	N/A
MISRA-C:2004	否	否	是（除了 7 个规则）
DO178B EUROCAE ED-12B	否	是	申请中
FDA 认证	否	是	申请中
SIL3/SIL4 IEC	否	是	申请中
IEC-61508	否	是	申请中

表 1-1

1-6 这本书的编排方式

这本书中可分为 2 部分

第一部分只是讲解了 uC/OS-III，并没有涉及到于 CPU 架构有关的内容。在这之中，读者可以学到实时内核的相关知识。例如，临界段代码、任务管理、就绪列表、任务调度、上下文切换、中断管理、阻塞列表、时间管理、软件定时器、资源管理、同步、内存管理、如何使用 uC/OS-III 的 API、如何配置 uC/OS-III、如何移植 uC/OS-III 到不同架构的 CPU 等。

第二部分讲解了移植 uC/OS-III 到一个流行的 CPU 架构中，这这里，你可以学到 CPU 的架构知识及 uC/OS-III 是如何最大限度地利用 CPU 的。例子都是在评估版中实际运行的。

正如我刚才提到的，这本书假定你已经有一个评估板。CD/DVD 中一系列免费的工具（或者从网上下载）与这本书和评估板是相互补充的。只要在工程中没有商业的意图，使用评估板时辅助工具和 uC/OS-III 是免费的（用于教育时也是免费的）。换句话说，没有附加的要价除了最初的书本费，评估板费以及辅助工具费。

这本书伴随着介绍了 Micrium 提供的测试工具 uC/Probe。这个测试工具允许用户监控和改变目标系统中 5 个变量。

1-7 探针 uC/Probe

uC/Probe 是一个基于应用的微型窗口，它允许用户在运行时查看目标系统中的变量。特别的，当目标系统在运行时可以显示或者改变任何变量。显示这些变量可以通过图表元件如仪表盘、电平表、柱状图、LEDs、数字指示器等等。滑动器、开关、按钮可以被用来改变变量的值。

uC/Probe 可以被连接到任何器件（8 位，16 位，32 位，64 位，DSP）通过 J-Tag、RS-232C、USB 或者以太网接口等。uC/Probe 可以显示或修改应用中任何变量（只要它是全局的），包括 uC/OS-III 的内部变量。

uC/Probe 配合任何编译器、汇编器、链接器产生一个 ELF/DWARF 或者 IEEE695 文件。用户需要把这个文件下载到评估板或者目标器件。从这个文件中，uC/Probe 可以提取出变量的信息，以及确定变量在 RAM 或者 ROM 中的存储位置。

uC/Probe 允许用户收集并记录数据到一个文件以便于最后分析。uC/Probe 是 uC/OS-III 中嵌入的一个功能。

这个测试版本只允许用户显示或者改变最多 5 个变量。

uC/Probe 是一个工具，它是嵌入式软件工程师所必须的。当你购买了 uC/OS-III，才会提供完整的 uC/Probe。详情请登录 www.micrium.com。

1-8 规定

首先，如果有图解的话先看图解，或者小括号中的解释。图片中都会标注“F”以及图的编号。例如，F3-4（2）说明相关介绍在图 F3-4 中的元素（2）中。这个规定也适用于列表“L”和表格“T”。

其次，新的章节开始于新的页。换句话说，不要奇怪于叙述到页的一半结束了。新的章节开始于下一页，这是为了防止内容被页面所打断。

第三，在我的职业生涯中非常注重于代码的质量。在 Micrium 公司，令我们感到骄傲的是我们的代码都是非常高效的，可以参考这本书的例子。在 1992 年的 uC/OS 的书中公布了一些我创造的代码标准。这些标准经过了多年的进化，但其核心思想还是一直保持着。这些标准可以从 www.micrium.com 下载。

第四，所有的函数、变量、宏、#define 常量都以“OS”为前缀（操作系统的标准）,接下来的是组件的缩写，然后是函数的功能。例如 OSSemPost()表明函数属于 OS，是信号量服务的一部分，函数的功能是提交信号量。所有相关的函数在参考手册中成组说明，以便让用户使用起来更方便。

发送信号量或消息到任务叫做提交 Post，等待信号量或者消息叫做 Pend。换句话说，ISR 或者任务要标记或者发送消息到另一个任务时需调用 OS???Post()。其中???是服务类型：Sem，TaskSem,Flag，Mutex，Q，TaskQ。类似的，一个任务等待信号量或消息的时候需调用 OS???Pend()。

1-9 章节目录

图 1-3 显示了 uC/OS-III 的章节布局。这个图表对于理解章节间的关系是很有帮助的。第一列介绍的是 uC/OS-III 的结构。第二列介绍的是 uC/OS-III 的一些扩展功能。第三列将会帮助你移植 uC/OS-III 到不同架构的 CPU 中。第四列的顶部介绍的是怎样从 uC/OS-III 中获得实时的编译时间，运行时间。当调试时使用内核感知功能或 uC/Probe 的时候这个是特别有用的。第四列的中部介绍的是 uC/OS-III 的 API 和配置指南。使用 uC/OS-III 的时候参考这些章节。最后，第四列的下部分介绍的是其它附件。

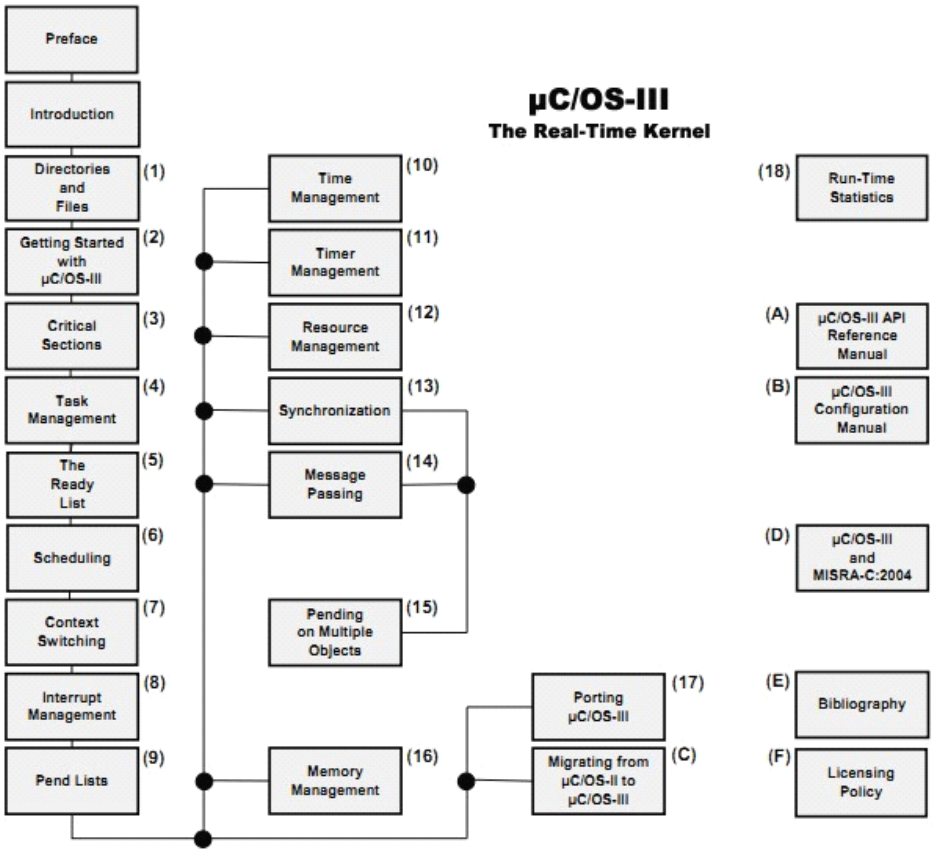


Figure 1-3 µC/OS-III Book Layout

章节 1：简介

章节 2：目录和文件。这个章节介绍了 uC/OS-III 所包括目录结构和文件。了解那些文件是必须的，这些文件该被放在哪里，模块的功能等。

章节 3:开始学习 uC/OS-III。在这个章节中，学习怎样配置和开始基于 uC/OS-III 的应用。

章节 4：临界段。介绍了什么是临界段，怎么保护临界段。

章节 5：任务管理。介绍了实时内核中最重要的部分，在多任务环境中管理任务。

章节 6：就绪队列。介绍 uC/OS-III 怎么有效地追踪所有的就绪任务。

章节 7： 任务调度。介绍了 uC/OS-III 的调度算法。

章节 8：上下文切换。介绍了什么是上下文切换，描述了任务被挂起或恢复的过程。

章节 9：中断管理。介绍了 uC/OS-III 如何处理 ISRs 产生的未预见服务。以及为什么 uC/OS-III 支持几乎所有的中断控制器。

章节 10：阻塞列表。任务可能位等待一个事件或资源而暂停运行。阻塞列表用来存放这些等待中的任务。本章介绍了 uC/OS-III 是如何管理这些列表的。

章节 11：时间管理。uC/OS-III 的服务允许用户定义任务挂起的时限。允许任务停止运行直到被恢复。这个章节也介绍了延时认识如何被恢复，怎样获取当前时基计数值，怎样设置时基计数值。

章节 12：软件定时器管理。uC/OS-III 允许用户定义任意数量的软件定时器。当一个定时到时时，函数可以被调用。定时器可以被设置为一次性的或者周期性的。这个章节还介绍了定时器管理模块的工作过程。

章节 13：资源管理。介绍了多种共享资源的技巧。每种技巧的优点和缺点都会被提及。还介绍了信号量、互斥信号量的管理。

章节 14：同步。介绍了 uC/OS-III 提供了的 2 种同步服务：信号量和事件标志组。以及当调用同步模块时的过程。

章节 15：消息通道：uC/OS-III 允许任务或 ISR 直接发送消息到任务。介绍了消息队列管理模块的一些服务。

章节 16：多对象挂起：uC/OS-III 允许应用同时挂起多个内核对象（信号量或消息队列）。这个功能使等待中的任务能在其中一个事件发生或超时时迅速被唤醒。

章节 17：内存管理。介绍了 uC/OS-III 的内存管理模块如何动态地分配和回收内存块。

章节 18：移植 uC/OS-III。如何移植 uC/OS-III 到任何架构的 CPU。

章节 19：实时统计。uC/OS-III 提供了实时运行环境的大量信息。例如上下文切换次数，CPU 使用率，每个任务的平均堆栈使用量。uC/OS-III 的 RAM 使用量，最大关中断时间，最大调度器锁存时间等。

附录 A：uC/OS-III 的 API 手册：按字母排序的 uC/OS-III 中提供的 API 服务。

附录 B: uC/OS-III 的配置手册：介绍了怎样基于应用配置 uC/OS-III。OS_CFG.H 用于配置 uC/OS-III 的功能（信号量、队列、事件标志等）。

OS_CFG_APP.H 用于配置实时特征（时钟速率，轮转值，闲置任务的堆栈大小等）。

附录 C：uC/OS-II 转换为 uC/OS-III。uC/OS-II 的 uC/OS-III 的基础。事实上，大部分能移植 uC/OS-II 的都能转换为移植 uC/OS-III。然后，很多 uC/OS-III 的 API 跟 uC/OS-II 的 API 是不一样的。这个附录将会介绍它们的不同之处。

附录 D:MISRA-C:2004：uC/OS-III 遵循 MISRA-C:2004 的大部分规则除了 7 条。

附录 E：参考文献

附录 F：uC/OS-III 许可证

2、目录和文件

这个章节将会介绍 uC/OS-III 的模块以及怎样设置他们。

图 2-1 显示的是 uC/OS-III 的架构以及与硬件的关系。包括硬件定时器和中断控制器。也应该包括 UARTs，ADCs，以太网等。

这个章节假定开发是基于 Window 平台的，并有供参考的典型目录结构。然而，因为 uC/OS-III 是以源代码形式提供的，所以它也可以被用于 Unix，Linux，或者其它的开发平台。

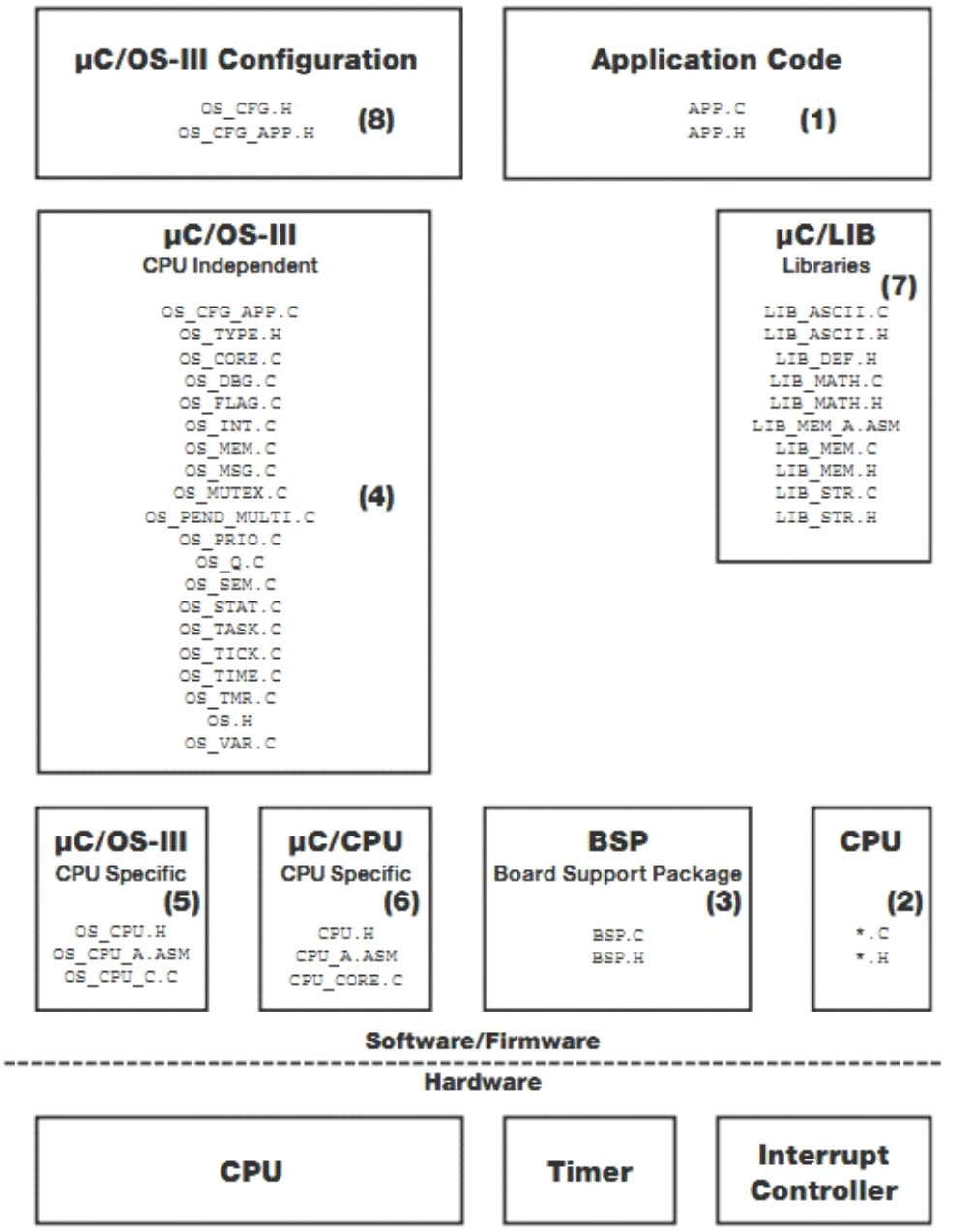


Figure 2-1 µC/OS-III Architecture

F2-1（1）应用代码包括与工程、产品相关文件。为了方便，这些被简单地叫做 APP.C 和 APP.H。Main()函数应该在 APP.C 代码中。

F2-1（2）半导体厂家通常会提供库函数以控制那些 CPU 或 MCU 的外设。这些库非常有用并且高效。因为对这些文件没有规定。所以假定为*.C，*.H。

F2-1（3）板级支持包通常被用来初始化目标板。例如打开或关闭 LED、继电器、读取开关值、读取温度传感器等。

F2-1（4）这些是 uC/OS-III 的与处理器无关的代码。这些代码都是高度遵循 ANSI C 标准。

F2-1（5）这些 uC/OS-III 代码用于适应不同架构的 CPU，在名为 port 的文件夹中。uC/OS-III 源于 uC/OS-II。uC/OS-II 能移植成功的，只要稍有改动便能移植 uC/OS-III。详见附录 C。

F2-1（6）在 Micrium，我们喜欢去总结 CPU 的功能。这些包括中断的使能和除能。CPU_???类型的文件都是独立于 CPU 的，在编译时用到，而且可能非常有用。

F2-1（7）uC/LIB 是一系列的源文件，提供了常用基本的功能如内存拷贝，字符串，ASCII 相关的函数。一些可以代替编译器提供的 stdlib 的功能。这些文件是应用与应用间，编译器与编译器间可移植。uC/OS-III 不需要这些文件，但是 uC/CPU 需要。

F2-1（8）uC/OS-III 功能的配置文件（OS_CFG.H）包含在应用中，OS_CFG_APP.H 定义了 uC/OS-III 所需的变量类型大小、数据的结构、空闲任务堆栈的大小、时钟速率、内存池大小等。

2-1 应用代码

如果 Micrium 提供了例子。那么它将被包含在如下的目录结构。

\Micrium

\Software

\EvalBoards

\<manufacturer>

\<board name>

\<compiler>

\<project name>

.

\Micrium

这是我们存放软件或工程的地方，通常位于电脑的根目录。

\Software

子目录中是软件成分。

\EvalBoards

子目录中包含了评估版的工程。

\<manufacturer>

制造商的名字　名字中不包括"<"和">"。

\<board name>

评估板的名字。Micrium 通常命名为 uC-Eval-xxxx。用 CPU 或 MCU 类型替代"xxxx"。

\<compiler>

代码所用编译器的名字

\<project name>

工程名。例如，uC/OS-III 工程会被命名为"OS-Ex1"。"-Ex1"表明工程中值包含 uC/OS-III。命名为 OS-Probe-Ex1 表示工程中包含 uC/OS-III 和 uC/Probe。

.

这些是工程的源文件，main 文件可以被命名为 APP*.*。目录中也包括配置文件 OS_CFG.H,OS_CFG_APP.H 以及其它需要的源文件。

2-2 CPU

在这个目录中，你会找到半导体厂商提供的外设库文件。

\Micrium

\Software

\CPU

\<manufacturer>

\<architecture>

.

2-3 板级支持包（BSP）

板级支持包通常是目标器件的特殊配置。实时上，写得好的话，BSP 将适用于多个工程。

\Micrium

\Software

\EvalBoards

\<manufacturer>

\<board name>

\<compiler>

\BSP

.

2-4 uC/OS-III 中独立于 CPU 的源文件

\Micrium

\Software

\uCOS-III

\Cfg\Template

\OS_APP_HOOKS.C

\OS_CFG.H

\OS_CFG_APP.H

\Source

\OS_CFG_APP.C

\OS_CORE.C

\OS_DBG.C

\0S_FLAG.C

\OS_INT.C

\OS_MEM.C

\OS_MSG.C

\OS_MUTEX.C

\OS_PEND_MULTI.C

\OS_PRIO.C

\OS_Q.C

\OS_SEM.C

\OS_STAT.C

\OS_TASK.C

\OS_TICK.C

\OS_TIME.C

\OS_TMR.C

\OS_VAR.C

\OS.H

\OS_TYPE.H

OS_APP_HOOKS.C 包含了 8 个空的能被 uC/OS-III 调用的 hook0 函数。

OS_CFG.H 配置 uC/OS-III 的功能，详见附录 B

OS_CFG_APP.H 用于配置空闲堆栈大小，时钟速率，内存池的消息数等，详见附录 B

\Source

该目录中包含了跟 CPU 无关的 uC/OS-III 源文件。这里面所有的文件需被加入到 uC/OS-III 工程中。一些功能可以在 OS_CFG.H 和 OS_CFG_APP.H 中除能。

OS_CFG_APP.C 申明变量和数组(基于 OS_CFG_APP.H 中的变量类型)。

OS_CORE.C 包含 uC/OS-III 的核心函数如 OSInit（），OSSchedule（），OSIntExit（）等等。

OS_DBG.C 申明了关于内核调试器或 uC/Probe 的常量。

OS_FLAG.C 包含了时间管理相关的代码。

OS_INT.C 包含了中断处理任务相关的代码当 OS_CFG_ISR_POST_DEFERRED_EN（OS_CFG.H 中）定义时。

OS_MEM.C 包含了 uC/OS-III 内存管理相关代码。

OS_MSG.C 包含了消息处理相关代码。

OS_MUTEX.C 包含了与互斥信号量相关的代码。

OS_PEND_MULTI.C 包含了与挂起多个信号量或消息队列相关的代码。

OS_PRIO.C 包含了管理任务优先级相关的代码。这个文件可以被改为同等功能的汇编语言写的文件(如果 CPU 支持位设置，位清除指令，计数清零指令)以提高性能。

OS_Q.C 包含了管理消息队列相关的代码。

OS_SEM.C 包含了管理与资源、同步相关的信号量代码。

OS_STAT.C 包含了与统计任务相关的代码。

OS_TASK.C 包含了与管理任务相关的代码如 OSTaskCreate()，OSTaskDel()，OSTaskChangePrio()等函数。

OS_TICK.C 包含了与任务延时，任务停止相关的代码。

OS_TIME.C 包含了任务定时相关的代码。

OS_TMR.C 包含了管理软件定时器相关的代码。

OS_VAR.C 包含了 uC/OS-III 的全局变量，这些变量都是用于管理 uC/OS-III 的，不应该被用户的代码访问。

OS.H 包含了 uC/OS-III 主要的头文件，那些声明了常量，宏，uC/OS-III 全局变量，函数原型等的文件。

OS_TYPE.H 包含了声明 uC/OS-III 文件类型的，移植时改动以更好地适应 CPU 架构。在这种情况下，这个文件需要被拷贝到 port 目录中并加以修改。详见附录 B

2-5 uC/OS-III 中 CPU 相关的源代码

\Micrium

\Software

\uCOS-III

\Ports

\<architecture>

\<compiler>

\OS_CPU.H

\OS_CPU_A.ASM

\OS_COU_C.C

OS_CPU.H 包含了 OS_TASK_SW() 宏的声明，至少需函数 OSCtxSW()，OSIntCtxSW()，OSStartHighRdy()的原型。

OS_CPU_A.ASM 包含了汇编语言定义的函数。至少需包含 OSCtxSW()，OSIntCtxSW()，OSStartHighRdy()的定义。

OS_CPU_C.C 包含了 C 代码编写的 hook 函数以及初始化任务创建时堆栈框架的相关代码。

2-6 uC/CPU,CPU 相关代码

\Micrium

\Software

\uC-CPU

\CPU_CORE.C

\CPU_CORE.H

\CPU_DEF.H

\Cfg\Template

\CPU_CFG.H

\<architecture>

\<compiler>

\CPU.H

\CPU_A.ASM

\CPU_C.C

CPU_CORE.C 适应于所有架构 CPU 的代码。包含了测量 CPU 关中断时间相关函数的代码。

CPU_CORE.H 包含了 CPU_CORE.C 函数的原型及分配了与测量关中断时间模块相关的变量。

CPU_DEF.H 包含了 uC/CPU 模块用到的其它宏定义。

CPU_CFG.H 可以配置是否使用测量关中断时间的模块，CPU 中是否有计数清零的汇编语言等。

CPU.H 定义了 uC/OS-III 变量的类型以独立于 CPU 和编译器。如 CPU_INT16U，CPU_INT32U，CPU_FP32 等数据类型。设定堆栈的生长方向。定义了宏 OS_CRITICAL_ENTER（），OS_CRITICAL_EXIT（）以及其它一些于 CPU 架构相关函数声明等。

CPU_A.ASM 包含了汇编语言编写的函数用于关中断，开中断，计数清零（如果 CPU 支持这条汇编指令的话），其它 CPU 相关的只能用汇编编写的函数。这个文件也包含开启 caches，设置 MPUs 和 MMU 等等。

CPU_C.C 特定 CPU 架构的 C 语言代码。一般来说，如果函数能用 C 编写就用 C 编写，除非用汇编编写能够产生更高的性能。

2-7 uC/LIB 可移植的函数库

uC/LIB 包含的库函数有很高的可移植性，它与适用于任何编译器。便于用户使用 Micrium 的产品。uC/OS-III 不使用任何 uC/OS-III 的函数。然而，uC/CPU 调用了 LIB_DEF.H，为了一些定义如：DEF_YES，DEF_NO，DEF_TRUE，DEF_FALSE，DEF_ON，DEF_OFF 等。

\Micrium

\Software

\uC-LIB

\LIB_ASCII.C

\LIB_ASCII.H

\LIB_DEF.H

\LIB_MATH.C

\LIB_MATH.H

\LIB_MEM.C

\LIB_MEM.H

\LIB_STR.C

\LIB_STR.H

\Cfg\Template

\LIB_CFG.H

\Ports

\<architecture>

\<compiler>

\LIB_MEM_A.ASM

LIB_CFG.H　　决定了是否用汇编语言优化（假定有一个用汇编写的文件 LIB_MEM_A.ASM）以及其它一些#define。

2-8 概要

下面的概要是基于 uC/OS-III 工程的所有目录文件。后缀为"<-Cfg"的文件通常要被拷贝到应用文件夹中并被修改。

```
\Micrium
   \Software
      \EvalBoards
         \<manufacturer>
            \<board name>
               \<compiler>
                  \<project name>
                     \APP.C
                     \APP.H
                     \other
      \CPU
         \<manufacturer>
            \<architecture>
               \*.*
```

```
    \<architecture>
        \<compiler>
            \CPU.H
            \CPU_A.ASM
            \CPU_C.C
\uC-LIB
    \LIB_ASCII.C
    \LIB_ASCII.H
    \LIB_DEF.H
    \LIB_MATH.C
    \LIB_MATH.H
    \LIB_MEM.C
    \LIB_MEM.H
    \LIB_STR.C
    \LIB_STR.H
    \Cfg\Template
        \LIB_CFG.H                  <-Cfg
    \Ports
        \<architecture>
            \<compiler>
                \LIB_MEM_A.ASM
```

```
\uCOS-III
    \Cfg\Template
        \OS_APP_HOOKS.C
        \OS_CFG.H                   <-Cfg
        \OS_CFG_APP.H               <-Cfg
    \Source
        \OS_CFG_APP.C
        \OS_CORE.C
        \OS_DBG.C
        \OS_FLAG.C
        \OS_INT.C
        \OS_MEM.C
        \OS_MSG.C
        \OS_MUTEX.C
        \OS_PEND_MULTI.C
        \OS_PRIO.C
        \OS_Q.C
        \OS_SEM.C
        \OS_STAT.C
        \OS_TASK.C
        \OS_TICK.C
        \OS_TIME.C
        \OS_TMR.C
        \OS_VAR.C
        \OS.H
        \OS_TYPE.H                  <-Cfg
    \Ports
        \<architecture>
            \<compiler>
                \OS_CPU.H
                \OS_CPU_A.ASM
                \OS_CPU_C.C
\uC-CPU
    \CPU_CORE.C
    \CPU_CORE.H
    \CPU_DEF.H
    \Cfg\Template
        \CPU_CFG.H                  <-Cfg
```

3、开始学习 uC/OS-III

uC/OS-III 以函数的形式提供特定的服务。uC/OS-III 提供的服务包括任务管理，信号量，消息队列，互斥信号量等。uC/OS-III 提供了大约 70 种功能。

在这个章节中，读者可以领会使用 uC/OS-III 是多么简单。参考附录 A 的 uC/OS-III 的 API。

假定工程已被设置为前一章节那样（目录和文件已经就位），并且在 C 编译器中进行。然而，这章节对工具和处理器没有做出要求。

3-1 单任务应用

列表 3-1 显示的是应用文件 APP.C 的顶部代码。

```
/*
*********************************************************************************************************
*                                       INCLUDE FILES
*********************************************************************************************************
*/
#include <app_cfg.h>                                                    (1)
#include <bsp.h>
#include <os.h>
/*
*********************************************************************************************************
*                                    LOCAL GLOBAL VARIABLES
*********************************************************************************************************
*/
static  OS_TCB          AppTaskStartTCB;                                (2)
static  CPU_STK         AppTaskStartStk[APP_TASK_START_STK_SIZE];       (3)
/*
*********************************************************************************************************
*                                    FUNCTION PROTOTYPES
*********************************************************************************************************
*/
static  void  AppTaskStart (void *p_arg);                               (4)
```

Listing 3-1 **APP.C (1st Part)**

L3-1(1)　正如所有的 C 程序一样，添加必要的头文件到应用中。APP_CFG.H 是用于配置的头文件。例如，APP_CFG.H 中包含的#define 常量确定了任务优先级，堆栈大小，以及其他特性。

BSP.H 是 BSP 的头文件，包含了#define 及函数原型如 BSP_Init(),,SP_LED_On()，OS_TS_GET()等。

OS.H 是 uC/OS-III 的主要头文件，包含了以下头文件：

OS_CFG.H

CPU.H

CPU_CFG.H

CPU_CORE.H

OS_TYPE.H

OS_CPU.H

L3-1（2）定义任务控制块（OS_TCB）。

L3-1（3）每个任务需要创建自己的堆栈。堆栈的数据类型必须是 CPU_STK。堆栈可以被静态地分配或者通过 malloc()动态地分配。没有必要释放堆栈空间，因为任务将不会被删除，堆栈将一直被使用。

L3-1（4）这是一个函数原型。

大部分的 C 应用程序开始于 main()，见列表 3-2.

```
void  main (void)
{
    OS_ERR  err;

    BSP_IntDisAll();                                                    (1)
    OSInit(&err);                                                       (2)
    if (err != OS_ERR_NONE) {
        /* Something didn't get initialized correctly ...                   */
        /* ... check OS.H for the meaning of the error code, see OS_ERR_xxxx */
    }
    OSTaskCreate((OS_TCB     *)&AppTaskStartTCB,                        (3)
                 (CPU_CHAR   *)"App Task Start",                        (4)
                 (OS_TASK_PTR )AppTaskStart,                            (5)
                 (void       *)0,                                       (6)
                 (OS_PRIO     )APP_TASK_START_PRIO,                     (7)
                 (CPU_STK    *)&AppTaskStartStk[0],                     (8)
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,            (9)
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,                (10)
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),  (11)
                 (OS_ERR     *)&err);                                   (12)
    if (err != OS_ERR_NONE) {
        /* The task didn't get created.  Lookup the value of the error code ... */
        /* ... in OS.H for the meaning of the error                             */
    }
    OSStart(&err);                                                      (13)
    if (err != OS_ERR_NONE) {
        /* Your code is NEVER supposed to come back to this point.          */
    }
}
```

Listing 3-2 **APP.C (2nd Part)**

L3-2（1）main()开始时调用一个 BSP 函数用于关闭所有中断。在大部分处理器中，中断在启动时是关闭的。无论如何，在启动时关闭所

有的外设中断是更安全的。

L3-2（2）调用 OSInit(),用于初始化 uC/OS-III。OSInit（）初始化内部变量和数据结构，同时产生 2 个到 5 个内部任务。最低程度，uC/OS-III 须创建空闲任务 OS_IdleTask（），当没有其他任务运行时就运行空闲任务。uC/OS-III 也创建时基任务。

根据配置文件中所配置的，uC/OS-III 会创建统计任务 OS_StatTask()、定时器任务 OS_TmrTask()、中断队列处理任务 OS_IntQTask()。

这些将会在第五章"任务管理"中详细介绍。

大多数的 uC/OS-III 函数会通过一个指向 OS_ERR 变量的指针返回一个错误代号。如果 OSInit()初始化函数运行成功，错误代号被设为 OS_ERR_NONE。如果在初始化不成功，uC/OS-III 会根据执行的结果返回对应的错误代号。参照 OS.H 中的错误代号。特别的，所有的错误代号都是以 OS_ERR_作为前缀的。

OSInit（）必须在 uC/OS-III 的其它函数之前调用。

L3-2(3) 通过调用 OSTaskCreate（）创建任务。OSTaskCreate（）需要 13 个参数。第一个参数是任务堆栈的地址。{该任务堆栈的开始地址}详见第 5 章。

L3-2(4)OSTaskCreate（）允许给每个任务分配名字。OS_TCB 中存储了指向任务名的指针。因而任务名长度无限制，必须以空字符结尾。

L3-2(5)　第 3 个参数是指向任务代码的指针。典型的 uC/OS-III 任务是无限循环执行的如下。

```
void  MyTask (void *p_arg)
{
    /* Do something with "p_arg".
    while (1) {
        /* Task body */
    }
}
```

任务第一次开始时接收一个参数。任务看起来像是一个可以被调用的 C 函数。然而，用户代码不允许调用任务。任务被创建后是由 uC/OS-III 调用的。

L3-2（6）OSTaskCreate()的第四个参数是一个实参，第一次被调用时 OSTaskCreate()接收这个变量，传递给所创建的任务 MyTask()中的 "p_arg"。

任务的参数可以是任意的指针。例如，用户可以传送数据结构等给任务。{参数类型是 void*}

L3-2（7）OSTaskCreate()的第五个参数是任务的优先级。优先级确立了任务间的重要性关系。参数值越小优先级越高。可以设置优先级数值为 1 到 OS_CFG_PRIO_MAX-2。要避免使用优先级#0 和优先级 OS_CFG_PRIO_MAX-1 。因为这些是为 uC/OS-III 保留的。OS_CFG_PRIO_MAX 是编译时配置的，在 OS_CFG.H 中定义。

L3-2（8）是任务堆栈的基地址。基地址通常是分配给该任务的堆栈的最低内存位置。

L3-2（9）第七个参数是地址“水印”，当堆栈生长到指定位置时就不再允许其生长。详见章节 5。在例子中，当堆栈空间只剩下 10%的时候将会限制堆栈的生长。

L3-2（10）OSTaskCreate()的第八个参数定义了任务的堆栈大小(以CPU_STK 为数据类型而不是字节)。例如，如果要分配 1KB 大小的堆栈空间，因为 CPU_STK 是 32 位的，所以这个参数是 256.

L3-2（11）接下来的三个参数将被跳过因为这三个参数跟当前的话题无关，直接设置为 0。再下面一个参数是 OSTaskCreate()的可选项。例如，在运行时堆栈会被检测（假定统计任务在 OS_CFG.H 中使能），任务创建时堆栈会被初始化。

L3-2（12）OSTaskCreate()的最后一个参数是一个指针，将接收根据函数执行结果所返回的错误代号。如果 OSTaskCreate()函数执行成功，错误代号将会是 OS_ERR_NONE，否则会返回其它的错误代号（参见 OS.H 中错误代号的定义）。

L3-2（13）调用 uC/OS-III 过程在 main()函数中的最后一个步骤是调用 OSStart()，开始多任务处理。特别的，在 OSStart()调用之前 uC/OS-III 会选择最高优先级任务。最高优先级的任务通常是 OS_IntQTask()（假定在 OS_CFG.H 中定义了 OS_CFG_ISR_POST_DEFERRED_EN）。在种情况下，OS_IntQTask()将会执行一些它自身的初始化操作，然后 uC/OS-III 将会切换到下一个最高优先级的任务。

OSTaskCreate()中的一些指针没有意义。首先，在调用 OSStart()之前创建你想要创建的任务。然而，我们推荐你此时最好只创建一个任务，因为在这种情况下 uC/OS-III 可以确定 CPU 的速率，可以测量 CPU 的使用率。如果应用需要另外的内核对象如信号量、消息队列，那么推荐在调用 OSStart()之前创建它们。最后，注意中断还没有开启。

这将会在下面一个函数 AppTaskStart()中讨论。见列表 3-3

```
static  void  AppTaskStart (void *p_arg)                        (1)
{
    OS_ERR  err;


    p_arg = p_arg;
    BSP_Init();                                                 (2)
    CPU_Init();                                                 (3)
    BSP_Cfg_Tick();                                             (4)
    BSP_LED_Off(0);                                             (5)
    while (1) {                                                 (6)
        BSP_LED_Toggle(0);                                      (7)
        OSTimeDlyHMSM((CPU_INT16U)  0,                          (8)
                      (CPU_INT16U)  0,
                      (CPU_INT16U)  0,
                      (CPU_INT32U)100,
                      (OS_OPT    )OS_OPT_TIME_HMSM_STRICT,
                      (OS_ERR   *)&err);
        /* Check for 'err' */
    }
}
```

Listing 3-3 **APP.C (3rd Part)**

L3-3(1)正如前面提到的，一个任务看起来像是一个 C 函数。参数“p_arg”是 OSTaskCreate()传递给任务 AppTaskStart()的参数。

L3-3（2）BSP_Init()用于初始化目标板的硬件。目标板可能会有一些 GPIO，继电器，传感器等需要被设置。这个函数是在 BSP.C 中定义的。

L3-3（3）CPU_Init()初始化 uC/CPU 的服务。uC/CPU 用于测量中断响应时间，读取时间戳，提供仿真的计数清零指令等（假定用户所使用的处理器没有那种汇编指令）。

L3-3（4）BSP_Cfg_Tick()设置 uC/OS-III 的时基中断。为此，这个函数需要初始化一个硬件定时器用于中断 CPU，其频率为 OS_CFG_TICK_RATE_HZ（在 OS_CFG_APP.H 中定义）。

L3-3（5）BSP_LED_Off()用于关闭 LED，参数为 0 表示关闭全部的 LED。这是个用户函数，可删除。

L3-3（6）所有的 uC/OS-III 任务需要被设置为无限循环。

L3-3（7）BSP_LED_Toggle()用于打开 LED，同样的，参数为 0 表示打开全部 LED。改参数为 1 表示标号为#1 的 LED 被打开。但哪个 LED 标号为#1 呢？这取决于设计者。特别的，可以将 LED 的操作封装为如 BSP_LED_On(),BSP_LED_Off()、BSP_LED_Toggle()的函数。我们更喜欢定义 LED 为逻辑位(1,2,3 等)，每一位对应一个端口值。该函数是用户函数，可删除

L3-3（8）最后，每个任务可以调用 uC/OS-III 中的函数，可以让任务待一个事件（信号量，或来自于中断的消息，或来自于其它任务的消息。）而被挂起。任务可以设置等待期限(通过调用 OSTimeDly()或者 OSTimeDlyHMSM())，章节 11“时间管理”介绍了等待期限的相关信息。

3-2 内核对象与多任务应用

列表 3-4 到 3-8 的代码展示了一个包含三个任务的完整例子，一个是 mutex，一个是信号量，一个是消息队列。

```
/*
*********************************************************************************************************
*                                          INCLUDE FILES
*********************************************************************************************************
*/
#include <app_cfg.h>
#include <bsp.h>
#include <os.h>
/*
*********************************************************************************************************
*                                      LOCAL GLOBAL VARIABLES
*********************************************************************************************************
*/
static  OS_TCB          AppTaskStartTCB;                            (1)
static  OS_TCB          AppTask1_TCB;
static  OS_TCB          AppTask2_TCB;
static  OS_MUTEX        AppMutex;                                   (2)
static  OS_Q            AppQ;                                       (3)
static  CPU_STK         AppTaskStartStk[APP_TASK_START_STK_SIZE];   (4)
static  CPU_STK         AppTask1_Stk[128];
static  CPU_STK         AppTask2_Stk[128];
/*
*********************************************************************************************************
*                                     FUNCTION PROTOTYPES
*********************************************************************************************************
*/
static  void  AppTaskStart (void *p_arg);                           (5)
static  void  AppTask1     (void *p_arg);
static  void  AppTask2     (void *p_arg);
```

Listing 3-4 **APP.C (1st Part)**

L3-4（1) 分别为每个任务分配一个 OS_TCB。

L3-4（2）互斥信号量（mutex）是一个内核对象(一个结构体)，用于保护共享资源。任务要访问共享资源就必须先获得 mutex。mutex 的拥有者使用完这个资源后就必须释放这个 mutex。这个例子示范了这个过程。

L3-4（3）消息队列是一个内核对象，ISR 或任务可以直接发送消息到另一个任务。发送者制定一个消息并将其发送到目标任务的消息

队列。目标任务等待消息的到达。如果消息到达了，目标任务取得这些消息。如果消息队列为空，目标将会被安放在挂起队列中并与消息队列保持联系。这个过程将会在这个例子中介绍。

L3-4 （4）为每个任务分配堆栈。{由于是 CPU_STK 类型，它是 32 位的，所以 128 个 CPU_STK 即位 512B}

L3-4（5）用户需要申明这些任务。

列表 3-5 展示了 C 语言的入口——main()函数。

```
void  main (void)
{
    OS_ERR  err;

    BSP_IntDisAll();
    OSInit(&err);
    /* Check for 'err' */

    OSMutexCreate((OS_MUTEX  *)&AppMutex,                                    (1)
                  (CPU_CHAR  *)"My App. Mutex",
                  (OS_ERR    *)&err);
    /* Check for 'err' */

    OSQCreate    ((OS_Q      *)&AppQ,                                        (2)
                  (CPU_CHAR  *)"My App Queue",
                  (OS_MSG_QTY )10,
                  (OS_ERR    *)&err);
    /* Check for 'err' */

    OSTaskCreate((OS_TCB     *)&AppTaskStartTCB,                             (3)
                 (CPU_CHAR   *)"App Task Start",
                 (OS_TASK_PTR )AppTaskStart,
                 (void       *)0,
                 (OS_PRIO     )APP_TASK_START_PRIO,
                 (CPU_STK    *)&AppTaskStartStk[0],
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR     *)&err);
   /* Check for 'err' */

   OSStart(&err);
   /* Check for 'err' */
}
```

Listing 3-5 **APP.C (2nd Part)**

L3-5（1)调用 OSMutexCreate()创建一个 mutex。指定 OS_MUTEX 对象的地址。详见章节 13"资源管理"。

可以为 mutex 定义一个 ASCII 名字，对调试会很有用处。

L3-5（2)调用 OSQCreate()创建消息队列，并指定 OS_Q 对象的地址。详见章节 15“消息传递”。

为消息队列命名。

定义该消息队列可接受消息的个数。这个值必须大于 0。如果消息者发送消息数超过了消息接收任务的承受能力。那么消息将会被丢失。可以通过增加消息队列的大小或者提供消息接收任务的优先级提升其承受能力。

L3-5（3）第一个应用任务被创建。

列表 3-6 展示了如何在多任务调度开始后创建任务。

```
static  void  AppTaskStart (void *p_arg)
{
    OS_ERR  err;


    p_arg = p_arg;
    BSP_Init();
    CPU_Init();
    BSP_Cfg_Tick();
    OSTaskCreate((OS_TCB     *)&AppTask1_TCB,                                    (1)
                 (CPU_CHAR   *)"App Task 1",
                 (OS_TASK_PTR )AppTask1,
                 (void       *)0,
                 (OS_PRIO     )5,
                 (CPU_STK    *)&AppTask1_Stk[0],
                 (CPU_STK_SIZE)0,
                 (CPU_STK_SIZE)128,
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR     *)&err);

    OSTaskCreate((OS_TCB     *)&AppTask2_TCB,                                    (2)
                 (CPU_CHAR   *)"App Task 2",
                 (OS_TASK_PTR )AppTask2,
                 (void       *)0,
                 (OS_PRIO     )6,
                 (CPU_STK    *)&AppTask2_Stk[0],
                 (CPU_STK_SIZE)0,
                 (CPU_STK_SIZE)128,
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR     *)&err);
    BSP_LED_Off(0);
    while (1) {
        BSP_LED_Toggle(0);
        OSTimeDlyHMSM((CPU_INT16U)  0,
                      (CPU_INT16U)  0,
                      (CPU_INT16U)  0,
                      (CPU_INT32U)100,
                      (OS_OPT    )OS_OPT_TIME_HMSM_STRICT,
                      (OS_ERR   *)&err);
    }
}
```

Listing 3-6 **APP.C (3rd Part)**

L3-6（1）通过调用 OSTaskCreate（）创建任务#1。如果被创建任务的优先级大于创建它的任务的优先级。uC/OS-III 会转向执行任务#1. 如果被创建任务的优先级小于创建它的任务的优先级，OSTaskCreate()将会返回 AppTaskStart()继续执行下面的代码。

L3-6（2）任务#2 被创建，如果其优先级大于 AppTaskStart()的优先级 。那么 uC/OS-III 将会立即去执行任务#2。

```
static  void  AppTask1 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;


    p_arg = p_arg;
    while (1) {
        OSTimeDly ((OS_TICK     )1,                                 (1)
                   (OS_OPT      )OS_OPT_TIME_DLY,
                   (OS_ERR     *)&err);
        OSQPost    ((OS_Q       *)&AppQ,                           (2)
                    (void       *)1;
                    (OS_MSG_SIZE)sizeof(void *),
                    (OS_OPT      )OS_OPT_POST_FIFO,
                    (OS_ERR     *)&err);
        OSMutexPend((OS_MUTEX   *)&AppMutex,                       (3)
                    (OS_TICK     )0,
                    (OS_OPT      )OS_OPT_PEND_BLOCKING;
                    (CPU_TS     *)&ts,
                    (OS_ERR     *)&err);
        /* Access shared resource */                               (4)
        OSMutexPost((OS_MUTEX   *)&AppMutex,                       (5)
                    (OS_OPT      )OS_OPT_POST_NONE,
                    (OS_ERR     *)&err);
    }
}
```

Listing 3-7 **APP.C (4th Part)**

L3-7（1）该任务在执行前先等待一个时基。如果 uC/OS-III 的时基频率为 1000HZ。那么这个任务每毫秒被执行一次。

L3-7（2) 该任务向消息队列 AppQ 发送一个消息。为了说明， 在例子中发送的是一个 void* 1。但实际上消息中包含着的是一个地址。内存地址、函数的地址、或者其它需要被传送的地址。

L3-7 （3）该任务等待一个信号量。如果它需要访问已被其它任务占用的资源，APPTaskStart1()等待这个 mutex 被释放。第二个参数为其等待时限，以时基为单位，若 0 时就会一直等待下去。

L3-7（4）当 OSMutexPend()返回了，就表明该任务占用了这个共享资源。共享资源可能是变量、数组、数据域、IO 等。

L3-7（5）如果任务完成对共享资源的使用，它必须调用 OSMutexPost()释放这个 mutex。

```
static  void  AppTask2 (void *p_arg)
{
    OS_ERR       err;
    void        *p_msg;
    OS_MSG_SIZE  msg_size;
    CPU_TS       ts;
    CPU_TS       ts_delta;


    p_arg = p_arg;
    while (1) {
        p_msg = OSQPend((OS_Q        *)&AppQ,                          (1)
                        (OS_MSG_SIZE *)&msg_size,
                        (OS_TICK      )0,
                        (OS_OPT       )OS_OPT_PEND_BLOCKING,
                        (CPU_TS      *)&ts,
                        (OS_ERR      *)&err);
        ts_delta = OS_TS_GET() - ts;                                  (2)
        /* Process message received */                               (3)
    }
}
```

Listing 3-8 **APP.C (5th Part)**

L3-8（1）任务#2 开始执行，并等待消息队列 AppQ 中的消息,这个任务会无限等待下去因为第三个参数为 0，意味着无限等待。

消息的发送者和接收者都必须知道这个消息中所包含的信息。接收消息的大小存于"msg_size"。"p_msg"是调用该函数后返回的消息地址，指向内存区且"msg_size"中包含这这个内存区的大小。

当接收到消息时，"ts"中包含的是消息被发送时的时间戳。时间戳的值读取于硬件定时器。时间戳是一个 32 位（或更大）的值。

L3-8（2）测得消息是什么时候被发送的，用户就能测得任务接收这个消息所用的时间。读取现在的时间戳并减去消息被发送时的时间戳。请注意，消息被发送时，等待消息的任务可能不会立即接收到消息，因为 ISR 或更高优先级的任务需要被运行。

L3-8（3）处理接收到的消息。

4、临界段

临界段代码，也称作临界域，是一段不可分割的代码。uC/OS-III 中包含了很多临界段代码。如果临界段可能被中断，那么就需要关中断以保护临界段。如果临界段可能被任务级代码打断，那么需要锁调度器保护临界段。

uC/OS-III 中的临界段的保护方法决定于 ISR 中对消息的处理方式。详见章节 9“中断管理”。如果 OS_CFG_ISR_POST_DEFERRED_EN 被设为 0（见 OS_CFG.H），在进入临界段之前 uC/OS-III 会关中断。如果 OS_CFG_ISR_POST_DEFERRED_EN 被设为 1，在进入大多数临界段之前会关调度器。

uC/OS-III 定义了一个进入临界段的宏和两个出临界段的宏。

OS_CRITICAL_ENTER（），

OS_CRITICAL_EXIT（），

OS_CRITICAL_EXIT_NO_SCHED（）

这些是 uC/OS-III 的内部宏，不能被用户代码调用。然而，如果你需要进入你自己定义的临界段。请查阅第十三章"资源管理"。

4-1 关中断

设置 OS_CFG_ISR_POST_DEFERRED_EN 为 0 后，在进入临界段之前 uC/OS-III 会关中断，在离开临界段之后开中断。

{这里所说的 CPU 寄存器是指辅助 CPU 进行处理的多个寄存器}

OS_CRITICAL_ENTER() 调用 uC/CPU 的宏 CPU_CRITICAL_ENTER()，然后调用 CPU_SR_Save()。CPU_SR_Save()是用汇编写的用于保存当前 CPU 寄存器并关中断。寄存器值以类型为“cpu_sr”的变量存于调用者堆栈。

OS_CRITICAL_EXIT() 和 OS_CRITICAL_EXIT_NO_SCHED() 都会调用 uC/CPU 的宏 CPU_CRITICAL_EXIT()。CPU_CRITICAL_EXIT()调用 CPU_SR_Restore()。CPU_SR_Restore()恢复所保存寄存器值到 CPU 寄存器，也就是 OS_CRITICAL_ENTER() 调用前的状态。

典型的宏代码如列表 4-1 所示

```
#define  OS_CRITICAL_ENTER()          { CPU_CRITICAL_ENTER(); }
#define  OS_CRITICAL_EXIT()           { CPU_CRITICAL_EXIT();  }
#define  OS_CRITICAL_EXIT_NO_SCHED()  { CPU_CRITICAL_EXIT();  }
```

Listing 4-1 **Critical section code – Disabling interrupts**

4-1-1 测量关中断时间

uC/CPU 提供了测量关中断时间的功能。通过设置 CPU_CFG.H 中的 CPU_CFG_TIME_MEAS_INT_DIS_EN 为 1 启用该功能。

每次关中断前开始测量，开中断后结束测量。测量功能保存了 2 个方面的测量值，任务总的关中断时间，每个任务最近一次关中断的时间。因此，用户可以根据任务的关中断时间对其加以优化。

每个任务的关中断时间在上文保存的时候被保存于 OS_TCB（详见 OS_CPU.C 中的 OSTaskSwHook()和第八章"上下文切换"）。

{我将上下文切换分成两个部分：上文保存、下文载入}

时间戳的控制单元位于 CPU_TS 中。时间戳的速率决定于 CPU 的速率。例如，如果 CPU 速率为 1MHz，时间戳的速率为 1MHz。那么 CPU_TS 的分辨率为 1 微秒。

显然，测出的关中断时间还包括了测量时消耗的额外时间。然而，减掉测量时所耗时间就是实际上的关中断时间。

关中断时间跟处理器的指令、速度、内存访问速度有很大的关系。在这种情况下，硬件设计者应介绍内存的访问速度，它是影响整个系统性能的。

4-2 锁住调度器

当设置 OS_CFG_ISR_POST_DEFERRED_EN 为 1 时，在进入临界段前 uC/OS-III 会锁住调度器，退出临界段后开启调度器。

OS_CRITICAL_ENTER()递增 OSSchedLockNestingCtr，给调度器加锁。这是一个决定调度器是否被开启的变量。如果它不为 0 则调度器被锁。

{称它为调度器锁嵌套值，表示调度器被加了几把锁}

OS_CRITICAL_EXIT()将 OSSchedLockNestingCtr 递减，给调度器解锁。

{调度器锁嵌套值被减为 0 时，就会调用调度器}

OS_CRITICAL_EXIT_NO_SCHED() 也递减 OSSchedLockNestingCtr 的值，不同的是当其值减为 0 时，不调用调度器。

```
#define  OS_CRITICAL_ENTER()          {                                                \
                                          CPU_CRITICAL_ENTER();                       \
                                          OSSchedLockNestingCtr++;                    \
                                          CPU_CRITICAL_EXIT();                        \
                                      }
#define  OS_CRITICAL_EXIT()           {
                                          CPU_CRITICAL_ENTER();                       \
                                          OSSchedLockNestingCtr--;                    \
                                          if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {  \
                                              CPU_CRITICAL_EXIT();                    \
                                              OSSched();                              \
                                          } else {                                    \
                                              CPU_CRITICAL_EXIT();                    \
                                          }                                           \
                                      }
#define  OS_CRITICAL_EXIT_NO_SCHED()  {
                                          CPU_CRITICAL_ENTER();                       \
                                          OSSchedLockNestingCtr--;                    \
                                          CPU_CRITICAL_EXIT();                        \
                                      }
```

Listing 4-2 **Critical section code – Locking the Scheduler**

4-2-1 测量锁调度器时间

uC/OS-III 提供了测量锁调度器时间的功能，通过设置 OS_CFG.H 中的 OS_CFG_SCHED_LOCK_TIME_MEAS_EN 为 1 开启。

加锁调度器前测量开始，解锁调度器后测量结束。测得的两种值为:总的锁调度器时间，每个任务的锁调度器时间。因此，用户可以知道每个任务的锁调度器时间，并根据此优化代码。

每个任务的锁调度器时间在上下文切换时保持于任务的 OS_TCB（详见 OS_CPU_C.C 中的 OSTaskSwHook()和第八章“上下文切换”）。

测得的锁调度器时间还包括测量时额外增加的时间。减掉额外的时间就是锁调度器时间的准确值。

4-3uC/OS-III 与长临界段

表 4-1 展示了 uC/OS-III 某些情况下的较长临界段。了解这些内容会帮助用户管理 uC/OS-III 的临界段。

功能	原因
多个任务具有相同优先级	这 uC/OS-III 的一个重要功能，多个任务具有相关优先级会产生一个长临界段。然而，当较少任务具有相同优先级时，中断延时会很小，此时可以用关中断方法保护临界段。
事件标志组 详见章节 14 “同步”	如果多个任务等待不同的事件，遍历这些任务需要很多处理时间，就会产生长临界段。如果较少的任务在等待事件，临界段会很短，那么可以使用关中断的方法保护临界段。
任务同时等待多个对象 详见章节 16	任务同时等待多个对象是uC/OS-III提供的很复杂的功能，它需要很长的临界段。如果这样，推荐使用关调度器的方法。
广播消息	uC/OS-III 在处理广播消息时锁调度器保护临界段。

表 4-1 关中断或锁调度器

4-4 总结

uC/OS-III 中会用到临界段，用关中断(OS_CFG.H 中设置 OS_CFG_ISR_POST_DEFERRED_EN 为 0)或者锁调度器（设置 OS_CFG_ISR_POST_DEFERRED_EN 为 1）实现保护临界段的功能。用户程序不能使用这些代码：

OS_CRITICAL_ENTER()

OS_CRITICAL_EXIT()

OS_CRITICAL_EXIT_NO_SCHED()

如果设置了 CPU_CFG.H 中的 CPU_CFG_TIME_MEAS_INT_DIS_EN 为 1 时。uC/CPU 将会任务总的关中断时间和每个任务的关中断时间。

如果设置了 OS_CFG.H 中的 OS_CFG_SCHED_LOCK_TIME_MEAS_EN 为 1 时。uC/OS-III 会测量任务总的锁调度器时间和每个任务锁调度器时间。

5、任务管理

实时应用中一般将工作拆分为多个任务，每个任务都需要是可靠的。使用 uC/OS-III 可以轻松地解决这个问题。任务（也叫做线程）是简单的程序。单 CPU 中，在任何时刻只能是一个任务被执行。

uC/OS-III 支持多任务且对任务数量没有限制，任务数仅取决于处理器内存的大小(RAM)。多任务调度是任务间占用 CPU 的过程。

CPU 有根据算法切换任务。多任务调度让人感觉到是有多个 CPU 在运行，并最大化利用 CPU。多任务调用有助于模块化应用，是最重要的功能之一，能帮助程序员管理复杂的实时性应用。它也使程序易于设计和维护。

任务用于监控输入、更新输出、计算、循环控制、显示、读按钮和键盘、与其它系统交流等。有些应用中可能只包含少数任务，有些应用中也可能包含上百个任务。任务数多并不意味这设计有多好或者有多有效，这依赖于应用的需要。任务的功能也要根据应用设计。一个任务可能只需要工作几微秒，然而有些任务可能就需要工作几十毫秒了。

任务看起来像 C 函数。有 2 种类型的任务:运行一次(列表 5-1)和无限循环（列表 5-2）。在大多数嵌入式系统中，任务通常是无限循环的。任务不能像 C 函数那样，它是不能 return 的。

当任务第一次执行时，会传入一个变量"p_arg"。这是一个指向 void 的指针。用于变量的地址、结构体地址、或者函数的地址等。如果需要，可以创建多个相同的任务，使用相同的代码（相同任务体），而产生有不同的运行结果。例如，4 个异步串行端口有各自的任务。然而，任务代码实际上是独立的，只是复制了 4 次而已，这些任务可以接收通过指针一个包含串口数据的结构体（如波特率、IO 端口地址，中断向量号等）。

只运行一次的任务结束时必须通过调用 OSTaskDel()删除自己。这样可以使系统中的任务数减少。在任务体中，任务可以调用 uC/OS-III 提供的大部分函数帮助完成其所需要完成的功能。

```
void MyTask (void *p_arg)
{
    OS_ERR  err;
    /* Local variables                                          */


    /* Do something with 'p_arg'                                */
    /* Task initialization                                      */
    /* Task body ... do work!                                   */
    OSTaskDel((OS_TCB *)0, &err);
}
```

Listing 5-1 **Run-To-Completion task**

在 uC/OS-III 中，不管是 C 语言函数还是调用汇编语言函数。该函数都用可能被多个任务同时调用。一个可重入的函数不含有静态变量以及全局变量除非被保护（uC/OS-III 提供了这种保护机构）。

在嵌入式系统中通常使用无限循环的任务，因为应用中有很多需要重复的工作（例如，读取输入值、更新显示、控制操作等）。这是不同于 C 函数的一个方面（C 函数可以用 while(1)或 for(;;)实现相同

的功能）。当你读懂 uC/OS-III 后，你会对这个观念更加清晰。

```
void MyTask (void *p_arg)
{
    /* Local variables                                          */


    /* Do something with "p_arg"                                */
    /* Task initialization                                      */
    while (DEF_ON) {      /* Task body, as an infinite loop.    */
        :
        /* Task body ... do work!                               */
        :
        /* Must call one of the following services:             */
        /*    OSFlagPend()                                      */
        /*    OSMutexPend()                                     */
        /*    OSPendMulti()                                     */
        /*    OSQPend()                                         */
        /*    OSSemPend()                                       */
        /*    OSTimeDly()                                       */
        /*    OSTimeDlyHMSM()                                   */
        /*    OSTaskQPend()                                     */
        /*    OSTaskSemPend()                                   */
        /*    OSTaskSuspend()     (Suspend self)                */
        /*    OSTaskDel()         (Delete  self)                */
        :
        /* Task body ... do work!                               */
        :
    }
}
```

Listing 5-2 **Infinite Loop task**

任务可以延时一段时间（调用 OSTimeDly() 或者 OSTimeDlyHSM()）。例如，应用中需要每 100ms 扫描键盘一次。在这种情况下，延时 100ms 然后检测键盘上是否有键被按下，然后执行相应的操作。

同样的，任务等待的事件可以是以太网控制器发送的包。在这种情况下，任务会调用 OS???Pend()函数（以这种形式定义的函数）。一旦包接收完成，任务根据包内容进行下一步操作。

值得注意的是，任务在等待事件时，它不会占用 CPU。

uC/OS-III 需要通过调用函数 OSTaskCreate()创建任务。OSTaskCreate()函数的原型如下所示。

```
void  OSTaskCreate (OS_TCB        *p_tcb,
                    OS_CHAR       *p_name,
                    OS_TASK_PTR    p_task,
                    void          *p_arg,
                    OS_PRIO        prio,
                    CPU_STK       *p_stk_base,
                    CPU_STK_SIZE   stk_limit,
                    CPU_STK_SIZE   stk_size,
                    OS_MSG_QTY     q_size,
                    OS_TICK        time_slice,
                    void          *p_ext,
                    OS_OPT         opt,
                    OS_ERR        *p_err)
```

关于 OSTaskCreate()完整的描述以及它的参数详见于附录 A。另外，创建一个任务时必须为其分配一个 TCB，一个堆栈，一个优先级和其它一些参数。如图 5-1

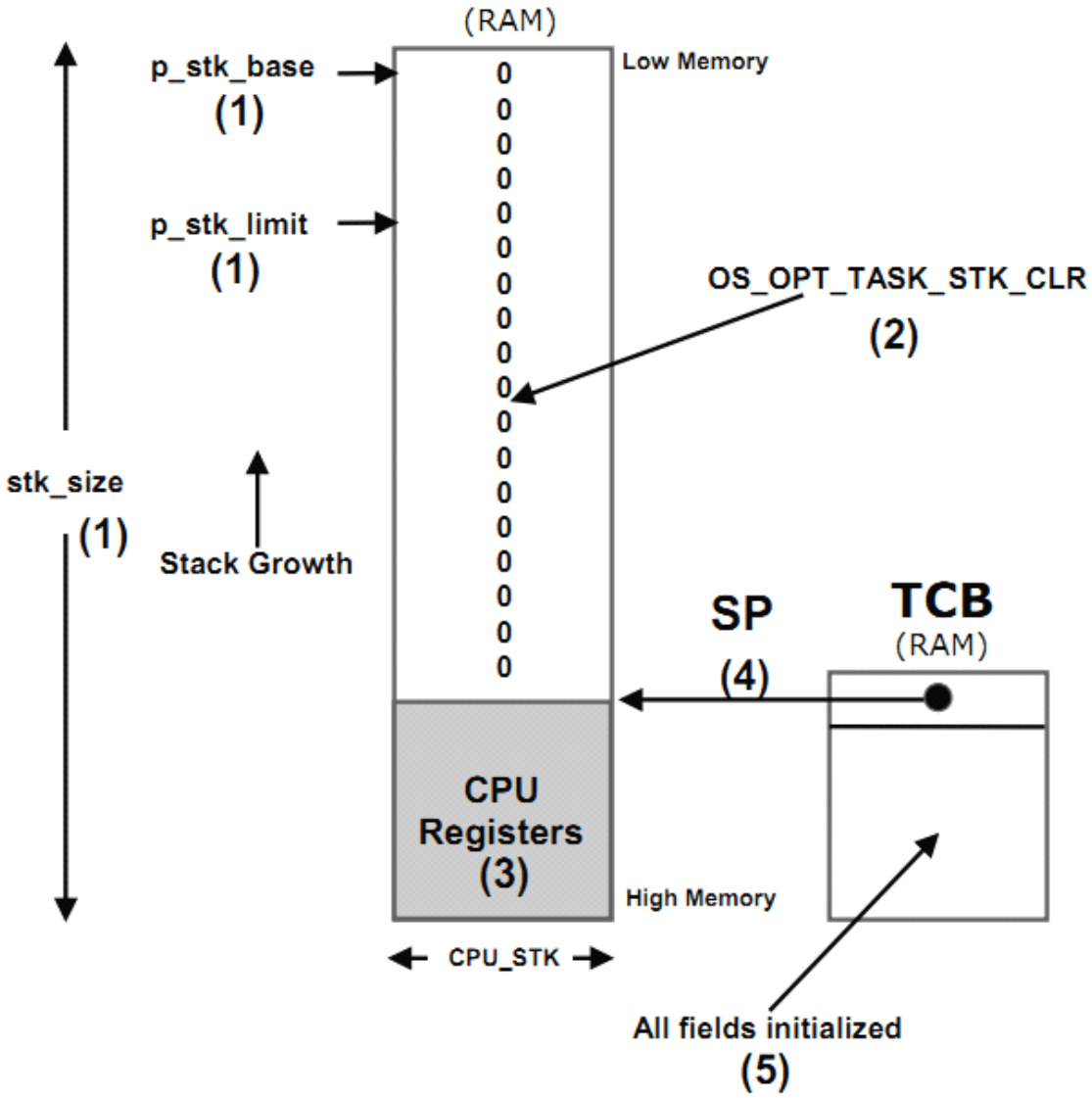


Figure 5-1 **OSTaskCreate() initializes the task's TCB and stack**

F5-1（1）调用 OSTaskCreate()时，参数为：堆栈的基地址（p_stk_base），堆栈的增长限制，堆栈大小等。

F5-1（2）若 OSTaskCreate()中的第 12 个参数设置为 OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR，uC/OS-III 将会初始化堆栈内容为全 0。

F5-1（3）uC/OS-III 将复制 CPU 寄存器内容并有序地存入任务堆栈的顶部。这使得上下文切换易于实现。

F5-1（4）堆栈指针 SP 存于任务的 TCB 中。

F5-1（5）TCB 中其它的字段会被赋值：任务优先级、任务名、任务状态、内部消息队列、内部信号量等。

接下来，会调用一些定义在 OS_CPU_C.C 中的函数如 OSTaskCreateHook()，TCB 中有指针指向这个函数，用于扩展应用。例如，可以打印最新创建的 TCB 内容到某终端（利于调试）。

然后该任务被放到就绪列表（详见第六章"就绪列表"）。uC/OS-III 调用调度器，并切换到优先级最高的任务。

任务可以调用 uC/OS-III 提供的函数。特别的，一个任务可以创建其它任务（调用 OSTaskCreate()）、停止或者恢复其它任务(调用 OSTaskSuspned()和 OSTaskResume())、提交信号量到其它任务、发送消息到其它任务、提供共享资源等。换句话说，任务不是只被限制于“等待事件”。

图 5-2 展示了任务与其它资源的典型关系。

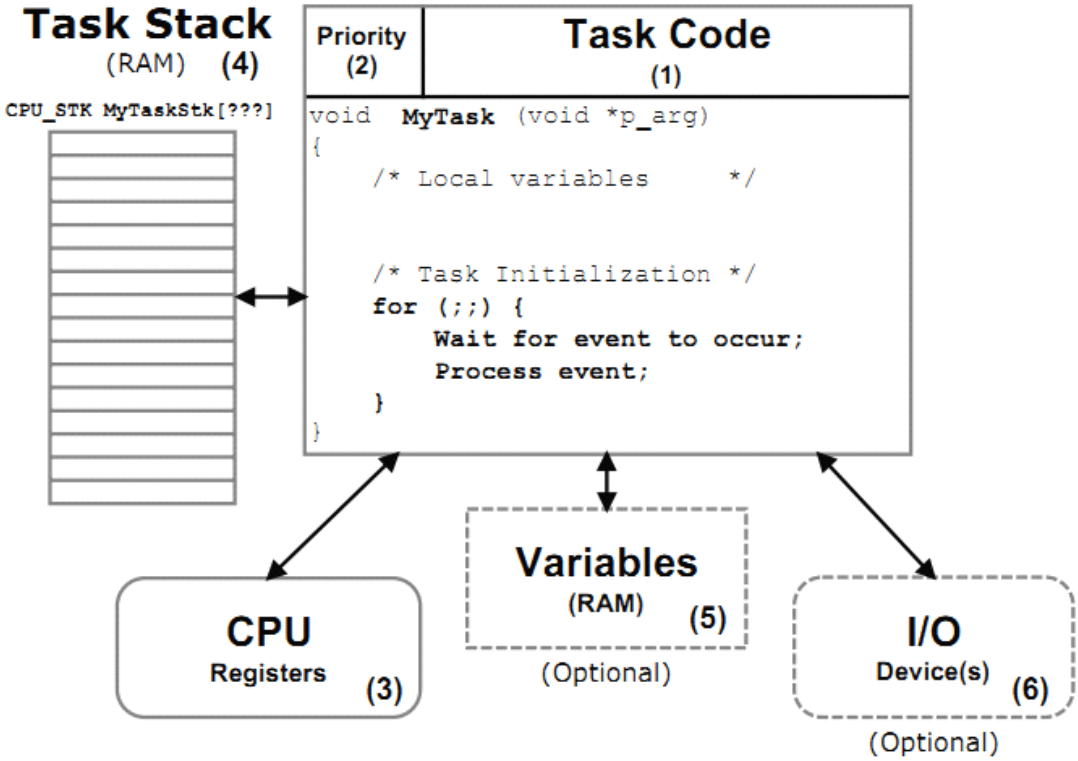


Figure 5-2 **Tasks interact with resources**

F5-2（1）任务最重要的部分是它的代码。正如前面提到的，任务的代码看起来像是个 C 函数，除了它实现无限循环的方式。另外，任务不允许有返回值。

F5-2（2）每个任务都需要被设定一个优先级。uC/OS-III 的工作是决定哪个任务应该占用 CPU。一般的，uC/OS-III 选择就绪队列中优先级最高的任务运行。

在 uC/OS-III 中，数值越小优先级越高。

uC/OS-III 允许用户在编译时配置优先级的范围（详见 OS_CFG.H 文件中的 OS_PRIO_MAX）。另外，uC/OS-III 允许任务具有相同优先级且对该任务数无限制。例如，uC/OS-III 可以被配置为拥有 64 种不同优先级，并可以设置几十个具有相同优先级的任务。（详见 5-1“设置任务优先级”）

F5-2（3）每个任务对 CPU 寄存器都会有自己设置。如果一个任务被运行，那么它就会占用了实际的 CPU。

F5-2（4）因为 uC/OS-III 是一个抢占式内核，每个任务都需要有自己的堆栈空间。堆栈驻留于 RAM 并经常用于保存变量、函数、嵌套的 ISR 地址等。

堆栈空间可以被分配为静态的（在编译时分配）或者是动态的（运行时分配）。定义静态的堆栈如下所示，定义是在任务之外的。

```
static  CPU_STK MyTaskStk[???];
```

或者

```
CPU_STK MyTaskStk[???];
```

注意“？？？”是堆栈的大小，依赖于任务的需求。通过 C 编译器提供的堆管理功能（malloc()）堆栈空间可以被动态分配。如下所示。但是，必须关注存储碎片。多次创建和删除任务后，内存可能有很多存储碎片而不足以再分配给任务了。在嵌入式系统中动态地分配堆栈是被允许的，但是，一旦堆栈被动态分配，它就不能被回收。换句话说，对于有些不需要被删除的任务，动态分配它们的堆栈是一种很好的解决方法。

```
void SomeCode (void)
{
    CPU_STK *p_stk;
    :
    :
    p_stk = (CPU_STK *)malloc(stk_size);
    if (p_stk != (CPU_STK *)0) {
        Create the task and pass it "p_stk" as the base address of the stack;
    }
    :
    :
}
```

F5-2（5）任务也可以访问全局变量。然而，因为 uC/OS-III 是抢占式内核，所有必须注意多个任务同时访问全局变量的情况。幸运的是，uC/OS-III 提供了一些功能管理这些共享资源如（信号量、mutex 等）。

F5-2（6）任务也可以访问一个或多个 IO 设备（或者外设）。

5-1 设置任务优先级

有些时候任务的优先级是显而易见的。例如，嵌入式系统中的重要的应用应该被设置为高优先级，一些显示操作就应该被设置为低优先级。然而，由于实时系统的复杂性，在大多数情况下任务的优先级是不能被事先确定的。多数系统中，不是所有的任务都是重要的，不重要的任务应该被设置为低优先级。

5-2 堆栈空间大小的确定

堆栈的大小取决于该任务的需求。设定堆栈大小时，你就需要考虑：所有可能被堆栈调用的函数及其函数的嵌套层数，相关局部变量的大小，中断服务程序所需要的空间。另外，堆栈还需存入 CPU 寄存器，如果处理器有浮点数单元 FPU 寄存器的话还需存入 FPU 寄存器。嵌入式系统的潜规则，避免写递归函数。

可以人工地计算出任务需要的堆栈空间大小，逐级嵌套所有可能被调用的函数，添加被调用函数中所有的参数，添加上下文切换时的 CPU 寄存器空间，添加切换到中断时所需的 CPU 寄存器空间（假如 CPU 没有独立的堆栈用于处理中断），添加处理 ISRs 所需的堆栈空间。把上述的全部相加，得到的值定义为最小的需求空间。因为我们不可能计算出精确的堆栈空间。通常是再乘以 1.5 到 2.0 以确保任务的安全运行。这个计算是假定在任务所有的执行路线都是已知的情况下的，可这是不太可能的。比如说，如果调用 printf()或者其它的函数，printf()函数所需要的空间是很难测得或者说就是不可能知道的。在这种情况下，刚开始时给任务设置一个较大的堆栈空间并监测运行时堆栈空间的实际使用量。{一般都是通过这种情况测得堆栈的最大使用量，然后乘 1.5 左右作为堆栈空间大小}

编译器/汇编器提供的的连接映像信息是很有用的。在每个函数中，连接映像提供了函数在最坏情况下堆栈使用量。这个功能能够比较精确地评估每个任务所需的堆栈空间。当然，还需添加一个 CPU 寄存器空间、再一个 CPU 寄存器空间（如果 CPU 没有独立的空间处理

ISR 的话)、再加上这些 ISR 需要的堆栈空间。最后，为了保证可靠还有再乘以一个 1.5 左右的值。

在产品的开发和测试阶段，通常在运行时用调试器测量堆栈的使用情况。

5-3 检测任务堆栈的溢出

使用 MMU 或 MPU

如果处理器有 MMU 或者 MPU，检测堆栈是否溢出是非常简单的。MMU 和 MPU 是 CPU 边上的特殊硬件设施，可以检测非法访问，如果任务企图访问未被允许的内存空间的话，就会产生警告。但设置 MMU 和 MPU 不是本书所要介绍的。

堆栈溢出检测寄存器

一些处理器中有一些堆栈溢出检测寄存器。当 CPU 的堆栈指针小于（或大于，取决于堆栈的生长方向）设置于这个寄存器的值时。异常产生时异常处理程序就要确保未允许空间代码的安全(可能会发送警告给用户)。OS_TCB 中的.StkLimitPtr 就是为这种目的设置的如图 5-3.堆栈必须被分配足够大的空间。在大多数情况下，该指针的值可以设置接近于&MYTaskStk[0]。

{假定堆栈是从高地址往低地址生长的}

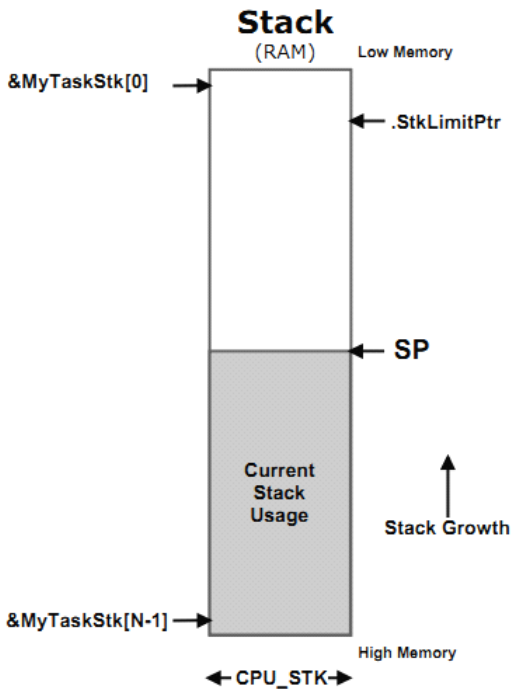


Figure 5-3 **Hardware detection of stack overflows**

在此提醒， StkLimitPtr 的值取决于 OSTaskCreate() 的参数 "stk_limit"如下所示：

```
OS_TCB  MtTaskTCB;
CPU_STK MyTaskStk[1000];


OSTaskCreate(&MyTaskTCB,
             "MyTaskName",
              MyTask,
             &MyTaskArg,
              MyPrio,
             &MyTaskStk[0],   /* Stack base address                                          */
               100,           /* Set .StkLimitPtr to trigger exception at stack usage > 90% */
              1000,           /* Total stack size (in CPU_STK elements)                      */
              MyTaskQSize,
              MyTaskTimeQuanta,
              (void *)0,
              MY_TASK_OPT,
             &err);
```

当然，当 uC/OS-III 执行上下文切换的时候，StkLimitPtr 的值会被改变。这是非常棘手的，因为当堆栈溢出检测寄存器的值被改变时需将它首先被设置为 NULL，然后改变 CPU 的堆栈指针，最后把 TCB

的 stkLimitPtr 值存到堆栈检测寄存器中。为何？如果不按照这个过程，那么在堆栈指针或堆栈溢出检测寄存器改变时发生了中断就麻烦了。通过首先改变堆栈溢出检测寄存器指向一个地址（通常设为 NULL）从而让堆栈指针总是有效的。注意，我是在假定堆栈是从高地址往低地址生长的，堆栈生长方向从低往高的道理也是一样的。

基于软件的堆栈溢出检测

当 uC/OS-III 从一个任务切换到另一个任务的时候，它会调用一个 hook 函数 OSTaskSwHook()，它允许用户扩展上下文切换时的功能。所以，如果处理器没有硬件支持溢出检测功能，就可以在该 hook 函数中添加代码软件模拟该功能。特别的，在切换到任务 B 前，代码会检测将要被载入 CPU 堆栈指针的值是否超出该任务 B 的 TCB 中 StkLimitPtr 限制。因为软件不能在溢出时就迅速地做出反应，所以应该设置 StkLimitPtr 的值尽可能远离&MyTaskStk[0]，保证有足够的溢出缓冲。如图 5-4。软件检测不会像硬件检测那样有效，但也可以防止堆栈溢出。

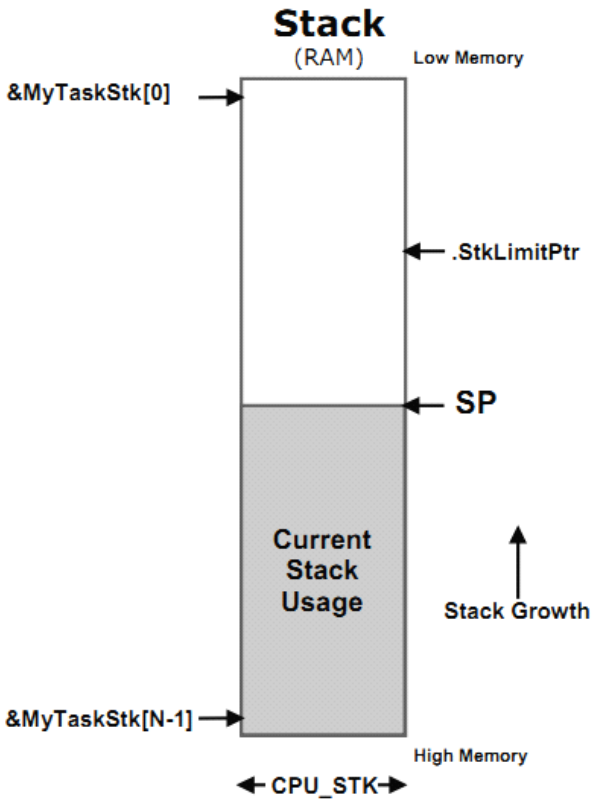


Figure 5-4 **Software detection of stack overflows, monitoring .StkLimitPtr**

计算空闲堆栈空间

另一种防止堆栈溢出的方法是分配的空间远大于可能需要用到的。然后，调试器检测在运行时任务所用的最大堆栈空间。这是易于实现的。首先，当任务创建时其堆栈被清零。然后，通过一个低优先级任务，计算该任务整个堆栈中值为 0 的内存大小。如果发现都不为 0，那么就需要扩展堆栈的大小。然后，调整堆栈为的相应大小。这是一种非常有效的方法。注意的是，程序需用运行很长的时间以让堆栈达到其需要的最大值。详见图 5-5。uC/OS-III 提供了一个函数 OSTaskStkChk()用于实现这个计算功能。事实上，这个函数被 OS_StatTask()用于统计每个任务的堆栈使用情况。

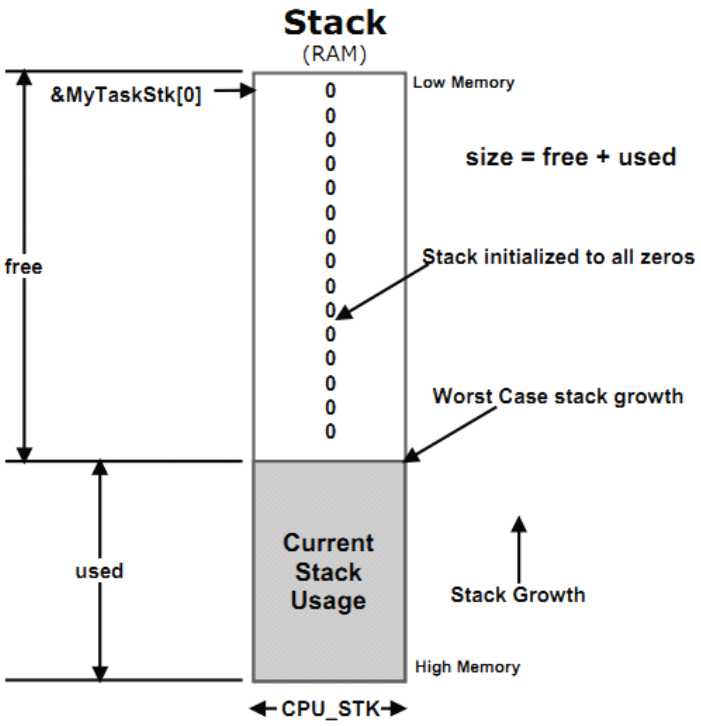


Figure 5-5 **Software detection of stack overflows, walking the stack**

5-4 任务管理服务

uC/OS-III 提供了很多与任务相关的函数。这些函数可以在 OS_TASK.C 中找到，它们都以以 OSTask???()形式命名的。

分组	函数
普通的	OSTaskCteate() OSTaskDel() OSTaskChangePrio() OSTaskRegSet() OSTaskRegGet() OSTaskSuspend() OSTaskResume() OSTaskTimeQuantaSet()
标记任务	OSTaskSemPend() OSTaskSemPost() OSTaskSemPendAbort()
给任务发送消息	OSTaskQPend() OSTaskQPost() OSTaskQPendAbort() OSTaskQFlush()

uC/OS-III 任务相关函数的介绍详见附录 A。

5-5 内部任务管理

5-5-1 任务状态

从用户的观点来看，任务可以是有 5 种状态，见图 5-6。展示了任务状态间的转换关系。

{休眠状态，就绪状态，运行状态，挂起状态，中断状态}

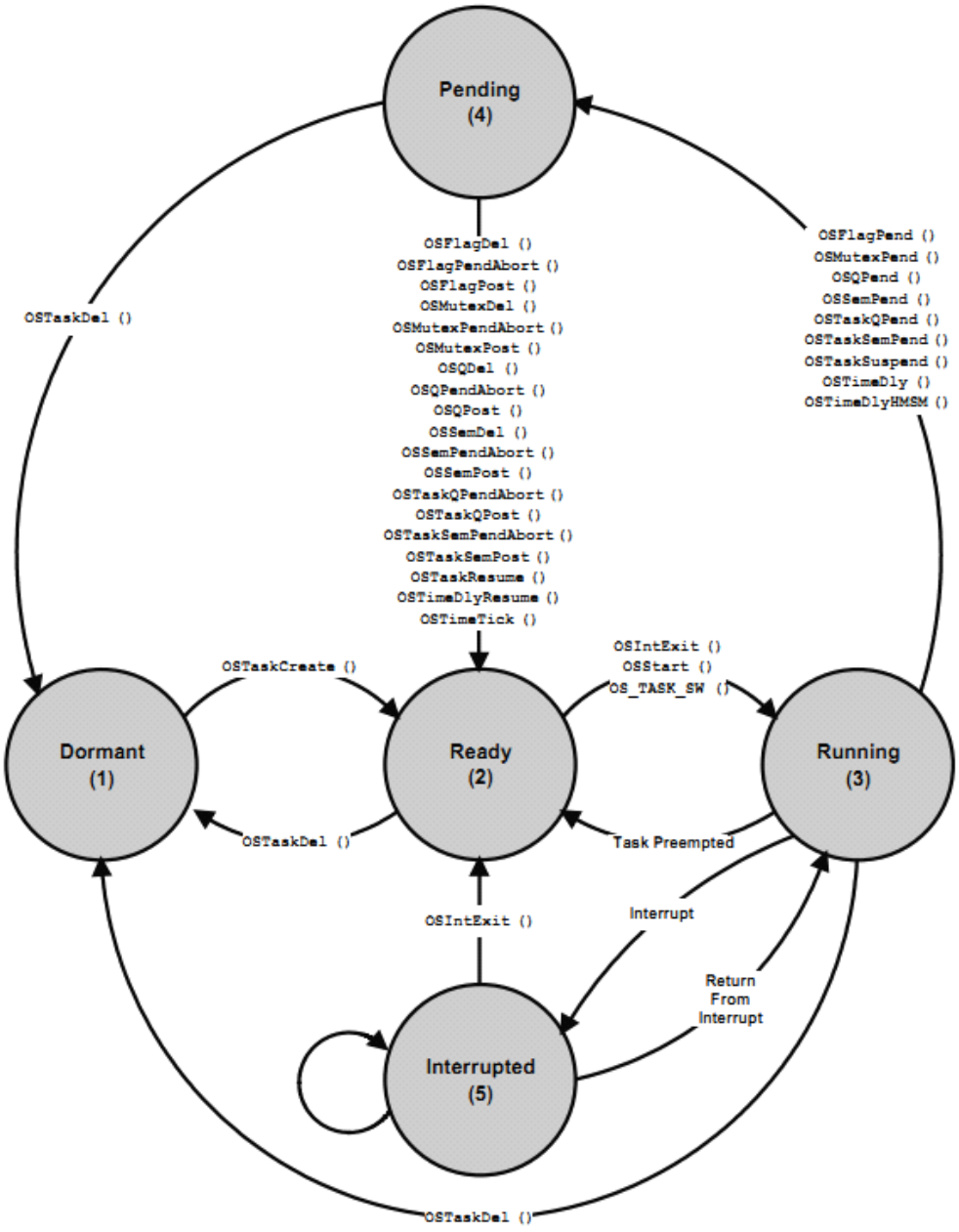


Figure 5-6 **Five basic states of a task**

F5-6（1）处于休眠状态的任务驻留于内存但未被 uC/OS-III 使能。

通过调用 OSTaskCreate()函数 uC/OS-III 创建任务。任务代码是存在于 ROM 的。但需要用 OSTaskCreate()函数通知 uC/OS-III 关于任务的相关信息。

如果任务的使命完成了，就要调用 OSTaskDel()删除该任务。OSTaskDel()实际上不是删除任务的代码，只是让任务不再具有使用 CPU 的资格而已。

F5-6（2）就绪状态的任务根据优先级有序地排列于就绪列表中。就绪列表中对就绪任务的个数没有限制。

F5-6（3）正在运行的任务被置为运行状态。在单 CPU 中，任何时刻只能有一个任务被运行。

当应用程序调用 OSStart()或者调用 OSIntExit()或者调用 OS_TASK_SW()时 uC/OS-III 从就绪队列中选择优先级最高的任务去运行。

正如前面所提到的，有些时候任务必须等待某些事件发生，若事件还未发生时，任务就会被设置为挂起状态。

F5-6（4）挂起状态的任务被放置在挂起列表中以表明任务在等待某些事件的发生。等待的时候，任务是不会占用 CPU 的。事件发生时，该任务会被放到就绪队列中。在这种情况下，正在运行的任务可能会被抢占（被放回就绪列表），并由 uC/OS-III 选择优先级最高的任务去运行。换句话说，如果新的任务优先级最高，那么它就会被立即运行。

请注意，调用 OSTaskSuspend()会任务无条件地停止运行。有些时候调用 OSTaskSuspend()不是为了等待某个事件的发生，而是等待另一个任务调用 OSTaskResume()函数恢复这个任务。

F5-6（5）若中断发生，中断会挂起正在执行的任务并去处理 ISR。ISR 中可能有某些任务等待的事件。一般来说，中断用来通知任务某些事件的发生，并让在任务级处理实际的响应操作。ISR 程序越短越好，实际响应中断的操作应该被设置在任务级以便能让 uC/OS-III 管理这些操作。ISR 中只允许调用一些提交函数(OSFlagPost()，OSQPost()，OSSemPost()，OSTaskQPost()，OSTaskSemPost())，除了 OSMutexPost()。因为 mutex 只允许在任务级被修改。

正如图所示的那样，一个中断可以被另一个中断所抢占。这叫做中断嵌套，大多数处理器支持中断嵌套。然而，如果管理不当，中断嵌套是很容易引起堆栈溢出的。

uC/OS-III 一直追踪着任务的状态如图 5-7。事实上，这些都是以一个变量的形式保存在每个任务的 TCB 中。图小括号中的数值表示着任务的状态，每个任务都可以有 8 种状态。（详见 OS.H，OS_TASK_STATE_???）

图中不包括休眠态，因为 uC/OS-III 不支持休眠态。在这里，中断以及中断的嵌套会有更深的讲解。

这个状态图有助于理解函数与任务状态间的关系。

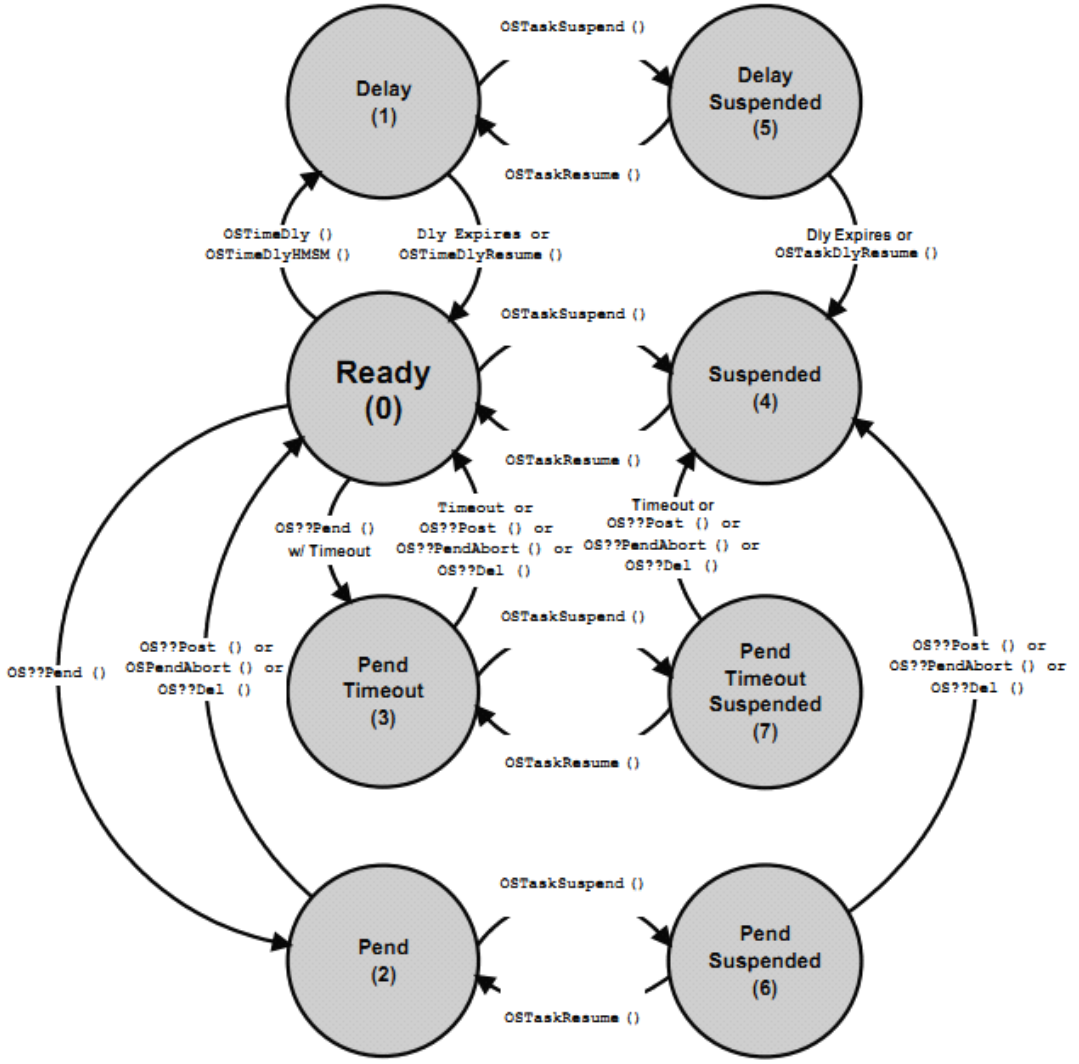


Figure 5-7 **µC/OS-III's internal task state machine**

F5-7（0）状态 0 表示任务已经就绪。每个任务在被运行之前都必须处于就绪状态。

F5-7（1）任务可以通过调用 OSTimeDly()或者 OSTimeDlyHMSM()等待期满。当期满或者延时删除时（通过调用 OSTimeDlyResume()），任务会转为就绪状态。

F5-7（2）任务可以通过调用挂起函数（OSFlagPend()，OSMutexPend()，OSQPend，OSSemPend，OSTaskQPend()，OSTaskSemPend()）等待某事件的发生。当事件发生时、该任务被删除、或者被另一个任务取消等待时，等待停止。

F5-7（3）如前面所说，任务可以等待事件发生。但任务也可以被设置等待多少时间。如果在这段时间内事件没有发生，任务也会被设为就绪状态，并通知这个任务是等待超时而被挂起的。

{挂起函数都有一个关于函数执行结果错误代号，可以查看这个代号知道任务是因何被就绪的}

F5-7（4）任务暂停自己或者被其他任务暂停（通过调用 OSTaskSuspend()）。暂停中的任务只能通过调用 OSTaskResume()被恢复。

F5-7（5）一个延时中的任务也可以被其它任务设置为停止。在这种情况下，效果会被叠加。换句话说，延时需被执行、停止状态需被解除。该任务才会被执行。

F5-7（6）一个挂起状态中的任务也可能被其它任务设置为停止。同样的，效果会被叠加。事件发生且停止状态被移除后，任务才会被执行。

F5-7（7）任务可以等待事件的发生，但可以给它设定一个期限。同样的，它也可能被设为停止，效果是叠加的。除非移除停止状态并事件发生或等待事件超时，任务才会被执行。

5-5-2 任务控制块 TCB

任务控制块是被 uC/OS-III 用于维护任务的一个结构体。每个任务都必须有自己的 TCB。uC/OS-III 在 RAM 中分配 TCB。当调用 uC/OS-III 提供的与任务相关的函数（以 OSTask???()形式命名）时，任务的 TCB 地址需会被提供给该函数。TCB 的结构定义于 OS.H 中，如列表 5-3 所示（在 OS.H 中代码是有注释的）。TCB 中的一些变量可以根据具体应用进行裁剪。

用户程序不应该访问这些变量（尤其不能更改它们）。换句话说，TCB 中的变量只能被 uC/OS-III 访问。

```
struct os_tcb {
    CPU_STK            *StkPtr;
    void               *ExtPtr;
    CPU_STK            *StkLimitPtr;
    OS_TCB             *NextPtr;
    OS_TCB             *PrevPtr;
    OS_TCB             *TickNextPtr;
    OS_TCB             *TickPrevPtr;
    OS_TICK_SPOKE      *TickSpokePtr;
    OS_CHAR            *NamePtr;
    CPU_STK            *StkBasePtr;
    OS_TASK_PTR         TaskEntryAddr;
    void               *TaskEntryArg;
    OS_PEND_DATA       *PendDataTblPtr;
    OS_OBJ_QTY          PendDataEntries;
    CPU_TS              TS;
    void               *MsgPtr;
    OS_MSG_SIZE         MsgSize;
    OS_MSG_Q            MsgQ;
    CPU_TS              MsgQPendTime;
    CPU_TS              MsgQPendTimeMax;
    OS_FLAGS            FlagsPend;
    OS_OPT              FlagsOpt;
    OS_FLAGS            FlagsRdy;
```

```
    OS_REG              RegTbl[OS_TASK_REG_TBL_SIZE];
    OS_SEM_CTR          SemCtr;
    CPU_TS              SemPendTime;
    CPU_TS              SemPendTimeMax;
    OS_NESTING_CTR      SuspendCtr;
    CPU_STK_SIZE        StkSize;
    CPU_STK_SIZE        StkUsed;
    CPU_STK_SIZE        StkFree;
    OS_OPT              Opt;
    OS_TICK             TickCtrPrev;
    OS_TICK             TickCtrMatch;
    OS_TICK             TickRemain;
    OS_TICK             TimeQuanta;
    OS_TICK             TimeQuantaCtr;
    OS_CPU_USAGE        CPUUsage;
    OS_CTX_SW_CTR       CtxSwCtr;
    CPU_TS              CyclesDelta;
    CPU_TS              CyclesStart;
    OS_CYCLES           CyclesTotal;
    CPU_TS              IntDisTimeMax;
    CPU                 SchedLockTimeMax;
    OS_STATE            PendOn;
    OS_STATUS           PendStatus;
    OS_STATE            TaskState;
    OS_PRIO             Prio;
    OS_TCB              DbgNextPtr;
    OS_TCB              DbgPrevPtr;
    CPU_CHAR            DbgNamePtr;
};
```

Listing 5-3 **OS_TCB Data Structure**

.StrPtr

这个变量中包含了指向当前任务堆栈的指针。uC/OS-III 允许每个任务有自己的堆栈且对该堆栈的大小没有限制。.StkPtr 可能是 OS_TCB 数据类型中唯一一个需要被汇编语言所访问的（用于上下文切换部分的代码）。因此，这个变量被放在结构体中的第一个，让汇编语言更容易地访问到它（使它在 TCB 结构体中的偏移量为 0）。

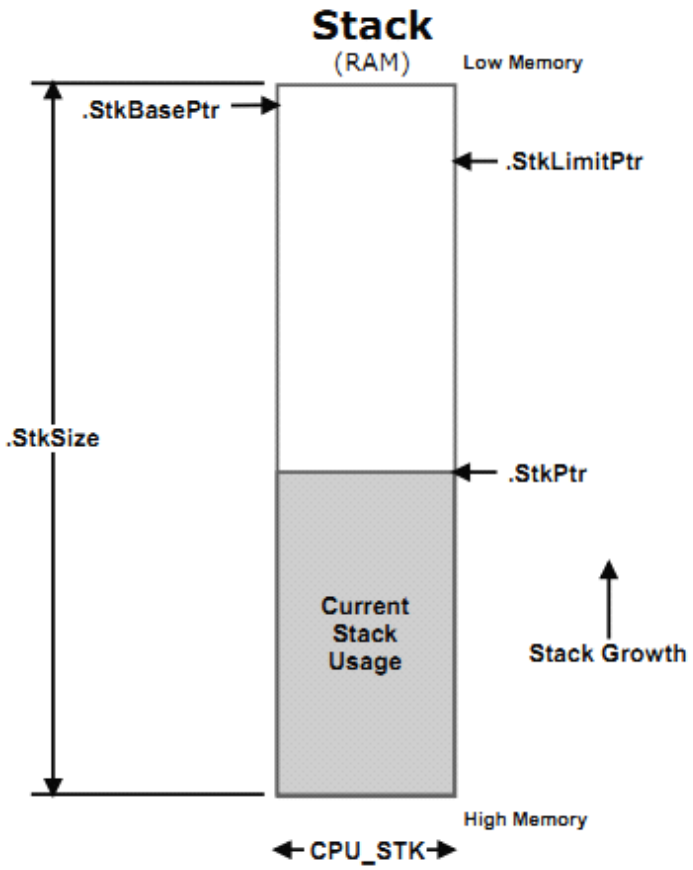
.ExtPtr

这个变量中定义了指向用户用于扩展 TCB（如果需要）的指针。这个指针也是易于被汇编语言访问的。

.StkLimitPtr

这个变量中保存了堆栈增长时的限制地址，它是在调用 OSTaskCreate()时传递的参数"stk_limit"。有些处理器有硬件寄存器可以自动地检测并确保堆栈不发生溢出，如果处理器没有这些硬件设施，堆栈检测可以用软件模拟。然而，软件模拟不如硬件可靠。如果这个功能没有被用到，那么在调用 OSTaskCreate()时可以设置"stk_limit"为 0。

详见 5-3。



.NextPtr 和**.PrevPtr**

这些指针用于就绪队列中双向链表。双向链表可以让 TCB 在列表中能更快地被插入或者删除。

.TickNextPtr 和**.TickPrevPtr**

这些指针用于挂起队列中双向链表。双向链表可以让 TCB 在列表中能更快地被插入或者删除。

.TickSpokePtr

这个指针用于表示时基轮转的轮辐。时基轮转会在第九章“中断管理”中详细介绍。

.NamePtr

这个指针存放了任务的名字。有名字的任务非常有助于调试，因为这样能友好地显示每个任务对应的 TCB 地址。该名字的字符串存于 ROM（如果以常量命名）或者 RAM。

.StkBasePtr

{任务堆栈是由高地址向低地址生长}

这个指针指向了任务堆栈的基地址。任务的基地址通常是任务所驻留堆栈区的最低内存地址。任务堆栈的定义：

CPU_STK　　MyTaskStk[???];

定义堆栈空间必须用 CPU_STK 数据类型,"？？？"是要定义的堆栈的大小。它的基地址通常是&MyTaskStk[0]；

.TaskEntryAddr

这个变量中包含了任务代码的入口地址，正如前面提到的，任务用如下方式申明，

```
void MyTask(void *p_arg);
```

.TaskEntryAtg

这个变量是当任务第一次运行时传递给任务的参数。正如上面提到的，这个变量值会传给 p_arg。

.PendDataTblPtr

uC/OS-III 允许任务同时挂起多个信号量和消息队列。这个指针指向了包含这些被挂起对象的表。

.PendDataEntries

这个变量与.PendDataTblPtr 一起工作，表示在同一时刻某任务等待的事件数。

.TS

这个变量存储了任务所等待事件出现的时间戳，当任务恢复执行时，时间戳会被返回给任务。

.MsgPtr

当有消息发送给任务时，这个变量保存了该消息的地址。在编译时使能了消息队列服务(OS_CFG.H 中设置 OS_CFG_Q_EN 为 1)或使能了任务内建消息队列（OS_CFG.H 中设置 OS_CFG_TASK_Q_EN 为 1）。这个变量才出现在 TCB 中，不然就会被裁减。

.MsgSize

当有消息发送给任务时，这个变量保存了消息的大小（以字节为单位）。这个变量仅出现在 TCB 中，如果消息队列服务(在 OS_CFG.H 中设置 OS_CFG_Q_EN 为 1)或者任务队列服务（在 OS_CFG.H 中设置 OS_CFG_TASK_Q_EN 为 1）编译时被使能的话。

.MsgQ

uC/OS-III 允许任务或 ISR 直接发送消息给任务。实际上也有消息队列内建于每个任务的 TCB 中了（假设编译时在 OS_CFG.H 中设置 OS_CFG_TASK_Q_EN 为 1）。.MsgQ 被用于 OSTaskQ???()函数。

.MsgQPendTime

保存了消息从创建到被接收所需的时间。当 OSTaskQPost()被调用时，读取当前的时间戳并保存在消息中。当 OSTaskQPend()接收到这个消息时，读取当前的时间戳，并与消息被提交时的时间戳相减，差值保存于这个变量中。调试器或者 uC/Probe 可以观察这个变量的值。当编译时设置 OS_CFG.H 中的 OS_CFG_TASK_PROFILE_EN 为 1 时，这个变量才有效。

.MsgQPendTimeMax

这个变量中保存了消息到达所用时间的最大值，它是.MsgQPendTime 的峰值。这个值可以被 OSStatReset()复位。在编译时设置 OS_CFG.H 中的 OS_CFG_TASK_PROFILE_EN 为 1 时，这个变量才有效。

.FlagsPend

当任务等待事件标志组，这个变量保存了任务所等待的标志位。当编译时设置了 OS_CFG.H 中的 OS_CFG_FLAG_EN 为 1 时这个变量才有效。

.FlagsOpt

当任务等待事件标志组，这个变量保存了任务所等待事件标志组的类型。在编译时设置了 OS_CFG.H 中的 OC_CFG_FLAG_EN 为 1 时这个变量才有效。

.FlagsRdy

这个变量保存了已经被提交的事件标志组（任务所等待的），换句话说，它让任务知道是哪个事件标志组让任务就绪的。在编译时设置了 OS_CFG.H 中的 OS_CFG_FLAG_EN 为 1 时这个变量才有效。

.RegTbl[]

这个数组中包含了任务的"寄存器"，不同于 CPU 寄存器。任务寄存器用于存储任务 ID、软件错误等。需要注意的是这个数组的数据类型是 OS_REG，它是在编译时定义的。所有的任务寄存器都必须用这种数据类型。在编译时设置 OS_CFG.H 中的任务寄存器数组大小宏 OS_CFG_TASK_REG_TBL_SIZE 大于 0 时这个数组才会有效。

.SemCtr

这个变量保存了信号量的计数值。每个任务都有其的内建信号量。ISR 或其它任务可以通过信号量标记这个任务。因此.SemCtr 用于记录该任务被标记了几次。.SemCtr 在 OSTaskSem???()中被用到。

.SemPendTime

中保存着信号量从产生到被接收所用的时间。当 OSTaskSemPost()被调用，当前的时间戳被读取并保存于信号量中。当 OSTaskSemPend()函数收到信号量时，当前的时间戳被读取并与之前的时间戳的差值存于这个变量。调试器或者 uC/Probe 可以观察这个变量值。当设置 OS_CFG.H 中的 OS_CFG_TASK_PROFILE_EN 为 1 时变量才会有效。

.SemPendTimeMax

保存了信号量从产生到被接收所用时间的最大值。它是.SemPendTime 是峰值。可以调用 OSStatReset()复位这个变量。当设置 OS_CFG.H 中的 OS_CFG_TASK_PROFILE_EN 为 1 时变量才会有效。

.SuspendCtr

这个变量被 OSTaskSuspend()和 OSTaskResume()使用，用于记录任务被停止的次数。任务停止可以被嵌套，当.SuspendCtr 为 0 时，任务可以被执行。当编译时设置 OS_CFG.H 中的 OS_CFG_TASK_SUSPEND_EN 为 1 时这个变量才会有效。

.StkSize

这个变量中保存了堆栈的大小（以 CPU_STK 为数据类型）。堆栈的定义如下

CPU_STK MyTaskStk[???];

.StkSize 就是???的值。

.StkUsed 和.StkFree

在运行时，uC/OS-III 可以计算出堆栈的实际使用量和空余量，这是通过调用 OSTaskStkChk()实现的。堆栈使用量计算是假定堆栈在创建时被初始化的情况下的。换句话说，当 OS_TASK_OPT_STK_CLR 和 OS_TASK_OPT_STK_CHK 被使能时，OSTaskCreate()在任务创建时会先初始化任务堆栈的 RAM。{一般初始化都是将堆栈全部清零的}

uC/OS-III 提供了一个叫做 OS_StatTask()的任务，它可以检测每一个任务的堆栈使用情况。OS_StatTask()通常被设置于低优先级，所以它可以在 CPU 空闲时运行。OS_StatTask()将检测结果保存于每个任务的 TCB 中。保存了任务堆栈的最大使用量（以字节形式的计算值）和空余量。当设置 OS_CFG.H 中的 OS_CFG_STAT_TASK_STK_CHK_EN 为 1 时这两个变量才有效。

.Opt

当任务创建时传递给 OSTaskCreate()的参数。它定义任务的附加功能。

.TickCtrPrev

当 OSTimeDly()选择 OS_OPT_TIME_PERIODIC 形式时，该变量为 OSTickCtr 的初值。

.TickCtrMatch

当任务被延时一段时间，或者因等待事件而设置时限。任务就会被放到挂起队列中。在这个队列中，任务等待自己的 TickCtrMatch 值与时基计数值（OSTickCtr）相匹配。匹配发生时，任务就会被移出挂起队列。

.TickRemain

这个变量中保存了任务到时的剩余时间值，它在 OS_TickTask() 中被计算。调试时这个变量是很有用的。

.TimeQuanta 和.TimeQuantaCtr

这两个变量用于时间切片，当多个就绪任务有相同的优先级时，.TimeQuanta 决定了时间片长度（多少个时基）。.TimeQuantaCtr 中保存了当前时间片的剩余长度。在任务切换开始时将.TimeQuanta 的值载入.TimeQuantaCtr。

.CPUUsage

保存了 CPU 的使用率（0 到 100%），它是被 OS_StatTask()计算出来的。编译时设置了 OS_CFG.H 中的 OS_CFG_TASK_PROFILE_EN 为 1 时该变量有效。

.CtxSwCtr

保存了该任务被执行的次数。调试时可以查看这个变量的值。编译时设置 OS_CFG_TASK_PROFILE_EN 为 1 时这个变量有效。

.CyclesDelta

这上下文切换时被计算，它保存了当前时间戳与 CyclesStart 的差值。调试时可以通过它知道该任务的执行时间。当设置 OS_CFG_TASK_PROFILE_EN 为 1 时该变量有效。

.CyclesStart

这个变量用于测量任务的执行时间，.CyclesStart 在上下文切换时被更新。它保存了任务切换时的时间戳(通过调用 OS_TS_GET()获得)。当编译时设置 OS_CFG_TASK_PROFILE_EN 为 1 时变量有效。

.CyclesTotal

这个变量是 CyclesDelta 的累加，所以它包含了该任务被执行的总时间。这个变量被定义为 64 位防止溢出。使用一个 64 位的变量可以确保 1GHz 的 CPU 大约运行 600 年才溢出。当然，它要求编译器支持 64 位的数据类型。

.IntDisTimeMax

这个变量中保存了该任务关中断的最大时间。使用这个的前提是 uC/CPU 支持关中断时间的测量。当设置了 OS_CFG.H 中的 OS_CFG_TASK_PROFILE_EN 为 1 且定义了 CPU_CFG.H 中的 CPU_CFG_TIME_MEAS_INT_DIS_EN 时变量才有效。

{在 CPU_CFG.H 中我只找到 CPU_CFG_INT_DIS_MEAS_EN。估计是后来改名字了但手册上更新过来}

.SchedLockTimeMax

保存了该任务锁调度器的最大时间。当设置了 OS_CFG.H 中的 OS_CFG_TASK_PROFILE_EN 为 1 且设置 OS_CFG_SCHED_LOCK_TIME_MEAS_EN 为 1 时变量才有效。

.PendOn

该变量的值取决于任务因何被挂起（详见 OS.H 中的 OS_TASK_PEND_ON_???）。

.PendStatus

这个变量保存了任务被挂起后的状态，（详见 OS.H 中的 OS_STATUS_PEND_???)。

{PendOn 和 TaskState 都好理解，这个包括 4 种状态，正常挂起、挂起取消、挂起对象被删除、挂起超时。详见 OS.H}

.TaskState

这个变量保存了任务当前的状态，包括 8 种状态（详见 OS.H 中的 OS_TASK_STATE_???）

.Prio

它保存了任务的优先级， 该值介于 0 到 OS_CFG_PRIO_MAX-1 之间。事实上，空闲任务需独占优先级 OS_CFG_PRIO_MAX-1。

.DbgNextPtr

该变量是一个指针，在双向 TCB 列表中，它指向下一个 TCB。通过 OSTaskCreate()函数 uC/OS-III 将 TCB 放入该列表中。当设置了 OS_CFG.H 中的 OS_CFG_DBG_EN 为 1 时该变量才有效。

.DbgPrevPtr

该变量是一个指针，在双向 TCB 列表中，它指向上一个 TCB。当设置了 OS_CFG.H 中的 OS_CFG_DBG_EN 为 1 时该变量才有效。

.DbgNamePtr

该变量是一个指针，当任务在等待信号量、事件标志组、mutex、消息队列时，它指向目标对象的名字。调试时很有用，当设置了 OS_CFG.H 中的 OS_CFG_DBG_EN 为 1 时变量才有效。

5-6 内部任务

在 uC/OS-III 初始化的时候，它会创建至少 2 个内部的任务(OS_IdleTask()和 OS_TickTask())，3 个可选择的任务（OS_StatTask(), OS_TmrTaks()，OS_IntQTask()）。这些可选择的任务在编译时由 OS_CFG.H 中的配置决定。

5-6-1 空闲任务 OS_IdleTask()

OS_IdleTask()是 uC/OS-III 最先创建的任务。它的优先级通常是 OS_CFG_PRIO_MAX-1。事实上，为了安全，它应该独占这个优先级。在其他任务创建的时候，OSTaskCreate()会确保他们不会跟空闲任务有相同的优先级。当 CPU 中没有其它就绪任务运行时，空闲会被运行。空闲任务的重要部分代码如下（详见 OS_CORE.C 中的全部代码）

```
void  OS_IdleTask (void *p_arg)
{
    while (DEF_ON) {                    (1)
        OS_CRITICAL_ENTER();
        OSIdleTaskCtr++;                (2)
        OSTaskStatCtr++;
        OS_CRITICAL_EXIT();
        OSIdleTaskHook();               (3)
    }
}
```

Listing 5-4 **Idle Task**

L5-4（1）空闲任务是一个无限循环的不会等待任何事件的任务。这是因为，在大部分的处理器中，当没有事情可做时，处理器依然会执

行指令。当 uC/OS-III 中没有其它更高的就绪任务待运行时，uC/OS-III 就会把 CPU 分配给空闲任务。

L5-4（2）空闲任务运行时，两个计数变量会递增。

OSIdleTaskCtr 是用 32 位无符号整数定义的，在 uC/OS-III 初始化的时候它的值被复位。它用于表示空闲任务的活动情况。换句话说，如果用调试器查看该变量，就会看到介于 0x00000000 和 0xffffffff 之间的数。OSIdleTaskCtr 的增长速度取决于 CPU 的空闲情况。CPU 越空闲，该值增长越快。

OSStatTaskCtr 也是用 32 位无符号整数定义的，提供给测量任务测量 CPU 的利用率。

L5-4（3）空闲任务的每次循环，都会调用 OSIdleTaskHook()函数，这个函数提供给用户扩展应用。在这个函数中不要编写会让空闲任务被挂起的代码，对于 uC/OS-III 移植者来说这是一个常识。

OSIdleTaskHook()可以编写使 CPU 处于低功耗的代码。然而，这样的话就意味着 OSStatTaskCtr 不能再用于测量 CPU 的使用率了。

```
void  OSIdleTaskHook (void)
{
    /* Place the CPU in low power mode */
}
```

通常情况下，当中断发生时处理器退出低功耗模式。ISR 中可能会设置某些寄存器恢复 CPU 速度为全速或其想要的速度。ISR 可以

唤醒了一个高优先级任务（每个任务的优先级都比空闲任务的优先级高），然后 ISR 不会返回到空闲任务，而是切换到这个高优先级任务。如果这个任务完成操作或者挂起，uC/OS-III 就会切换到空闲任务并进入 OSIdleTaskHook()并进入低功耗模式。然后，进入 OS_IdleTask()并循环。

5-6-2 时基任务 OS-TickTask()

几乎所有的实时系统都需要有一个能提供周期性时间的时间源，叫做时基周期或系统周期。uC/OS-III 的时基周期处理程序封装在 OS_TICK.C 文件中。

OS_TickTask()任务被 uC/OS-III 创建，其优先级是用户可配置的。（通过配置 OS_CFG_APP.H 中的 OS_CFG_TICK_TASK_PRIO）。通常设置其优先级较高。事实上，它的优先级应该设置比重要任务的优先级稍低。

OS_TickTask()用于追踪等待期满的任务、挂起超时的任务。OS_TickTask()是一个周期性任务，它等待来自于 ISR 的信号量（详见章节 9"中断管理"）。

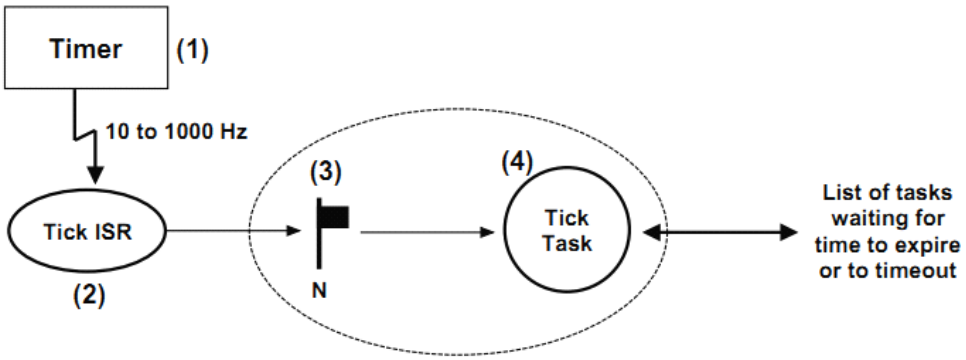


Figure 5-8 **Tick ISR and Tick Task relationship**

F5-8（1）使用硬件定时器并被设置为以 10 到 1000Hz 之间的频率产生中断，同时要设置 OS_CFG_APP.H 中 OS_CFG_TICK_RATE 为硬件定时器的中断频率。

时基中断并不是一定要用 CPU 产生，事实上，它可以从其他的具有较精确的周期性时间源中获得，比如电源线（50-60Hz）等。

F5-8（2）假定 CPU 中断使能，CPU 接收时基中断，并抢占当前任务，程序指针 SP 指向时基中断服务程序。时基中断服务程序必须调用 OSTimeTick()（详见 OS_TIME.C）,然后时基 ISR 清除该中断标志位。然而，有些应用中就需要先清中断标志再调用 OSTimeTick()。如下所示

```
void  TickISR (void)
{
    OSTimeTick();
    /* Clear tick interrupt source             */
    /* Reload the timer for the next interrupt */
}
```

或

```
void  TickISR (void)
{
    /* Clear tick interrupt source             */
    /* Reload the timer for the next interrupt */
    OSTimeTick();
}
```

OSTimeTick()首先调用 OSTimeTickHook()，它提供给用户扩展。（当时定时断产生时用户需要做的工作）

F5-8（3）

OSTimeTick()用于标记时基任务并就绪时基任务。定时器中断后基任务可能不被立即执行，因为中断程序打断的可能是一个比时基任务更高优先级的任务，完成时基 ISR 后，uC/OS-III 会返回被打断的这个任务。

F5-8（4）

当时基任务执行时，它会遍历队列中所有等待期满的任务、等待事件超时的任务。按照这个观点，这个会被叫做时基列表。时基任务会就绪时基列表中的那些期满、超时的任务。

uC/OS-III 的时基队列中有时也有可能存放了上百个任务（如果应用需要很多任务）。时基队列通过一种方法检测这些任务是否期满，是否可以被设置为就绪，该方法不会占用太多 CPU 时间。如图 5-9

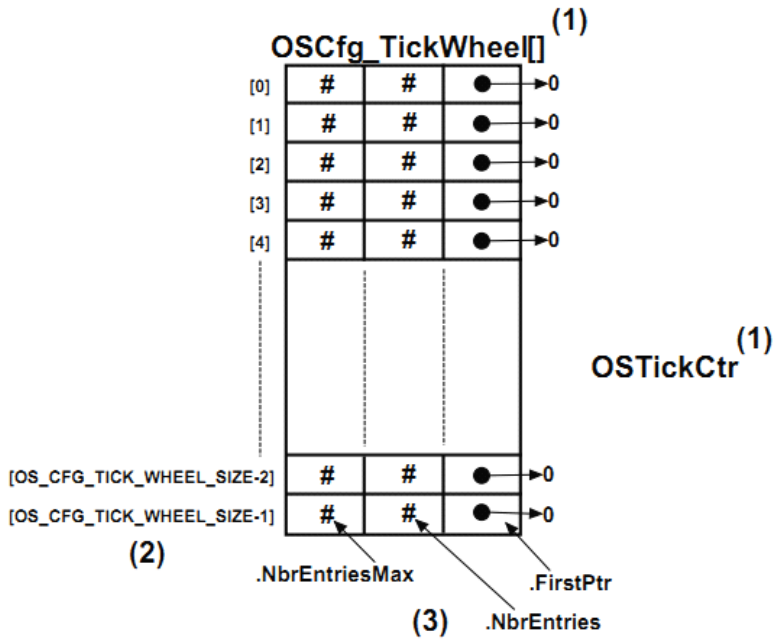


Figure 5-9 **Empty Tick List**

F5-9（1）时基列表中包含了一个表（OSCfg_TickWheel[]）和一个计数器(OSTickCtr)。

F5-9（2）这个表多达 OS_CFG_TICK_WHEEL_SIZE 个记录，它是在编译时配置的（详见 OS_CFG_APP.H）。记录数取决于处理器的 RAM 及应用中最大的任务数。推荐值为所有任务/4，不推荐使用偶数，避免设置 OS_CFG_TICK_WHEEL_SIZE 为 10（用 11 代替）。事实上，质数是一个很好的选择。

F5-9(3)表中的每个记录包含 3 个变量：.NbrEntriesMax，NbrEntries 和 FirstPtr。

NbrEntries 表明链接到该记录的任务序号。

NbrEntriesMax 追踪到表中优先级最高的记录。这个值在调用 OSStatReset()时被复位。

FirstPtr 包含了一个指向双向任务列表的指针。

当时基中断每产生一次，OSTickCtr 的值就会被 OS_TickTask()递增一次。

当调用 OSTimeDly???()或者 OS???Pend()时（所允许的超时时间大于 0），任务会被自动的插人时基列表。

例子 5-1

用一个例子来阐述时基列表中插入任务的过程，我们先假设时基列表为空，OS_CFG_TICK_WHEEL_SIZE 被设置为 12，当前的 OSTickCtr 值为 10。如图 5-10，当 OSTimeDly()被调用时，对应任务被放入时基列表。（假定 OSTimeDly()函数如下被调用）

```
:
OSTimeDly(1, OS_OPT_TIME_DLY, &err);
:
```

上述表示该任务需要被延时 1 个时基。因为 OSTickCtr 的值为 10，该任务将会处于等待状态直到 OSTickCtr 为 11。任务将会被插入到 OSCfg_TickWheel[]表中用如下的等式：

MatchValue = OSTickCtr+dly

Index into OSCfg_TickWheel[] =

MatchValue%OS_CFG_TICK_WHEEL_SIZE

"dly"是传递给 OSTimeDly()的第一个参数。在这个例子中为 1。

根据例子

即：

MatchValue = 11

Index into OSCfg_TickWheel[] = 11

因为这表是循环的（以模操作），表是一个时基轮，每一条记录都是时基轮的一部分。

OSCfg_TickWheel[]的索引 11 指向任务的 TCB。第一个任务的 TCB 的 TickNextPtr 地址被插入到表中 OSCfg_TickWheel[11].FirstPtr。表中索引 11 中的任务计数值递增（OSCfg_TickWheel[11].NbrEntries 会被设置为 1）。需要注意的是，TCB 中的.TickSpokePtr 会被链接到&OSCfg_TickWheel[11]。"MatchValue"会被存于 TCB.TickCtrMatch 中。因为这是插入到记录 11 的第一个任务，.TickNextPtr 和.TickPrevPtr 会被设置为指向 NULL。

OSTimeDly()还要照顾其它一些细节。特别的，当任务不再具备运行的资格时，该任务会被从 uC/OS-III 的就绪列表中移除。当 uC/OS-III 需要运行下一个最重要的就绪任务时，调度器会被调用。

在下一次时基到来之前，其它任务调用了 OSTimeDly()如下

```
:
OSTimeDly(13, OS_OPT_TIME_DLY, &err);
:
```

uC/OS-III 会计算其匹配值（match value）和记录号

MatchValue = 10+13

Index into OSCfg_TickWheel[] = (10+13)%12

或

MatchValue = 23

Index into OSCfg_TickWheel[] = 11

第二个任务会被插入到同一表中如图 5-11 所示。任务共享这条记录并重新排序，确保等待时间最少的任务位于记录 11 队列的头部。

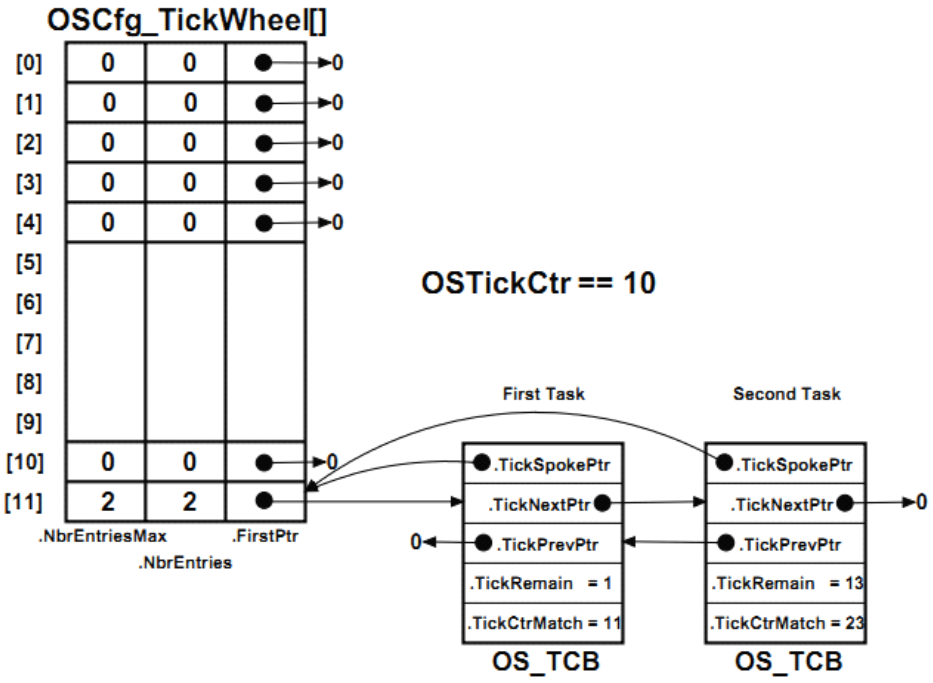


Figure 5-11 **Inserting a second task in the tick list**

{表中 OS_TCB 是任务 TCB 的一部分，其实是 TCB 中以 OS_TCB 类型定义的变量}

当时基任务被执行时（详见 OS_TICK.C 中的 OS_TickTask()和 OS_TickListUpdate()）。它会递增 OSTickCtr 的值并确定时基表中的哪些记录需要被执行。然后，如果有任务在这个记录中（记录的.FirstPtr 不为 NULL），记录中的每个 OS_TCB 会被检查其.TickCtrMatch 的值

是否与 OSTickCtr 的值相匹配。如果匹配，将该任务的 OS_TCB 从记录列表中移除，如果该任务只是在等待期满，它将会被放入就绪列表中。如果任务是等待一个对象，且等待超时了，它不仅会被移出时基列表，也会被移出该对象的挂起队列。遍历完记录中的队列，直到 TickCtrMatch 的值与 OSTickCtr 的值不再匹配为止。

需要注意的是，OS_TickTask()中与时基队列操作相关的大部分工作都需要以临界段的形式工作的。然而，因为记录被分类，临界段可以被缩为很短。

5-6-3 统计任务 OS_StatTask()

这个任务能够统计总的 CPU 使用率(0 到 100%),每个任务的 CPU 使用率（0 到 100%），每个任务的堆栈使用量。

统计任务在 uC/OS-III 中是可选的，当设置 OS_CFG.H 中的 OS_CFG_STAT_TASK_EN 为 1 时，统计任务的代码会被包含在程序中。

当然，统计任务的优先级和它的任务堆栈大小在 OS_CFG_APP.H 中配置。

最好在 main()中只创建的一个任务，通常叫做 AppTaskStat()，当使能了统计任务时，就必须在 AppTaskStat 任务中首先调用 OSStatTaskCPUUsageInit()。如列表 5-5 所示。在调用 OSStart()之前，用户的启动代码只能创建一个任务，而是由这个任务创建其它任务。

```
void main (void)                        (1)
{
    OS_ERR  err;
    :
    OSInit(&err);                       (2)
    if (err != OS_ERR_NONE) {
        /* Something wasn't configured properly, µC/OS-III not properly initialized  */
    }
    /* (3) Create ONE task (we'll call it AppTaskStart() for sake of discussion)     */
    :
    OSStart(&err);                      (4)
}


void AppTaskStart (void *p_arg)
{
    OS_ERR  err;
    :
    /* (5) Initialize the tick interrupt                                             */
#if OS_CFG_STAT_TASK_EN > 0
    OSStatTaskCPUUsageInit(&err);       (6)
#endif
    :
    /* (7) Create other tasks                                                        */
    while (DEF_ON) {
        /* AppTaskStart() body                                                       */
    }
}
```

Listing 5-5 **Proper startup for computing CPU utilization**

L5-5（1）CPU 进入 main()函数中。

L5-5（2）main()函数调用 OSInit()初始化 uC/OS-III。假定在 OS_CFG_APP.H 中设置 OS_CFG_STAT_TASK_EN 为 1，使能统计任务。通过 uC/OS-III 返回的错误代号检测系统初始化是否成功。（详见 OS.H 中的错误代号 OS_ERR_???）。

L5-5（3）创建一个叫做 AppTaskStart()的任务。创建这个任务的时候，给它一个相当高的优先级（不要用优先级 0，因为这是为 uC/OS-III 保留的）。

uC/OS-III 允许用户在调用 OSStart()之前创建任意个任务。然而，当用到统计任务统计 CPU 的使用率时，调用 OSStart()之前只能创建一个任务。

L5-5（4）调用 OSStart()，让 uC/OS-III 开始运行优先级最高的任务，根据例子，这个任务是 AppTaskStart()。在这个时候，已经有五个任务被创建了：OS_IdleTask()，OS_TickTask()，OS_StatTask()，OS_TaskTmr()，AppTaskStart()。

L5-5（5）这个任务应该先设置和开启时基中断，初始化用于时基时钟的硬件定时器，设置其中断的速率。（编译时设置 OS_CFG_APP.H 中的 OS_CFG_STAT_TASK_RATE）。另外，Micrium 提供的例子工程中包含了基本的板级支持包 BSP。BSP 初始化了 CPU 很多方面的也包括 uC/OS-III 需要的周期时间源。如果需要，用户可以在开启任务中调用 BSP_Init()初始化 BSP 服务。

L5-5（6）调用 OSStatTaskCPUUsageInit()。当没有其它应用任务运行时，经过 1/OS_CFG_STAT_TASK_RATE 秒后 OSStatTaskCtr 的计数值就是 OSStatTaskCtr 的最大值，它意味着 CPU 的空闲时的工作速率。例如，如果系统中不包含应用任务，OSStatTaskCtr 从 0 开始计数到 10,000,000 用了

1/OS_CFG_STAT_TASK_RATE 秒。添加了应用任务后，

OSStatTaskCtr 每 1/OS_CFG_STAT_TASK_RATE 秒检测一次。得到的值不会达到 10,000,000。CPU 的利用率如下计算：

CPU 利用率%=（100-100*OSTaskStatCtr/OSTaskStatCtr(max)）

如果重新测得的 OSStatTaskCtr 值为 7,500,000。那么 CPU 的利用率为 25%。

25%=（100-100*7500000/10000000)

L5-5（7）然后在 AppTaskStart()创建其它的应用任务。

正如先前描述的，uC/OS-III 将任务的 CPU 使用率存于任务的 TCB。

OS_StatTask()也可以计算每个任务堆栈的使用量（通过调用 OSTaskStkChk()）。并将计算结果存于每个任务 OS_TCB 的 StkFree 和 StkUsed 中。

5-6-4 定时器任务 OS_TmrTask()

{这节所说的定时器都是软件定时器}

uC/OS-III 为用户提供了定时器任务，相应代码在 OS_TMR.C 中。

定时器任务是可选的，通过将 OS_CFG.H 中的 OS_CFG_TMR_EN 设置为 1 使能。当设置为 1 时，它的代码才会被添加到最终代码中。

当定时器任务递减计数变量到 0 时，任务中就会调用回调函数。回调函数是一个函数，它被用户定义。因此，回调函数可以用来开启或关闭 LED、电机、或者其他的一些操作。用户可以创建任意个定时器（只限制于处理器的 RAM）。定时器管理在 12 章详细介绍。

OS_TmrTask()是一个被 uC/OS-III 创建的任务（假定设置 OS_CFG.H 中的 OS_CFG_TMR_EN 为 1），它通过 OS_CFG_APP.H 中的 OS_CFG_TMR_TASK_PRIO 设置优先级。一般情况下，它的优先级被设为中等。

OS_TmrTask()是周期函数。中断源产生时基。然而，定时器通常需要较低的率（典型为 10Hz），可以通过软件将时基分频。换句话说，如果时基速率为 1000Hz,但是想要的定时器速率为 10Hz,定时器任务会每 100 个时基被标记一次。如图 5-12

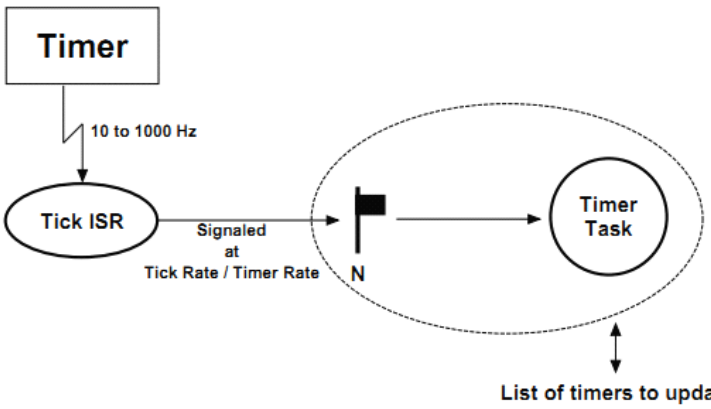


Figure 5-12 **Tick ISR and Timer Task relationship**

5-6-5 中断处理任务 OS_IntQTask()

当设置 OS_CFG.H 中的 OS_CFG_ISR_POST_DEFERRED_EN 为 1 时，uC/OS-III 就会创建一个任务，它的作用是尽快完成 ISR 中对 post 函数的调用，将信号量、消息等对象先存在媒介中，退出中断后，由中断处理任务完成将这些对象提交给任务。

正如第 4 章“临界段”所介绍的，uC/OS-III 通过开启/关闭中断、锁/开锁调度器管理临界段。如果选择后一种方法，ISR 调用的 post 函数不允许直接访问就绪列表、挂起队列等。

当 ISR 调用 uC/OS-III 提供的 post 函数时，ISR 要提交的数据会被复制，该数据的目的地会被放到一个特殊的队列—"holding"队列。当所有的嵌套中断结束时，uC/OS-III 切换到中断处理任务，它会将放置在"holding"队列中的信息重新提交到适当的任务。这个额外步骤的目的是减小关中断时间。

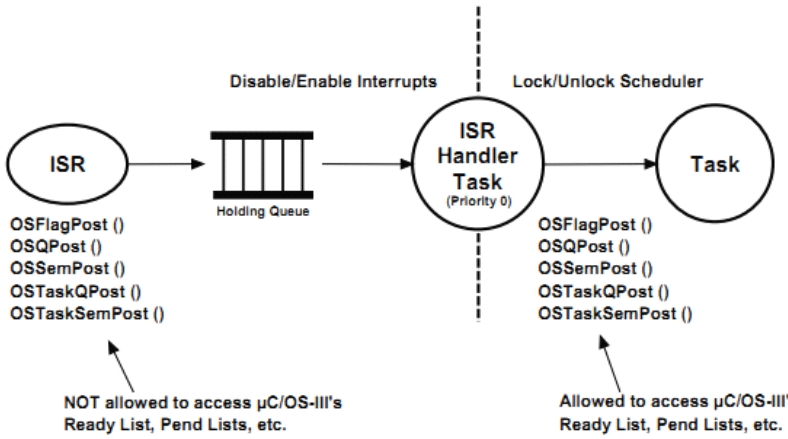


Figure 5-13 **ISR Handler Task**

OS_IntQTak()被 uC/OS-III 创建，它的优先级通常被设为 0（最高优先级）。如果 OS_CFG_ISR_POST_DEFERRED_EN 被设置为 1，使能了这个任务，其它的任务的优先级就不能设置为 0。

5-7 总结

任务是一段简单的程序。在单 CPU 系统中，任何时间只能有一个任务被 CPU 运行。uC/OS-III 支持多任务并对任务数没有限制（仅限制于 CPU 的存储空间，包括代码空间和数据空间）。

uC/OS-III 创建五个内部任务：空闲任务，时基任务，中断处理任务，统计任务，定时器任务。空闲任务、时基任务是必须的，统计任务、定时器任务、中断处理任务是可选择的。

6、就绪列表

准备运行的任务被放置于就绪列表中。就绪列表包括 2 个部分：位映像组包含了优先级信息，一个表包含了所有指向就绪任务的指针。

6-1 优先级

图 6-1 到 6-3 显示了优先级的位映像组。它的宽度取决于 CPU_DATA 的数据类型（见 CPU.H），它可以是 8 位、16 位、32 位。根据处理器相应地设定。

uC/OS-III 支持多达 OS_CFG_PRIO_MAX 种不同的优先级（见 OS_CFG.H）。在 uC/OS-III 中，数值越小优先级越高。因此优先级 0 是优先级最高的。优先级 OS_CFG_PRIO_MAX-1 的优先级最低。uC/OS-III 将最低优先级唯一地分配给空闲任务，其它任务不允许被设置为这个优先级。当任务准备好运行了，根据任务的优先级，位映像表中相应位就会被设置为 1。如果处理器支持位清零指令 CLZ，这个指令会加快位映像表的设置过程。

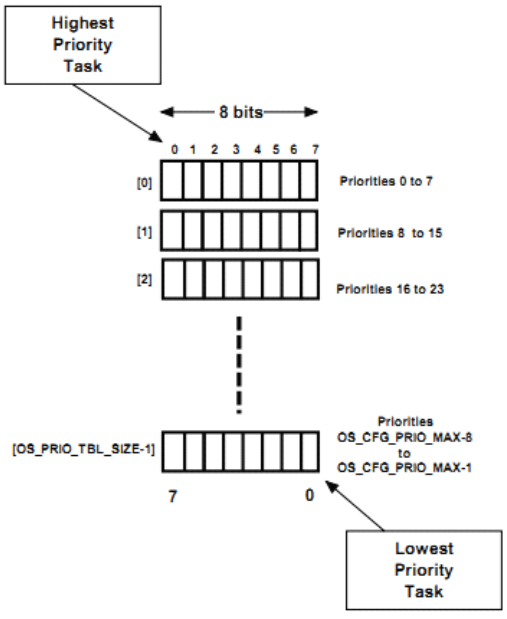


Figure 6-1 **CPU_DATA declared as a CPU_INT08U**

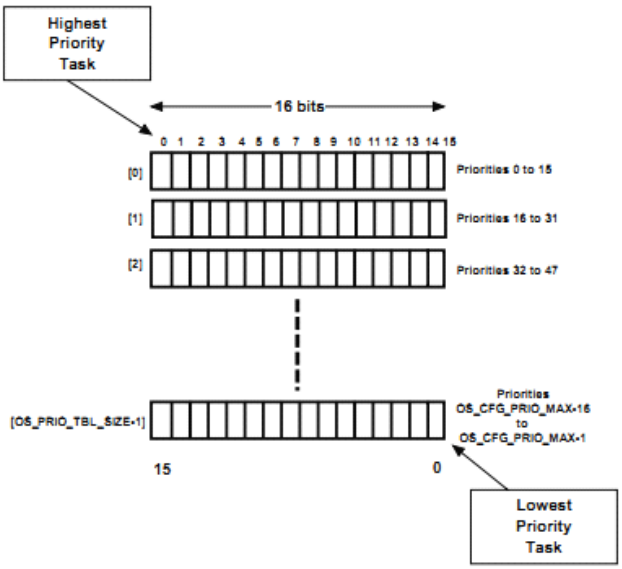


Figure 6-2 **CPU_DATA declared as a CPU_INT16U**

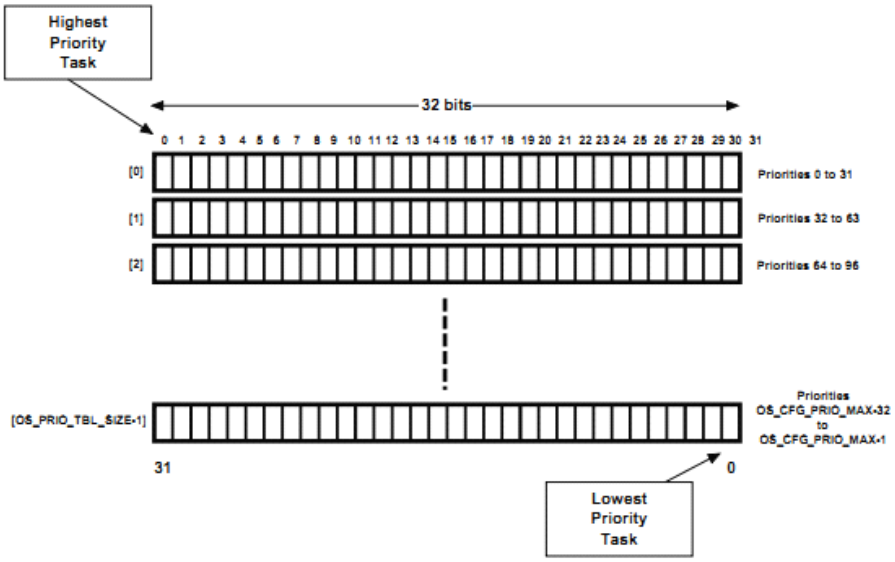


Figure 6-3 **CPU_DATA declared as a CPU_INT32U**

OS_PRIO.C 中包含了位映像表的设置、清除、查找的相关代码。这些函数都是 uC/OS-III 的内部函数，可以用汇编语言优化。

函数	功能呢
OS_PrioGetHighest()	查找最高优先级
OS_PrioInsert()	设置位映像表中相应的位
OS_PrioInsert()	清除位映像表中相应的位

表 6-1 优先级相关的函数

为了确定就绪列表中优先级最高的任务，位映像表会被扫描，通过 OS_PrioGetHighest()函数找到优先级最高的任务。代码如列表 6-1 所示

```
OS_PRIO  OS_PrioGetHighest (void)
{
    CPU_DATA  *p_tbl;
    OS_PRIO    prio;


    prio  = (OS_PRIO)0;
    p_tbl = &OSPrioTbl[0];
    while (*p_tbl == (CPU_DATA)0) {              (1)
        prio += sizeof(CPU_DATA) * 8u;           (2)
        p_tbl++;
    }
    prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);   (3)
    return (prio);
}
```

Listing 6-1 **Finding the highest priority level**

L6-1（1）OS_PrioGetHighest()函数扫描 OSPrioTbl[]表直到找到非 0 的记录。这个循环最终会停止，因为总是有非 0 记录（空闲任务的存在)。

L6-1（2） 当这个表中全是 0 记录时，就会从下一个表中查找。优先级"prio"会被增加（如果表长 32 位，就将 prio 加上 32)。

L6-1（3）当找到第一个非 0 位时，计算该位之前 0 位的个数，返回该优先级值。如果 CPU 提供清零指令，可以通过这个指令优化代码。如果 CPU 没有这个指令，那么这个指令就只能用 C 语言模拟了。

函数 CPU_CntLeadZeros()统计了 CPU_DATA 记录中 0 的个数（从左边开始，以位计）。例如，假定位映像表长 32 位，0xF0001234 返回的非 0 位前 0 位数的个数是 0。0x00F01234 返回的是 8。

显而易见，线性地扫描这个表是效率低下的。然而，如果优先级数少的话，扫描是非常快的。事实上，有一些方法可以优化扫描。例如，如果使用 32 位处理器且应用中需要的优先级数少于 64，那么上述代码就可以被优化如列表 6-2 所示。若处理器中清零指令，该部分代码可以通过汇编被优化为很短的代码。

```
OS_PRIO  OS_PrioGetHighest (void)
{
    OS_PRIO  prio;


    if (OSPrioTbl[0] != (OS_PRIO_BITMAP)0) {
        prio = OS_CntLeadingZeros(OSPrioTbl[0]);
    } else {
        prio = OS_CntLeadingZeros(OSPrioTbl[1]) + 32;
    }
    return (prio);
}
```

Listing 6-2 **Finding the highest priority level within 64 levels**

6-2 就绪列表

准备好运行的任务被放到就绪列表中，如图 6-1。就绪列表是一个数组（OSRdyList[]），它一共有 OS_CFG_PRIO_MAX 条记录，记录的数据类型为 OS_RDY_LIST(见 OS.H)。就绪列表中的每条记录都包含了三个变量 .Entries 、.TailPtr 、.HeadPtr。

.Entries 中该优先级的就绪任务数。当该优先级中没有任务就绪时，.Entries 就会被设置为 0。

.TailPtr 和.HeadPtr 用于该优先级就绪任务的建立双向列表。.HeadPtr 指向列表的头部，.TailPtr 指向列表的尾部。

表中的记录跟任务的优先级有关。例如，如果一个任务的优先级是 5，那么当它就绪时会被放入 OSRdyList[5]中。

表 6-2 中列出了与就绪列表相关的操作函数。这些都是 uC/OS-III 的内部函数，用户不能调用它们。

函数名	功能
OS_RdyListInit()	初始化就绪列表为空
OS_RdyListInsert()	插入一个 TCB 到就绪列表
OS_RdyListInsertHead()	插入一个 TCB 到就绪列表的头部
OS_RdyListInsertTail()	插入一个 TCB 到就绪列表的尾部
OS_RdyListMoveHeadToTail()	将 TCB 从列表的头部移到尾部
OS_RdyListRemove()	将 TCB 从就绪列表中移除

表 6-2 与就绪列表相关的函数

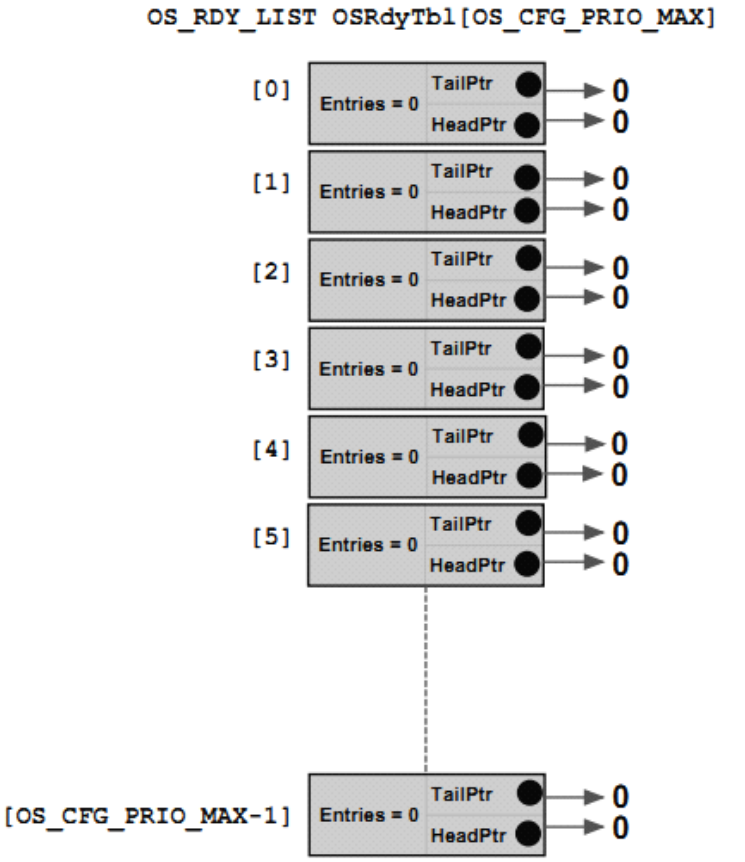


Figure 6-4 **Empty Ready List**

F6-4（1）该图显示的是就绪列表为空时的状态。

F6-4（2）有多少种优先级，就绪列表中就由多少条记录。每个记录中都有 3 个变量。Entries 为该记录中的任务数。.PrevPtr 和.NextPtr 用于指向具有相同优先级到 TCB 组成的双向列表。对于空闲任务，这两个值为 NULL。

F6-4（3）优先级 0 是为中断处理任务保留的（当设置 OS_CFG.H 中的 OS_CFG_ISR_DEFERRED_EN 为 1）。在这种情况下，它是唯一一个能被设为优先级 0 的任务。

假定 uC/OS-III 内部的任务都被使能。图 6-5 显示了当 OSInit()被调用后的就绪列表，

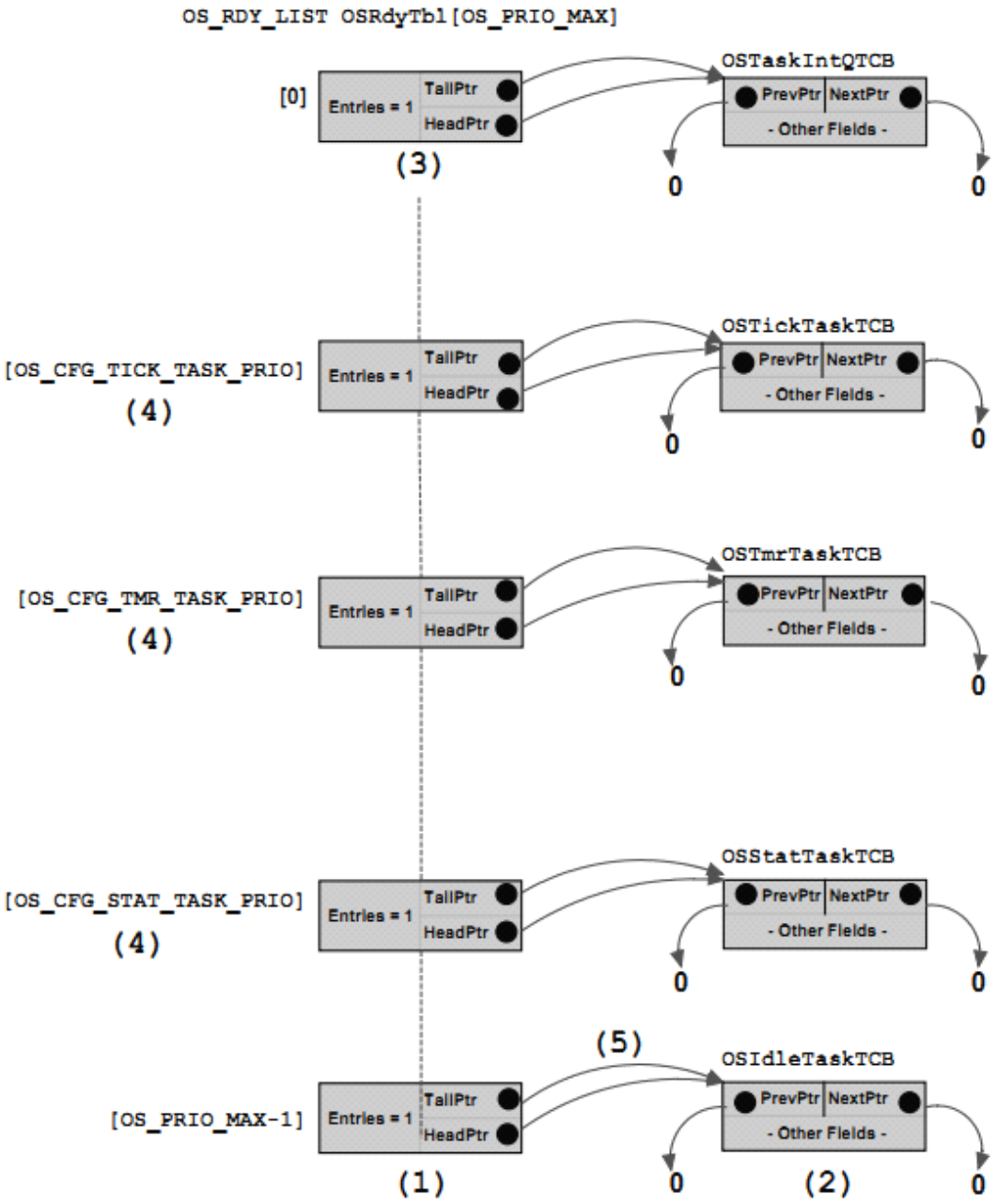


Figure 6-5 **Ready List after calling OSInit()**

F6-5（1）如图所示，时基任务和另两个任务有它们自己的优先级。uC/OS-III 允许多个任务有相同的优先级。特别的，时基任务的优先级要高于定时器任务，定时器任务的优先级需要于统计任务。

F6-5（2）当某个优先级中只有一个任务的时候，它的头指针和尾指针会指向同一个 TCB。

6-3 添加任务到就绪队列

uC/OS-III 提供很多服务可以把任务添加到就绪列表中。最明显的服务是 OSTaskCreate()，它通常创建准备运行的任务并将任务放入就绪列表中。如图 6-6 所示，就绪列表中该优先级中已经有两个任务了。OSTaskCreate()就会将这个任务插入到列表的未部。

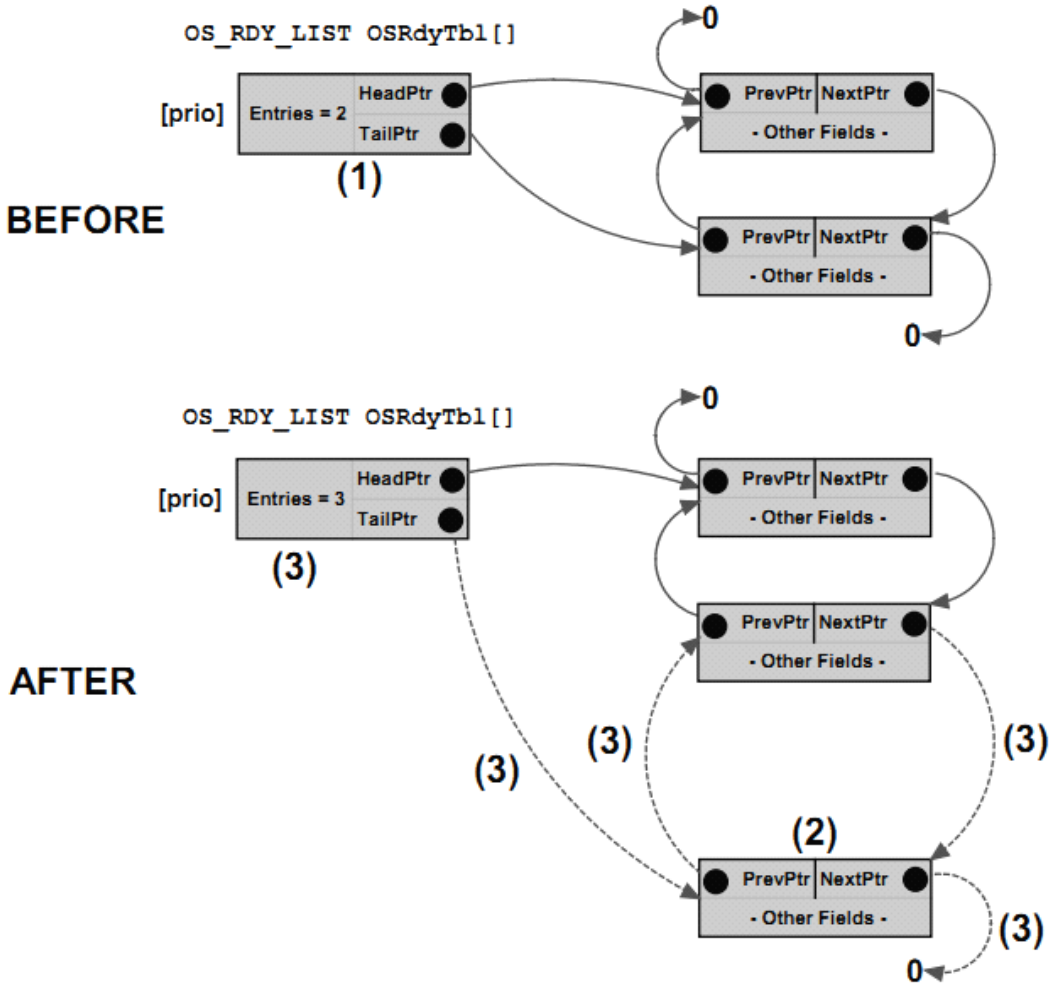


Figure 6-6 **Inserting a newly created task in the ready list**

F6-6（1）在调用 OSTaskCreate()之前，已经有两个任务在就绪列表中的该优先级中了。

F6-6（2）一个新的 TCB 传递给 OSTaskCreate()，然后 uC/OS-III 初始化这个 TCB。

F6-6（3）OSTaskCreate()调用 OS_RdyListInsertTail()，OS_RdyListInsertTail()的作用是将新添加进来的 TCB 链接到就绪队列中对应优先级记录（需设置四个指针，就绪列表中两个，任务重两个，并将记录中的.Entries 的值加一）。

OSTaskCreate()还会调用 OS_PrioInsert()设置位映像表中的相应位。然而，处理 OS_PrioInsert()是非常快的，它不会影响到应用的性能。

当一个任务创建了一个具有相同优先级的任务，这个新任务会被添加到该优先级队列的尾部（因为具有相同优先级情况下，没有理由让新任务先运行）。然而，当一个任务创建了一个具有不同优先级的任务时，这个新的任务就会放到对应优先级列表中的首部。

6-4 总结

uC/OS-III 支持任意数量的优先级。然而，绝大多数系统的需求量不会超过 64 种。

就绪队列包含两个数据结构：位映像表保存了哪个优先级中有任务待运行，优先级列表中包含了这该优先级下等待运行的任务。

处理器支持清零指令 CLZ 会加快位映像表的扫描速度。

{注意：正在运行的任务也被放在就绪列表中。}

7、调度

调度器，决定了任务的运行顺序。uC/OS-III 是一个可抢占的，基于优先级的内核。根据其重要性每个任务都被分配了一个优先级。uC/OS-III 支持多个任务拥有相同的优先级。

“可抢占的”意味当事件发生时，如果事件让高优先级任务被就绪，uC/OS-III 马上将 CPU 的控制权交给高优先级任务。因此，当一个任务提交信号量、发送消息给一个高优先级的任务（若该任务被就绪了），当前的任务就会被停止，更高优先级的任务获得 CPU 的控制权。类似的，当 ISR 提交信号量或发送消息给一更高优先级的任务（若该任务被就绪了），那么中断返回的时候不会返回到原任务，而是高优先级任务。

7-1 抢占式调度

uC/OS-III 通过两种方法处理中断提交的事件:直接提交或延迟提交。这些会在章节 9 中详细介绍。从调度的角度看，这两种方法产生的结果是一样的；最高优先级的就绪任务会占用 CPU 如图 6-1 和 6-2 所示。

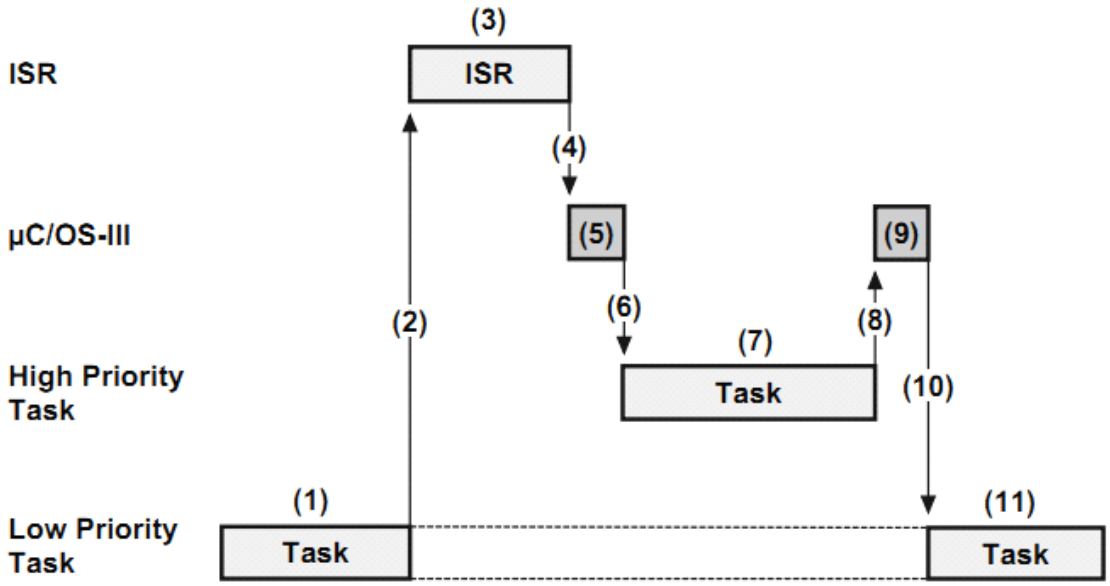


Figure 7-1 **Preemptive scheduling – Direct Method**

F7-1（1）一个低优先级任务正在执行，这时中断发生了。

F7-1（2）如果中断使能，指令指针 IP 会跳转到中断服务程序。

F7-1（3）中断服务程序处理相关的程序，发送信号量或消息给一个高优先级任务。高优先级任务被就绪。

F7-1（4）中断服务程序完成操作后，它将 CPU 的控制权还给 uC/OS-III。

F7-1（5）uC/OS-III 执行相应的操作。

F7-1（6）因为就绪队列中有一个更重要的任务，uC/OS-III 将不会返回中断前那个低优先级的任务。而是执行高优先级任务。

F7-1（7）开始执行高优先级任务

F7-1（8）高优先级任务处理完成后，将 CPU 的控制权交给 uC/OS-III。

F7-1（9）uC/OS-III 执行相应的操作。

F7-1（10）uC/OS-III 转向执行原先那个低优先级任务。

F7-1（11）低优先级任务从被中断处继续执行。

图 7-2 显示了当选择延迟提交中断产生的事件的方式时的一些额外步骤。最后的结果是一样的：高优先级任务抢占了低优先级任务。

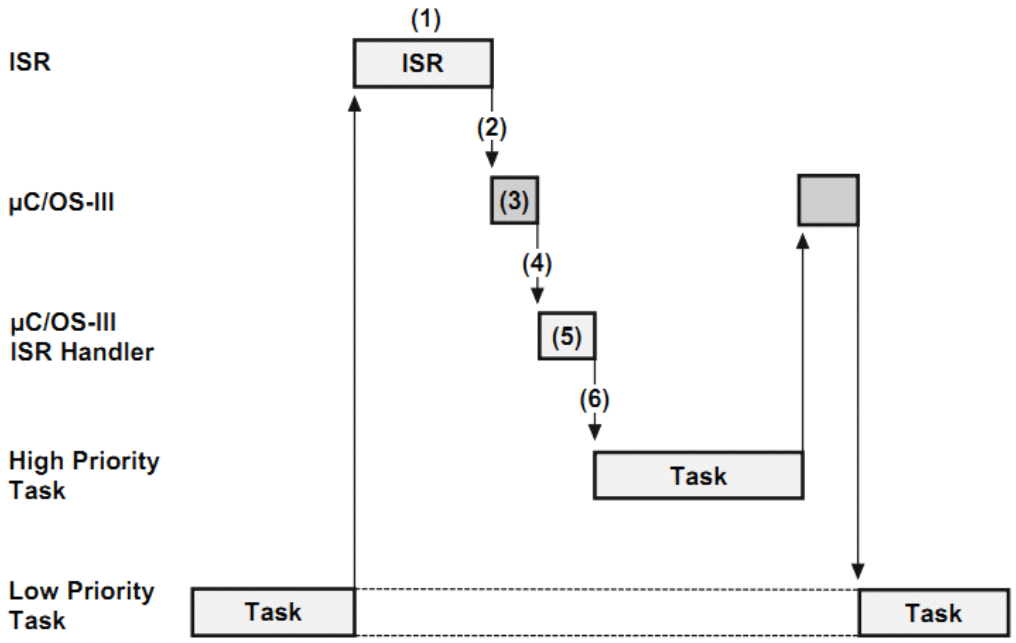


Figure 7-2 **Preemptive scheduling – Deferred Post Method**

F7-2（1）中断服务程序中，不是直接发送信号量或消息给任务。而是先将信号量或消息放人外部消息队列，并就绪中断处理任务。

F7-2 （3）当 ISR 处理完它的工作时，就把 CPU 的使用权交给 uC/OS-III。

F7-2（3）uC/OS-III 处理一些操作。

F7-2（4）因为中断处理任务被就绪，uC/OS-III 将 CPU 的使用权交给它。

F7-2（5）uC/OS-III 处理一些操作。

F7-2（6) 中断处理任务将外部消息队列中的消息移除并重新提交给相应的任务。这样，就将这个过程消息提交从中断级变成了任务级。使用这种方法的目的是为了减小关中断时间（详见第 9 章）。当消息队列为空时，uC/OS-III 将中断处理任务从就绪队列中移除，然后执行就绪队列中的最高优先级任务。

7-2 调度点

当出现以下情况时会发生调度：

任务被标记或发送消息给另一个任务

任务调用提交服务函数 OS???Post()，发送信号量或消息给其它任务时调度发生。调度在 OS???Post()函数的结束时发生。注意在有些情况下，调度是不会发生的（见 OS_OPT_POST_NO_SCHED 的可选参数）。

任务调用 OSTimeDly()或 OSTimeDlyHMSM():

如果延时参数不是 0，调度发生，调度会在该任务被放入挂起队列后马上执行。

任务所等待的事件发生或超时

当 OS???Pend()被调用时。接收到该事件的任务、或超时的任务就会被移出等待队列。然后调度器选择就绪列表中优先级最高的任务执行。{移出等待队列的任务不一定就是就绪状态，因为它还可能在停止队列中，效果是可以叠加的。}

任务取消挂起

一个任务可以被取消挂起，若另一个任务调用 OS???PendAbort()。当任务被移出等待列表中时调度发生。

新任务被创建

新的任务被创建时调度发生。

任务被删除

当一个任务被删除时调度发生。

内核对象被删除

任务所等待的内核对象被删除时（事件标志组、信号量、消息队列、mutex 都是内核对象），这些任务就可能被就绪。然后调度发生。

任务改变自身的优先级或其它任务的优先级

当任务改变自身优先级或其它任务优先级时，调度发生。

当任务通过调用 OSTaskSuspend()停止自身

当任务调用 OSTaskSuspend()停止自身时，调度发生。

任务调用 OSTaskResume()恢复其它停止了的任务

任务调用 OSTaskResume()恢复其它停止了的任务时，调度发生。

退出中断服务程序

当退出中断服务程序时，调度发生。这种情况下，调度器调用OSIntExit()函数开始调度而不是 OSSched()。

通过调用 OSSchedUnlock()调度器被解锁

调用 OSSchedLock()锁调度器，调度发生。需要注意的是，锁调度器可以被嵌套，解锁次数必须等于加锁次数。

调用 OSSchedRoundRobinYield()任务放弃了分配给它的时间片

假定多个任务有相同的优先级，正在运行的任务放弃了分配给它的时间片。调度发生。

用户调用 OSSched()

用户程序调用 OSSched()时，调度发生。若调用 OS???Post()函数时设置参数为了 OS_OPT_POST_NO_SCHED，事件在被提交后，不调用调度器。当然，任务可以一次性提交多个事件，但在最后一个事件提交时才调用调度器。{因为当一个事件被提交时，调度器就会发生调度。所以用户设置了 OS_OPT_POST_NO_SCHED 后，用户一次性提交多个事件而只发生一次发生调度。}

7-3 循环轮转调度

当多个任务有相同的优先级时，uC/OS-III 允许每个任务运行规定的时间片。当任务没有用完分配给它的时间片时，它可以自愿地放弃 CPU。uC/OS-III 允许任务在运行时开启或者关闭循环轮转调度。

图 F7-3 是相同优先级下任务运行的时序图。如图，有三个优先级都为 X 的任务。为了说明，时间片的长度为 4。

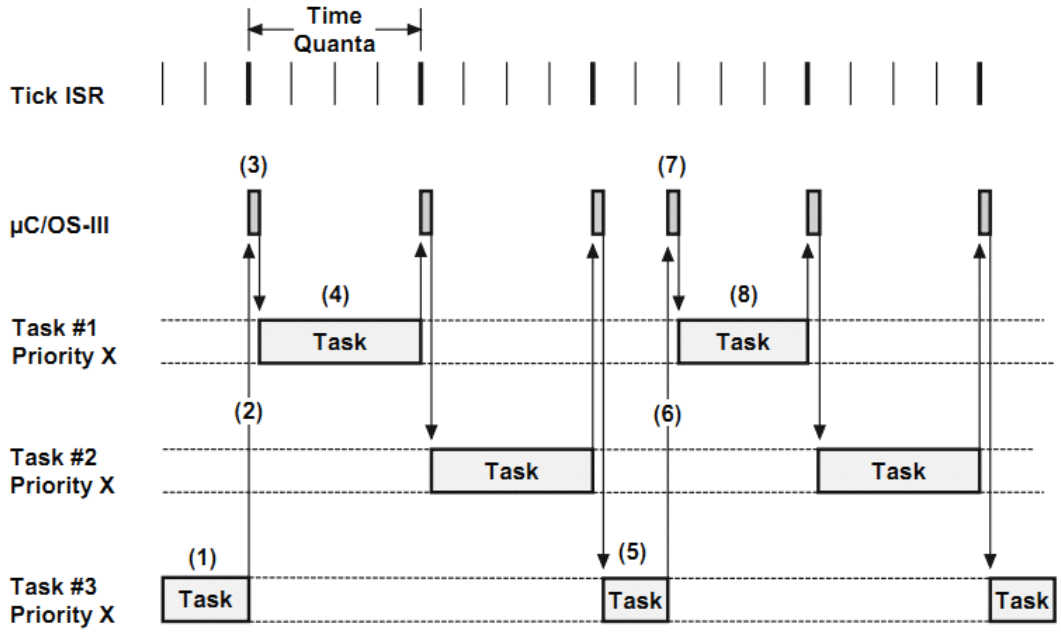


Figure 7-3 **Round Robin Scheduling**

F7-3（1）任务 3 正在被运行。在这段时间内，时基中断发生，但任务 3 还没有到期。

F7-3（2）任务 3 主动放弃剩下的时间片。

F7-3（3）uC/OS-III 恢复任务 1，因为它是队列中任务 3 的下一个任务。

F7-3（4）任务 1 被执行直到分配给它的时间片到期。

F7-3（5）任务 3 正在被运行。在这段时间内，时基中断发生，但任务 3 还没有到期。

F7-3（6）任务 3 主动放弃剩下的时间片。

F7-3（7）有趣的事情发生了，uC/OS-III 会重新设置任务 1 的时间片长度为 4 个时基。但剩余时间片的计数是每个时基中断递减，即在第 4 个时基中断发生时时间片到期。而任务 1 是在时基即将发生时接手的，所以事实上它的时间片会少一个时基。

F7-3（8）任务 1 执行，uC/OS-III 允许用户在任务运行时修改时间片的长度（通过 OSSchedRoundRobinCfg()，详见附录 A）。这个函数也可以用于开启或关闭循环轮转调度。

uC/OS-III 允许用户为每个任务设置不同的时间片。不同的任务可以有不同的时间片。当任务被创建时，其时间片长度被设置。也可以在运行时调用 OSTaskTimeQuantaSet()修改。

7-4 调度的内部实现

通过这两个函数完成调度功能：OSSched()和 OSIntExit()。OSSched()在任务级被调用，OSIntExit()在中断级被调用。这两个函数都在 OS_CORE.C 中定义。

图 7-1 阐述了调度所需用的两个数据结构。位映像表和就绪列表。

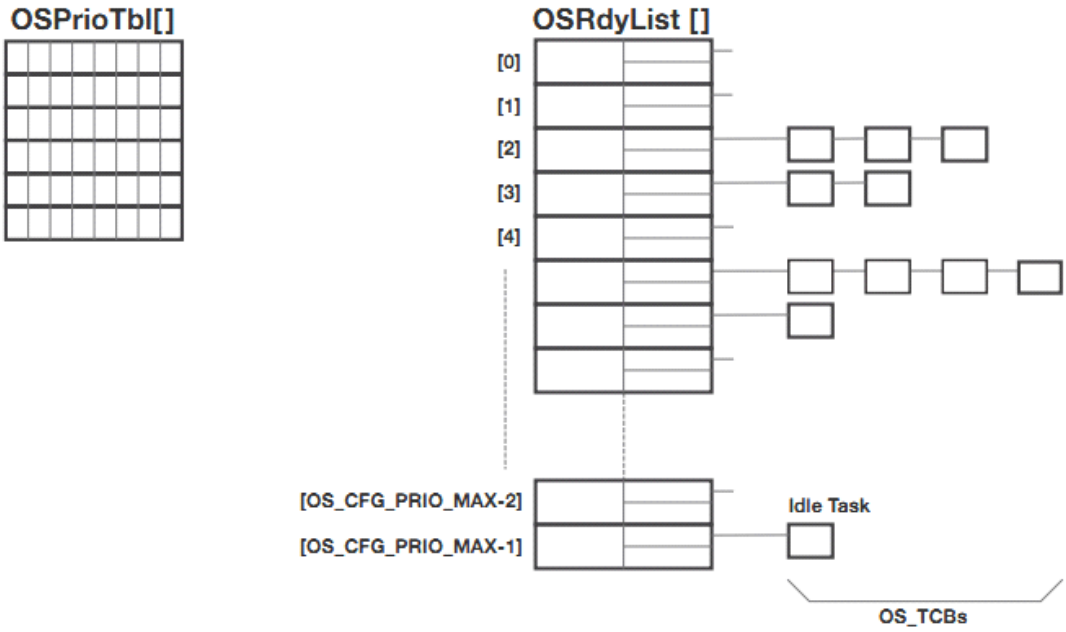


Figure 7-4 **Priority ready bitmap and Ready list**

7-4-1 OSSched()

以下是任务级调度的伪代码。

```
void  OSSched (void)
{
    Disable interrupts;
    if (OSIntNestingCtr > 0) {                                    (1)
        return;
    }
    if (OSSchedLockNestingCtr > 0) {                              (2)
        return;
    }
    Get highest priority ready;                                   (3)
    Get pointer to OS_TCB of next highest priority task;          (4)
    if (OSTCBNHighRdyPtr != OSTCBCurPtr) {                        (5)
        Perform task level context switch;
    }
    Enable interrupts;
}
```

Listing 7-1 **OSSched() pseudocode**

L7-1（1）因为要进入调度程序，关中断。OSSched()首先确定自己不是被中断服务程序调用，因为它是任务级调度程序。中断级调度需调用 OSIntExit()。

L7-1（2）第二步是确保调度器没有被锁。如果调度器被锁，就不能运行调度器，函数马上返回。

L7-1（3）OSSched()通过扫描位映像表 OSPrioTbl[]找到就绪列表中优先级最高的位。（详见章节 6）

L7-1（4）确定好优先级后，索引到 OSRdyList[]并提取该记录中的首个 TCB（OSRdyList[highest priority].HeadPtr）。到现在为止，已经知道了哪个任务将要切换为运行状态。特别的，OSTCBCurPtr 指向的是当前任务的 TCB，OSTCBHighRdyPtr 指向的是将要被切换的 TCB。

L7-1（5）当就绪表中的最高优先级任务不为当前正在运行的任务时（OSTCBCurPtr!=OSTCBHighRdyPtr），进行任务级的上下文切换。然后在开中断。

注意的是，调度时和上下文切换是运行在关中断的状态下的。这是必要的因为这些操作是原子性的。

7-4-2 OSIntExit()

以下是 ISR 级的调度器代码 OSIntExit()如列表 7-2 所示。需要注意它假定当调用了 OSIntExit()之后中断被关闭。

```
void  OSIntExit (void)
{
    if (OSIntNestingCtr == 0) {                                  (1)
        return;
    }
    OSIntNestingCtr--;
    if (OSIntNestingCtr > 0) {                                   (2)
        return;
    }
    if (OSSchedLockNestingCtr > 0) {                             (3)
        return;
    }
    Get highest priority ready;                                  (4)
    Get pointer to OS_TCB of next highest priority task;         (5)
    if (OSTCBHighRdyPtr != OSTCBCurPtr) {                        (6)
        Perform ISR level context switch;
    }
}
```

Listing 7-2 **OSIntExit() pseudocode**

L7-2（1）OSIntExit()开始时确保中断嵌套级不为 0。{如果为 0 的话，那么接下来的 OSIntNestingCtr--就会发生溢出了}

L7-2（2）OSInrExit()将 OSIntNestingCtr 递减，若 OSIntNestingCtr 不为 0 时则返回。确保该函数是在第一级 ISR 中执行。当还有中断程序未被处理时就不能运行调度器。

L7-2（3）OSIntExit()检查调度器是否被锁。如果被锁，OSIntExit()不会运行调度器并返回到中断前的任务。

L7-2（4）当这是第一级中断且调度器没有被锁时。查找优先级最高的任务。

L7-2（5）从 OSRdyList[]中获得最高优先级的 TCB。

L7-2（6）如果最高优先级就绪任务不是当前正在运行的任务。

uC/OS-III 就执行中断级的上下文切换。中断级切换的上文已经在中断开始时被保存了，它可以直接切换到任务。这些在第 8 章中详细介绍。

7-4-3 OS_SchedRoundRobin()

当同一优先级中有多个任务。一个任务的时间片期满，uC/OS-III 就会运行同优先级就绪的任务。OS_SchedRoundRobin()就是执行这种操作的，它被 OSTimeTick()或者 OS_IntQTask()调用。

OS_SchedRoundRobin()在 OS_CORE.C 中定义。

当你选择直接提交方式时，OS_SchedRoundRobin()被 OSTimeTick()调用。当选择延迟提交方式时，OS_SchedRoundRobin()被 OS_IntQTask()调用。

OS_SchedRoundRobin()的伪代码如列表 7-3 所示

```
void  OS_SchedRoundRobin (void)
{
    if (OSSchedRoundRobinEn != TRUE) {                          (1)
        return;
    }
    if (Time quanta counter > 0) {                              (2)
       Decrement time quanta counter;
    }
    if (Time quanta counter > 0) {
        return;
    }
    if (Number of OS_TCB at current priority level < 2) {       (3)
        return;
    }
    if (OSSchedLockNestingCtr > 0) {                            (4)
        return;
    }
    Move OS_TCB from head of list to tail of list;              (5)
    Reload time quanta for current task;                        (6)
}
```

Listing 7-3 **OS_SchedRoundRobin() pseudocode**

L7-3（1）OS_SchedRoundRobin()开始时先确定循环轮转调度已经被使能。（必须通过调用 OSSchedRoundRobinCfg()开启循环轮转调度）。

L7-3（2）时间片计数值，驻留于正在运行任务的 TCB 中，每次时基中断时它被递减。当它的值不为 0 时 OS_SchedRoundRobin()返回。

L7-3（3）当时间片计数值为 0 时，检查是否有其它同优先级任务等待被运行。如果没有，就返回。

L7-3（4）当调度器被锁时，OS_SchedRoundRobin()返回。

L7-3（5）若当前任务的时间片到期，OS_SchedRoundRobin()将当前任务的 TCB 移动到该优先级队列的末尾（从头部移到尾部）。

L7-3（6）在该优先级记录队列首部的任务的时间片值被载入到任务。每个任务都可以有自己的时间片值。（任务创建时被设置或通过 OSTaskTimeQuantaSet()设置）。如果该值为 0，uC/OS-III 就假定时间片值为默认值 OSSchedRoundRobinDfltTimeQuanta。

7-5 总结

uC/OS-III 是可抢占的调度器，所以它总是执行优先级最高的就绪任务。

uC/OS-III 运行多个任务具有相同的优先级。这时，它可以被设置为循环轮转调度。

调度发生在一些特定的时间点。

uC/OS-III 有 2 种调度方式：OSSched()被用于任务级。OSIntExit()被用于中断级。

8、上下文切换

当 uC/OS-III 转向执行另一个任务的时候，它保存了当前任务的 CPU 寄存器到堆栈。并从新任务的的堆栈中 CPU 寄存器载入 CPU。这个过程叫做上下文切换。

上下文切换需要一些开支。CPU 的寄存器越多，开支越大。上下文切换的时间基本取决于有多少个 CPU 寄存器需要被存储和载人。

上下文切换部分的代码是移植 uC/OS-III 时需编写的。该部分代码要适用于处理器。这些代码被放在 C 和汇编语言文件中：OS_CPU.H，OS_CPU_C.C，OS_CPU_A.ASM（章节 18“移植 uC/OS-III”会详细介绍）。根据不同架构的 CPU，移植 uC/OS-III 时需要注意很多细节。

在这个章节中，我们在常见的 CPU 架构中讨论上下文切换的过程如图 8-1。这个 CPU 包含了 16 个整数寄存器(R0 到 R15)和一个中断堆栈寄存器和一个状态寄存器{共 18 个寄存器，R0 到 R15,R14"和 SR}。每个寄存器都是 32 位的，且这 16 个整数寄存器都能存储数据或地址。指令指针寄存器是 R15，两个堆栈指针寄存器是 R14 和 R14"。R14 是指向任务的堆栈的指针 TSP，R14"是指向 ISR 的堆栈的指针 ISP。当 CPU 接收到一个中断自动切换到 ISR 堆栈。
{也就是处理器有独立的用于处理中断的堆栈}

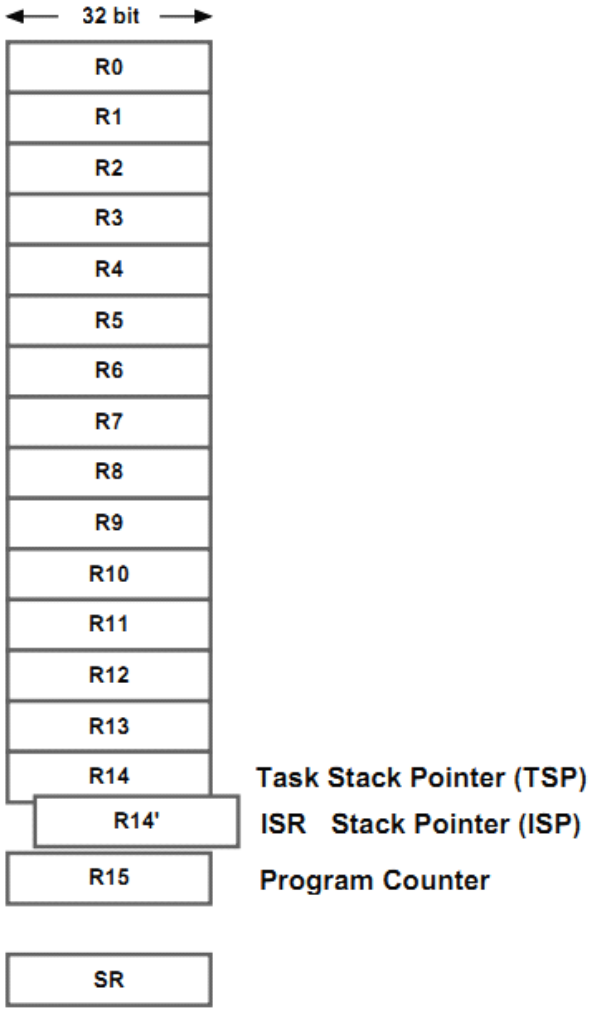


Figure 8-1 **Fictitious CPU**

在 uC/OS-III 中，任务切换时的堆栈设置类似于中断发生时的那样，所有的 CPU 寄存器都被保存。我们假定任务堆栈中的信息将要被载入到 CPU 中，如图 8-2。

TSP 指向任务堆栈中最后一个被保存的寄存器。程序指针寄存器和状态寄存器是最先被保存在任务堆栈中。事实上，当中断发生时这些是被 CPU 自动执行的。其它的寄存器通过软件被压入任务堆栈，TSP 不会被保存到堆栈，但会被保存到任务的 TCB。

ISP 指向当前中断堆栈的顶部。当中断服务程序被执行时，处理器把 R14"作为堆栈指针用于指向函数和局部参数。

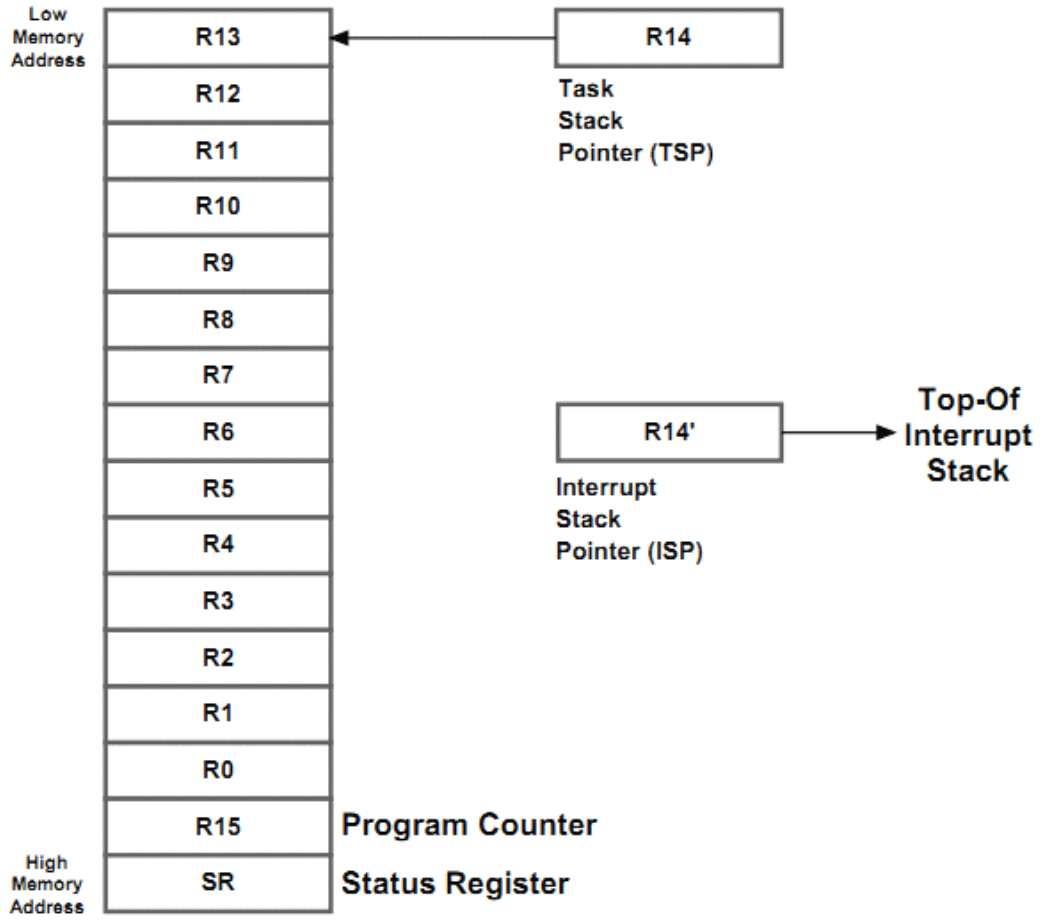


Figure 8-2 **CPU register stacking order of ready task**

有两种上下文切换方式：一个是任务级，一个是中断级。任务级切换通过调用 OSCtxSw()实现，实际上它是被宏 OS_TASK_SW()调用的。

中断级切换通过调用 OSIntCtxSw()实现。它是用汇编语言写的，保存于 OS_CPU_A.ASM。

8-1 OSCtxSw()

当有一个高优先级就绪任务需要被执行，任务级调度器会调用 OSCtxSw()。图 8-3 显示了在调用 OSCtxSw() 之前的一些变量和数据结构。

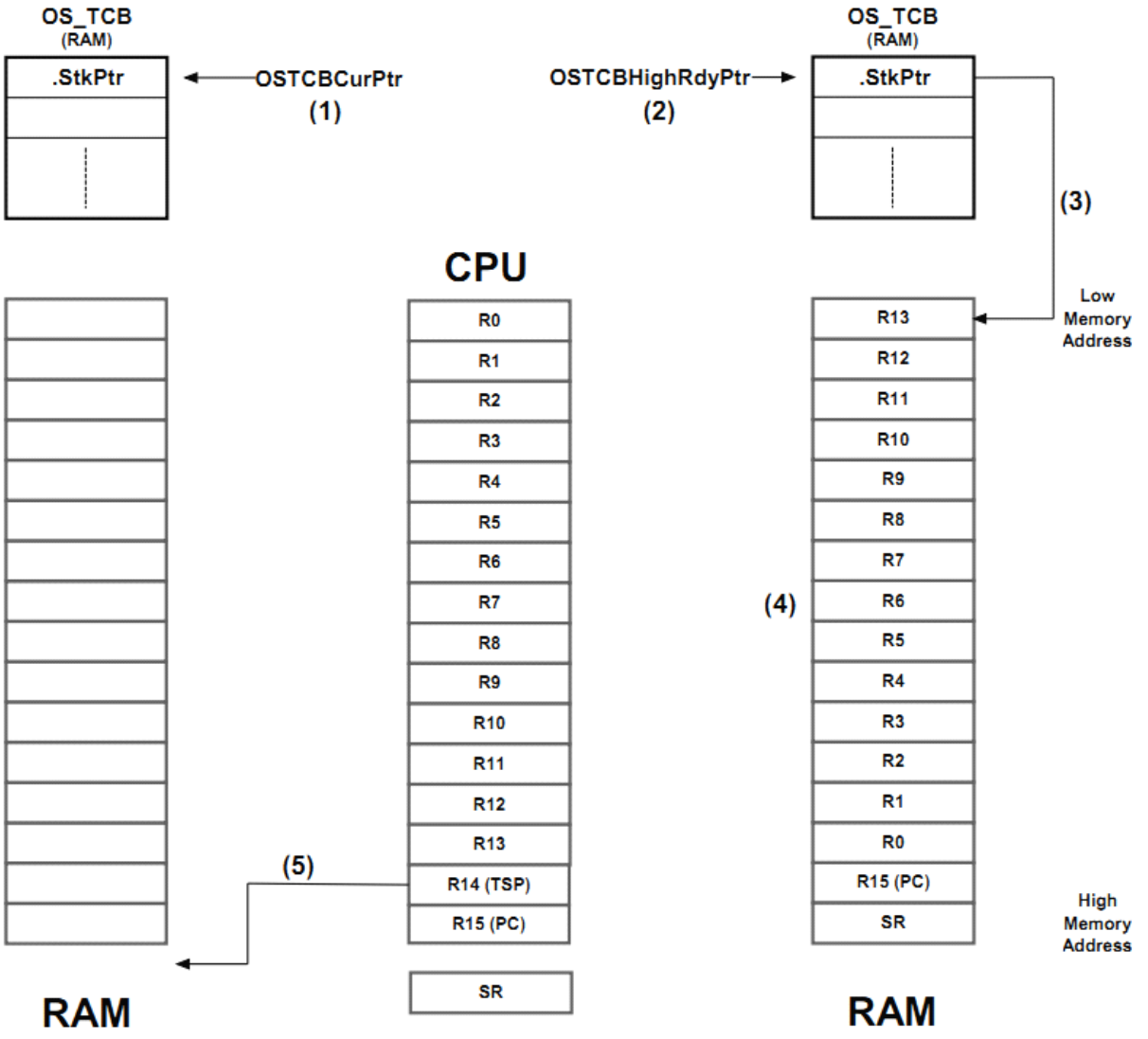


Figure 8-3 **Variables and data structures prior to calling OSCtxSw()**

F8-3（1）OSTCBCurPtr 指向了当前正在运行任务的 OS_TCB。然后 OSSchedule()被调用。

F8-3（2）通过 OSTCBHighRdyPtr，OSSched()找到将要被运行任务的 OS_TCB。

F8-3（3）OSTCBHighRdyPtr->StkPtr 指向将要被运行任务堆栈的顶部。{假定堆栈是从内存高地址向地地址生长的}

F8-3（4）执行任务的上下文切换时，所有的 CPU 寄存器被保存到该任务堆栈。由于保存了任务的当前状态，所以任务可以被恢复。

F8-3（5）调用 OSSched()后 CPU 的 TSP 会指向任务的堆栈。根据 OSCtxSw()的被调用形式。TSP 也可能指向 OSCtxSw()的返回地址。

图 8-4 展示了 OSCtxSw()实现上下文切换的相关步骤。

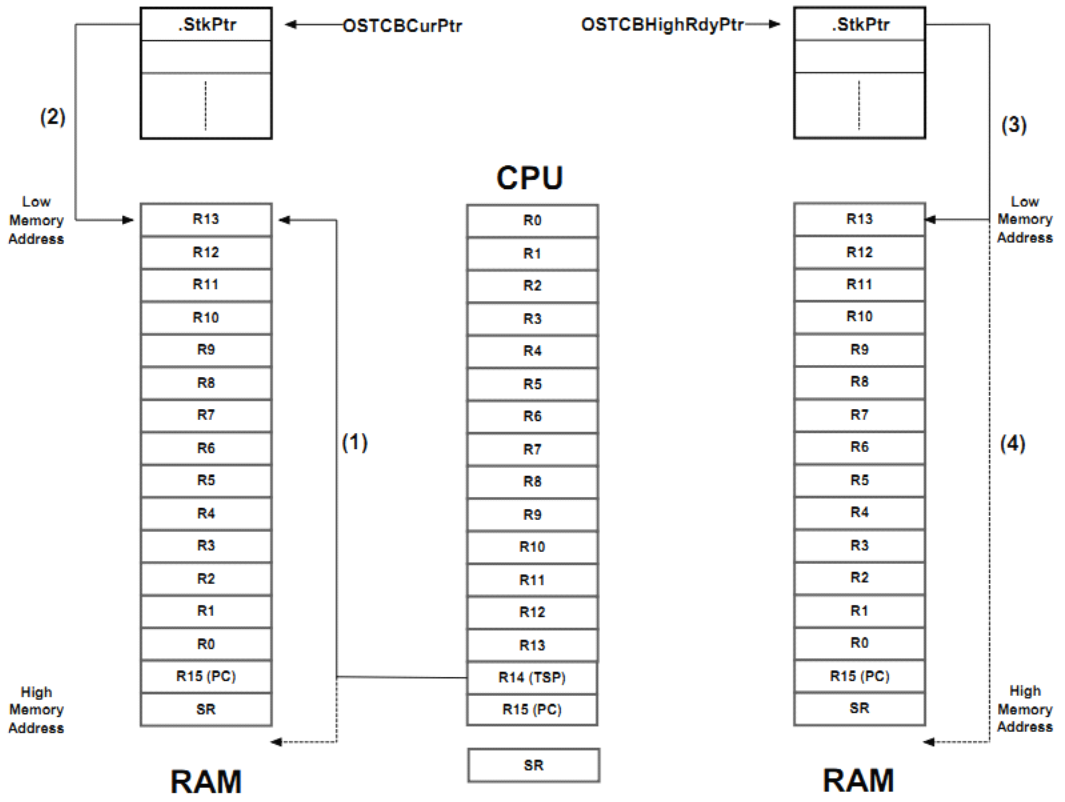


Figure 8-4 **Operations performed by OSCtxSw()**

F8-4（1）OSCtxSw()开始执行，保存状态寄存器和程序指针寄存器到当前的任务堆栈。保存的顺序与中断发生时 CPU 保存寄存器的顺序相同。假定 SR 先入栈，然后其它寄存器入栈。

F8-4（2）当任务被停止时，OSCtxSw()保存 CPU 的 TSP 到该任务的 OS_TCB 中，换句话说，OSTCBCurPtr->StkPtr=R14。

F8-4（3）然后 OSCtxSw()将新任务 OS_TCB 的顶部地址存入 CPU 的 TSP 中。换句话说，R14=OSTCBCurPtr->StkPtr。

F8-4（4）最后，OSCtxSw()将新任务堆栈中 R13~~R0 关内容载入 CPU 寄存器。再然后（此时的会 TSP 指向 PC，如图），通过一个中断返回指令{比如汇编中的 IRET}，程序指针寄存器和状态寄存器被恢复到 CPU 的寄存器中。

8-2 OSIntCtxSw()

在 ISR 中就绪了高优先级任务 B，ISR 返回时将不会回到中断前的任务 A，而是直接转向到执行高优先级任务 B。此时，由于中断产生时已经将任务 A 的状态保存在任务 A 的堆栈中，所以 ISR 返回时无需再保存任务 A 的状态，而是直接载入任务 B 的 CPU 寄存器到硬件 CPU 寄存器中即可。调用 OSIntCtxSw()执行该操作。图 8-5 显示了在调用 OSIntCtxSw() 之前的一些变量和数据结构。

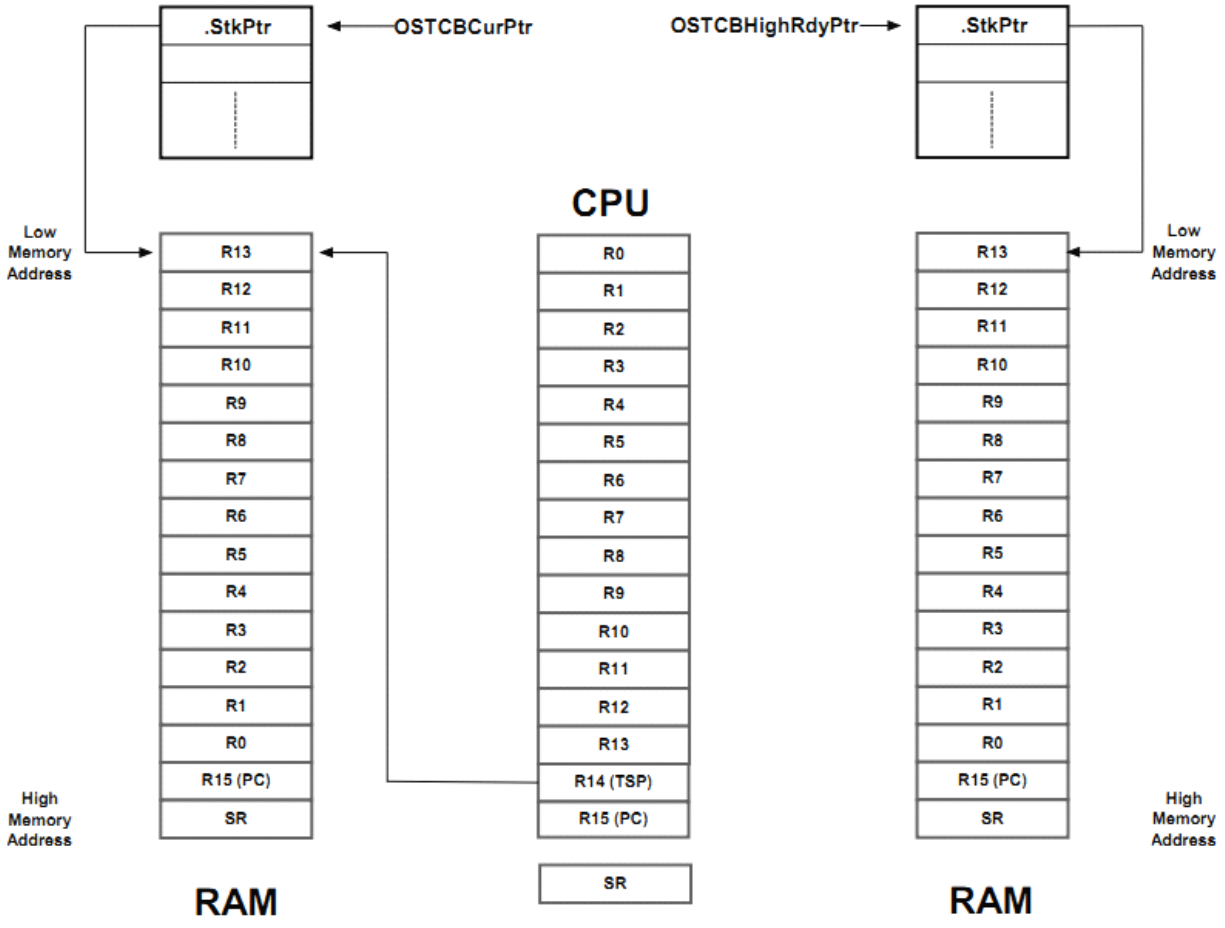


Figure 8-5 **Variables and data structures prior to calling OSIntCtxSw()**

在进入 ISR 之前 CPU 寄存器会被保存在任务 A 的堆栈中。因此, OSTCBCurPtr->StkPtr 包含了指向任务顶部寄存器的指针。

图 8-6 显示了 OSIntCtxSw()执行载入任务 B 的 CPU 寄存器步骤。这部分的步骤跟 OSCtxSw()的下半部分步骤是一样的。

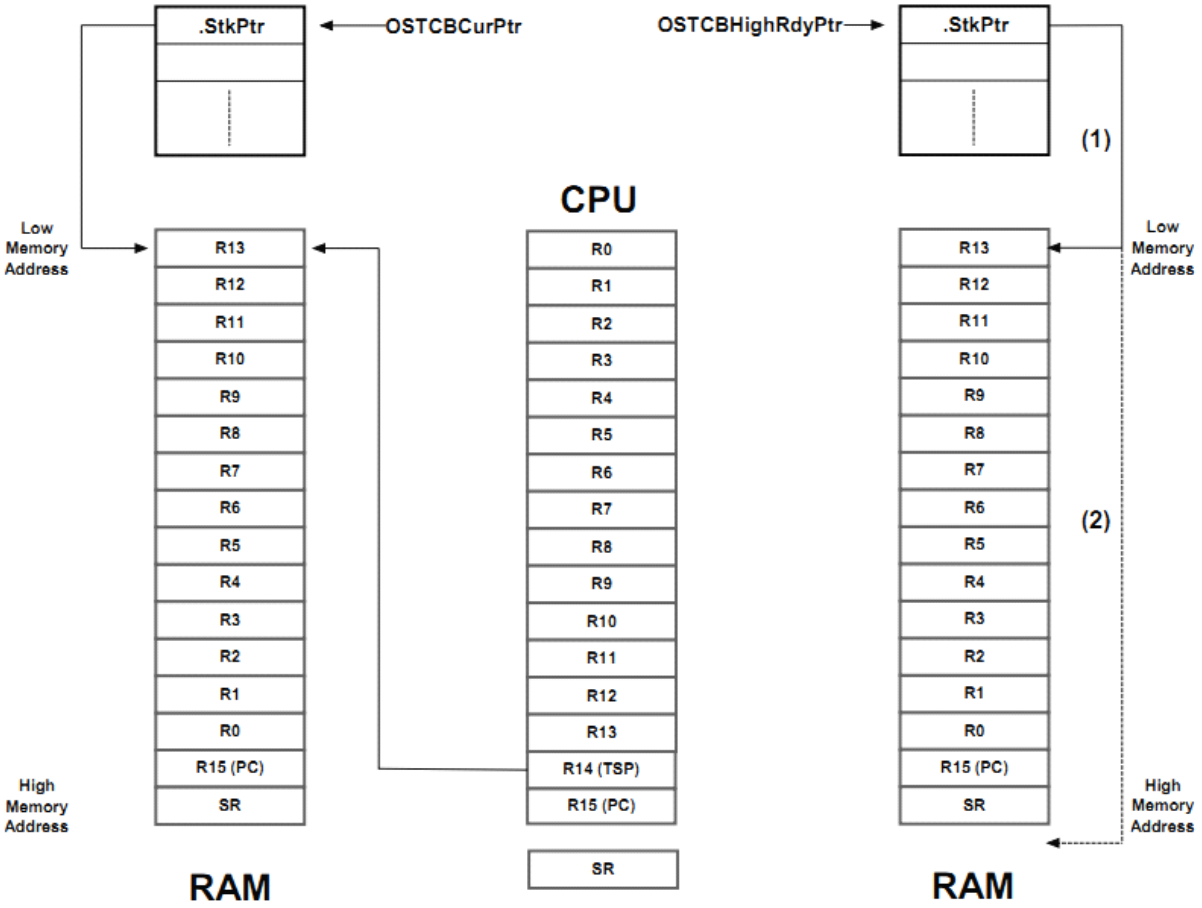


Figure 8-6 **Operations performed by OSIntCtxSw()**

F8-6 （1）OSIntCtxSw()将新任务 OS_TCB 的顶部堆栈地址值载入 CPU 中的 TSP。即 R14=OSTCBHighRdyPtr->StkPtr。

F8-6（2）然后 OSIntCtxSw()将新堆栈的相关内容载入到 CPU 寄存器中。通过一个中断返回指令，程序指针寄存器和状态寄存器被载入。

8-3 总结

上下文切换包括两部分内容，保存旧任务的内容，载入新任务的内容。

任务级切换时，通过调用 OSSched()实现。中断级切换时，通过调用 OSIntExit()实现。

OSSched()中调用 OSCtxSw()实现上下文切换。OSIntExit()中调用 OSIntCtxSw()实现上下文切换。然而，OSIntCtxSw()只需用做上下文切换的第二部分，因为中断时被中断任务的 CPU 寄存器已经被保存到被中断任务的堆栈中了。

9、中断管理

中断是硬件机制，用于通知 CPU 有异步事件发生。当中断被响应时，CPU 保存部分（或全部）寄存器值并跳转到中断服务程序（ISR）。ISR 响应这个事件，当 ISR 处理完成后，程序会返回中断前的任务或更高优先级的任务。

在实时系统中，关中断的时间越短越好。长时间关中断可能会导致中断来不及响应而重叠，即多次中断被当做一次中断。

通常处理器允许中断嵌套，它意味着当处理 ISR 时，可以有另一个中断发生。

在实时系统中，最大的关中断时间是一个重要的性能，关中断是为了处理临界段代码。

中断响应时间定义为：接收到中断到开始处理 ISR 中代码的这段时间。通常，中断时用户代码的上文（CPU 寄存器）会被放入堆栈中。

中断恢复时间定义为：执行完 ISR 中最后一句代码后到恢复到任务级代码的这段时间。

任务延迟时间定义为：中断发生到恢复到任务级代码的这段时间。

9-1 CPU 的中断处理

目前市场上有很多种架构的 CPU，处理器通常有多个中断源。例如，UART 中断、DMA 中断、ADC 中断、定时器中断等。

在大多数情况下，中断控制器捕捉所有的中断提交给处理器，如图 9-1 所示

中断器件标志中断处理器，然后中断处理器将优先级最高的中断提交给 CPU。

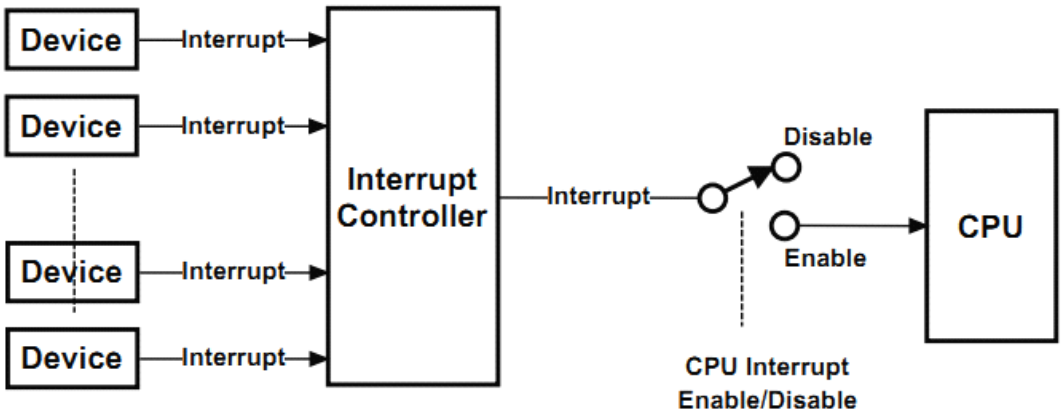


Figure 9-1 **Interrupt controllers**

现在的中断控制器都是智能的，允许定义中断优先级，记住哪些中断还处于挂起状态。在大多数情况下，中断控制器会直接提供对应 ISR 的向量地址给 CPU。

如果全局中断被关闭，CPU 就会忽视来自于中断控制器的中断请求，但这些请求会在中断控制器中被挂起直到 CPU 重新开启中断。

CPU 处理中断有两种模式：

1 所有的中断指向同一个 ISR

2 每个中断指向各自的 ISR

9-2 典型的中断服务程序

进入中断时 CPU 需执行的一些步骤。其中（2）（3)（10）（11）是处理器自动完成的。（4）~（9）需用户在 ISR 中编写。

```
MyISR:                                                          (1)
    Disable all interrupts;                                     (2)
    Save the CPU registers;                                     (3)
    OSIntNestingCtr++;                                          (4)
    if (OSIntNestingCtr == 1) {                                 (5)
        OSTCBCurPtr->StkPtr = Current task's CPU stack pointer register value;
    }
    Clear interrupting device;                                  (6)
    Re-enable interrupts (optional);                            (7)
    Call user ISR;                                              (8)
    OSIntExit();                                                (9)
    Restore the CPU registers;                                 (10)
    Return from interrupt;                                     (11)
```

Listing 9-1 **ISRs under µC/OS-III (assembly language)**

L9-1（1）ISR 通常是用汇编写的，也可以用 C 语言编写。

L9-2（2）在进入临界段之前，关中断。有些处理器进入中断服务程序时会自动关中断，有些则需要用户手动关中断。

L9-3（3）中断服务程序首先要做的就是保存当前任务的上文到任务堆栈。

有些 CPU 会自动将中断前的上文保存到中断堆栈，这样就节省了使用宝贵的任务堆栈。然而，对于 uC/OS-III，中断的上文需要被保存到任务堆栈。

如果处理器没有专用于指向 ISR 的堆栈指针{也就是 R14"}，那么它就需要用软件模拟。特别的，进入 ISR 时，保存当前的任务堆栈，然后将 R14"指向处理器的中断堆栈{也可以用户软件模拟}。当然，这意味着需要写一些附加的代码，这是非常有用的因为没必要再分配

给任务堆栈额外的空间（嵌套层数很多情况下所需的堆栈空间）。

L9-1（4）下一步，调用 OSIntEnter()，或将 OSIntNestingCtr 的值递增。将 OSIntNestingCtr 的值递增更高效。OSIntNestingCtr 保存了当前的嵌套层数。

L9-1（5)如果这是嵌套的第一层，将中断前的任务的 TSP 保存到任务的 OS_TCB。指针 OSTCBCurPtr 指向中断前的任务的 OS_TCB。OSTCBCurPtr->StkPtr 保存为 OS_TCB 中偏移量为 0 的地址。{OSTCBCurPtr->StkPtr 等价于 当前任务的 TCB->StkPtr}

L9-1（6）可以在此清除中断标志。

L9-1（7）如果用户相应支持嵌套中断，重新开启中断。这个步骤是可选的。

L9-1（8）此时，可以调用用户函数。然而，ISR 越短越好。事实上，最好在 ISR 发送信号量或消息给任务，让任务处理中断时需要执行的大部分操作。

ISR 可以调用 OSSemPost()、OSTaskSemPost()、OSFlagPost()、OSQPost()、OSTaskQPost()通知任务。ISR 越短越好。

L9-1（9）ISR 的工作完成后，用户必须调用 OSIntExit()告诉 uC/OS-III 中断服务程序已经完成。OSIntExit()中将 OSIntNestingCtr 递减，如果其值变为 0，就意味着 ISR 将会返回到任务级代码（否则返回前一层

的中断）。中断服务中可能使能了比原任务更高优先级的任务。这时 uC/OS-III 选择优先级最高的就绪任务或返回原任务。

L9-1（10）如果 ISR 使能了一个更低优先级的任务，OSIntExit()返回原任务。

L9-1（11）ISR 返回，恢复原任务的状态。

9-3 短中断服务程序（ISR）

上述顺序的代码是假定 ISR 会发送信号量或消息给任务。然而，在很多情况下，ISR 不需要发送通知任务，而是在 ISR 中直接完成需要的工作（假定需要完成的工作代码较短）。在这种情况下，ISR 的结构如列表 9-2 所示

```
MyShortISR:                                          (1)
    Save enough registers as needed by the ISR;      (2)
    Clear interrupting device;                       (3)
    DO NOT re-enable interrupts;                     (4)
    Call user ISR;                                   (5)
    Restore the saved CPU registers;                 (6)
    Return from interrupt;                           (7)
```

Listing 9-2 **Short ISRs with µC/OS-III**

L9-2（1）正如上面提到的，ISR 通常是用汇编写的。

L9-2（2）在这里，保存 CPU 寄存器。

L9-2（3）清除中断标志位，防止 ISR 返回时产生同样的中断。

L9-2（4）在这里不要重新开启中断。

L9-2（5）响应该中断需要的操作，调用用户程序。

L9-2（6）结束之后，重新恢复原任务的 CPU 寄存器。

L9-2（7）返回到原任务。

短 ISR 程序，如上所定义，是一个例外。它不遵循 uC/OS-III 的规则，所有 uC/OS-III 不知道 ISR 中发生了什么。

9-5 每个中断向量指向不同的地址

中断向量控制器中的中断向量都指向对应的 ISR 地址。ISR 中的程序尽可能用汇编写（如章节 9-2 所示）。当然，这样会导致编程的复杂化。然而，其中大部分的代码都是从另一个 ISR 中复制和黏贴的。除了一些用户要改变的代码。

9-6 直接提交和延迟提交

uC/OS-III 有两种方法处理来自于中断的时间；直接提交和延迟提交。通过 OS_CFH.H 中的 OS_CFG_ISR_POST_DEFERRED_EN 来选择。当设置为 0 时，uC/OS-III 使用直接提交方法。 当设置为 1 时，使用推迟提交方法。

根据应用选择一种方法。

9-6-1 直接提交

图 9-2 显示了直接提交方法的过程。

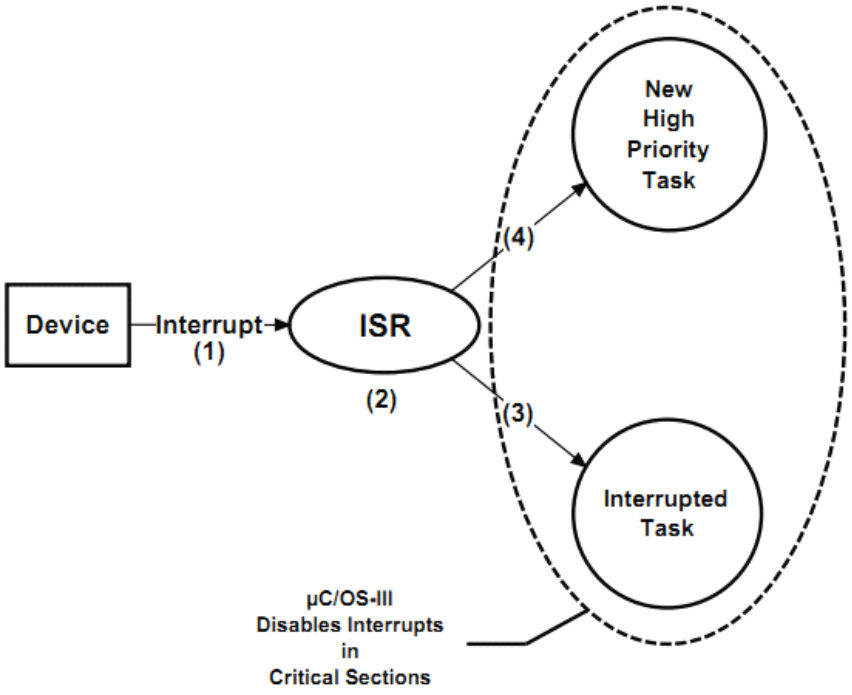


Figure 9-2 **Direct Post Method**

F9-2（1）中断产生

F9-2（2）中断服务程序开始执行。通常中断中包含着任务等待的事件。

F9-2（3）若 ISR 使能了低于或等于原任务优先级的任务，ISR 结束时，uC/OS-III 返回原任务并从被中断处继续执行。

F9-2（4）若 ISR 使能了高于原任务优先级的任务，ISR 结束后，进入调度。uC/OS-III 切换到高优先级任务。

F9-2（5）在直接提交方法中，uC/OS-III 必须保护好临界段（通过关中断方式）。

然而，如果用户程序调用某些 uC/OS-III 服务时，中断可能被关闭。当设置 OS_CFG_ISR_POST_DEFERRED_EN 为 0 时，在遇到临界段

代码时 uC/OS-III 会关闭中断。这样，中断就可能不被响应直到 uC/OS-III 重新开中断。

直接提交方法的一个重要的因素是 uC/OS-III 关中断的时间。这个容易被确定因为 uC/CPU 中提供了测量最大关中断时间的功能。这些代码可以被使能用于测试。测量时，让应用程序运行够长的时间并读取变量 CPU_IntDisMeasMaxRaw_cnts。这个变量的精度取决于用于测量的定时器速度。

中断延迟时间、中断响应时间 、中断恢复时间、任务延迟时间的计算代码如下：

中断延迟 = uC/OS-III 的最大关中断时间；

{假定中断在 uC/OS-III 中刚进入临界段时发生}

中断响应 = 中断延迟

+ 转向 ISR 时间

+ ISR 的前序时间

中断恢复 = 处理中断器件

+ 发送信号量或消息给任务

+ OSIntExit()

+ OSIntCtxSw()

任务延迟 = 中断响应

+ 中断恢复

+ 锁调度器时间

uC/OS-III 的 ISR 前序，ISR 后序，OSIntExit()，OSIntCtxSw()操作的时间可以被测量。

调用 OS_TS_GET()也可以测量“post”函数的处理时间。

在直接提交方式中，只有在处理定时器任务时才会锁调度器。如果定时器任务中创建的定时器不多的话，任务延迟将会很短。uC/OS-III 也可以测量锁调度器时间。

9-6-2 延迟提交方式

延迟提交方式（通过设置 OS_CFG_ISR_POST_DEFERRED_EN 为 1），代替了关中断方式用于处理临界段。uC/OS-III 锁住调度器，这可以避免其它任务访问临界段代码。延迟提交方式中，中断几乎没被关闭。然而，该方式有点复杂。如图 9-3 所示

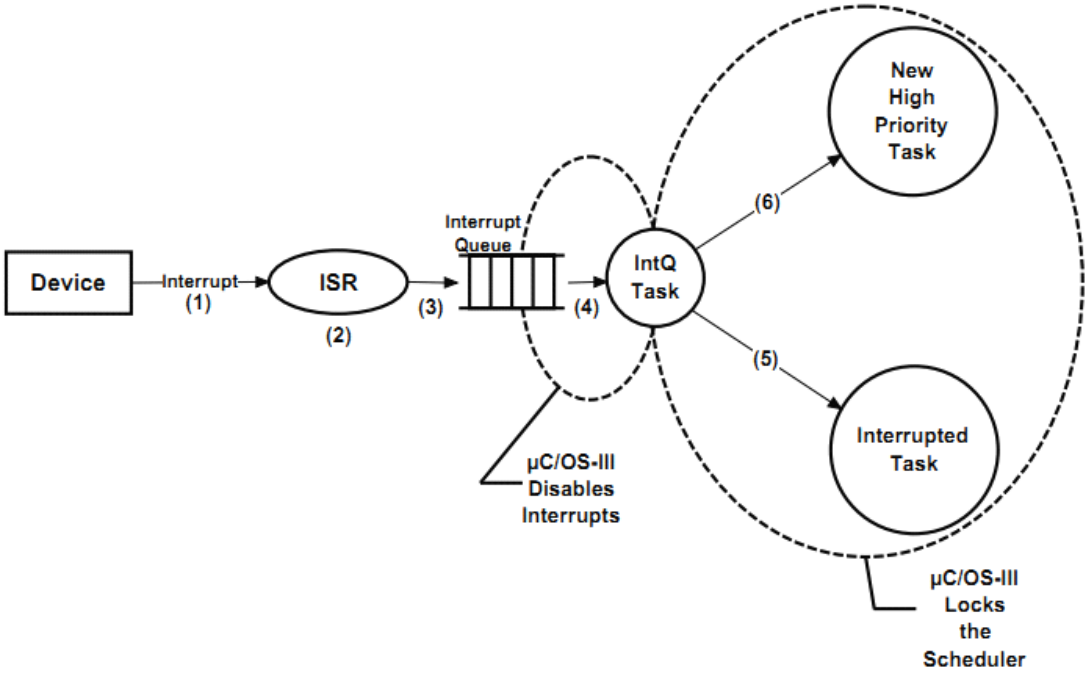


Figure 9-3 **Deferred Post Method block diagram**

F9-3（1）中断产生

F9-3（2）ISR 被执行。

F9-3（3）ISR 中调用"post"函数发送信号量或消息给任务。然而，它不是直接发送给任务，而是先发送到中断队列。然后中断处理函数被就绪。这是 uC/OS-III 的内部任务且具有最高优先级(优先级为 0)。

F9-3（4）ISR 的最后，uC/OS-III 会切换到中断处理任务，它将中断队列中消息发送给任务。在此，我们关闭中断防止在处理中断队列时被另外的中断程序打断。最后，该任务使能中断，锁住调度器，提交信息。信息是在任务级被提交的。这样，就是在任务级完成临界段的处理了。

F9-3（5）当中断处理任务清空中断队列时，它就会将自己停止，重新开启调度器，调用调度器选择下一个任务运行。

F9-3（6）如果优先级更高的任务被使能，uC/OS-III 会上换到该任务。

执行一些额外的处理，在处理临界段时可以避免关中断。额外的处理只包括调用"post"函数，将消息或信号量插入队列，并从队列中提取，并执行额外的上下文切换。

类似于直接提交方式，

中断延迟 = uC/OS-III 的最大关中断时间；

中断响应 = 中断延迟
+ 转向 ISR 时间
+ ISR 的前序时间

中断恢复 = 处理中断器件
+ 发送信号量或消息给中断队列
+ OSIntExit()
+ OSIntCtxSw()切换到中断队列处理任务

任务延迟 = 中断响应
+ 中断恢复
+ 将信号量或消息提交给任务
+ 上下文切换到任务
+ 锁调度器时间

事实上，延迟提交方式中的"post"函数会很短，因为这包括复制"post"函数和该函数的参数到中断队列中。

不同的是，延迟提交方式中关中断的时间是非常短的。然而，任务延迟时间变长因为 uC/OS-III 会锁住调度器，并访问临界段。

9-7 直接提交 VS 延迟提交

在直接提交方式中，uC/OS-III 访问临界段时关中断。然而，在延迟提交方式中，uC/OS-III 访问临界段时锁调度器。

在延迟提交方式中，访问中断队列时 uC/OS-III 需要关中断。然而，这段关中断时间是非常短的且是相当固定的。

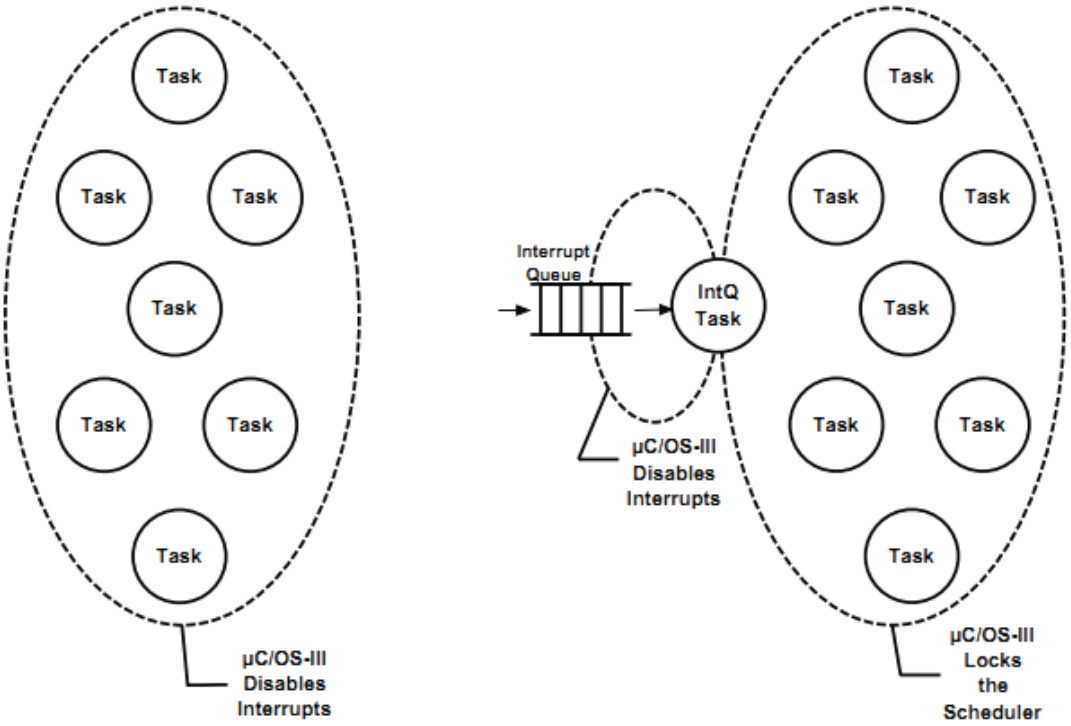


Figure 9-4 **Direct vs. Deferred Post Methods**

如果在应用中关中断时间是关键性的，因为应用中有非常频繁的中断源，且应用不能接受直接提交方式那样较长的关中断时间。推荐使用延迟提交方式。

然而，如果你想要使用表 9-1 的一些功能，那么就考虑用延迟提交方式。

功能	原因
多个任务具有相同优先级	这 uC/OS-III 的一个重要功能，多个任务具有相关优先级会产生一个长临界段。然而，当很少任务具有相同优先级时，中断延时会很小。当多任务具有不同优先级时，可以用关中断方法。

事件标志组 详见章节 14 “同步”	如果多个任务等待不同的事件，遍历这些等待事件的任务需要很多处理时间，意味着会产生长临界段。如果很少的任务在等待事件，临界段会很短，那么可以使用关中断的方法处理。
任务等待多个对象 详见章节 16	挂起多个对象是 uC/OS-III 提供的很复杂的功能，它需要很长的临界段。如果这样，推荐使用关调度器的方法。如果挂起较少，关中断也是可以的。
广播消息 详见附录 A 中的 OSSemPost()和 OSQPost()	uC/OS-III 在处理广播消息时通过关调度器保护临界段。当不使用广播功能的时候，你可以用关中断方法

表 9-1 当使用直接提交方式时避免使用这些功能

9-8 系统时基

uC/OS-III 需要一个能提供周期性时间的时基源，叫做系统时基。

硬件定时器可以被设置为每秒产生 10 到 1000Hz 的中断提供系统时基。也可以从交流电中获得 50Hz 到 60Hz 的时基源。事实上，也可以从交流电中获得 100Hz 到 120Hz 的过零点作为时基。

时基可以看做是系统的心跳。它的速率决定于时基源。然而，时基速率越快，系统的额外支出就越大。

时基中断程序必须调用 OSTimeTick()。OSTimeTick()的伪代码如列表 9-4 所示。

```
void  OSTimeTick (void)
{
    OSTimeTickHook();                                   (1)
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    Get timestamp;                                      (2)
    Post "time tick" to the Interrupt Queue;
#else
    Signal the Tick Task;                               (3)
    Run the round-robin scheduling algorithm;           (4)
    Signal the timer task;                              (5)
#endif
}
```

Listing 9-4 **OSTimeTick() pseudocode**

L9-4（1）时基中断程序首先调用一个 hook 函数，OSTimeTickHook()。它允许当时基中断出现时用户扩展一些操作。Hook 函数在时基中断发生时首先被调用的，用它访问一些周期性元素。如读取传感器（需被周期性检测）、更新 PWM 寄存器等等。

L9-4（2）如果 uC/OS-III 被设置为延迟提交方式，uC/OS-III 读取当前时间戳并放置到中断处理队列。然后中断队列处理任务发送信号量给时基任务。

L9-4（3）如果 uC/OS-III 被设置为直接提交方式，时基 ISR 直接发送信号量给时基任务。

L9-4（4）使用循环轮转算法检测是否当前任务是否期满。

L9-4（5）标记定时器任务，定时器任务也用时基任务的时基作为基准。

uC/OS-III 必须有系统时基是普遍的误解。事实上，很多低功耗应用中没有系统时基，因为需额外的能量用于维护时基源。换句话说，将能量用于维护时基源是不合理的。因为 uC/OS-III 是一个可抢占式内核，一个事件可以唤醒进入低功耗模式处理器（按键或其它事件）。没有时基意味着用户不能再对任务进行延时或超时设置。用户在研发低功耗产品时可以考虑这个特性。

9-9 总结

uC/OS-III 提供管理中断的服务。ISR 应该越短越好，发送信号量或消息给任务，让该任务实现该中断所需的大部分操作。即尽量让操作在任务级完成。

ISR 也可以不发送信号量或消息给任务。ISR 自己处理响应操作。但 ISR 越短越好。

uC/OS-III 允许处理器中断时指向同一个 ISR，或是指向各自的 ISR。

uC/OS-III 支持两种方式:直接提交方式和延迟提交方式。直接提交方式需要关中断保护临界段。延迟提交方式需要关调度器保护临界段。

uC/OS-III 需要时基源，用于任务的延时和超时功能。

10、挂起队列

当任务等待信号量、mutex、事件标志组、消息队列时，该任务会被放入挂起队列。

参见	挂起队列	内核对象
章节 13“资源管理”	信号量、mutex	OS_SEM、OS_MUTEX
章节 14“同步”	信号量、事件标志组	OS_SEM、OS_FLAG_GRP
章节 15“消息提交”	消息队列	OS_Q

表 10-1 挂起队列与内核对象

挂起队列类似于就绪队列，挂起队列中放的是等待内核对象的任务。另外，任务在挂起队列中是根据优先级分类的。高优先级任务被放置在队列的头部，低优先级任务被放置在队列的尾部。

挂起队列是一个 OS_PEND_LIST 类型的数据结构，它包含了三部分内容，如图 10-1

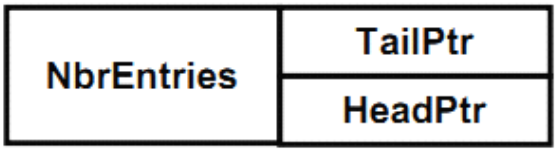


Figure 10-1 **Pend List**

.NbrEntries 挂起队列中有几个任务。

.TailPtr 指向队列的尾部（最低优先级的任务）

.HeadPtr 指向队列的首部（最高优先级的任务）

图 10-2 显示了每个内核对象中的包含了三个相同的部分，我们把这三个部分命名为 OS_PEND_OBJ。注意的是，第一个部分“Type”用于区别这个内核对象的类型：信号量、mutex、事件标记组、消息队列。

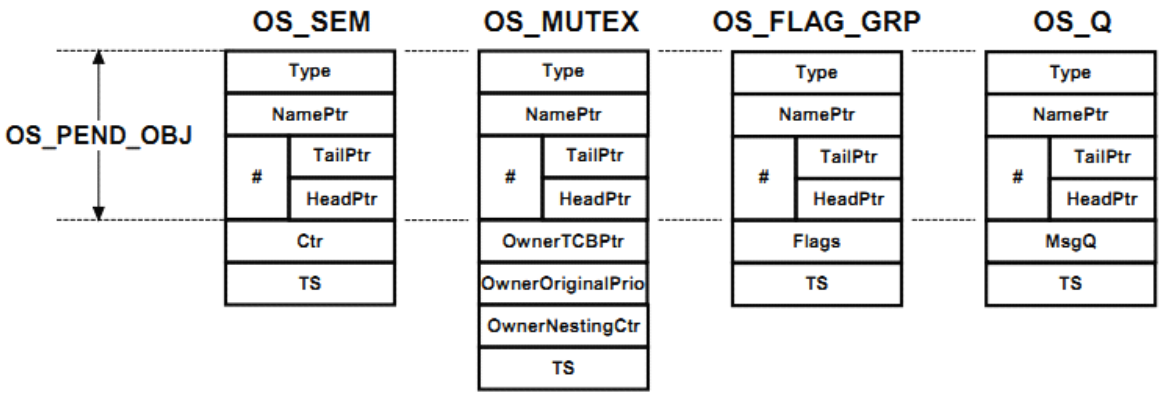


Figure 10-2 **OS_PEND_OBJ at the beginning of certain kernel objects**

表 10-2 显示了当内核对象被创建时"Type"部分被设置的值。调试时可以方便用户识别内核对象的类型。

Kernel Object	**Type**
Semaphore	'S' 'E' 'M' 'A'
Mutual Exclusion Semaphore	'M' 'U' 'T' 'X'
Event Flag Group	'F' 'L' 'A' 'G'
Message Queue	'Q' 'U' 'E' 'U'

Table 10-2 **Kernel objects with initialized “Type” field**

事实上，挂起队列中不是指向任务的 OS_TCB，而是指向 OS_PEND_DATA。如图 10-3 所示。当任务被放入挂起队列时，OS_PEND_DATA 结构体被动态地分配到该任务的堆栈中。这意味着任务堆栈中需有足够的空间用于存储这个结构体。

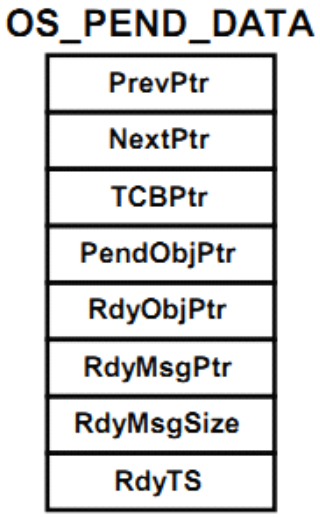


Figure 10-3 **Pend Data**

.PrevPtr　它指向正在队列中等待的高优先级或同等优先级的任务。

.NextPtr　它指向正在队列中等待的低优先级的任务。

.TCBPtr　指向该任务的 OS_TCB。

.PendObjPtr　指向任务所等待的内核对象。换句话说，这个指针可以指向 OS_SEM，OS_MUTEX，OS_FLAG_GRP，OS_Q。

（内核对象中都有一个相同的结构体 OS_PEND_OBJ）

.RdyObjPtr　如果任务等待多个内核对象，该指针指向任务被放入挂起列表前已经被提交的内核对象（详见章节 16“挂起多个对象”）。

.RdyMsgPtr　如果任务等待多个内核对象，该指针指向任务被放入挂起列表后被提交的内核对象（详见章节 16“挂起多个对象”）。

.RdyTS　　它存储了内核对象被提交时的时间戳。任务等待多个内核对象时会用到这个变量。（详见章节 16“挂起多个对象”）。

图 10-4 展示了当任务被插入挂起队列时，OS_PEND_DATA 结构体之间是相互联系的。该图假定两个任务等待同一个信号量。

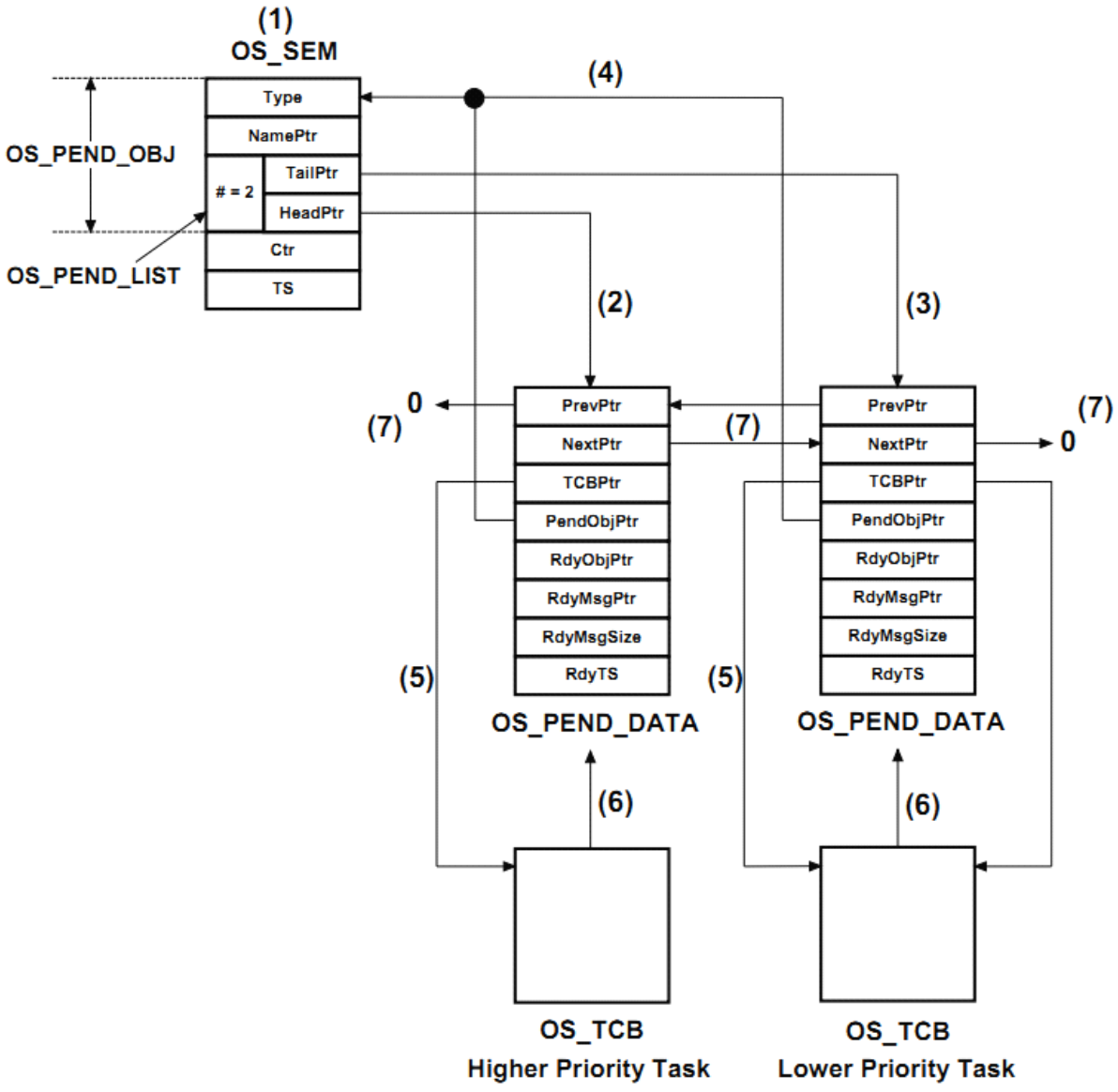


Figure 10-4 **Pend Data**

F10-4（1）结构体 OS_SEM 中包含了一个 OS_PEND_OBJ。OS_PEND_OBJ 中包含了 OS_PEND_LIST，其中.NbrEntries 值为 2 表示有两个任务正在等待该信号量。

F10-4（2）.HeadPtr 指向高优先级任务的结构体 OS_PEND_DATA。

F10-4（3）.TailPtr　指向低优先级任务的结构体 OS_PEND_DATA。

F10-4（4）这两个任务的结构体 OS_PEND_DATA 中分别通过指针 PendObjPtr 指回 OS_SEM 结构体。该指针认为自己指向的是 OS_PEND_OBJ{也就是该指针是以 OS_PEND_OBJ 类型定义的}。通过检查 OS_PEND_OBJ 中的.Type 就可以知道该内核对象是信号量。

F10-4（5）每个 OS_PEND_DATA 结构体分别指向各自的 OS_TCB。换句话说，这样可以知道哪个任务在等待这个信号量。

F10-4（6）每个任务指针指回 OS_PEND_DATA 结构体。

F10-4（7）最后，OS_PEND_DATA 结构体组织成一个双向列表，uC/OS-III 就可以方便地添加或移除列表中的记录。

表 10-3 显示了 uC/OS-III 用于维护挂起队列的函数。这些函数都是 uC/OS-III 的内部函数（代码在 OS_CORE.C 中），用户不能调用它们。

函数	功能
OS_PendListChangePrio()	改变挂起队列中的任务的优先级
OS_PendListInit()	初始化挂起队列
OS_PendListInsertHead()	插入一个 OS_PEND_DATA 到队首
OS_PendListInsertPrio()	根据优先级插入一个 OS_PEND_DATA 到队列
OS_PendListRemove()	队列中移除多个 OS_PEND_DATA
OS_PendListRemove1()	队列中移除一个 OS_PEND_DATA

表 10-3 挂起队列相关函数

10-1 总结

uC/OS-III 将等待信号量、mutex、事件标志组、消息队列的任务存储到挂起队列中。

挂起队列中包括数据结构 OS_PEND_LIST。它包含在另一个叫做 OS_PEND_OBJ 的数据结构中

任务不是直接链接到挂起队列中，而是通过叫做 OS_PEND_DATA 的数据结构作为媒介。这个媒介在任务被放入挂起队列中分配到任务堆栈的。

用户代码不能访问挂起队列，必须调用 uC/OS-III 提供的函数。

11、时间管理

uC/OS-III 为用户提供了与时间管理相关的服务。

在第 9 章“中断管理”中，在 uC/OS-III 中设置了能提供时基中断的中断源。该中断源提供 10Hz 到 1000Hz 之间的中断（需设置 OS_CFG_APP.H 中的 OS_CFG_TICK_RATE_HZ 为中断源提供的频率）。然而，频率越高，CPU 额外的消耗就越多。

uC/OS-III 提供了一些与时间相关的函数如表 11-1,这些代码在 OS_TIME.C 中

函数名	功能
OSTimeDly()	延时执行任务 n 个时基
OSTimeDlyHMSM()	延时执行任务 HH:MM:SS.mmm
OSTimeDlyResume()	恢复处于延时状态的任务
OSTimeGet()	获得当前的时基计数值
OSTimeGet()	设置当前的时基计数值
OSTimeTick()	用于标记，表示发生了一个时基中断

表 11-1 时间服务相关的 API 总结

详见附录 A“uC/OS-III 的 API 参考手册”。

11-1 OSTimeDly()

任务调用这个函数后就会被挂起直到期满。这个函数可以有三种模式：相对延时模式，周期性延时模式，绝对定时模式。

列表 11-1 显示了 OSTimeDly()的相关模式。

```
void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    while (DEF_ON) {
        :
        :
        OSTimeDly(2,                              (1)
                  OS_OPT_TIME_DLY,                (2)
                  &err);                          (3)
        /* Check "err" */                         (4)
        :
        :
    }
}
```

Listing 11-1 **OSTimeDly() - Relative**

L11-1（1）第一个参数是任务的延时时基数。如果时基速率被设置为 1000Hz，任务会每次执行都会被延时大约 2 毫秒。然而，并不是精确地延时 2 个时基，因为任务被挂起后是检测时基中断发生的次数与任务的延时值是否相同来判断是否超时的。也就是说，当任务在时基中断将要到来时被挂起，那么实际的延时时基会少 1 个时基。

L11-1（2）参数为 OS_OPT_TIME_DLY 表明用户选择的相对延时模式。

L11-1（3）如大多数 uC/OS-III 函数一样，错误代号会被返回。当所有的参数都是有效时会返回 OS_ERR_NONE。

L11-1（4）当返回不为 OS_ERR_NONE 时，OSTimeDly()将不会执行延时操作。

如上所述，延时时间不是精确的。详见图 11-1

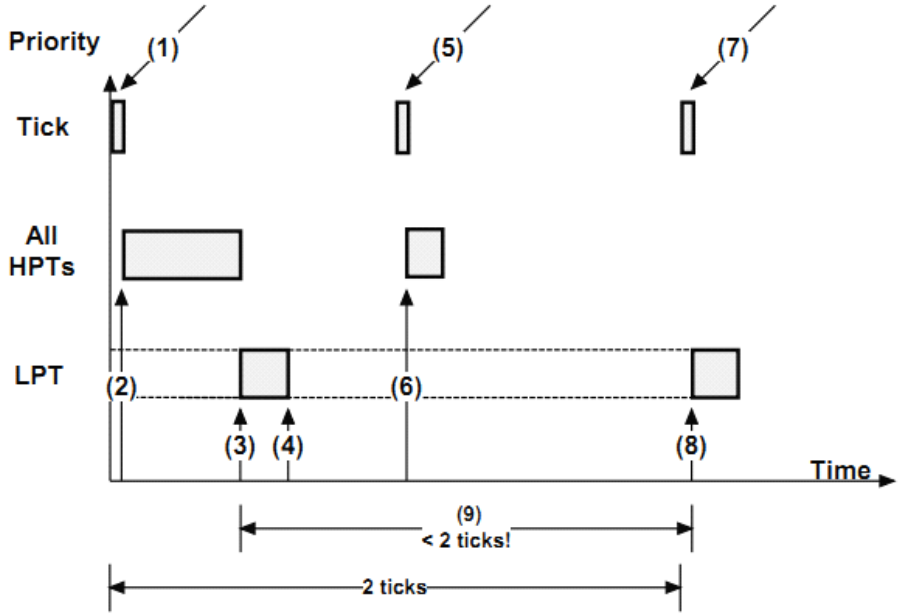


Figure 11-1 **OSTimeDly() - Relative**

HPT: 高优先级任务，LPT 低优先级任务

F11-1（1）时基中断产生，进入时基 ISR。

F11-1（2）ISR 结束后，切换到高优先级任务。这段执行时间是不可预测的。

F11-1（3）当高优先级任务执行完毕后，uC/OS-III 转到想要调用 OSTimeDly()的任务。

F11-1（4）任务调用 OSTimeDly()，并设置为等待两个时基。此时，uC/OS-III 会将这个任务放入时基队列。等待期满时任务不会占用 CPU 时间。

F11-1（5）下一个时基中断产生。当有高优先级的任务等待此次时基中断时，高优先级任务会被执行。

F11-1（6）执行高优先级任务。

F11-1（7）再下一个时基中断产生。当有高优先级的任务等待此次时基中断时，高优先级任务会被执行。

F11-1（8）因为此刻没有高优先级的中断就绪，uC/OS-III 上下文切换到低优先级任务，也就是调用了 OSTimeDly()的任务。

F11-1（9）时基中断后，因为高优先级任务需要被执行，所以该任务的延时时间不会是精确的 2 个时基。

需要考虑的是：任务调用 OSTimeDly()延时 2 个时基时，若下一个时基中断会迅速地产生，如上图。在这种情况下，要延时 2 个时基时，就需要设置延时 3 个时基。

当参数设置为了 OT_OPT_TIME_PERIDSIC 时，OSTimeDly()被设置为周期性延时模式。

任务设置匹配值决定了任务被唤醒的周期。如图 11-2 所示。当匹配值等于 OSTickCtr 时，任务被唤醒。

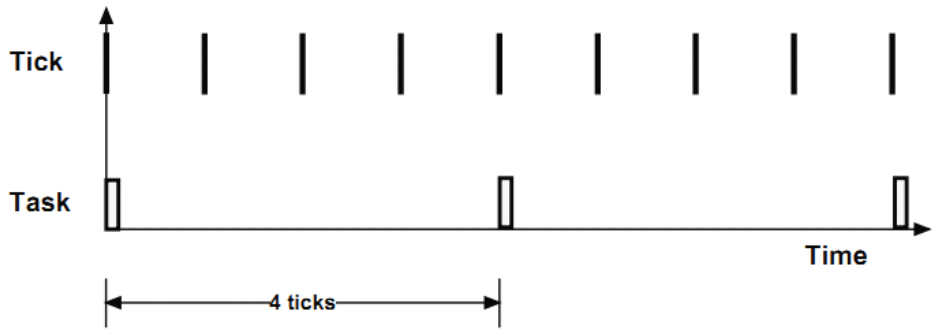


Figure 11-2 **OSTimeDly() - Periodic**

```
void  MyTask (void *p_arg)
{
    OS_ERR   err;
    :
    :
    while (DEF_ON) {
        OSTimeDly(4,                              (1)
                  OS_OPT_TIME_PERIODIC,           (2)
                  &err);
        /* Check "err" */                         (3)
        :
        :
    }
}
```

Listing 11-2 **OSTimeDly() - Periodic**

L11-2（1）第一个参数设置了任务执行的周期，如上被设置为 4 个时基。

L11-2（2）第二个参数 OS_OPT_TIME_PERIODIC 表明任务被设置为周期性运行。

L11-2（3）检查 uC/OS-III 返回的错误代号。

相对延时模式和周期性延时模式看起来是不一样的，但是它们类似的。它们都可能丢失一个时基当有高优先级任务被执行很长时间时。

最后，你可以使用绝对时间模式处理对时间要求很高的任务。例如，必须产品在上电的第 10 秒关闭 LED。在这种情况下，你就需要设置为绝对时间模式 OS_OPT_TIME_MATCH。设置 OSTickCtr 值为 10 乘以时基频率。

总结：当 OSTickCtr 与以下值时匹配时任务被唤醒：

选项	唤醒时的匹配值
OS_OPT_TIME_DLY	OSTickCtr + dly
OS_OPT_TIME_PERIODIC	OSTCBCurPtr->TickCtrPrev+dly
OS_OPT_TIME_MATCH	dly

dly 为上述函数的第一个参数

11-2 OSTimeDlyHMSM()

任务可以调用这个函数为任务设置延时，这个函数更“友好”于用户。特别的，可以设置为小时，分钟，秒，毫秒（HMSM 由此四个英文首字母得来）。这个函数只在相对延时模式下运行。

列表 L11-3 介绍了 OSTimeDlyHMSM()。

```
void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    while (DEF_ON) {
        :
        :
        OSTimeDlyHMSM(0,                          (1)
                      0,
                      1,
                      0,
                      OS_OPT_TIME_HMSM_STRICT,    (2)
                      &err);                      (3)
        /* Check "err" */
        :
        :
    }
}
```

Listing 11-3 **OSTimeDlyHMSM()**

L11-3（1）这四个参数设置了延时的时间(分别对应为时、分、秒、毫秒)。在这个例子中，设置了延时 1 秒。延时的分辨率决定于时基频率。例如，如果时基频率为 1000Hz 那么延时的分辨率为 1 毫秒。如果时基的频率为 100Hz 那么延时的分辨率为 10 毫秒。同样的，延时时间不会很精确。

L11-3（2）设置 OS_OPT_TIME_HMSM_STRICT 后会检测函数的参数是否合理。小时的范围是 0 到 99，分的范围是 0 到 59，秒的范围是 0 到 59，毫秒的范围是 0 到 999。

如果设置为 OS_OPT_TIME_HMSM_NON_STRICT，函数会接受参数的范围变大。小时的范围是 0 到 999，分的范围是 0 到 9999，秒的范围是 0 到 65535，毫秒的范围是 0 到 4294967295。

限制小时范围为 0 到 999 的原因是：一般是用 32 位的数记录时基值的。如果时基的频率为 1000Hz，那么最多能计数 4294967 秒，大约 1193 小时。因此设置 999 小时为上限。

L11-3（3）如大多数 uC/OS-III 函数一样返回一个错误代号。

虽然 uC/OS-III 允许任务有很长的延时时间，但不推荐任务长时间延时。因为任务可能早就被删除了。

OSTimeDly()和 OSTimeDlyHMSM()经常被用于创建周期性的任务。例如，设置任务每 50 毫秒扫描一次键盘、每 10 毫秒读取 AD 输入等。

11-3 OSTimeDlyResume()

任务可以调用 OSTimeDlyResume()恢复其它被 OSTimeDly()或 OSTimeDlyHMSM()延时的任务。列表 11-4 显示了如何使用 OSTimeDlyResume()。

```
OS_TCB  MyTaskTCB;


void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    while (1) {
        :
        :
        OSTimeDly(10,
                  OS_OPT_TIME_DLY,
                  &err);
        /* Check "err" */
        :
        :
    }
}


void  MyOtherTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    while (1) {
        :
        :
        OSTimeDlyResume(&MyTaskTCB,
                        &err);
        /* Check "err" */
        :
        :
    }
}
```

Listing 11-4 **OSTimeDlyResume()**

11-4 OSTimeSet()和 OSTimeGet()

每个时基中断发生时 uC/OS-III 会递增时基计数值。通过这个计数值能大概看出系统上电后经过了多长时间。

OSTimeGet()能获得时基计数值。

OSTimeSet()允许用户设置时基计数值。虽然 uC/OS-III 允许这种操作，但调用这个函数时需慎重。

11-5 OSTimeTick()

当时基中断发生时，时基 ISR 必须调用这个函数。uC/OS-III 用这个函数更新时基计数值。OSTimeTick()是 uC/OS-III 的内部函数。

11-6 总结

uC/OS-III 提供了延时函数，用户可以以时基为单位将任务延时。可以以小时、分、秒、毫秒为单位将任务延时。延时的精确地依赖于时基的频率，延时的最小分辨率为时基。

用户可以调用 OSTimeDlyResume()使处于延时状态的任务恢复。

通过 OSTimeSet() 设置时基计数值，通过 OSTimeGet()获取时基计数值。

12、软件定时器管理

uC/OS-III 提供了软件定时器服务（相关代码在 OS_TMR.C 中）。当设置 OS_CFG.H 中的 OS_CFG_TMR_EN 为 1 时软件定时器服务被使能。{为了简洁，这章中的定时器均为软件定时器}

定时器递减其计数值，当计数值为 0 的，就是定时期满的时候。通过回调函数执行相应的操作（打开或关闭 LED、开启电机或其它操作）。回调函数是用户定义的，当定时器期满时可以被调用。然而，不要用回调函数调用如下函数 OSTimeDly()，OSTimeDlyHMSM()，OS???Pend()，或其它能导致该定时器任务被阻塞或被删除的函数。

应用中可以定义任意个定时器（限制于处理器的 RAM）。uC/OS-III 的定时器服务通过调用 OSTmr???()开始。详见附录 A

uC/OS-III 定时器的分辨率决定于时基频率，也就是变量 OS_CFG_TMR_TASK_RATE_HZ 的值，它是以 Hz 为单位的。如果时基任务的频率设置为 10Hz，所有定时器的分辨率为十分之一秒。事实上，这是用于定时器的推荐值。定时器用于不精确时间尺度的任务。

uC/OS-III 提供了一些函数用于管理定时器如表 12-1

函数名	功能
OSTmrCreate()	创建和设置定时器
OSTmrDel()	删除一个定时器
OSTmrRemainGet()	获得定时器的剩余期限值
OSTmrStart()	开始定时器运行
OSTmrStateGet()	获得定时器当前的状态
OSTmrStop()	暂停定时器

表 12-1　定时器的 API 总结

定时器被使用之前必须被创建。通过调用 OSTmrCreate()，并设置这个函数的相关参数。一旦定时器的操作模式被设置，就不能被改动直到定时器被删除并被重新创建。OSTmrCreate()的原型如下：

```
void  OSTmrCreate (OS_TMR              *p_tmr,           /* Pointer to timer     */
                   CPU_CHAR            *p_name,          /* Name of timer, ASCII */
                   OS_TICK              dly,             /* Initial delay        */
                   OS_TICK              period,          /* Repeat period        */
                   OS_OPT               opt,             /* Options              */
                   OS_TMR_CALLBACK_PTR  p_callback,      /* Fnct to call at 0    */
                   void                *p_callback_arg,  /* Arg. to callback     */
                   OS_ERR              *p_err)
```

一旦定时器被创建，它可以被开始或停止任意次。定时器可以被设置为 3 种模式：一次性定时模式，无初始定时周期模式（没有初始的定时），有初始定时周期模式（有初始的定时）。

12-1 一次性定时模式

正如其名字所表达，定时器会递减被设置初始的定时值，当该值为 0 时就会调用回调函数并停止定时器。如图 12-1。初始的定时值通过调用 OSTmrSrart()设置，延时期满时，回调函数被调用（假定回调函数在定时器创建的时候被提供）。完成之后，定时器不做任务事情直到调用 OSTmrStart()被重新开启。

通过调用 OSTmrStop()停止定时器。

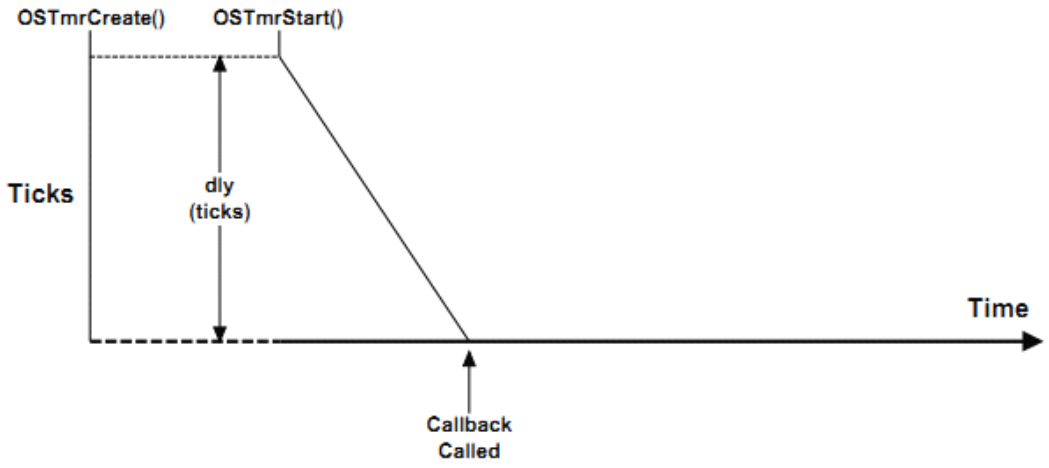


Figure 12-1 **One Shot Timers (dly > 0, period == 0)**

如图 12-2 所示，在定时值为 0 之前调用 OSTmrStart()后一次性定时器被再次触发。这个特性可以被用来模拟看门狗的功能。

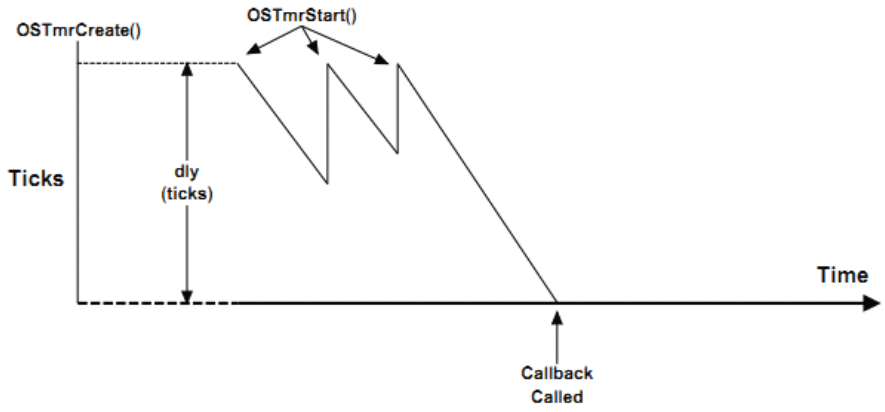


Figure 12-2 **Retriggering a One Shot Timer**

12-2 无初始定时周期模式

如图 12-3 所示，定时器被设置为无初始定时周期模式。当定时器期满时，回调函数被调用，定时值被定时周期值重载，如此周期性地重复。

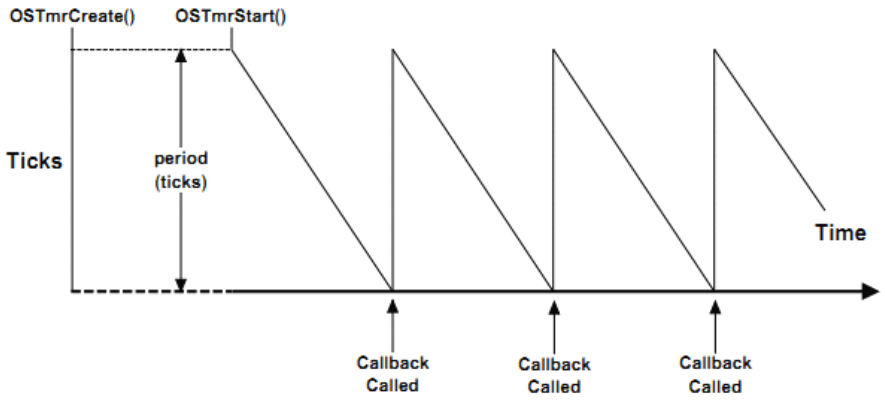


Figure 12-3 **Periodic Timers (dly == 0, period > 0)**

12-3 有初始定时周期模式

如图 12-4 所示，定时器可以被设置为有初始定延周期模式。第一周期的递减值由 OSTmrCreate()中的参数"dly"设置，以后的重载值由"period"值确定。调用 OSTmrStart()重新开始。

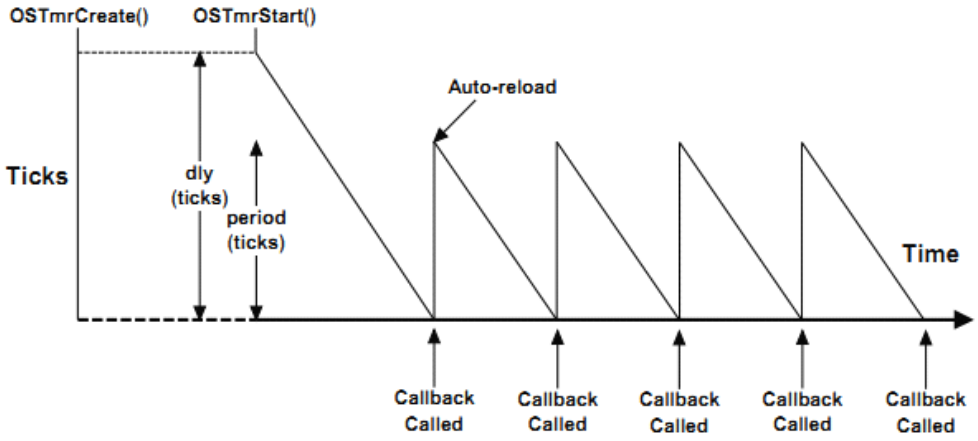


Figure 12-4 **Periodic Timers (dly > 0, period > 0)**

12-4 内部定时器管理

12-4-1 内部定时器管理-定时器状态

图 12-5 显示了定时器的状态图

任务调用 OSTmrStateGet()获得定时器的状态。当然，也可以调用 OSTmrRemainGet()获得剩余定时时间。定时值是以时基为单位的。如果时基定时器的频率率为 10Hz，那么定时值设置为 50 意味着延时 5 秒。如果定时器被停止，那其定时值也将被停止，直到定时器被恢复时，定时器值继续被定时器任务递减。

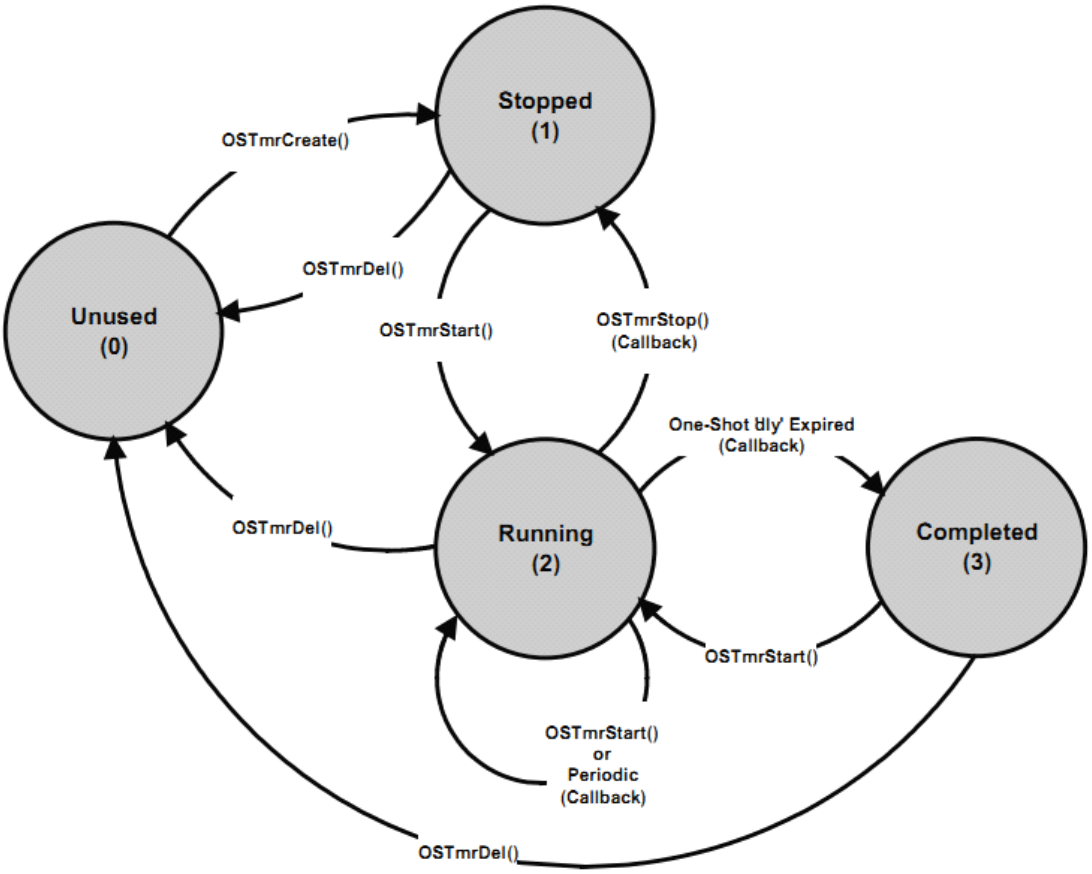


Figure 12-5 **Timer State Diagram**

F12-5（1）"Unused"状态意味着定时器尚未被创建或已经被删除。换句话说，uC/OS-III 不知道该定时器的相关状态。

F12-5（2）当创建了定时器或调用了 OSTmrStop()，定时器会处于停止模式。

F12-5（3）当调用 OSTmrStart()后定时器处于运行状态。

F12-5（4）一次性定时模式的定时器延时期满后处于完成状态"Completed"。

12-4-2 定时器内部管理——OS_TMR

定时器是 uC/OS-III 中的内核对象，其数据类型为 OS_TMR（见 OS.H）。如列表 12-1 所示

uC/OS-III 中管理定时器的相关代码在文件 OS_TMR.C 中。在编译时通过设置 OS_CFG.H 中的 OS_CFG_TMR_EN 为 1 开启定时器功能。

```
typedef  struct  os_tmr  OS_TMR;                (1)


struct  os_tmr {
    OS_OBJ_TYPE          Type;                  (2)
    CPU_CHAR            *NamePtr;               (3)
    OS_TMR_CALLBACK_PTR  CallbackPtr;           (4)
    void                *CallbackPtrArg;        (5)
    OS_TMR              *NextPtr;               (6)
    OS_TMR              *PrevPtr;
    OS_TICK              Match;                 (7)
    OS_TICK              Remain;                (8)
    OS_TICK              Dly;                   (9)
    OS_TICK              Period;               (10)
    OS_OPT               Opt;                  (11)
    OS_STATE             State;                (12)
};
```

Listing 12-1 **OS_TMR data type**

L12-1（1）在 uC/OS-III 中，所有的结构体都分配一个数据类型。事实上，所有的数据类型都是以"OS_"为开头的并且全部大写。当定时器被创建时，用数据类型为 OS_TMR 的变量描述这个定时器。

L12-1（2）结构体开始于一个"Type"域，uC/OS-III 可以通过这个域辨认它是个定时器（其它内核对象的结构体首部也有"Type"）。如果函数需传递一种内核对象，uC/OS-III 会检测"Type"域是否为参数所需的类型。例如，如果传递消息队列 QS_Q 到定时器，uC/OS-III 会检测是否为消息队列被提交，并返回错误代号。

L12-1（3）每个内核对象都可以被命名，以利于调式器或 uC/Probe 的调试。这是一个指向内核对象名的指针。

L12-1（4）.CallbackPtr 是一个指向函数的指针，被指向的函数称作回调函数，当定时器期满时回调函数被调用。如果定时器创建时该指针值为 NULL，回调函数将不会被调用。

L12-1(5)当回调函数需要接受一个参数时(.CallbackPtr 不为 NULL),这个参数通过该指针传递给回调函数。

L12-1（6）.NextPtr 和.PrevPtr 都是指针，用于将定时器链接成一个双向链表。

L12-1（7） 当定时器管理器中的变量 OSTmrTickCtr 的值等于.Match 值时，定时器期满。

L12-1（8）.Remain 中保存了距定时器期满还有多少个时基。这个值每经过一个 OS_CFG_TMR_WHEEL_SIZE（稍后说明）被更新。

L12-1(9).Dly 域包含了定时器的定时值。这个值以时基为最小单位。

L12-1（10）.Period 是定时器的定时周期（当被设置为周期模式时)。这个值以时基为最小单位。

L12-1（11）.Opt 域包含了传递给 OSTmrCreate()的参数（可选)。

L12-1（12）.State 中包含了定时器当前的状态（见图 12-5)。

即使了解 OS_TMR 结构体的内容，用户代码也不能直接访问这些内容。必须通过 uC/OS-III 提供的 API 访问。

12-4-3 内部定时器管理——定时器任务

通过设置 OS_CFG.H 中的 OS_CFG_TMR_EN 为 1 使能定时器任务 OS_TmrTask()，该任务的优先级通过 OS_CFG_APP.H 中的 OS_CFG_TMR_TASK_PRIO 设置。OS_TmrTask()的优先级通常被设置为中等大小。

OS_TmrTask()是一个周期性的任务，它使用时基中断源作为它的时钟计数源。然而，定时器通常产生较低的周期信号（可以为 10Hz 等）。它的周期信号是从时基信号中分频得来的。如果时基频率为 1000Hz，定时器想要的频率为 10Hz，那么定时器任务需被设置为每 100 个时基产生一次信号，也就是分频值为 100。如图 12-6

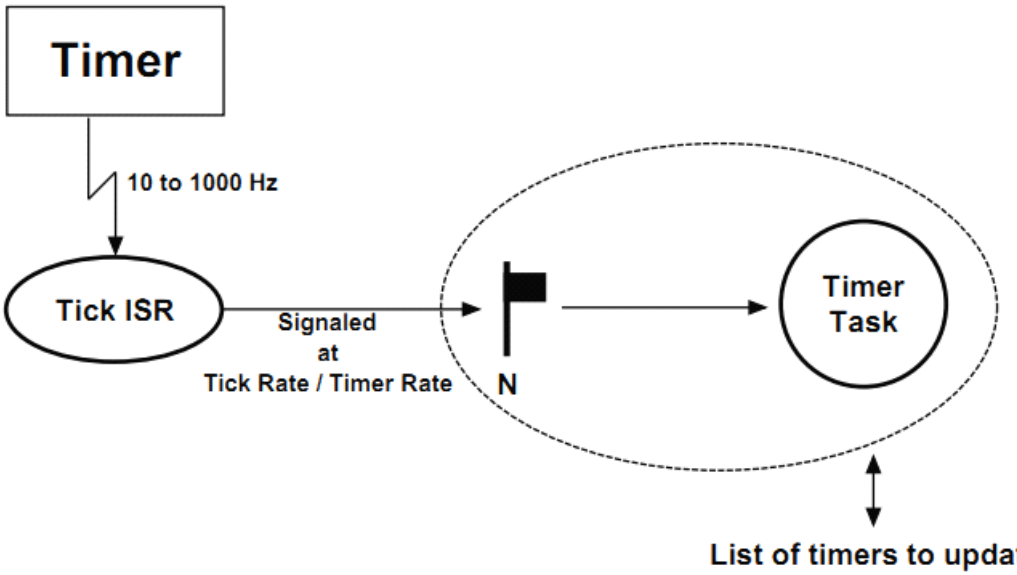


Figure 12-6 Tick ISR and Timer Task relationship

图 12-7 定时器管理任务的相关运行情况

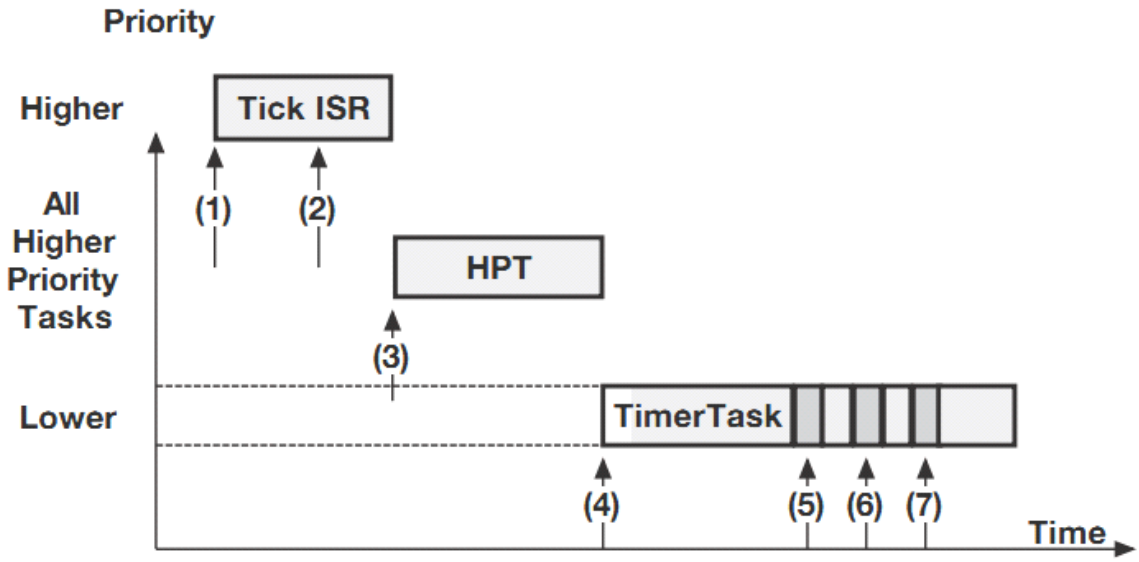


Figure 12-7 **Timing Diagram**

F12-7（1）中断产生

F12-7（2）时基 ISR 标记时基任务。

F12-7（3）然而，有高优先级任务需要被执行。因此，uC/OS-III 切换到高优先级任务。

F12-7（4）当所有的高优先级任务被执行完毕，uC/OS-III 切换到定时器任务，假设有三个定时器期满。

F12-7（5）第一个定时器的回调函数被执行。

F12-7（6）第二个定时器的回调函数被执行。

F12-7（7) 第三个定时器的回调函数被执行。

有一些有趣的事情需注意：

1）回调函数是在定时器任务被切换后执行的。这意味着定时 器 任务需要有足够的堆栈空间供回调函数去执行。

2）回调函数是在根据定时器队列中依次存放的。所以期满后回调函数是依次被执行的。

3）定时器任务的执行时间决定于：有多少个定时器期满，执行定时器中的回调函数需多少时间。因为回调函数是用户提供，它可能很大程度上影响了定时器任务的执行时间。

4）定时器中的回调函数不能等待事件的发生，因为这样可能会让定时器任务被挂起。

5）回调函数被执行时会锁调度器，所以你必须让回调函数尽可能地短。

12-4-4 内部定时器管理——定时器列表

有些情况下，uC/OS-III 可能要维护上百个定时器。使用定时器列表会大大降低更新定时器列表所占用的 CPU 时间。定时器列表类似于时基列表，如图 12-8 所示。

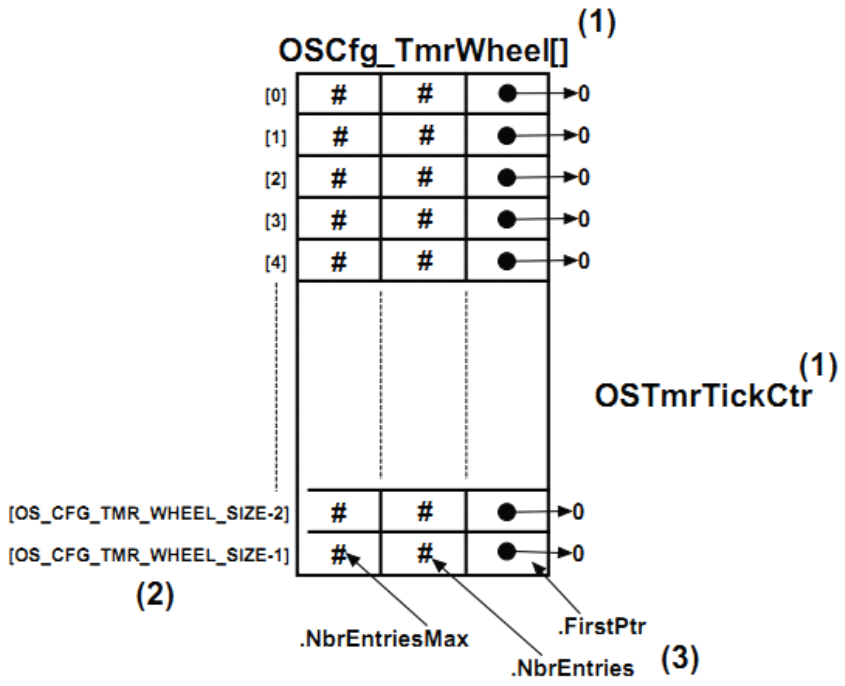


Figure 12-8 **Empty Timer List**

F12-8（1）定时器列表包含两部分：表(OSCfg_TmrWheel[])和定时器计数变量(OSTmrTickCtr)。

F12-8（2）表中包含了 OS_CFG_TMR_WHEEL_SIZE 条记录，这是在编译时设定的（见 OS_CFG_APP.H）。记录的多少仅限于处理器的 RAM 空间。推荐的设置值为定时器数/4。不推荐设置 OS_CFG_TMR_WHEEL_SIZE 为偶数值。换句话说，如果定时器任务是 10Hz 的，避免设置 OS_CFG_TMR_WHEEL_SIZE 为 10，而是用 11 代替。

F12-8（3）表中的每个记录都由3部分组成：.NbrEntriesMax，.NbrEntries，.FirstPtr。.NbrEntriesMax 表明该记录中有多少个定时器。.NbrEntriesMax 表明该记录中最大时存放了多少个定时器。.FirstPtr 指向当前记录的定时器链表。

当时基ISR标记这个定时器任务时，定时器任务调用OS_TmrTask()递增定时器计数值。

通过调用 OSTmrStart()将定时器插入到定时器列表中。然而，定时器必须在被使用之前被创建。

用一个例子阐述一个定时器被插入到定时器列表中的过程。如列表 12-2 所示。我们先假定定时器列表是空的，设置 OS_CFG_TMR_WHEEL_SIZE 为 9，当前的 OSTmrTickCtr 为 12。调用 OSTmrStart()时定时器被放入定时器列表。假定定时器创建时被设置延时为 1，且这个任务是一次性定时。

```
OS_TMR  MyTmr1;
OS_TMR  MyTmr2;


void MyTask (void *p_arg)
{
    OS_ERR  err;


    while (DEF_ON) {
        :
        OSTmrCreate((OS_TMR            *)&MyTmr1,
                    (OS_CHAR           *)"My Timer #1",
                    (OS_TICK            )1,
                    (OS_TICK            )0,
                    (OS_OPT             )OS_OPT_TMR_ONE_SHOT,
                    (OS_TMR_CALLBACK_PTR)0;
                    (OS_ERR            *)&err;
        /* Check 'err" */
        OSTmrStart ((OS_TMR *)&MyTmr1,
                    (OS_ERR *)&err);
        /* Check "err" */
        // Continues in the next code listing!
```

Listing 12-2 **Creating and Starting a timer**

因为 OSTmrTickCtr 的值为 12，定时器的定时值为 1。当 OSTmrTickCtr 的值变为 13 时定时器期满。定时器会被放入 OSCfg_TmrWheel[]，经过下式运算：

MatchValue =OSTmrTickCtr + dly

Index intoOSCfg_TmrWheel[] = MatchValue% OS_CFG_TMR_WHEEL_SIZE

"dly"是传递给 OSTmrCreate()的第三个参数。将上述公式代入本例，我们会得到以下等式：

MatchValue =13

Index into OSCfg_TmrWheel[] =4

定时器被插入到OScfg_TmrWheel[4] 中，在这种情况下，OS_TMR被放在队列中的首位置(OSCfg_TmrWheel[4].FirstPtr 指向这个OS_TMR)，并且索引 4 的计数值递增（OSCfg_TmrWheel[4].NbrEntries 此时为 1）。匹配值"MatchValue"被放在 OS_TMR 的.Match 中。因为这是索引 4 的唯一一个定时器，.NextPtr 和.PrevPtr 都指向 NULL。

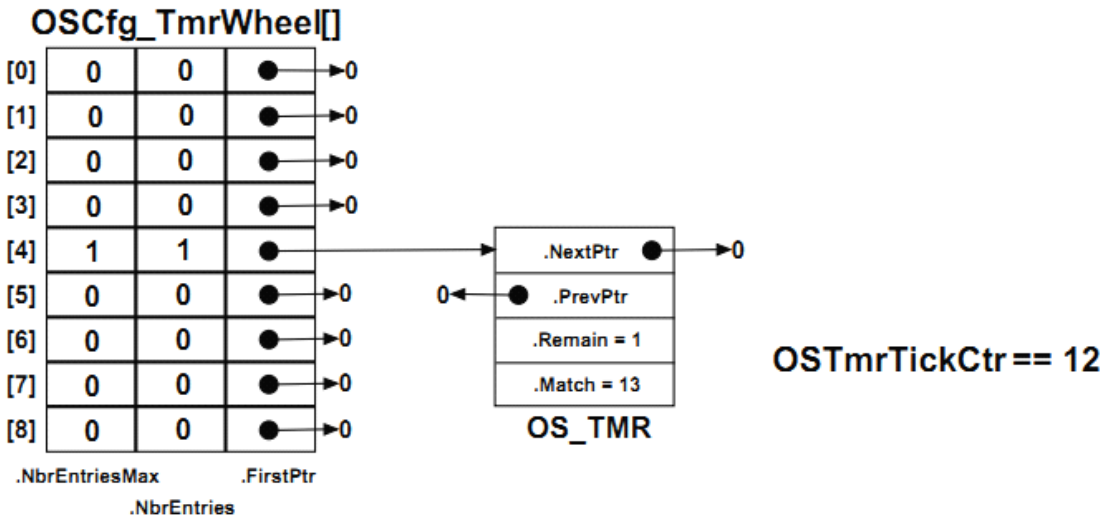


Figure 12-9 **Inserting a timer in the timer list**

以下的代码显示创建和开启定时器的操作。

```
        // Continuation of code from previous code listing.
        :
        :
        OSTmrCreate((OS_TMR            *)&MyTmr2,
                    (OS_CHAR           *)"My Timer #2",
                    (OS_TICK            )10,
                    (OS_TICK            )0,
                    (OS_OPT             )OS_OPT_TMR_ONE_SHOT,
                    (OS_TMR_CALLBACK_PTR)0;
                    (OS_ERR            *)&err;
        /* Check 'err" */
        OSTmrStart ((OS_TMR *)&MyTmr,
                    (OS_ERR *)&err);
        /* Check 'err" */
    }
}
```

Listing 12-3 **Creating and Starting a timer - continued**

uC/OS-III 会计算匹配值和索引值如下：

MatchValue =12+10 =22

Index into OSCfg_TmrWheel[] =22%9 =4

第二个定时器也被插入到索引为 4 的表记录中，如图 12-10。队列被重新分配，剩余时间最少的定时器被放到队列的首部，剩余时间最多的定时器被放到队列的尾部。

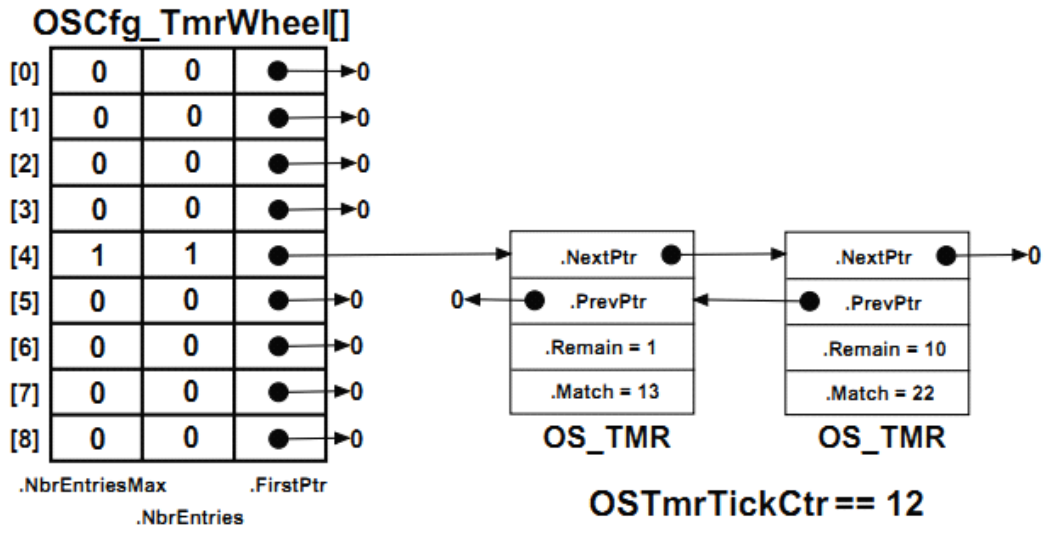


Figure 12-10 **Inserting a second timer in the tick list**

当定时器任务被执行（见 OS_TMR.C 中的 OS_TmrTask()），它首先递增 OSTmrTickCtr，然后决定表中的哪条记录需被更新。然后，如果这条记录中有定时器队列（.FirstPtr 不为 NULL），每个 OS_TMR 都会被检测其.Match 是否与 OSTmrTickCtr 相等。如果相等，这个定时器会被移出该队列，然后 OS_TmrTask()调用这个定时器的回调函数（假定这个定时器被创建时有回调函数）。遍历该记录的整个队列直到没有定时器的.Match 值与 OSTmrTickCtr 匹配。注意的是：队列会被重新排序。

OS_TmrTask()任务的大部分工作都是在锁调度器的状态下进行的。然而，因为队列会被重新分配（依次排序），所以遍历这个队列的时间会非常短的，也就是临界段会非常短的。

12-5 总结

当定时器的计数值递减为 0 时定时器期满，并执行回调函数。回调函数由用户定义。

uC/OS-III 允许用户建立任意数量的定时器（只限制于处理器的 RAM 大小）。

回调函数在定时器任务中被调用，且运行回调函数时调度器处于被锁状态。回调函数越短越好，且不能在回调函数中等待事件发生。否则定时器任务会被挂起，导致定时器任务崩溃。

13、资源管理

这个章节会介绍共享资源相关的服务。共享资源可以是：变量（静态的或全局的）、结构体、内存空间、I/O 等。

我们推荐用 mutex 保护共享资源。在这个章节中，其它的方法也会被介绍。

多个任务可能会同时要求占用资源：内存空间、全局变量、指针、缓冲区、列表、环形缓冲区等。通过共享资源，任务间通信将会比较简单。当然，避免任务间的竞争和防止资源被破坏是非常重要的。

例如，如果用实现手表的计时功能，那么就需有小时、分钟、秒。可定义为 TimeOfDay()任务如列表 13-1 所示

想象一下，如果这个任务的分钟递增到 60 被设置为 0，且将要递增小时的时候，中断发生了，高优先级任务抢占 CPU。这时高优先级任务读取当前时间，会是什么样的情况？因为，在中断前小时没有被递增，所以高优先级任务读到的值是错误的，它会慢了整整一个小时。

让 TimeOfDay()任务中更新时间部分的代码具有不可分割性。这些变量可以被认为是共享资源，且代码要访问这些变量时必须以临界段的形式看待这些变量。uC/OS-III 提供 mutex 服务保护这些共享资源，并创建临界段。

```
CPU_INT08U  Hours;
CPU_INT08U  Minutes;
CPU_INT08U  Seconds;


void  TimeOfDay (void *p_arg)
{
    OS_ERR  err;


    (void)&p_arg;
    while (DEF_ON) {
        OSTimeDlyHMSM(0,
                      0,
                      1,
                      0,
                      OS_OPT_TIME_HMSM_STRICT,
                      &err);
        /* Examine "err" to make sure the call was successful */
        Seconds++;
        if (Seconds > 59) {
            Seconds = 0;
            Minutes++;
            if (Minutes > 59) {
                Minutes = 0;
                Hours++;
                if (Hours > 23) {
                    Hours = 0;
                }
            }
        }
    }
}
```

Listing 13-1 **Faulty Time-Of-Day clock task**

大部分独占资源的方法都是创建临界段：

1) 关中断方式

2) 锁调度器方式

3) 信号量方式

4) mutex 方式

采用哪种保护机制决定于访问共享资源的代码长度，如表 13-1 所示

资源共享方式	什么时候该用
关中断方式	能很快地结束访问共享资源，不推荐使用这种方法，因为会导致中断延迟
锁调度器方式	访问共享资源比较久时
信号量方式	当该共享资源经常被多个任务使用时采用。但信号量可能会造出优先级倒置。然而，信号量方式的执行时间少于 mutex 方式
mutex 方式	推荐使用这种方法访问共享资源，尤其当任务要访问的共享资源有截止时间。 uC/OS-III 的 mutex 有内置的优先级，这样可防止优先级倒置。 然而，mutex 方式慢于信号量方式，多了个操作：需改变信号量的优先级

表 13-1　资源共享

13-1 关中断

独占共享资源的最快和最简单方法是关中断，如列表 13-2 所示

```
Disable Interrupts;
Access the resource;
Enable  Interrupts;
```

Listing 13-2 **Disabling and Enabling Interrupts**

然而，关/开中断是和 CPU 相关的操作，其相关代码被放在与 CPU 相关的文件中（见 CPU.H）。uC/OS-III 中与 CPU 相关的模块叫做 uC/CPU。每种架构的 CPU 都需要设置相适应的 uC/CPU 文件。

```
void OS_Function (void)
{
    CPU_SR_ALLOC();                 (1)


    CPU_CRITICAL_ENTER();           (2)
    Access the resource;            (3)
    CPU_CRITICAL_EXIT();            (4)
}
```

Listing 13-3 **Using CPU macros to disable and enable interrupts**

L13-3（1）当使用开/关中断宏时 CPU_SR_ALLOC()宏是必需的。这个宏为一个本地的变量分配了存储空间用于保存关中断前的 CPU 状态寄存器 SR。{临界段时关中断前只需保存 SR}

L13-3（2）CPU_CRITICAL_ENTER()关全局中断。

L13-3（3）访问临界段时不会被中断或任务打断，因为中断被关闭。换句话说，临界段的操作是原子性的。

L13-3（4）CPU_CRITICAL_EXIT()从这个本地变量中恢复关中断前的 CPU 状态，并开全局中断。

CPU_CRITICAL_ENTER()和 CPU_CRITICAL_EXIT()经常成对地使用。关中断时间越短越好，不然会影响系统响应外部事件的及时性。当临界段很短时可以使用关中断方法。

注意的是：只有这种方法才可以让任务和 ISR 共享资源。

13-2 锁调度器

如果任务不需要和 ISR 共享资源，就可以通过锁调度器来访问共享资源。如列表 13-4 所示。

```
void OS_Function (void)
{
    CPU_SR_ALLOC();                (1)
    CPU_CRITICAL_ENTER();          (2)
    Access the resource;           (3)
    CPU_CRITICAL_EXIT();           (4)
}
```

Listing 13-4 **Accessing a resource with the scheduler locked**

使用这种方法，多个任务就可以无竞争的访问共享资源。注意，只有调度器被锁，中断是使能的，如果在处理临界段时中断发生，ISR 程序就会被执行。在 ISR 的末尾，uC/OS-III 会返回原任务（即使 ISR 中有高优先级任务被就绪）。

OSSchedLock()和 OSSchedUnlock()可以被嵌套多达 250 级。当 OSSchedUnlock()与 OSSchedLock()被调用的次数相同时调度器才被开启。

直到调度器开启，uC/OS-III 切换到高优先级任务。

uC/OS-III 不允许用户在锁调度器时堵塞呼叫{不能在锁调度器时等待某事件的发生}。不然程序会崩溃。

虽然这个方法用起来挺好，然而锁调度器影响了抢占式内核的初衷。

13-3 信号量

将信号量用于同步的概念是荷兰的电脑科学家 Edgser Dijkstra 在 1959 年发明的。在电脑软件中，信号量是一种用于多任务调度的协议机制。最初用于控制共享资源的访问，现在用于同步（详见第 14 章“同步”）。然而，介绍信号量是如何被用于控制共享资源是非常必要的。

信号量是一个“锁定机构”，代码需要获得钥匙才可以访问共享资源。占用该资源的任务不再使用该资源并释放资源时，其它任务才能够访问这个资源。

通常有两种类型的信号量：二值信号量和多值信号量。二值信号量的值只能是 0 或 1.多值信号量计数值可以是 0 到 4294967295（依赖于计数值是 8 位，16 位或 32 位）。特别的，uC/OS-III 中的信号量计数值最大为 OS_SEM_CTR(见 OS_TYPE.H)。根据信号量计数值，uC/OS-III 可以知道有该信号量可以再被多少个任务获得。

只有任务才允许使用信号量，ISR 是不允许的。

信号量是内核对象，通过数据类型 OS_SEM 定义（见 OS.H）。应用中可以有任意多个信号量（只限于处理器的 RAM）。

与信号量操作相关的函数如表 13-2 所示。在这个章节中，只介绍三个经常使用的函数：OSSemCreate()，OSSemPend()，OSSemPost()。其它的函数在附录 A 中介绍。当信号量被用于共享资源时，信号量相关函数只能被任务调用（绝不能被 ISR 调用）。但将信号量用于标记任务时可以被 ISR 调用。

函数名	功能
OSSemCreate()	创建一个信号量
OSSemDel()	删除一个信号量
OSSemPend()	等待某个信号量
OSSemPendAbort()	取消等待某个信号量
OSSemPost()	释放或标记信号量
OSSemSet()	设置信号量计数值

表 13-2　信号量相关的 API 函数总结

13-3-1 二值信号量

任务要访问共享资源就必须执行一个等待操作。如果信号量是有效的（信号量计数值大于 0），信号量计数值递减，任务访问该共享资源。如果信号量计数值为 0，任务会被放入挂起队列中等待该信号量有效。uC/OS-III 允许设置等待的时限。如果等待超时，该挂起任务会被就绪，等待该信号量的"pend"函数会返回一个错误代号。

任务通过"post"函数释放信号量。如果没有任务在等待这个信号量，信号量计数值会被递增。如果有任务在等待这个信号量，其中高优先级的挂起任务被就绪，但信号量计数值不会递增。

如列表 13-5 所示

```
OS_SEM  MySem;                          (1)


void  main (void)
{
    OS_ERR  err;
    :
    :
    OSInit(&err);
    :
    OSSemCreate(&MySem,                 (2)
                "My Semaphore",         (3)
                1,                      (4)
                &err);                  (5)
    /* Check "err" */
    :
    /* Create task(s) */
    :
    OSStart(&err);
    (void)err;
}
```

Listing 13-5 **Using a semaphore to access a shared resource**

L13-5（1）定义一个信号量，信号量的数据类型必须是 OS_SEM。

L13-5（2）通过调用 OSSemCreate()创建一个信号量，将信号量地址传递给函数的第一个参数。信号量必须在创建后才能被其他任务使用。在这里，信号量是在 main()函数中被创建的，也可以在任务中被创建。

L13-5（3）为信号量分配一个名字，调试时就很容易识别。名字可以存储于 ROM，因为 ROM 的空间远大于 RAM。如果需要在程序运行时改变信号量的名字，将名字以数组的形式存在 RAM 中，并传递数组地址给 OSSemCreate()。当然，该数组必须以空字符截止。

L13-5（4）设置信号量计数值。当共享资源只能有被一个任务占有时设置信号量计数值为 1.

L13-5（5）根据信号量的创建结果 OSSemCreate()返回一个错误代号。如果所有的参数都是有效的，错误代号为 OS_ERR_NONE。

```
void  Task1 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;


    while (DEF_ON) {
        :
        OSSemPend(&MySem,                              (6)
                  0,                                   (7)
                  OS_OPT_PEND_BLOCKING,                (8)
                  &ts,                                 (9)
                  &err);                              (10)
        switch (err) {
            case OS_ERR_NONE:
                 Access Shared Resource;              (11)
                 OSSemPost(&MySem,                    (12)
                           OS_OPT_POST_1,             (13)
                           &err);                     (14)
                 /* Check "err" */
                 break;

            case OS_ERR_PEND_ABORT:
                 /* The pend was aborted by another task     */
                 break;

            case OS_ERR_OBJ_DEL:
                 /* The semaphore was deleted                */
                 break;

            default:
                 /* Other errors                             */
        }
        :
    }
}
```

Listing 13-6 **Using a semaphore to access a shared resource**

L13-6（6）通过调用 OSSemPend()函数等待一个信号量。任务必须指定所等待的信号量，且这个信号量之前已经被创建。

L13-6（7）这个参数是等待信号量的期限值，以时基为最小单位。事实上，等待期限决定于时基频率。当时基频率为 1000Hz 时，10 个时基的期限等待时间是 10 毫秒。设置期限值为 0 意味着任务将一直等待者信号量。

L13-6（8）第三个参数设置了挂起的方式。有两种方式：OS_OPT_PEND_BLOCKING 和 OS_OPT_PEND_NON_BLOCKING。第一种方式：有等待挂起。在信号量无效时，调用 OSSemPend()后任务会被挂起直到接收到信号量或期满。第二种方式：无等待挂起。在信号量无效时，调用 OSSemPend()后回迅速返回（任务不被挂起）。在使用信号量保护共享资源时第二种方式很少被用到。

L13-6（9）信号量被提交时，uC/OS-III 读取当前时间戳，并当 OSSemPend()返回时返回这个时间戳。这个功能可以让用户知道信号量是什么时候被提交的。任务接收到信号量后，调用 OS_TS_GET()读取当前的时间戳并计算差值，就知道等待时间了
{信号量被提交后到任务接收到信号量所等待的时间}。

L13-6（10）根据信号量的创建结果 OSSemPend()返回一个错误代号。如果信号量创建成功，错误代号为 OS_ERR_NONE。如果创建失败，错误代号会包含错误的原因。检测错误代号值是很有必要的，因为其它任务可能删除了这个信号量或取消这个任务的挂起状态。然而，在程序运行时不推荐删除内核对象，因为这可能会导致严重的后果。

L13-6（11）当 OSSemPend()正确返回时，任务就可以访问这个共享资源。

L13-6（12）资源访问结束后，任务调用 OSSemPost()释放这个信号量。

L13-6（13）OS_OPT_POST_1 意味着这个信号量只能被单个任务占用。

L13-7（14）如大多数 uC/OS-III 函数一样，该地址对应的变量中存放函数返回的错误代号。

```
void  Task2 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;


    while (DEF_ON) {
        :
        OSSemPend(&MySem,                                (15)
                  0,
                  OS_OPT_PEND_BLOCKING,
                  &ts,
                  &err);
        switch (err) {
            case OS_ERR_NONE:
                 Access Shared Resource;
                 OSSemPost(&MySem,
                           OS_OPT_POST_1,
                           &err);
                 /* Check "err" */
                 break;

            case OS_ERR_PEND_ABORT:
                 /* The pend was aborted by another task     */
                 break;

            case OS_ERR_OBJ_DEL:
                 /* The semaphore was deleted                */
                 break;

            default:
                 /* Other errors                             */
        }
        :
    }
}
```

Listing 13-7 **Using a semaphore to access a shared resource**

L13-7（15）其它要访问共享资源的任务也要经过同样的步骤。

尤其在共享 I/O 时，信号量非常有用。想象一个如果两个任务共同发字符给打印机。打印机交错地打印这两个任务的字符。例如，任务一想要打印"I am Taks 1"，任务二想要打印"I am Task 2"，那么结果就可能为"I Ia amm TTasask k1 2"。

在这种情况下，可以用信号量并将信号量计数值设为 1（二值信号量）。规则是简单的：在访问打印机之前任务必须先获得该信号量。如图 13-1 显示了两个任务竞争，都想独占问打印机，设置一个锁，在访问打印机之前必须会得钥匙，信号量就是这个钥匙。

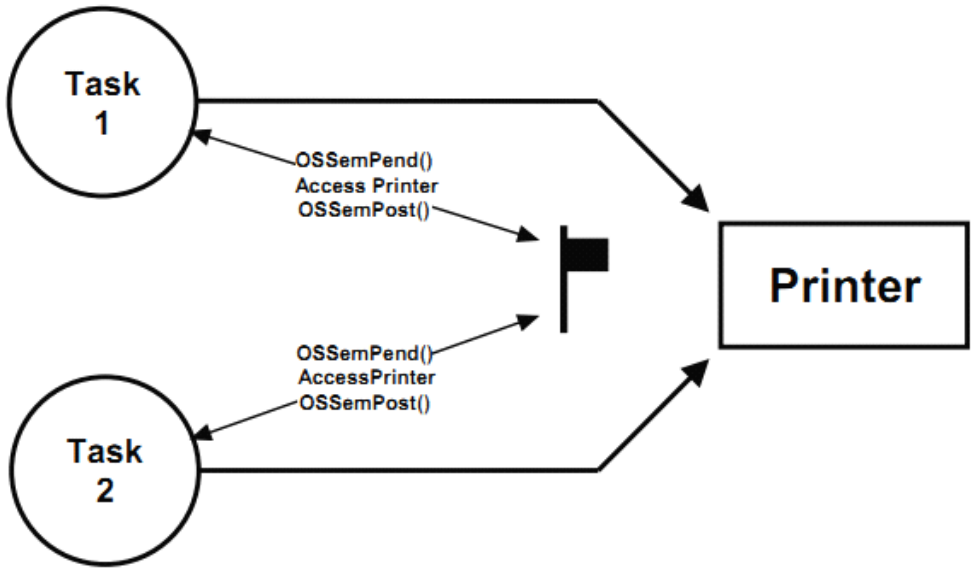


Figure 13-1 **Using a semaphore to access a printer**

上述例子表明任务必须获得对应的信号量才能访问资源。这是一种能保护临界段很好的方法。任务获得共享资源前必须先获得信号量。例如，一个 RS-232C 端口被多个任务共享，发送命令到目标器件并接收响应。如图 13-2 所示

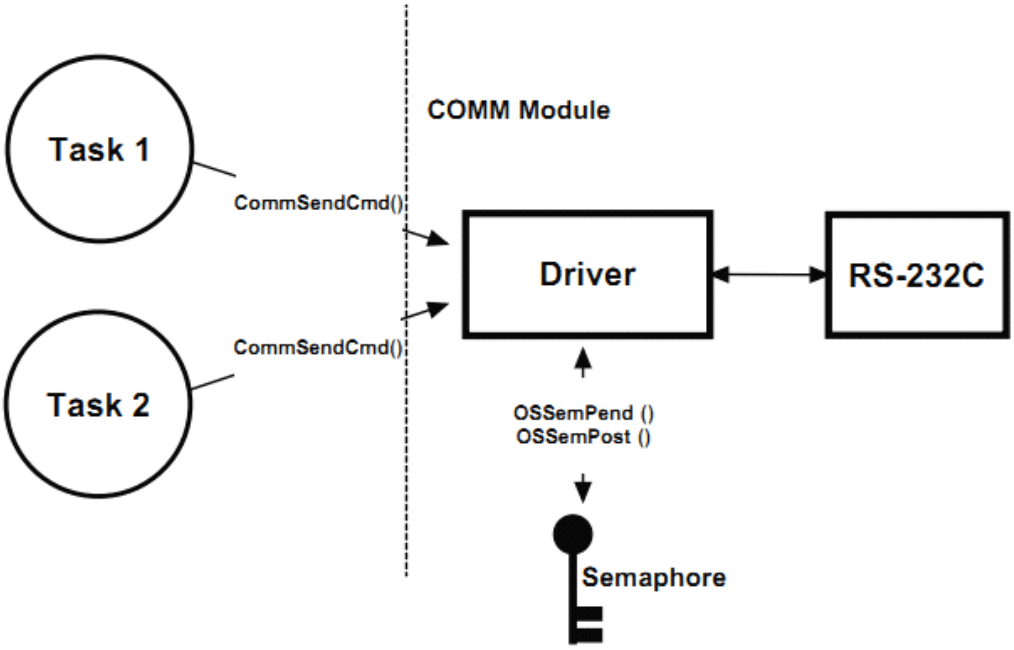


Figure 13-2 **Hiding a semaphore from a task**

函数 CommSendCmd()被调用，它有三个参数：命令字符串、指向响应字符串的指针、时间期限（在规定时间内可能没有响应）。这个函数的伪代码如列表 13-8

```
APP_ERR  CommSendCmd (CPU_CHAR  *cmd,
                      CPU_CHAR  *response,
                      OS_TICK    timeout)
{
    Acquire serial port's semaphore;
    Send "cmd" to device;
    Wait for response with "timeout";
    if (timed out) {
        Release serial port's semaphore;
        return (error code);
    } else {
        Release serial port's semaphore;
        return (no error);
    }
}
```

Listing 13-8 **Encapsulating the use of a semaphore**

每个任务都必须调用这个函数才能发送命令给 RS-232C 端口。假定这个信号量计数值已经被设置为 1（通过该器件的初始化函数）。

第一个任务调用 CommSendCmd()获得信号量，并发送命令，等待响应。此时端口处于忙状态，这时第二个任务想要发送命令，那么第二个任务就被挂起，直到信号量被释放。第一个任务释放信号量后，第二个任务获得信号量并被允许访问 RS-232C 端口。

13-3-2 信号量计数值

当共享资源同时可以被多个任务访问时，信号量计数值用于标记共享资源能同时被多少个任务访问。例如，缓冲池可以被多个任务同时访问，如图 13-3，假定缓冲池中定义了 10 个缓冲区。任务通过调用 BufReq()获得一个缓冲区，任务通过调用 BufRel()释放一个缓冲区。伪代码如列表 13-9 所示。

缓冲池允许 10 个任务各自占用一个缓冲区，所以信号量计数值被初始化为 10。当所有的缓冲区被分配完，任务再申请缓冲区时会被挂起直到信号量被释放。当任务完成对缓冲区的使用后调用 BufRel()释放缓冲区给缓冲区管理器，若有任务等待缓冲区，则直接将此缓冲区分配给任务。若没有任务等待缓冲区，这个缓冲区被插入到缓冲区队列，并将信号量计数值递增。通过缓冲区管理器的接口 BufReq()和 BufRel()，调用者不必了解具体的实现细节。

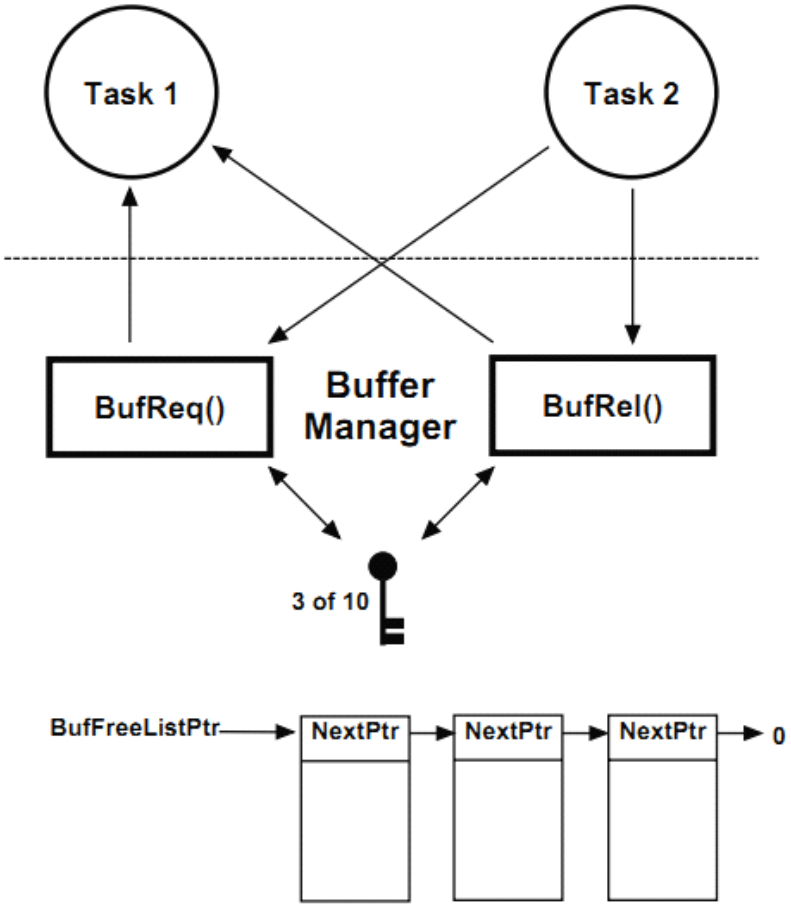


Figure 13-3 **Using a counting semaphore**

```
BUF  *BufReq (void)
{
    BUF  *ptr;


    Wait on semaphore;
    ptr = OSMemGet(...) ;                    /* Get a buffer      */
    return (ptr);
}


void  BufRel (BUF *ptr)
{
    OSMemPut(..., (void *)ptr, ...);         /* Return the buffer */
    Signal semaphore;
}
```

Listing 13-9 **Buffer management using a semaphore.**

13-3-3 信号量需注意的事项

用信号量访问共享资源不会导致中断延迟。当任务在执行信号量所保护的共享资源时，ISR 或高优先级任务可以抢占该任务。

应用中可以有任意个信号量用于保护共享资源。然而，推荐将信号量用于 I/O 端口的保护，而不是内存地址。

信号量经常被过度使用。很多情况下，访问一个简短的共享资源时不推荐使用信号量，请求和释放信号量会消耗 CPU 时间。通过关/开中断能更有效地执行这些操作。为了说明，假设两个任务共享一个 32 位的整数变量。第一个任务将这个整数变量加 1，第二个任务将这个变量清零。考虑到执行这些操作用时很短，不需要使用信号量。执行这个操作前任务只需关中断，执行完毕后再开中断。若操作浮点数变量且处理器不支持硬件浮点操作时，就需要用到信号量。因为在这种情况下处理浮点数变量需较长时间。

信号量会导致一种严重的问题：优先级反转。

13-3-4 信号量的结构

正如前面提到的，信号量是内核对象，通过数据类型 OS_SEM 定义，OS_SEM 源于结构体 os_sem(见 OS.H)。如表 13-10 所示。

uC/OS-III 中与信号量相关的代码都被放在 OS_SEM.C 中。通过设置 OS_CFG.H 中的 OS_CFG_SEM_EN 为 1 使能信号量。

```
typedef  struct  os_sem  OS_SEM;              (1)


struct  os_sem {
    OS_OBJ_TYPE          Type;                (2)
    CPU_CHAR            *NamePtr;             (3)
    OS_PEND_LIST         PendList;            (4)
    OS_SEM_CTR           Ctr;                 (5)
    CPU_TS               TS;                  (6)
};
```

Listing 13-10 **OS_SEM data type**

L13-10（1）在 uC/OS-III 中，所有的结构体都会特定的数据类型，以“OS_”开头并且全部大写。

L13-10（2）这个结构体的第一个变量是“Type”域，表明 uC/OS-III 识别所定义的是一个信号量。其它的内核对象也有“Type”域作为结构体的第一个变量。如果函数要调用一个内核对象，uC/OS-III 会检测所调用的内核对象的数据类型是否对应。例如，如果需要传递一个消息队列 OS_Q 给函数，但实际传递的是一个信号量 OS_SEM，uC/OS-III 就会检测出这是一个无效的参数，并返回错误代号。

L13-10（3）每个内核对象都可以被赋予一个名字，名字有 ASCII 字符串组成，但必须以空字符结尾。

L13-10（4）若有多个任务等待信号量，信号量就会将这些任务放入其挂起队列中。（见章节 10“挂起队列”）

L13-10（5）信号量中包含一个信号量计数变值。信号量计数值可以定义为 8 位，16 位，或 32 位，取决于 OS_TYPE.H 中的 OS_SEM_CTR 是如何被定义的。

uC/OS-III 中二值信号量和多值信号量的定义是一样的。只有信号量被创建后才有区别。如果创建信号量时将信号量计数值被初始化为 1，那么它就是二值信号量。如果创建信号量时将信号量计数值初始化大于 1，那么它就是多值信号量。

L13-10（6）信号量中包含了一个时间戳变量，存储了上一次信号量被提交时的时间戳。当信号量被提交时，CPU 的时间戳被读取并存在信号量的时间戳变量中，当 OSSemPend()被调用时就能读取这个时间戳变量。

即使用户了解 OS_SEM 的组成，用户代码也不能直接访问信号量中的变量。必须通过 uC/OS-III 提供的 API。

信号量在使用之前必须被创建。

在访问共享资源之前任务必须通过 OSSemPend()获得这个信号量。如列表 13-11 所示

```
OS_SEM  MySem;


void MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    :
    while (DEF_ON) {
        :
        OSSemPend(&MySem,               /* (1) Pointer to semaphore                          */
                  10,                   /*     Wait up until this time for the semaphore     */
                  OS_OPT_PEND_BLOCKING, /*     Option(s)                                     */
                  &ts,                  /*     Returned timestamp of when sem. was released  */
                  &err);                /*     Pointer to Error returned                     */
        :
        /* Check "err" */               /* (2)                                               */
        :
        OSSemPost(&MySem,               /* (3) Pointer to semaphore                          */
                  OS_OPT_POST_1,        /*     Option(s) … always OS_OPT_POST_1              */
                  &err);                /*     Pointer to Error returned                     */
        /* Check "err" */
        :
        :
    }
}
```

Listing 13-11 **Pending on and Posting to a Semaphore**

L13-11（1）调用 OSSemPend()，该函数它首先检测传递给它的参数是否有效。

如果信号量计数值大于 0（OS_SEM.Ctr），参数均有效，信号量计数值会被递减，并且 OSSemPend()返回。任务就可以占用该共享资源。

如果选择了 OS_OPT_PEND_NON_BLOCKING 模式，当信号量无效时 OSSemPend()迅速返回错误代号。

如果选择了 OS_OPT_PEND_BLOCKING 模式，信号量无效时任务会被放入该信号量的挂起队列中，并根据优先级排序，高优先级任务排在队列是首部。

如果设置了非 0 的等待期限，该任务也会被插入到时基队列中，等待期限为 0 意味着要永远地等待下去直到信号量被释放。大多数情况下，设置无限的等待时间。设置等待期限可以解决的死锁问题，这比在应用级中防止死锁（不要在同一时间占用多个信号量）好多了。

OSSemPend()返回的原因：

1）获得了信号量

2）其它任务取消了该任务的挂起

3）等待超时

4）信号量被删除

OSSemPend()的返回代号包含这些信息。

L13—11（2）检测 OSSemPend()返回的错误代号，若为 OS_ERR_NONE 则表示任务获得这个信号量。

L13-11（3）任务对共享资源访问完毕后，必须调用 OSSemPost()释放这个信号量。同样的，OSSemPost()会检测传递给它的参数是否有效。

然后 OSSemPost()调用 OS_TS_GET()获得当前的时间戳，并保持在信号量中的变量中。再检查是否有其它任务等待这个信号量。

如果没有，则递增信号量计数变量值，然后返回。

如果有，OSSemPost()选取等待任务中优先级最高的任务并分配该信号量给这个任务。这个操作是快速的因为队列是依照优先级排序的。

任务调用 OSSemPost()时，也可以设置不调用调度器。这意味着当信号量被提交后，调度器不会被调用即使有更高优先级的任务等待信号量被释放。这可以让原任务执行其它的信号量提交函数（如果需要），直到所有的提交完成。

{原任务可以提交多个信号量，并在最后一次提交后开启调度器}

13-3-5 优先级反转

优先级反转是实时系统中的一个常见问题，仅存在于基于优先级的抢占式内核中。图 13-4 显示了一种优先级反转的情况。任务 H 的优先级高于任务 M，任务 M 的优先级高于任务 L。

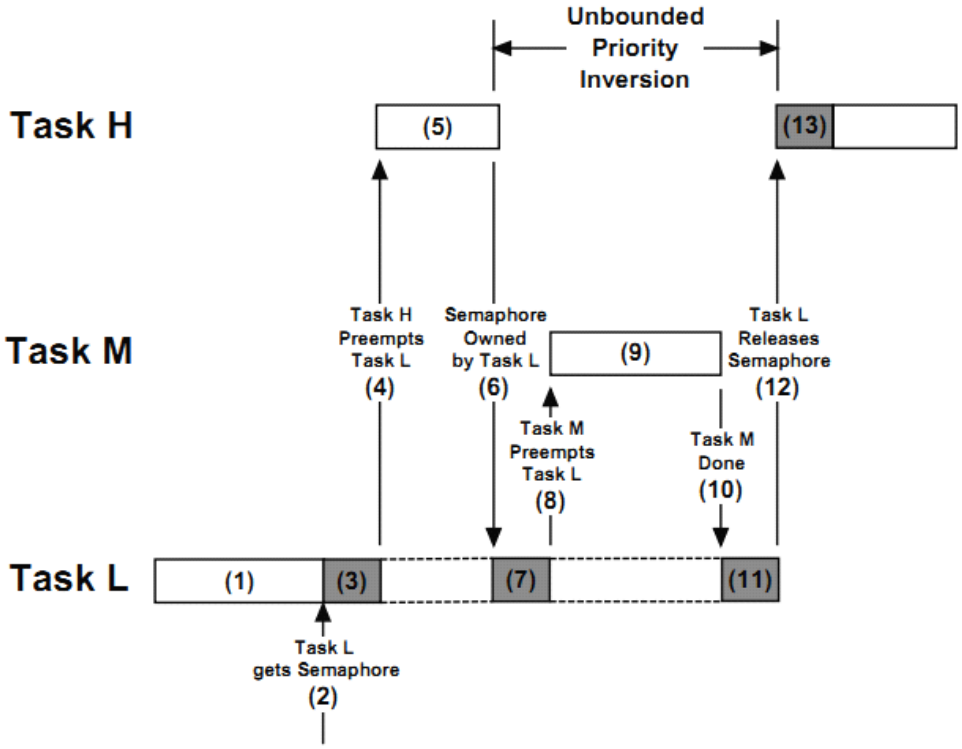


Figure 13-4 **Unbounded priority Inversion**

F13-4（1）任务 H（High 优先级）和任务 M（Medium 优先级）都在等待事件发生，任务 L（Low 优先级）正在被运行。

F13-4（2）这时，任务 L 想申请信号量。

F13-4（3）任务 L 访问共享资源。

F13-4（4）任务 H 所等待的事件发生，uC/OS-III 挂起任务 L。

F13-4（5）开始执行任务 H。

F13-4（6）现在任务 H 想要访问任务 L 所占用的共享资源（此时该共享资源被任务 L 占用）。因为该资源被任务 L 占用，任务 H 被放入挂起队列中等待这个信号量被释放。

F13-4（7）任务 L 被恢复并继续执行。

F13-4（8）任务 L 被任务 M 抢占因为任务 M 所等待的事件发生。

F13-4（9）任务 M 开始执行。

F13-4（10）任务 M 执行完毕，uC/OS-III 将 CPU 控制权交还给任务 L。

F13-4（11）任务 L 被执行。

F13-4（12）任务 L 执行完毕并释放资源。此时，任务 H 获得该资源，uC/OS-III 上下文切换到任务 H。

F13-4（13）任务 H 开始执行。

可以看出，任务 H 在任务 L 后被执行，因为任务 H 等待任务 L 所占用的资源。麻烦在于任务 M 抢占任务 L，若任务 M 需要执行很长时间，则任务 H 会被延迟很长时间才执行，这叫做优先级反转。

可以通过提升任务 L 的优先级解决这种问题（只在该任务访问共享资源时），访问结束后就恢复任务的优先级。任务 L 的优先级需要被上升到任务 H 的优先级。

13-4 互斥信号量 mutex

uC/OS-III 支持一种特殊类型的二值信号量叫做 mutex，用于解决优先级反转问题。图 13-5 显示了优先级反正是如何通过 mutex 解决的。

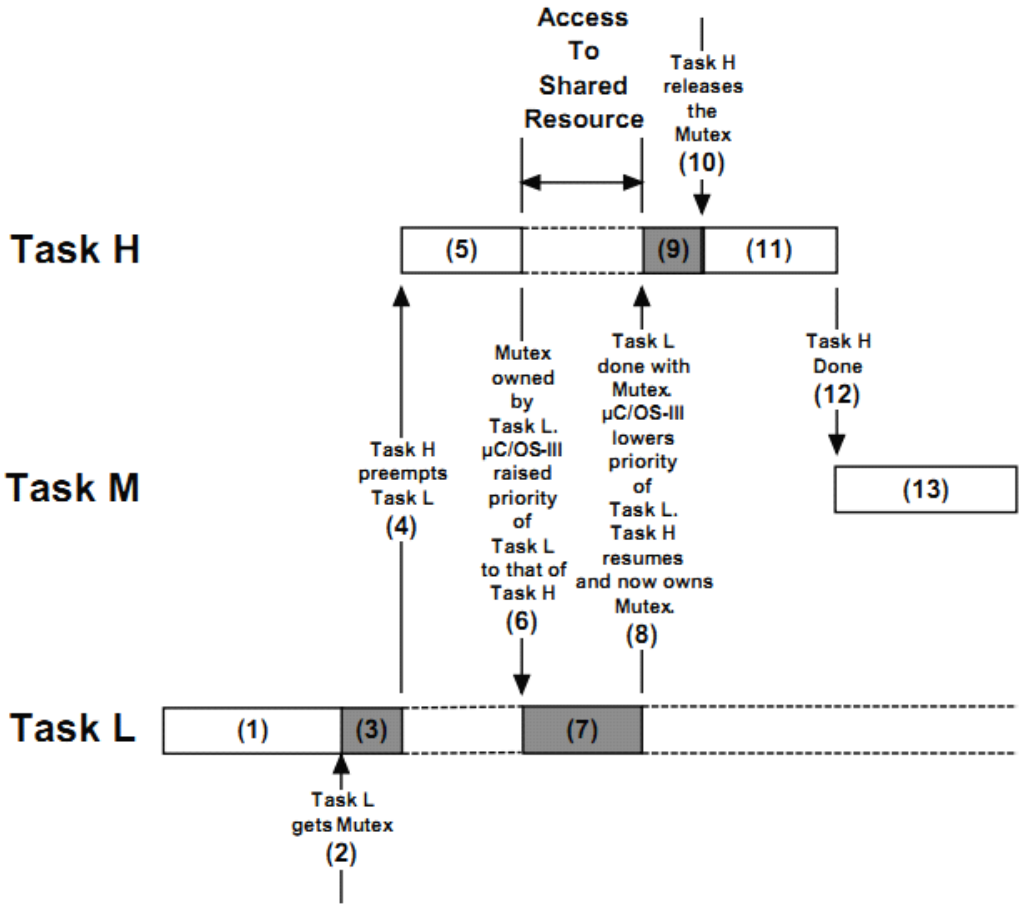


Figure 13-5 **Using a mutex to share a resource**

F13-5（1）任务 H 和任务 M 等待事件的发生，任务 L 被执行。

F13-5（2）此时，任务 L 申请并获得一个 mutex。

F13-5（3）任务 L 被执行。

F13-5（4）任务 H 和任务 M 所等待的事件发生，任务 L 被抢占。

F13-5（5）任务 H 开始执行。

F13-5（6）任务 H 想要访问任务 L 占用的共享资源。考虑到任务 L 占用这个资源，uC/OS-III 提升任务 L 的优先级与任务 H 相同。这样就防止了任务 L 被任务 M 抢占（被中等优先级的任务抢占）。

F13-5（7）任务 L 继续访问这个资源，但现在任务 L 的优先级等于任务 H 的优先级。注意的是任务 H 被挂起因为它要等待任务 L 释放 mutex。换句话说，任务 H 被插入到 mutex 的挂起队列中。

F13-5（8）任务 L 对共享资源的访问执行完毕并释放 mutex。uC/OS-III 恢复任务 L 到原有的优先级。然后 uC/OS-III 将这个 mutex 交给任务 H。{并不是任务 L 已经全部执行完毕，而是当它释放信号量时就会发生调度而转到高优先级任务}

F13-5（9）任务 H 开始执行。

F13-5（10）任务 H 对共享资源的访问完毕，并释放这个 mutex。

F13-5（11）任务 H 继续执行。

F13-5（12）任务 H 执行完毕。uC/OS-III 将 CPU 控制权交给任务 M。

F13-5（13) 任务 M 被执行。

现在没有优先级反转的问题了。当然，任务 L 占用共享资源的时间越短越好。

mutex 是一个内核对象，它被数据类型 OS_MUTEX 所定义，(OS_MUTEX 的原型是 os_mutex，见 OS.H)。应用中可以有任意个 mutex（仅限于处理器的 RAM）。

只有任务才可以使用 mutex（ISR 不可以使用 mutex）。

uC/OS-III 允许任务嵌套占有 mutex。如果任务获得 mutex，那么它可以嵌套获得这个 mutex 多达 250 次。该任务需要释放相同次数才能释放掉这个 mutex。在一些情况下，应用中很难知道任务调用了多少次 OSMutexPend()。如列表 13-12 所示

```
OS_MUTEX           MyMutex;
SOME_STRUCT        MySharedResource;


void  MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    :
    while (DEF_ON) {
        OSMutexPend((OS_MUTEX *)&MyMutex,                    (1)
                    (OS_TICK   )0,
                    (OS_OPT    )OS_OPT_PEND_BLOCKING,
                    (CPU_TS   *)&ts,
                    (OS_ERR   *)&err);
        /* Check 'err"                                  */   (2)
        /* Acquire shared resource if no error          */
        MyLibFunction();                                     (3)
        OSMutexPost((OS_MUTEX *)&MyMutex,                    (7)
                    (OS_OPT    )OS_OPT_POST_NONE,
                    (OS_ERR   *)&err);
        /* Check "err"                                  */
    }
}
```

```
void  MyLibFunction (void)
{
    OS_ERR  err;
    CPU_TS  ts;


    OSMutexPend((OS_MUTEX *)&MyMutex,                          (4)
                (OS_TICK   )0,
                (OS_OPT    )OS_OPT_PEND_BLOCKING,
                (CPU_TS   *)&ts,
                (OS_ERR   *)&err);
    /* Check "err"                          */
    /* Access shared resource if no error  */                  (5)
    OSMutexPost((OS_MUTEX *)&MyMutex,                          (6)
                (OS_OPT    )OS_OPT_POST_NONE,
                (OS_ERR   *)&err);
    /* Check "err"                          */
}
```

Listing 13-12 **Nesting calls to OSMutexPend()**

L13-12（1）任务请求并获得 mutex。OSMutexPend()设置嵌套计数值为 1。

L13-12（2）检测返回的错误代号，如果没有错误，任务 Mytask()占用共享资源 MySharedResource。

L13-12（3）函数 MyLibFunction()，其中将访问共享资源。

L13-12（4）函数 MyLibFunction()在访问共享资源前需获得 mutex。因为先前已经获得 mutex，所以没必要再次获得。但是，函数 MyLibFunction()可能被其它不需要访问共享资源的函数调用。uC/OS-III 允许嵌套 mutex，所以 MyLibFunction()可以再次被调用。嵌套计数值增加到 2.

L13-12（5）MyLibFunction()访问共享资源。

L13-12（6）mutex 被释放，嵌套计数值减为 1，OSMutexPost()返回，MyLibFunction()返回。mutex 还是被这个任务占用。

L13-12（7）mutex 再次被释放，此时，嵌套计数值减为 0。其它任务可以获得这个 mutex。

通常检测 OSMutexPend()的返回值，确保已经获得了 mutex。因为在这些情况下，OSMutexPend()也将返回错误代号：mutex 被删除、被其它任务调用 OSMutexPendAbort()取消了对该 mutex 的申请。

一般而言，不用在临界段中调用函数。

表 13-3 列出了与 mutex 相关的函数。然而，在这个章节中，我们只介绍三个经常用到的函数 OSMutexCreate()，OSMutexPend()，OSMutexPost()。其它任务在附录 A 中介绍。

函数名	功能
OSMutexCreate()	创建一个 mutex
OSMutexDle()	删除一个 mutex
OSMutexPend()	等待一个 mutex
OSMutexPendAbort()	任务取消等待 mutex
OSMutexPost()	释放 mutex

表 13-3　mutex 的 API 总结

13-4-1 mutex 的内部机制

```
typedef  struct  os_mutex  OS_MUTEX;          (1)


struct  os_mutex {
    OS_OBJ_TYPE          Type;                (2)
    CPU_CHAR            *NamePtr;             (3)
    OS_PEND_LIST         PendList;            (4)
    OS_TCB              *OwnerTCBPtr;         (5)
    OS_PRIO              OwnerOriginalPrio;   (6)
    OS_NESTING_CTR       OwnerNestingCtr;     (7)
    CPU_TS               TS;                  (8)
};
```

Listing 13-13 **OS_MUTEX data type**

L13-13（1）在 uC/OS-III 中，所有的结构体都会被定义一个数据类型。所有的数据类型以"OS_"开头且全部大写。用数据类型 OS_MUTEX 定义 mutex。

L13-13（2）结构体开始于"Type"域，让 uC/OS-III 能识别它是一个 mutex。

L13-13（3）每个内核对象都被分配一个名字。

L13-13（4）因为可能有多个任务等待这个 mutex，mutex 中包含了一个挂起队列用于存放等待该 mutex 的任务。

L13-13（5）如果任务占用这个 mutex，那么该变量.OwnerTCBPtr 会指向占用这个 mutex 的任务的 OS_TCB。

L13-13（6）如果任务占用这个 mutex，那么该变量 OwnerOriginalPrio 中存放着任务的原优先级，当占用 mutex 任务的优先级被提升时就会用到这个变量。

L13-13（7）uC/OS-III 允许任务占用同样的 mutex 多达 250 次。但也需要释放相同次数才能真正释放调这个 mutex。

L13-13（8）mutex 中的变量 TS 用于保存该 mutex 最后一次被释放的时间戳。当 mutex 被释放，读取时基计数值并存放到该变量中。

用户代码不能直接访问这个结构体。必须通过 uC/OS-III 提供的 API 访问。

mutex 被使用前必须被创建。列表 13-14 显示了如何创建一个 mutex。

```
OS_MUTEX  MyMutex;                          (1)


void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    OSMutexCreate(&MyMutex,                 (2)
                  "My Mutex",               (3)
                  &err);                    (4)
    /* Check "err" */
    :
    :
}
```

Listing 13-14 **Creating a mutex**

L13-14（1）定义一个 OS_MUTEX 类型的变量。

L13-14（2）调用 OSMutexCreate()创建一个 mutex，传递变量地址到该函数的第一个参数。

L13-14（3）分配一个名字给 mutex。

L13-14（4）OSMutexCreate()基于创建的结果返回一个错误代号。如果函数中所有的参数都是有效的，那么错误代号将是 S_ERR_NONE。

注意：mutex 是二值信号量，所以没必要初始化计数值。

任务占用共享资源前必须获得 mutex。通过调用 OSMutexPend()申请这个信号量。如列表 13-15 所示

```
OS_MUTEX  MyMutex;


void MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;
    :
    while (DEF_ON) {
        :
        OSMutexPend(&MyMutex,             /* (1) Pointer to mutex                          */
                    10,                   /*     Wait up until this time for the mutex     */
                    OS_OPT_PEND_BLOCKING, /*     Option(s)                                 */
                    &ts,                  /*     Timestamp of when mutex was released      */
                    &err);                /*     Pointer to Error returned                 */
        :
        /* Check "err"                       (2)                                          */
        :
        OSMutexPost(&MyMutex,             /* (3) Pointer to mutex                          */
                    OS_OPT_POST_NONE,
                    &err);                /*     Pointer to Error returned                 */
        /* Check "err"                                                                    */
        :
        :
    }
}
```

Listing 13-15 **Pending (or waiting) on a Mutual Exclusion Semaphore**

L13-15（1）调用 OSMutexPend()时先检测传递给它的参数释放有效。如果 mutex 有效，OSMutexPend()将 mutex 分配给该任务，并将该任务的 OS_TCB 地址放入 mutex 的指针 OwnerTCPPtr 中，将该任务的优先级放入 mutex 的变量 OwnerOriginalPrio 中，设置 mutex 的嵌套

计数值为 1。然后 OSMutexPend()返回的错误代号为 OS_ERR_NONE。

如果该任务已经占用这个 mutex，OSMutexPend()只是简单地将嵌套计数值递增。并返回错误代号 OS_ERR_MUTEX_OWNER。

如果其它任务已经占用这个 mutex，该任务调用"pend"函数时为 OS_OPT_PEND_NON_BLOCKING 模式，OSMutexPend()立即返回错误代号而不被挂起，因为任务不想等到该 mutex 被释放。

如果 mutex 已被更低优先级任务占用，uC/OS-III 会提升这个任务的优先级与该任务的优先级相同。

如果其它高优先级任务已经占用这个 mutex，该任务调用"pend"函数时为 OS_OPT_PEND_BLOCKING 模式，该任务会被放入挂起队列直到 mutex 被释放。任务在队列中根据优先级被排序，最高优先级排在队列的首部。

如果设置的等待期限为非 0，任务还会被插入到时基队列。如果等待期限为 0，则该任务不会被插入时基队列（也就是说任务会一直等待该 mutex 被释放）。

调度器被调用，因为当前任务不再被执行（它在等待 mutex）。

mutex 被释放后，任务获得 mutex。

需检测为什么 OSMutexPend()会返回，有以下 4 种可能：

1)任务获得 mutex

2)在 mutex 中的等待任务被其它任务取消等待

3)等待超时

4)这个 mutex 被删除

可以通过 OSMutexPend()函数返回的错误代号可以知道执行该函数的结果。

L13-15（2）如果 OSMutexPend()返回的错误代号为 OS_ERR_NONE，就表明该 mutex 被任务占用。如果错误代号为其它值，OSMutexPend()就会暂停。检测 OSMutexPend()返回的错误代号是很重要的。

L13-15（3）当任务完成对共享资源的访问后，就必须调用 OSMutexPost()。同样的，OSMutexPost()先检测传递给它的参数。OSMutexPost()调用 OS_TS_GET()获得当前的时间戳并放入 mutex 中的变量中，其它任务调用 OSMutexPend()时可以读取到这个时间戳。

OSMutexPost()递减嵌套计数值，如果该值依旧非 0，OSMutexPost()返回。在这种情况下，mutex 的占用任务不是全部地释放了 mutex。返回的错误代号为 OS_ERR_MUTEX_NESTING。

Mutex 被完成释放后，如果没有任务等待这个 mutex，OSMutexPost()设置 mutex 中的 OwnerTCBPtr 为 NULL，并清空 mutex 的嵌套计数值。

如果 uC/OS-III 提升了占用 mutex 任务的优先级，OSMutexPost()中会设置任务为原有的优先级。

若有任务等待这个 mutex。OSMutexPost()运行后，挂起队列中优先级最高的任务占用这个 mutex。这个操作非常快因为队列是按优先级排序的。

注意：任何时候 mutex 只能提供给一个任务。事实上，推荐当你申请一个 mutex 时，最好不要申请其它内核对象。

13-5 用信号量代替 mutex

然而，如果该等待有期限，就必须优先考虑使用 mutex。因为信号量容易导致优先级反转。

13-6 死锁

死锁，就是两个任务互相等待对方所占用的资源的情况。

假定任务 T1 占用资源 R1，任务 T2 占用资源 R2。如列表 13-16 所示。

```
void T1 (void *p_arg)
{
    while (DEF_ON) {
        Wait for event to occur;          (1)
        Acquire M1;                       (2)
        Access  R1;                       (3)
        :
        :
        \--------  Interrupt!             (4)
        :
        :                                 (8)
        Acquire M2;                       (9)
        Access  R2;
    }
}
```

```
void  T2 (void *p_arg)
{
    while (DEF_ON) {
        Wait for event to occur;          (5)
        Acquire M2;                       (6)
        Access  R2;
        :
        :
        Acquire M1;                       (7)
        Access  R1;
    }
}
```

Listing 13-16 **Deadlock problem**

L13-16（1）假定任务 T1 所等待的事情发生，任务 1 被执行。

L13-16（2）任务 T1 申请 M1。(mutex 1)

L13-16（3）任务 T1 访问资源 R1。

L13-16（4）中断发生，中断中使能了任务 T2。由于任务 T2 的优先级高于任务 T1，CPU 切换到任务 T2。

L13-16（5）该中断即是任务 T2 所等待的事件。

L13-16（6）任务 T2 申请 M2 并占用资源 R2。

L13-16（7）任务 T2 申请 M1，但是 M1 已被任务 T1 占用。任务 T2 被挂起。

L13-16（8）uC/OS-III 切换到任务 T1。

L13-16（9）此时任务 T1 申请 M2，但是 M2 已经被任务 T2 占用。此时，两个任务互相等待，这就算死锁。

可以用以下方式防止死锁：

1）同一个时间不要申请多于一个 mutex

2）不要直接地申请 mutex（该申请放到器件驱动中和可重入函数中）

3）在处理之前先获得全部所需要的 mutex

4）任务间以同样的顺序申请资源

当申请信号量或 mutex 时允许设置期限，这样能防止死锁，但是同样的死锁可能稍后再次出现。

处理之前先获得全部所需要的 mutex 的伪代码如列表 13-7

```
void T1 (void *p_arg)
{
    while (DEF_ON) {
        Wait for event to occur;
        Acquire M1;
        Acquire M2;
        Access  R1;
        Access  R2;
    }
}


void  T2 (void *p_arg)
{
    while (DEF_ON) {
        Wait for event to occur;
        Acquire M1;
        Acquire M2;
        Access  R1;
        Access  R2;
    }
}
```

Listing 13-17 **Deadlock avoidance – acquire all first and in the same order**

任务间以同样的顺序申请资源的伪代码如 13-18 所示。这个例子类似于先前的例子，它不是先获得全部的 mutex，而是所有任务都是以相同顺序申请资源的。

```
void T1 (void *p_arg){
    while (DEF_ON) {
        Wait for event to occur;
        Acquire M1;
        Access  R1;
        Acquire M2;
        Access  R2;
    }
}


void  T2 (void *p_arg)
{
    while (DEF_ON) {
        Wait for event to occur;
        Acquire M1;
        Access  R1;
        Acquire M2;
        Access  R2;
    }
}
```

Listing 13-18 **Deadlock avoidance – acquire in the same order**

13-7 总结

mutex 的使用依赖于访问共享资源的操作能否快速执行。如表 13-4。

资源共享方式	什么时候该用
关中断方式	能很快地结束访问共享资源，不推荐使用这种方法，因为会导致中断延迟
锁调度器方式	访问共享资源比较久时。
信号量方式	当该共享资源经常被多个被使用时。但信号量可能会导致优先级反转。
mutex 方式	推荐使用这种方法访问共享资源，尤其当任务要访问的共享资源有截止时间。 uC/OS-III 的 mutex 有内置的优先级，这样可防止优先级倒置。 然而，mutex 方式慢于信号量方式，mutex 需执行额外的操作：改变信号量的优先级

14、同步

这个章节主要介绍任务是如何于 ISR 或其它任务同步运行的。

当 ISR 执行时，它可以发送信号量或消息给任务，表明任务所等待的事件发生。ISR 执行完毕后，调用调度器。推荐在任务级执行中断所需的大部分操作。因为这样会减少关中断的时间且利于调试。

uC/OS-III 中用于同步的两种机制：信号量和事件标志组。

14-1 信号量

正如第 13 章所介绍，信号量是绝大多数多任务内核所提供的协议机制。信号量最初用于控制共享资源的访问。信号量可用于 ISR 与任务间、任务与任务间的同步，如图 14-1 所示。

图中信号量被画成一个旗帜，用于表明该事件发生的标志。信号量的初始值通常为 0，表明没有事件发生。

“N”表示信号量可以被累计。初始化时也可以设置为非 0 值，表明已经有事件发生。ISR 或任务可以提交信号量多次，信号量计数值会记录该信号量一共可被多少次。

任务旁边的沙漏表示任务可以设置等待该信号量的期限。

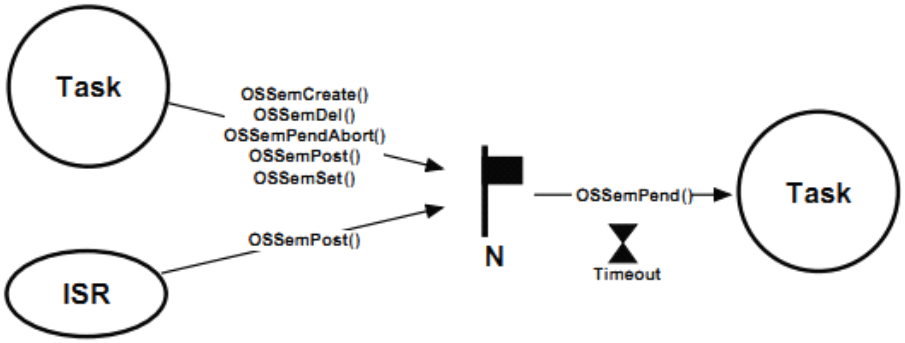


Figure 14-1 **µC/OS-III Semaphore Services**

信号量操作的相关函数如表 14-1 所示。注意的是所有的信号量函数都可以被任务调用，但是 ISR 中只能调用 OSSemPost()。

函数名	功能
OSSemCreate()	创建一个信号量
OSSemDel()	删除一个信号量
OSSemPend()	等待一个信号量
OSSemPendAbort()	取消等待该信号量
OSSemPost()	提交一个信号量
OSSemSet()	设置信号量计数值

表 14-1　信号量的 API 总结

用于同步时，信号量计数值中记录了它该信号量可被分配的次数。该计数值在 0 到 25565535，或 0 到 4294987295 之间，决定于该计数变量的位数，8 位，16 位，32 位。特别的，信号量计数值的上限为 OS_SEM_CTR(见 OS_TYPE.H)。

14-1-1 单向同步

图 14-2 显示了通过信号量，任务可以与 ISR 或任务同步。通过信号量的传递，这表明了 ISR 或任务发生了。使用信号量实现同步叫做单向同步。

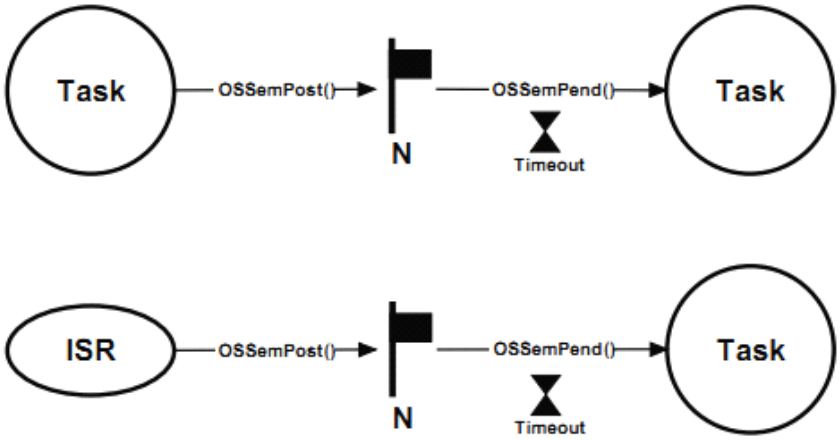


Figure 14-2 **Unilateral Rendezvous**

当任务要使用 I/O 端口，它就需获得信号量而调用 OSSemPend()。当任务完成对 I/O 端口的访问完成后，就必须调用 OSSemPost()释放这个信号量。这个过程是单向同步的。如图 14-3，上述操作的 ISR 和任务代码如列表 14-1 所示。

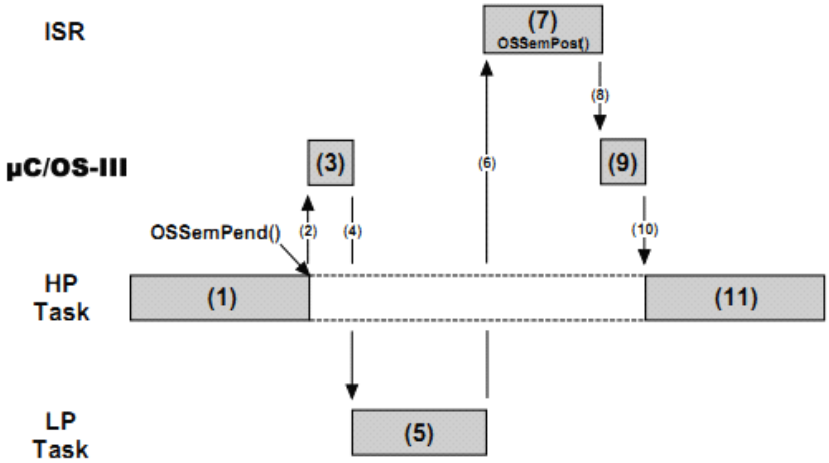


Figure 14-3 **Unilateral Rendezvous, Timing Diagram**

F14-3（1）任务 H 被执行。该任务与 ISR 同步（也就是等待 ISR 的发生），然后调用 OSSemPend()申请一个信号量。

F14-3（2）转向 uC/OS-III 函数。

F14-3（3）OSSemPend()的相关操作。

F14-3（4）因为 ISR 尚未发生，任务 H 被放入挂起队列，并调用调度器。

F14-3（5）任务 L 被执行。

F14-3（6）中断发生，任务 L 被保存，CPU 转向 ISR。

F14-3（7）开始执行 ISR。

F14-3（8）ISR 中调用 OSSemPost()提交了任务 H 所等待的信号量。

F14-3（9）信号量被发送给任务 H。

F14-4（10）任务 H 被就绪，调度器将 CPU 的控制权交给任务 H。

F14-4（11）任务 H 获得信号量，并继续执行。

```
OS_SEM  MySem;


void MyISR (void)
{
    OS_ERR  err;


    /* Clear the interrupting device */
    OSSemPost(&MySem,                    (7)
              OS_OPT_POST_1,
              &err);
    /* Check "err" */
}


void MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;
    :
    :
    while (DEF_ON) {
        OSSemPend(&MySem,                (1)
                  10,
                  OS_OPT_PEND_BLOCKING,
                  &ts,
                  &err);
        /* Check "err" */                (11)
        :
        :
    }
}
```

Listing 14-1 **Pending (or waiting) on a Semaphore**

注意到一些有趣的事情：第一，uC/OS-III 总是让优先级最高的就绪任务被执行，最大化地利用了 CPU。第二，任务等待信号量时是不消耗 CPU 的执行时间。最后，当任务所等待的信号量出现时，uC/OS-III 能迅速地告知任务，并调用调度器。

14-1-2 信号量计数值

前面提到，信号量计数值中保存了它还能被分配多少次。换句话说，当 ISR 提交该信号量 n 次，那么该信号量计数值就会增加 n。如图 14-4

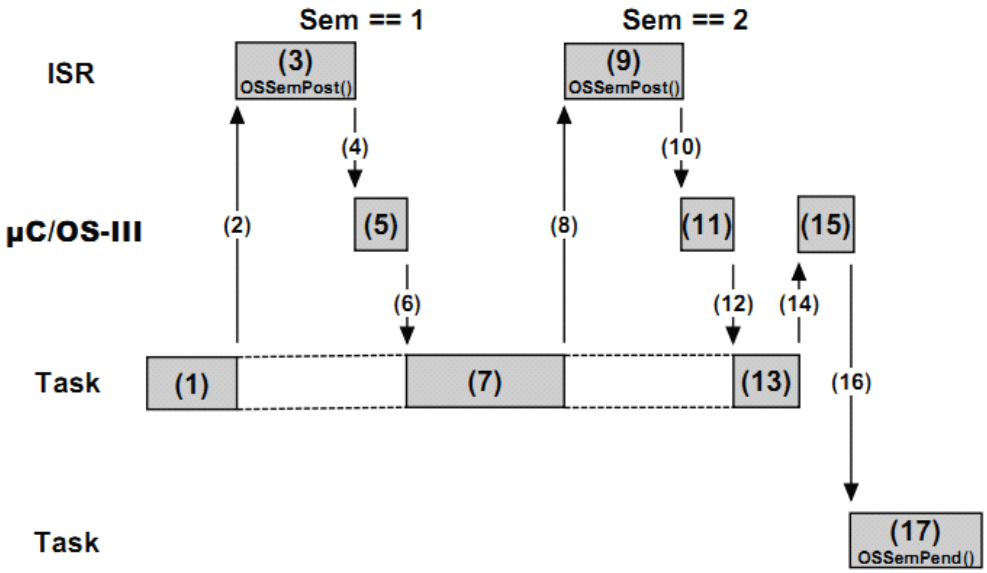


Figure 14-4 **Semaphore Credit Tracking**

F14-4（1）高优先级任务 H 被执行。

F14-4（2）中断发生。

F14-4（3）ISR 中提交了一个信号量。

F14-4（4）信号量被提交后，uC/OS-III 需进行一些操作。

F14-4（5）uC/OS-III 执行相应操作。

F14-4（6）由于没有更高优先级任务被就绪，uC/OS-III 继续执行任务 H。

F14-4（7）任务 H 被执行。

F14-4（8）中断发生。

F14-4（9）第二次中断发生了，ISR 中提交了一个信号量。

F14-4（10）信号量被提交后，uC/OS-III 需进行一些操作。

F14-4（11）uC/OS-III 执行相应操作。

F14-4（12）由于没有更高优先级任务被就绪，uC/OS-III 继续执行任务 H。

F14-4（13）任务 H 被执行。

F14-4（14）任务 H 执行完毕。

F14-4（15）uC/OS-III 执行相应的上下文切换工作。

F14-4（16）uC/OS-III 将 CPU 的控制权交给任务 L。

F14-4（17）任务 L 被执行。

14-1-3 多个任务等待一个信号量

多个任务可以同时等待同样的信号量，假设每个任务都被设置了定时期限。如图 14-5 所示

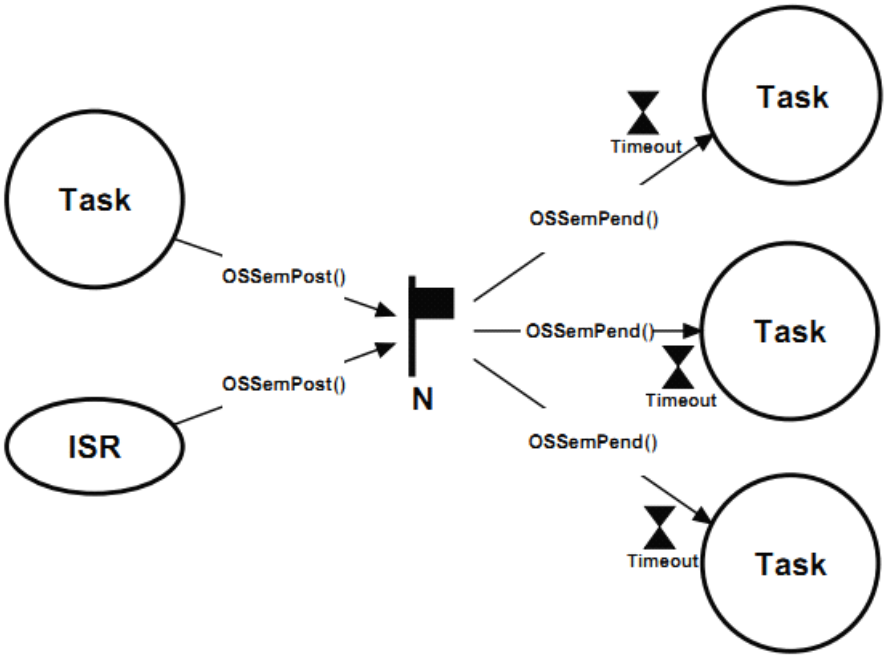


Figure 14-5 **Multiple Tasks waiting on a Semaphore**

当该信号量被提交时，uC/OS-III 会让挂起队列中优先级最高的任务就绪。然而，也可以让挂起队列中所有的任务被就绪，这叫做广播信号量，调用 OSSemPost()时选择参数 OS_OPT_POST_ALL 就能实现广播的功能。

广播用于多个任务间的同步。然而，若任务还需要与不在信号量挂起队列中的其它任务同步，可以同时使用信号量和事件标志组实现同步的功能。

14-2 任务内建信号量

经常通过发送信号量实现同步。每个任务都有内建的信号量，通过任务内建的信号量不仅可以简化信号量通信的代码而且更加有效。如图 14-7 所示

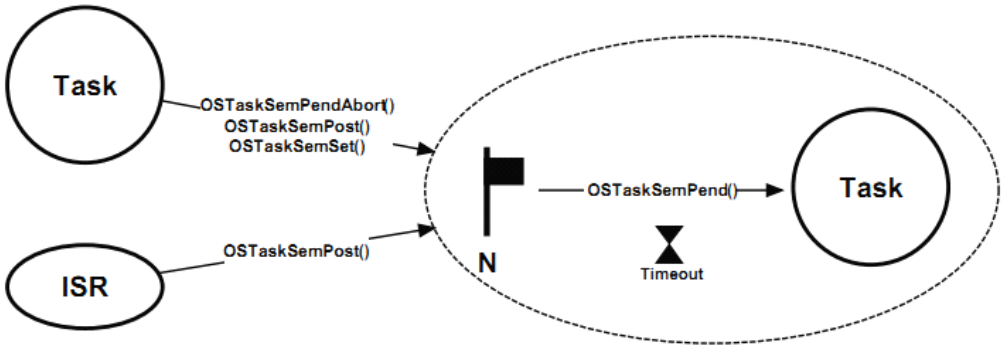


Figure 14-7 **Semaphore built-into a Task**

与任务内建的信号量相关的函数都是以 OSTaskSem???()为前缀的。相关的代码都在 OS_TASK.C 中。如表 14-2

函数名	功能
OSTaskSemPend()	等待一个任务信号量
OSTaskSemPendAbort()	取消等待
OSTaskSemPost()	发送信号量给任务
OSTaskSemSet()	设置信号量计数值

表 14-2 任务内建信号量函数总结

14-2-1 挂起（等待）任务信号量

当任务被创建时，也会内建一个信号量，信号量计数值初始化为0。等待任务信号量如列表 14-6 所示。

```
void MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;
    :
    while (DEF_ON) {
        OSTaskSemPend(10,                    (1)
                      OS_OPT_PEND_BLOCKING,  (2)
                      &ts,                   (3)
                      &err);                 (4)
        /* Check "err" */
    :
    :
}
```

Listing 14-6 **Pending (or waiting) on Task's internal semaphore**

L14-6（1）任务通过调用 OSTaskSemPend()等待任务信号量（其它任务）。第一个参数为等待期限，以时基为单位。

L14-6（2）第二个参数是挂起方式。一共有两种方式：OS_OPT_PEND_BLOCKING 和 OS_OPT_PEND_NON_BLOCKING。

L14-6（3）函数返回时，该参数中保存着信号量被提交时的时间戳。

L14-6（4）函数返回时，根据执行结果保存着一个错误代号。

14-2-2 提交（标记）任务信号量

ISR 或任务通过调用 OSTaskSemPost()提交任务信号量，如列表 14-7 所示

```
OS_TCB  MyTaskTCB;


void MyISR (void *p_arg)
{
    OS_ERR  err;
    :
    OSTaskSemPost(&MyTaskTCB,           (1)
                  OS_OPT_POST_NONE,     (2)
                  &err);                (3)
    /* Check "err" */
    :
    :
}
```

Listing 14-7 **Posting (or signaling) a Semaphore**

L14-7（1）通过调用 OSTaskSemPost()将任务信号量提交给任务。该参数为目标任务的 OS_TCB 地址。

L14-7（2）这个参数为提交方式的选择，一个有两种提交方式：

OS_OPT_POST_NONE 提交完成后调用调度器。

OS_OPT_POST_NO_SCHED 提交完成后调用调度器。（提交多个信号量时，可以只执行一次调度）

L14-7（3）函数返回时，根据执行结果保存着一个错误代号。

14-2-3 双向同步

两个任务间可以用两个信号量实现双向同步，如图 14-8。任务与 ISR 间不能双向同步，因为 ISR 中不能等待信号量（ISR 中不能有阻塞呼叫）。

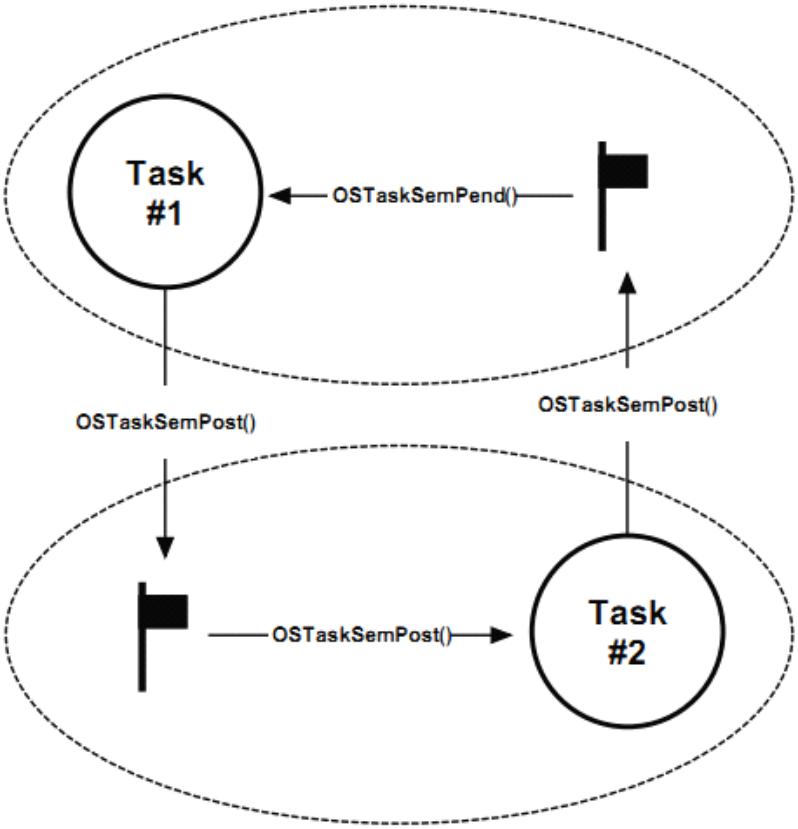


Figure 14-8 **Bilateral Rendezvous**

双向同步的代码如列表 14-8 所示。当然，双向同步可以由两个外部信号量实现，但使用任务信号量会更加简单。

```
OS_TCB  MyTask1_TCB;
OS_TCB  MyTask2_TCB;


void Task1 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    while (DEF_ON) {
        :
        OSTaskSemPost(&MyTask2_TCB,             (1)
                      OS_OPT_POST_NONE,
                      &err);
        /* Check 'err" */
        OSTaskSemPend(0,                        (2)
                      OS_OPT_PEND_BLOCKING,
                      &ts,
                      &err);
        /* Check 'err" */
        :
    }
}


void Task2 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    while (DEF_ON) {
        :
        OSTaskSemPost(&MyTask1_TCB,             (3)
                      OS_OPT_POST_NONE,
                      &err);
        /* Check 'err" */
        OSTaskSemPend(0,                        (4)
                      OS_OPT_PEND_BLOCKING,
                      &ts,
                      &err);
        /* Check 'err" */
        :
    }
}
```

Listing 14-8 **Tasks synchronizing their activities**

L14-8（1）任务 1 发送信号量给任务 2。

L14-8（2）任务 1 等待其内部的信号量，与任务 2 同步。因为任务 2 尚未被执行，任务 1 被挂起，等待任务 2 给其发送信号量。

L14-8（3）切换到任务 2 并执行，给任务 1 发送信号量。

L14-8(4)此时任务 2 与任务 1 同步。如果任务 1 优先级较高，uC/OS-III 切换到任务 1，否则，任务 2 继续执行。

14-3 事件标志组

当任务要与多个事件同步时可以使用事件标志。若其中的任意一个事件发生时任务被就绪，叫做逻辑或(OR)。若所有的事件都发生时任务被就绪，叫做逻辑与（AND）。如图 14-9 所示。

用户可以创建任意个事件标志组（限制于 RAM）。uC/OS-III 中与事件标志组相关的函数都是以 OSFlag???()为前缀。与事件标志组相关的函数代码都在 OS_FLAG.C 中。

设置 OS_CFG.H 中的 OS_CFG_FLAG_EN 为 1 开启事件标志组功能。

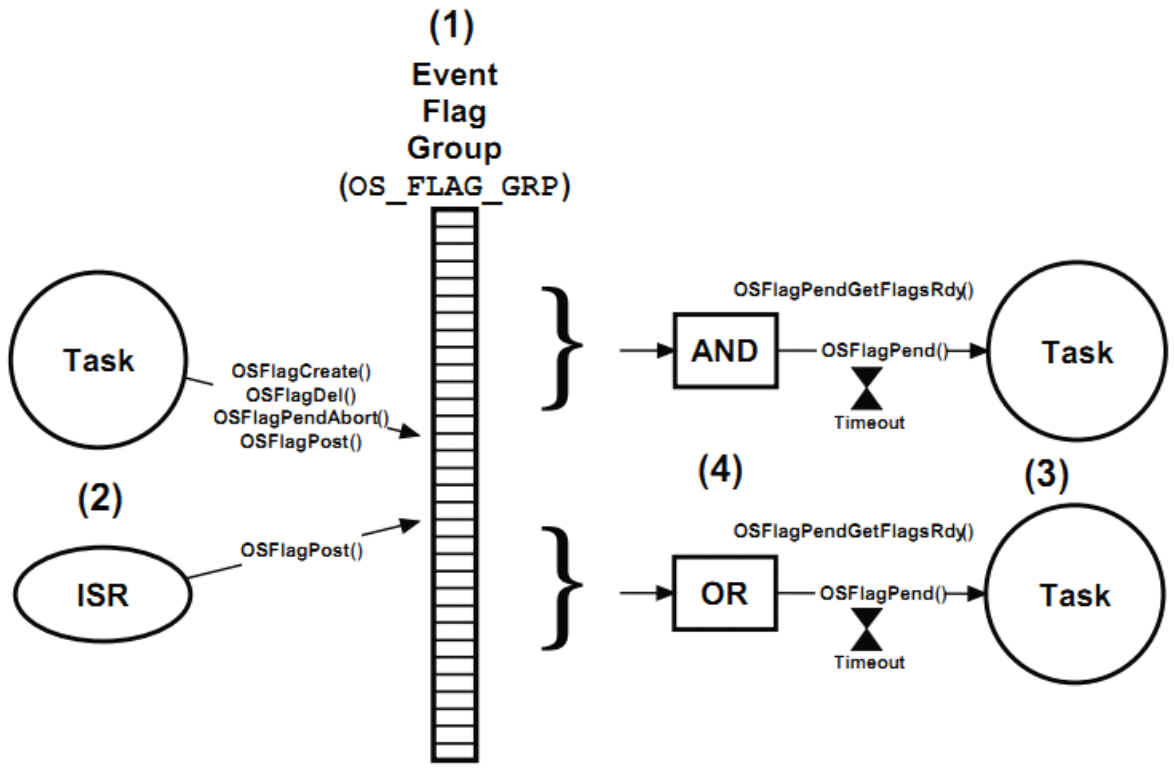


Figure 14-9 **Event Flags**

F14-9（1）事件标志组是 uC/OS-III 的内核对象，以 OS_FLAG_GRP 为数据类型（见 OS.H）。 它可以是 8 位，16 位，32 位，决定于 OS_TYPE.H 中所定义的 OS_FLAGS。事件标志组中的位是任务所等待事件是否发生的标志。事件标志组必须在创建后使用。

F14-9（2）任务或 ISR 可以提交标志。然而，只有任务可以将在事件标志组中等待的其它任务删除，取消等待，只有任务才能让任务在事件标志组中等待。

F14-9（3）任务可以等待事件标志组中的任意个位被设置。等待也可以被设置期限，以时基为单位。

F14-9（4）任务等待事件标志组中的位，可以被设置为 OR 模式，或者是 AND 模式。

函数名	功能
OSFlagCreate()	创建一个事件标志组
OSFlagDel()	删除一个事件标志组
OSFlagPend()	在事件标志组中挂起
OSFlagPendAbort()	取消等待
OSFlagPendGetFalgsRdy()	获得事件标志组中导致任务被就绪的位
OSFlagPost()	提交标志到事件标志组

表 14-3 事件标志组 API 总结

14-3-1 使用事件标志组

当任务或 ISR 提交标志到事件标志组，满足条件的任务会被就绪。

事件标志组中位的含义由用户定义，应用中可以有多个事件标志组。在事件标志组中，可以定义位 0 表示温度过低（由温度传感器接收），位 1 表示电压过低，位 2 表示某个开关被按下等。任务或 ISR 检测这些传感器并调用 OSFlagPost()设置相应的标志位。任务可以调用 OSFlagPend()检测相应的标志是否发生。

列表 14-9 显示了如何使用事件标志组。

```
#define      TEMP_LOW   (OS_FLAGS)0x0001                          (1)
#define      BATT_LOW   (OS_FLAGS)0x0002
#define      SW_PRESSED (OS_FLAGS)0x0004

OS_FLAG_GRP  MyEventFlagGrp;                                      (2)

void main (void)
{
    OS_ERR  err;

    OSInit(&err);
    :
    OSFlagCreate(&MyEventFlagGrp,                                 (3)
                 "My Event Flag Group",
                 (OS_FLAGS)0,
                 &err);
    /* Check 'err" */
    :
    OSStart(&err);
}


void  MyTask (void *p_arg)                                        (4)
{
    OS_ERR  err;
    CPU_TS  ts;

    while (DEF_ON) {
        OSFlagPend(&MyEventFlagGrp,                               (5)
                   TEMP_LOW + BATT_LOW,
                   (OS_TICK )0,
                   (OS_OPT)OS_OPT_PEND_FLAG_SET_ANY,
                   &ts,
                   &err);
        /* Check 'err" */
        :
    }
}
```

```
void  MyISR (void)                                                (6)
{
    OS_ERR  err;
    :
    OSFlagPost(&MyEventFlagGrp,                                   (7)
               BAT_LOW,
               (OS_OPT)OS_OPT_POST_FLAG_SET,
               &err);
    /* Check 'err" */
    :
}
```

Listing 14-9 **Using Event Flags**

L14-9（1）设置相应位所表示的含义。

L14-9（2）定义一个事件标志组，其类型为 OS_FLAG_GRP。为了说明，假定事件标志组的位长为 16，（在 OS_TYPE.H 中设置）

L14-9（3）事件标志组必须被创建后才能使用，最后在应用的启动代码（或者说初始化代码）中就创建事件标志组。在这个例子中，事件标志组被赋予一个名字，且全部位被清零。

L14-9（4）假定 MyTask()在事件标志组中被挂起。

L14-9（5）调用 OSFlagPend()，并传入事件标志组地址，任务就会被挂起在事件标志组中。

第二个参数是任务等待哪些位被置位，这里设置了 2 个位。

第三个参数是等待时限。

第四个参数是等待的方式，若设置为 OS_OPT_FLAG_SET_ANY，表明两个位中只要有一个位被置位，任务就会被就绪。

第五个参数中存放了当事件标志被提交时的时间戳。

第六个参数是根据函数执行结果返回的错误代号。

L14-9（6）中断被设置为检测电池电压。当电压过低时，中断发生，ISR 就需要告诉任务，并让任务执行相应操作。

L14-9（7）通过调用 OSFlagPost()函数，ISR 中设置事件标志组中相应的位。第三个参数是选项参数，位被置一或清零。

事件标志组通常用于表示的短暂的事件或状态信息。通常用不同的事件标志组处理如列表 14-10。

ISR 或任务可以测量一些状态信息，例如温度、转速、电压等。用于状态信息时一般不设置等待时限。

ISR 或任务可以测量一些短暂的事件，如按下开关、爆炸等。用于短暂事件时一般设置了等待时限。

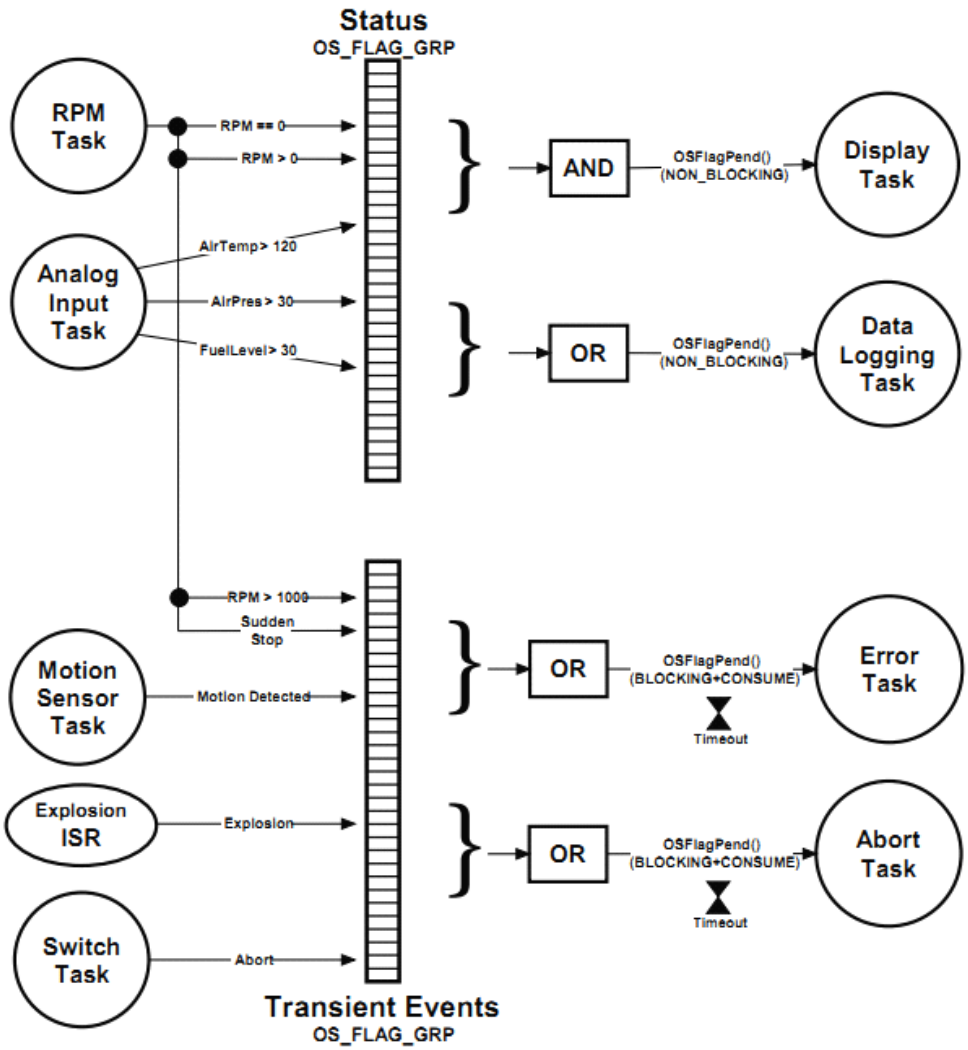


Figure 14-10 **Event Flags used for Status and Transient Events**

14-3-2 事件标志组内部结构

用户可以创建任意个事件标志组（仅限制于处理器的 RAM）。通过设置 OS_CFG.H 中的 OS_CFG_FLAG_EN 为 1 开启事件标志组功能。

事件标志组是一个内核对象，由数据类型 OS_FLAG_GRP 定义，该数据类型由 os_flag_grp 定义（见 OS.H）。与事件标志组相关的代码在 OS_FLAG.C 中。如列表 14-10。

```
typedef  struct  os_flag_grp  OS_FLAG_GRP;   (1)


struct  os_flag_grp {
    OS_OBJ_TYPE          Type;              (2)
    CPU_CHAR            *NamePtr;           (3)
    OS_PEND_LIST         PendList;          (4)
    OS_FLAGS             Flags;             (5)
    CPU_TS               TS;                (6)
};
```

Listing 14-10 **OS_FLAG_GRP data type**

L14-10（1）在 uC/OS-III 中，所有的结构体都会定义一个数据类型。都是以"OS_"开头并且全部大写。

L14-10（2）结构体的第一个变量为"Type"。用于辨认该对象为事件标志组。

L14-10（3）每个内核对象都可以被分配一个名字。

L14-10（4）因为可以有多个任务同时等待事件标志组中的事件，所以事件标志组中包含了一个用于控制挂起队列的结构体（见第十章）。

L14-10（5）事件标志组中包含了很多标志位，这个变量中保存了当前这些标志位的状态。这个变量可以为 8 位，16 位或 32 位。决定于

OS_TYPE.H 中的 OS_FLAGS 的位数。

L14-10（6）事件标志组中包含了一个变量，存储了最后一次标志被提交的时间戳。

用户代码不能直接访问事件标志组，必须通过 uC/OS-III 提供的函数访问事件标志组。

创建事件标志组，如列表 14-11 所示

```
OS_FLAG_GRP  MyEventFlagGrp;                (1)


void  MyCode (void)
{
    OS_ERR  err;
    :
    OSFlagCreate(&MyEventFlagGrp,           (2)
                 "My Event Flag Group",     (3)
                 (OS_FLAGS)0,               (4)
                 &err);                     (5)
    /* Check 'err" */
    :
}
```

Listing 14-11 **Creating a Event Flag Group**

L14-11（1）定义事件标志组。

L14-11（2）调用 OSFlagGreate()创建一个事件标志组。第一个参数为事件标志组的地址。

L14-11（3）给事件标志组赋予名字。名字必须以空字符结尾（不是空格）。

L14-11（4）一般情况下，将事件标志组中的所有标志位初始化为 0.

L14-11（5）返回一个错误代号。

调用 OSFlagPend()，任务等待一个或多个标志位被设置。

```
OS_FLAG_GRP  MyEventFlagGrp;


void MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;
    :
    while (DEF_ON) {
        :
        OSFlagPend(&MyEventFlagGrp,          /* (1) Pointer to event flag group        */
                   (OS_FLAGS)0x0F,           /*     Which bits to wait on              */
                   10,                       /*     Maximum time to wait               */
                   OS_OPT_PEND_BLOCKING +
                   OS_OPT_PEND_FLAG_SET_ANY, /*     Option(s)                          */
                   &ts,                      /*     Timestamp of when posted to        */
                   &err);                    /*     Pointer to Error returned          */
        /* Check "err"                          (2)                                    */
        :
        :
    }
}
```

Listing 14-12 **Pending (or waiting) on an Event Flag Group**

L14-12（1）OSFlagPend()开始时先检查参数的有效性。若函数所等待的标志位被设置，OSFlagPend()会返回哪个位被设置了。

如果设置为 OS_OPT_PEND_NON_BLOCKING，无论所等待的位有没有被设置，OSFlagPend()会立即返回，并返回错误代号。

如果设置为 OS_OPT_PEND_BLOCKING，所等待位没有被设置时任务会被放入挂起队列。可以设置等待时限。队列中的任务不是以优先级方式排序的，而是后入先出方式排序，也就是后到的任务会被放入队首。这是因为，无论哪个位被设置，uC/OS-III 会遍历整个队列的。

OSFlagPend()的返回有 4 种可能

1）所等待的标志位被设置

2）挂起被其它任务取消

3）等待超时

4）事件标志组被删除

检测 OSFlagPend()返回的错误代号就能知道是什么原因导致返回。

L14-12（2）如果 OSFlagPend()返回的错误代号为 OS_ERR_NONE，表明任务等待的位被设置，并正常返回。

调用 OSFlagPost()设置或清除标志位，如列表 14-13 所示

```
OS_FLAG_GRP  MyEventFlagGrp;

void MyISR (void)
{
    OS_ERR  err;
    :
    OSFlagPost(&MyEventFlagGrp,       (1)
               (OS_FLAGS)0x0C,        (2)
               OS_OPT_POST_FLAG_SET,  (3)
               &err);                 (4)
    /* Check 'err" */
    :
    :
}
```

Listing 14-13 **Posting flags to an Event Flag Group**

L14-13（1）ISR 或任务调用 OSFlagPost()提交标志位。这个参数是事件标志组的地址。

L14-13（2）第二个参数是时间标志组中的哪些位需要被设置或清除。

L14-13 （ 3 ） 可 选 ： OS_OPT_POST_FLAG_SET 或 OS_OPT_POST_FLAG_CLR。

如果设置为 OS_OPT_POST_FLAG_SET，第二个参数提交对应位被置 1。

如果设置为 OS_OPT_POST_FLAG_CLR，第二个参数提交对应位被置 0。

当还设置为 OS_OPT_POST_NO_SCHED（与上面的选项相加“+”，传入该参数）。任务可以提交标志位而不发生调度。当有多个事件标志组时，通过该设置，可以实现仅在最后一次提交时发生调度（需在最后一次提交时不设置该选项）。

L14-13（4）根据函数执行结果返回一个错误代号。

14-4 多任务同步

通过广播信号量实现多任务同步是通用的方法。显然的，在单 CPU 系统中，同一时间只能执行一个任务。然而，多个任务可以同时被就绪。这叫做多任务同步。然而，需要同步一些不需要接收广播的信号量的任务，解决这个问题的办法是将信号量和时间标志组同时用于同步。如图 14-11 所示。假定左边任务的优先级比右边任务的优先级低。

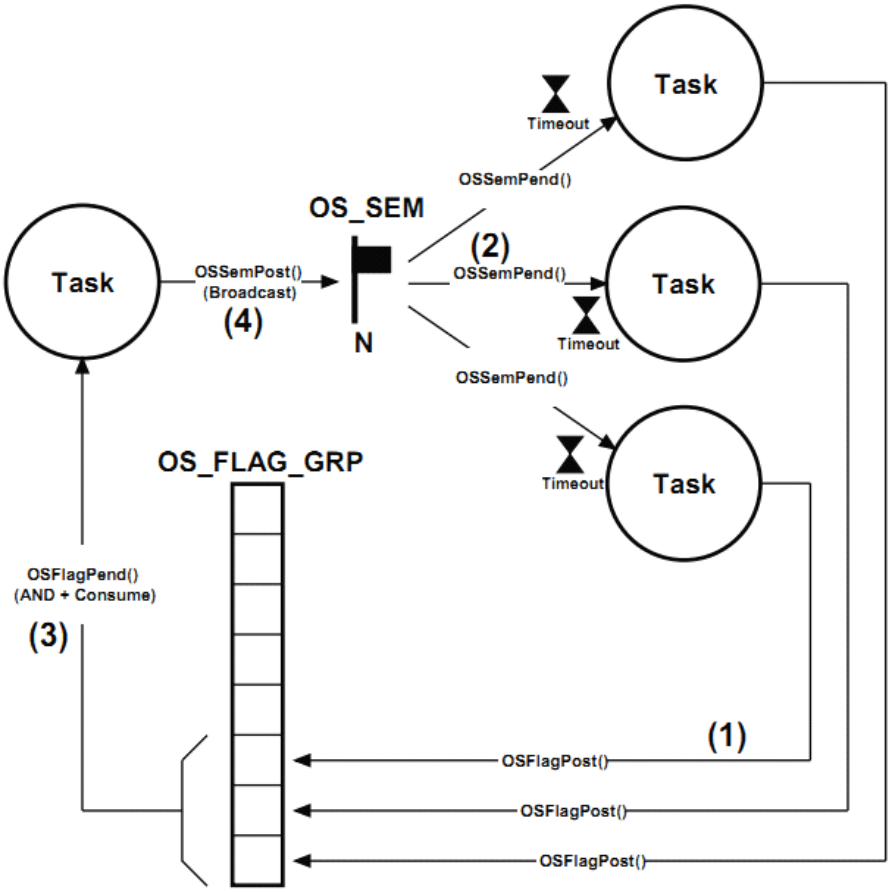


Figure 14-11 **Multiple Task Rendezvous**

F14-11（1）右边的每个任务都需要跟事件标志组的位对应。

F14-11（2）右边任务需等待信号量才能被就绪。

F14-11（3）当事件标志组中与需要同步的任务对应位都被置位后，左边任务才能广播信号量。

F14-11（4）左边任务广播信号量给右边任务。

14-5 总结

有三种方法可以让 ISR 或任务标记另一个（或多个）任务：信号量、任务信号量、事件标志组。

信号量和任务信号量中都有信号量计数值表示该信号量还可以被分配几次。若 ISR 或任务需要标记一个任务时，推荐使用任务信号量因为这可以避免定义一个外部信号量且更为有效。

当任务需要与一个或多个事件同步时使用事件标志组。

15、消息传递

有些情况下任务或 ISR 与另一个任务间进行通信，这种信息交换叫做作业间的通信。可以有两种方法实现这种通信：全局变量、发送消息。

正如第 13 章“资源管理”所介绍。如果使用全局变量，任务或 ISR 就须确保它独占该变量。如果防止被 ISR 嵌套，那么就只有关中断这种方法来保护这个变量。如果是任务间共享该变量，那么可以通过关中断、锁调度器、信号量、mutex 保护该变量。需注意的是：任务与 ISR 通信只能通过全局变量。如果全局变量被 ISR 改变，任务将不会知道全局变量被改变，除非该任务检测该变量或者 ISR 标记任务告知该变量被改变。

消息可以被发送到媒介—消息队列中，也可以直接发送给任务，因为 uC/OS-III 中每个任务都有其内建的消息队列。如果多个任务等待这个消息时就将该消息发送到外部的消息队列。当只有一个任务等待该消息时直接将消息发送给任务。

任务等待消息到达时，不占用 CPU。

15-1 消息

消息中包含一个指向数据的指针、该数据的大小、时间戳变量。该指针可以指向数据区域甚至是一个函数。当然，消息的发送方和消息的接收方都应该知道消息所包含的意义。换句话说，接收方知道接收发到消息的含义。

消息的内容（即数据）通常保留在其作用域中因为发送的是数据的地址而不是数据。换句话说，数据不是被拷贝并发送给任务，而是告诉任务数据的地址，并让任务自己去访问。

15-2 消息队列

消息队列是内核对象。事实上，可以分配任意个消息队列（只要处理器的 RAM 足够的话）。

通过消息队列用户可以做很多事情，如图 15-1.然而，ISR 中只能调用 OSQPost()。

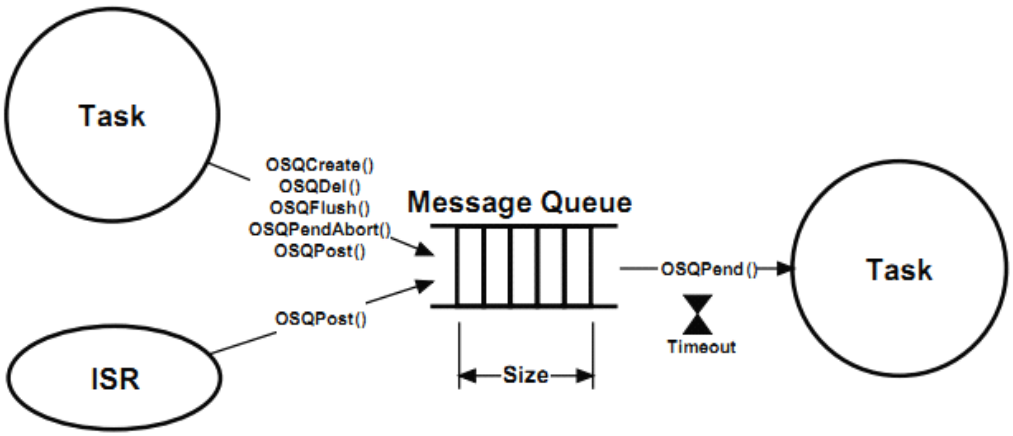


Figure 15-1 **Operations on message queue**

消息队列是先入先出模式（FIFO）。然而，uC/OS-III 也可以将其设置为后入先出模式（LIFO）。若任务或 ISR 发送紧急消息给另一个任务时，后入先出模式是非常有用的，在这种情况下，该紧急消息绕过消息队列中的其他消息。消息队列的长度可以在运行时设置。

如上图，接收任务旁的沙漏表示该任务可以设置等待期限。如果任务没有在规定时间内接收到该消息，uC/OS-III 会返回一个错误代号表示任务被就绪不是因为接收到消息，而是等待超时。

消息队列中存放了等待该消息的任务。多个任务可以在消息队列中等待消息，如图 15-2 所示。当一个消息被发送到消息队列时，等待该消息的高优先级任务接收这个消息。消息发送者可以广播这个消息给消息队列中的所有任务。在这种情况下，如果接收到消息中有优先级高于消息发送者优先级的任务，uC/OS-III 就会切换到这个高优先级的任务。注意：不是每个任务都需要设置等待期限，有些任务可能需要永远等待这个消息。

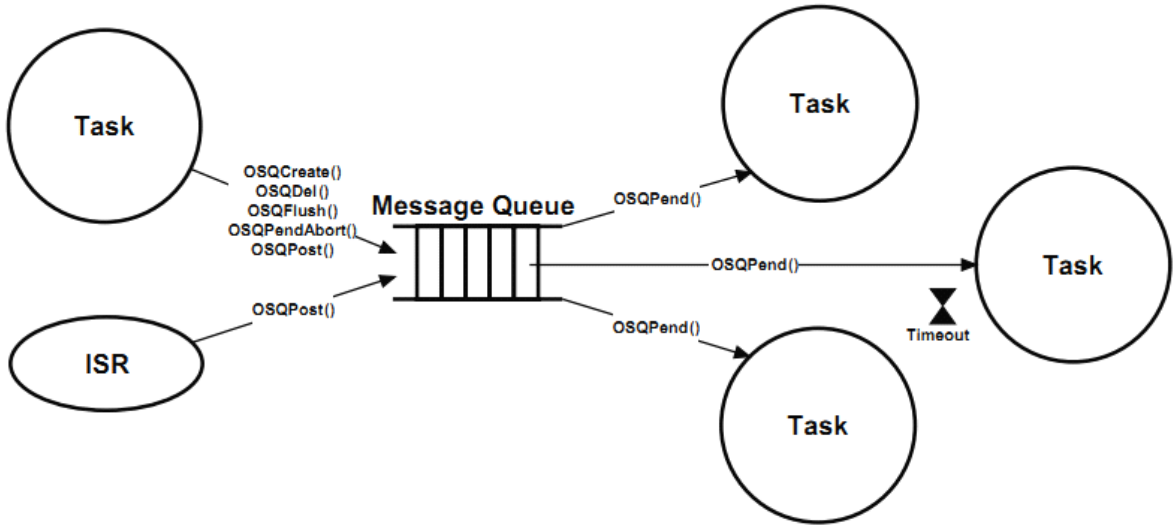


Figure 15-2 **Multiple tasks waiting on a message queue**

15-3 任务的消息队列

很少会见到多个任务同时在一个消息队列中等待。因为这样，uC/OS-III 在任务中内建了一个消息队列。用户可以直接发送消息给任务而不通过外部消息队列。这个特性不仅简化了代码，还提供了效率。每个任务都内建一个消息队列，如图 15-3 所示。

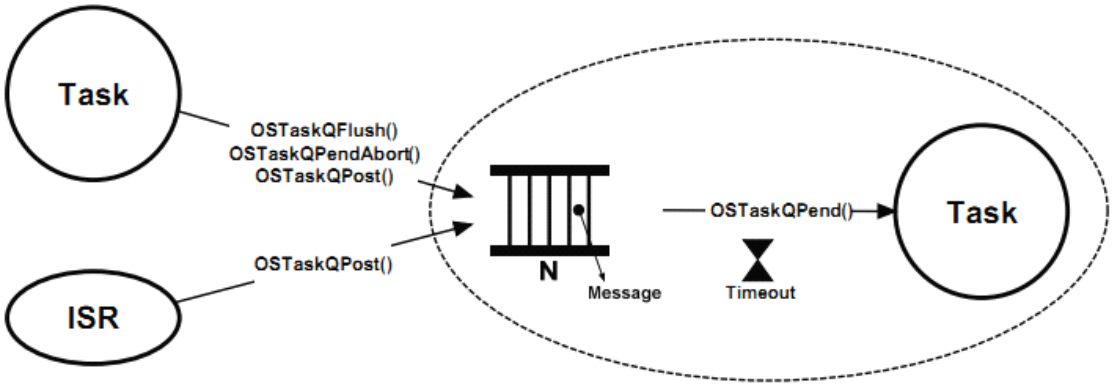


Figure 15-3 **Task message queue**

uC/OS-III 中与任务消息队列相关的服务都是以 OSTask???()开头的。设置 OS_CFG.H 中的 OS_CFG_TASK_EN 使能任务的消息队列服务。与任务消息队列相关的代码在 OS_TASK.C 中。

15-4 双向通信

两个任务可以通过两个消息队列同步，如图 15-4 所示。这叫做双向通信，这两个任务间可能互相发送消息给对方。任务与 ISR 间不能双向通信，因为 ISR 不能在消息队列中等待。

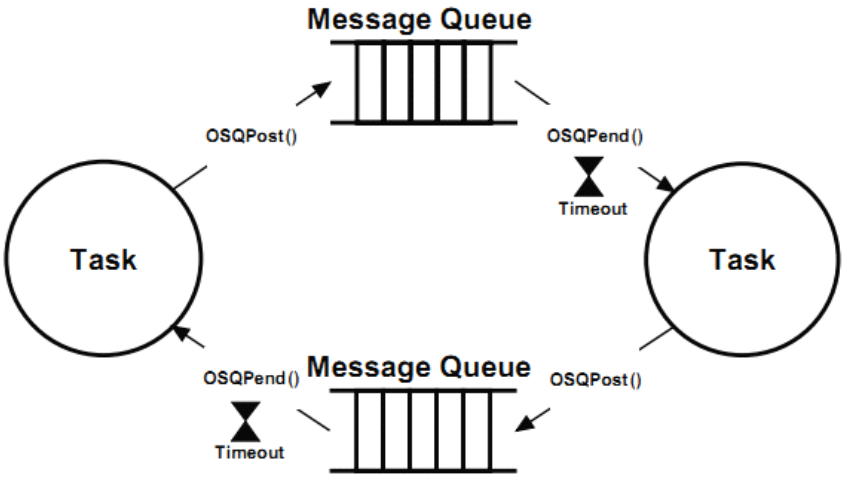


Figure 15-4 **Bilateral Rendezvous**

这两个消息队列都被初始化为空。当左边的任务到达约定点时，它就会发送消息给顶部消息队列并在底部消息队列中等待消息。类似的，当右边的任务到达某个时刻，它就会发送消息给底部消息队列，并在顶部消息队列中等待消息。

图 15-5 显示了如何用实现双向通信

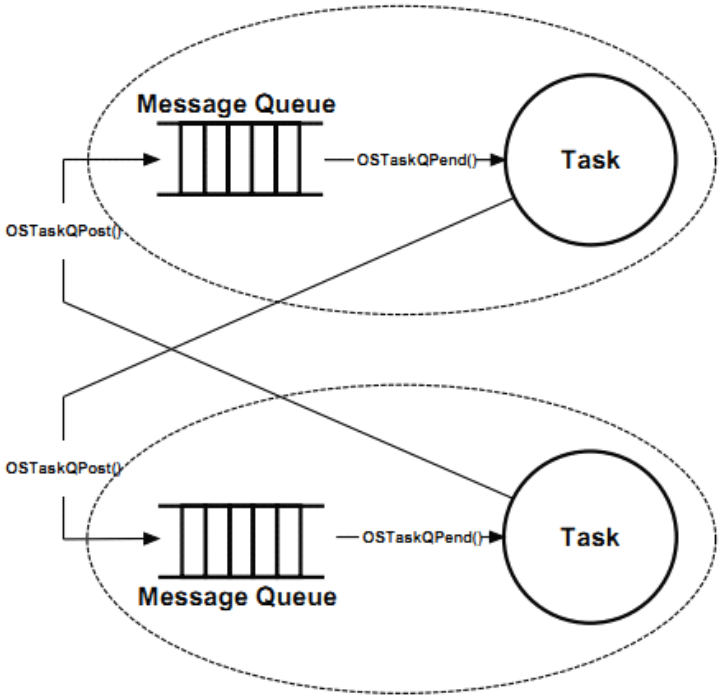


Figure 15-5 **Figure Bilateral Rendezvous with task message queues**

15-5 流量控制

任务间的通信经常通过从任务 A 中传送数据给任务 B。然而，任务 A 创建该数据需要时间，且任务 B 可能在该数据被创建后经过很长时间才接收到该数据。换句话说，如果更高优先级的任务抢占了任务 B，那么任务 A 所存放在消息队列中的数据就可能被溢出。解决这个问题的一种方法是在处理中添加流量控制

如图 15-6 所示

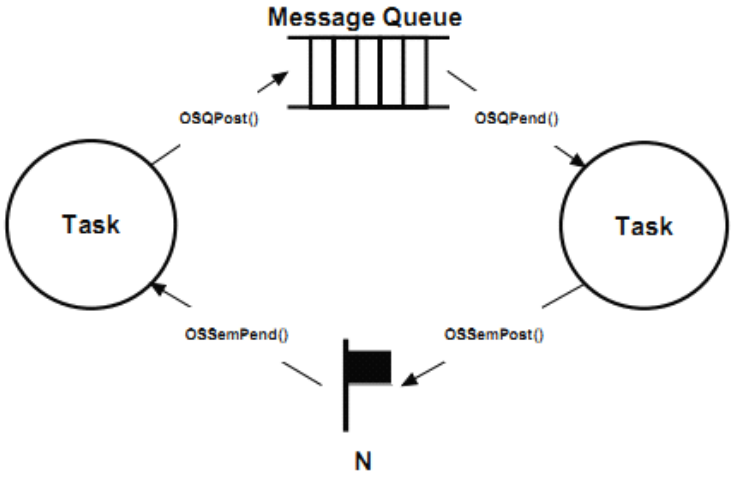


Figure 15-6 **Producer and consumer tasks with flow control**

在这里，使用了一个信号量计数值，该值初始化为任务 B 中消息队列长度。

如列表 15-1 所示，任务 A 在发送消息给任务 B 之前必须获得信号量。任务 B 消息队列的空余量为多少，信号量计数值就为多少。

```
Producer Task:
Pend on Semaphore;
Send message to message queue;

Consumer Task:
Wait for message from message queue;
Signal the semaphore;
```

Listing 15-1 **Producer and consumer flow control**

将任务的消息队列和信号量队列一起使用（见第 14 章“同步”），实现流量控制将是简单的，如图 15-7 所示。在这种情况下，需要调用 OSTaskSemSet()函数设置任务 B 的信号量计数值为任务 B 消息队列的消息容量。

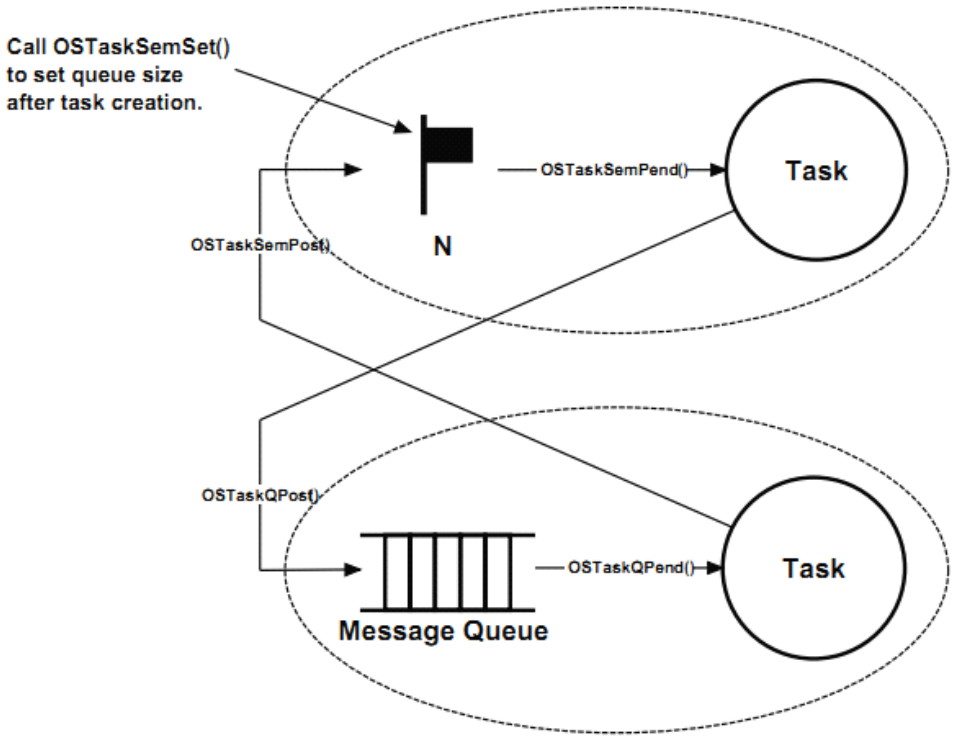


Figure 15-7 **Flow control with task semaphore and task message queue**

15-6 保持数据在作用域中

消息通常指向结构体、变量、数组等。然而，数据必须被保持在其作用域（结构体、变量、数组）中直到接收者完成对这些数据的操作。消息一旦被发送，发送者就不能访问被发送的消息中所包含的数据。这看起来很显然，但经常被忘记。

通过 uC/OS-III 中的分区管理器（见第 17 章“内存管理”）动态地分配和回收内存块用于传递这些数据（消息中所包含的数据）。图 15-8 显示了一个例子，假定一个器件通过某种协议发送字节数据包给 UART。在这种情况下，数据包的开始字节和结束字节应该独一无二的。

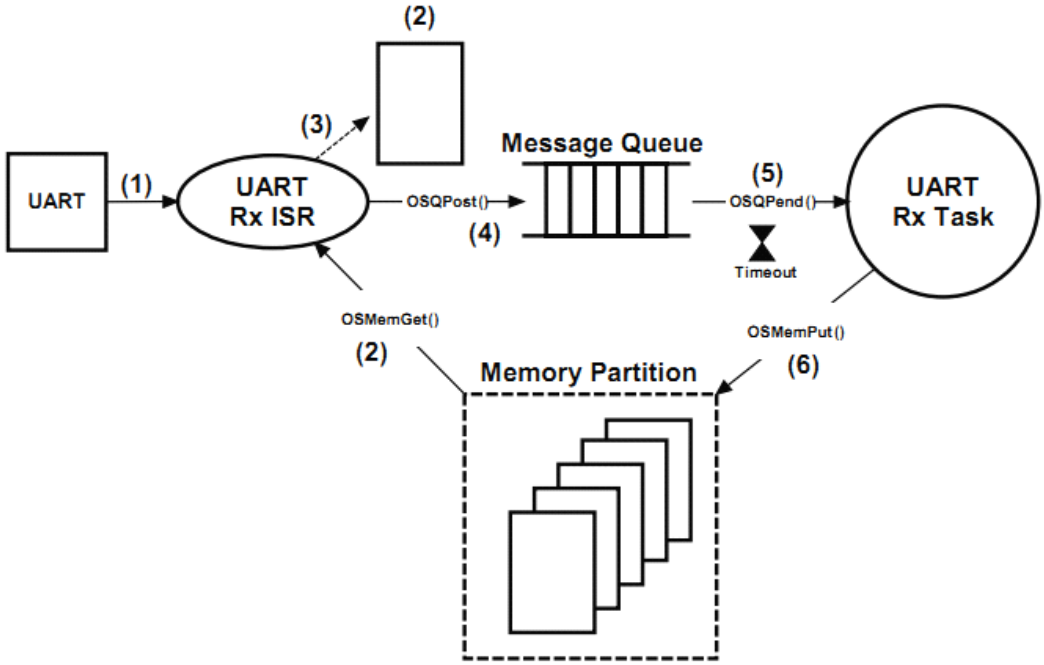


Figure 15-8 **Using memory partitions for message contents**

F15-8（1）这里，UART 接收到开始字节并中断。

F15-8（2）列表 15-2 中显示了 UART 的 ISR 代码。ISR 从 UART 中读取接收到的数据并观测它是否为数据包的开始字节。如果是，就获得一个内存块。

F15-8（3）接收到的数据被放入内存块中。

F15-8（4）如果检测到包的结束字节，就提交这个内存块的地址给消息队列，以便让任务处理这个接收包。

F15-8（5）如果这个消息使 UART 任务就绪，且其为最高优先级任务，uC/OS-III 在执行完 ISR 之后就会跳转到该任务。该任务从消息队列中获得数据包。注意的是：OSQPend()被调用后返回数据包的字节数和时间戳（存了该消息被发送时的时间戳）。

F15-8（6）任务处理完这个数据包后，通过调用 OSMemPut()将这个内存块释放。

```
void  UART_ISR (void)
{
    OS_ERR  err;


    RxData = Read byte from UART;
    if (RxData == Start of Packet) {              /* See if we need a new buffer      */
        RxDataPtr = OSMemGet(&UART_MemPool,       /* Yes                              */
                             &err);
        RxDataCtr = 0;
    } else {
        RxDataCtr++;                              /* Update number of bytes received  */
    }
    if (RxData == End of Packet byte) {           /* See if we got a full packet      */
        OSQPost((OS_Q      *)&UART_Q,             /* Yes, post to task for processing */
                (void      *)RxDataPtr,
                (OS_MSG_SIZE)RxDataCtr,
                (OS_OPT     )OS_OPT_POST_FIFO,
                (OS_ERR    *)&err);
        RxDataPtr = NULL;                          /* Don't point to sent buffer       */
        RxDataCtr = 0;
    } else; {
       *RxDataPtr++ = RxData;                     /* Save the byte received           */
   }
}
```

Listing 15-2 **UART ISR Pseudo-code**

15-7 使用消息队列

表 15-1 是关于消息队列函数的总结。详见附录 A

函数名	功能
OSQCreate()	创建一个消息队列
OSQDel()	删除一个消息队列
OSQFlush()	清空一个消息队列
OSQPend()	任务等待消息
OSQPendAbort()	任务被不再等待该消息
OSQPost()	提交一个消息给消息队列

表 15-1　消息队列的 API 总结

表 15-2 是任务中消息队列函数的总结，详见附录 A。

函数名	功能
OSTaskQPend()	等待一个消息
OSTaskQPendAbort()	任务被不再等待该消息
OSTaskQPost()	发送一个消息给任务
OSTaskQFlush()	清空这个消息队列

表 15-2 任务消息队列的 API 总结

图 15-9 显示了通过消息队列配置旋转轮的旋转速率。

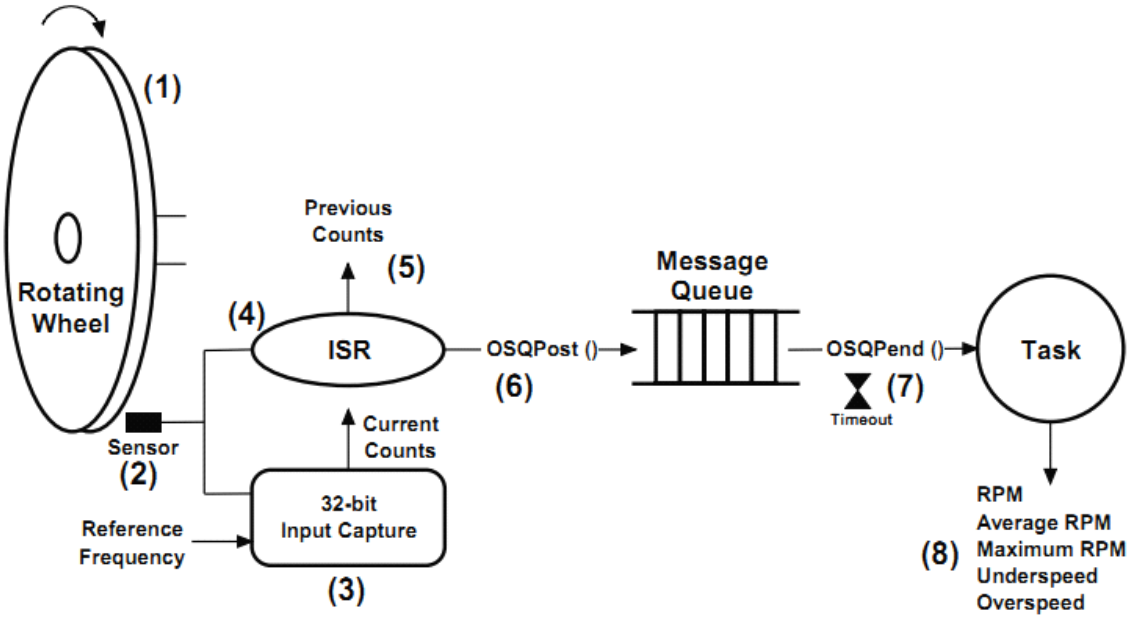


Figure 15-9 **Measuring RPM**

F15-9（1）本例的目标是测量旋转轮的旋转速率。

F15-9（2）传感器被用于测量该旋转轮的速率。轮的周围有很多小孔，这是个光传感器。

F15-9（3）32 位的输入捕获寄存器用于捕获每次中断时的 CPU 计数值（也就是时间戳）。

F15-9（4）检测到一个孔时中断产生。ISR 读取当前的捕获寄存器值并减去先前的捕获值，就能知道旋转的速率。

```
Delta Counts    = Current Counts – Previous Counts;
Previous Counts = Current Counts;
```

F15-9（5）Delta 值被发送到消息队列。因为消息实际上是一个指针，如果这个指针是 32 位的，直接将 Delta 值存到这个指针中被发送到消息队列。但更安全的方法是分配一个内存块并将 Delta 值存入内存块并将内存块地址存入消息。

F15-9（6）当消息被发送后，测速任务被唤醒，并计算每分钟的旋转速率：

```
RPM = 60 * Reference Frequency / Delta Counts;
```

用户可以给测速任务定义期限并且任务可能因为超时而被唤醒。这可能就表明旋转轮的速率为 0.

F15-9（7）该任务也可以计算平均旋转速率，最大旋转速率等待。

在这个了例子中有一些有趣的事情。第一，ISR 非常短，读取捕获值并将 Delta 值发送给任务。第二，若测速任务挂起超时，就可以知道旋转轮被停止了。最后，测速任务中还可以添加一些操作，比如测量旋转轮是否转得过快或过慢，并可以告知旋转结果给其它任务。

列表 15-3 显示了如何用消息队列测量旋转速率。

```
OS_Q        RPM_Q;                                                  (1)
CPU_INT32U  DeltaCounts;
CPU_INT32U  CurrentCounts;
CPU_INT32U  PreviousCounts;


void main (void)
{
    OS_ERR  err ;
    :
    OSInit(&err) ;                                                  (2)
    :
    OSQCreate((OS_Q     *)&RPM_Q,
              (CPU_CHAR *)"My Queue",
              (OS_MSG_QTY)10,
              (OS_ERR   *)&err);
    :
    OSStart(&err);
}


void RPM_ISR (void)                                                 (3)
{
    OS_ERR  err;


    Clear the interrupt from the sensor;
    CurrentCounts  = Read the input capture;
    DeltaCounts    = CurrentCounts - PreviousCounts;
    PreviousCounts = CurrentCounts;
    OSQPost((OS_Q      *)&RPM_Q,                                    (4)
            (void      *)DeltaCounts,
            (OS_MSG_SIZE)sizeof(void *),
            (OS_OPT     )OS_OPT_POST_FIFO,
            (OS_ERR    *)&err);
}
```

```
void RPM_Task (void *p_arg)
{
    CPU_INT32U   delta;
    OS_ERR       err;
    OS_MSG_SIZE  size;
    CPU_TS       ts;


    DeltaCounts    = 0;
    PreviousCounts = 0;
    CurrentCounts  = 0;
    while (DEF_ON) {
        delta = (CPU_INT32U)OSQPend((OS_Q        *)&RPM_Q,          (5)
                                    (OS_TICK      )OS_CFG_TICK_RATE * 10,
                                    (OS_OPT       )OS_OPT_PEND_BLOCKING,
                                    (OS_MSG_SIZE *)&size,
                                    (CPU_TS      *)&ts,
                                    (OS_ERR      *)&err);
        if (err == OS_ERR_TIMEOUT) {                                (6)
            RPM = 0;
        } else {
            if (delta > 0u) {
                RPM = 60 * Reference Frequency / delta;             (7)
            }
        }
        Compute average RPM;                                        (8)
        Detect maximum RPM;
        Check for overspeed;
        Check for underspeed;
        :
        :
    }
}
```

Listing 15-3 **Pseudo-code of RPM measurement**

L15-3（1）消息队列被定义。

L15-3（2）调用 OSInit()后，创建消息队列。

L15-3（3）ISR 清除传感器中断并读取捕获寄存器的值。在 ISR 中计算 Delta 计数值。也可以直接发送捕获值，并在测速任务中计算 Delta 值。然而，这个计算是很快的，它所增加的 ISR 时间可忽略不计。

L15-3（4）将 Delta 值发送到测量任务，并执行相应计算。注意的是：当消息队列满的时侯再往里提交就会导致溢出。消息被快速提交时就会出现这种情况。很不幸，不能通过流量控制解决这个问题因为消息是在 ISR 中被提交的。

L15-3（5）测量任务等待测量 ISR 中的消息。第三个参数设置了等待期限。在这里设置了等待期限为 10 秒。参数 ts 包含了消息被提交时的时间戳。通过 OS_TS_GET()获得当前的时间戳。该消息的相应时间就能计算：

```
response_time = OS_TS_GET() – ts;
```

L15-3（6）等待超时（假定如旋转轮停止旋转）。

L15-3（7）计算旋转速率。

L15-3（8）其它的一些操作。事实上，在检测到错误的情况下消息可以被发送给其它任务。例如，当旋转速率很快，其它任务就可以让旋转轮减速。

如列表 L15-4，在这个例子中，用 OSTaskQPost()和 OSTaskQPend()代替 OSQPost()和 OSQPend()。代码将会更简单且不需要创建一个外部消息队列。然而，当创建测量任务时，就必须设置其消息队列大小并在编译时配置 OS_CFG_TASK_Q_EN 为 1。使用外部消息队列和任务内部消息队列的区别如下介绍。

```
OS_TCB      RPM_TCB;                                                    (1)
OS_STK      RPM_Stk[1000];
CPU_INT32U  DeltaCounts ;
CPU_INT32U  CurrentCounts ;
CPU_INT32U  PreviousCounts ;


void main (void)
{
    OS_ERR  err ;
    :
    OSInit(&err) ;
    :
    void  OSTaskCreate ((OS_TCB       *)&RPM_TCB,                       (2)
                        (CPU_CHAR     *)"RPM Task",
                        (OS_TASK_PTR   )RPM_Task,
                        (void         *)0,
                        (OS_PRIO       )10,
                        (CPU_STK      *)&RPM_Stk[0],
                        (CPU_STK_SIZE  )100,
                        (CPU_STK_SIZE  )1000,
                        (OS_MSG_QTY    )10,
                        (OS_TICK       )0,
                        (void         *)0,
                        (OS_OPT        )(OS_OPT_TASK_STK_CHK + OS_OPT_TASK_STK_CLR),
                        (OS_ERR       *)&err);
    :
    OSStart(&err);
}
```

```
void RPM_ISR (void)
{
    OS_ERR  err;

    Clear the interrupting from the sensor;
    CurrentCounts  = Read the input capture;
    DeltaCounts    = CurrentCounts – PreviousCounts;
    PreviousCounts = CurrentCounts;
    OSTaskQPost((OS_TCB    *)&RPM_TCB,                                    (3)
                (void      *)DeltaCounts,
                (OS_MSG_SIZE)sizeof(DeltaCounts),
                (OS_OPT     )OS_OPT_POST_FIFO,
                (OS_ERR    *)&err);
}

void RPM_Task (void *p_arg)
{
    CPU_INT32U   delta;
    OS_ERR       err;
    OS_MSG_SIZE  size;
    CPU_TS       ts;


    DeltaCounts    = 0;
    PreviousCounts = 0;
    CurrentCounts  = 0;
    while (DEF_ON) {
        delta = (CPU_INT32U)OSTaskQPend((OS_TICK       )OS_CFG_TICK_RATE * 10,   (4)
                                        (OS_OPT        )OS_OPT_PEND_BLOCKING,
                                        (OS_MSG_SIZE *)&size,
                                        (CPU_TS      *)&ts,
                                        (OS_ERR      *)&err);
        if (err == OS_ERR_TIMEOUT) {
            RPM = 0;
        } else {
            if (delta > 0u) {
                RPM = 60 * ReferenceFrequency / delta;
            }
        }
        Compute average RPM;
        Detect maximum RPM;
        Check for overspeed;
        Check for underspeed;
        :
        :
    }
}
```

Listing 15-4 **Pseudo-code of RPM measurement**

L15-4（1）任务的 OS_TCB 也可以接收消息。

L15-4（2）创建测量任务并设置其消息队列的大小为 10。当然，在实时系统中最好不要将该值固定。但在这里为了说明，将该值固定为 10。

L15-4（3）ISR 直接将消息发送给任务。

L15-4（4）测速任务通过调用 OSTaskQPend()等待 ISR 发送的消息。这是一个内部的调用，所以没必要设置 OS_TCB 的地址。第二个参数设置等待期限。

15-8 客户端和服务器端

消息队列另一种用法如图 15-10 所示。错误管理任务管理其它任务或 ISR 发给它的错误情况。例如，测速任务测得转盘旋转速率过快，它发送消息给错误管理任务。错误管理任务作相应的操作。

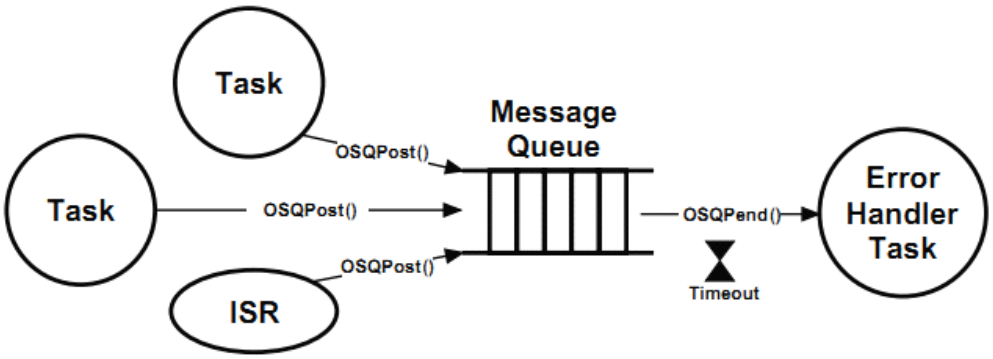


Figure 15-10 **Clients and Servers**

15-9 消息队列的组成

消息由四个变量组成：指向下一条消息的指针、用于表明该消息所指向数据的大小的变量、存放消息最后一次被提交的时间戳的变量、消息中包含一个指向实际数据的指针。如图 15-11 所示

消息发送者和接收者都不知道消息中数据的结构因为这些都通过 uC/OS-III 的 API 隐藏起来了。

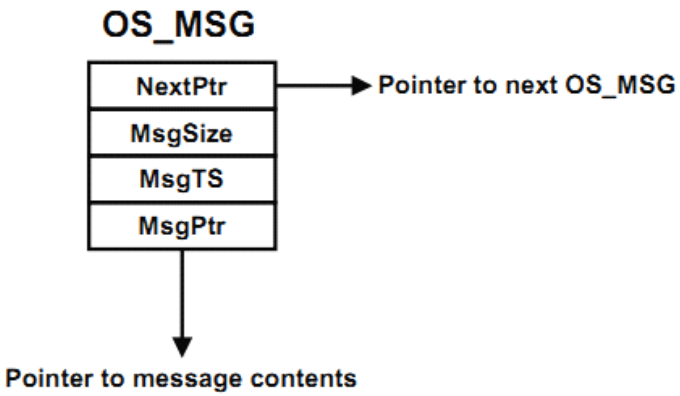


Figure 15-11 **OS_MSG structure**

uC/OS-III 维护一个消息池。消息池的大小通过 OS_CFG_APP.H 中的 OS_CFG_MSG_POOL_SIZE 设置。当 uC/OS-III 被初始化时，消息就会以单向列表的形式链接起来如图 15-12。注意的是这个列表由结构体 OS_MSG_POOL 管理，它包含 3 个部分：.NextPtr 指向该消息列表、.NbrFree 包含了该队列的空闲消息数、.NbrUsed 包含了该队列中已被使用的消息数。

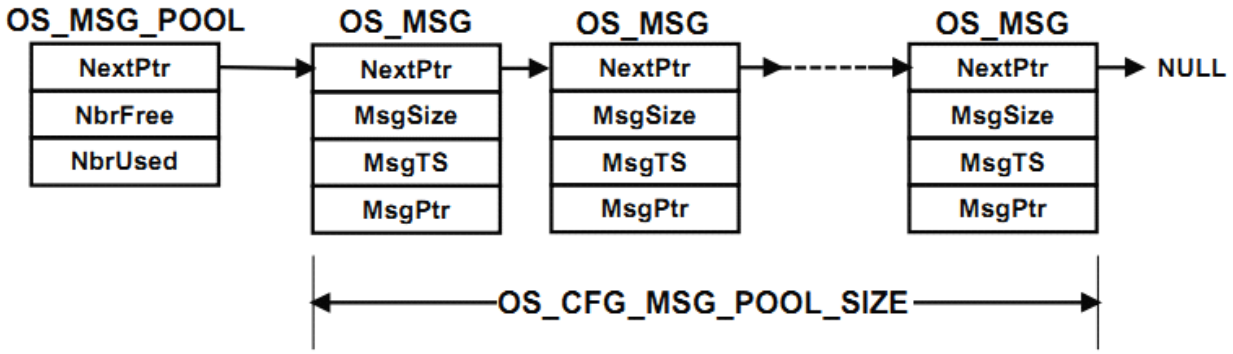


Figure 15-12 **Pool of free OS_MSGs**

消息的排列由结构体 OS_MSG_Q 控制，如图 15-13

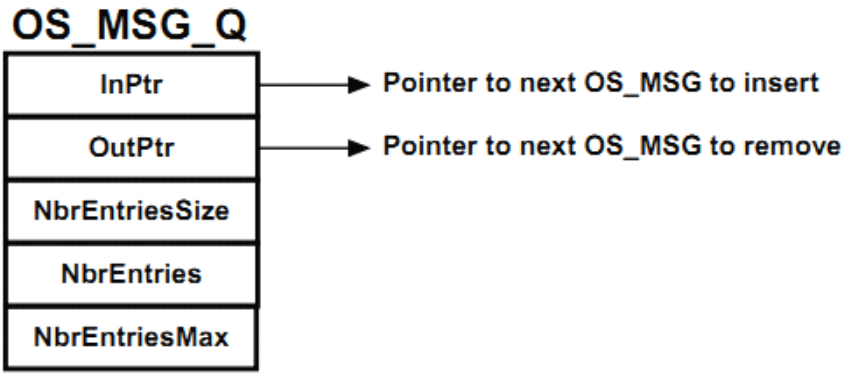


Figure 15-13 **OS_MSG_Q structure**

.InPtr　　指向下一个将要被插入到队列的消息。

.OutPtr　　指向下一个将要被释放的消息。

.NbrEntriesSize　包含了该队列所能接受的最大消息数。队列满后再往其中发送消息，消息将不会被插入。

.NbrEntries　　当前队列中的消息数

.NbrEntriesMax　记录了到目前为止队列中存放的最大消息数。

uC/OS-III 中有一些函数用于操纵空闲的队列和消息。比如：OS_MsgQPut()将消息插入到 OS_MSG_Q，OS_MsgQGet()从 OS_MSG_Q 中得到一个消息，OS_MsgQFreeAll()将所有 OS_MSQ_Q 中的消息释放回消息池中。OS_MSG.C 中的其它一些 OS_MsgQ??() 在初始化时使用。

图 15-14 显示了有 4 个消息的 OS_MSG_Q。

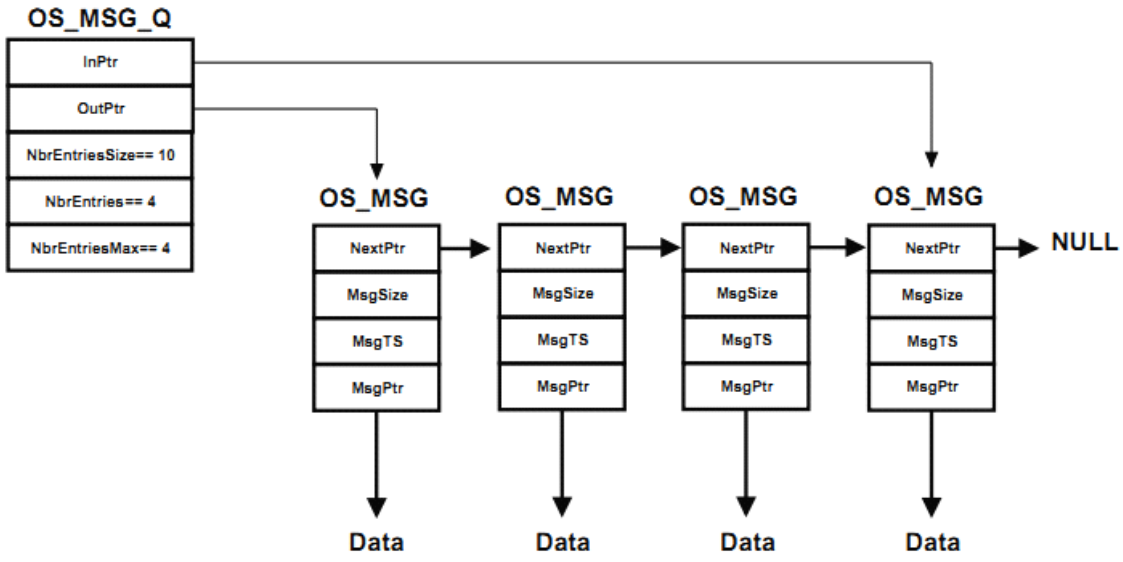


Figure 15-14 **OS_MSG_Q with four OS_MSGs**

OS_MSG_Q 通常包含在两种结构体内：OS_Q 和 OS_TCB。创建一个 OS_Q 时就内建一个 OS_MSG_Q。当设置 OS_CFG.H 中的 OS_CFG_TASK_Q_EN 为 1 时，每个任务都有其内建的消息队列。如图 15-15 所示。

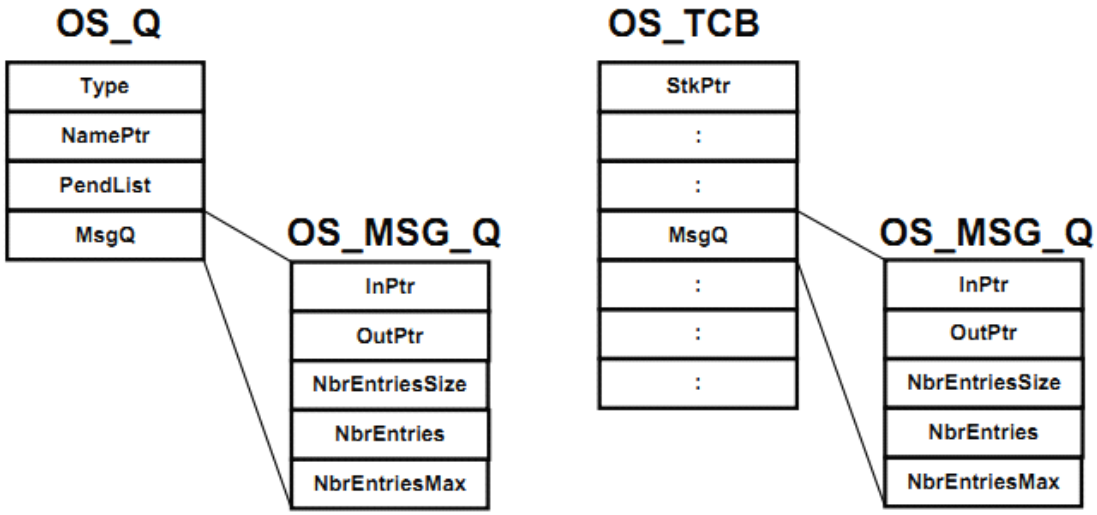


Figure 15-15 **OS_Q and OS_TCB each contain an OS_MSG_Q**

15-10 总结

当任务或 ISR 发送消息给另一个任务时，通过消息队列是很有效的方法。被发送的消息中的数据必须被存储，因为传送的是数据地址而不是实际的数据。

任务在等待消息的时候不会占用 CPU。

使用任务内部的消息堆栈更为简单和高效。当设置 OS_CFG.H 中的 OS_CFG_TASK_Q_EN 为 1 时使能任务内部消息队列。

如果多个任务等待同一个消息队列中的消息，分配一个 OS_Q 并让任务所等待的消息发送到 OS_Q 中。特别的，可以广播消息给所有在消息队列中等待的任务。设置 OS_CFG.H 中的 OS_CFG_Q_EN 为 1 开启消息队列服务。

消息通过结构体 OS_MSG（从 uC/OS-III 的消息池中获得）被发送。设置消息队列的大小，和消息池的大小。

16、挂起多个对象

在章节 10“挂起队列”中介绍了多个任务如何等待一个内核对象如信号量、mutex、事件标志组、消息队列。在这个章节中，将介绍任务等待多个对象。然而，uC/OS-III 只允许同时等待多个信号量和消息队列。换句话说，不能同时等待多个事件标志组或 mutex。

如图 16-1 所示，任务可以同时等待多个信号量和消息队列。任务接收到一个信号量或消息就会被就绪。任务通过调用 OSPendMulti() 等待多个对象，并可使设置等待时限。这个时限对应于所有的对象。当在这个时限内没有收到一个对象，任务就会返回一个错误代号表示等待超时。

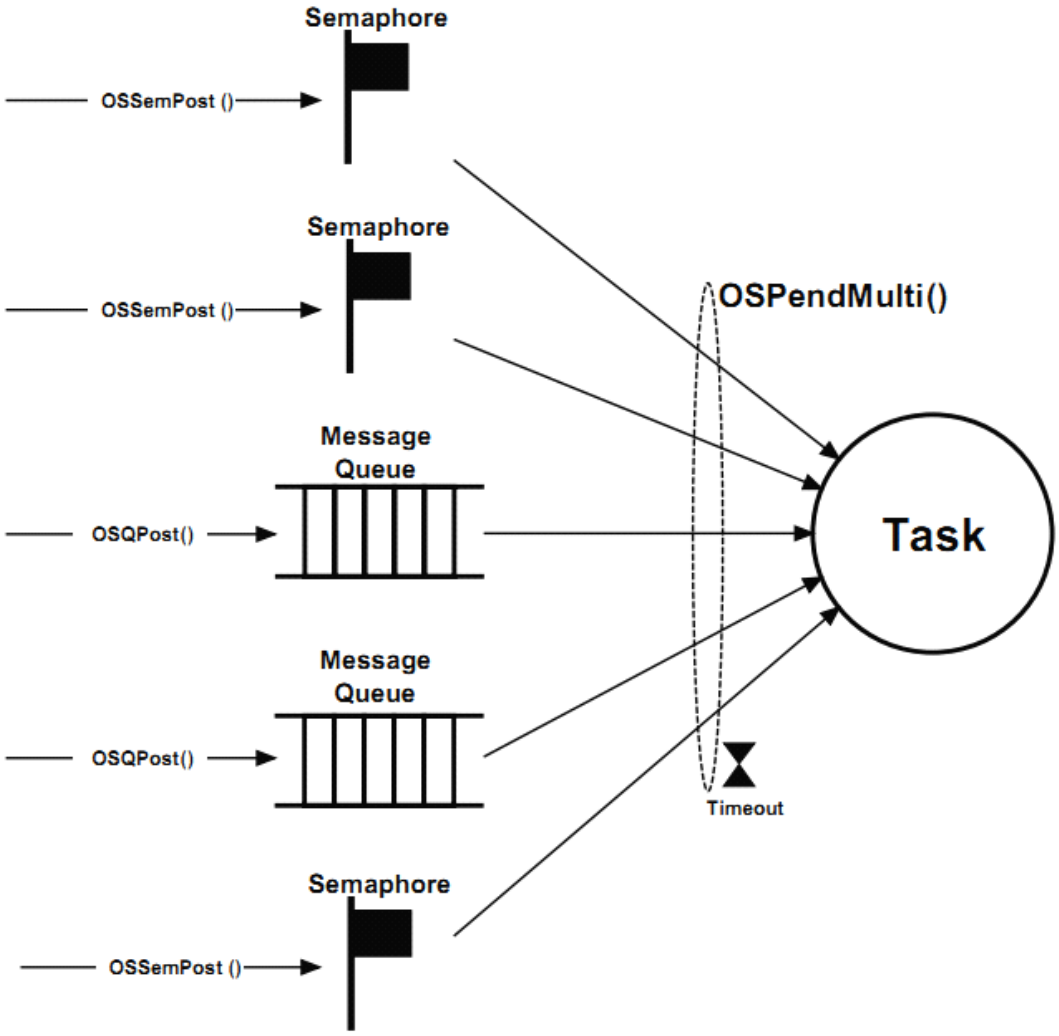


Figure 16-1 **Task pending on multiple objects**

列表 16-1 显示了 OSPendMulti()的原型，图 16-2 显示了一个数据类型为 OS_PEND_DATA 的数组。

```
OS_OBJ_QTY  OSPendMulti (OS_PEND_DATA  *p_pend_data_tbl,                (1)
                         OS_OBJ_QTY     tbl_size,                       (2)
                         OS_TICK        timeout,                        (3)
                         OS_OPT         opt,                            (4)
                         OS_ERR        *p_err);                         (5)
```

Listing 16-1 **OSPendMulti() prototype**

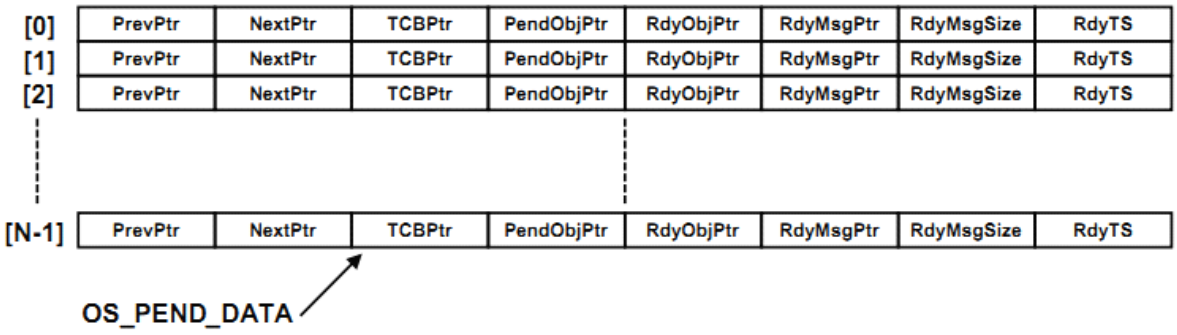


Figure 16-2 **Array of OS_PEND_DATA**

L16-1（1）OSPendMulti 的第一个参数：数据类型为 OS_PEND_DATA 的数组。数组的大小决定了任务所等待的内核对象数。例如，如果任务要等待 3 个信号量和 2 个消息，那么数组的大小为 5。

```
OS_PEND_DATA   my_pend_multi_tbl[5];
```

每个数组元素中的.PendObjPtr 都要被初始化、指向所等待的对象。例如：

```
OS_SEM  MySem1;
OS_SEM  MySem2;
OS_SEM  MySem3;
OS_Q    MyQ1;
OS_Q    MyQ2;


void  MyTask (void)
{
    OS_ERR        err;
    OS_PEND_DATA  my_pend_multi_tbl[5];
    :
    while (DEF_ON) {
        :
        my_pend_multi_tbl[0].PendObjPtr = (OS_PEND_OBJ)&MySem1;     (6)
        my_pend_multi_tbl[1].PendObjPtr = (OS_PEND_OBJ)&MySem2;
        my_pend_multi_tbl[2].PendObjPtr = (OS_PEND_OBJ)&MySem3;
        my_pend_multi_tbl[3].PendObjPtr = (OS_PEND_OBJ)&MyQ1;
        my_pend_multi_tbl[4].PendObjPtr = (OS_PEND_OBJ)&MyQ2;
        OSPendMulti((OS_PEND_DATA *)&my_pend_multi_tbl[0],
                    (OS_OBJ_QTY    )5,
                    (OS_TICK       )0,
                    (OS_OPT        )OS_OPT_PEND_BLOCKING,
                    (OS_ERR       *)&err);
        /* Check 'err" */
        :
    }
}
```

L16-1（2）第二个参数为数组的大小，这个例子中为 5。

L16-1（3）设置等待期限，如果在这段时间内没有一个对象被提交就会超时。设置 0 值表示永远等待下去。

L16-1（4）参数"opt"设置了等待的方式。OS_OPT_PEND_BLOCKING 和 OS_OPT_PEND_NON_BLOCKING（任务不被挂起）。

F16-2（1）如大多数 uC/OS-III 函数一样，函数会返回代表此函数执行结果的错误代号。

F16-2（2）所有的对象的类型都是 OS_PEND_OBJ。

当 OSPendMulti()被调用，它首先确认数组中所有的元素是否为 OS_SEM 或 OS_Q。如果不是，就返回对应的错误代号。

OSPendMulti()先遍历数组，查看其中的对象是否已经被提交。如果有，OSPendMulti()就在表中该索引中填入相应的值：RdyObjPtr，RdyMsgPtr，.RdyMsgSize，.RdyTs。

.RdyObjPtr　指向已经被提及的对象。例如，表中索引 0 的信号量已经被提交，那么 my_pend_multi_tbl[0].RdyObjPtr 的值会被设置为 my_pend_multi_tbl[0].PendObjPtr。

.RdyMsgPtr　如果表中该索引等待的是消息队列，且有消息被接收，那么该指针指向消息。

.RdyMsgSize　如果表中该索引等待的是消息队列，且有消息被接收，那么该值为消息中数据的大小。

.RdyTS　存放着对象被提交时的时间戳。

如果没有对象被提交，OSPendMulti()就会将任务放入所有对象的挂起队列中，这是一个复杂的操作因为其它任务也可能在这些对象中等待。图 16-3 是任务在两个信号量中挂起的例子：

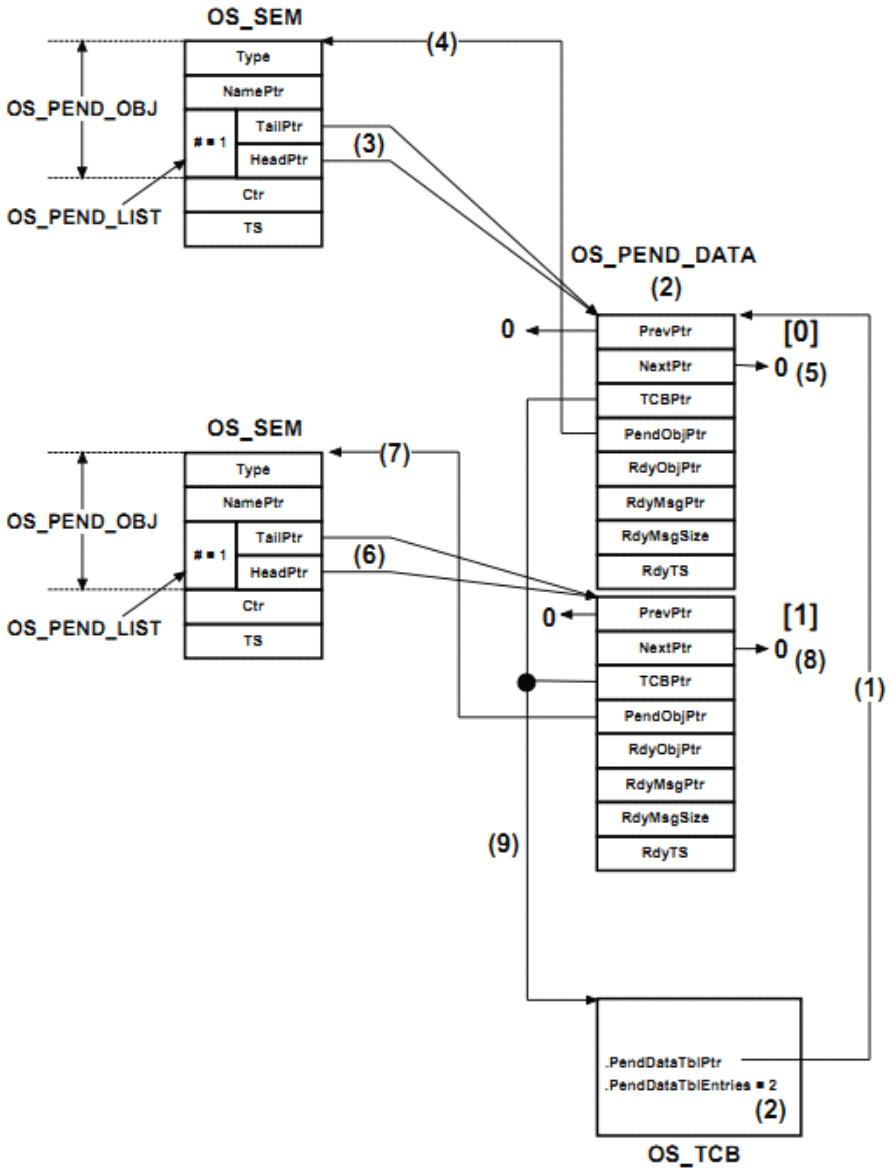


Figure 16-3 **Task pending on two semaphores**

F16-3（0）任务的 OS_TCB 中有指针指向由 OS_PEND_DATA 组成的表（如上面所述）。

F16-3（2）OS_TCB 中存放了 OS_PEND_DATA 所组成的表中的 OS_PEND_DATA 的数量。表中有两个记录。

F16-3（3）第一个信号量指向第一个 OS_PEND_DATA。

F16-3（4）OS_PEND_DATA 表中记录 0 的.PendObjPtr 指向第一个信号量。通过调用 OSPendMulti()等待对象并设定指针指向。

F16-3（5）因为信号量中只有 1 个任务在等待，所以.PrevPtr 和.NextPtr 都指向 NULL。

F16-3（6）第二个信号量指向第二个 OS_PEND_DATA。

F16-3（7）OS_PEND_DATA 表中记录 0 的.PendObjPtr 指向第二个信号量。通过调用 OSPendMulti()等待对象并设定指针指向。

F16-3（8）因为信号量中只有 1 个任务在等待，所以.PrevPtr 和.NextPtr 都指向 NULL。

F16-3（9）任务中有指针链接到两个 OS_PEND_DATA 结构体。

图 16-4 是一个更加复杂的例子：一个任务等待两个信号量，另一个任务等待其中的一个信号量。例子只是呈现了信号量，也可以扩展到消息队列及两者的组合。

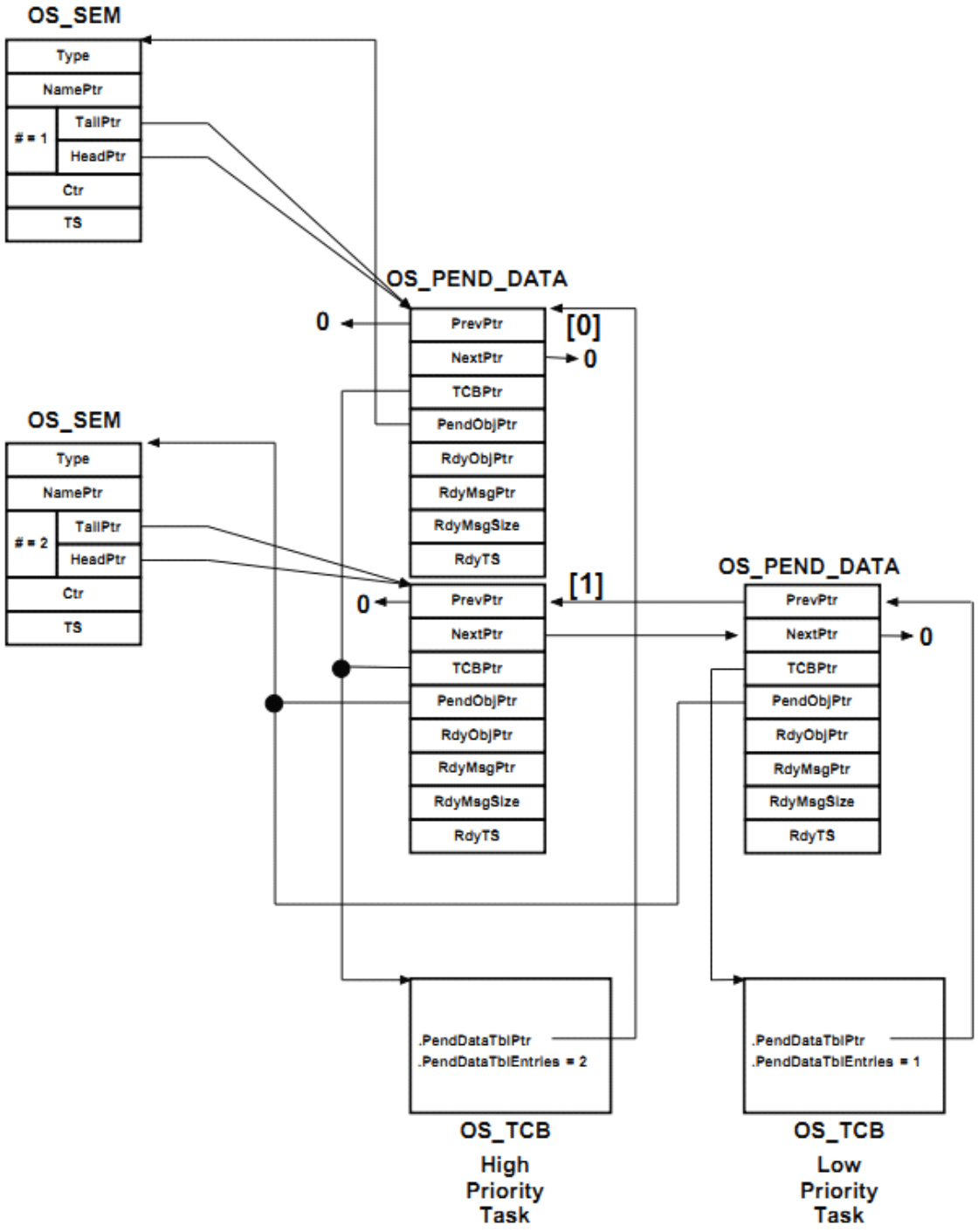


Figure 16-4 **Tasks pending on semaphores**

当任务或 ISR 发送消息给等待该消息的任务，OSPendMulti()返回。表示 OS_PEND_DATA 表中有对象被提交，这些通过在表中相应索引中填入对应值实现，如图 16-2。

当任务等待 5 个内核对象时，且在任务调用 OSPendMulti()之前已经有一个对象就绪，那么表中结构将会是图 16-5

	.PrevPtr	.NextPtr	.TCBPtr	.PendObjPtr	.RdyObjPtr	.RdyMsgPtr	.RdyMsgSize	.RdyTS
[0]	0	0	TCBPtr	PendObjPtr	0	0	0	0
[1]	0	0	TCBPtr	PendObjPtr	0	0	0	0
[2]	0	0	TCBPtr	PendObjPtr	0	0	0	0
[3]	0	0	TCBPtr	&MyQ1	&MyQ1	Msg Ptr	Msg Size	Timestamp
[4]	0	0	TCBPtr	PendObjPtr	0	0	0	0

Figure 16-5 **Message queue #1 posted before timeout expired**

16-1 总结

uC/OS-III 允许任务等待多个内核对象。

OSPendMulti()只能挂起多个信号量和消息队列。不能挂起多个事件标志组和 mutex。

在调用 OSPendMulti()之前对象已经被提交，uC/OS-III 会“告诉”OSPendMulti()那些对象已经被提交。

如果没有已提交的对象，OSPendMulti()就会将任务放入所有对象的挂起队列中。当有一个对象被提交时 OSPendMulti()就会返回。这这种情况下，OSPendMulti()会标记哪个对象已被提交。

OSPendMulti()是一个复杂的函数，可能会导致长临界段。

17、内存管理

可以通过使用编译器提供的函数 malloc()和 free()动态地分配和释放内存快。然而，在嵌入式实时系统中使用 malloc()和 free()可能是非常危险的。最后，它可能会导致很多内存碎片。

uC/OS-III 可以获得连续的内存块，如图 17-1 所示。内存块大小可以相同，所有的内存分区包含了整数个内存块。在特定的时间执行内存块的分配和释放。内存分区以内存块数组的形式被静态分配的。如果分配后不被释放，也可以调用 malloc()动态分配。

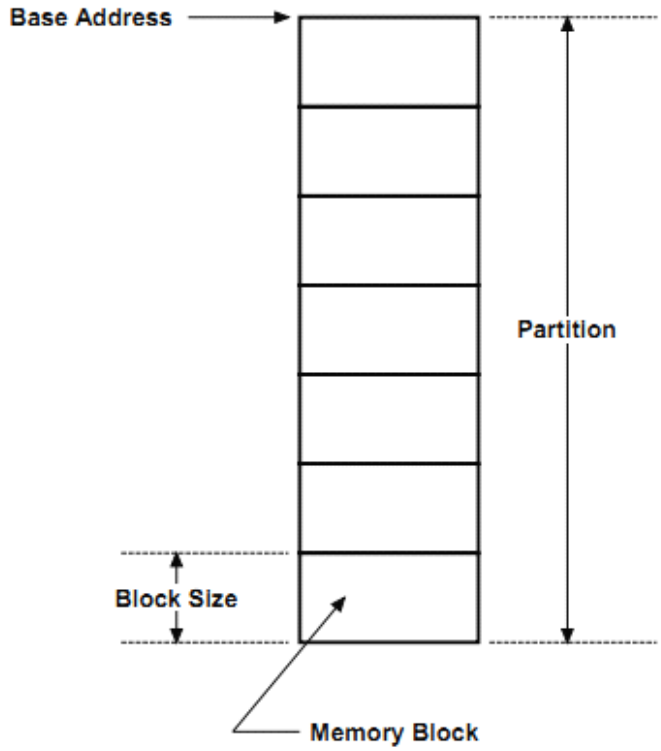


Figure 17-1 **Memory Partition**

如图 17-2 所示，可以同时有多个内存分区且每个分区的内存块个数和大小可以不同，根据需求设置，但内存块被分配后必须返回给它所在的内存分区。这种管理方式仅会导致内存块块内的碎片，决定于应用中所设置内存块的大小。

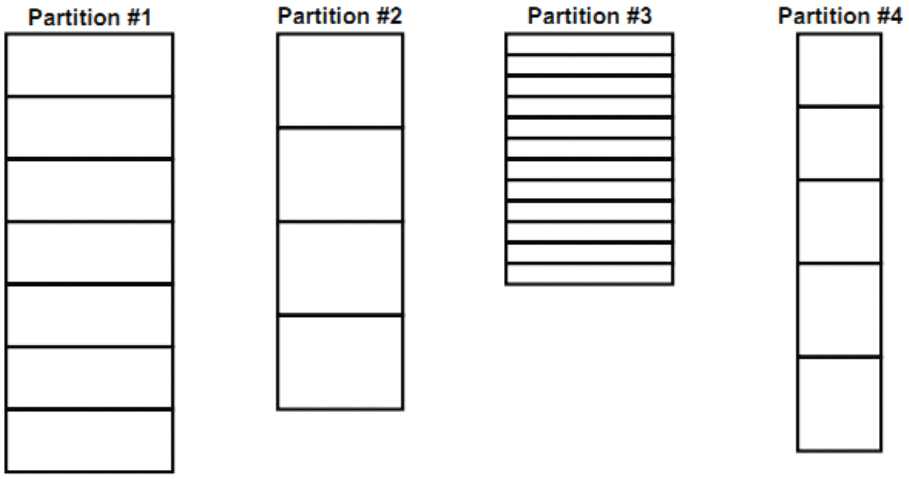


Figure 17-2 **Multiple Memory Partitions**

17-1 创建一个内存分区

在使用内存分区之前必须创建它。这样 uC/OS-III 就可以知道这个内存分区的相关信息并管理他们。调用 OSMemCreate()创建一个内存分区。如图 17-3 所示

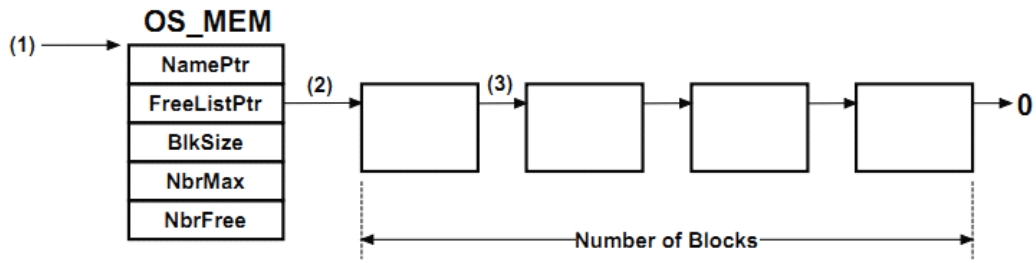


Figure 17-3 **Created Memory Partition**

F13-3（1）当创建一个内存分区时，内存控制块(OS_MEM)的地址需作为参数被提供。通常内存控制块从静态内存中分配，然而，也可以调用 malloc()从堆中获得。用户代码不能释放内存控制块。

F13-3（2）OSMemCreate()将内存块链接起来并将其链表的首地址赋值给 OS_MEM 结构体。

F13-3（3）每个内存块的大小必须大于一个指针。

列表 17-1 中显示了如何用 uC/OS-III 创建一个内存分区

```
OS_MEM       MyPartition;                                    (1)
CPU_INT08U   MyPartitionStorage[12][100];                    (2)


void  main (void)                                            (3)
{
    OS_ERR  err;
    :
    :
    OSInit(&err);
    :
    OSMemCreate((OS_MEM    *)&MyPartition,                   (4)
                (CPU_CHAR  *)"My Partition",                 (5)
                (void      *)&MyPartitionStorage[0][0],      (6)
                (OS_MEM_QTY ) 12,                            (7)
                (OS_MEM_SIZE)100,                            (8)
                (OS_ERR    *)&err);                          (9)
    /* Check 'err" */
    :
    :
    OSStart(&err);
}
```

Listing 17-1 **Creating a memory partition**

L17-1(1)给内存分区分配一个内存控制块。可以静态分配或用 malloc()分配。但是，用户代码不能删除这个内存控制块。

L17-1(2)给内存分区分配多个内存块。这可以静态分配或用 malloc()分配。

L17-1（3）内存分区在被创建之前需被分配内存控制块和内存块。最好在 main()中创建内存分区。当然，也可以在 main()中调用能分配内存分区的函数。

L17-1（4）将内存控制块的地址传递给 OSMemCreate()。用户代码不能访问 OS_MEM 结构体。只能通过 uC/OS-III 的 API 访问。

L17-1（5）给内存分区分配一个名字。

L17-1（6）将内存块数组的首地址传递给 OSMemCreate()。

L17-1（7）该参数为分配给内存分区的内存块数。

L17-1（8）标明该内存分区中每个内存块的大小。用数字值是为了说明，最好用#define 定义的宏代替。

L17-1（9）OSMemCreate()根据函数的执行结果返回一个错误代号。

列表 17-2 介绍了如何使用 malloc()创建一个内存分区。同样的，用户代码不能释放内存控制块和内存块。

```
OS_MEM      *MyPartitionPtr;                                    (1)


void  main (void)
{
    OS_ERR   err;
    void    *p_storage;
    :
    OSInit(&err);
    :
    MyPartitionPtr = (OS_MEM *)malloc(sizeof(OS_MEM));          (2)
    if (MyPartitionPtr != (OS_MEM *)0) {
        p_storage = malloc(12 * 100);                           (3)
        if (p_storage != (void *)0) {
            OSMemCreate((OS_MEM    *)MyPartitionPtr,            (4)
                        (CPU_CHAR  *)"My Partition",
                        (void      *)p_storage,                 (5)
                        (OS_MEM_QTY ) 12,                       (6)
                        (OS_MEM_SIZE)100,                       (6)
                        (OS_ERR    *)&err);
            /* Check 'err" */
        }
    }
    :
    OSStart(&err);
}
```

Listing 17-2 **Creating a memory partition**

L17-2（1）定义一个指针指向 malloc()分配的内存控制块。

L17-2（2）malloc()分配空间给内存控制块 OS_MEM。

L17-2（3）给内存分区分配内存块。

L17-2（4）将 OS_MEM 的地址传递给 OSMemCreate()。

L17-2（5）将内存块的基地址传递给 OSMemCreate()。

L17-2（6）最后，传递内存块的个数和大小给 OSMemCreate()。同样的，用数字分配是为了说明，但最好用#defines 定义的宏常量代替。

17-2 获得内存分区中的内存块

应用代码通过调用 OSMemGet()可以从内存分区中申请内存块。如列表 17-3 所示。（假定内存分区已经被创建）

```
OS_MEM       MyPartition;                                          (1)
CPU_INT08U  *MyDataBlkPtr;


void  MyTask (void *p_arg)
{
    OS_ERR  err;

    :
    while (DEF_ON) {
        :
        MyDataBlkPtr = (CPU_INT08U *)OSMemGet((OS_MEM    *)&MyPartition,   (2)
                                              (OS_ERR    *)&err);
        if (err == OS_ERR_NONE) {                                   (3)
            /* You have a memory block from the partition */
        }
        :
        :
    }
}
```

Listing 17-3 **Obtaining a memory block from a partition**

L17-3（1）所有要访问该内存分区的任务或 ISR 必须可以访问这个内

存分区的内存控制块。（内存控制块在任务或 ISR 的作用域范围内）

L17-3（2）调用 OSMemGet()从内存分区中获得一个内存块。函数返回后该指针将指向被分配内存块的基地址。（类似于 malloc()）

L17-3（3）根据返回的错误代号检测函数的执行结果。

17-3 归还内存块给内存分区

当用户对内存块的使用完毕后，必须将该内存块归还给对应的内存分区。调用 OSMemPut()实现这个功能。如列表 17-4 所示。

```
OS_MEM       MyPartition;                                          (1)
CPU_INT08U  *MyDataBlkPtr;


void  MyTask (void *p_arg)
{
    OS_ERR  err;

    :
    while (DEF_ON) {
        :
        OSMemPut((OS_MEM  *)&MyPartition,                          (2)
                 (void    *)MyDataBlkPtr,                          (3)
                 (OS_ERR  *)&err);
        if (err == OS_ERR_NONE) {                                  (4)
            /* You properly returned the memory block to the partition */
        }
        :
        :
    }
}
```

Listing 17-4 **Returning a memory block to a partition**

L17-4（1）所有要访问该内存分区的任务或 ISR 必须可以访问这个内

存分区的内存控制块。

L17-4（2）调用 OSMemPut()将内存块归还给对应内存分区。注意的是 uC/OS-III 不检测内存块是否被归还给对应内存分区。所以使用时必须非常小心。

L17-4（3）该指针指向要归还的内存块地址。它的类型为"void *"。

L17-4（4）检测错误代号。

17-4 使用内存分区

编译时设置 OS_CFG.H 中的 OS_CFG_MEM_EN 为 1 开启内存管理服务。

表 13-1 是内存管理相关函数的总结。

函数名	功能
OSMemCreate()	创建一个内存分区
OSMemGet()	从内存分区中获得一个内存块
OSMemPut()	归还一个内存块给相应的内存分区

表 17-1 内存分区 API 总结

OSMemCreate()只能在任务级被调用，但是 OSMemGet()和 OSMemPut()可以在 ISR 中被调用。

列表 17-4 显示了如何动态地分配内存。在这个例子中，左边任务读取模拟输入值（可以是压力、温度、电压），并发送消息给右边任

务，消息中包含了根据模拟输入所计算出的相关信息的地址。

在这个例子中错误的处理被集中在右边任务中。其它任务或ISR也可以发送消息给该错误处理任务。

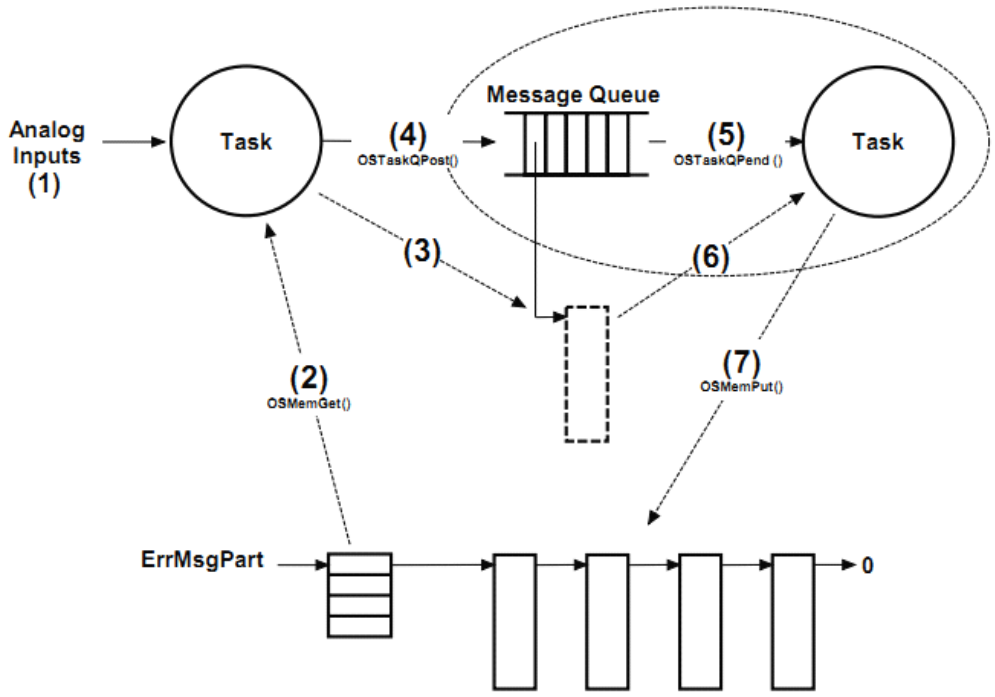


Figure 17-4 **Using a Memory Partition – non blocking**

F17-4（1）任务读取模拟输入。如果测得模拟输入超范围，就发送消息给错误处理函数。

F17-4（2）任务从内存分区中申请一个内存块用于存放检测模拟输入所得的相关数据。

F17-4（3）任务将检测模拟输入所得的相关数据存入内存块。然而，没必要在内存块中存放时间戳因为时间戳被存放在消息中。

F17-4（4）任务提交消息给错误处理任务。当然，错误处理任务知道消息中所包含的数据的含义。一旦消息被发送，发送者任务不能再访问消息所指向的内存块。

F17-4（5）错误处理任务通常在消息队列中等待。

F17-4（6）错误处理任务接收到消息后，执行相应的操作。

F17-4（7）错误处理任务处理完毕相应的操作后，将内存块归还给对应的内存分区。因此，发送者和接收者都必须知道内存块是属于哪个内存分区的，或者发送者可以将内存分区的地址作为要发送消息的数据中的一部分。这样，错误处理函数就知道将这个内存块归还给哪个内存分区了。

任务在内存分区被分配完时可以等待内存块。uC/OS-III 不支持任务等待内存分区，但是可以通过一个信号量用于内存分区中内存块的分配。如图 17-5

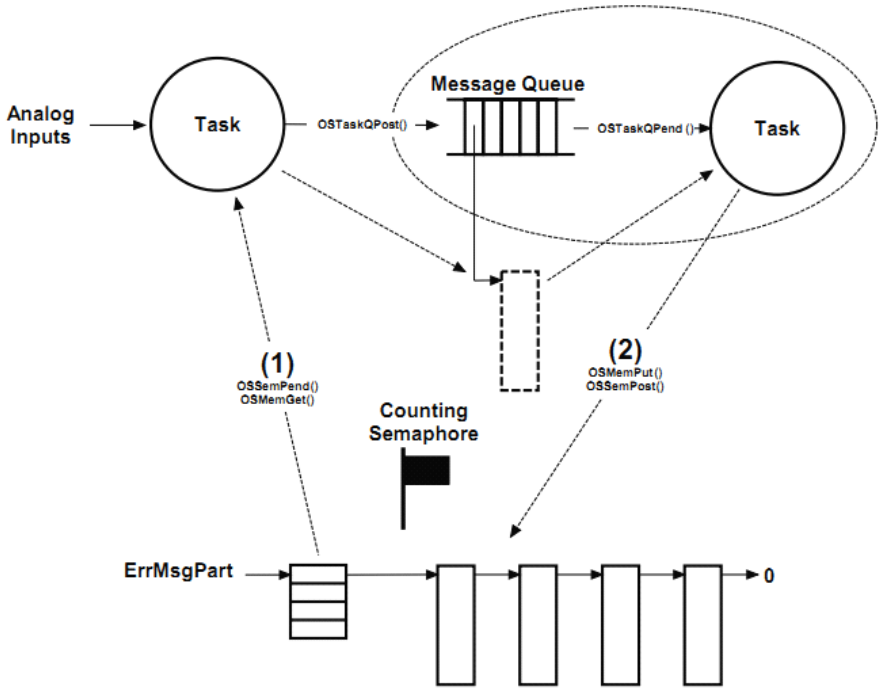


Figure 17-5 **Using a Memory Partition - blocking**

F17-5（1）获得一个内存块时，先调用 OSSemPend()获得一个信号量，然后再调用 OSMemGet()获得一个内存块。

F17-5（2）释放一个内存块时，先调用 OSMemPut()释放这个内存块，然后再调用 OSSemPost()释放这个信号量。

上面的操作必须以这个顺序执行。

用户可以在 ISR 中使用 OSMemGet()和 OSMemPut()，因为这两个函数不会被阻塞且能快速地被执行。（不能在 ISR 中调用会引起阻塞的函数）

17-5 总结

不要在嵌入式系统中使用 malloc()和 free()，因为这样会导致内存碎片。

可以用 malloc()动态的分配内存空间，但不要释放这些内存空间。{就是说定义不需要释放的空间时可以使用 malloc()，这样能使所定义的空间的利用率接近为 100%}

用户可以创建任意个内存分区（限制于处理器的 RAM）。

uC/OS-III 中与内存分区相关的函数都是以 OSMem???()为前缀。详见附录 A。

通过设置 OS_CFG.H 中的 OS_CFG_MEM_EN 为 1 开启内存管理服务。

OSMemGet()和 OSMemPur()可以在 ISR 中被调用。

18、移植 uC/OS-III

这个章节介绍了如何移植 uC/OS-III 到不同的处理器中。为了提高可移植性，绝大多数的 uC/OS-III 代码都是用 C 编写的。然而，依旧有一些需根据特定的处理器而编写相应的 C 和汇编代码。比如 uC/OS-III 直接控制处理器寄存器部分的代码，就只能用汇编实现。因为 uC/OS-III 类似于 uC/OS-II，用户可以先从移植 uC/OS-II 开始。

如果处理器满足如下条件，就能移植 uC/OS-III。

1）处理器有对应的能产生可重入代码的 C 编译器

2）处理器支持中断且能提供周期性的中断（通常介于 10 到 1000Hz 之间）。

3）可以关中断和开中断

4）处理器支持存储和载入堆栈指针、CPU 寄存器、堆栈的指令。

5）处理器有足够的 RAM 用于存放 uC/OS-III 的变量、结构体、内部任务堆栈、任务堆栈等

6）编译器支持 64 位的数据类型

图 18-1 显示了 uC/OS-III 的架构和它与其他软件、硬件成分的关系。

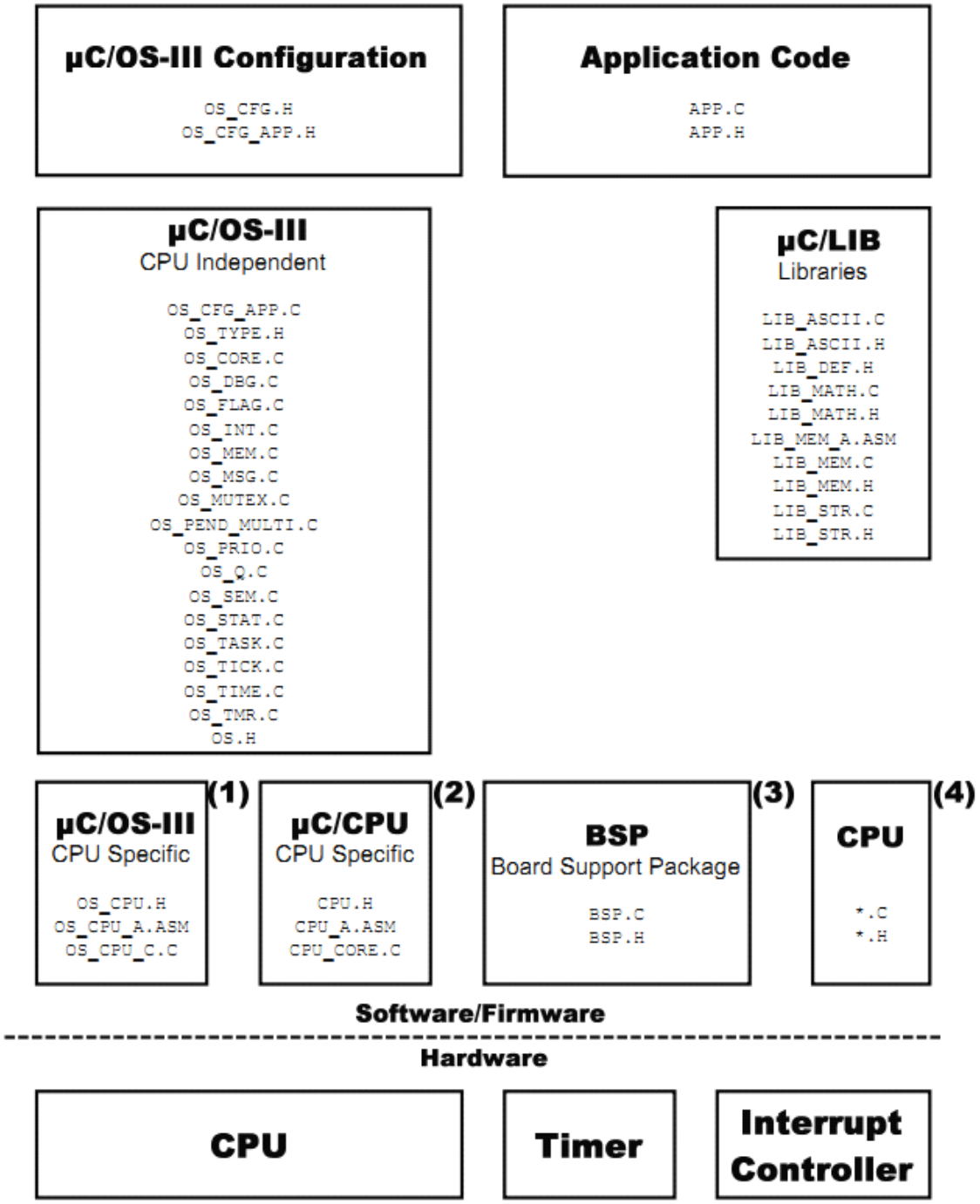


Figure 18-1 **µC/OS-III architecture**

F18-1（1）移植 uC/OS-III 需修改 3 个与内核相关的文件：OS_CPU.H、OS_CPU_A.ASM、OS_CPU_C.C。

F18-1（2）移植 uC/OS-III 需修改 3 个与 CPU 相关的文件：CPU.H、CPU_A.ASM、CPU_CORE.C。

F18-1（3）BSP 中通常包含了 uC/OS-III 与定时器（产生时基的定时器）、中断控制器的接口。

F18-1（4）有些半导体厂商会提高相应的固件库文件，这些文件会被包含在 CPU/MCU 中。

根据不同的处理器，移植可能会改变 100 到 400 行代码。

uC/OS-III 的移植看起像是 uC/OS-II 的移植。从 uC/OS-II 转换到 uC/OS-III 大约需要一小时时间。这个过程详见附录 C。

移植包括三方面内容：CPU、OS、BSP。

18-1 uC/CPU

与 CPU 相关的代码决定于 CPU 的架构。例如，关中断和开中断、堆栈的字长，堆栈的生长方向等等。与 CPU 相关的代码被封装在叫做 uC/CPU 的模块中。

表 18-1 中显示了 uC/CPU 中的文件及他们所在的位置。

File	Directory
CPU_DEF.H	\Micrium\Software\uC-CPU\
CPU.H	\Micrium\Software\uC-CPU\<processor>\<compiler>
CPU_C.C	\Micrium\Software\uC-CPU\<processor>\<compiler>
CPU_CFG.H	\Micrium\Software\uC-CPU\CFG\TEMPLATE
CPU_CORE.C	\Micrium\Software\uC-CPU\
CPU_CORE.H	\Micrium\Software\uC-CPU\
CPU_A.ASM	\Micrium\Software\uC-CPU\<processor>\<compiler>

Table 18-1 **µC/CPU files and directories**

CPU_DEF.H

该文件不需要被改变：CPU_DEF.H 中包含了 Micrium 公司提供的软件所用到的#define 定义的宏。

CPU.H

不同 CPU 间的字长可能不同，CPU.H 中定义了很多数据类型。在 Micrium 公司中，不会使用 C 编译器的数据类型 int，short，long，char 等，而是定义了更易于看懂的数据类型。查阅编译器的用户手册，看其数据类型所对应的字长。如果使用 32 位 CPU 时，以下的定义无需改变。

```
typedef            void        CPU_VOID;
typedef  unsigned  char        CPU_CHAR;
typedef  unsigned  char        CPU_BOOLEAN;
typedef  unsigned  char        CPU_INT08U;
typedef    signed  char        CPU_INT08S;
typedef  unsigned  short       CPU_INT16U;
typedef    signed  short       CPU_INT16S;
typedef  unsigned  int         CPU_INT32U;
typedef    signed  int         CPU_INT32S;
typedef  unsigned  long  long  CPU_INT64U;
typedef    signed  long  long  CPU_INT64S;
typedef            float       CPU_FP32;
typedef            double      CPU_FP64;
typedef  volatile  CPU_INT08U  CPU_REG08;
typedef  volatile  CPU_INT16U  CPU_REG16;
typedef  volatile  CPU_INT32U  CPU_REG32;
typedef  volatile  CPU_INT64U  CPU_REG64;
typedef            void      (*CPU_FNCT_VOID)(void);
typedef            void      (*CPU_FNCT_PTR )(void *);
typedef  CPU_INT32U            CPU_ADDR;
typedef  CPU_INT32U            CPU_DATA;
typedef  CPU_DATA              CPU_ALIGN;
typedef  CPU_ADDR              CPU_SIZE_T;
typedef  CPU_INT32U            CPU_STK;              (1)
typedef  CPU_ADDR              CPU_STK_SIZE;
typedef  CPU_INT16U            CPU_ERR;
typedef  CPU_INT32U            CPU_SR;               (2)
typedef  CPU_INT32U            CPU_TS;               (3)
```

（1）uC/OS-III 中非常重要的数据类型是 CPU_STK，它设置了堆栈的长度。决定了压入和弹出堆栈的数据是 8 位、16 位、32 位或者是 64 位的。

（2）CPU_SR 数据类型表示的是处理器的状态寄存器，中断时用于保存 CPU 状态的寄存器。

（3）CPU_TS 是时间戳的数据类型。

CPU.H 中也定义了关中断、开中断的宏：CPU_CRITICAL_ENTER() 和 CPU_CRITICAL_EXIT()。还有定义了 CPU_C.C 和 CPU_CORE.C 中函数的原型。

CPU_C.C

这是个可选的文件，存放了 CPU 的中断控制器、定时器的相关代码。绝大多数应用中是不包含这个文件的。

CPU_CFG.H

这是个配置文件，根据应用更改相应的#define。

CPU_CORE.C

这是通用的文件，不需要被改变。然而，它必须被包含。CPU_CORE.C 中定义了 CPU_Init()，CPU_CntLeadZeros()以及测量 CPU 最大关中断时间的函数等。

必须在调用 OSInit()之前调用 CPU_Init()。

CPU_CntLeadZeros()被调度器用于查找最高优先级的就绪任务（。CPU_CORE.C 用 C 模拟了计数清零指令。如果处理器中有该指令，那么就#define　CPU_CFG_LEAD_ZEROS_ASM_PRESENT。CPU_CntLeadZeros()可以用汇编语言编写的函数代替（放在 CPU_A.ASM 中）。使用这个函数之前需在 CPU.H 中定义 CPU_CFG_DATA_SIZE 的值。

CPU_CORE.C 中也包含了能读取时间戳的函数。uC/CPU 的时间戳是 32 位的。然而，uC/CPU 可以返回 64 位的时间戳。用时间戳可以知道事件发生的时刻，可以测量代码的执行时间。使用时间戳的前提是 CPU 计数值可以被读取。读取时间戳的代码可以被放在 BSP 中。

CPU_CORE.H

这里包含了 CPU_CORE.C 中函数的原型。

CPU_A.ASM

这个文件中包含了汇编代码，如关中断函数、开中断函数，计数清零函数等待。最低程度，这个函数需包括 CPU_SR_Save()和 CPU_SR_Restore()。

CPU_SR_Save()在中断时保存当前 CPU 的状态寄存器到堆栈。然而，在返回时，必须关掉中断。CPU_SR_Save()被宏 CPU_CRITICAL_ENTER()调用。

CPU_SR_Restore（）恢复中断前 CPU 的状态寄存器。CPU_SR_Restore()被宏 CPU_CRITICAL_EXIT()调用。

18-2 uC/OS-III 移植

表 18-2 显示了需被配置的 uC/OS-III 文件。

File	Directory
OS_CPU.H	\Micrium\Software\uCOS-III\Ports\<processor>\<compiler>\
OS_CPU_A.ASM	\Micrium\Software\uCOS-III\Ports\<processor>\<compiler>\
OS_CPU_C.C	\Micrium\Software\uCOS-III\Ports\<processor>\<compiler>\

Table 18-2 **µC/OS-III files and directories**

OS_CPU.H

这个文件中必须定义宏 OS_TASK_SW()，这个宏被 OSSched() 调用用于上下文切换。

OS_CPU.H 中必须定义宏 OS_TS_GET()，用于获得当前的时间戳。时间戳的数据类型为 CPU_TS，是 32 位的。

OS_CPU.H 中还需要定义 OSCtxSw()、OSIntCtxSw()、OSStartHighRdy()等函数的原型。

OS_CPU_A.ASM

这个文件中包含了以下汇编函数：

OSStartHighRdy()

OSCtxSw()

OSIntCtxSw()

和一个可选函数

OSTickISR()

OS_CPU_A.ASM 中存放的大多是直接操作 CPU 寄存器的函数，这些函数不能被 C 语言实现。

OS_CPU_C.C

这个文件中包含了钩子函数：

OSIdleTaskHook()

OSInitHook()

OSStatTaskHook()

OSTaskCreateHook()

OSTaskDelHook()

OSTaskReturnHook()

OSTaskStkInit()

OSTaskSwHook()

OSTimeTickHook()

详见附录 A。OS_CPU_C.C 中可以定义其他函数，但这些函数是强制的。

18-3 板级支持包 BSP

板级支持包的代码跟用户所使用的目标板有关。例如，BSP 中定义的函数用于开启关闭 LED、读按键值、初始化时钟等。

BSP 的文件包括：

BSP.C

BSP.H

BSP_INT.C

BSP_INT.H

这些文件所在的目录为：

```
\Micrium\Software\EvalBoards\<manufacturer>\
<board_name>\<compiler>\BSP\
```

BSP.C 和 **BSP.H**

这两个文件中包含了函数的定义和申明如 BSP_Init()、BSP_LED_On()、BSP_LED_Off()、BSP_LED_Toggle()、BSP_PB_Rb()等。用户可以定义自己的函数，最好以 BSP_作为前缀。

在 BSP.C 中，可以添加 CPU_TS_TmrInit()用于初始化 CPU 的时钟速率。

CPU_TS_TmrGet()用于读取 CPU 时钟计数值。如果定时器是 16 位的，那么就需定义两个字，将其转换为 32 位的。如果定时器是 32 位的，CPU_TS_TmrGet()将直接返回这个 32 位的值。

BSP_INT.C 和 BSP_INT.H

这些文件中用于存放与中断控制器相关的函数。例如，可以设置特定的关中断、开中断，响应中断控制器，将所有中断的 ISR 指向同一地址等（ 见第 9 章“中断管理”）。

```
void  BSP_IntHandler (void)                                          (1)
{
    CPU_FNCT_VOID   p_isr;

    while (interrupts being asserted) {                              (2)
        p_isr = Read the highest priority interrupt from the controller; (3)
        if (p_isr != (CPU_FNCT_VOID)0) {                             (4)
           (*p_isr)();                                               (5)
        }
        Acknowledge interrupt controller;                            (6)
    }
}
```

（1）中断控制器程序是用汇编写的，并在进入 ISR 之前保存 CPU 的相关寄存器。

（2）中断控制器可以返回相应中断处理程序的地址。

（3）中断产生，并中断处理器的响应。

（4）检测中断控制器的返回值，确保它不是 NULL。

（5）调用 ISR。

（6）ISR 被正常运行。

18-4 总结

移植包括三个部分代码：CPU，OS，BSP。

移植需修改 3 个内核文件：OS_CPU.H，OS_CPU_A.ASM，OS_CPU_C.C。

移植需修改 3 个 CPU 文件：CPU.H，CPU_A.ASM，CPU_C.C。

最后创建适应于目标板的板级支持包 BSP。

uC/OS-III 的移植类似于 uC/OS-III 的移植。