一步一步教你使用 uCOS-II

前言篇

前段时间看了 Linux 版块“zjw50001”网友上传的资料《一步一步教你开发嵌入式 Linux 应用程序》感觉对于学习 Linux 的新手来说有很大帮助。同时也很受启发。决定在 UCOS 版块发这样一个连续贴来介绍一下 uCOS-II。主要包括 uCOS-II 的介绍、UCOS-II 的移植、uCOS-II 的的源码解析和 UCOS-II 的应用案例。

uCOS-II 的版本采用常用的 2.52 版本，开发平台用我手头现有的 STM32F103XXX 自制的开发板。

在这采用边写边讨论的方式。如果在过程中大家有疑问请及时跟帖提出。会在每篇后给出解决方法，同时考虑到工作量比较大，所以希望大家积极参与，让我们共同把 uCOS-II 这个嵌入式操作系统的知识传授个每一位需要的网友。

第一章 UCOS 介绍

第一篇 UCOS 介绍

这个大家都知道。呵呵。考虑到咱们学习的完整性还是在这里唠叨一下。让大家再熟悉一下。高手们忍耐一下吧！ uC/OS II(Micro Control Operation System Two)是一个可以基于 ROM 运行的、可裁减的、抢占式、实时多任务内核，具有高度可移植性，特别适合于微处理器和控制器，是和很多商业操作系统性能相当的实时操作系统(RTOS)。为了提供最好的移植性能，uC/OS II 最大程度上使用 ANSI C 语言进行开发，并且已经移植到近 40 多种处理器体系上，涵盖了从 8 位到 64 位各种 CPU(包括 DSP)。

uC/OS II 可以简单的视为一个多任务调度器，在这个任务调度器之上完善并添加了和多任务操作系统相关的系统服务，如信号量、邮箱等。其主要特点有公开源代码，代码结构清晰、明了，注释详尽，组织有条理，可移植性好，可裁剪，可固化。内核属于抢占式，最多可以管理 60 个任务。

μC/OS-II 的前身是 μC/OS，最早出自于 1992 年美国嵌入式系统专家 Jean J.Labrosse 在《嵌入式系统编程》杂志的 5 月和 6 月刊上刊登的文章连载，并把 μC/OS 的源码发布在该杂志的 B B S 上。

μC/OS 和 μC/OS-II 是专门为计算机的嵌入式应用设计的， 绝大部分代码是用 C 语言编写的。CPU 硬件相关部分是用汇编语言编写的、总量约 200 行的汇编语言部分被压缩到最低限度，为的是便于移植到任何一种其它的 CPU 上。用户只要有标准的 ANSI 的 C 交叉编译器，有汇编器、连接器等软件工具，就可以将 μC/OS-II 嵌人到开发的产品中。μC/OS-II 具有执行效率高、占用空间小、实时性能优良和可扩展性强等特点，最小内核可编译至 2KB 。μC/OS-II 已经移植到了几乎所有知名的 CPU 上。

严格地说 uC/OS-II 只是一个实时操作系统内核，它仅仅包含了任务调度，任务管理，时间管理，内存管理和任务间的通信和同步等基本功能。没有提供输入输出管理，文件系统，网络等额外的服务。但由于 uC/OS-II 良好的可扩展性和源码开放，这些非必须的功能完全可以由用户自己根据需要分别实现。

uC/OS-II 目标是实现一个基于优先级调度的抢占式的实时内核，并在这个内核之上提供最基本的系统服务，如信号量，邮箱，消息队列，内存管理，中断管理等。

uC/OS-II 以源代码的形式发布，但并不意味着它是开源软件。你可以将其用于教学和私下研究（peaceful research）；但是如果你将其用于商业用途，那么你必须通过 Micrium 获得商用许可。

虽然 uCOS-II 在商业上使用时需要的得到授权并且费用也是一笔不小的数字，但是他的开源毕竟带领我们走入了内核的世界。在此我代表嵌入式工程师向 Mr Jean J.Labrosse 致谢。

任务管理

uC/OS-II 中最多可以支持 64 个任务，分别对应优先级 0～63，其中 0 为最高优先级。63 为最低级，系统保留了 4 个最高优先级的任务和 4 个最低优先级的任务，所有用户可以使用的任务数有 56 个。
uC/OS-II 提供了任务管理的各种函数调用，包括创建任务，删除任务，改变任务的优先级，任务挂起和恢复等。
系统初始化时会自动产生两个任务：一个是空闲任务，它的优先级最低，该任务仅给一个整形变量做累加运算；另一个是系统任务，它的优先级为次低，该任务负责统计当前 cpu 的利用率。
在系统初始化完毕后启动任务时必须创建一份用户任务，也就是说必须有一个应用程序（用户任务，使用应用程序对于我们经常使用 Windows 用户容易接受一些。呵呵），否则系统会崩溃。当然还有一些其他的要求，咱们后续再说，下面简要概述一下任务管理相关的函数
1：建立任务 OSTaskCreat()/OSTaskCreatExt()
如果想让 UCOS 管理用户的任务，必须先建立任务。可以通过将任务的地址和其他参数传递到以下两个函数之一来建立任务。当调用 OSTaskCreat()时，需要四个参数：
OSTaskCreate(void(*task)(void*pd),void*pdata,OS_STK*ptos,INTU prio)

Task：是指向任务代码的指针，pdata：是任务开始执行是，传递给任务的参数的指针，ptos：是分配给任务的堆栈的栈顶指针，prio 是分配给任务的优先级。
也可以用 OSTaskCreatExt()，不过该函数需要 9 个参数，前四个参数与 OSTaskCreat()一样，例如：
INT8U OSTaskCreateExt(void(*task)(void *pd),void *pdata,OS_STK *ptos, INT8U prio, INT16U id, OS_STK *pbos, OS_STK *pbos, OS_STK *pbos, INT16U opt)
id 参数为要建立的任务创建一个特殊的标识符。pbos 是指向任务的堆栈栈底的指针，用于堆栈的检验。stk _size 用于指定堆栈成员数目的容量。pext 是指向用户附加的数据域的指针，用来扩展任务的 OS_TCB。opt 用于设定 OSTaskCreateExt()的选项，指定是否允许堆栈检验，是否将堆栈清零，任务是否要进行浮点操作等等。
2：任务堆栈 OS_STK（）
每个任务都有自己的堆栈，堆栈必须申明为 OS_STK 类型，并且由连续的内存空间组成。可以静态分配堆栈空间，也可以动态分配堆栈空间。
3：堆栈检验 OSTaskStkChk()
有时确定任务实际需要的堆栈空间的大小是很有必要的，因为这样就可以避免为任务分配过多的堆栈空间，从而减少应用程序代码所需的 RAM 空间。
4：删除任务 OSTaskDel()
有时需要删除任务，删除任务，是说任务返回并处于休眠态，并不是说任务的代码被删除了，只是任务的代码不再被 UCOS 调用。删除任务前应保证所删任务并非空闲任务。
5：请求删除任务 OSTaskDelReq()
有时，任务会占用一些内存缓冲或信号量一类的资源。这时，假如另一个任务试图删除该任务，这些被占用的资源就会因为没有被释放而丢失。在这种情况下，需想办法拥有这些资源的任务在使用完资源后先释放资源，再删除自己。
6：改变任务的优先级 OSTaskChangePrio()
在建立任务时，会分配给任务一个优先级。在程序运行期间，可以通过调用该函数改变任务的优先级。也就是说，UCOS 允许动态的改变任务的优先级。
7：挂起任务 OSTaskSuspend()
任务挂起是一个附加功能，也就是说，如果任务在被挂起的同时也在等待延迟时间到，那么，需要对任务做取消挂起的操作，并且等待延迟时间到，任务才能转让就绪状态。任务可以挂起自己或者其他任务。
8：恢复任务 OSTaskResume()
挂起的任务只有通过该函数才能被恢复。
9：获得任务的信息 OSTaskQuery()
通过调用该函数，来获得自身或其他应用任务的信息

时间管理

uC/OS-II 的时间管理是通过定时中断来实现的，该定时中断一般为 10 毫秒或 100 毫秒发生一次（这个时间片段是 OS 的作者推荐的，大家可以参考邵贝贝翻译的《嵌入式实时操作系统 ucos-II》这本书），时间频率取决于用户对硬件系统的定时器编程来实现。中断发生的时间间隔是固定不变的，该中断也成为一个时钟节拍。这里隐含的意思就是你选择的芯片如果想使用 UCOS 系统，前提条件一定要有一个 Timer。
uC/OS-II 要求用户在定时中断的服务程序中，调用系统提供的与时钟节拍相关的系统函数，例如中断级的任务切换函数，系统时间函数。

uCOS 时间管理的相关函数
1：任务延迟函数 OSTimeDly()
Ucos 提供一个可以被任务调用而将任务延时一段特定时间的功能函数，即 OSTimeDly().任务调用 OSTimeDly()后，一旦规定的时间期满或者有其他的任务通过调用 OSTimeDlyResume()取消了延时，他就会进入就绪状态。只有当该任务在所有就绪态任务中具有最高的优先级，它才会立即运行。
2：按时，分，秒延时函数 OSRimeDLyHMSM()
与 OSTimeDly()一样，调用 OSRimeDlyHMSM()函数也会是 UCOS 进行一次任务调度，并且执行下一个优先级最高的就绪任务。当 OSTimeDlyHMSM()后，一旦规定的时间期满，或者有 OSTimeDlyResume()，它就会马上处于就绪态。同样，只有当该任务在所有就绪态任务中具有最高的优先级，他才开始运行。
3：恢复延时的任务 OSTimeDlyResume()
延时的任务可以不等待延时的期满，而是通过其他任务取消延时而使自己处于就绪态，可以通过该函数来实现，实际上，OSTimeDlyResume()也可以唤醒正在等待的事件。
4：系统时间 OSTimeGet()和 OSTimeSet()

内存管理

在 ANSI C 中是使用 malloc 和 free 两个函数来动态分配和释放内存。例如在 Linux 系统中就是这样。但在嵌入式实时系统中，多次这样的操作会导致内存碎片，因为嵌入式系统尤其是 uCOS 是实地址模式，这种模式在分配任务堆栈时需要整块连续的空间，否则任务无法正确运行。且由于内存管理算法的原因，malloc 和 free 的执行时间也是不确定。这点是实时内核最大的矛盾。
基于以上的原因 uC/OS-II 中把连续的大块内存按分区管理。每个分区中包含整数个大小相同的内存块，但不同分区之间的内存快大小可以不同。用户需要动态分配内存时，系统选择一个适当的分区，按块来分配内存。释放内存时将该块放回它以前所属的分区，这样能有效解决碎片问题，同时执行时间也是固定的。
同时 uCOS-II 根据以上的处理封装了适合于自己的动态内存分配函数 OSMemGet（）和 OSMemPut（），但是使用这两个函数动态分配内存前需要先创建内存空间，也就是第二段咱们介绍的内存分块。呵呵，不罗嗦了，具体的关于内存管理的函数如下：
内存控制块的数据结构

```
Typedef
struct
{void     *osmemaddr      ;指向内存分区起始地址的指针。
Void     *osmemfreelist   ;指向下一个空余内存控制块或者下一个空余内存块的指针，
Int32u   osmemblksize    ;内存分区中内存块的大小，是建立内存分区时定义的。
Int32u osmemnblks        ;内存分区中总的内存块数量，也是建立该内存分区时定义的。
Int32u   osmemnfree      ;内存分区块中当前获得的空余块数量。
}os_mem;
```

1；建立一个内存分区，OSMemCreate()
2：分配一个内存块，OSMemGet()
应用程序通过调用该函数，从已经建立的内存分区中申请一个内存块。该函数唯一的参数是指向特定内存分区的指针。
3：释放一个内存块，OSMemPut()

当应用程序不再使用一个内存块时，必须及时的把它释放，并放回到相应的内存分区中，这个操作就是通过调用该函数实现的。
4：查询一个内存分区的状态，OSQMemQuery()。

任务间通信与同步

对一个多任务的操作系统来说，任务间的通信和同步是必不可少的。uC/OS-II 中提供了 4 种同步对象，分别是信号量，邮箱，消息队列和事件。所有这些同步对象都有创建，等待，发送，查询的接口用于实现进程间的通信和同步。
对于这 4 种同步对象将在后面一一讨论。

任务调度

uC/OS-II 采用的是可剥夺型实时多任务内核。可剥夺型的实时内核在任何时候都运行就绪了的最高优先级的任务。
uC/os-II 的任务调度是完全基于任务优先级的抢占式调度，也就是最高优先级的任务一旦处于就绪状态，则立即抢占正在运行的低优先级任务的处理器资源。为了简化系统设计，uC/OS-II 规定所有任务的优先级不同，因为任务的优先级也同时唯一标志了该任务本身。
UCOS 的任务调度在一下情况下发生：
1） 高优先级的任务因为需要某种临界资源，主动请求挂起，让出处理器，此时将调度就绪状态的低优先级任务获得执行，这种调度也称为任务级的上下文切换。
2）高优先级的任务因为时钟节拍到来，在时钟中断的处理程序中，内核发现高优先级任务获得了执行条件(如休眠的时钟到时)，则在中断态直接切换到高优先级任务执行。这种调度也称为中断级的上下文切换。
这两种调度方式在 uC/OS-II 的执行过程中非常普遍，一般来说前者发生在系统服务中，后者发生在时钟中断的服务程序中。
调度工作的内容可以分为两部分：最高优先级任务的寻找和任务切换。其最高优先级任务的寻找是通过建立就绪任务表来实现的。u C / O S 中的每一个任务都有独立的堆栈空间，并有一个称为任务控制块 TCB(Task Control Block)的数据结构，其中第一个成员变量就是保存的任务堆栈指针。任务调度模块首先用变量 OSTCBHighRdy 记录当前最高级就绪任务的 TCB 地址，然后调用 OS_TASK_SW()函数来进行任务切换。

第二章 搭建 UCOS-II 2.52 版的调试平台

在这一章中我们主要讨论 UCOSII 的源码调试环境，为了给大家一个共同的学习平台，我搜集整理了一写资料，就是以 X86 为平台，使用 BC31（这个堪称骨灰级的编译器）来调试 UCOSII 源码。当然你也可以用 BC45 或更高版本的编译器，具体方法大同小异，我在此就不再啰嗦。
本章节的主要内容包括四点：
1、下载并安装 BC31 编译器
2、下载并安装 UCOS-II2.52 版本源代码
3、使用 BC31 编译 UCOS-II 源码
4、让 OS 的第一个任务 RUN 起来

接下来会在每个帖子中讨论一点。耐心等待哦！

下载并安装 BC31 编译器

我在这里提供给大家这个骨灰级的编译器 BC31.需要的可以下载。见附件（骨灰级编译器 BC31）由于这个软件的比较大，分成两个压缩包。下班了，先到这里，回家再传附件！

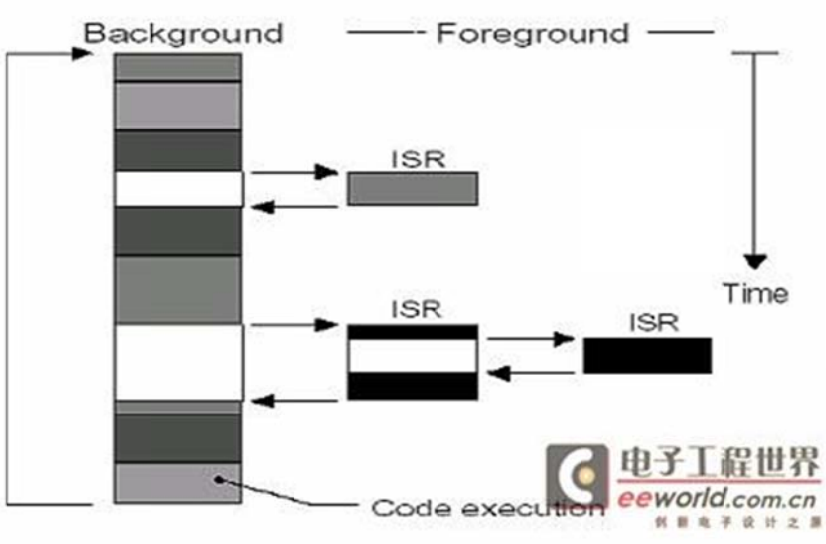
让自己的第一个任务 Run 起来

前面已经给大家介绍了如何在 PC 机上调试 UCOS，方法和需要的软件都介绍给大家了，相信有兴趣的朋友已经安装调试了，下面咱们就让自己的第一个任务在 PC 上 Run 起来。
OK，下面我就分步介绍建立自己的第一个任务
第一步：CopyC:\SOFTWARE\uCOS-II 目录下的 EX1_x86L 文件夹。作为我们的工程模板
第二步：修改工程模板的名字为：HelloEEWorld
第三部：按照咱们前面的《使用 BC31 工具编译 UCOS - II 的源码过程 》修改配置文件；
第四步：修改 Test.c 文件，建立自己的第一个任务
具体的内容我就不再帖子上写了。大家可以参考附件 HelloEEWorld.rar 里面的 Test.c 文件。
然后编译
OK，第一个任务就 Run 起来了，显示如下界面

第三章：关于 UCOS 任务的理解

关于 UCOS 任务的理解

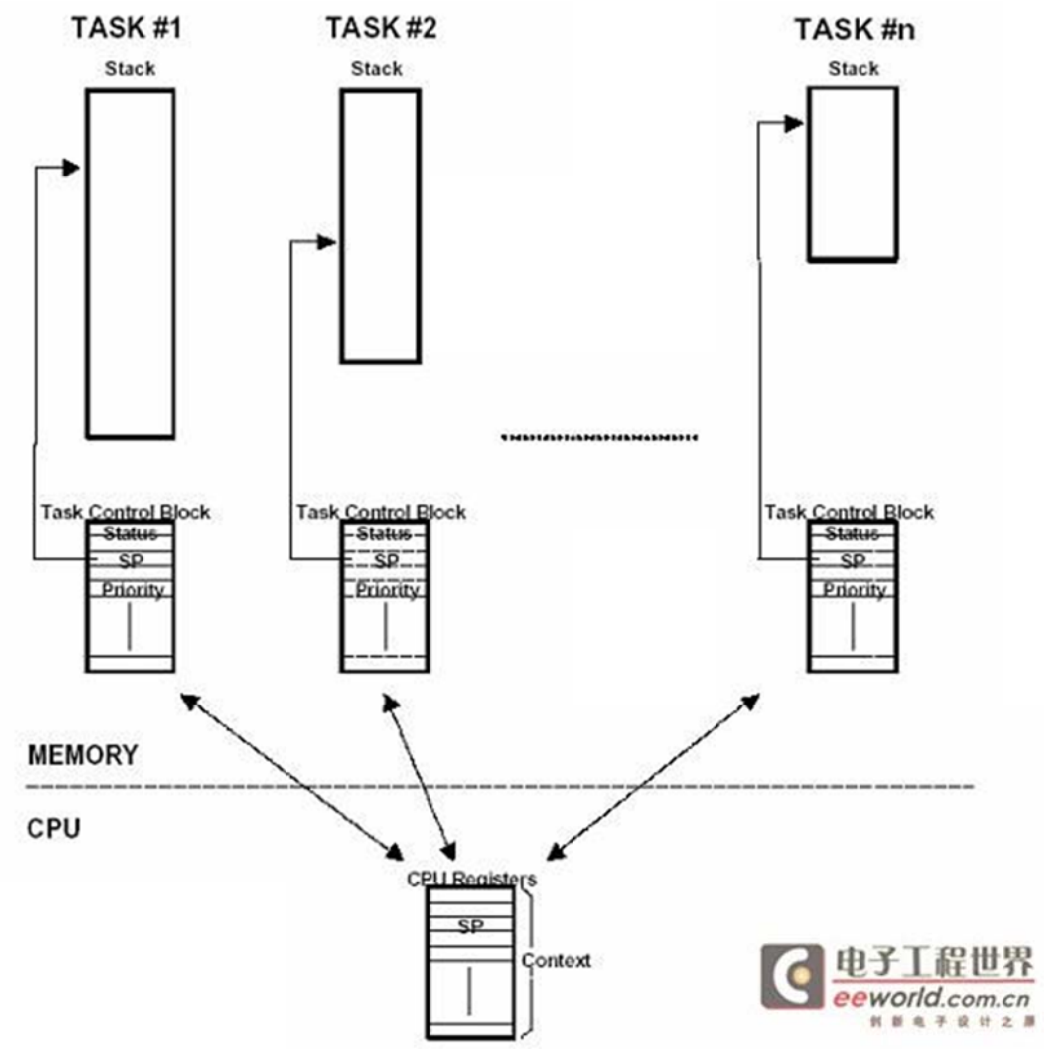
UCOS 的运行是基于任务运行的，为了能够好的使用 UCOS 我们先要对 UCOS 的任务的概念做一个理解
在学习ＵＣＯＳ任务前我们先对我们以前使用的模式做一个回顾－－前后台模式。



2011-3-6 22:23

这种系统可称为前后台系统或超循环系统(Super-Loops)。应用程序是一个无限的循环，循环中调用相应的函数完成相应的操作，这部分可以看成后台行为(background)。中断服务程序处理异步事件，这部分可以看成前台行 foreground。后台也可以叫做任务级。前台也叫中断级。时间相关性很强的关键操作(Critical operation)一定是靠中断服务来保证的。因为中断服务提供的信息一直要等到后台程序走到该处理这个信息这一步时才能得到处理，这种系统在处理信息的及时性上，比实际可以做到的要差。这个指标称作任务级响应时间。最坏情况下的任务级响应时间取决于整个循环的执行时间。因为循环的执行时间不是常数，程序经过某一特定部分的准确时间也是不能确定的。进而，如果程序修改了，循环的时序也会受到影响。这种系统是在我们上学时和做小项目时经常用到，很多工程师称这种方式为“裸奔”。哈哈！我大学毕业后的钱三年写的项目都是在裸奔。

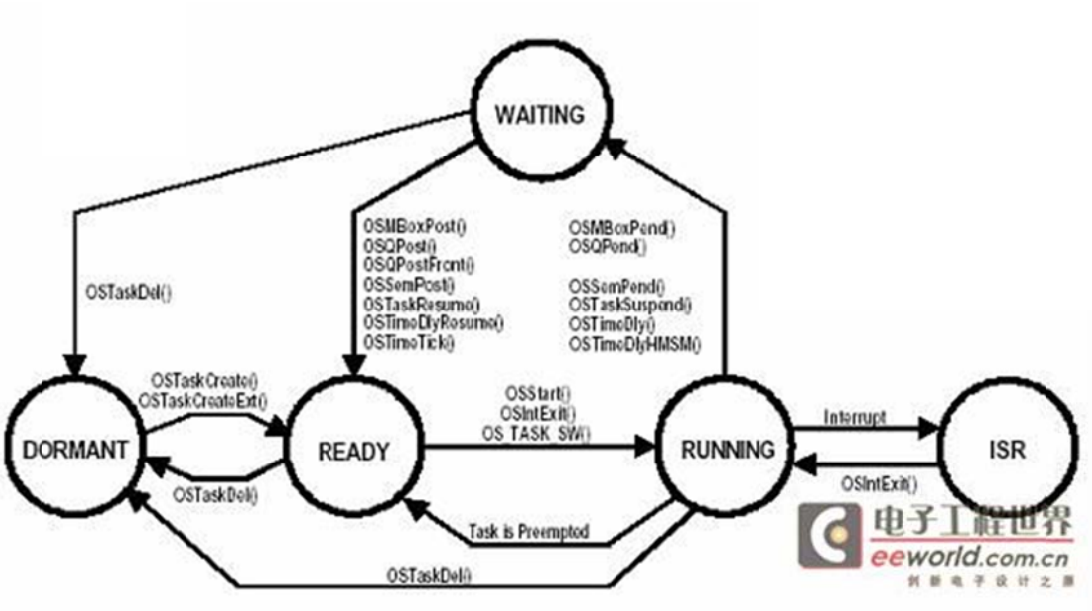
UCOS-II 是基于任务运行的。一个任务，也称作一个线程，是一个简单的程序，该程序可以认为 CPU 完全只属该程序自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间(如下图所示)。



Pic 22011-3-6 22:23

可以这么理解，UCOS-II 的每一个任务都有一个 CPU，任务在运行时占用 CPU 的全部资源，同时拥有自己的一套寄存器，当任务执行完毕后（时间片到），他把自己的 CPU 寄存器所有内容保存到自己的堆栈中，同时把 CPU 让给别的任务，那么得到 CPU 使用权的任务把自己的 CPU 寄存器从自己的堆栈中放到真正的 CPU 寄存器中开始运行，就这样周而复始。

大家一定不要把任务的运行当成是函数的调用，这完全是两回事。这个我们到后面的任务调度时在细说。每个任务都是一个无限的循环。每个任务都处在以下 5 种状态之一的状态下，这 5 种状态是休眠态， 就绪态、 运行态、 挂起态(等待某一事件发生)和被中断态（参见下图） 休眠态相当于该任务驻留在内存中，但并不被多任务内核所调度。就绪意味着该任务已经准备好， 可以运行了， 但由于该任务的优先级比正在运行的任务的优先级低， 还暂时不能运行。运行态的任务是指该任务掌握了 CPU 的控制权，正在运行中。挂起状态也可以叫做等待事件态 WAITING，指该任务在等待，等待某一事件的发生， （例如等待某外设的 I/O 操作，等待某共享资源由暂不能使用变成能使用状态， 等待定时脉冲的到来或等待超时信号的到来以结束目前的等待，等等） 。最后，发生中断时，CPU 提供相应的中断服务，原来正在运行的任务暂不能运行，就进入了被中断状态。如下图表示μC/OS-Ⅱ中一些函数提供的服务，这些函数使任务从一种状态变到另一种状态。



Pic 32011-3-6 22:23

简单的我们可以把每一次任务的切换当成一次中断，这个中断不同于我们在使用前后台模式时的中断，那个中断是硬件中断，中断时需要保存的 CPU 寄存器是由硬件实现的，而在 UCOS 中的任务切换是软中断，CPU 保存了必要的寄存器后在切换时系统会在保存任务使用的寄存器。

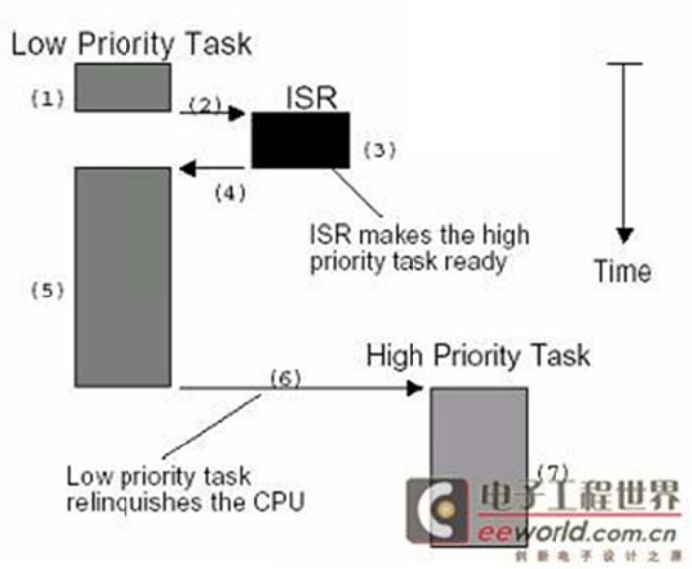
补充知识-可剥夺型内核和不可剥夺型内核

不可剥夺型内核

不可剥夺型内核要求每个任务自我放弃 CPU 的所有权。不可剥夺型调度法也称作合作型多任务，各个任务彼此合作共享一个 CPU。异步事件还是由中断服务来处理。中断服务可以使一个高优先级的任务由挂起状态变为就绪状态。 但中断服务以后控制权还是回到原来被中断了的那个任务，直到该任务主动放弃 CPU 的使用权时，那个高优先级的任务才能获得 CPU 的使用权。

不可剥夺型内核允许每个任务运行，直到该任务自愿放弃 CPU 的控制权。中断可以打入运行着的任务。 中断服务完成以后将 CPU 控制权还给被中断了的任务。任务级响应时间要大大好于前后系统，但仍是不可知的， 商业软件几乎没有不可剥夺型内核。

不可剥夺型内核的工作过程见下图：



2011-3-6 22:26

可剥夺型内核

当系统响应时间很重要时，要使用可剥夺型内核。因此，μC/OS-II 以及绝大多数商业上销售的实时内核都是可剥夺型内核。最高优先级的任务一旦就绪，总能得到 CPU 的控制权。当一个运行着的任务使一个比它优先级高的任务进入了就绪态， 当前任务的 CPU 使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了 CPU 的控制权。如果是中断服务子程序使一个高优先级的任务进入就绪态，中断完成时，中断了的任务被挂起，优先级高的那个任务开始运行。使用可剥夺型内核，最高优先级的任务什么时候可以执行，可以得到 CPU 的控制权是可知的。使用可剥夺型内核使得任务级响应时间得以最优化。

可剥夺型内核的工作过程是这样的：

2011-3-6 22:26

UCOS-II 任务调度

任务调度是内核的主要职责之一，就是要决定该轮到哪个任务运行了。多数实时内核是基于优先级调度法的，UCOS 也不例外。每个任务根据其重要程度的不同被赋予一定的优先级。基于优先级的调度法指，CPU 总是让处在就绪态的优先级最高的任务先运行。然而，究竟何时让高优先级任务掌握 CPU 的使用权，有两种不同的情况，这要看用的是什么类型的内核，是不可剥夺型的还是可剥夺型内核。

上一次咱们已经介绍了可剥夺型内核和不可剥夺型内核的工作过程了。在此不再赘述！

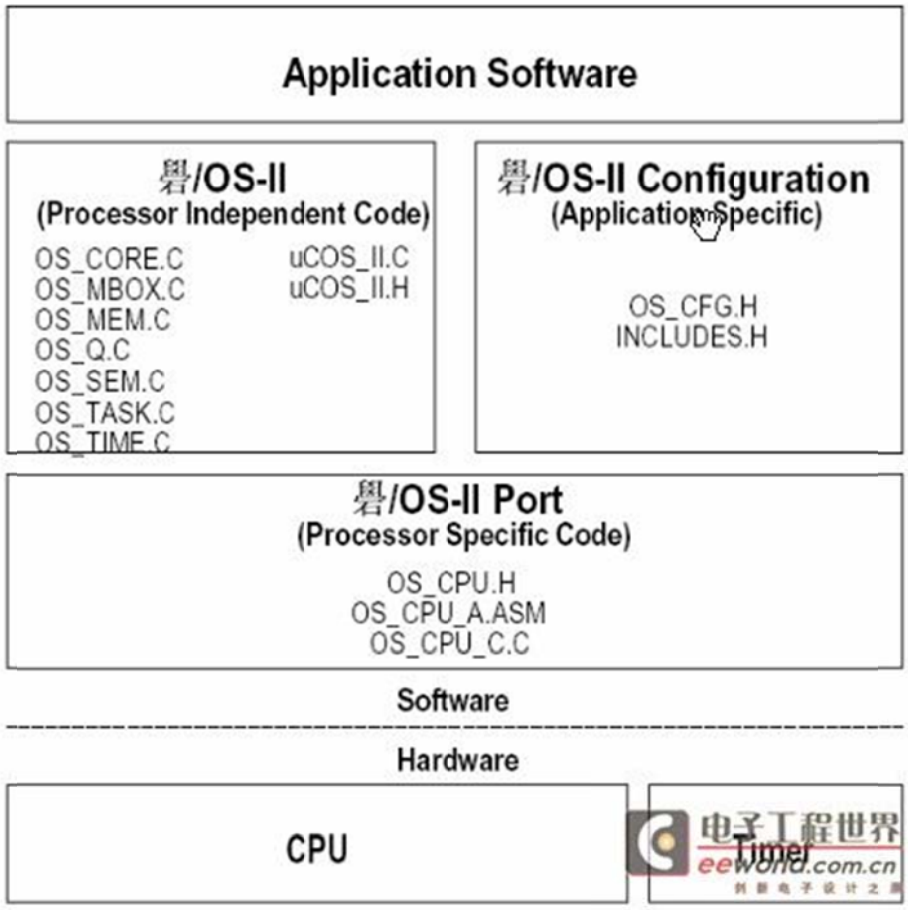
当多任务内核决定运行另外的任务时，它保存正在运行任务的当前状态，即 CPU 寄存器中的全部内容。这些内容保存在任务的当前状况保存区，也就是任务自己的栈区之中，上一次讨论的内容中有这个图示。入栈工作完成以后，就是把下一个将要运行的任务的当前状况从该任务的栈中重新装入 CPU 的寄存器，并开始下一个任务的运行。这个过程叫做任务切换。任务切换过程增加了应用程序的额外负荷。CPU 的内部寄存器越多，额外负荷就越重。做任务切换所需要的时间取决于 CPU 有多少寄存器要入栈。实时内核的性能不应该以每秒钟能做多少次任务切换来评价。而是要看 OS 总的关中断时间。总的关中断时间越短说明这个内核的实时性越好。这个问题在前面一个坛友的问题中我做了详细的描述，有兴趣的朋友可以在 UCOS 这个版块找找这个帖子。

任务调度的算法有很多种。一种是基于优先级的。一种是基于时间片的。这两种算法在邵贝贝教授翻译的《UCOS-II 内核详解》这本书中有详细解释。我就不再重复。如果坛子里有朋友对此有什么不明白。可以在这里留言。咱们再讨论。

UCOS-II 的文件结构

前面我们对 UCOS 的基础知识做了了解，其中有些地方由于邵贝贝翻译的树上讲解的很少我就没有班门弄斧，大家可以结合那本书来看。有问题或不明白的在这里讨论，欢迎大家剔除问题。

这次我们主要了解 UCOS-II 的文件结构。等对 UCOS 文件结构了解以后，我们就逐一的去讲解其各章的重点和难点，达到在短时间内学会使用 UCOS。



UCOS-II 内核文件结构图

2011-3-17 06:56

我们利用这张图片把 UCOS 的内部做一个解剖，我们可以清楚的看到 UCOS 内核的结构及层次，在这个图的最下面是我们使用的硬件，就是我们的移植平台，比如 STM32F103XX 系列的最小系统版、51 最小系统版。呵呵，我本人觉得把 UCOS 移植到 51 上的意义不大。只是学习可以，使用我就不建议了！从图中我们可以知道，要想移植 UCOS 你的硬件平台必须具备一个定时器，也就是上图中的 TIMER。这个 TIMER 是用来给 UCOS 提供时钟节拍的，相当于我们人的心跳。如果没有这个 TIMER，统统就无法运行。

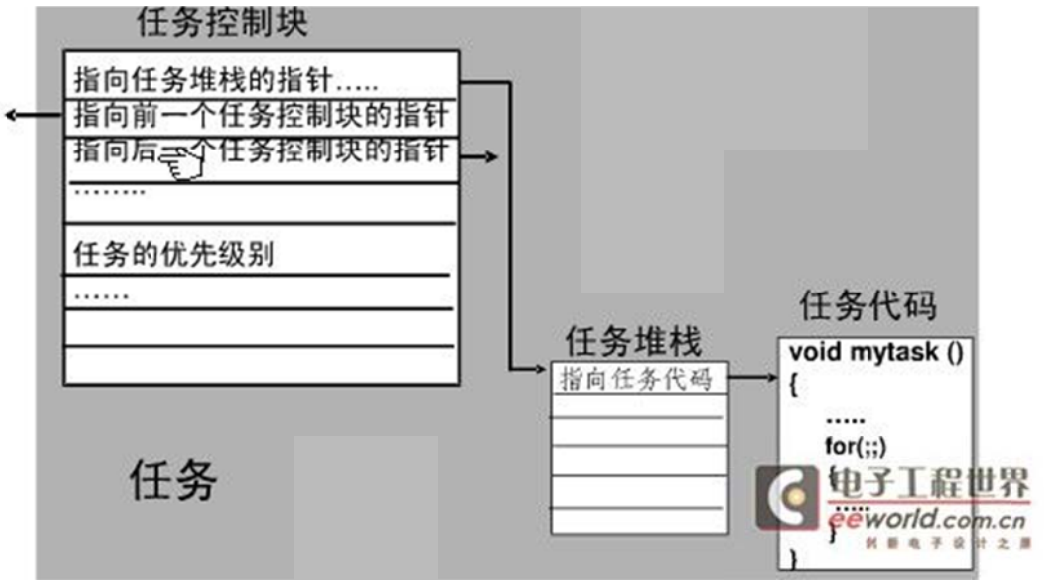
再往上就是软件了，软件的第一层是我们移植的重点，这三个文件内主要包括一些与处理器相关的代码，在后面我们我们再讲解移植过程的时候会详细的讨论到这三个文件。

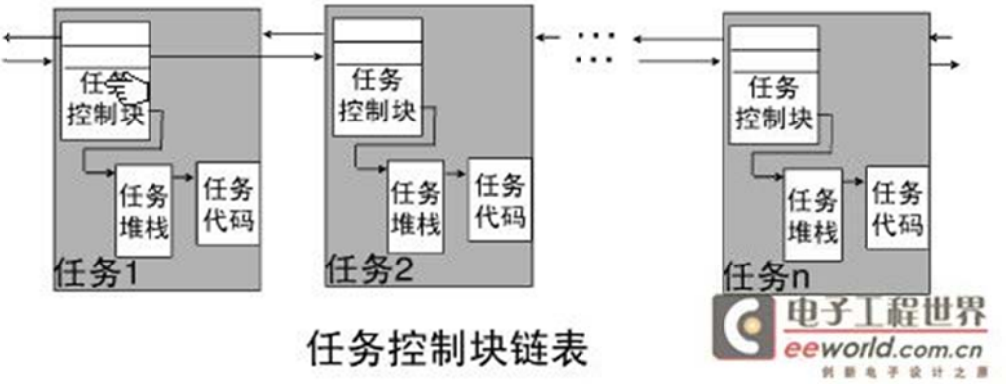
在往上左侧就是系统内核源码的各个文件。有兴趣的坛友可以参考邵贝贝教授翻译的书进行深入学习，由于我在这里的主要任务是告诉大家如何使用 UCOS，故不再过多的讲解源码部分，只是告诉大家如何使用即可。当然，如果你在研究过程中遇到问题可以拿出来和大家共同讨论，右侧是系统的配置文件，相对比较简单，主要涉及到一些功能的裁剪。

最上层是我们的应用软件，相当于我们在电脑上使用的 Office 软件等，当然这里是你自己的任务代码。

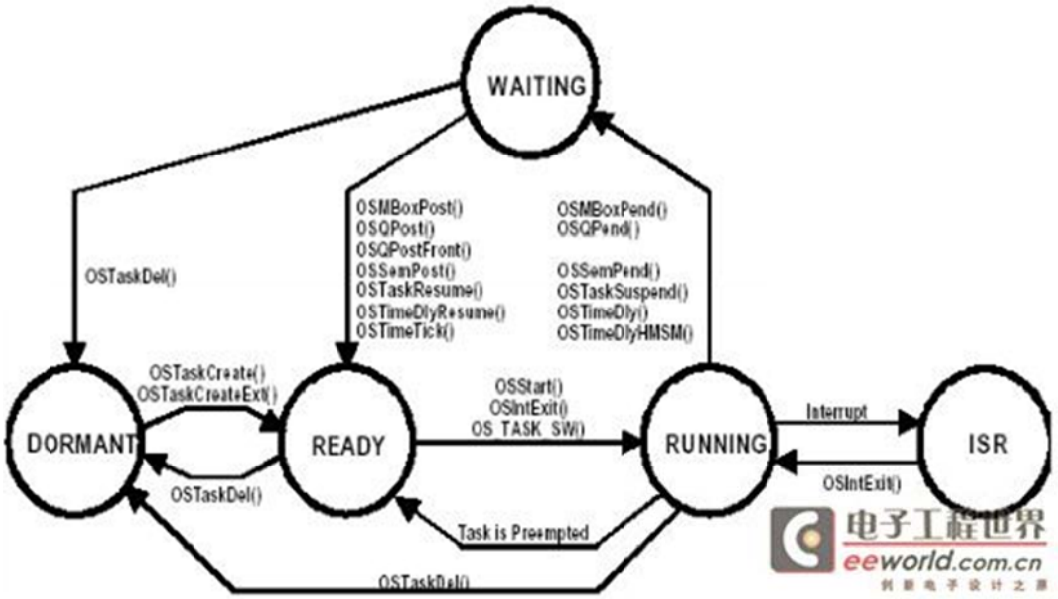
UCOS 的任务及状态

任务的资源主要包括以下几部分，ECB 控制块、任务堆栈、任务代码及与 CPU 共用的寄存器和 CPU 的使用权



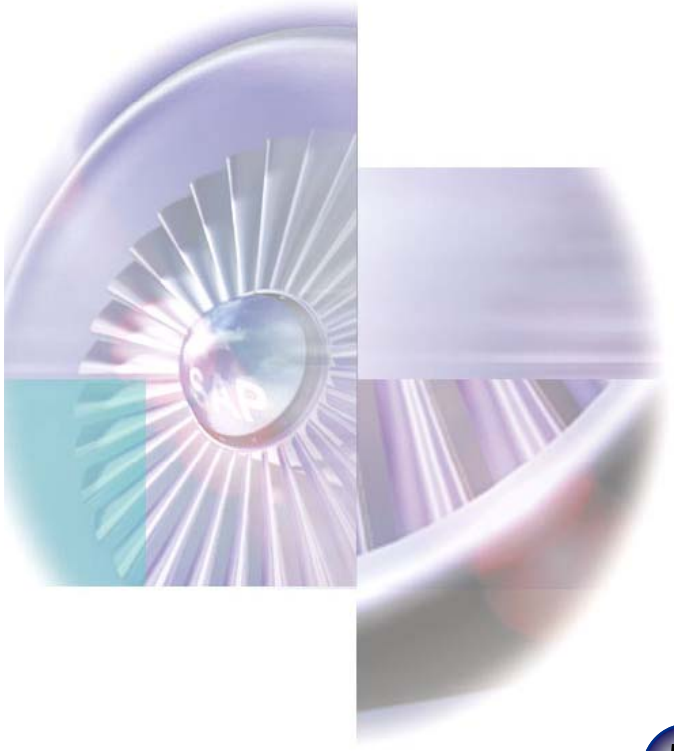


任务控制块链表



第 4 章_uCOS-II 及其任务.ppt.pdf

第4章 进程与线程_uCOS的任务



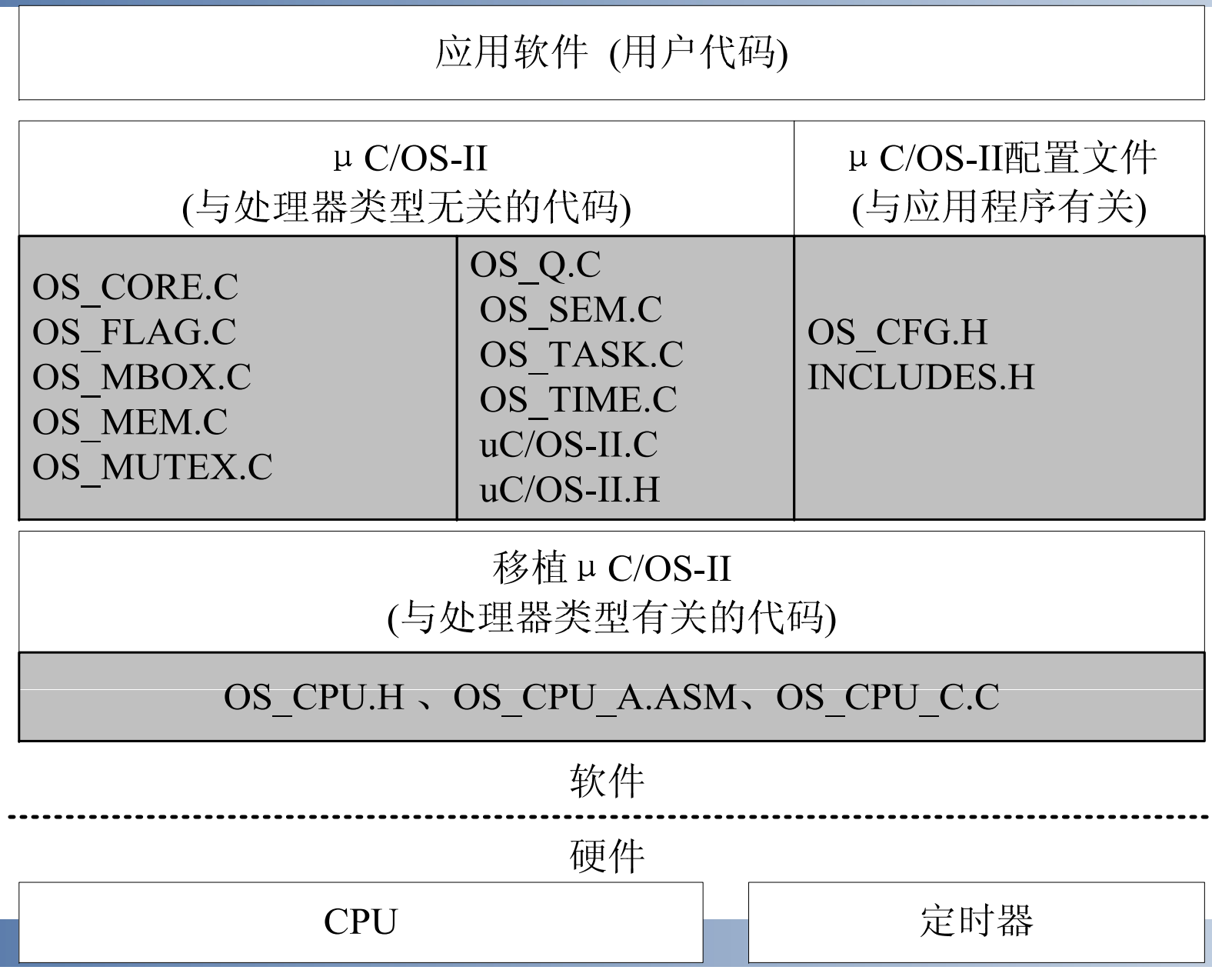
1. uC/OS-II 概述
2. uC/OS-II的任务
3. 任务控制块
4. 任务堆栈
5. 系统任务

uC/OS-II 概述

- **μC/OS——Micro Controller OS**，微控制器操作系统
  - 美国人**Jean Labrosse 1992**年完成
  - 应用面覆盖了诸多领域，如照相机、医疗器械、音响设备、发动机控制、高速公路电话系统、自动提款机等
  - **1998**年出**μC/OS-II**，目前的版本**μC/OS -II V2.61**
  - **2000**年，得到美国航空管理局（**FAA**）的认证，可以用于飞行器中
  - 网站**www.ucos-ll.com**（**www.micrium.com)**

uC/OS-II 概述--文件结构

体系结构



uC/OS-II 概述-性能特点

- 源代码公开
- 可移植（**Portable**）
  - 大部分代码用**ANSI C**写，与处理器无关，移植时不需修改
  - 少量与微处理器硬件相关的部分用**C**与汇编编写，移植时需修改：
    - **OS_CPU.H** //与硬件相关，移植时需修改
    - **OS_CPU_A.ASM** //集中了所有与处理器相关的汇编语言代码
    - **OS_CPU.C** //集中了所有与处理器相关的汇编语言代码

uC/OS-II 概述-性能特点

- 可裁剪（**Scalable**）
  - 可以只使用μ C/OS-II中应用程序需要的那些系统服务。也就是说某产品可以只使用很少几个μ C/OS-II调用，而另一个产品则使用了几乎所有μ C/OS-II的功能，这样可以减少产品中的μ C/OS-II所需的存储器空间（RAM和ROM）。
  - 可剪裁性通过条件编译实现。

uC/OS-II 概述-性能特点

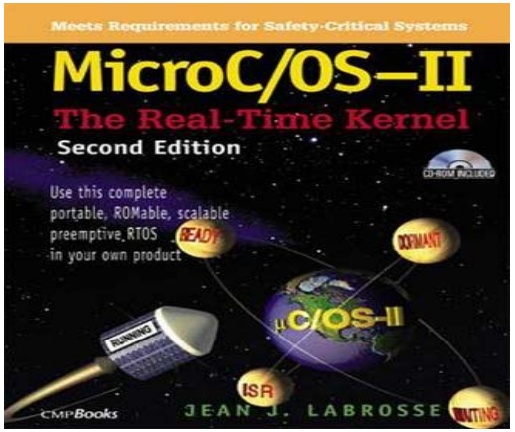
- 可剥夺性（**Preemptive**）与可确定性
  - 内核可剥夺、函数调用或系统服务的执行时间具有可确定性，是硬实时操作系统。
- 支持多任务
  - **μC/OS-II**可以管理**64**个任务
- 任务栈
  - 每个任务有自己单独的栈， **μC/OS-II**允许每个任务有不同的栈空间，以便压低应用程序对**RAM**的需求。

uC/OS-II 概述-性能特点

- 系统服务
  - μC/OS-II提供很多系统服务，例如邮箱、消息队列、信号量、块大小固定的内存的申请与释放、时间相关函数等。
- 中断管理
  - 中断可以使正在执行的任务暂时挂起，如果优先级更高的任务被该中断唤醒，则高优先级的任务在中断嵌套全部退出后立即执行，中断嵌套层数可达255层。

uC/OS-II 概述-图书

- 描述了μC/OS-II内部的工作原理
- 随书的CD中包含了源代码
  - 工业界最清晰的源代码
- 除英文版外，有中文和韩文版



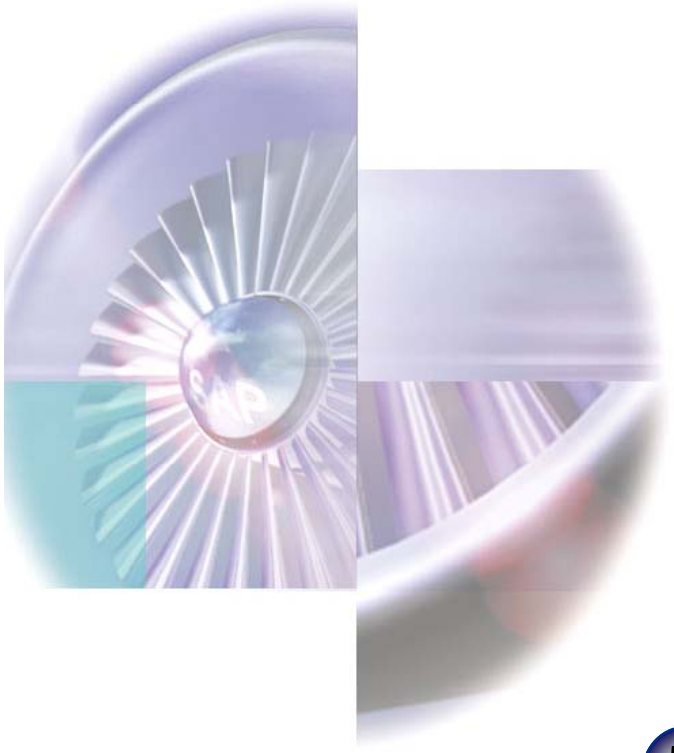
English

ISBN 1-57820-103-9
美国CMP BOOK

Chinese

ISBN 7-81077-290-2
北京航空航天大学出版社

第4章 进程与线程_uCOS的任务



1. uC/OS-II 概述
2. uC/OS-II的任务
3. 任务控制块
4. 任务堆栈
5. 系统任务

uC/OS-II的任务

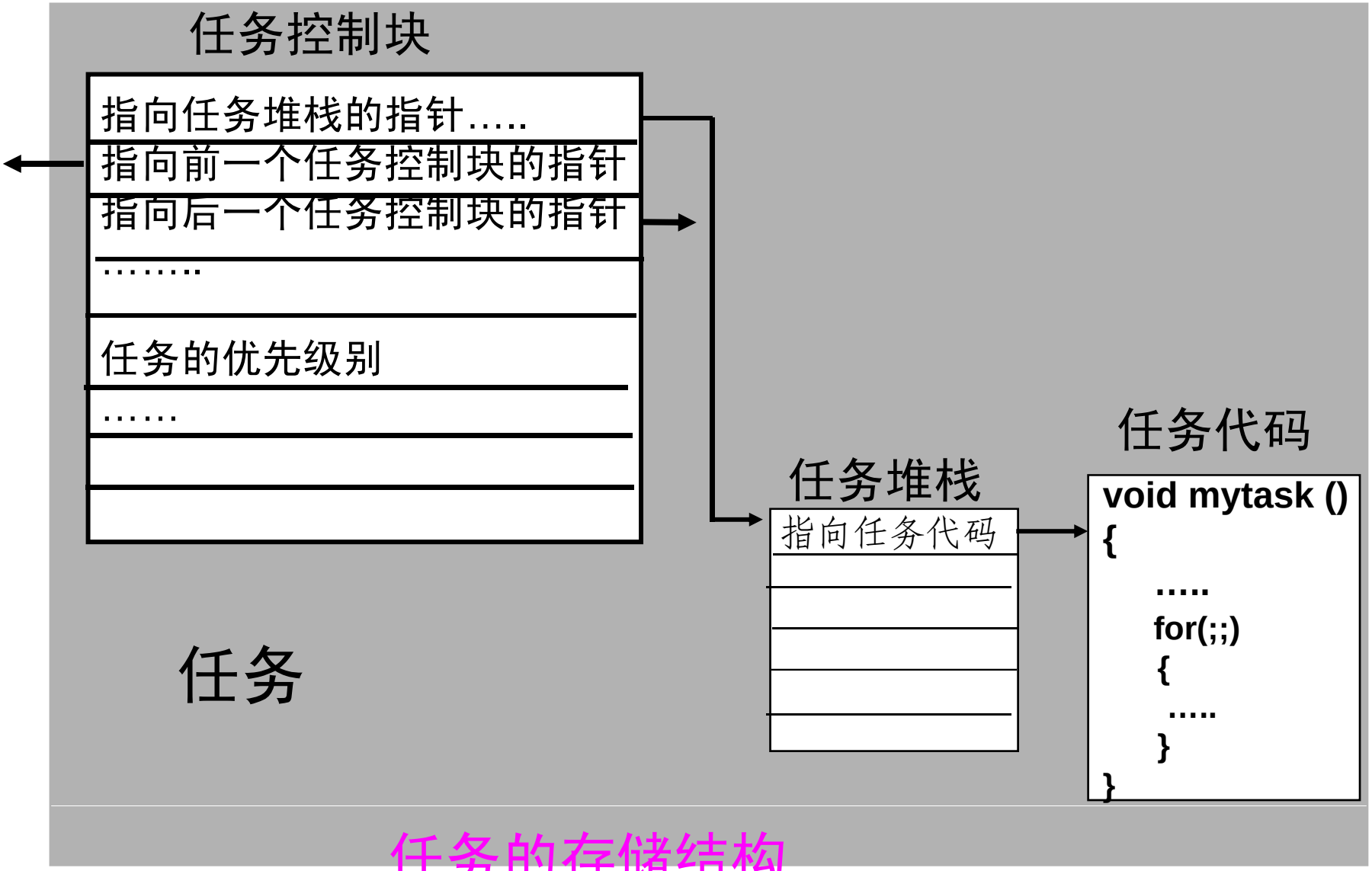
1) 任务代码结构
2) 任务存储结构
3) 任务状态
4) 任务优先级

uC/OS-II的任务—代码结构

- uCOS中的任务是一个线程，其代码通常是一个无限循环结构/超循环结构，看起来像其它C函数一样。

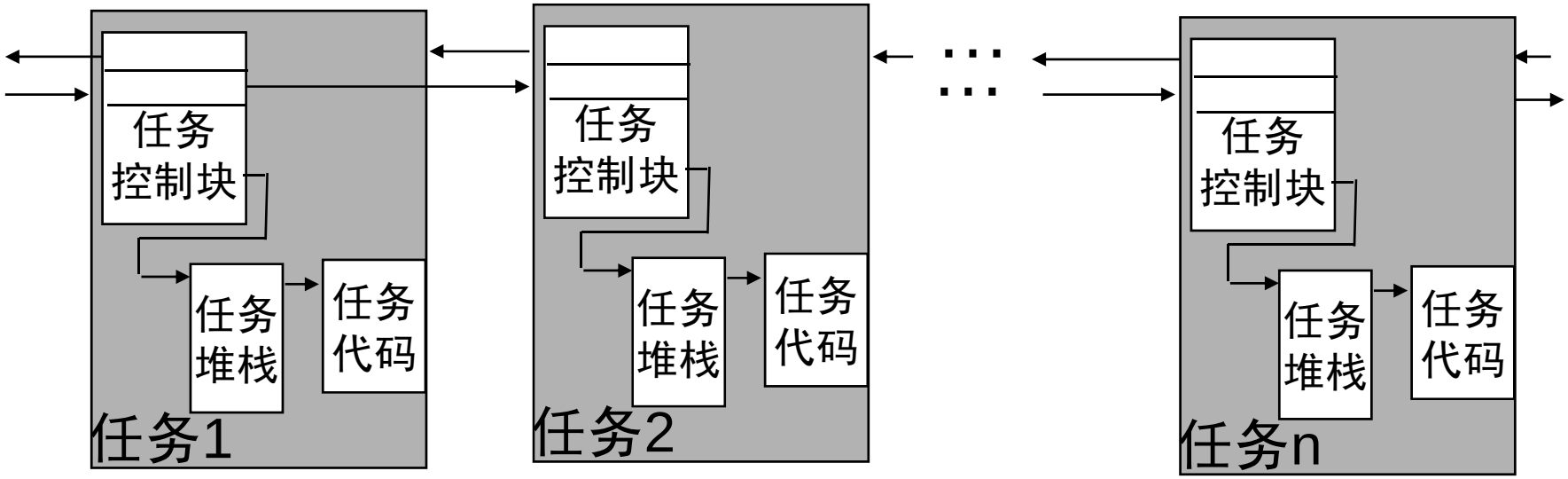
```
void mytask(void *pdata)
{
   for (;;)
   {
   do something;
   waiting;
   do something;
   }
}
```

uC/OS-II的任务--存储结构



任务的存储结构

uC/OS-II的任务--存储结构

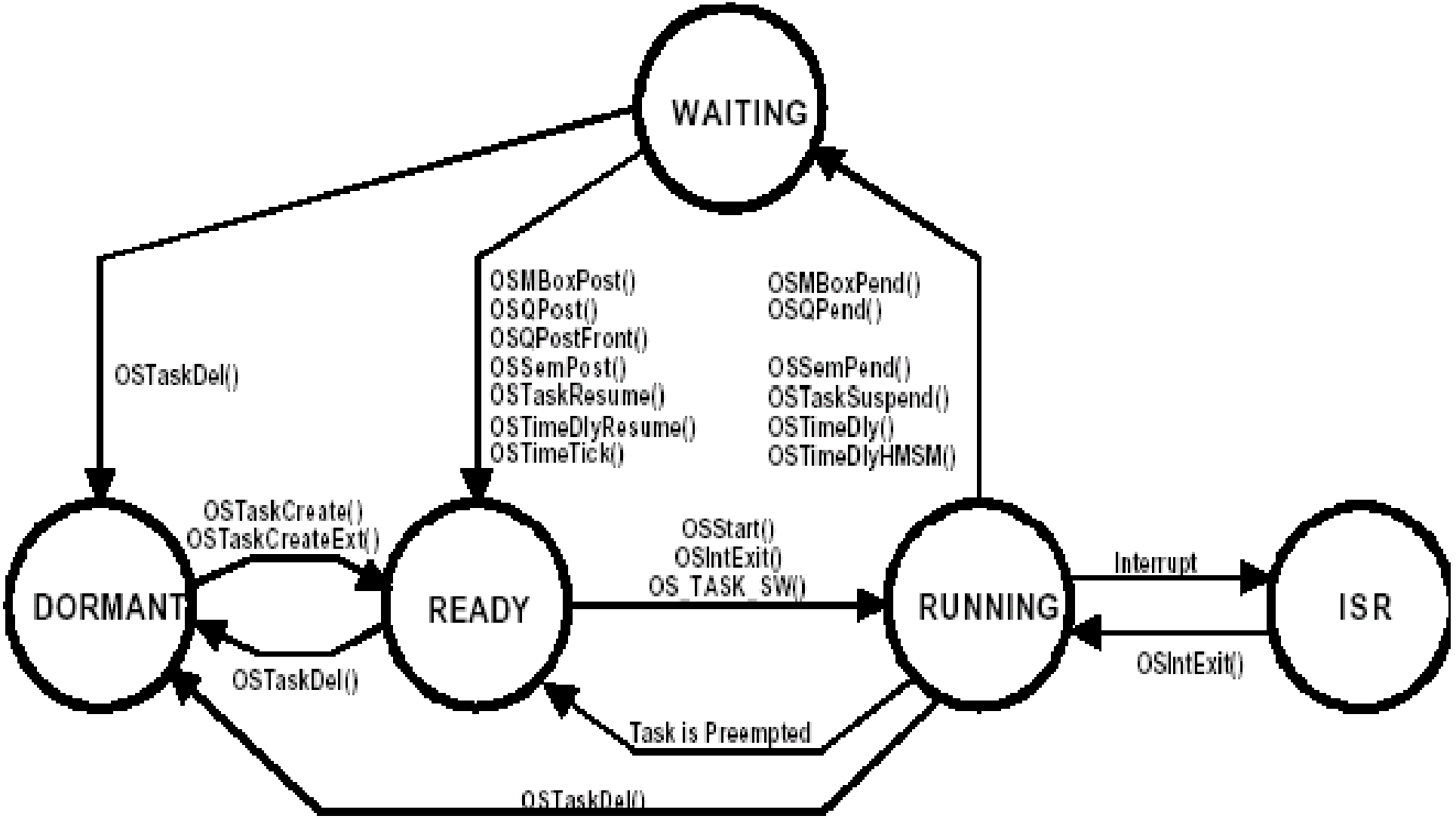


任务控制块链表

uC/OS-II的任务--状态

- uC/OS-II的任务有5种状态
  - 睡眠态（DORMANT）：任务驻留在程序空间，还没有交给uCOS管理,即还没有配备任务控制块,还没有被创建。
  - 就绪态(READY)：任务一旦建立，就进入就绪态准备运行，“万事具备，只欠CPU”。
  - 运行态(RUNNING)：正在使用CPU的状态称运行态。
  - 等待态(WAITING):等待某事件发生的状态.
  - 中断服务态(ISR):正在运行的任务被中断时就进入了中断服务态（ISR）。

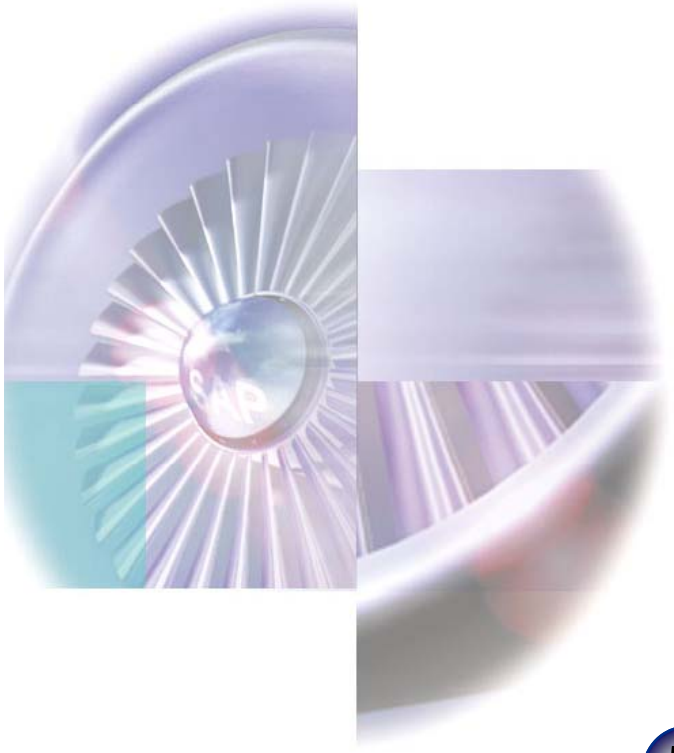
uC/OS-II的任务--状态



uC/OS-II的任务--优先级

- μCOS 支持64个任务，每个任务有一个特定的优先级。
- 任务的优先级别用数字表示，0表示的任务的优先级最高，数字越大表示的优先级越低。
- 通过常数OS_LOWEST_PRIO （在OS_CFG.H中）定义系统的最低优先级别，同时限定系统能容纳的最多任务数量。
- OS_LOWEST_PRIO给空闲任务，OS_LOWEST_PRIO-1给统计任务。

第4章 进程与线程_uCOS的任务



1. uC/OS-II 概述
2. uC/OS-II的任务
3. 任务控制块
4. 任务堆栈
5. 系统任务

任务控制块

1）任务控制块结构

2）任务控制块链表

3）任务控制块初始化

任务控制块--结构

- 任务控制块（Task Control Blocks, OS_TCBs）是μCOS用来存储任务堆栈指针、当前状态、优先级及任务链表指针等属性的一个数据结构。
- 任务控制块是任务的身份证，每个任务都有一个属于自己的任务控制块，当任务的CPU使用权被剥夺时，任务的属性被保存在任务控制块中，而当任务重新得到CPU使用权时，任务控制块能确保任务从当时被中断的那一点丝毫不差地继续执行。
- OS_TCBs全部驻留在RAM中。
- OS_TCBs 在任务建立的时候被初始化.

任务控制块--结构

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;
#if OS_TASK_CREATE_EXT_EN>0
    void            *OSTCBExtPtr;
    OS_STK          *OSTCBStkBottom;
    INT32U           OSTCBStkSize;
    INT16U           OSTCBOpt;
    INT16U           OSTCBId;
#endif
    struct os_tcb *OSTCBNext;
    struct os tcb *OSTCBPrev;
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
|| OS_SEM_EN
    OS_EVENT        *OSTCBEventPtr;
#endif
```

```
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN

    void              *OSTCBMsg;
#endif
    INT16U             OSTCBDly;
    INT8U              OSTCBStat;
    INT8U              OSTCBPrio;
    INT8U              OSTCBX;
    INT8U              OSTCBY;
    INT8U              OSTCBBitX;
    INT8U              OSTCBBitY;
#if OS_TASK_DEL_EN
    BOOLEAN            OSTCBDelReq;
#endif
} OS_TCB;
```

任务控制块--结构

- OSTCBStkPtr：指向当前任务堆栈栈顶的指针。
- OSTCBExtPtr：指向用户定义的任务控制块扩展的指针。用户可以扩展任务控制块而不必修改μCOS-II的源代码,只在函数OstaskCreateExt()中使用,使用时将OS_TASK_CREAT_EN设为1.
- OSTCBStkBottom：指向任务堆栈栈底的指针。递减栈指针指向任务使用的栈空间的最低地址；递增型栈则指向栈空间的最高地址。。

任务控制块--结构

- OSTCBStkSize：堆栈尺寸。
- OSTCBOpt ：OSTaskCreateExt()中的选项，μCOS-Ⅱ目前只支持3个选择项：
  - OS_TASK_OTP_STK_CHK- STK检查
  - OS_TASK_OPT_STK_CLR-清零
  - OS_TASK_OPT_SAVE_FP-浮点运算

任务控制块--结构

- **OSTCBId**：存储任务的识别码。
- OSTCBNext和OSTCBPrev：任务控制块OS_TCBs双向链接，将任务控制块链接起来。
- **OSTCBEventPtr**：指向事件控制块的指针。
- **OSTCBMsg**：指向传给任务的消息的指针

任务控制块--结构

- **OSTCBDly**：任务延时的时钟节拍数。当需要把任务延时若干时钟节拍时，或者需要把任务挂起一段时间以等待某事件的发生时需要用到这个变量。如果这个变量为**0**，表示任务不延时，或者表示等待事件发生的时间没有限制 .

任务控制块--结构

- **OSTCBStat：任务状态字, 可取下列值：**
  - **OS_STAT_RDY:　处于就绪状态**
  - **OS_STAT_SEM：　处于等待信号量状态**
  - **OS_STAT_MBOX：处于等待邮箱状态**
  - **OS_STAT_Q：　处于等待消息队列状态**
  - **OS_STAT_SUSPEND：处于被挂起状态**
  - **OS_STAT_MUTEX：处于等待互斥信号量状态**

任务控制块--结构

- OSTCBPrio：任务优先级。高优先级任务的OSTCBPrio值小，低优先级任务的 OSTCBPrio值大
- OSTCBX、OSTCBY、OSTCBBitX与OSTCBBitY：与优先级有关的量，用于加速任务进入就绪态的过程或进入等待事件发生状态的过程。这些值是在任务建立时算好的，或者是在改变任务优先级时算出的 。

任务控制块--结构

OSTCBX、OSTCBY、OSTCBBitX与OSTCBBitY 的计算

```
OSTCBY    = priority >> 3;

OSTCBBitY  = OSMapTbl[priority >> 3];


OSTCBX     = priority & 0x07;

OSTCBBitX  = OSMapTbl[priority & 0x07];
```

OSMapTbl[]的值

Index	Bit Mask (Binary)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

任务控制块--结构

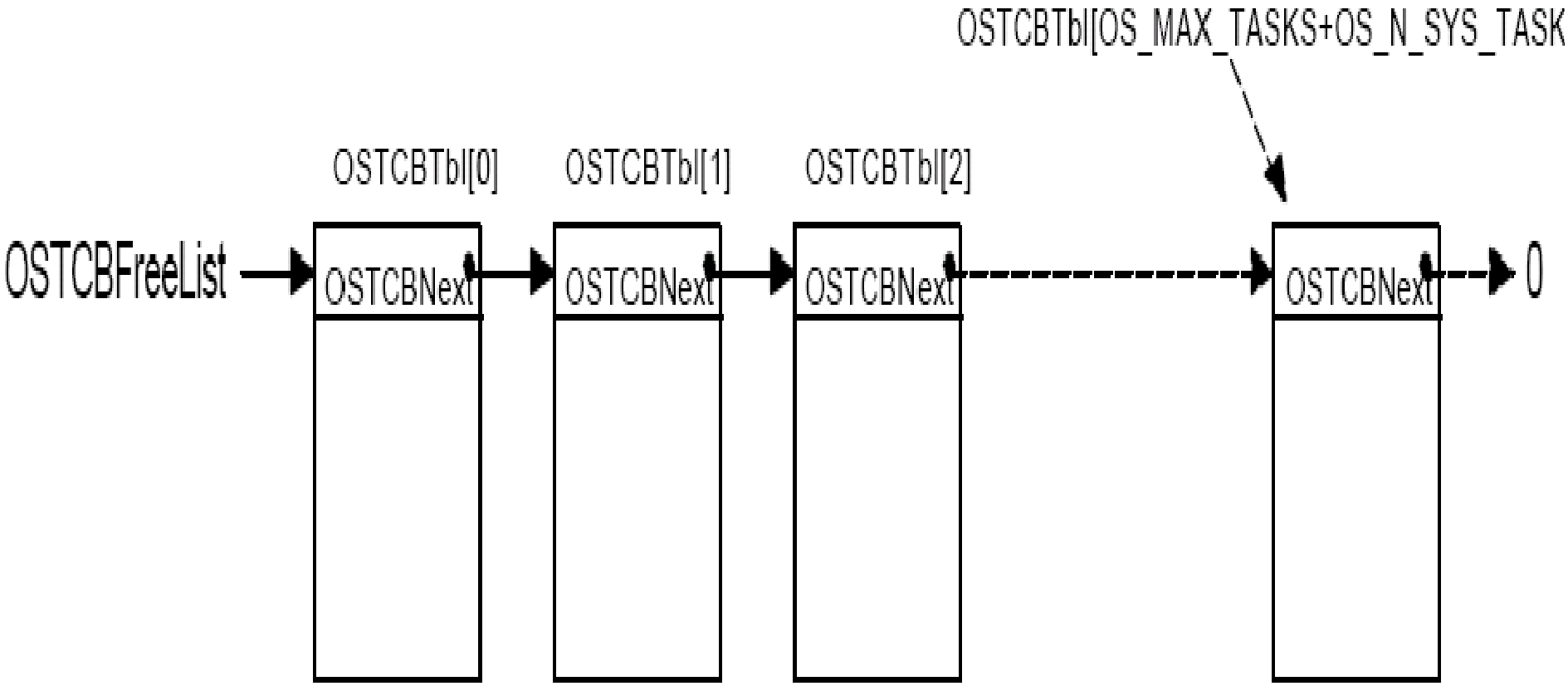
- OSTCBDelReq：一个布尔量，用于表示该任务是否需要删除。

任务控制块--链表

- 任务控制块空闲链表：任务控制块空白链表。
  - 系统初始化时，初始化函数**OSInit()**将创建一个任务控制块空闲缓冲池，其中有一定数量的空任务控制块，并将其链接成一个单向链表，即空闲链表，并用**OSTCBFreeList** 指向表头。
  - 当建立一个任务时，即将空闲链表表头指针**OSTCBFreeList**指向的空任务控制块赋给该任务，然后将**OSTCBFreeList**指向链表中的下一个空任务控制块

任务控制块--链表

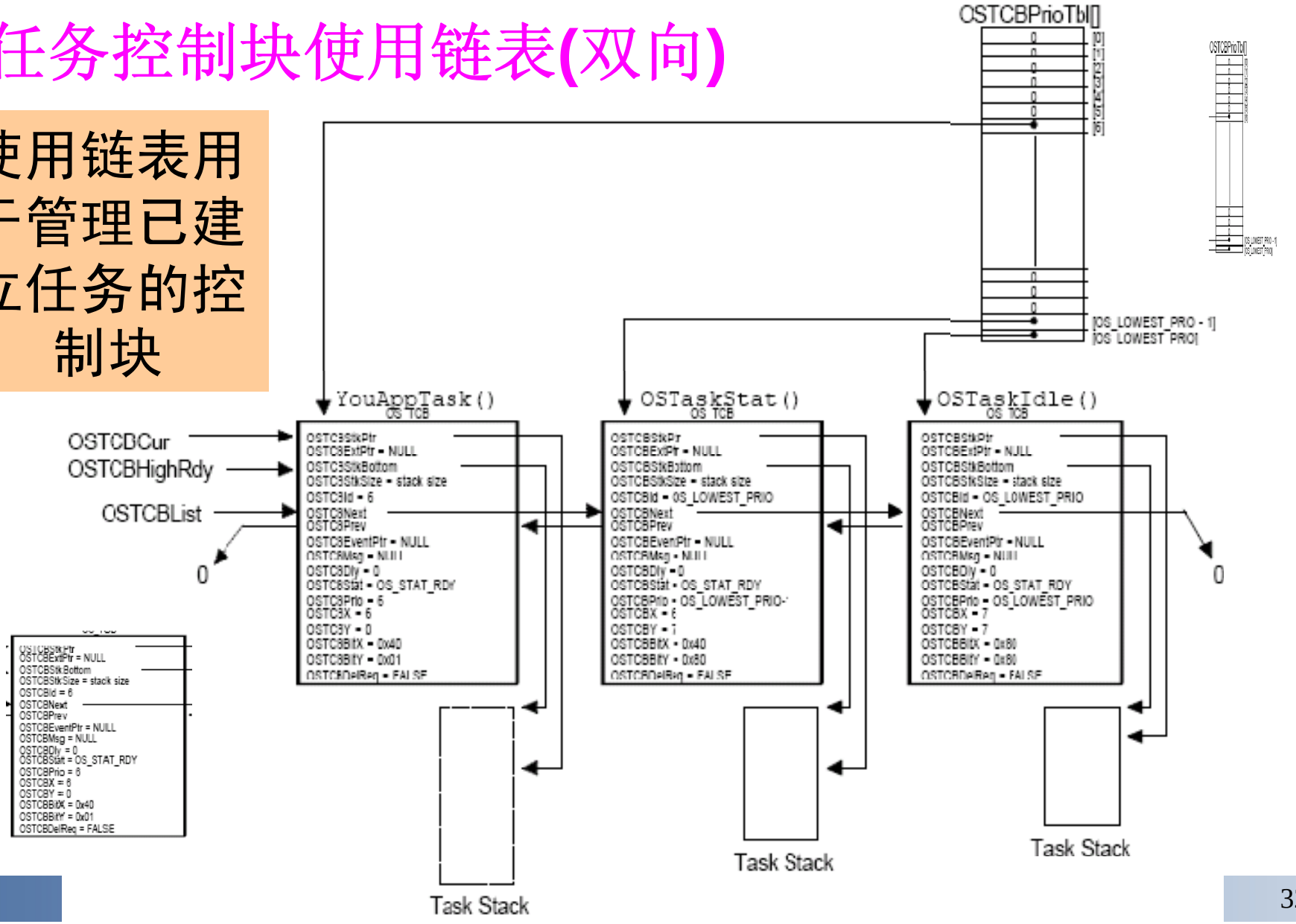
- 任务控制块空闲链表(List of free OS_TCBs)



任务控制块--链表

- 任务控制块使用链表(双向)

使用链表用于管理已建立任务的控制块

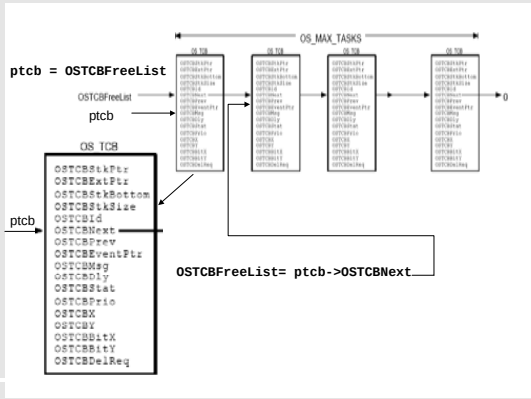


任务控制块--初始化

- 创建任务时，必须创建任务的控制块，通过控制块初始化函数**OSTCBInit()**完成，其做三件事：
  - **1**、从空白/闲任务控制块链表中获取一个任务控制块；
  - **2**、用任务的属性值对任务控制块各个成员进行赋值；
  - **3**、把这个任务控制块链入到任务控制块使用链表的头部。

任务控制块--初始化*OSTCBInit()*

```
INT8U OSTCBInit (INT8U  prio, OS_STK *ptos,  OS_STK *pbos, INT16U id,

        INT16U stk_size, void   *pext,   INT16U  opt)

{

    OS_TCB *ptcb;

    OS_ENTER_CRITICAL();
```




```
        //从空闲的OS_TCB缓冲池中获得一个OS_TCB

    ptcb = OSTCBFreeList;                (1)
```

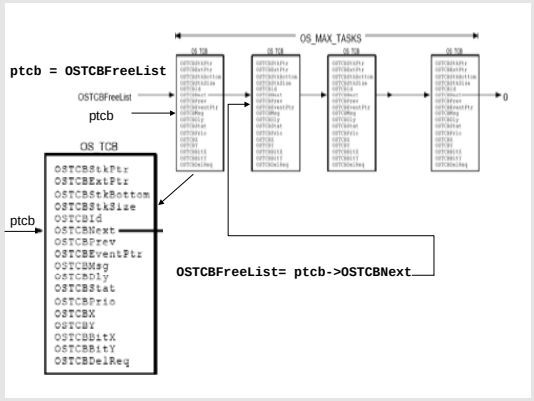
任务控制块--初始化*OSTCBInit()*

//如果OS_TCB池中有空闲的OS_TCB,它就被初始化了

```
if (ptcb != (OS_TCB *)0) {                (2)

    OSTCBFreeList       = ptcb->OSTCBNext;
```

//一旦OS_TCB被分配，该任务的创建者就已经完全拥有它了，不担心被同时建立的另一个任务夺取,故可以重新开中断,并继续初始化OS_TCB的数据单元。

```
    OS_EXIT_CRITICAL();

    ptcb->OSTCBStkPtr    = ptos;          (3)

    ptcb->OSTCBPrio      = (INT8U)prio;

    ptcb->OSTCBStat      = OS_STAT_RDY;

    ptcb->OSTCBDly       = 0;
```



任务控制块--初始化*OSTCBInit()*

```
#if OS_TASK_CREATE_EXT_EN

    ptcb->OSTCBExtPtr    = pext;

    ptcb->OSTCBStkSize   = stk_size;

    ptcb->OSTCBStkBottom = pbos;

    ptcb->OSTCBOpt       = opt;

    ptcb->OSTCBId        = id;
```

任务控制块--初始化*OSTCBInit()*

```
#else
    pext            = pext;
    stk_size        = stk_size;
    pbos            = pbos;
    opt             = opt;
    id              = id;
#endif

#if OS_TASK_DEL_EN
    ptcb->OSTCBDelReq   = OS_NO_ERR;
#endif
```

任务控制块--初始化*OSTCBInit()*

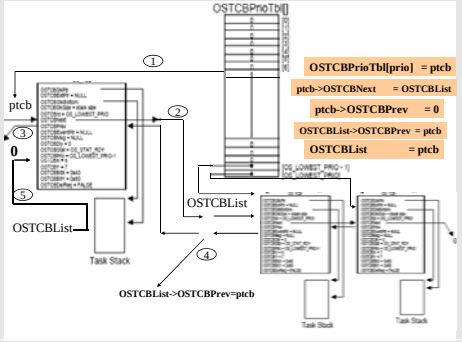
```
    ptcb->OSTCBY        = prio >> 3;

    ptcb->OSTCBBitY     = OSMapTbl[ptcb->OSTCBY];

    ptcb->OSTCBX        = prio & 0x07;

    ptcb->OSTCBBitX     = OSMapTbl[ptcb->OSTCBX];



#if OS_MBOX_EN ||(OS_Q_EN && (OS_MAX_QS >= 2))|| OS_SEM_EN

    ptcb->OSTCBEventPtr  = (OS_EVENT *)0;

#endif
```

任务控制块--初始化*OSTCBInit()*

```
#if   OS_MBOX_EN || (OS_Q_EN && (OS_MAX_QS >= 2))

      ptcb->OSTCBMsg      = (void *)0;

#endif

      OS_ENTER_CRITICAL();          (4)
```

//将OS_TCB插入到已建立任务的OS_TCB的双向链表中,该双向链表开始于OSTCBList，而一个新任务的OS_TCB常常被插入到链表的表头

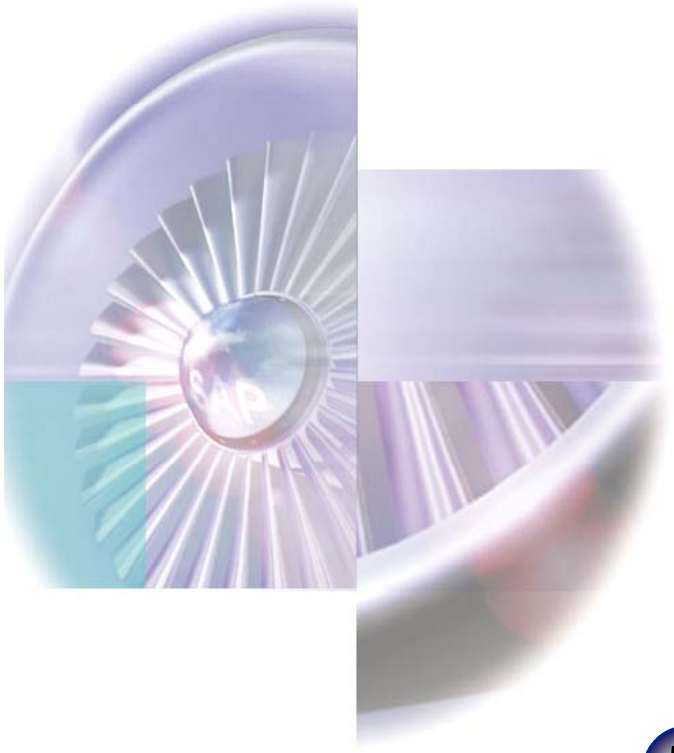
```
      OSTCBPrioTbl[prio]   = ptcb;    (5)

      ptcb->OSTCBNext      = OSTCBList;

      ptcb->OSTCBPrev      = (OS_TCB *)0;

      if (OSTCBList != (OS_TCB *)0) {

         OSTCBList->OSTCBPrev = ptcb;

      }

                              OSTCBList           = ptcb
```



任务控制块--初始化*OSTCBInit()*

```
                                    //使任务进入就绪态
        OSRdyGrp               |= ptcb->OSTCBBitY;   (6)
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        OS_EXIT_CRITICAL();
        return (OS_NO_ERR);                          //返回一个代码表明OS_TCB已经被分配和初始化了
    } else {
        OS_EXIT_CRITICAL();
        return (OS_NO_MORE_TCB);
    }
}
```

第4章 进程与线程_uCOS的任务



1. uC/OS-II 概述
2. uC/OS-II的任务
3. 任务控制块
4. 任务堆栈
5. 系统任务

任务堆栈

1) 堆栈创建
2) 堆栈增长方向
3) 堆栈的初始化

任务堆栈--创建

1）堆栈创建

- 堆栈是在存储器中按数据“后进先出”的原则组织的连续存储空间。用于任务切换和响应中断时保存**CPU**寄存器中的内容及其它任务私有数据。**uCOS**如何创建堆栈？
- **OS_STK MyTaskStack[stack_size];**
- **typedef unsigned int OS_STK； //16位**

任务堆栈--增长方向

2）堆栈增长方向：μCOS支持向上增长(低地址往高地址）及向下增长堆栈。用户在调用OSTaskCreate() 或OSTaskCreateExt() 的时候必须确定堆栈增长方式。

向上增长堆栈（OS_STK_GROWTH=0）

```
OS_STK  TaskStack[TASK_STACK_SIZE];

OSTaskCreate(task,
                    pdata,
                   &TaskStack[0],
                    prio);
```

任务堆栈--增长方向

向下增长堆栈（OS_STK_GROWTH=1）

```
OS_STK  TaskStack[TASK_STACK_SIZE];


OSTaskCreate(task,
             pdata,
             &TaskStack[TASK_STACK_SIZE-1],
              prio);
```

任务堆栈--增长方向

可上下两方向增长堆栈

```
OS_STK  TaskStack[TASK_STACK_SIZE];
```

```
#if OS_STK_GROWTH == 0

    OSTaskCreate(task, pdata, &TaskStack[0], prio);

#else

   OSTaskCreate(task, pdata, &TaskStack[TASK_STACK_SIZE-1],
                             prio);
#endif
```

任务堆栈--初始化

3）任务堆栈的初始化

- 当处理器启动一个任务时，处理器的各寄存器总是需要预置一些与待运行任务相关的初始数据，如指向任务代码的指针、指向任务堆栈的指针、程序状态字**PSW**等，这些初始数据从何而来？
- 系统在创建一个新任务时，应该把启动该任务所需的初始数据（指向任务代码的指针、指向任务堆栈的指针、程序状态字**PSW**等）事先存放到这个任务的堆栈中。
- 任务堆栈初始化函数**OSTaskStkInit()**完成上述工作（其在**OSTaskCreate()**创建任务时被调用）。

任务堆栈--初始化

OSTaskStkInit ()

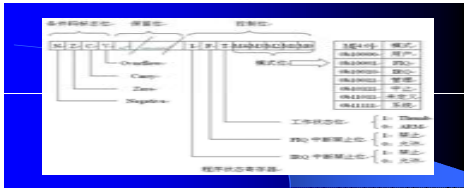
```
OS_STK  *OSTaskStkInit (void (*task)(void *pd), void
  *pdata, OS_STK *ptos, INT16U opt)
{
    unsigned int *stk ;

    opt    = opt;        /* 'opt' is not used, prevent warning */
    stk    = (unsigned int *)ptos;    /* Load stack pointer  */
```

任务堆栈--初始化

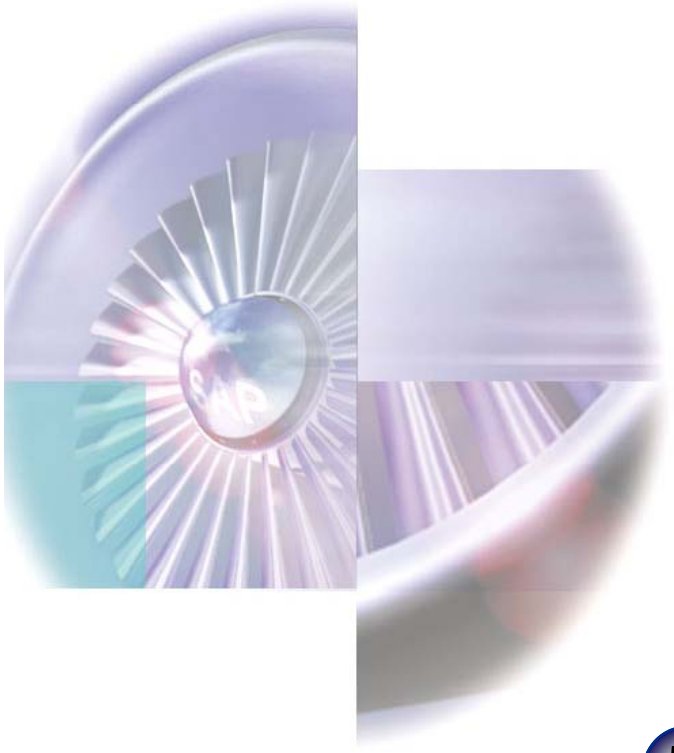
```
/* build a context for the new task */
    *--stk = (unsigned int) task;       /* pc */
    *--stk = (unsigned int) task;       /* lr */
    *--stk = 0;                         /* r12 */
    *--stk = 0;                         /* r11 */
    *--stk = 0;                         /* r10 */
    *--stk = 0;                         /* r9 */
    *--stk = 0;                         /* r8 */
    *--stk = 0;                         /* r7 */
    *--stk = 0;                         /* r6 */
```

任务堆栈--初始化

```
    *--stk = 0;                         /* r5 */
    *--stk = 0;                         /* r4 */
    *--stk = 0;                         /* r3 */
   *--stk = 0;                         /* r2 */
    *--stk = 0;                         /* r1 */
    *--stk = (unsigned int) pdata;      /* r0 */
    *--stk = (SVC32MODE|0x0); /* cpsr  IRQ, FIQ disable*/
    *--stk = (SVC32MODE|0x0);  /* spsr  IRQ, FIQ disable */
   return ((void *)stk);
}
```



第4章 进程与线程_uCOS的任务



1. uC/OS-II 概述
2. uC/OS-II的任务
3. 任务控制块
4. 任务堆栈
5. 系统任务

系统任务 --空闲任务

- 空闲任务OSTaskIdle()
  - μCOS总要建立一个空闲任务，这个任务在没有其它任务进入就绪态时投入运行。这个空闲任务永远设为最低优先级，即OS_LOWEST_PRIO。
  - 空闲任务OSTaskIdle()什么也不做，只是在不停地给一个32位的名叫OSIdleCtr的计数器加1，统计任务使用这个计数器以确定现行应用软件实际消耗的CPU时间。
  - 空闲任务不可能被应用软件删除

系统任务 --空闲任务

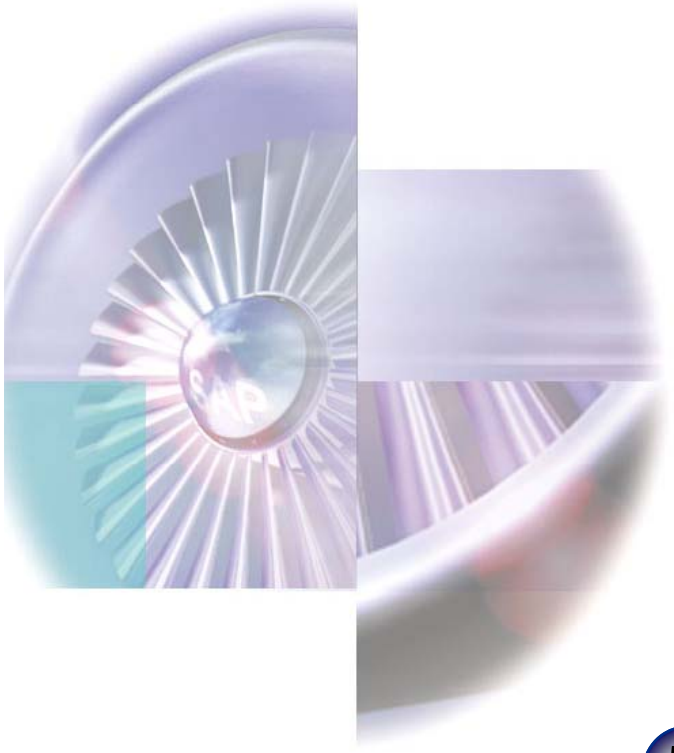
μCOS的空闲任务.

```
void OSTaskIdle (void *pdata)
{
   pdata = pdata;
      for (;;) {
     OS_ENTER_CRITICAL();
      OSIdleCtr++;
      OS_EXIT_CRITICAL();
      OSTaskIdleHook();
      }
}
```

系统任务--统计任务

- 统计任务OSTaskStat()
  - 统计任务是负责统计运行时间的任务，如果用户将系统定义常数OS_TASK_STAT_EN（见文件OS_CFG.H）设为1，这个任务就会建立。一旦得到允许，统计任务每秒运行一次（OS_CORE.C）以计算当前的CPU利用率。也即计算应用程序使用了多少CPU时间，用百分比表示，这个值放在一个有符号8位整数OSCPUsage中，精确度是1个百分点。

第4章 进程与线程_uCOS的任务



1. uC/OS-II 概述
2. uC/OS-II的任务
3. 任务控制块
4. 任务堆栈
5. 系统任务

关于 UCOS 信号量

一：信号量的理解：

（1）信号量可以分为两种：一种是二值信号量（0 和 1），一种是 N 值信号量（计数式信号量）。

二值信号量的意思是可以有多少任务同时享用这个信号量。比如二值信号，就是只有 1 个任务可以使用。当有一个任务使用该信号量的时候，那么其他需要使用该信号量的任务就必须等待，直到该任务释放该信号量。这种信号量可以看作一把钥匙。

对于 N 值信号量（计数式信号量），就是说可以同时有 N-1 个任务同时使用该信号量。对于二值信号量，N=1。

（2）建立信号量的工作必须在任务级代码中或者多任务启动之前完成。

二：任务如何得到信号量的问题：

想得到信号量的任务，必须执行等待操作（pend）。在信号量的建立的时候，我们首先确定了该信号量可以被共享的资源数（N），并将其赋值给 pevent->OSEventCnt。如果信号量有效(非 0)，即 pevent->OSEventCnt>0，则信号量减 1，任务得以继续运行。如果信号量无效，即 pevent->OSEventCnt==0，则等待信号量的任务就被列入等待信号量的任务表中。许多内核允许定义等待超时，当等待时间超过了设定值，该信号量还是无效，则等待该信号量的任务进入就绪态，准备运行，并返回出错代码(等待超时错误)。

三：任务对信号量的释放问题：

任务执行发信号（post）操作来释放信号量。如果没有任务等待信号量，那么信号量的值仅是简单的加 1（则信号量大于 0，有效）；如果有任务等待该信号量，那么就会有另一个任务进入就绪态，信号量的值就不加 1。

之后，这个释放的信号量给那个等待中的任务，要看内核如何调度的。收到信号量的任务可能是如下两者之一：

◆等待任务中，优先级最高的；（uc/os-ii 仅支持这种方式）。

◆最早开始等待信号量的任务（如果是按先进先出 FIFO 原则）。

四：信号量的有效与无效的问题：

信号量有效：信号量的计算器非 0（.OSEventCnt！=0）。信号量有效表示任务对资源可用。

信号量无效：信号量的计算器为 0。信号量无效表示任务对目前资源不可用，需要等待其他另一个任务（或者中断服务子程序）发出该信号量（OSSemPost）。

五：关于信号量的三个重要函数：

◆OSSemCreate() 创建一个信号量 （注：由任务或启动代码操作）

创建工作必须在任务级代码中或者多任务启动之前完成。功能只要是先获取一个事件控制块 ECB，写入一些参数。其中调用了 OS_EeventWaitListInt()函数，对事件控制块的等待任务列表进行初始化。完成初始化工作后，返回一个该信号量的句柄(Handle)。

◆OSSemPend() 等待一个信号量 （注：只能由任务操作）

本函数应用于任务试图获得共享资源的使用权、任务需要与其他任务或中断同步及任务需要等待特定事件发生的场合。

如果任务 Task_A 调用 OSSemPend()，且信号量的值有效(非 0)，那么 OSSemPend()递减信号量计数器（.OSEventCnt），并返回该值。换句话说，Task_A 获取到共享资源的使用权了，之后就执行该资源。

如果如果任务 Task_A 调用 OSSemPend()，信号量无效(为 0)，那么 OSSemPend()调用 OS_EventTaskWait()函数，把 Task_A 放入等待列表中。（等待到什么时候呢？要看 OSSemPost()(或者等待超时情况)，由它释放信号量并检查任务执行权，见下资料）

◆OSSemPost() 发出（释放）一个信号量 （注：由任务或中断操作）

本函数其中调用 OS_EventTaskRdy()函数，把优先级最高的任务 Task_A（在这假如是 Task_A，另外假设当前调用 OSSemPost()的任务是 Task_B）从等待任务列表中去除，并使它进入就绪态。然后调用 OSSched()进行任务调度。如果 Task_A 是当前就绪态中优先级最高的任务，则内核执行 Task_A；否则，OSSched()直接返回，Task_B 继续执行。

UCOS 另类信号量--互斥信号量

在 UCOS 的信号量使用过程中，我们经常会用的是二值信号量，而在二值信号两种用的醉的情况就是互斥信号量。互斥信号是本身是一种二进制信号，具有超出 uCOS－II 提供的一般信号机制的特性。由于其特殊性，UCOS 的作者将其独立成章，单独对待。组织了一套对于互斥信号量管理的单独函数。互斥信号量具有以下特点： 1） 降解优先级反转。 2） 实现对资源的独占式访问（二值信号量）。

在应用程序中使用互斥信号是为了减少优先级翻转问题，当一个高优先级的任务需要的资源被一个低优先级的任务使用时，就会发生优先级翻转问题。为了减少优先级翻转问题，内核可以提高的优先级任务的优先级，先于高优先级的任务运行，释放占用的资源。

为了实现互斥，实时内核需要具有支持在同一优先级具有多个任务的能力。不幸的是，UC/OS-II 不允许在相同的优先级有多个任务，必须只有一个任务。但是我们有另外的方法解决这个问题。可以把需要资源的高优先级任务上面的一个任务使用 Mutex 保留，允许提高的优先级任务的优先级。

举一个 mutexes 信号工作的例子，如 l 下面的程序所示。

其中有三个任务可以使用共同的资源，为了访问这个资源，每个任务必须在互斥信号 ResourceMutex 上等待（pend）,任务＃1 有最高优先级 10，任务＃2 优先级为 15，任务＃3 优先级为 20，一个没有使用的正好在最高优先级之上的优先级＃9 用来作为优先级继承优先级。如 main()所示，代码中(1)进行 uC/OS-II 初始化，并通过调用 OSMutexCreate()代码中(2)创建了一个互斥信号。需要注意的是，OSMutexCreate()函数使用 PIP 最为参数。然后创建三个任务代码中(3)，启动 uC/OS-II 代码中(4).

假设任务运行了一段时间，在某个时间点，任务#3 最先访问了共同的资源，并得到了互斥信号，任务＃3 运行了一段时间后被任务＃1 抢占。任务＃1 需要使用这个资源，并通过调用 OSMutexPend()企图获得互斥信号，这种情况下，OSMutexPend()会发现一个高优先级的任务需要这个资源，就会把任务＃3 的优先级提高到 9，同时强迫进行上下文切换退回到任务＃3 执行。任务＃3 可以继续执行然后释放占用的共同资源。任务＃3 通过调用 OSMutexPost()释放占用的 mutex 信号，OSMutexPost()会发现 mutex 被一个优先级提升的低优先级的任务占有，就会把任务＃3 的优先级返回到 20。把资源释放给任务＃1 使用，执行上下文切换到任务＃1

```
OS_EVENT *ResourceMutex;
OS_STK TaskPrio10Stk[1000];
OS_STK TaskPrio15Stk[1000];
OS_STK TaskPrio20Stk[1000];
void main (void)
{
INT8U err;
OSInit(); /* (1) */
/* ----------  应用程序初始化  ---------- */
OSMutexCreate(9, &err); /* (2) */
OSTaskCreate(TaskPrio10, (void *)0, &TaskPrio10Stk[999], 10); /* (3) */
```

```
OSTaskCreate(TaskPrio15, (void *)0, &TaskPrio15Stk[999], 15);
OSTaskCreate(TaskPrio20, (void *)0, &TaskPrio20Stk[999], 20);
/* ---------- Application Initialization ---------- */
OSStart(); /* (4) */
}
void TaskPrio10 (void *pdata)
{
INT8U err;
pdata = pdata;
while (1) {
/* ---------  应用程序代码  ---------- */
OSMutexPend(ResourceMutex, 0, &err);
/* -------  访问共享资源  ------ */
OSMutexPost(ResourceMutex);
/* ---------  应用程序代码  ---------- */
}
}
void TaskPrio15 (void *pdata)
{
INT8U err;
pdata = pdata;
while (1) {
/* ---------应用程序代码  ---------- */
OSMutexPend(ResourceMutex, 0, &err);
/* -------  访问共享资源  ------ */
OSMutexPost(ResourceMutex);
/* ---------  应用程序代码  ---------- */
}
}
void TaskPrio20 (void *pdata)
{
INT8U err;
pdata = pdata;
while (1) {
/* ---------应用程序代码---------- */
OSMutexPend(ResourceMutex, 0, &err);
/* -------访问共享资源------ */
OSMutexPost(ResourceMutex);
/* ---------应用程序代码---------- */
}
}
```

上面代码为互斥信号使用示例

uC/OS-II'互斥信号包含三个元素，一个 flag 表示当前 mutex 是否能够获得（0 或 1）；一个 priority 表示使用这个 mutex 的任务，以防一个高优先级的任务需要访问 mutex；还包括一个等待这

个 mutex 的任务列表。
为了启动 uC/OS-II’s mutex 服务，应该在 OS_CFG.H 中设置 OS_MUTEX_EN=1。在使用一个互斥信号之前应该首先创建它，创建一个 mutex 信号通过调用 OSMutexCreate()完成，mutex 的初始值总是设置为 1，表示资源可以获得。
uC/OS-II 提供了六种访问互斥信号量的操作 OSMutexCreate(), OSMutexDel(),OSMutexPend(), OSMutexPost(), OSMutexAccept() and OSMutexQuery().
展示了任务和互斥信号量的关系。一个互斥信号量只能被任务访问。在以后的资料中使用钥匙符号表示互斥信号。钥匙符号表明互斥信号用来访问共享资源。没有钥匙就无法访问。只有得到钥匙的任务才有资格访问共享资源

UCOS 互斥信号量操作函数分析

```
//建立并初始化一个互斥型信号量(优先级继承优先级(PIP)、出错代码指针)
OS_EVENT *OSMutexCreate (INT8U prio, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3 /* Allocate storage for CPU status register */
OS_CPU_SR cpu_sr;
#endif
OS_EVENT *pevent;
if (OSIntNesting > 0) { /* See if called from ISR ... */
*err = OS_ERR_CREATE_ISR; /* can''t CREATE mutex from an ISR */
return ((OS_EVENT *)0);
}//不能从 ISR 中建立，不允许在 ISR 中调用此函数
#if OS_ARG_CHK_EN > 0
if (prio >= OS_LOWEST_PRIO) { /* Validate PIP */
*err = OS_PRIO_INVALID;
return ((OS_EVENT *)0);
}//不合理的 PIP
#endif
OS_ENTER_CRITICAL();
if (OSTCBPrioTbl[prio] != (OS_TCB *)0) { /* Mutex priority must nalready exist*/
//确认 PIP 没有被任何任务占用。OSTCBPrioTbl[ ]中的一个指向 NULL 的空指针指示//PIP 有效
OS_EXIT_CRITICAL(); /* Task already exist at priority ... */
*err = OS_PRIO_EXIST; /* ... inheritance priority */
//如果优先级存在 ，则出错。
return ((OS_EVENT *)0);
}
OSTCBPrioTbl[prio] = (OS_TCB *)1; /* Reserve the table entry */
//置非空指针，将这个优先级保留下来。
pevent = OSEventFreeList; /* Get next free event control block */
//从空余 ECB 中得到一块空的 ECB。
if (pevent == (OS_EVENT *)0) { /* See if an ECB was available */
//看 ECB 是否可用
```

```
OSTCBPrioTbl[prio] = (OS_TCB *)0; /* No, Release the table entry */
//如果不可用，释放此优先级表入口
OS_EXIT_CRITICAL();
*err = OS_ERR_PEVENT_NULL; /* No more event control blocks */
return (pevent);
}
OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr; /* Adjust the free list
//如果可用，重新调整事件控制块的表头
OS_EXIT_CRITICAL();
pevent->OSEventType = OS_EVENT_TYPE_MUTEX; //将其标记为互斥型信号量
pevent->OSEventCnt = (prio << 8) | OS_MUTEX_AVAILABLE;/* Resource is available */ // (#define
OS_MUTEX_AVAILABLE 0x00FF)
//mutex 为有效值，同时将 PIP 保存起来。值得注意的是，事件计数器.OSEventCnt
//在此处的用法不同，高八位用于保存 PIP 的值，低侘位在资源无任务占用
//时的值为 0xff，有任务占用时为占用 mutex 任务的优先级。这个避免了增加额
//外的空间，节约对 RAM 的占用量
pevent->OSEventPtr = (void *)0; /* No task owning the mutex */
//消息正在初始化，所以没有等待这个 mutex 的任务
OS_EventWaitListInit(pevent);//初始化事件等待列表
*err = OS_NO_ERR;
return (pevent);

}
```

PIP 是该函数的参数，指定优先级继承优先级。当发生优先级反转时，将占用该 mutex 的任务的优先级太高到 PIP。

UCOS 互斥信号量操作函数分析

等待（申请）一个互斥信号量：OSMutexPend（）

关键代码剖析：

```
void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
//不得在中断中调用该函数
if (OSIntNesting > 0) {
*err = OS_ERR_PEND_ISR;
return;
}
OS_ENTER_CRITICAL();
//信号量可用
if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8)
== OS_MUTEX_AVAILABLE) {
pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
//将计数器低 8 为置成占用该 mutex 的任务（当前任务）的优先级。
```

```
pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
//在 mutex 中保存占用信号量的任务：修改该 mutex 的 OSEventPtr ，使其指向当前任务
pevent->OSEventPtr = (void *)OSTCBCur;
OS_EXIT_CRITICAL();
//信号量可用，正常返回。
*err = OS_NO_ERR;
return;
}
//信号量不可用：即已被占用
//从该信号量中获得 PIP
pip = (INT8U)(pevent->OSEventCnt >> 8);
//从该信号量中获得占用该信号量的任务的优先级。
mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
//从信号量中获得占用该信号量的任务
ptcb = (OS_TCB *)(pevent->OSEventPtr);
/*
如果原先占用该 mutex 的优先级比提出申请该 mutex 的任务的优先级低
（mprio > OSTCBCur->OSTCBPrio），则提升原任务的优先级至 PIP
*/
if (ptcb->OSTCBPrio != pip && mprio > OSTCBCur->OSTCBPrio) {
if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {
if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
OSRdyGrp &= ~ptcb->OSTCBBitY;
}
//若原任务已就绪，则将其从就绪表中删除，并置就绪标志 rdy
rdy = TRUE;
} else {
rdy = FALSE;
}
//修改优先级，及相关参数
ptcb->OSTCBPrio = pip;
ptcb->OSTCBY = ptcb->OSTCBPrio >> 3;
ptcb->OSTCBBitY = OSMapTbl[ptcb->OSTCBY];
ptcb->OSTCBX = ptcb->OSTCBPrio & 0x07;
ptcb->OSTCBBitX = OSMapTbl[ptcb->OSTCBX];
//如果原任务是就绪的，则继续让新的优先级就绪
if (rdy == TRUE) {
OSRdyGrp |= ptcb->OSTCBBitY;
OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
}
OSTCBPrioTbl[pip] = (OS_TCB *)ptcb;
}
//让提出申请的任务先等待（从就绪表中删除，如 mutex 的等待队列）……
OSTCBCur->OSTCBStat |= OS_STAT_MUTEX;
```

```
OSTCBCur->OSTCBDly = timeout;
OS_EventTaskWait(pevent);
OS_EXIT_CRITICAL();
//执行任务切换（如果原来低优先级的任务优先级被抬高了，则该任务将被执行）
OS_Sched();
OS_ENTER_CRITICAL();
//提出申请的任务被唤醒继续执行
if (OSTCBCur->OSTCBStat & OS_STAT_MUTEX) {
//1）由于等待超时被定时器唤醒
OS_EventTO(pevent);
OS_EXIT_CRITICAL();
*err = OS_TIMEOUT;
return;
}
/*
2）原先占用 mutex 的任务执行完成释放了 mutex
并唤醒了等待该 mutex 的最高优先级的任务
*/
OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
OS_EXIT_CRITICAL();
*err = OS_NO_ERR;
}
```

释放一个互斥信号量

释放一个互斥信号量：OSMutexPost（）
关键代码剖析：

```
INT8U OSMutexPost (OS_EVENT *pevent)
{
//不得在中断中调用该函数
if (OSIntNesting > 0) {
return (OS_ERR_POST_ISR);
}
OS_ENTER_CRITICAL();
//从该信号量中获得 PIP
pip = (INT8U)(pevent->OSEventCnt >> 8);
/*
从该信号量中获得占用该信号量的任务的优先级。
在 OSEventCnt 中的低 8 位保存占用 mutex 的任务的原始优先级，
不随优先级的提高而改变。
*/
prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
/*
```

```
确认释放 mutex 的任务确实是占用 mutex 的任务自身。
占用/申请 mutex 的任务的优先级可能是 pip(被提高)，也可能是原先任务的优先级。
*/
if (OSTCBCur->OSTCBPrio != pip &&
OSTCBCur->OSTCBPrio != prio) {
OS_EXIT_CRITICAL();
//若释放 mutex 的任务非占用/申请的任务，则返回错误信息。
return (OS_ERR_NOT_MUTEX_OWNER);
}
//若当前释放 mutex 的任务的优先级为 pip，则需将该任务的优先级降到原来水平
if (OSTCBCur->OSTCBPrio == pip) {
//首先将 pip 从就绪表删除
if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
}
//将任务优先级修改为原始优先级，并修改相关参数
OSTCBCur->OSTCBPrio = prio;
OSTCBCur->OSTCBY = prio >> 3;
OSTCBCur->OSTCBBitY = OSMapTbl[OSTCBCur->OSTCBY];
OSTCBCur->OSTCBX = prio & 0x07;
OSTCBCur->OSTCBBitX = OSMapTbl[OSTCBCur->OSTCBX];
//将修改优先级后的任务重新如就绪表
OSRdyGrp |= OSTCBCur->OSTCBBitY;
OSRdyTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
OSTCBPrioTbl[prio] = (OS_TCB *)OSTCBCur;
}
OSTCBPrioTbl[pip] = (OS_TCB *)1;
//若 mutex 的等待列表不空，唤醒等待列表中最高优先级的任务，并将 mutex 分配给它
if (pevent->OSEventGrp != 0x00) {
/*
唤醒等待列表中最高优先级的任务（从 mutex 等待列表中删除，使其入就绪表），
清除等待任务的 OS_STAT_MUTEX 标志，并返回其优先级 prio
*/
prio = OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
//将 mutex 分配给新任务：置 OSEventCnt 为 prio
pevent->OSEventCnt |= prio;
//在 mutex 中保存占用信号量的任务
pevent->OSEventPtr = OSTCBPrioTbl[prio];
OS_EXIT_CRITICAL();
//任务切换（如果唤醒的任务优先级比当前任务高，则使唤醒的任务得到运行）
OS_Sched();
return (OS_NO_ERR);
}
```

```
//mutex 的等待列表为空，即该 mutex 可用：置 mutex 可用标志及占用任务指针。
pevent->OSEventCnt |= OS_MUTEX_AVAILABLE;
pevent->OSEventPtr = (void *)0;
OS_EXIT_CRITICAL();
return (OS_NO_ERR);
}
```

UCOS 事件标志组管理

今天我们就看看事件标志组的使用和管理吧
事件标志组（event flag）包含两部分：

```
typedef struct
{
INT8U OSFlagType;
void *OSFlagWaitList;
OS_FLAGS OSFlagFlags;
}OS_FLAG_GRP;
```

1 组中各事件状态的标志位

2 等待这些标志位或清除的任务列表

（这里是双向链表） 用于删除标志时检查是否有等待该
标志的任务链表包含 3 个数据结构： OS_FLAG_GRP, OS_TCB,OS_FLAG-NODE 用来记录任务在等待哪些标志位及等待方式（与/或），当一个任务开始等待某些标志位时建立一个 OS_FLAG-NODE，当这些等待的事件标志位发生后，删除数据结构。
当某个任务需要与多个任务同步时，须要使用事件标志组。
1.弄清楚 OS_FLAG_GRP、OS_FLAG_NODE 和 OS_TCB 之间的关系。
当一个任务开始等待某些事件标志位时，就回建立一个事件标志节点 OS_FLAG_NODE 数据结构，并且将任务所要等待的事件标志位写入 OS_FLAG_NODE 的分量.OSFlagNodeFlags。然后将该数据结构分量.OSFlagNodeFLagGrp 指向事件标志组 OS_FLAG_GRP，将.OSFlagNodeTCB 指向该任务的控制块 OS_TCB，建立起任务与事件标志组之间的联系，说明该任务是等待该事件标志组中某些事件标志位的任务。当有多个任务都需要等待某个事件标志组中某些事件标志位时，这些任务分别建立自己的事件标志节点。并且将这些事件标志节点通过分量.OSFlagNodeNext 和.OSFlagNodePrev 连接成链。
2.任务可以等待事件标志组中某些位置位 1，也可以等待事件标志组中某些位清 0，而置 1（或清 0）又可以分为所有事件都发生的“与”型和任何一个事件发生的“或”型。这样便有了 4 种不同的类型存放在.OSFlagNodeWaitType（OS_FLAG_NODE）中。
3.事件标志组和信号量我觉得是有不同的。
信号量建立以后，假设初始值为 N，前 N 个任务调用 OSSemPend（）函数都会得到信号量。之后如果第Ｎ＋１个任务调用 OSSemPend（）函数申请信号量，该任务将会被置为等待事件发生的状态（睡眠态）。只到前 N 个任务中有任务运行完了所要运行的程序，调用 OSSenmPost（）函数，释放了所占用了信号量，第 N+1 个任务。（这里假设该任务是所有等待信号量任务中优先级最高的任务）才会获得信号量，被从睡眠态转入就绪态。
而事件标志组是事件标志组建立之后，某个任务需要事件标志组中某些事件标志位（置位或者清 0）才能继续运行，于是任务调用 OSFlagPend（）函数，而此时若这些标志位满足要求，

任务返回，继续执行。否则，任务将被挂起。而当有另外一个任务调用 OSFlagPost（）函数将前一个任务所需要的标志位（置位或清 0）使之满足要求，前一个被挂起的任务将被置为就绪态。因此几个任务可以同时得到所需要的事件标志进入就绪态。注意：只要任务所需要的标志位满足要求，任务便进入就绪态。与信号量不同，信号量中的任务需要是在等待该信号量中优先级最高的任务才能得到信号量进入就绪态。事件标志组可以一个任务与多个任务同步，而信号量只能是一个任务与另一个任务同步。

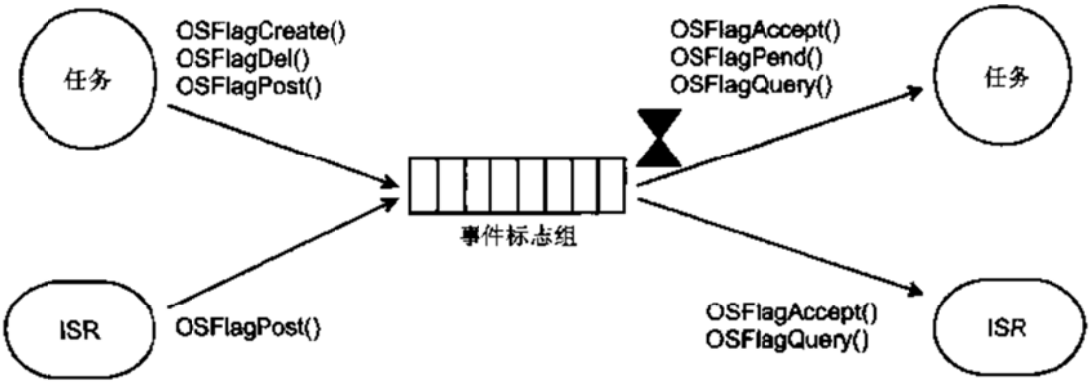


图 F9.1　μC/OS－II 事件标志组功能

**********。

等讲完邮箱和消息队列后然后对 UCOS 的内部通信机制做一个总结，然后就该讲关于移植的过程了

考虑了好多方法。最后还是在有问必答的帖子上有个坛友要 51 下移植好的 UCOS 源码+在 Proteus 下仿真。我觉得这道是一个好的办法

所以我决定这几天抽出点时间移植一个 UCOS 到 STC51 单片机上。使用 Keil4.0+Proteus 仿真。作为大家共同学习移植的平台。

基本外设如下：

输出设备：液晶屏（主要为以后讲解 UCGUI 做准备）+串口

输入设备：4*4 键盘

其它：DS18B20 温度采集+LED 跑马灯

大家有好的主意可以提出来，咱们争取做一个完善的开发板。用于学习 UCOS+UCGUI