Meets Requirements for Safety-Critical Systems

嵌入式实时操作系统

μC/OS-II（第2版）

MicroC/OS-II The Real-Time Kernel

Second Edition

[美] Jean J. Labrosse 著

邵贝贝 等译

CD-ROM INCLUDED

CMPBooks

北京航空航天大学出版社

http://www.buaapress.com.cn

嵌入式实时操作系统
μC/OS－II
（第2版）

MicroC /OS－II The Real-Time Kernel
Second Edition

［美］Jean J. Labrosse　著
邵贝贝　等译

北京航空航天大学出版社
http://www.buaapress.com.cn

内 容 简 介

μC/OS-II是著名的、源码公开的实时内核，是专为嵌入式应用设计的，可用于各类8位、16位和32位单片机或DSP。从μC/OS算起，该内核已有10余年应用史，在诸多领域得到了广泛应用。

本书是*MicroC/OS-II The Real Time Kernel*一书的第2版本，在第1版本(V2.0)基础上做了重大改进与升级。通过对μC/OS-II源代码的分析与描述，讲述了多任务实时的基本概念、竞争与调度算法、任务间同步与通信、存储与定时的管理以及如何处理优先级反转问题；介绍如何将μC/OS-II移植到不同CPU上，如何调试移植代码，在所附光盘中，给出已通过FAA安全认证的μC/OS-II V2.52的全部源码以及可在PC机上运行的移植范例。

本书可用做高等院校嵌入式实时系统课程教材或工程师培训教材，也可供嵌入式应用开发人员研究与使用。

图书在版编目(CIP)数据

嵌入式实时操作系统μC/OS-II/(美)拉伯罗斯著；
邵贝贝等译. 2版. —北京：北京航空航天大学出版社，2003.5
ISBN 7-81077-290-2

Ⅰ.嵌… Ⅱ.①拉… ②邵… Ⅲ.实时操作系统，μC/OS-II
Ⅳ.TP316.2

中国版本图书馆CIP数据核字(2003)第005702号

本书英文版原名：MicroC/OS-II The Real-Time Kernel, Second Edition
Copyright © 2002 by CMP Books except where noted otherwise.
Published by CMP Books, CMP Media LLC.
1601West 23rd Street, Suite 200, Lawrence, KS. 66046, USA
All rights reserved.
本书中文简体字版由美国CMP Books公司授权北京航空航天大学出版社在中华人民共和国境内(不包括香港特别行政区)独家出版发行。版权所有。
北京市版权局著作权登记号：图字：01-2002-4806

嵌入式实时操作系统μC/OS-II (第2版)
Micro C/OS-II The Real-Time Kernel, Second Edition
[美] Jean J. Labrosse 著
邵贝贝 等译
责任编辑 王 瑛
*
北京航空航天大学出版社出版发行
北京市海淀区学院路37号(100083) 发行部电话：(010)82317024 传真：(010)82328026
http://www.buaapress.com.cn
E-mail:bhpress@263.net
北京市松源印刷有限公司印装 各地书店经销
*
开本：787×1092 1/16 印张：38.5 字数：986千字
2003年5月第1版 2003年5月第1次印刷 印数：6 000册
ISBN 7-81077-290-2 定价：79.00元

献给我亲爱的妻子Manon
以及我们可爱的孩子
James 和 Sabrina

译者序

本书是Jean J. Labrosse先生撰写的*Micro C/OS-II The Real-Time Kernel*一书第2版本的中译本，原文书于2002年出版。该书的第1版本面世于1999年，其中译本名为《μC/OS-II——源码公开的实时嵌入式操作系统》，2001年由中国电力出版社出版。

与第1版本比较，第2版本增添了许多新内容。原文书第1版本的内容只有12章，而第2版本扩展到18章。第1版本的正文只有357页，其余的122页是μC/OS-II V2.00的源程序清单，源码同时也附在一张3寸软盘上；而第2版本有598页，且去掉了μC/OS-II的源程序清单，仅将其附在随带的光盘上。粗略地说，书中有250页是全新的。其中如何测试实时内核中的每个函数及如何调试RTOS，对于拟将μC/OS-II移植到其他CPU上去的读者会特别有用。作者对原有的约350页内容也做了修订，在编排风格上也有不少改变，使之更便于学习和查阅。

原文书第1版本给出的是μC/OS-II V2.00的全部源代码，第2版本给出的是μC/OS-II V2.52的全部源代码。新版本增加了一些新的功能函数，如mutex，Event-flag及浮点运算。最重要的是，μC/OS-II V2.52通过了美国航空航天管理局(FAA)的安全认证，可以用于飞机、航天器等与人性命攸关的控制系统中。也就是说，用户可以放心地使用μC/OS-II V2.52，放心地使用本书中提供的源代码。

提倡在计算机嵌入式应用中使用实时操作系统RTOS(Real Time Operating System)，是因为RTOS将应用分解成多任务，简化了应用系统软件的设计；RTOS使控制系统的实时性得到保证，可以接近理论上能达到的最好水平；良好的多任务设计，有助于提高系统的稳定性与可靠性。嵌入式RTOS是多年来计算机专家们潜心研究的成果，是他们智慧的结晶。嵌入式系统应用的范围不胜枚举，在一个现代化的社会里，可以说，只要能想到的领域都有计算机的嵌入式应用。开发嵌入式应用的工程师，往往只是他相关领域的专家，对计算机的理解，尚不能与计算机专家们相比，这就是国外人士提到的嵌入式应用工程师与计算机专家之间的“缝隙(gap)”。把计算机专家的研究成果拿给嵌入式应用开发的工程师使用，是填补这一缝隙的好方法。

在我国广范地提倡学习和使用RTOS还是最近几年的事。这是由于RTOS的商业软件价格昂贵，著名的商业RTOS需几万美元，而且其中大部分是不提供源码的，即是所谓的黑盒子，不便于学习和使用。另外早年的单片机存储器资源有限，容不下RTOS本身的开销。随着半导体技术的飞速发展，片上系统(system on chip)成为嵌入式应用的发展趋势，RTOS的使用也随之越来越普及。

开卷有益。在产品中用也好，不用也好，对于嵌入式应用工作者，学习和理解RTOS是绝对必要的，故译者更看中μC/OS-II用做教材的意义。本书对于开发计算机嵌入式应用的工

KJS/01/01

程师来说，是一本非常实用的参考书；而对理工科大学的学生，特别是研究生来说，则是一本极好的教科书。说它是一本好书，至少在以下几方面是那些泛泛介绍嵌入式实时系统原理的书所无法比拟的：

- 本书不但提供了完整的嵌入式实时内核的源代码，而且对这些代码的细节做了详尽的解释；它不仅告诉读者这个实时内核是怎么写的，还解释了为什么要这样写。
- 这个专门为嵌入式应用设计的RTOS，绝大部分源代码是用C语言写的，有PC机上运行的范例，可以边读书，边实践。用汇编语言写的部分只有200行左右。该实时内核可以方便地移植到几乎所有的嵌入式应用类CPU上，包括8位、16位和32位单片机及DSP。移植范例可以从互联网上下载。
- 这一点是最重要的。从最老版本的μCOS，以及后来的μC/OS和μC/OS-II V2.00，到今天的μC/OS-II V2.52，已经有10年以上的发展历史了。10年来，许多行业都有成功应用的实例，这些应用实例正是该内核实用性、无误性的最好证据。

向读者介绍μC/OS-II，不能不提及原书作者的第1本书*μC/OS The Real-Time Kernel*。这本书出版于1992年，包括程序清单也只有266页，是μC/OS-II的前身，俗称μC/OS-II的第1版本。过去也曾经称1999年版的μC/OS-II V2.00为μC/OS的第2版本。这样算起来，这本书，即μC/OS-II V2.52应该是第3版本了。总之，这本书已经是作者关于μC/OS的第3本书了。

μC/OS-II V2.52兼容以前的μC/OS-II V2.00，也兼容过去的μC/OS V1.08(作者的第1本书)。从μC/OS到μC/OS-II，内核的基本结构和调度方法并没有什么变化，而源代码的文件结构与组织方式有不少改进，使之更便于移植；增加了一些功能函数，其中最有用的是2个系统任务——CPU负荷监测任务与堆栈容量检查任务。这2个任务在调试应用程序时十分有用。

比较3个版本的RTOS，还有一个显著的变化，即在它的使用许可证策略方面。这是由于μC/OS-II的商业价值得到了越来越广泛的认同。μC/OS是免费软件，在μC/OS一书中，关于使用许可证，作者写道："应用μC/OS的源代码，并将μC/OS的目标代码嵌入到产品中，不需要使用许可证，但用户应当指出，产品中使用了μC/OS。如果销售μC/OS的源代码，则要有许可证。"从μC/OS-II开始，以盈利为目的，将μC/OS-II目标代码嵌入到产品中去，应得到"目标代码销售许可证"；而不论是哪个版本，用于教学都是免费的。

*μC/OS*一书，是1995年欧洲核子研究中心(那里是互联网的发祥地)的计算部主任Verkerk博士，在筹备"国际第四届实时系统在物理中的应用讲习班"时，向我推荐的。以后的几年中，我和我的研究生开始研究和使用μC/OS。最初是在Motorola公司的68HC11上，感觉不错。1999年再次去CERN工作时，也使用了μC/OS，是在M68K(CPU32)上。我向原书作者问及能否将μC/OS介绍给中国读者，得到了他热情的支持。他建议我介绍新出版的*μC/OS-II*一书，并积极帮助与出版商联系。2000年下半年，中国电力出版社购得*μC/OS-II*一书第1版中译本的版权，译者是我和我的研究生们。

在"中国计算机学会微机专业委员会嵌入式系统2000年会"上，我向与会者介绍了μC/OS和μC/OS-II，并得知国内一些同行也在使用μC/OS。第1版中译本出版后，我们在清华大学"嵌入式实时系统与单片机应用"课程中增加了μC/OS-II的内容，并在Motorola公司68HC08，HC11，S12，CPU32，ColdFire，M.core CPU以及DSP56800上实现了μC/OS-II的

运行。一些科研单位和公司还基于 μC/OS－II，对员工做 RTOS 方面的培训 -μC/OS 和 μC/OS－II 逐渐开始在中国得到了越来越多的认可。

原文 μC/OS－II 第 2 版本于 2002 年出版了，北京航空航天大学出版社及时购得中译本版权。有翻译第 1 版中译本的经验，我们力图将第 2 版本译得更好些。在一些专业词汇的翻译上，我们是以通俗、符合国语习惯及便于理解为基础翻译的。如“Context Switch”，过去很多书都译成“上下文切换”，实际是 CPU 寄存器切换、运行环境切换，其实就是任务切换，于是我们就译成“任务切换”；“Preemptive Kernel”，我们译成“可剥夺型内核”，意为“运行着的任务，其 CPU 使用权随时有可能被内核剥夺而交给优先级更高的任务使用”，在 μC/OS－II 的第 1 版中译本中，中国电力出版社将我们的译法改为“占先式内核”。在本书中，北京航空航天大学出版社仍采用了我们的译法。个别词汇，如 mutex，作者称之为“互斥型信号量”，作用是暂时提高占用共享资源任务的优先级，以解决优先级反转问题，因找不到合适的中文与之对应，仍使用原文。若读者有好的建议，我们愿意采纳。

本书仍由我和我的研究生们翻译。宫辉译了第 4，5，12 章；蒋俊峰译了第 6，7，10 章；龚光华译了第 1，9 章；杨世明译了第 17，18 章；马伟、薛涛、刘永毅、邵森龙分别译了第 11，13，14 及第 15 章；其余是我翻译的，全书由我统稿。

在本书翻译过程中，我们的工作得到了作者的鼎力支持，实际上，我们一直保持着密切的电子邮件往来。根据我们的请求，作者还特意为本书写了“致中国读者”，这是我们要特别感谢的。Labrosse 先生还告诉我，目前 μC/OS－II 的最新版本已经到了 V2.62。关于 V2.52 版本，自去年出版以来发现的一些问题，作者希望读者访问网站：

http://uCOS－II.com/contents/support/bugfixes.html

自本书第 1 版中译本出版以来，深得广大中国读者（包括香港和台湾地区的读者）的厚爱。一些读者还中肯地指示了译文中的一些错误和不妥之处，使我们能将第 2 版译得更好一些。对这些热心的读者，我们在此表示衷心的感谢。

由于水平有限，不妥之处请读者批评指正。

邵贝贝

2003 年 4 月　于清华大学

致中国读者

几年前，邵贝贝教授将 *MicroC/OS-II The Real-Time Kernel* 一书译成了中文，并在中国出版，说明该书的价值得到了更多人的认可。2002 年上半年，笔者出版了这本书的第 2 版本，对书的内容和书后所附的源代码都做了大量的改动。笔者相信，翻译第 2 版本的工作量决不会少于第 1 版本。感谢邵教授百忙中抽时间完成这一繁琐的工作。

过去几年中，一直有很多中国用户与笔者保持着通信联系。中国用户对嵌入式系统、特别是对 μC/OS-II 表现出极大的兴趣。多数通信交往来自学术界，随着基地在中国的电子产品制造业的发展，他们对 μC/OS-II 的兴趣也越来越浓厚。

对于大专院校的学生，μC/OS-II 是实时操作系统(RTOS)的极好教材。试着用一用 RTOS，获得一些 RTOS 的使用经验，不需要花什么钱，μC/OS-II 是在校学生完成研究生论文和本科生毕业设计的绝好工具。μC/OS-II 可以使读者专注于自己的研究课题，而不必为程序结构花很多时间。有大量移植范例可以从 Micrium 的网站(www.micrium.com)上免费下载。μC/OS-II 的性能可以与许多高端商业软件产品媲美，某些性能比它们还要好。

μC/OS-II 已经在世界范围内得到广泛使用，包括诸多领域，如手机、路由器、集线器、不间断电源、飞行器、医疗设备及工业控制等。实际上，μC/OS-II 已经通过了非常严格的测试，并且得到了美国航空管理局(Federal Aviation Administration)的认证，可以用在飞行器上。这说明 μC/OS-II 是稳定可靠的，可用于与人性命攸关的安全紧要(safety critical)系统；当然，也可用于非安全紧要系统。

μC/OS-II 一直在不断地向前发展。书的出版有个过程，书出版时，书中描述的可能已不是当前的最新版本；故我请邵教授在翻译这本书时，仍用 μC/OS-II 的 V2.52 版本，以同原书版本一致。笔者建议，如果读者只想学习 μC/OS-II，仍应使用 V2.52 版本。如果想将 μC/OS-II 嵌入到商业产品中，请与我们 Micrium 公司(或我们的中国代理)联系，以获得使用许可证并得到最新版本。关于使用许可证，可直接向以下地址申请：licensing @ micrium.com，请附上用户姓名、公司名以及联系电话。

关于 μC/OS-II 今后升级方面的设想是多方面的，请关注 Micrium 网站。笔者打算今后将 μC/OS-II 可以管理的任务数目增加到 250 个，进一步改进调度算法，使之效率更高。将来的 μC/OS-II 可以在任务或中断服务子程序中直接给别的任务发消息，而不必通过诸如消息队列等。也还会给 μC/OS-II 增加一些新的功能函数。

1999 年，笔者注册了一家名为 Micrium 的公司，目的是开发并销售类似 μC/OS-II 的软件。目前可提供的其他软件产品还有：

- μC/GUI (嵌入式用户图形界面)；
- μC/FS(嵌入式文件系统)。

Micrium 计划中的产品也都是专门面向嵌入式系统应用的。所有产品都提供源代码，其中大多数软件的质量也同 μC/OS-II 一样好。

多年来，我一直参加美国的嵌入式系统年会(Embedded System Conference)，并在会上做两三次讲演。我也打算参加一次中国的嵌入式系统年会，或许我们能在会上见面。

如果发 e-mail 给我，我会尽力给予回复；但是请用英文书写，以谅解我不懂中文。不必顾虑自己的英文不够好，总比我的中文要好得多吧。

如同笔者从写书得到乐趣，愿读者从阅读此书中得到更多的乐趣。

Jean J. Labrosse

2003 年 1 月

前　言

10 年前，我写了我的第 1 本书，书名是 *μC/OS The Real-Time Kernel*（《实时内核 μC/OS》）。1998 年，这本书被 *Micro C/OS-II The Real-Time Kernel*（《实时内核 Micro C/OS-II》）替代。书的封面以 Micro 一词换下了希腊字母 μ，因为从书店管理上考虑，非英文字母开头的书不便于管理。然而无论是动机还是效果，Micro C/OS 和 μC/OS 都是同义词。本书中，多数情况下我使用 μC/OS-II 的写法。这本书是 μC/OS-II 的第 2 版本，也可以说是 μC/OS 的第 3 版本。

适用于对安全性要求苛刻的系统

2000 年 7 月，μC/OS-II 在一个航空项目中得到了美国联邦航空管理局(Federal Aviation Administration)对用于商用飞机的、符合 RTCA DO-178B 标准的认证。该标准对用于航空设备方面的软件提出了要求。为了符合这一标准，必须尽可能地通过文件描述和测试，展示软件在稳定性与安全性这两方面都符合要求。这对于操作系统来说特别重要，因为这一结论表明，该操作系统的质量得到了认证，可以在任何场合中应用。为了表明 μC/OS-II 具有足够的安全性与稳定性，能用于与人性命攸关的、安全性条件极为苛刻的系统，μC/OS-II 的每一种功能、每一个函数及每一行代码都经过了考验与测试。

新版本新在何处

本书对 *Micro C/OS-II The Real-Time Kernel* 一书做了重新修订。

增加了新的章节

本书老版本只有 12 章，而新版本有 18 章。笔者将老版本的第 6 章“任务间通信与同步”分成了 6 章：1 章介绍事件控制块，1 章讲信号量，1 章讲互斥型信号量，1 章介绍事件标志，消息邮箱占 1 章，然后是消息队列。

老版本中有“将 μC/OS-II 移植到 Intel 80x86 处理器上”的内容，但这种移植只能处理整数寄存器的切换。笔者增加了 1 章，讲述的移植方法包括如何处理带浮点寄存器的 CPU。80486 和奔腾处理器都带有浮点运算寄存器。

笔者还增加了 1 章，介绍书中用到的 PC 机中的系统调用。

删除了一些章节

笔者删除了“从 μC/OS 升级到 μC/OS-II”一章，因为很少有人还在用 μC/OS，μC/OS-II要通用得多。

笔者还删除了“附录 HPLISTC 和 TO”，因为现在很多好的文本编辑软件都可以打印出漂亮的源码清单。

删除了源代码清单

笔者决定删除附录中的 μC/OS-II 的源代码。删除这些源代码清单有3个理由：第1，这一版本增加了150多页新内容，如果不删除附录中的源代码，本书将会超过750页，那它将是一个难以携带的庞然大物(原书已经够厚的了)；第2，这些源代码可以在本书所附的光盘中找到，在计算机上读这些源代码可能感觉更好，而且，这些源代码已经在书中讲解过了，附录中只是这些代码的重复；第3，和任何其他软件一样，μC/OS-II 也会被修改和升级，因此，附录中给出的源代码清单也会完全过时，以至毫无价值。

新增了系统服务

μC/OS-II 的代码基本上与先前的版本相同，只是增加了一些新的系统服务。

先前的版本包括以下服务：

- 时间管理；
- 二值信号量和计数型信号量管理；
- 消息邮箱管理；
- 消息队列管理；
- 存储块容量固定的存储管理。

新版本增加的系统服务有：

- 互斥型信号量(mutexes)管理；
- 事件标志管理。

更多的范例

在一些章节中，笔者增加了一些范例，教给读者如何使用书中提到的服务。

新的编排法

为了更方便使用，笔者对一些编排方法做了改动。过去对一些代码的解释较为繁琐，故重新写了这些部分。读者会注意到，笔者提到一幅图中的某个特定元素时，使用的是字母 F，跟着是图号，然后是想让读者注意的那个特定的元素号。F1.2(3)的含义是：**请注意图1.2中的第3项**。在先前的版本中使用的是这种方法，而新版本中笔者将图与程序清单中的项素描述符全部靠左边开头对齐，而不是插在描述的文字之间。笔者发现，这样对代码和图表的阅读要方便许多，希望读者喜欢。

μC/OS－II 之所求

笔者最主要的意图是将实时内核的内部机理讲清楚。通过理解内核是如何工作的，读者可以更理性地决定是否在产品中使用实时内核。本书讲到的大多数概念，大量商业内核中都使用了。笔者的另一个重要目的是，给读者提供一个高质量的实时内核产品，并有可能用到自己的产品中去。μC/OS－II 及其开放的源码并非自由软件，如果读者将 μC/OS－II 用到商业产品中去，应得到许可（见附录 B："μC/OS－II 的使用许可证"）。

读者对象

本书是写给嵌入式系统的程序员、策划人员以及对实时操作系统有兴趣的学生的。μC/OS－II是一个性能好、实时性有保证的实时内核，可以并且已经嵌入到商业产品中。

可以考虑直接使用 μC/OS－II，而不必自己写内核。读者会发现，自己写一个实时内核并非那么简单。

笔者认为，本书的读者应会用 C 语言，具有汇编语言的基本知识，了解微处理器的基本构架。

μC/OS－II 的运行条件

本书提供的源码假定，读者使用的 PC 机是与 PC/AT 兼容的（至少是 80386），运行 DOS 4.x 或以上，C 的编译使用的是 Boland C++V4.51，应该有大约 10 MB 的硬盘空间。本书提供的样本代码实际上是在 300 MHz 的奔腾 II 上运行的，用的是微软的 Windows 2000，这些代码也已经在 Windows 95，98 和基于 NT 的机器上成功地运行过。

在 PC 以外的不同目标 CPU 上运行 μC/OS－II，可以自行将 μC/OS－II 移植到该 CPU 上，也可以从正式的 μC/OS－II 网站上得到相应的移植范例。网址为

http://www.uCOS－II.com

还要有合适的软件开发工具，如 ANSI 的 C 编译、汇编、链接、下载以及某种调试手段（debugging）。

μC/OS－II 的故事

多年以前，我设计了一个基于 Intel 80C188 的产品，需要一个实时内核。在前一家雇主那里工作时，我一直在使用一个知名的内核（以下称为内核 A），但该内核对我当时设计的产品来说太贵了。我找到了一个廉价的内核（以下称为内核 B），当时大约 1 000 $，并开始了我的设

计工作。我试着让那几个很简单的任务运行起来，竟用了2个月时间。我一天到晚给该厂商打电话求援。该厂商声称，内核B是用C语言写的；不管怎样，我须用汇编语言给程序中每个对象初始化。虽然该厂商很耐心，但我实在是够了！产品的开发也耽误了。我真是不想再花时间去调试那个廉价的内核了。后来得知，我是该厂商的第一个客户。这个内核实际上没有完全测试和调试过。

还是回到正路上来，我决定使用内核A。5个用户的价格是5 000 $，此外，我的每个产品还要付200 $的内核使用费。在当时，那可是一笔很大的开支，就算是花钱买个心里踏实吧。用了2天时间，内核转起来了。项目进展到3个月上，我的一个工程师发现，内核好像有毛病(bug)。当内核中的这个毛病被确认无疑后，我把我的代码寄给了该厂商。此时，厂商提供的90天的担保已经过期了。为了得到答复，我还须支付每年500 $的维护费。我和销售商争辩了好几个月，我认为，他们应该尽快处理这一处毛病才是，我实际是在为他们做好事；可是，他们不予理睬。最后，我只好认了，又签了程序维护合同，而该厂商6个月以后才处理了那个毛病。是的，6个月以后！我简直气坏了，最要紧的是，我的产品开发耽误了。用了将近1年的时间，内核A才在我的产品中稳定运行。然而，我得承认，从那儿以后，内核A再没有什么问题了。

渐渐地，我自然会想到，写个实时内核有那么难吗？不就是不断地保存、恢复CPU的那些寄存器嘛。那是我决定试试写我自己的实时内核时的情景，主要是用业余时间、晚上和周末。也用了大约1年的时间，才使内核转起来，在某些方面还比内核A好。我并不想开公司去卖这个内核，因为当时市场上已经有大约50个实时内核的产品了，人们会想，怎么又多出一个来了呢？

我想到给杂志写篇文章，打算投《C语言用户日记》(*C User's Journal*)杂志，因为内核是用C写的。我听说该杂志稿费为每页100 $，而其他杂志一般是每页75 $。我的文章大约70页，这笔稿费对我写内核所用的时间应是不错的补偿。很不幸，文章被拒载了。有两个原因，一个是文章太长，杂志不愿意连载；另一个原因是，他们也不想要"又是一篇关于内核的文章"。

我决定转向《嵌入式系统编程》(*Embedded System Programming*)杂志，因为我的内核是为嵌入式应用设计的。我和该杂志的编辑Tyler Sperry先生联系，告诉他，我想在他的杂志上发表我的实时内核。我得到的答复和《C语言用户杂志》是一样的：我们不要"又是一篇关于内核的文章"。我告诉他们，我的实时内核和其他人的不一样，是可剥夺型的，可以和许多商业上出售的内核产品竞争，源代码可以放在《嵌入式系统编程》杂志网站的留言板上(BBS)。我每周给Tyler先生打两三个电话，几乎是求他发表我的文章。他终于同意了，大概是我的电话扰得他烦了。

我的文章从70页被压缩到30页，连载在1992年5月期和6月期杂志上。该文章几乎是1992年最热门的文章。头1个月，就有500多人从《嵌入式系统编程》杂志的网站留言板上下载我的源程序。Tyler先生真怕实时内核的商家们会要了他的命，因为在他的杂志上刊登了实时内核的源码，这使商家们感到不安。我猜测，商家们一定已经认识到μC/OS(当时称做μCOS)的质量和功能。该文章实际上是第1次给实时内核的内在工作原理曝光，一些过去的秘密公开了。几乎是文章在《嵌入式系统编程》杂志上被刊登出来的同时，R&D出版社的Bernard(Berny)Williams博士来电话了，《C语言用户日记》杂志就是这家出版社出版的。此时距我最初与该杂志联系已有6个月了。

他留言给我的妻子，说他对我的文章感兴趣；我回电话给他："你不觉得你有点太晚了吗？我的文章已经在《嵌入式系统编程》杂志上发表了"。Berny 只是说："不，不，你没明白我的意思，因为文章那么长，我打算出一本书"。

起初，Berny 只是想简单地把我手上已有的书稿出版成一本大约 80 页的书。我告诉他，如果写成一本书的话，我打算写得更好一些。以后，我又用了大约 6 个月的时间增写书的内容，就成了本书的第 1 版本。书大约 250 页。我把实时内核的名字由 μCOS 改成 μC/OS，因为《嵌入式系统编程》杂志的读者一直称 μCOS 为 mucus，那个词发音像英语的"痰"，听起来让人不舒服。回想起来，这可能是第 1 次给实时内核起名字。总之，实时内核 μC/OS 就这样诞生了。

开始时，书销售得很慢。Berny 和我预计在书的售期内能售出 4 000～5 000 册，可是照开始时的销售速度，能卖出 2 000 册就不错了。Berny 坚持说，让人知道这本书需要时间。于是他连续在《C 语言用户日记》上刊登广告，一直持续了 1 年。

约在书正式出版的 1 个月之前，我第 1 次参加了在加利福尼亚州的圣・克莱尔(Santa Clara)举行的嵌入式系统会议，那是 1992 年的 9 月，我第 1 次见到 Mr. Tyler Sperry，我给他看了即将出版的书的草稿。他很快地翻了翻，问我愿不愿意在下一届嵌入式系统会议上做个讲演，下届会议在亚特兰大召开。那再好不过了，我说我愿意。但我讲些什么呢？他建议我讲"怎样使用小实时内核"。在从加利福尼亚返回的路上，我想，我这是怎么了？我从来没有在那么多人面前讲过话，到时候我自己慌了怎么办？如果我讲的都是些很平常的知识……，人们可都是花了钱来听会的。以后的 6 个月中，我准备了讲稿。后来的会上，听我课的有 70 多人。起初的 20 多分钟里，我出的汗至少有 1 磅多。课后，约有 15 人上前来表示非常喜欢我的课，也喜欢我那本书。我得到了 1993 年在圣・克莱尔举行的会议的邀请，可我没能参加，我参加了 1991 年波士顿举行的会议。从那以后，每次嵌入式系统会议上都有我的例行讲演。过去几年中，我进了会议的顾问委员会(Advisory Committee)。最近的几次会上，每次的会议我都要做至少 3 次讲演，每次听众都在 200～300 人。会议对我的讲演的评价总是在前 10%。

至今，μC/OS 和 μC/OS－II 的书在全球的销量已超过 25 000 多册。我收到并答复了来自至少 44 个国家的数千封电子邮件，我仍然试图答复他们每一个人。我确信，如果您写给我什么，回应您是我的责任。1995 年，《实时内核 μC/OS》被翻译成日文，在《接口》(*Interface*)杂志上刊登。2001 年，μC/OS－II 被翻译成中文；韩文版的译文出现于 2002 年初；μC/OS－II 的日文翻译工作正在进行中，2002 年已面世。

μC/OS 和 μC/OS－II 已经被移植到 40 多种不同的 CPU 上，移植上去的 CPU 数量还在不断增加。读者可以访问 μC/OS－II 网站 http://www.uCOS－II.com，看您想用的那一款处理器是否有现成的移植范例。

1994 年，我决定写第 2 本书《嵌入式系统构件——实用的 C 语言模块》(*Embedded System Building Blocks, Complete and Ready to use Modules in C* (ESBB))。ESBB 一书的第 2 版已于 2000 年出版。因某些缘故，此书不如 μC/OS 那么流行，虽然书中的很多信息在其他地方是找不到的。我始终认为，对于刚刚步入嵌入式世界的人们来说，这是一本理想的书。

1998 年，我开通了 μC/OS 在互联网上的正式网站 http://www.uCOS－II.com。我打算在该网站上设移植、应用短文、相关网站、常遇问题解答及 μC/OS－II 的版本更新等，只是这都要花时间去做。

2001年，我建立了一个新闻组，用户可以共享关于μC/OS-II的消息和交换彼此的使用经验。

回到1992年，我绝想不到写篇文章会改变了我的生活。实际上确实是这样，我遇到了很多非常有趣的人，并在交往中结识了很多好朋友。

感谢您选择了这本书，希望您喜欢！

感 谢

首先，并且是最重要的，要感谢我的妻子Manon对我的支持、鼓励和理解，特别是她的容忍。我又一次低估了重新编辑这本书的工作量——我曾经以为只需要几个星期就可以完成。我还要感谢我的两个孩子James(11岁)和Sabrina(8岁)，原谅我总是长时间地俯身于计算机上。

向Gino Vannelli先生致以特别的谢意，他创作了美妙的音乐。我个人认为，Gino重新定义了“完美”一词，谢谢你，Gino，你的音乐陪伴我近30年了。

感谢CMP书局(CMP Books)的所有好心人，是他们帮助出版了这本书，且都是按照我的意愿办的。

最后，我要感谢这些年来所有买过我的*μC/OS*，*μC/OS-II*以及*Embedded System Building Blocks*(《嵌入式系统构件》)图书的读者。

概　述

本书讲述 μC/OS－II 的设计与实现。μC/OS－II 读做“micro C O S 2”，意为“微控制器操作系统版本 2”。

μC/OS－II 是一个完整的，可移植、固化、裁剪的占先式实时多任务内核。μC/OS－II 是用 ANSI 的 C 语言编写的，包含一小部分汇编语言代码，使之可供不同架构的微处理器使用。至今，从 8 位到 64 位，μC/OS－II 已在超过 40 种不同架构的微处理器上运行。

μC/OS－II 是基于《实时内核 μC/OS》的，该书最初在 1992 年出版。世界上已有数千人在各个领域中使用 μC/OS，这些领域包括：照相机行业、航空业、高端音响、医疗器械、电子乐器、发动机控制、网络设备、高速公路电话系统、自动提款机及工业机器人等。很多高等院校将 μC/OS 用于实时系统教学。

μC/OS－II 和 μC/OS 版本 V1.11（μC/OS 的最终版本）是向上兼容的，并且改进了很多。如果应用程序是基于 μC/OS 的，实际上不需要做什么改动，就可以在 μC/OS－II 上运行。μC/OS 所有的功能，即函数调用，在 μC/OS－II 中都有，只须改变 include 文件和 build 文件，使它们指向新的文件名即可。

本书附带的 CD 包含 μC/OS－II 的全部源码。CPU 相关部分使用的是 Intel 80x86 微处理器的代码，运行是在实模式下，C 编译使用大模块方式。μC/OS－II 是在 PC 机上开发的，PC 机运行 Windows 2000，在 Windows 95，98，Me，NT 和 XP 上运行应该同样可行。例子是在 Windows 环境下的 DOS 窗口内运行的。C 编译器使用的是“Boland C/C＋＋编译器国际版 V 4.51”。虽然 μC/OS－II 是在 PC 机上开发和测试的，但 μC/OS－II 的实际对象是嵌入式系统，并且很容易移植到不同架构的微处理器上。

μC/OS－II 的特点

有源代码　前面已经提到，本书附带的 CD 包含 μC/OS－II 的全部源代码（约 5 500 行）。笔者尽了最大的努力，以提供给读者高质量的软件。读者可能不喜欢源程序中使用的某些格式，但这份源码清晰易读，且结构协调。许多商业实时内核的软件都是以源代码形式提供的，读者可以找一个做比较，看它是否像 μC/OS－II 那样干净、漂亮、和谐一致，是否注解得那么详尽，组织得那么有序。笔者认为仅仅给出一个源代码是远远不够的，读者还想知道这些代码的工作原理是什么，以及这一段段的代码是怎么拼到一起的，这些信息都在这本书里。有那么多文件，数千行的程序，如果单从读源码入手，实时内核的整体架构就不那么容易弄清楚。

可移植性（portable）　μC/OS－II 源码绝大部分是用移植性很强的 ANSI C 写的，与微处理器

硬件相关的部分是用汇编语言写的。汇编语言写的部分已经压到最低限度，以使 μC/OS-II 便于移植到其他微处理器上。如同 μC/OS 一样，μC/OS-II 可以移植到许许多多微处理器上，条件是：该微处理器具有堆栈指针，具有 CPU 内部寄存器入栈、出栈指令。另外，使用的 C 编译器必须支持内嵌汇编(in line assembly)，或者该 C 语言可扩展和可链接汇编模块，使得关中断和开中断能在 C 语言程序中实现。μC/OS-II 可以在绝大多数 8 位、16 位、32 位以至 64 位微处理器、微控制器及数字信号处理器(DSP)上运行。

从移植了的 μC/OS 升级到 μC/OS-II，全部工作 1 小时左右就可完成。因为 μC/OS-II 和 μC/OS 是向上兼容的，应用程序从 μC/OS 升级到 μC/OS-II 几乎不须改动或根本不须改动。移植的范例可从互联网上找到，网址是 http://www.uCOS-II.com。

可固化(ROMable) μC/OS-II 是为嵌入式应用而设计的，这就意味着，只要具备合适的系列软件工具(C 编译、汇编、链接及下载/固化)，实际上就可以将 μC/OS-II 嵌入到产品中作为产品的一部分。

可裁剪(scalable) 可以只使用 μC/OS-II 中应用程序需要的系统服务。也就是说，某产品可以只使用很少几个 μC/OS-II 调用，而另一个产品则可能使用了几乎所有 μC/OS-II 的功能，这样可以减少产品中 μC/OS-II 所需的存储器空间(RAM 和 ROM)。可剪裁性是靠条件编译实现的，只要在用户的应用程序中(用 # define constants 语句)定义哪些 μC/OS-II 中的功能是应用程序需要的就可以了。

可剥夺性(preemptive) μC/OS-II 是完全可剥夺型的实时内核，即 μC/OS-II 总是运行就绪条件下优先级最高的任务。大多数商业内核也是可剥夺型的，μC/OS-II 在性能上与它们类似。

多任务 μC/OS-II 可以管理 64 个任务；然而，笔者建议用户保留 8 个，给 μC/OS-II。留给用户的应用程序最多可有 56 个任务。赋予每个任务的优先级必须是不相同的，这意味着 μC/OS-II不支持时间片轮转调度法(round-robin scheduling)。该调度法适用于调度优先级平等的任务。

可确定性 绝大多数 μC/OS-II 的函数调用和服务的执行时间具有可确定性。也就是说，用户总是能知道 μC/OS-II 的函数调用与服务执行了多长时间。进而可以说，除了函数 OSTimeTick()和某些事件标志服务，μC/OS-II 系统服务的执行时间不依赖于用户应用程序任务数目的多少。

任务栈 每个任务都有自己单独的栈。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。使用 μC/OS-II 的栈空间校验函数，可以确定每个任务到底需要多少栈空间。

系统服务 μC/OS-II 提供很多系统服务，例如信号量、互斥型信号量、事件标志、消息邮箱、消息队列、信号量、块儿大小固定的内存的申请与释放及时间管理函数等。

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。

稳定性与可靠性　μC/OS－II 是基于 μC/OS 的，μC/OS 自 1992 年以来已经有数百个商业应用。μC/OS－II 与 μC/OS 的内核是一样的，只是提供了更多的功能。另外，2000 年 7 月，μC/OS－II在一个航空项目中得到了美国联邦航空管理局(Federal Aviation Administration)对用于商用飞机的、符合 RTCA DO－178B 标准的认证，该标准对用于航空设备方面的软件提出了要求。为了符合这一标准，必须尽可能地通过文件描述和测试，展示软件在稳定性与安全性这两方面都符合要求。这一结论对于操作系统来说特别重要，因为这一结论表明，该操作系统的质量得到了认证，可以在任何应用中使用。为了表明 μC/OS－II 具有足够的安全性与稳定性，能用于与人性命攸关的、安全性条件极为苛刻的系统，μC/OS－II 的每一种功能、每一个函数及每一行代码都经过了考验与测试。

图、程序清单及列表规则

图、程序清单及列表在书中的排序规则是：图以 F 开头，接着是按章排序的阿拉伯数字，最后一项指示图中的内容。如 F1.2(3)表示图 1.2 中的第 3 项。

程序清单和列表也使用同一规则。程序清单以 L 开头，列表以 T 开头。

各章内容

图 F0.1 所示为本书的编排与章节顺序。笔者认为，此图对理解书中各章的关系是有用的。第 2 章是单独的一章，不依赖于其他任何章节。最基本的，笔者建议，应阅读前言、概述、第 1 章和第 3 章。有了这些知识，实际上已经拥有了 μC/OS－II，应该可以开始使用 μC/OS－II 了；此时可以接着读第 16，17 章，看 μC/OS－II 有哪些功能可以使用；如果想进一步理解 μC/OS－II，可以阅读第 4，5 和第 6 章；在理解了第 6 章以后，可以跳到“同步”或者“通信服务”那部分。

第 1 章　初识 μC/OS－II　目的是使读者立即感受一下 μC/OS－II。实际上，笔者假设读者对 μC/OS－II 和多任务一无所知，仅介绍一些急需的基本概念。相对于本书的前一版本，这一章完全重写了。

第 2 章　实时系统概念　介绍一些实时系统中的基本概念，如前/后台系统、临界段、资源、多任务、任务切换、调度、可重入性、任务的优先级、互斥、信号量、任务间通信及中断等等。

第 3 章　内核结构　介绍 μC/OS－II 和其内部结构。具体内容包括任务、任务状态和任务控制块；μC/OS－II 是如何处理就绪任务表、任务调度和空闲任务的；如何确定 CPU 的使用权；μC/OS－II 是如何处理中断的；如何初始化和启动 μC/OS－II 等等。

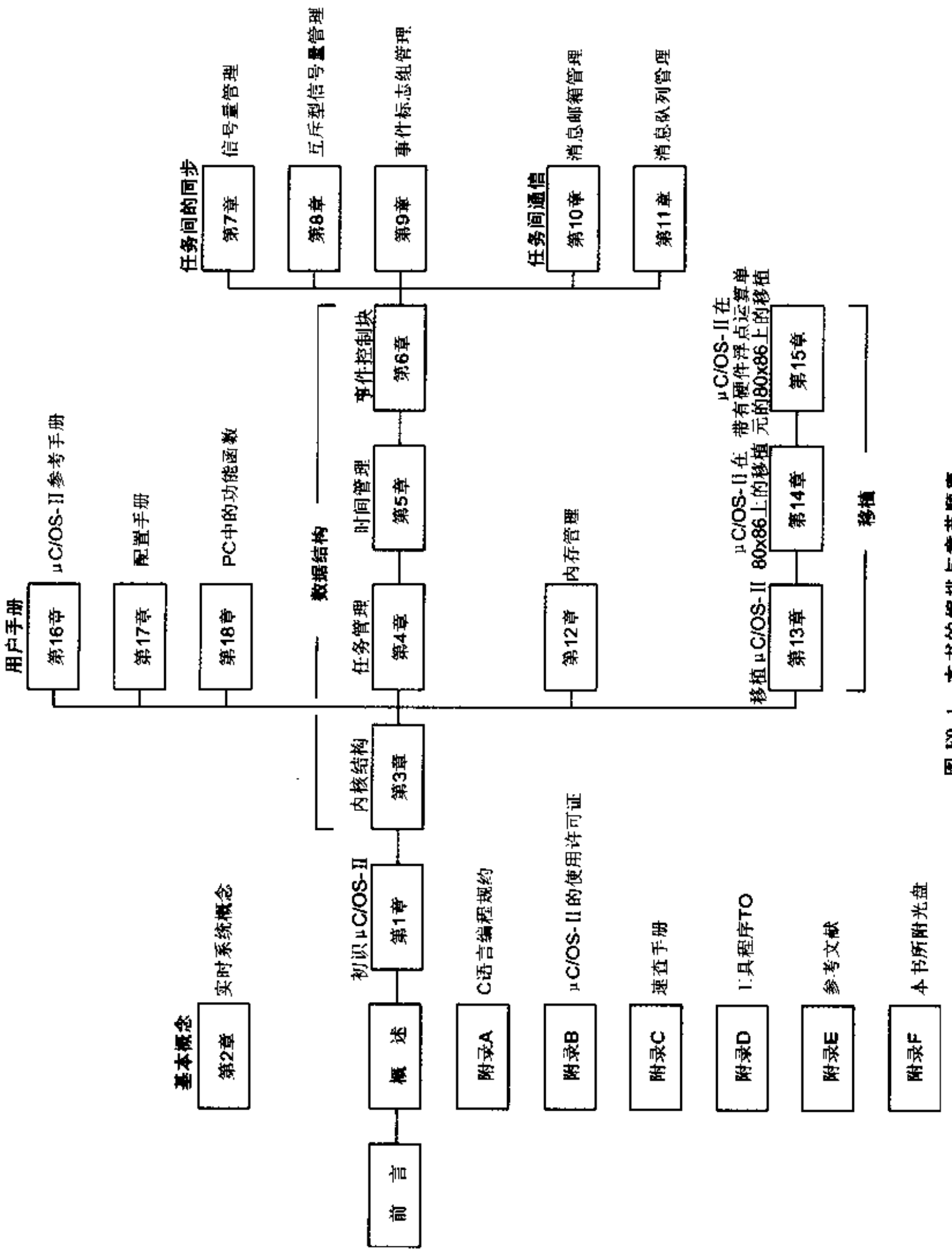


图 F0.1 本书的编排与章节顺序

第 4 章　任务管理　描述 μC/OS - II 提供的任务服务。内容包括建立一个任务，删除一个任务，检查任务栈的余量，改变任务的优先级，挂起和恢复任务以及获取有关任务的信息等。

第 5 章　时间管理　讲述 μC/OS - II 如何将一个任务挂起一段用户定义的时间，这个挂起的任务是如何恢复的，以及如何得到和设定 32 位的时钟节拍计数器的值。

第 6 章　事件控制块　描述事件控制块的数据结构，内核中的这种数据结构面向同步与通信。这种数据结构使得任务能完成任务与任务、任务与中断服务子程序及任务与共享资源间的通信。这一章提供第 7～11 章的预备知识。

第 7 章　信号量管理　信号量是组成内核的一个子项，面向用户任务，用于实现对共享资源的互斥性占用。本章描述 μC/OS - II 中信号量的工作机理。

第 8 章　互斥型信号量管理　互斥型信号量(mutex)是一种二进制信号量，用于满足对共享资源的互斥性处理。如果需要，mutex 通过自动改变任务的优先级，降低发生优先级反转的可能性。本章描述 μC/OS - II 中 mutex 是如何实现的。互斥型信号量是本书这一版本提供的新的服务。

第 9 章　事件标志组管理　事件标志是一些状态位(bits)，任务可以等待这些状态位，等待某一位或某些位的置位与复位。本章描述事件标志是如何实现的，讲述用户程序中应如何使用事件标志。事件标志是本书这一版本提供的新的服务。

第 10 章　消息邮箱管理　消息邮箱管理用于任务间互相传送消息。本章讲述消息邮箱是如何实现的。

第 11 章　消息队列管理　消息队列类似于消息邮箱，只是消息邮箱可以把多条消息传给一个或几个任务。本章介绍消息队列是如何实现的。

第 12 章　内存管理　描述 μC/OS - II 中的动态内存分配，内存块的大小是固定的。

第 13 章　移植 μC/OS - II　一般性地描述将 μC/OS - II 移植到不同的微处理器架构上需要做些什么。与老版本相比，这一章重新写了。

第 14 章　μC/OS - II 在 80x86 上的移植　讲述如何将 μC/OS - II 移植到 Intel/AMD 公司的 80x86 微处理器架构上，采用的是实模式，在大模块下编译。

第 15 章　μC/OS - II 在带有硬件浮点运算单元的 80x86 上的移植　是在原版本基础上的扩展，指明如何把 80486，5x86 及奔腾处理器中的浮点寄存器加到任务切换中去。这一章是老版本没有的。

第16章 μC/OS-II参考手册 从方便应用程序开发者查阅的角度出发，讲述μC/OS-II提供的每一个函数(系统服务)。包括主要功能描述、样板、从哪个文件中可以找到、函数变量和返回值、注意和警告以及调用实例。与老版本相比，这一章增加了不少新的服务(互斥型信号量、事件标志等)。

第17章 μC/OS-II配置手册 介绍如何在用户程序中用常数定义语句(#define constants)配置μC/OS-II。用户通过配置μC/OS-II，可只使用应用程序需要的那一部分服务，使得在减少μC/OS-II占用的代码量(程序和数据空间)方面有较大的灵活性。新版本的配置选项较老版本多了3倍多，以减小μC/OS-II本身占用的代码与数据空间。

第18章 PC中的功能函数 第1章的例子中假定使用的是IBM/PC机。这是新增加的一章，笔者列出了PC机可提供的对μC/OS-II有用的系统服务。

附录A C语言编程规约 列出书中用到的和笔者日常习惯使用的C语言编程规约。

附录B μC/OS-II的使用许可证 指明以源码及目标码方式传发μC/OS-II的许可证政策。

附录C μC/OS-II速查手册 为μC/OS-II服务提供快速查找。

附录D 工具程序TO TO是一则DOS下的实用程序，允许在不同的DOS目录间换来换去，而不必敲很长的改变路径命令CD path。

附录E 参考文献 如果用户想得到更多的关于嵌入式实时系统的资料，这一附录中的参考文献是有用的。

附录F 本书所附光盘 告诉用户本书所附光盘中的内容以及如何安装μC/OS-II。

μC/OS-II网页 为了向用户提供更好的服务，笔者建立了μC/OS-II网页(http://www.uCOS-II.com)，从网页上可以得到如下信息：

- 关于μC/OS和μC/OS-II的新闻；
- μC/OS-II的版本升级；
- μC/OS-II中小毛病的修复(bugs fixes)；
- 向不同CPU移植的范例；
- 经常遇到的问题的解答(FAQs)；
- 应用范例；
- 相关书籍；
- 相关网站。

第 1 章

初识 μC/OS－II

这一章将提供 4 个范例，说明如何使用 μC/OS－II。在本书一开始就说明，这一章是为了让读者尽快开始使用 μC/OS－II。实际上，本书假设读者对 μC/OS－II 和多任务知之甚少，因而在本章有必要介绍一些基本概念。

这些例子是用 Borland C/C＋＋编译器（V 4.51）编译的，用选择项产生 Intel/AMD 80186 处理器（大模式下编译）的代码。这些代码在 Intel Pentium II PC（300 MHz）的计算机上，在 Windows 2000 的 MSDOS 模式下运行和测试过。从任何角度来讲，Intel Pentium II PC 都可以看成是速度特别快的 80186。Borland C/C＋＋ V 4.51（ Borland Turbo C＋＋4.5）可以从 www.Borland.com 网站得到，并且 Borland 承诺，在未来的几年内都可以购买到该编译器。

选择 PC 作为目标系统是由于以下原因：首先，最为重要的是，以 PC 作为目标系统，比起以其他嵌入式环境（评估板、仿真器）作为目标系统，更容易进行代码的测试，无需不断地烧写 EPROM，不断地向 EPROM 仿真器或 CPU 仿真器下载程序，只需简单地编译、链接和运行；其次，使用 Borland C/C＋＋产生的 80186 的目标代码（实模式，在大模式下编译）与所有 Intel，AMD 或其他公司的 80x86 CPU 兼容。

1.00 安装 μC/OS－II

本书附带 1 张光盘，包括了所有涉及的源代码。可以参考本书的附录 F，安装 μC/OS－II 的代码并且运行范例程序。假定是在安装了 Windows 95，98，Me，NT，2000 或 XP 的计算机上安装 μC/OS－II。

1.01 范例 1

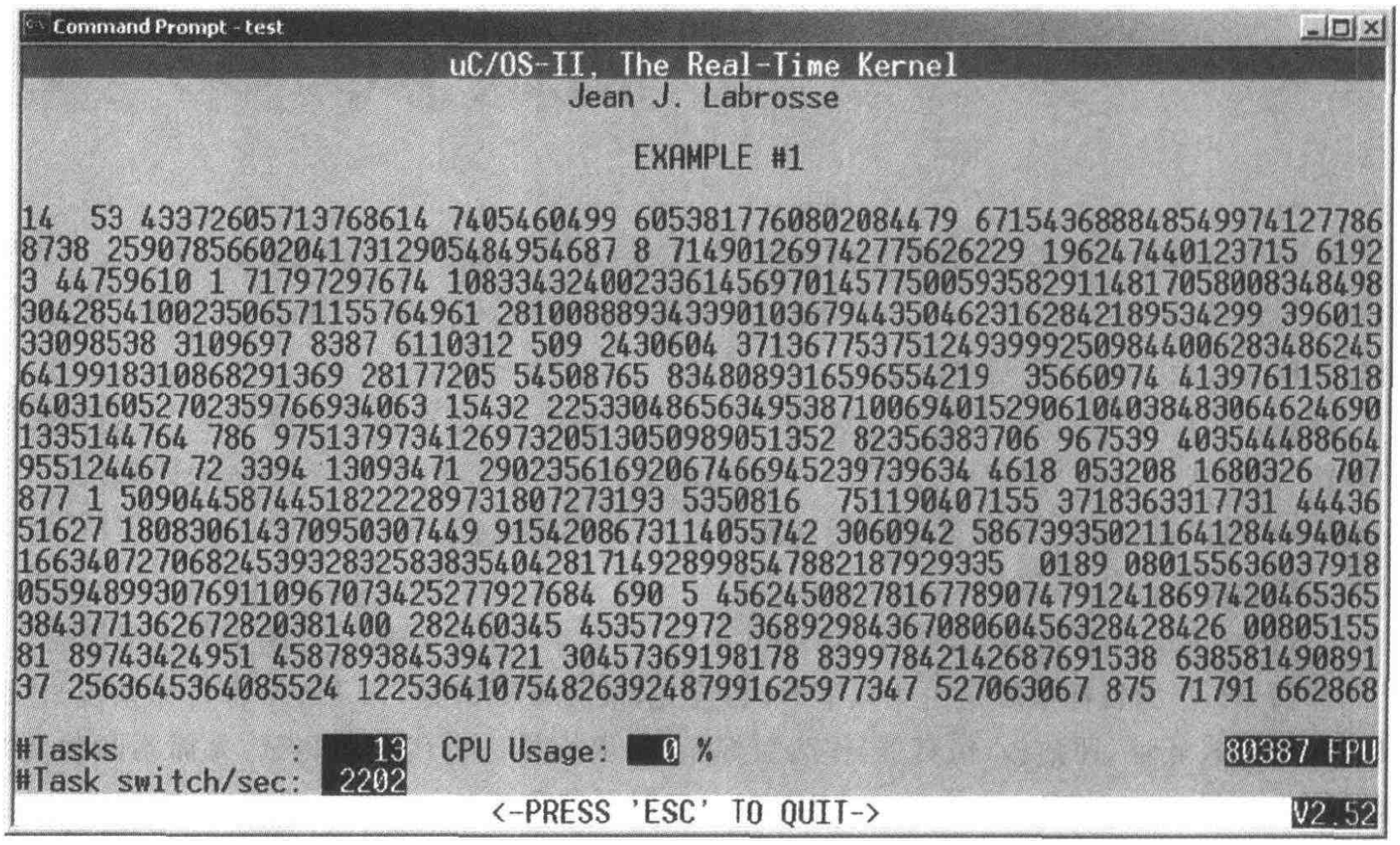
范例 1 演示 μC/OS－II 的多任务处理能力。共有 10 个任务在屏幕随机的位置上显示一个 0～9 的数字。每个任务只显示同一个数字，也就是其中一个任务在随机位置显示 0，另一个显示 1，等等。

这个范例的代码在安装盘的 \SOFTWARE\uCOS－II\EX1_x86L\BC45 目录下，可以打开一个 MSDOS 窗口（在 Windows 2000 下称做命令行方式），输入：

```
CD \SOFTWARE\uCOS－II\Ex1_x86L\BC45\TEST
```

CD命令将切换到范例1的TEST目录。该目录包含4个文件：MAKETEST.BAT，TEST.EXE，TEST.LNK和TEST.MAK。只需在命令行方式下输入TEST，就可以开始运行范例1。TEST.EXE程序在DOS窗口中执行。

约1 s后，可以看到DOS窗口被0～9的数字随机地填满了，如图F1.1所示。



图F1.1 范例1的运行窗口

范例1包含了13个任务。在图F1.1“范例1的运行窗口”的左下角显示出来。μC/OS-II增加了2个内部任务——空闲任务和计算CPU利用率的统计任务。范例1的代码建立了11个其他任务。

范例1的源代码在SOURCE目录下的TEST.C文件中。在命令行方式下输入CD ..\SOURCE，将从TEST目录切换到SOURCE目录。TEST.C的部分代码在程序清单L1.1中列出，可以使用任何文本编辑软件打开TEST.C文件，进行对比。

程序清单L1.1 范例1，TEST.C

```
#include   "includes.h"                                                    (1)

#define        TASK_STK_SIZE            512                                (2)
#define        N_TASKS                   10
```

```
OS_STK        TaskStk[N_TASKS][TASK_STK_SIZE];        (3)
OS_STK        TaskStartStk[TASK_STK_SIZE];            (4)
char          TaskData[N_TASKS];                      (5)
OS_EVENT     *RandomSem;                              (6)
```

说明：为了描述程序清单和图表，在空白处添加了索引。这些索引值得引起注意。例如，L1.1(1)代表"请参阅程序清单 L1.1 标号(1)的代码"。这种方式同样应用在图表上，相应的 F3.1(2)表示"请参阅图 F3.1 并留意标号(2)的地方"。

L1.1(1)　首先注意，这里只有一句简单的 #include 语句。这是因为所有相关的头文件都放在了一个主控头文件 includes.h 中。每个文件通常都包含这个主控头文件，这样就无须考虑每个文件应该包含哪些相应的头文件，这些文件都通过 includes.h 包含进来了。可以使用文本编辑软件查看 includes.h 文件的内容，这个文件也在 SOURCE 目录下。

μC/OS-II 是一个多任务的实时内核，允许建立多达 63 个用户任务。μC/OS-II 根据程序建立和运行的情况，决定在什么时候从一个任务切换到另一个任务。任务的优先级是 μC/OS-II必须得到的信息之一。任务切换 task switch 又称为 context switch*，这 2 个名词在后面的章节中可能混用。

后面需要时，还会继续讲解程序清单 L1.1 的内容。现在来看通常的 C 程序都有的 main()函数，如程序清单 L1.2 所列。

程序清单 L1.2　范例 1，TEST.C，main()

```
void  main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);          (1)

    OSInit();                                                  (2)

    PC_DOSSaveReturn();                                        (3)
    PC_VectSet(uCOS, OSCtxSw);                                 (4)

    RandomSem = OSSemCreate(1);                                (5)

    OSTaskCreate(TaskStart, (void *)0, &TaskStartStk[TASK_STK_SIZE - 1], 0);   (6)

    OSStart();                                                 (7)
}
```

* context switch 在有些中文书中被翻译成上下文切换，其实是指 CPU 寄存器内容的切换，也就是任务切换，本书中都译成任务切换。——译者注

L1.2(1) main()程序从清整个屏幕开始，为的是保证屏幕上不留以前的DOS下的显示。PC_DispClrScr()函数可在PC.C文件中找到(见第18章："PC中的功能函数")。PC.C包含一些在DOS环境(或是Windows 95，98，Me，NT，2000及XP的MSDOS模式)中非常有用的功能。PC_的前缀使读者很容易判断在哪里找到这个函数的源代码。

注意：笔者定义了白色的字符和黑色的背景色。因为要清屏幕，所以可以只定义背景色而不定义前景色，但这样在返回DOS之后，就什么也看不见了。这也是为什么总要定义一个可见的前景色的原因。

L1.2(2) 在使用μC/OS-II提供的任何功能之前，必须调用OSInit()函数。OSInit()函数建立了2个任务：空闲任务——在所有其他任务均未就绪时运行；统计任务——计算CPU的利用率。

L1.2(3) 调用PC_DOSSaveReturn()函数保存当前的DOS环境。这使得用户可以返回到没有运行μC/OS-II之前的DOS环境。可参阅第18章："PC中的功能函数"，了解PC_DOSSaveReturn()函数的功能。

L1.2(4) 调用PC_VectSet()(见第18章："PC中的功能函数")，指定μC/OS-II中的任务切换处理函数。任务级的任务切换通过调用80x86的INT指令引发中断，通过其向量地址调用指定的处理函数。这里使用了0x80的向量(128)，因为DOS和BIOS都没有使用这个向量。

L1.2(5) 建立一个信号量，保护Borland C/C++库中产生随机数的函数。信号量是由μC/OS-II提供的一种机制，用于防止多个任务同时访问同一资源(产生随机数的函数)。之所以使用信号量，是因为假设产生随机数的函数不具备可重入性(这是一种安全的做法)。信号量初值设置为1，通知μC/OS-II在某一时刻只有一个任务可以调用随机数产生函数。OSSemCreate()函数建立该信号量，并指定信号量的初始值，该函数返回一个指向信号量的指针[L1.1(6)]，接下来对该信号量的操作都通过这个指针来实现。

L1.2(6) 在开始多任务之前，必须建立至少一个任务。在这个范例中，建立了一个叫做TaskStart()的任务。这个任务有4个参数：第1个参数是指向该任务运行代码的指针，在这里就是TaskStart()；第2个参数是一个指向任务初始化数据的指针，在这个范例中不需要任何初始化数据，所以传递了一个空(NULL)指针，在范例4中将会讨论任务初始化数据的用法；第3个参数是任务的堆栈栈顶TOS(Top-of-Stack)，对μC/OS-II和绝大多数的可剥夺型内核来说，每个任务都有各自的堆栈空间，在μC/OS-II中每个任务的堆栈容量可以单独指定，在这个范例中，为了简化起见，都定义成相同的容量。

在80x86处理器上，堆栈是从高地址向低地址递减的，所以必须把堆栈的高地址TOS传递给OSTaskCreate()函数。在这个范例中，堆栈空间被定义为TaskStartStk[]，并在编译时确定。堆栈必须定义为OS_STK变量类型[见L1.1(4)]。堆栈的容量在程序清单L1.1(2)中定义。对于80x86处理器，OS_STK变量类型是16位变量，所以堆栈大小是1 024 B。最后的参数指定建立的任务的优先级。这个参数的数值越小，任务的优先级越高(这一点非常

重要)。

前面提到,μC/OS-II 允许建立多达 63 个用户任务。必须给这些任务分配一个独一无二的介于 0～62 的优先级。应根据实际应用的需求,为每个任务分配不同的优先级,0 是最高的优先级。

L1.2(7)　调用 OSStart(),将控制权交给 μC/OS-II 内核,开始运行多任务。在启动多任务 OSStart()之前,至少要先建立一个任务。这一点非常重要,不这样做,应用程序就会崩溃。实际上,如果希望计算 CPU 的利用率时,也须先建立一个任务。

OSStart()将判断所有建立的任务中哪一个任务是最重要的(优先级最高的),并开始运行这个任务。在这个范例中,μC/OS-II 建立了 2 个低优先级的任务——空闲任务和统计任务;而 main()函数建立了 TaskStart()任务,并且分配了最高的优先级 0,这样 OSStart()将开始执行 TaskStart()任务。

读者也许注意到了,OSStart()并不返回 main()函数。无论如何,若调用 PC_DOSReturn(),那么多任务运行将被中止,并且返回到 DOS 环境(请注意不是返回到 main())。在嵌入式系统中,并不需要一个类似 PC_DOSReturn()的函数,因为在这种情况下并不希望返回到任何地方。

正如前面所述,OSStart()让优先级最高的就绪任务——TaskStart()任务开始运行。TaskStart()任务的代码列在程序清单 L1.3 中。

程序清单 L1.3　范例 1,TEST.C,TaskStart()

```
void  TaskStart (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    char       s[100];
    INT16S     key;

    pdata = pdata;                                          (1)

    TaskStartDispInit();                                    (2)

    OS_ENTER_CRITICAL();                                    (3)
    PC_VectSet(0x08, OSTickISR);                            (4)
    PC_SetTickRate(OS_TICKS_PER_SEC);                       (5)
    OS_EXIT_CRITICAL();                                     (6)
    OSStatInit();                                           (7)

    TaskStartCreateTasks();                                 (8)
```

```
    for (;;) {                                                          (9)
        TaskStartDisp();                                                (10)

        if (PC_GetKey(&key) == TRUE) {                                  (11)
            if (key == 0x1B) {                                          (12)
                PC_DOSReturn();                                         (13)
            }
        }

        OSCtxSwCtr = 0;                                                 (14)
        OSTimeDlyHMSM(0, 0, 1, 0);                                      (15)
    }
}
```

L1.3(1) TaskStart()首先将 pdata 的值赋给它自己。这样做实际上没有任何意义，只是为了避免有些编译器会警告（或者认为是错误）pdata 这个参数没有使用。换句话说，这里假装使用了 pdata，以避免编译器的警告。pdata 是当任务建立时传递过来的一个指针；但实际上在 OSTaskCreate()建立任务时，第 2 个参数是 0(NULL)，而不是一个指向任何数据的指针。这样实际上在建立任务的时候没有传递任何数据。

L1.3(2) TaskStart()调用 TaskStartDispInit()，以初始化屏幕显示，如图 F1.2 所示。

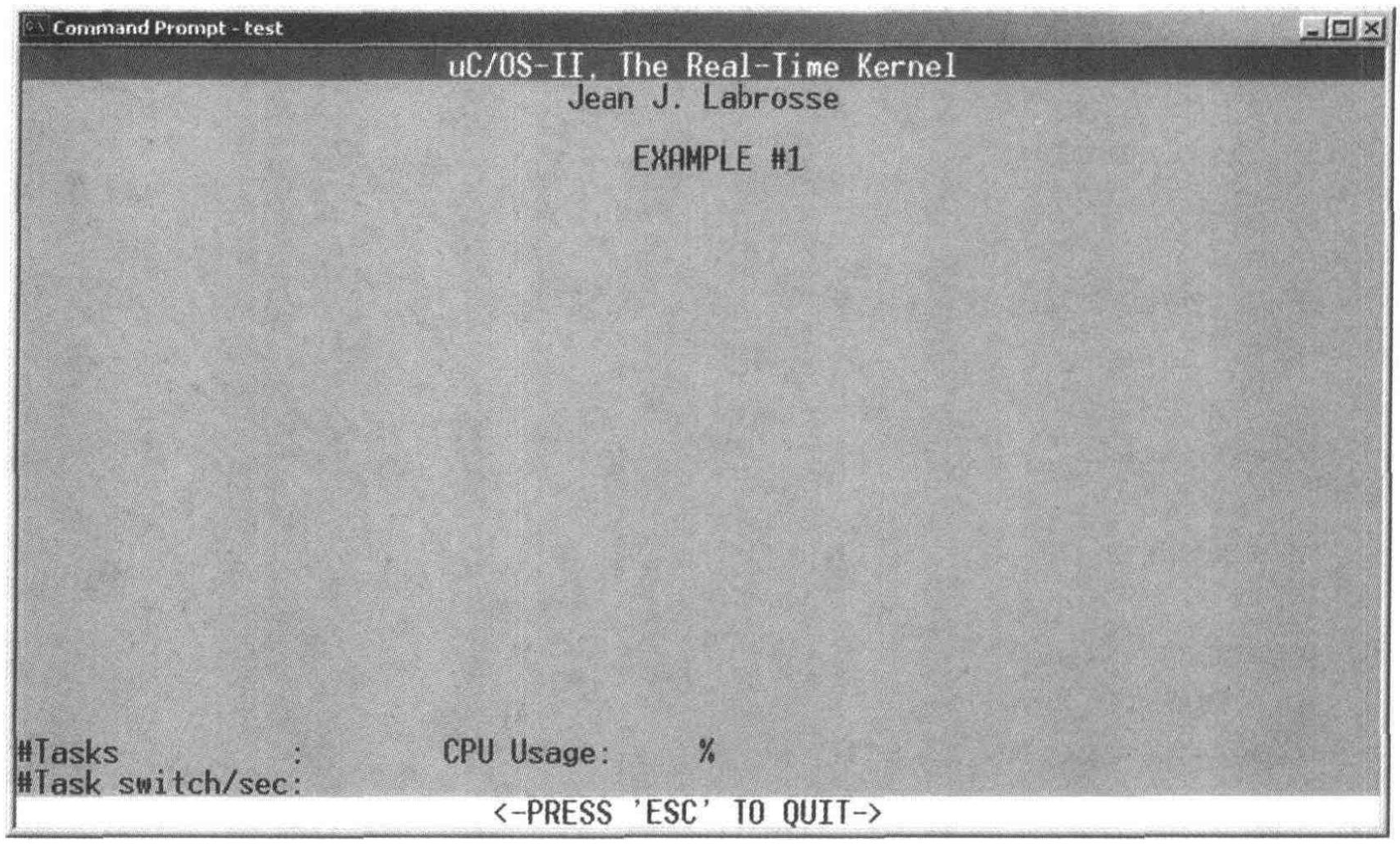


图 F1.2 TaskStartDispInit()初始化屏幕显示

TaskStartDispInit()包含了连续 25 次调用 PC_DispStr()(见第 18 章:"PC 中的功能函数"),在屏幕上打印出 25 个字符串,构成一个典型的 DOS 窗口。

L1.3(3)　TaskCreate()调用宏 OS_ENTER_CRITICAL()。OS_ENTER_CRITICAL()是一个同处理器密切相关的宏,用来禁止处理器响应中断请求,即关中断(见第 13 章:"移植 μC/OS - II")。

L1.3(4)　μC/OS - II 同其它所有的实时内核一样,都需要一个时钟源,以此作为延时和超时判断的定时依据。在实模式下,计算机提供了一个 18.206 48 Hz 的时钟中断,其周期为 54.925 ms,称为一个时钟节拍。PC_VectSet()可以把计算机本来为时钟节拍提供的中断服务程序替换成 μC/OS - II 需要用到的中断服务处理程序;但是 μC/OS - II 里每个时钟节拍仍然会调用原来的中断服务子程序。这是在 PC_DOSSaveReturn()里设置完成的(见第 18 章:"PC 中的功能函数")。

L1.3(5)　将时钟节拍的频率从 18.2 Hz 更改为 200 Hz。200 Hz 约为 18.2 Hz 的 11 倍,为后面的处理提供了方便。说实在的,真是搞不懂,为什么 IBM 当初的 PC 设计成使用 18.2 Hz 的频率而不是 20 Hz 的频率。看起来他们是让 82C54 的 16 位定时器每次自由运行溢出(65 536 次计数)产生了 18.2 Hz,而不是调整到 59 659 次计数,以得到精确的 20 Hz。通过调用 PC 功能函数 PC_SetTickRate(),更改时钟节拍的频率到 200 Hz(在 OS_CPU.H 中,OS_TICKS_PER_SEC已经被更改为 200)。

L1.3(6)　宏 OS_EXIT_CRITICAL()被调用。OS_EXIT_CRITICAL()也是一个和处理器密切相关的宏,用来允许处理器相应的中断请求(见第 13 章:"移植 μC/OS - II")。OS_ENTER_CIRITAL()和 OS_EXIT_CRITICAL()必须成对使用。

L1.3(7)　调用 OSStatInit()),测试所使用的处理器的速度*(见第 3 章:"初识 μC/OS - II")。μC/OS - II 通过这个函数得知处理器在运行所有应用任务时实际的 CPU 使用率。

L1.3(8)　TaskStart()接着调用 TaskStartCreateTasks()函数来建立更多的任务。实际上,这里添加了 N_TASKS [见程序清单 L1.1(2)]个相同的任务。函数 TaskStartCreateTasks()的代码列在程序清单 L1.4 中。在继续读程序清单 L1.3 之前,先来分析一下 TaskStartCreateTasks()的代码。

程序清单 L1.4　范例 1,TEST.C, TaskStartCreateTasks()

```
static  void  TaskStartCreateTasks (void)
{
    INT8U  i;
```

* 原书有误,应为"内核结构"。——译者注

```
    for (i = 0; i < N_TASKS; i++) {
        TaskData[i] = '0' + i;                                  (1)
        OSTaskCreate(Task,                                      (2)
                     (void *)&TaskData[i],                      (3)
                     &TaskStk[i][TASK_STK_SIZE - 1],            (4)
                     i + 1);                                    (5)
    }
}
```

L1.4(1) 初始化一个数组,包含0~9的ASCII字符[见程序清单L1.1(5)]。

L1.4(2) 这里的循环建立了N_TASKS个完全相同的任务Task(),Task()负责在屏幕的随机位置显示一个0~9的ASCII字符。实际上,这N_TASKS个任务每一个负责显示一个不同的字符。

L1.4(3) 每一个任务接受一个指向字符数组中的元素的指针,也就是每个任务接受了一个不同的字符。

L1.4(4) 每个任务都需要分配自己的堆栈空间[见程序清单L1.1(3)]。

L1.4(5) 在μC/OS-II中,必须给每个任务分配一个独一无二的优先级。因为优先级0已经被任务TaskStart()占用了,新建立的任务将使用1~10的优先级。

在每个任务建立的时候,μC/OS-II判断新建立的任务是否比建立它们的任务优先级更高。如果新建立的任务优先级更高,那么这个新建立的任务将立刻开始运行。在这个范例里,建立新任务的TaskStart()任务具有最高的优先级0,所以这些新建立的任务暂时不能运行。

现在接着分析程序清单L1.3。

L1.3(9) 在μC/OS-II中,每一个任务都是一个无限的循环。

L1.3(10) 调用TaskStartDisp()在DOS窗口的底部,显示相关信息(见图F1.1)。这些信息包括建立的任务个数、当前的CPU利用率、任务切换的次数、μC/OS-II的版本号及所使用的处理器是否包含浮点处理单元(FPU)。

L1.3(11) 通过调用PC_GetKey(),检查是否有按键动作。

L1.3(12)

L1.3(13) 判断是否按下Esc键,如果按下,则调用PC_DOSReturn(),返回DOS环境。见第18章:"PC中的功能函数",可以了解这个函数的具体细节。

L1.3(14) 如果没有按下Esc键,则记录任务切换次数的全局变量OSCtxSwCtr被清0,以便更新为下一个1 s内发生的任务切换次数。

L1.3(15) 最后,通过调用OSTimeDlyHMSM(),TaskStart()任务将自身挂起1 s。HMSM代表[小]时(hour)、分(minute)、秒(second)及毫秒(millisecond),这是传递给OSTimeDlyHMSM()的参数。因为TaskStart()任务将自身挂起1 s,μC/OS-II开始调度,找到下一个最高优先级的就绪任务。在这种情况

下就是优先级为 1 的 Task()任务*。请注意，若没有 OSTimeDlyHMSM()或其他类似的函数，TaskStart()将是一个真正的无限循环，任何其他的任务都没有运行的机会。

Task()任务的代码如程序清单 L1.5 所列。

程序清单 L1.5　范例 1，TEST.C，Task()

```
void  Task (void *pdata)
{
    INT8U  x;
    INT8U  y;
    INT8U  err;

    for (;;) {                                                          (1)
        OSSemPend(RandomSem, 0, &err);                                  (2)
        x = random(80);                                                 (3)
        y = random(16);                                                 (4)
        OSSemPost(RandomSem);                                           (5)

        PC_DispChar(x, y + 5, *(char *)pdata, DISP_FGND_LIGHT_GRAY);    (6)
        OSTimeDly(1);                                                   (7)
    }
}
```

L1.5(1)　正如前面提到的，任务是一个无限循环。

L1.5(2)　Task()任务首先检查 RandomSem 信号量。这个信号量用来防止多个任务同时访问 Borland 编译器提供的产生随机数的函数。查询信号量时，需要使用 OSSemPend()函数，并将需要查询的信号量的指针**作为参数传递给该函数。OSSemPend()的第 2 个参数用来定义超时时限。如果这个值为 0，代表任务将无限期地等待信号量。因为信号量在建立时被初始化为 1，并且没有其他的任务在使用该信号量，所以 Task()任务立刻得到该信号量并继续运行。如果该信号量被其他任务占用，μC/OS-II 将挂起这个任务并开始调度，找到下一个就绪任务中优先级最高的。

L1.5(3)　调用产生随机数的函数，返回一个 0～79(含 79)的数值。这个值用来确定在屏幕上显示字符的 x 方向的位置。

L1.5(4)　再次调用产生随机数的函数，返回一个 0～15(含 15)的数值。这个值用来确定在屏幕上显示字符的 y 方向的位置。

---

* 这里有 N_TASKS 个相同的 Task()任务，但是其优先级和初始化数据各不相同。——译者注

** 信号量的指针在建立信号量时得到。——译者注

L1.5(5) 调用 OSSemPost(),释放信号量。这里只须传递信号量的指针即可。

L1.5(6) 开始在屏幕上显示字符。这个字符在建立任务时作为初始化数据得到。对于第1个建立的任务,该字符是0;对于最后一个建立的任务,该字符是9。在显示字符时,y方向加了5行的偏移量,这样不会影响顶部显示的其他信息(见图F1.1)。

L1.5(7) 最后,Task()任务调用 OSTimeDly(),通知 μC/OS-II 该任务本次运行已经结束,可以让其他低优先级的任务运行了。参数1代表该任务将被延时1个时钟节拍,在200 Hz情况下就是5 ms。一旦调用 OSTimeDly(),μC/OS-II 将挂起该任务,并让下一个最高优先级的就绪任务运行。在这个范例中,就是下一个优先级为2的 Task()任务。所有的 Task()任务的处理过程都是这样,其运行结果就是图F1.1显示的情况。

如果在 C:\BC45 目录下安装了 Borland C/C++ V4.5x 编译器,可以试着修改 TEST.C。修改后,在命令行方式下进入 TEST 目录,输入 MAKETEST,即可编译、链接,得到更新过的 TEST.EXE。如果没有 Borland C/C++ V4.5x 编译器,或者安装在不同的目录下,则可以更改相应的 TEST.MAK,INCLUDES.H 及 TEST.LNK 文件。

SOURCE 目录包含4个文件:INCLUDES.H,OS_CFG.H,TEST.C 及 TEST.LNK。OS_CFG.H 文件用来配置 μC/OS-II 的编译选项;TEST.LNK 是 Borland 链接器 TLINK 使用的链接命令文件。

1.02 范例2

范例2演示了 μC/OS-II 的堆栈检查功能。每个任务已经使用了的堆栈空间和剩余的堆栈空间都可以显示出来。同时,范例2还显示堆栈检查函数 OSTaskStkChk()的运行时间,因为这个时间是与任务所指定的堆栈大小密切相关的。该范例表明,如果一个堆栈使用量很大,堆栈检查的时间就比较短。

这个范例的代码在安装盘的\SOFTWARE\uCOS-II\EX2_x86L\BC45 目录下,可以打开一个 MSDOS 窗口(在 Windows 2000 下称做命令行方式),输入:

```
CD \SOFTWARE\μCOS-II\Ex2_x86L\BC45\TEST
```

CD 命令将切换到范例2的 TEST 目录。要执行范例2,只需接着输入 TEST,于是 DOS 窗口下 TEST.EXE 程序开始运行。

大约1 s后,可以看到如图F1.3所示的窗口。

范例2包含了9个任务,在图F1.3的左下角有显示。μC/OS-II 增加了2个内部任务——空闲任务和计算 CPU 利用率的任务。范例2的代码建立了7个其他任务。

在范例1里,已经得到建立任务数、任务切换次数及 CPU 利用率等信息。范例2进一步演示了如何得到更多的任务统计数据。例如,展示了如何统计任务实际使用的堆栈空间大小以及如何确定统计堆栈的使用情况所花费的时间。

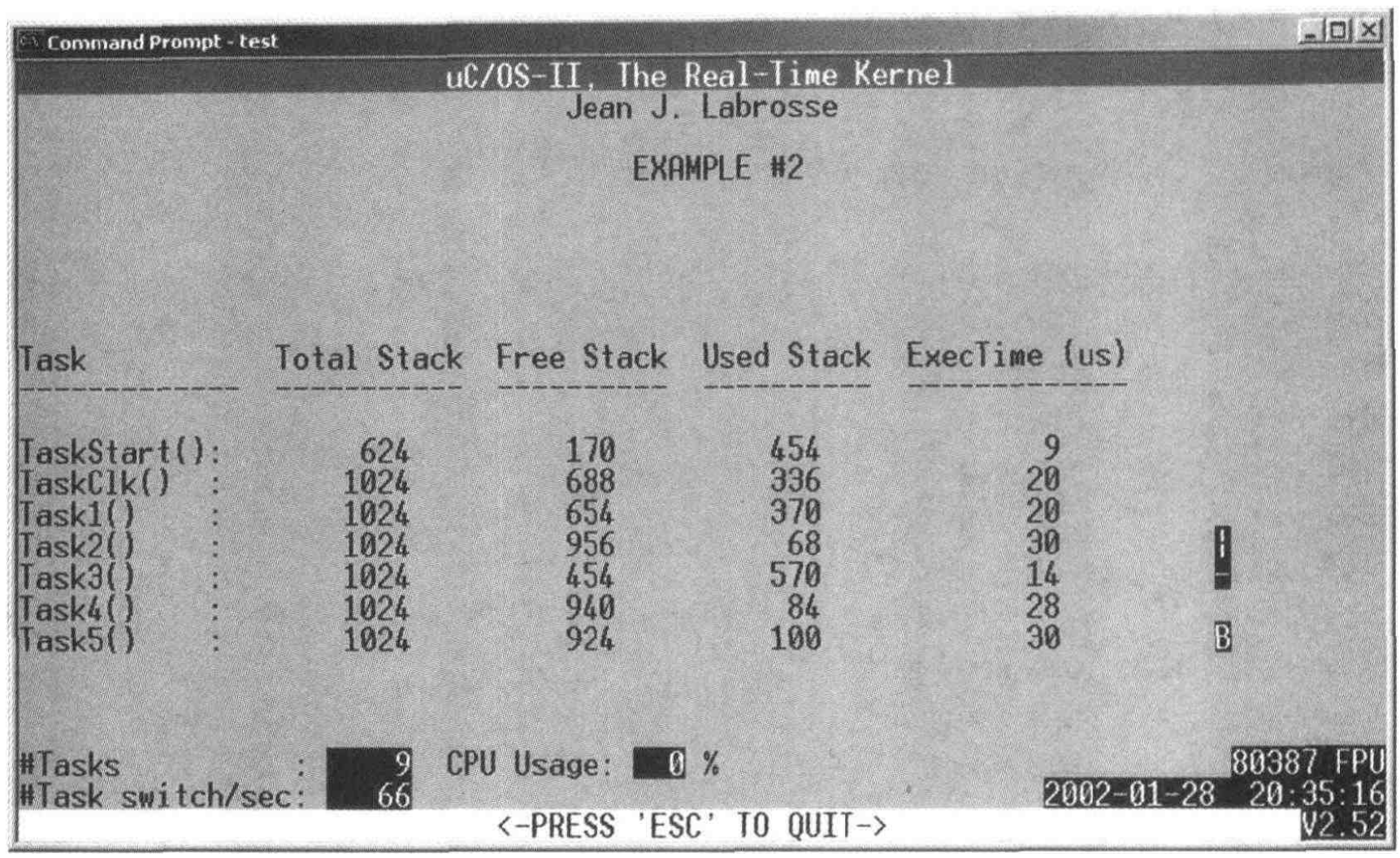


图 F1.3　范例 2 的显示

范例 2 使用了带扩展功能的任务建立函数 OSTaskCreateExt()和 μC/OS - II 的堆栈检查操作*。当不知道应该给任务分配多少堆栈空间时，堆栈检查功能是很有用的。在这个例子里，先分配足够的堆栈空间给任务，然后用堆栈检查操作查看任务到底需要多少堆栈空间。显然，任务要运行足够长的时间，并要考虑各种情况，才能得到正确数据。最后确定的堆栈大小，还须考虑系统今后的扩展，一般多分配 10%～25%，或者更多的栈空间。如果系统对稳定性要求高，则应该多 1 倍以上。通过堆栈检查操作，得到的是曾经被使用过的堆栈区域的大小，而不可能得到完全精确的堆栈实际使用情况。

μC/OS - II 的堆栈检查功能要求任务建立时给堆栈清 0。OSTaskCreateExt()可以执行此项操作(将选项 opt 参数 OS_TASK_OPT_STK_CHK 和 OS_TASK_OPT_STK_CLR 置为 1，允许此项操作)。如果任务运行过程中有建立、删除任务的操作，应该设置好上述选项，确保任务建立后堆栈是清空的。同时要意识到 OSTaskCreateExt()进行堆栈清 0 操作是一项很费时间的工作，其取决于堆栈的容量。

执行堆栈检查操作时，μC/OS - II 从栈底向栈顶搜索非 0 元素，同时用一个计数器记录 0 元素的个数。

范例 2 的源代码在 SOURCE 目录下的 TEST.C 文件中。在命令行方式下输入 CD ..\SOURCE，从 TEST 目录切换到 SOURCE 目录。TEST.C 的部分代码如程序清单 L1.6 中所列。可以使用任何文本编辑软件打开 TEST.C 文件，进行对比。

---

* 要使用堆栈检查操作必须用 OSTaskCreateExt()建立任务。——译者注

程序清单 L1.6 范例 2,TEST.C

```
#include "includes.h"                                          (1)

#define        TASK_STK_SIZE      512                           (2)

#define        TASK_START_ID      0                             (3)
#define        TASK_CLK_ID        1
#define        TASK_1_ID          2
#define        TASK_2_ID          3
#define        TASK_3_ID          4
#define        TASK_4_ID          5
#define        TASK_5_ID          6

#define        TASK_START_PRIO    10                            (4)
#define        TASK_CLK_PRIO      11
#define        TASK_1_PRIO        12
#define        TASK_2_PRIO        13
#define        TASK_3_PRIO        14
#define        TASK_4_PRIO        15
#define        TASK_5_PRIO        16

OS_STK      TaskStartStk[TASK_STK_SIZE];                       (5)
OS_STK      TaskClkStk[TASK_STK_SIZE];
OS_STK      Task1Stk[TASK_STK_SIZE];
OS_STK      Task2Stk[TASK_STK_SIZE];
OS_STK      Task3Stk[TASK_STK_SIZE];
OS_STK      Task4Stk[TASK_STK_SIZE];
OS_STK      Task5Stk[TASK_STK_SIZE];

OS_EVENT    *AckMbox;                                          (6)
OS_EVENT    *TxMbox;
```

根据范例 1 的知识，可以看出：

L1.6(1) 包含主控头文件 includes.h。

L1.6(2) 每个堆栈的容量由 TASK_STK_SIZE 决定，同样，这里的所有任务堆栈的容量都设定成相同值，但是 μC/OS-II 允许为每个任务指定不同容量的堆栈。

L1.6(3) 为堆栈分配实际存储空间。

范例 2 的 main()函数代码列在程序清单 L1.7 中，它和范例 1 的 main()函数差不多。下面只重点讲解不同的地方。

程序清单 L1.7　范例 2,TEST.C,main()

```
void main (void)
{
    OS_STK *ptos;
    OS_STK *pbos;
    INT32U  size;

    PC_DispClrScr(DISP_FGND_WHITE);

    OSInit();

    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);

    PC_ElapsedInit();                                                    (1)

    ptos        = &TaskStartStk[TASK_STK_SIZE - 1];                      (2)
    pbos        = &TaskStartStk[0];
    size        = TASK_STK_SIZE;
    OSTaskStkInit_FPE_x86(&ptos, &pbos, &size);                          (3)
    OSTaskCreateExt(TaskStart,                                           (4)
                    (void *)0,
                    ptos,                                                (5)
                    TASK_START_PRIO,                                     (6)
                    TASK_START_ID,                                       (7)
                    pbos,                                                (8)
                    size,                                                (9)
                    (void *)0,                                           (10)
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);          (11)

    OSStart();
}
```

L1.7(1)　main()函数调用 PC_ElapsedInit()函数,初始化时间测量功能。这个功能用来计算 OSTaskStkChk()函数的运行时间,精确地记录 PC_ElapsedStart()和 PC_ElapsedStop()函数的调用时刻。通过计算这 2 个时刻的差值,可以很容易地得到这 2 个时刻之间的执行代码的运行时间。

L1.7(2)

L1.7(3)　范例 2 中的 TaskStart()任务调用浮点运算仿真程序库,而不是使用浮点处理器 FPU,因为只有在 80486 和更高级别的计算机中才有这一部分。如果系统中没有浮点处理器,Borland 的编译器将默认使用其提供的浮点运算仿真程序库。换句话说,如果在一台安装了 DOS 系统的 80386EX(没有安装 80387 浮

点处理器)的计算机上运行范例2的TEST.EXE程序,那么浮点运算将由仿真实现。浮点运算仿真程序库的程序都是不可重入的,必须找到某些方法,以使多个任务都能够进行浮点运算。范例2中通过对堆栈的修改,满足浮点运算仿真程序库的要求。这个修改是通过OSTaskStkInit_FPE_x86()函数来完成的(见第14章:"μC/OS-II在80x86上的移植")。注意在图F1.3中,报告的任务堆栈容量是624 B,而不是建立时定义的1 024 B。这是因为函数OSTaskStkInit_FPE_x86()把一部分堆栈保留给浮点运算仿真程序库使用了。

L1.7(4) 这里没有使用OSTaskCreate()函数来建立TaskStart(),而是使用了函数OSTaskCreateExt()。这个函数是OSTaskCreate()函数的功能扩展函数。只有这个功能扩展的函数才能支持对堆栈的修改和运行时对堆栈容量的检查。

L1.7(5) OSTaskStkInit_FPE_x86()函数修改了堆栈顶指针(top-of-stack pointer)。这里必须把修改后的堆栈顶指针传递给OSTaskCreateExt()函数,以建立任务。

L1.7(6) 这里使用了"#define"定义一个常量的方法,替代范例1中使用立即数给任务赋优先级的方法[见L1.6(4)]。这样对程序的维护和修改都更方便。

L1.7(7) OSTaskCreateExt()函数要求一个任务标志符(task identifier)。现在这个参数可以为任意值,因为范例2中μC/OS-II并没有使用这个参数。

L1.7(8) OSTaskStkInit_FPE_x86()函数还修改了堆栈底指针(bottom-of-stack pointer)。这里必须把修改后的堆栈底指针传递给OSTaskCreateExt()函数,以建立任务。

L1.7(9) OSTaskStkInit_FPE_x86()函数还修改了堆栈的容量。这里必须把修改后的堆栈容量传递给OSTaskCreateExt()函数,以建立任务。

L1.7(10) OSTaskCreateExt()还需要一个指向用户定义的TCB扩展数据结构的指针。由于这个范例中该指针并未使用,所以只简单地传递了一个空指针。

L1.7(11) OSTaskCreateExt()的最后一个参数是一个选项。这里的设置表明,现在建立的这个任务允许堆栈检查,并且需要在任务建立时将堆栈清0。

TaskStart()任务和范例1的对应任务也非常相似,其代码列在程序清单L1.8中。这里也只说明和范例1不同的地方。

程序清单L1.8 范例2,TEST.C,TaskStart()

```
void  TaskStart (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT16S     key;
    pdata = pdata;
```

```
    TaskStartDispInit();                                                    (1)

    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(OS_TICKS_PER_SEC);
    OS_EXIT_CRITICAL();

    OSStatInit();

    AckMbox = OSMboxCreate((void *)0);                                      (2)
    TxMbox  = OSMboxCreate((void *)0);

    TaskStartCreateTasks();                                                 (3)

    for (;;) {
        TaskStartDisp();

        if (PC_GetKey(&key)) {
            if (key == 0x1B) {
                PC_DOSReturn();
            }
        }

        OSCtxSwCtr = 0;
        OSTimeDly(OS_TICKS_PER_SEC);                                        (4)
    }
}
```

L1.8(1)　虽然这个函数的调用看起来和范例 1 中是一样的，但是 TaskStartDispInit()会把屏幕初始化，如图 F1.4 所示。

L1.8(2)　μC/OS-II 允许任务或中断与另外的任务通信。在范例 2 中，任务 4 将向任务 5 发送消息，并且任务 5 会回复一个应答消息(后面由消息描述)。为了实现这个通信过程，这里建立了 2 个由 μC/OS-II 提供的通信工具——邮箱。邮箱允许任务或中断向另一个任务发送指针变量。邮箱仅能提供一个指针型变量的传递。这个指针型变量指向什么样的数据结构，是由用户的应用程序决定的。当然发送邮箱的任务和接收邮箱的任务对这个指针必须使用相同的定义。

L1.8(3)　TaskStartCreateTasks()函数使用 OSTaskCreateExt()函数建立了另外的 6 个任务。这些任务不进行浮点运算，也就没有必要使用 OSTaskStkInit_FPE_x86()函数来更改堆栈；但是，在这些任务中仍然进行了堆栈检查，所以必须使用 OSTaskCreateExt()函数。

L1.8(4)　在范例 1 中，使用了 OSTimeDlyHMSM()函数，将 TaskStart()任务挂起 1 s；

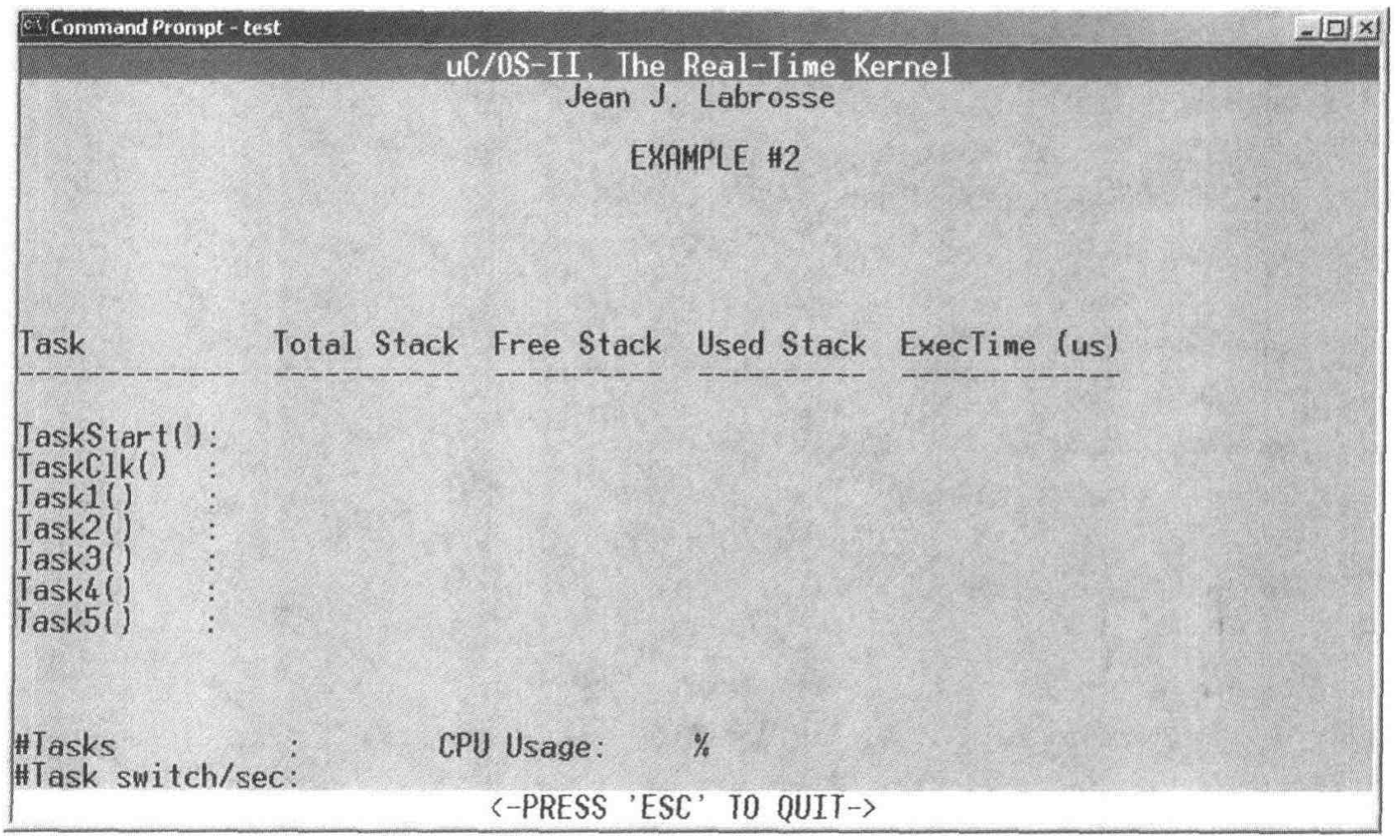


图 F1.4 范例 2 的初始化屏幕

这里使用了OSTimeDly(OS_TICKS_PER_SEC),以完成同样的功能。这2种方式都是可以的,区别仅在于OSTimeDly()函数执行时会稍稍快一点。

Task1()的代码列在程序清单L1.9中,这个任务检查建立的7个任务的堆栈使用情况。这7个任务是,由TaskStart()建立的6个任务和TaskStart()任务本身。

程序清单 L1.9 范例 2,TEST.C, Task1()

```
void  Task1 (void  *pdata)
{
    INT8U           err;
    OS_STK_DATA     data;
    INT16U          time;
    INT8U           i;
    char            s[80];

    pdata = pdata;
    for (;;) {
        for (i = 0; i < 7; i++) {
            PC_ElapsedStart();                                          (1)
            err  = OSTaskStkChk(TASK_START_PRIO + i, &data);            (2)
            time = PC_ElapsedStop();                                    (3)
            if (err == OS_NO_ERR) {
```

```
            sprintf(s, "%4ld        %4ld        %4ld        %6d",                    (4)
                    data.OSFree + data.OSUsed,
                    data.OSFree,
                    data.OSUsed,
                    time);
            PC_DispStr(19, 12 + i, s, DISP_FGND_YELLOW);                             (5)
        }
    }
    OSTimeDlyHMSM(0, 0, 0, 100);                                                     (6)
  }
}
```

L1.9(1)

L1.9(3)　为了测量 OSTaskStkChk()函数的运行时间，需要把这个函数放在 PC_ElapsedStart()和 PC_ElapsedStop() 2 个函数之间。PC_ElapsedStop()函数将返回以 ms 计量的时间间隔。

L1.9(2)　OSTaskStkChk()是由 μC/OS - II 提供的，是用来检查任务堆栈使用情况的函数。

调用这个函数时，需将待检查的任务的优先级作为代入参数。另外，这个函数检查的结果须存放在一个数据结构 OS_STK_DATA 中，这个数据结构包含了堆栈已经使用和尚未使用的字节数目。所以在调用时，须传递一个指向 OS_STK_DATA 数据结构的指针，作为第 2 个参数。OSTaskStkChk()函数返回一个出错代码，以指示该函数调用是否成功。

L1.9(4)

L1.9(5)　把由 OSTaskStkChk()得到的统计结果打印出来。

L1.9(6)　在范例 2 中，该任务每秒运行 10 次；而在实际应用中，可能数秒才进行一次堆栈检查。

浪费宝贵的 CPU 的运行时间去检查最坏情况下的堆栈的变化情况，是没有意义的。

Task2()和 Task3()的代码列在程序清单 L1.10 中。这 2 个任务都向屏幕输出一个由字符"|","\","—","/"交替出现而构成的转动的标志。这 2 个任务基本上是相同的。Task3()分配并且初始化了一个 500 B 的数组。这就占用了 Task3()的堆栈，在程序运行后可以看到，Task3()的空闲堆栈比 Task2()少了 502 B(500 B 的数组和一个 2 B 的整型变量)。Task2()的标志每秒转动 5 次，而 Task3()的标志每秒转动 2.5 次。Task4()和 Task5()的代码在程序清单 L1.11 中列出。

注意：在 Windows 95，98，Me，NT，2000 或 XP 的命令行窗口中运行这个程序时，标志的转动可能会比较慢。只要按 Alt+Enter 键，将这个命令窗口扩展为全屏幕方式，标志的转动就正常了。要返回窗口模式也可以按 Alt+Enter 键。

程序清单 L1.10　范例 2，TEST.C，Task2()和 Task3()

```
void  Task2 (void *data)
{
    data = data;
    for (;;) {
        PC_DispChar(70, 15, '|',  DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
        PC_DispChar(70, 15, '/',  DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
        PC_DispChar(70, 15, '-', DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
        PC_DispChar(70, 15, '\\', DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
    }
}

void  Task3 (void *data)
{
    char    dummy[500];
    INT16U  i;

    data = data;
    for (i = 0; i < 499; i++) {
        dummy[i] = 'Y';
    }
    for (;;) {
        PC_DispChar(70, 16, '|',  DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '\\', DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '-', DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '/',  DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
    }
}
```

程序清单 L1.11　范例 2，TEST.C，Task4()和 Task5()

```
void  Task4 (void *data)
{
    char   txmsg;
    INT8U  err;

    data   = data;
    txmsg  = 'A';
    for (;;) {
        OSMboxPost(TxMbox, (void *)&txmsg);                                  (1)
        OSMboxPend(AckMbox, 0, &err);                                        (2)
        txmsg ++ ;                                                           (3)
        if (txmsg == 'Z') {
            txmsg = 'A';
        }
    }
}

void  Task5 (void *data)
{
    char   *rxmsg;
    INT8U   err;

    data = data;
    for (;;) {
        rxmsg = (char *)OSMboxPend(TxMbox, 0, &err);                         (4)
        PC_DispChar(70, 18, *rxmsg, DISP_FGND_YELLOW + DISP_BGND_RED);       (5)
        OSTimeDlyHMSM(0, 0, 1, 0);                                           (6)
        OSMboxPost(AckMbox, (void *)1);                                      (7)
    }
}
```

L1.11(1)　Task4()向邮箱 TxMbox 发送一个 ASCII 字符，由 Task5()接收。

L1.11(2)　Task4()等待 Task5()的应答，Task5()的应答发送到邮箱 AckMbox 中。OSMboxPend()函数的第 2 个参数指定了等待超时的时限，参数值为 0，表示将永远等待下去；如果指定一个非 0 的值，那么在这个指定数目的时钟节拍过后，Task4()将不再等待而继续运行。

L1.11(3)　当 Task5()应答以后，更新发送到 TxMbox 邮箱的消息。

L1.11(4)　Task5()运行时，无限期等待 TxMbox 邮箱的消息。

L1.11(5)　当邮箱有消息(一个 ASCII 字符)时，Task5()将这个消息显示到屏幕上。

L1.11(6)

L1.11(7) Task5()等待1 s后给出应答，这样可以看出屏幕的变化情况。实际上，Task4()或Task5()必须有一定时间的等待，否则比这2个任务优先级更低的任务将没有机会运行。

最后，TaskClk()的代码列在程序清单L1.12中。这个任务每秒运行1次，通过调用PC_GetDateTime()(见第18章："PC中的功能函数")得到计算机当前的日期和时间，并且显示在屏幕上。

程序清单 L1.12 范例2，TEST.C，TaskClk()

```
void  TaskClk (void *data)
{
    char s[40];


    data = data;
    for (;;) {
        PC_GetDateTime(s);
        PC_DispStr(60, 23, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

如果在C:\BC45目录下安装了Borland C/C++ V4.5x编译器，可以试着修改TEST.C。修改完以后，在命令行方式下进入TEST目录，输入MAKETEST，即可编译、链接，得到更新过的TEST.EXE。

如果没有Borland C/C++ V4.5x编译器，或者安装在不同的目录下，则可以修改相应的TEST.MAK、INCLUDES.H及TEST.LNK文件。

SOURCE目录包含4个文件：INCLUDES.H，OS_CFG.H，TEST.C及TEST.LNK。OS_CFG.H文件用来配置μC/OS-II的编译选项；TEST.LNK是Borland链接器TLINK使用的链接命令文件。

1.03 范例3

范例3中使用了许多μC/OS-II提供的附加功能。范例3使用了OSTaskCreateExt()中TCB的扩展数据结构、用户定义的任务切换对外接口函数OSTaskSwHook()、用户定义的统计任务(statistic task)的对外接口函数OSTaskStatHook()以及消息队列。在这个范例里可以看到，如何确定每个任务运行的次数及每个任务的运行时间。任务的执行时间可以用来确定每个任务的CPU使用率。

这个范例的代码在安装盘的\SOFTWARE\uCOS-II\EX3_x86L\BC45目录下，可以打

开一个 MSDOS 窗口(在 Windows 2000 下称做命令行方式),输入:

CD \SOFTWARE\μCOS-II\Ex3_x86L\BC45\TEST

CD 命令将当前目录切换到范例 3 的 TEST 目录。只需接着输入 TEST,就可以开始执行范例 3 代码,于是 DOS 窗口开始运行 TEST.EXE 程序。

程序执行 1 s 后,可以看到如图 F1.5 所示的窗口。

计数(Counter)栏显示程序中 7 个任务各自的运行次数,执行时间(Execution Time)栏显示每个任务本次运行的时间(以 μs 计量),累计时间(Total Execution Time)栏显示从程序开始运行每个任务的累计运行时间,相对 CPU 占用率栏显示每个任务累计运行时间相对其他任务的百分比。

范例 3 包括 9 个任务——μC/OS-II 建立了空闲任务(idle task)和统计任务(statistic task),在图 F1.5 的左下角可以看出,范例 3 的代码建立了另外 7 个任务。范例 3 中TEST.C 的源代码见程序清单 L1.13。

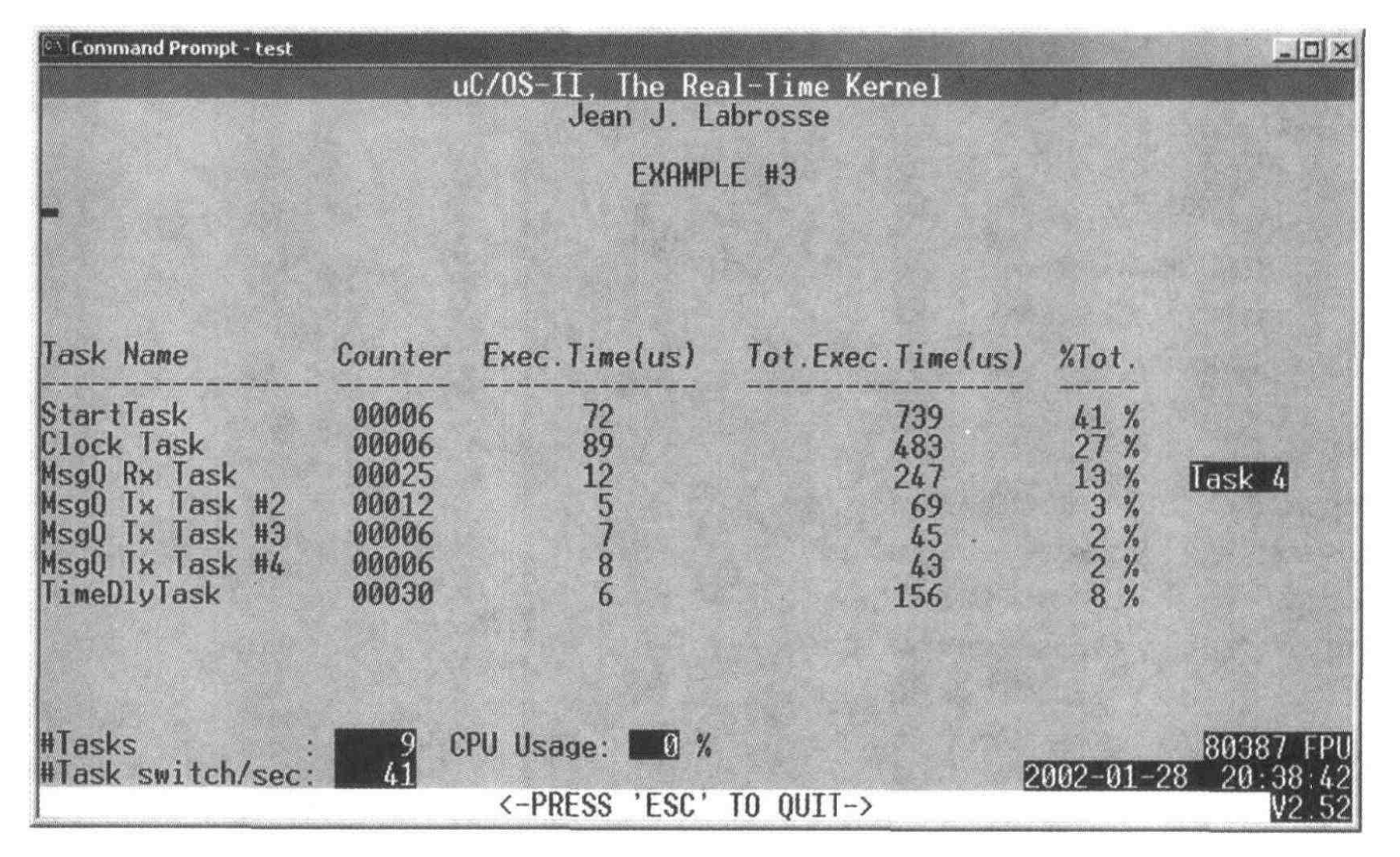


图 F1.5　范例 3 的显示

程序清单 L1.13　范例 3,TEST.C

```
#include   "includes.h"

#define          TASK_STK_SIZE        512

#define          TASK_START_ID          0
#define          TASK_CLK_ID            1
```

```
#define        TASK_1_ID          2
#define        TASK_2_ID          3
#define        TASK_3_ID          4
#define        TASK_4_ID          5
#define        TASK_5_ID          6

#define        TASK_START_PRIO    10
#define        TASK_CLK_PRIO      11
#define        TASK_1_PRIO        12
#define        TASK_2_PRIO        13
#define        TASK_3_PRIO        14
#define        TASK_4_PRIO        15
#define        TASK_5_PRIO        16

#define        MSG_QUEUE_SIZE     20

typedef struct {                                                    (1)
    char     TaskName[30];
    INT16U   TaskCtr;
    INT16U   TaskExecTime;
    INT32U   TaskTotExecTime;
} TASK_USER_DATA;

OS_STK          TaskStartStk[TASK_STK_SIZE];
OS_STK          TaskClkStk[TASK_STK_SIZE];
OS_STK          Task1Stk[TASK_STK_SIZE];
OS_STK          Task2Stk[TASK_STK_SIZE];
OS_STK          Task3Stk[TASK_STK_SIZE];
OS_STK          Task4Stk[TASK_STK_SIZE];
OS_STK          Task5Stk[TASK_STK_SIZE];

TASK_USER_DATA  TaskUserData[7];                                    (2)

OS_EVENT        *MsgQueue;                                          (3)
void            *MsgQueueTbl[20];
```

范例3的TEST.C文件的部分代码如程序清单L1.13中所列。可以使用任何文本编辑软件，阅读所有代码。

L1.13(1) 首先建立了一个TASK_USER_DATA数据结构，用来描述任务额外的信息。这个结构可以用来指定任务的名称（μC/OS-II并没有提供这样的特性）、记录任务运行的次数和本次运行的时间及总的运行时间。

L1.13(2) 定义了一个TASK_USER_DATA数据结构的数组，每一个建立的任务对应

其中的一个数据结构。空闲任务和统计任务不需要这样的数据结构。

L1.13(3)　除了信号量和邮箱外，μC/OS-II 还提供了消息队列的通信方式。消息队列和邮箱非常相似，只是消息队列可以容纳多则消息，而邮箱只能容纳一则消息。使用消息队列，一个任务或中断可以很容易与另一个任务通信。每则消息都是一个指针，但是这个指针指向的内容是由用户的应用程序决定的，发送消息和处理消息的任务必须约定好这个指针的用法。建立消息队列需要 2 个元素：一个 OS_EVENT 的数据结构（在后面的章节会详细描述）和一串指针。消息队列的深度决定于这一串指针的数目，在范例 3 中，消息队列的深度是 20。

Main()函数的代码列在程序清单 L1.14 中。这里仍然只讨论范例 1 和范例 2 中没有出现过的新特性。

程序清单 L1.14　范例 3，TEST.C，main()

```
void  main (void)
{
    PC_DispClrScr(DISP_BGND_BLACK);

    OSInit();

    PC_DOSSaveReturn();

    PC_VectSet(uCOS, OSCtxSw);

    PC_ElapsedInit();

    strcpy(TaskUserData[TASK_START_ID].TaskName, "StartTask");                    (1)
    OSTaskCreateExt(TaskStart,
                    (void *)0,
                    &TaskStartStk[TASK_STK_SIZE - 1],
                    TASK_START_PRIO,
                    TASK_START_ID,
                    &TaskStartStk[0],
                    TASK_STK_SIZE,
                    &TaskUserData[TASK_START_ID],                                 (2)
                    0);
    OSStart();
}
```

L1.14(1)　在一个任务建立之前，使用 ANSI C 规定的字符串处理函数 strcpy()为任务命名。任务的名称保存在为这个任务划分的 TASK_USER_DATA 数据结构中。

L1.14(2) 使用 OSTaskCreateExt()函数建立 TaskStart()任务，并且将一个指向分配给该任务的 TASK_USER_DATA 数据结构的指针作为参数传递进去。在 μC/OS-II 中，每个任务的 TCB 控制块都可以保存一个用户定义的数据结构的指针（见第3章："内核结构"）。这个特性允许扩展 μC/OS-II 的功能。下面的程序就是一个很好的例子。

TaskStart()任务的代码列在程序清单 L1.15 中。

程序清单 L1.15　范例 3，TEST.C，TaskStart()

```
void  TaskStart (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT16S     key;

    pdata = pdata;

    TaskStartDispInit();

    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(OS_TICKS_PER_SEC);
    OS_EXIT_CRITICAL();

    OSStatInit();

    MsgQueue = OSQCreate(&MsgQueueTbl[0], MSG_QUEUE_SIZE);                    (1)

    TaskStartCreateTasks();                                                   (2)

    for (;;) {
        TaskStartDisp();

        if (PC_GetKey(&key)) {
            if (key == 0x1B) {
                PC_DOSReturn();
            }
        }

        OSCtxSwCtr = 0;
```

```
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

L1.15(1)　建立了 Task1(),Task2(),Task3()及 Task4()使用的消息队列。

L1.15(2)　像范例 2 一样,TaskStartCreateTasks()建立了另外 6 个任务。不同之处在于,建立任务时,为每个任务分配了一个 TASK_USER_DATA 数据类型。并且用在建立 TaskStart()时的方法,为每个任务指定了任务名称。

当 TaskStart()调用 OSTimeDly(OS_TICKS_PER_SEC)后,μC/OS-II 立刻开始调度,让最高优先级的就绪任务运行,在范例 3 中就是 Task1()。程序清单 L1.16 中列出了 Task1(), Task2(), Task3()及 Task4()的代码,下面将一一讲述。

程序清单 L1.16　范例 3,TEST.C, Task1()～Task4()

```
void  Task1 (void *pdata)
{
    char   *msg;
    INT8U   err;


    pdata = pdata;
    for (;;) {
        msg = (char *)OSQPend(MsgQueue, 0, &err);                          (1)
        PC_DispStr(70, 13, msg, DISP_FGND_YELLOW + DISP_BGND_BLUE);        (2)
        OSTimeDlyHMSM(0, 0, 0, 100);                                       (3)
    }
}

void  Task2 (void *pdata)
{
    char  msg[20];


    pdata = pdata;
    strcpy(&msg[0], "Task 2");
    for (;;) {
        OSQPost(MsgQueue, (void *)&msg[0]);                                (4)
        OSTimeDlyHMSM(0, 0, 0, 500);                                       (5)
    }
}
```

```
void  Task3 (void *pdata)
{
    char  msg[20];


    pdata = pdata;
    strcpy(&msg[0], "Task 3");
    for (;;) {
        OSQPost(MsgQueue, (void *)&msg[0]);                          (6)
        OSTimeDlyHMSM(0, 0, 0, 500);
    }
}

void  Task4 (void *pdata)
{
    char  msg[20];


    pdata = pdata;
    strcpy(&msg[0], "Task 4");
    for (;;) {
        OSQPost(MsgQueue, (void *)&msg[0]);                          (7)
        OSTimeDlyHMSM(0, 0, 0, 500);
    }
}
```

L1.16(1) Task1()无限期地等待消息队列的消息。

L1.16(2) 接收到一则消息,在屏幕上显示。

L1.16(3) 任务延时 100 ms,以便能看到消息已经接到。

L1.16(4) Task2()发送一个指向字符串"Task 2"的指针到消息队列。

L1.16(5) Task2()延时 0.5 s,再次发送消息。

L1.16(6)

L1.16(7) Task3()和 Task4()同样发送各自的消息,并延时 0.5 s。

另外一个任务 Task5()只是简单地每次将自己延时 1/10 s。在 μC/OS-II 里,每个任务必须使用 μC/OS-II 提供的功能函数进行延时或者等待特定事件的发生。如果一个任务里没有这样的步骤,那么所有比这个任务优先级低的任务将不可能运行。

最后,调用 TaskClk()函数显示当前日期和时间。

程序运行时,产生的事件很难通过阅读每个任务的代码来确定。μC/OS-II 是提供源代码的,这样很容易通过 μC/OS-II 提供的接口函数(Hook 函数)添加新的功能代码。在μC/OS-II V2.52 中,共有 9 个这样的接口函数。这些接口函数的函数原型如程序清单L1.17所列。

程序清单 L1.17　μC/OS-II 接口函数

```
void  OSInitHookBegin(void);
void  OSInitHookEnd(void);
void  OSTaskCreateHook(OS_TCB *ptcb);
void  OSTaskDelHook(OS_TCB *ptcb);
void  OSTaskIdleHook(void);
void  OSTaskStatHook(void);
void  OSTaskSwHook(void);
void  OSTCBInitHook(OS_TCB *ptcb);
void  OSTimeTickHook(void);
```

这些接口函数都在 OS_CPU_C.C 文件中，通常由为处理器移植 μC/OS-II 的人完成；但是，如果将配置常数 OS_CPU_HOOKS_EN 定义为 0，就可以在另一个文件里重新声明这些接口函数。OS_CPU_HOOKS_EN 是在 OS_CFG.H 文件中提供的众多的配置常数之一。每个使用 μC/OS-II 的项目，其要求各不相同，其 OS_CFG.H 文件也不相同。本书中每个范例都在 SOURCE 目录下提供了该范例的 OS_CFG.H 文件。

在范例 3 中，配置常数 OS_CPU_HOOKS_EN 定义为 0，并且在 TEST.C 文件中重新定义了这些接口函数。TEST.C 的代码列在程序清单 L1.18 中。可以看出这 7 个接口函数没有任何实际功能，只是一些空函数。

程序清单 L1.18　范例 3，TEST.C，空接口函数

```
void  OSInitHookBegin (void)
{
}

void  OSInitHookEnd (void)
{
}

void  OSTaskCreateHook (OS_TCB *ptcb)
{
    ptcb = ptcb;
}

void  OSTaskDelHook (OS_TCB *ptcb)
{
    ptcb = ptcb;
}

void  OSTaskIdleHook (void)
```

```
{
}

void  OSTCBInitHook (OS_TCB *ptcb)
{
    ptcb = ptcb;
}

void  OSTimeTickHook (void)
{
}
```

OSTaskSwHook()接口函数代码列在程序清单 L1.19 中。该接口函数允许计算每个任务的运行时间，记录每个任务的运行次数和总的运行时间。每当 μC/OS-II 从一个低优先级的任务切换到一个高优先级的任务时，该接口函数被调用。

程序清单 L1.19　OSTaskSwHook()接口函数

```
void  OSTaskSwHook (void)
{
    INT16U              time;
    TASK_USER_DATA    *puser;


    time  = PC_ElapsedStop();                                    (1)
    PC_ElapsedStart();                                           (2)
    puser = OSTCBCur->OSTCBExtPtr;                               (3)
    if (puser != (TASK_USER_DATA *)0) {                          (4)
        puser->TaskCtr++;                                        (5)
        puser->TaskExecTime      = time;                         (6)
        puser->TaskTotExecTime += time;                          (7)
    }
}
```

L1.19(1)　通过调用 PC_ElapsedStop()函数，得到当前被切换的任务的运行时间。

L1.19(2)　当下一个任务被切换到运行态之前，调用 PC_ElapsedStart()，记录当时的时刻。第1次任务切换时得到的值是不正确的，但这并不是什么要紧的错误。

L1.19(3)　当 OSTaskSwHook()被调用时，全局变量 OSTCBCur 指向当前任务的 TCB 任务控制块，而 OSTCBHighRdy 指向即将被调度运行的任务的 TCB 任务控制块。在范例3中，OSTCBHighRdy 并没有用到。OSTaskSwHook()通过 OSTCBCur 得到当前任务在建立时指定的 TASK_USER_DATA 数据结构的指针。

L1.19(4)　检查该任务是否正确定义了一个 TASK_USER_DATA 数据结构。此情况有可能发生，实际上空闲任务和统计任务就没有定义 TASK_USER_DATA 数据结构。

L1.19(5)　任务运行次数加 1。这个记录可以用来判断这个任务是否曾经运行过。

L1.19(6)　任务的本次运行时间记录在 TASK_USER_DATA 数据结构中。

L1.19(7)　任务的总的运行时间也记录在 TASK_USER_DATA 数据结构中，这个记录可以用来判断几个任务之间的相对 CPU 占用率。

当允许使用统计任务时(OS_CFG.H 文件中的 OS_TASK_STAT_EN 配置常数设为 1)，统计任务将首先调用定义的 OSTaskStatHook()接口函数，范例 3 中的该接口函数的代码列在程序清单 L1.20 中。统计任务每秒运行一次，该接口函数也是每秒运行一次。

程序清单 L1.20　OSTaskStatHook()统计任务接口函数

```
void  OSTaskStatHook (void)
{
    char    s[80];
    INT8U   i;
    INT32U  total;
    INT8U   pct;

    total = 0L;
    for (i = 0; i < 7; i++) {
        total += TaskUserData[i].TaskTotExecTime;                          (1)
        DispTaskStat(i);
    }                                                                      (2)
    if (total > 0) {
        for (i = 0; i < 7; i++) {
            pct = 100 * TaskUserData[i].TaskTotExecTime / total;           (3)
            sprintf(s, " %3d %%", pct);
            PC_DispStr(62,                                                 (4)
                       i + 11,
                       s,
                       DISP_FGND_BLACK + DISP_BGND_LIGHT_GRAY);
        }
    }
    if (total > 1000000000L) {
        for (i = 0; i < 7; i++) {
            TaskUserData[i].TaskTotExecTime = 0L;
        }
    }
}
```

L1.20(1) 计算所有任务的运行时间(除空闲任务和统计任务)。

L1.20(2) 调用 DispTaskStat(),在合适的位置显示每个任务的累计运行时间,同时给每个任务定义的名字也显示出来。

L1.20(3)

L1.20(4) 显示以百分比表示的每个任务的相对 CPU 占用率。

如果在 C:\BC45 目录下安装了 Borland C/C++ V4.5 x 编译器,则可以试着修改 TEST.C。修改完成后,在命令行方式下进入 TEST 目录,输入 MAKETEST,即可编译、链接,得到更新过的 TEST.EXE;如果没有 Borland C/C++ V4.5 x 编译器,或者安装在不同的目录下,可以修改相应的 TEST.MAK,INCLUDES.H 及 TEST.LNK 文件。

SOURCE 目录包含 4 个文件:INCLUDES.H,OS_CFG.H,TEST.C 及 TEST.LNK。OS_CFG.H 文件用来配置 μC/OS-II 的编译选项;TEST.LNK 是 Borland 链接器 TLINK 使用的链接命令文件。

1.04 范例 4

μC/OS-II 的绝大部分代码都是用 C 语言写成的,只有部分与使用的处理器相关的代码须重新编写。为特定的处理器编写特定代码的过程称为"移植"。这本书为 2 种 80x86 的处理器提供了移植范例:Ix86L(见第 14 章)和 Ix86L-FP(见第 15 章)。Ix86L 用于所有不包含浮点处理单元 FPU 的 80x86 处理器,前面所有的例子也都是基于 Ix86L 的,并且 Ix86L 在含有浮点处理单元的 80x86 的处理器上也是可以运行的;而 Ix86L-FP 允许程序使用高端的 80x86 处理器(或兼容的处理器)的硬件浮点处理功能。范例 4 就使用了 Ix86L-FP 处理器。

在这个例子中,建立了 10 个基本相同的任务,每个任务运行 200 次/秒。每个任务都计算一个角度的正弦值和余弦值,每个任务计算的角度相差 36°(正好是 360°的 1/10)。每次执行任务时,这个角度都增加 0.01°。*

这个范例的代码在安装盘的\SOFTWARE\uCOS-II\EX4_x86L.FP\BC45 目录下,可以打开一个 MSDOS 窗口(在 Windows 2000 下称做命令行方式),输入:

```
CD \SOFTWARE\μCOS-II\Ex4_x86L.FP\BC45\TEST
```

CD 命令将当前目录切换到范例 4 的 TEST 目录。只需接着输入 TEST,就可以开始执行范例 4 的代码,于是 DOS 窗口开始运行 TEST.EXE 程序。

程序运行 2 s 后,可以看到如图 F1.6 所示的窗口。在左下角可以看到,这个程序共有 13 个任务,除 μC/OS-II 建立的空闲任务和统计任务外,范例 4 的代码建立了另外的 11 个任务。

通过前面 3 个范例的解释,读者应该可以自己来分析 TEST.C。范例 4 并没有涉及很多新概念,只有很少的一些内容需要解释。程序清单 L1.21 列出了建立 10 个相似任务的 TaskStartCreateTasks()任务的代码。

---

* 正弦和余弦涉及到大量的浮点运算,如果使用浮点运算仿真程序库,计算 2 000 次/秒是比较困难的。——译者注

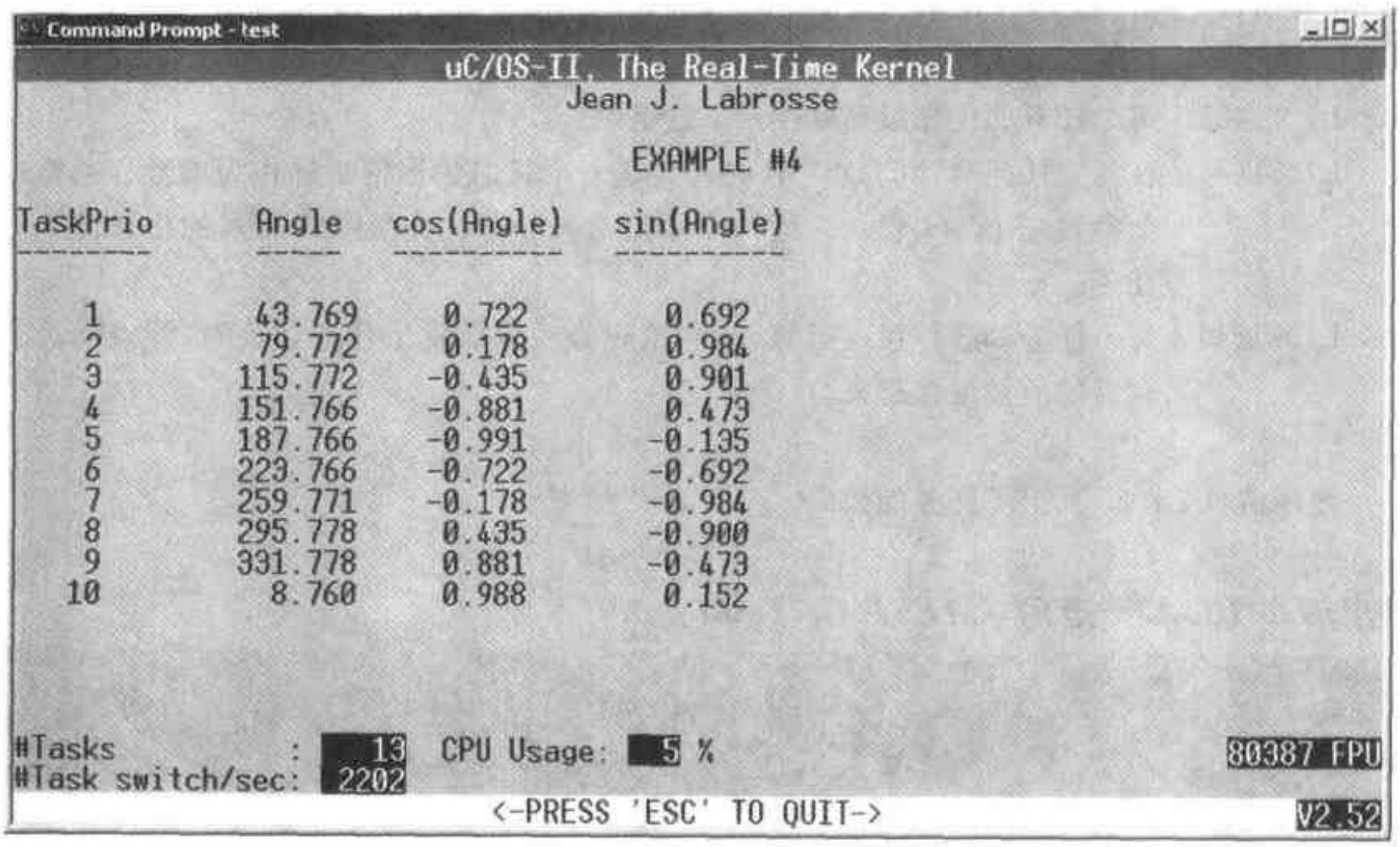


图 F1.6 范例 4 的显示

程序清单 L1.21 范例 4, TEST.C, TaskStartCreateTasks()

```
static  void  TaskStartCreateTasks (void)
{
    INT8U  i;
    INT8U  prio;


    for (i = 0; i < N_TASKS; i ++) {
        prio        = i + 1;                                    (1)
        TaskData[i] = prio;                                     (2)
        OSTaskCreateExt(Task,
                        (void *)&TaskData[i],                   (3)
                        &TaskStk[i][TASK_STK_SIZE - 1],
                        prio,
                        0,
                        &TaskStk[i][0],
                        TASK_STK_SIZE,
                        (void *)0,
                        OS_TASK_OPT_SAVE_FP);                   (4)
    }
}
```

L1.21(1) μC/OS-II 中不允许 2 个任务具有相同的优先级。TaskStartCreateTasks() 任务本身的优先级是 0,所以建立的 10 个任务优先级从 1 开始。

L1.21(2) 每个任务的优先级都放在一个数组中。

L1.21(3) 在建立任务时,μC/OS-II 允许传递一个指向数据的指针作为参数。通常这个参数用 pdata 表示。范例 4 中把指向任务优先级数组元素的指针传入给任务。

L1.21(4) 每个任务都进行浮点运算,在移植时必须保证发生任务切换时,浮点运算寄存器的值被保存起来。

程序清单 L1.22 列出了任务的代码。

程序清单 L1.22 范例 4,TEST.C, Task()

```
void  Task (void *pdata)
{
    FP32    x;
    FP32    y;
    FP32    angle;
    FP32    radians;
    char    s[81];
    INT8U   ypos;

    ypos  =  *(INT8U *)pdata + 7;
    angle = (FP32)(*(INT8U *)pdata) * (FP32)36.0;                                   (1)
    for (;;) {
        radians = (FP32)2.0 * (FP32)3.141592 * angle / (FP32)360.0;                 (2)
        x       = cos(radians);
        y       = sin(radians);
        sprintf(s, "    %2d        %8.3f    %8.3f       %8.3f",
                *(INT8U *)pdata, angle, x, y);
        PC_DispStr(0, ypos, s, DISP_FGND_BLACK + DISP_BGND_LIGHT_GRAY);
        if (angle >= (FP32)360.0) {
            angle  = (FP32)0.0;
        } else {
            angle += (FP32)0.01;
        }
        OSTimeDly(1);                                                               (3)
    }
}
```

L1.22(1) pdata 参数指向包含任务优先级的整型数组元素。每个任务计算一个不同的角度,用任务的优先级作为角度偏移量,这样可以保证每个任务的计算角度

都不相同并且相差 36°。

L1.22(2) sin()函数和 cos()函数都是默认使用弧度，对于角度要先做参数转换。

L1.22(3) 每个任务延时 1 个时钟节拍(50 ms)*，运行 200 次/秒。

除在 TaskStartCreateTasks()函数中调用了 OS_TASK_OPT_SAVE_FP，其他与不带浮点处理单元的范例程序完全相同。实际上，如果在具有浮点处理单元的处理器上运行 μC/OS-II，则完全可以在移植的代码里保证每次任务切换时都保存浮点计算单元。这就是使用 μC/OS-II 的一个好处 —— 应用程序可以适应不同的处理器。

使用这 2 种不同的 μC/OS-II 移植时，只须更改下列文件：

INCLUDES.H(在 SOURCE 目录下)：

把包含

\software\uCOS-II\ix86l\bc45\os_cpu.h

的头文件更改为另一个目录下的头文件：

\software\uCOS-II\ix86l-fp\bc45\os_cpu.h

TEST.LNK(在 SOURCE 目录下)：

链接命令文件将在不包含浮点处理单元的处理器上，使用浮点处理模拟程序库：

C:\BC45\LIB\EMU.LIB

如果在含有浮点处理单元的处理器上，则要使用硬件浮点处理的程序库：

C:\BC45\LIB\FP87.LIB

TEST.MAK(在 TEST 目录下)：

移植的目录需要更改，原为：

PORT = \SOFTWARE\uCOS-II\Ix86L\BC45

更改为：

PORT = \SOFTWARE\uCOS-II\IX86L-FP\BC45

编译选项的宏定义中包含"-f287"。如果不包含浮点处理单元的处理器，则须去掉这个选项。

* 原文如此。1 个时钟节拍应为 5 ms。——译者注

第 2 章

实时系统概念

实时系统的特点是，如果逻辑和时序出现偏差，将会引起严重后果。有 2 种类型的实时系统：软实时系统和硬实时系统。

在软实时系统中，系统的宗旨是使各个任务尽快地运行，而不要求限定某一任务在多长时间内完成；在硬实时系统中，各任务不仅须执行无误，而且要做到准时。

大多数实时系统是二者的结合。实时系统的应用涵盖广泛的领域，而多数实时系统又是嵌入式的。这意味着计算机建在系统内部，用户看不到有个计算机在系统里面。以下是一些嵌入式系统应用的例子：

过程控制——
- 食品加工；
- 化工厂。

汽车业——
- 发动机控制；
- 防抱死系统(ABS)。

办公自动化——
- 传真机；
- 复印机；

计算机外设——
- 打印机；
- 计算机终端；
- 扫描仪；
- 调制解调器。

通信类——
- 交换机；
- 路由器。

航空航天——
- 飞机管理系统；
- 喷气发动机控制。

民用消费品——
- 微波炉；
- 洗碗机；
- 洗衣机；
- 稳定温度调节器。

机器人和武器系统。

实时应用软件的设计一般比非实时应用软件设计难一些。本章讲述实时系统概念。

2.00 前/后台系统

不复杂的小系统的设计如图 F2.1 所示。这种系统可称为前/后台系统(foreground/background)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为(foreground)。后台也可以叫做任务级。前台也叫做中断级。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运行到该处理这个信息时，才能得到处理。这种系统在处理信息的及时性上，比实际上可以做到的要差。处理信息的及时性，称做任务级响应时间。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。

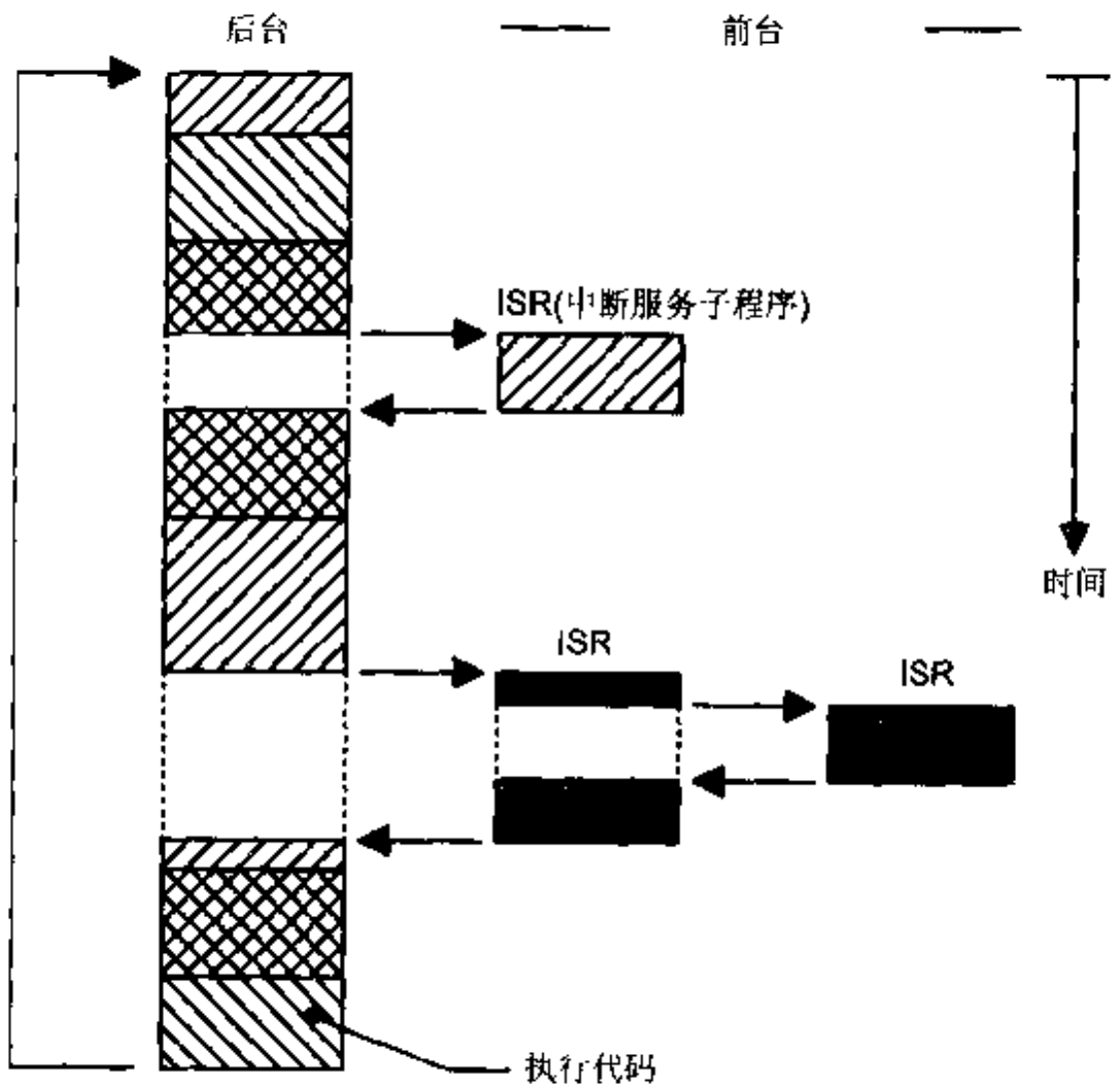


图 F2.1　前/后台系统

很多基于微处理器的产品采用前/后台系统设计，例如微波炉、电话机及玩具等。在另外一些基于微处理器的应用中，从省电的角度出发，平时微处理器处在停机状态(halt)，一切都靠中断服务来完成。

2.01　代码的临界段

代码的临界段也称为临界区，指处理时不可分割的代码。一旦这部分代码开始执行，则不允许任何中断打入。为确保临界段代码的执行不被中断，在进入临界段之前须关中断，而临界段代码执行完后，要立即开中断(参阅 2.03 节："共享资源")。

2.02 资 源

任何为任务所占用的实体都可称为资源。资源可以是输入/输出设备,例如打印机、键盘及显示器;资源也可以是一个变量、一个结构或一个数组等。

2.03 共享资源

可以被一个以上任务使用的资源叫做共享资源。为了防止数据被破坏,每个任务在与共享资源打交道时,必须独占该资源。这叫做互斥(mutual exclusion)。在2.18节:"互斥条件"中,将对技术上如何保证互斥条件做进一步讨论。

2.04 多任务

多任务运行的实现实际上是靠CPU(中央处理单元)在许多任务之间转换和调度。CPU只有一个,轮番服务于一系列任务中的某一个。多任务运行很像前/后台系统,只是后台任务有多个。多任务运行使CPU的利用率达到最高,并使应用程序模块化。在实时应用中,多任务化的最大特点是,开发人员可以将很复杂的应用程序层次化。使用多任务,应用程序将更容易设计与维护。

2.05 任 务

一个任务,也称做一个线程,是一个简单的程序,该程序可以认为CPU完全只属于该程序自己。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一部分,都被赋予一定的优先级,有自己的一套CPU寄存器和栈空间(如图F2.2所示)。

典型的是,每个任务都是一个无限的循环,都可能处在以下5种状态之一——休眠态、就绪态、运行态、挂起态(等待某一事件发生)及被中断态(参见图F2.3)。

休眠态相当于任务驻留在内存中,但并不被多任务内核所调度;就绪态意味着任务已经准备好,可以运行,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;运行态是指任务掌握了CPU的使用权,正在运行中;挂起态也可以叫做等待事件态(waiting),指任务在等待,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。图F2.3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为另一种状态。

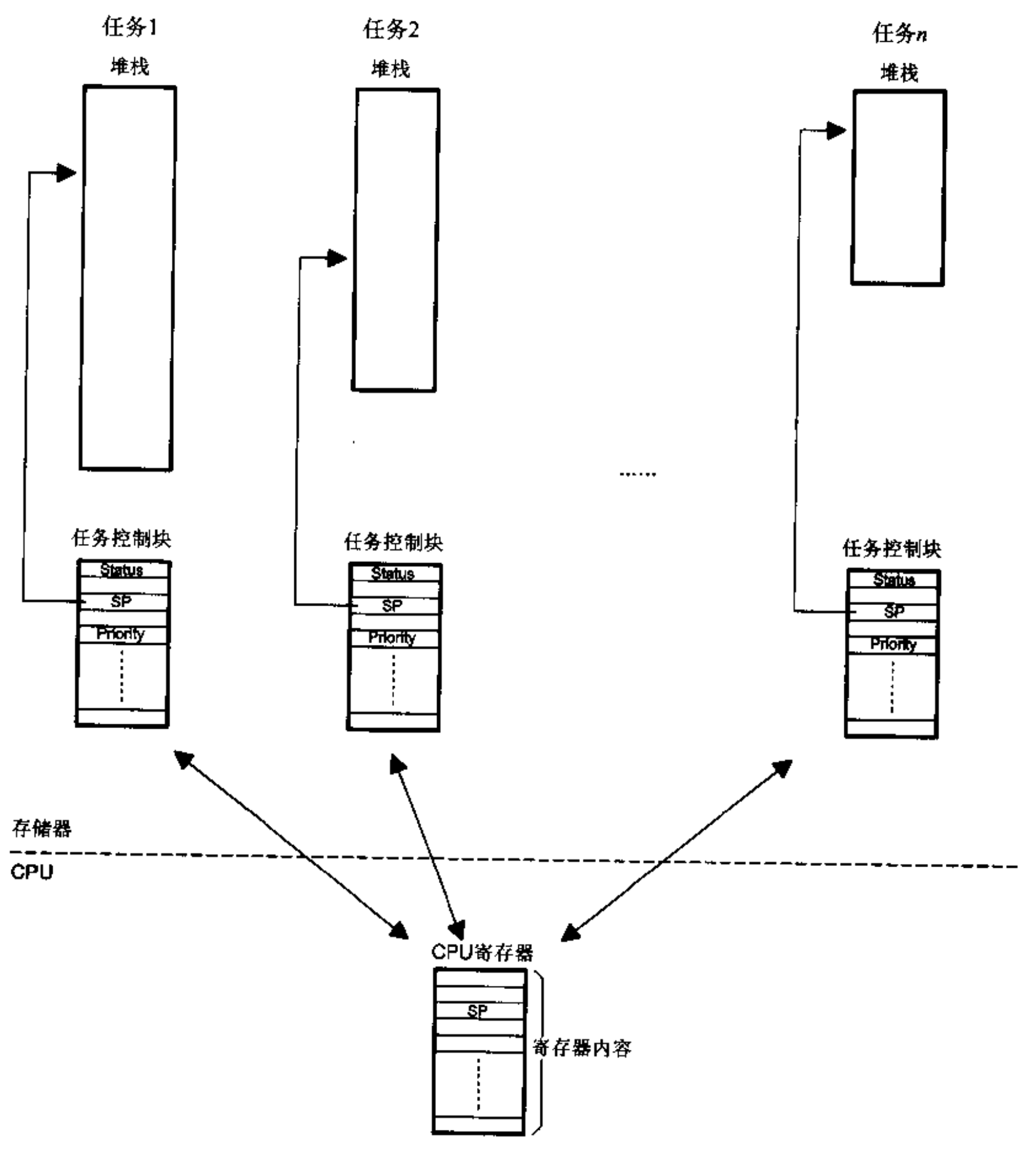

任务1
堆栈
任务2
堆栈
任务n
堆栈
.....
任务控制块
Status
SP
Priority
任务控制块
Status
SP
Priority
任务控制块
Status
SP
Priority
存储器
CPU
CPU寄存器
SP
寄存器内容

图 F2.2　多任务

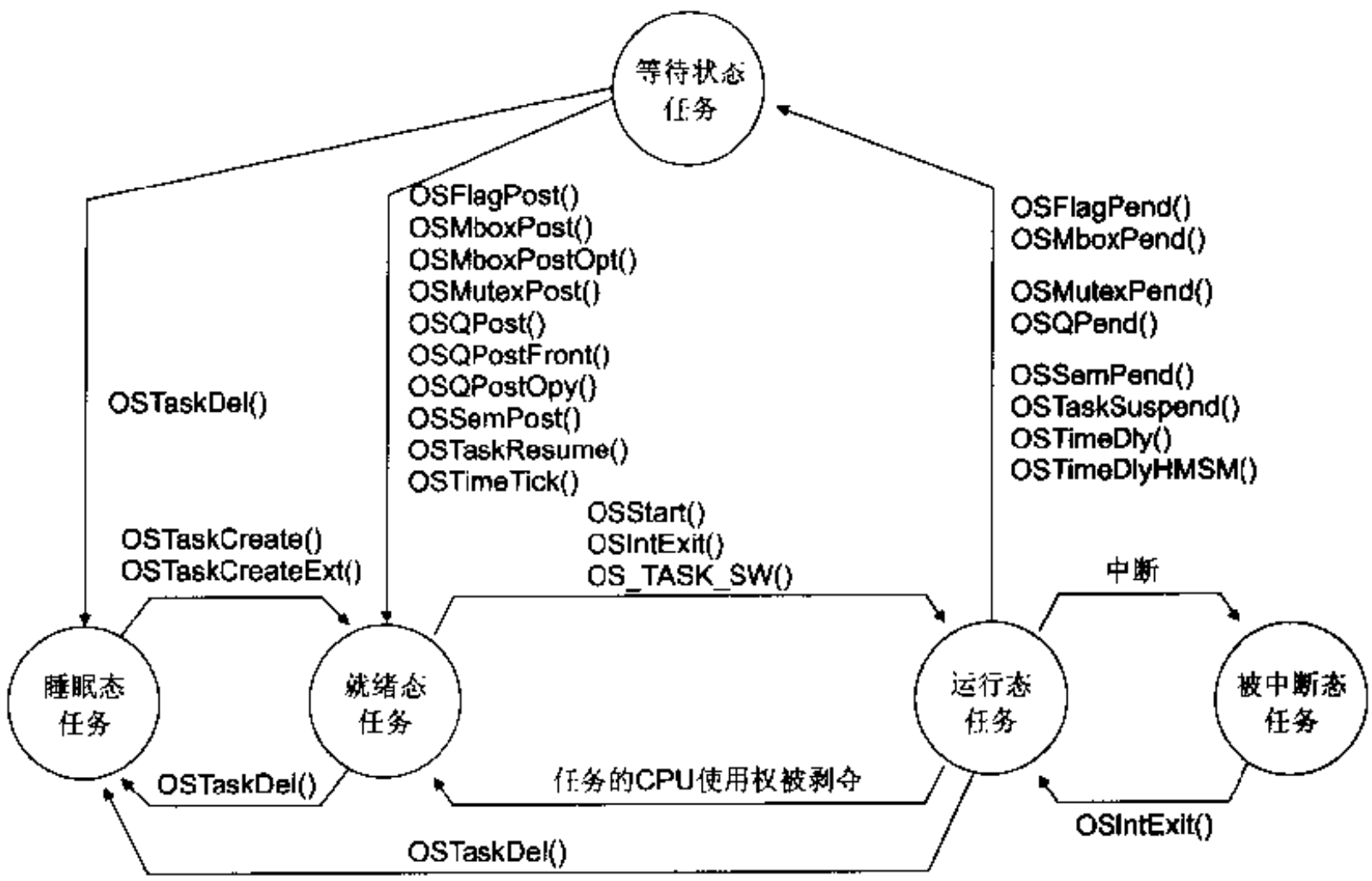


图 F2.3 任务的状态

2.06 任务切换

context switch 在有些书中被翻译成上、下文切换，实际含义是任务切换，或 CPU 寄存器内容切换。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(context)，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(task's context storage area)，也就是任务自己的栈区之中(见图 F2.2)。入栈工作完成以后，就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，这一过程叫做任务切换。

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。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以每秒能做多少次任务切换来评价。

2.07 内 核

多任务系统中，内核(kernel)负责管理各个任务，或者说为每个任务分配 CPU 时间，并且负责任务间的通信。内核提供的基本服务是任务切换。使用实时内核可以大大简化应用系统的设计，因为实时内核允许将应用分成若干个任务，由实时内核来管理它们。

内核本身也增加了应用程序的额外负荷，因为内核提供的服务需要一定的执行时间。额外负荷的量取决于用户多久调用一次这类服务。在设计得较好的应用系统中，内核占用 2%～5%的 CPU 负荷。因为内核是加在用户应用程序中的软件，会增加 ROM(代码空间)的用量，内核本身的数据结构增加了 RAM(数据空间)的用量。更主要的是，每个任务要有自己的栈空间，这部分占用内存相当多。

单片机一般不能运行实时内核，因为单片机的 RAM 很有限。通过提供必不可少的系统服务，诸如信号量管理、邮箱、消息队列及时间延时等，实时内核使得 CPU 的利用更为有效。用实时内核做过系统设计以后，将绝不会再想返回到前/后台系统。

2.08　调　度

调度(schedulers)，英文还有一词 dispatcher，也是调度的意思。这是内核的主要职责之一，就是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。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是指，CPU 总是让处于就绪态的、优先级最高的任务先运行。然而，何时高优先级任务掌握 CPU 的使用权，由使用的内核的类型确定。基于优先级的内核有 2 种类型：不可剥夺型和可剥夺型。

2.09　不可剥夺型内核

不可剥夺型内核(non-preemptive kernel)要求每个任务主动放弃 CPU 的使用权。不可剥夺型调度法也称做合作型多任务，各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。中断服务可使一个高优先级的任务由挂起态变为就绪态，但中断服务以后，使用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，一个新的高优先级的任务才能获得 CPU 的使用权。

不可剥夺型内核的一个优点是响应中断快(见 2.26 节："中断延迟")。在任务级，不可剥夺型内核允许使用不可重入函数(见 2.11 节："可重入函数")。每个任务都可以调用不可重入函数，而不必担心其他任务可能正在使用该函数，从而造成数据的破坏，因为每个任务要运行到完成时，才释放 CPU 的使用权。当然该不可重入函数本身不得有放弃 CPU 使用权的企图。

使用不可剥夺型内核时，任务级响应时间比前/后台系统快得多。此时的任务级响应时间取决于最长的任务的执行时间。

不可剥夺型内核的另一个优点是，几乎无须使用信号量保护共享数据，运行着的任务占有 CPU，而不必担心被别的任务抢占；但这也不是绝对的，在某种情况下，信号量还是用得着的。处理共享 I/O 设备时，仍须使用互斥型信号量。例如，打印机的使用，仍须满足互斥条件。

图 F2.4 表示不可剥夺型内核的运行情况。

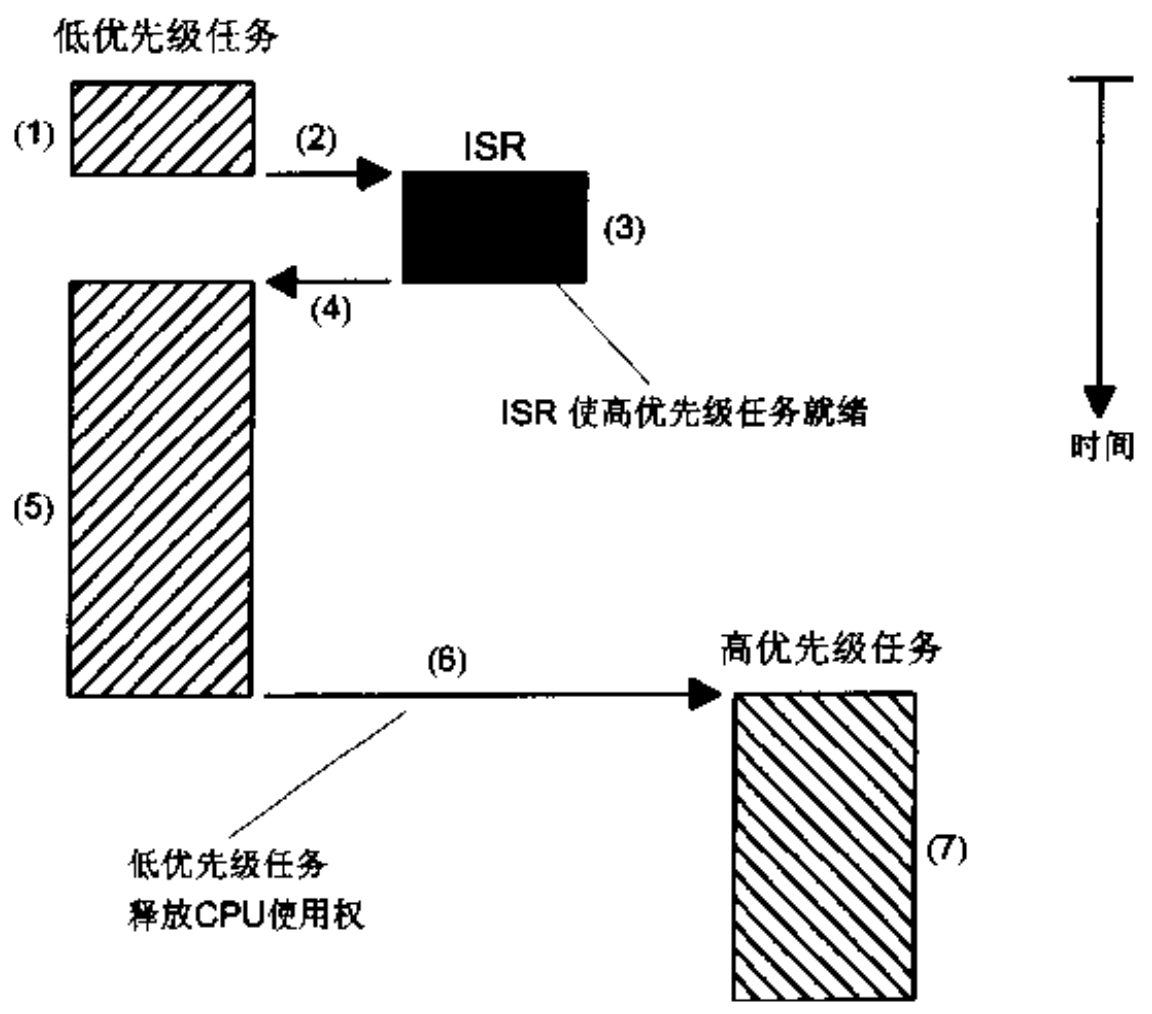


图 F2.4 不可剥夺型内核

F2.4(1) 任务在运行过程之中，此时中断来了。

F2.4(2) 若此时中断是开着，则 CPU 由中断向量进入(跳转到)中断服务子程序(ISR)。

F2.4(3) 中断服务子程序做事件处理，使一个有更高优先级的任务进入就绪态。

F2.4(4) 中断服务完成后，中断返回指令，使 CPU 回到原来被中断的任务。

F2.4(5) 接着执行该任务被中断了的代码。

F2.4(6) 直到该任务完成，调用一个内核服务函数，以释放 CPU 的使用权给其他任务。

F2.4(7) 内核看到有个优先级更高的任务已进入就绪态(内核并不需要知道该任务进入就绪态是由于 ISR 还是由于其他什么原因)，然后内核做任务切换，这个优先级更高的任务才开始处理中断服务程序标志的事件。

不可剥夺型内核最大的缺陷在于其响应时间。高优先级的任务已经进入就绪态，但还不能运行，要等，也许要等很长时间，直到当前运行着的任务释放 CPU。与前/后台系统一样，不可剥夺型内核的任务级响应时间是不确定的，无法确定何时最高优先级的任务能获得 CPU 的使用权，这完全取决于应用程序何时释放 CPU。

总之，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 的使用权。中断可以打入运行着的任务。中断服务完成后，将 CPU 使用权还给被中断了的任务。任务级的响应要大大好于前/后台系统，但仍是不可确定的。商业软件几乎没有不可剥夺型内核。

2.10 可剥夺型内核

当系统响应时间很重要时，须使用可剥夺型内核；因此，μC/OS－II 以及绝大多数商业的

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 的使用权。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时，当前任务的 CPU 使用权就被剥夺了，或者说被挂起了，更高优先级的任务立刻得到了 CPU 的使用权。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，优先级高的任务开始运行。

可剥夺型内核的执行过程如图 F2.5 所示。

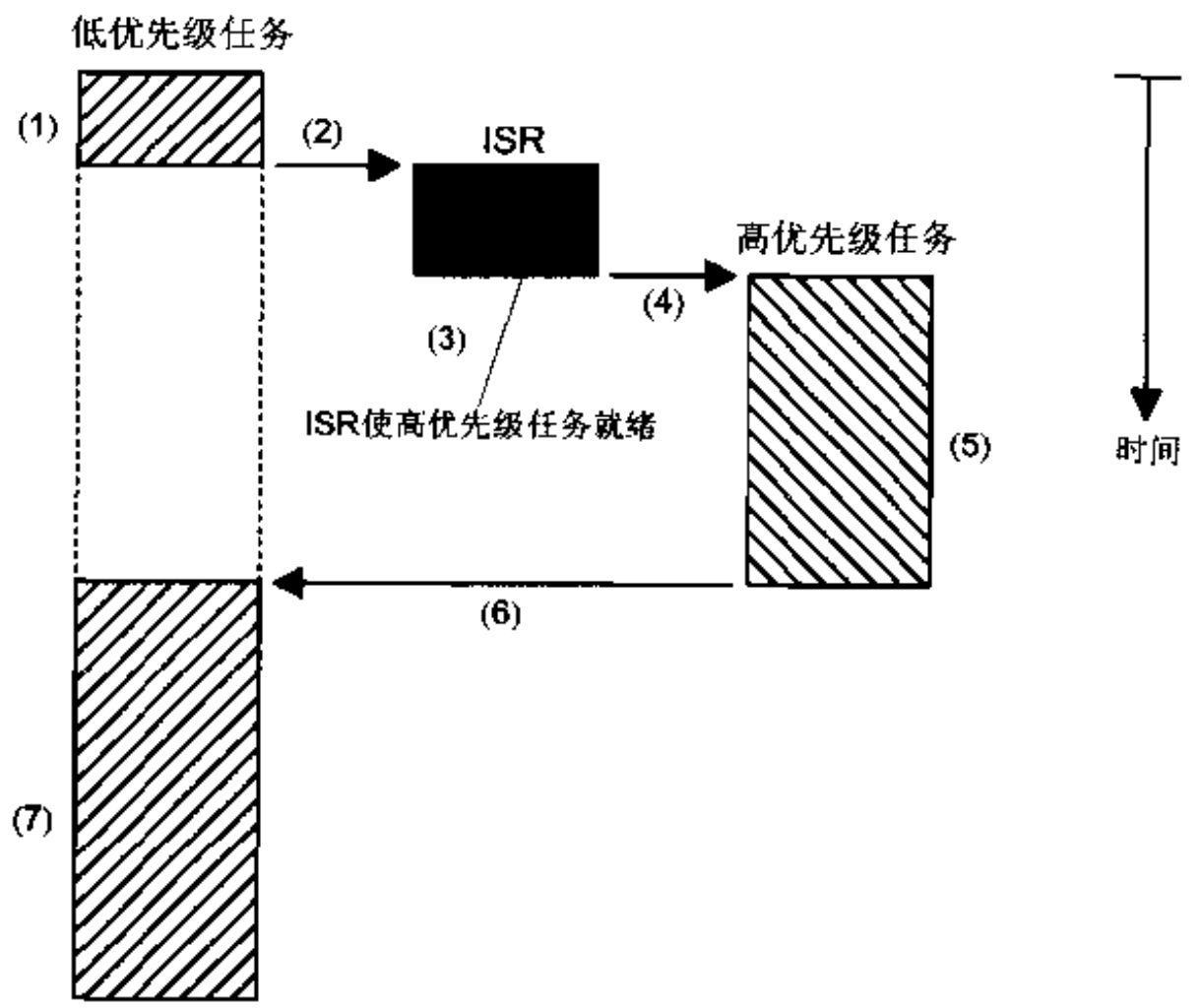


图 F2.5　可剥夺型内核

F2.5(1)　任务在运行过程中，此时中断来了。

F2.5(2)　如果此时中断是开着的，CPU 由中断向量进入(跳转到)中断服务子程序(ISR)。

F2.5(3)　中断服务子程序 ISR 做事件处理，使一个有更高优先级的任务进入就绪态。当 ISR 完成时，进入内核提供的一种服务(一个内核提供的函数被调用)。

F2.5(4)

F2.5(5)　这个函数知道一个更重要的任务已经进入就绪态，于是内核做任务切换，执行更重要的任务的代码，而不再运行原来被中断了的任务。当更重要的任务完成后，内核提供的另一个函数被调用，让该任务进入休眠态，等待事件的发生(ISR)。

F2.5(6)

F2.5(7)　然后，内核看到那个低优先级的任务要运行，于是发生另一次任务切换，被中断了的任务得以继续运行。

使用可剥夺型内核，最高优先级的任务何时可以执行，何时可以得到 CPU 的使用权，这些是可知的。可剥夺型内核使得任务级响应时间得以最优化。

使用可剥夺型内核时，应用程序不应直接使用不可重入函数。调用不可重入函数时，应满足互斥条件。这一点可以用互斥型信号量来实现，因为优先级低的和优先级高的2个任务可能会同时调用同一个函数。如果调用不可重入函数时，低优先级任务的CPU使用权被高优先级任务剥夺，不可重入函数中的数据有可能被破坏。

综上所述，可剥夺型内核总是让就绪态的高优先级的任务先运行，中断服务程序可以抢占CPU。中断服务完成时，内核让此时优先级最高的任务运行(不一定是那个被中断了的任务)。任务级系统响应时间得到了最优化，且是可知的。μC/OS-II属于可剥夺型内核。

2.11 可重入函数

可重入函数可以被一个以上的任务调用，而不必担心数据被破坏。可重入函数任何时候都可以被中断，一段时间以后又可以运行，而相应的数据不会丢失。可重入函数或者只使用局部变量，即变量保存在CPU寄存器中或堆栈中；或者使用全局变量，则要对全局变量予以保护。程序清单L2.1是一个可重入函数的例子。

程序清单 L2.1 可重入函数

```
void strcpy (char *dest, char *src)
{
    while ( *dest ++ = *src ++ ) {
        ;
    }
    *dest = NUL;
}
```

函数strcpy()做字符串复制。因为参数是存在堆栈中的，故函数strcpy()可以被多个任务调用，而不必担心各任务调用函数期间会互相破坏对方的指针。

不可重入函数的例子如程序清单L2.2所列。swap()是一个简单函数，它使函数的2个形式变量的值互换。为便于讨论，假定使用的是可剥夺型内核，中断是开着的，Temp定义为整数全局变量。

程序清单 L2.2 不可重入函数

```
int Temp;

void swap (int *x, int *y)
{
    Temp = *x;
    *x   = *y;
    *y   = Temp;
}
```

程序员打算让 swap() 函数可以为任何任务所调用。如果一个低优先级的任务正在执行 swap()函数，而此时中断发生了，于是可能发生的事情如图 F2.6 所示。

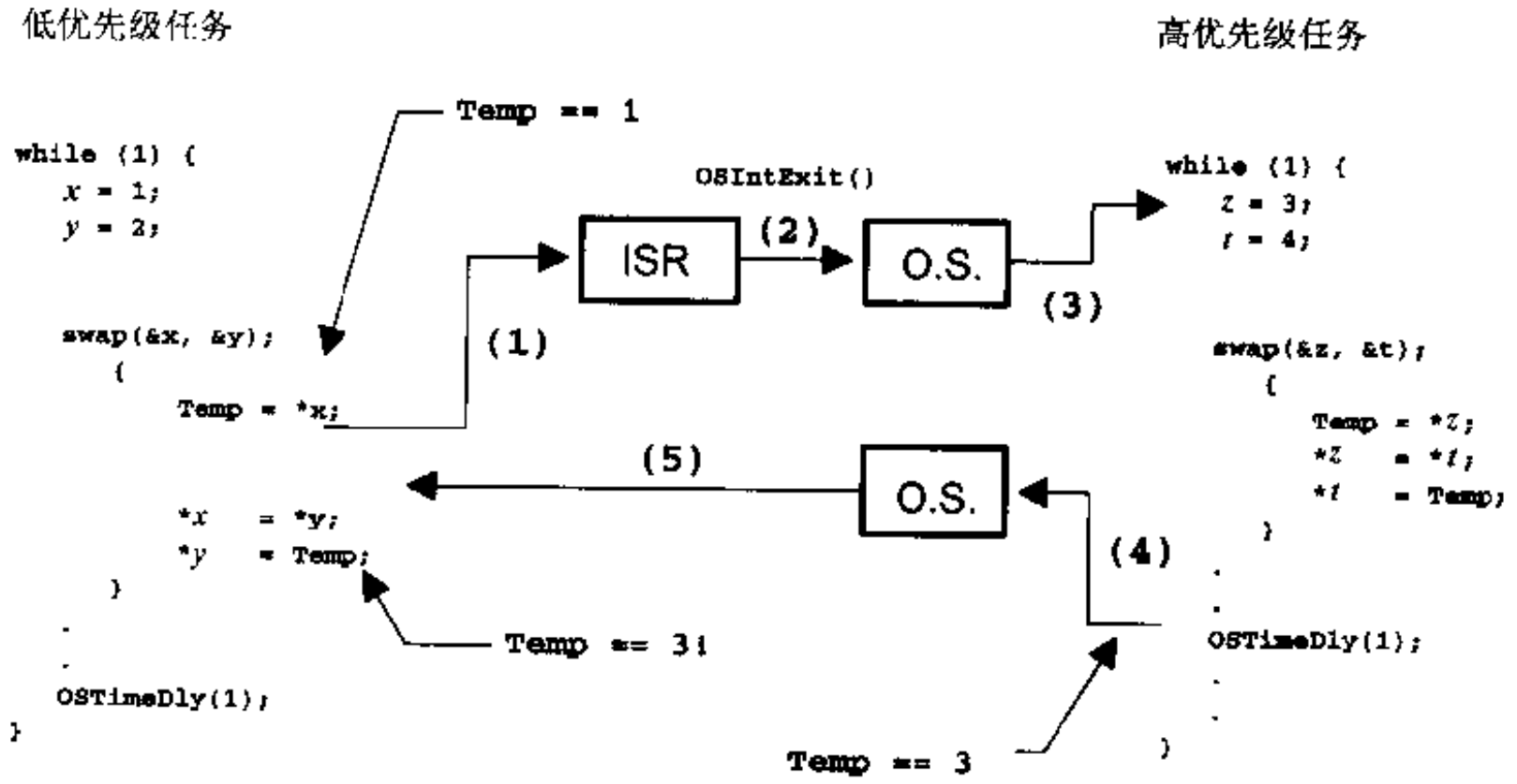


图 F2.6　不可重入函数

F2.6(1)　中断发生时 Temp 已被赋值 1。

F2.6(2)

F2.6(3)　中断服务子程序使更高优先级的任务就绪。当中断完成时，内核(假定使用的是 μC/OS－II)使高优先级的任务得以运行，高优先级的任务调用 swap()函数，使 Temp 赋值为 3。这对该任务本身来说，实现 2 个变量的交换是没有问题的。交换后 z 的值是 4，x 的值是 3。

F2.6(4)　然后高优先级的任务通过调用内核服务函数中的“延迟一个时钟节拍”，释放了 CPU 的使用权(见 2.32 节：“时钟节拍”)。

F2.6(5)　低优先级任务得以继续运行。

注意：此时 Temp 的值仍为 3！在低优先级任务接着运行时，y 的值被错误地赋为 3，而不是正确值 1。

请注意，这只是一个简单的例子，如何能使代码具有可重入性，一看就明白。

使用以下技术之一，即可使 swap()函数具有可重入性：

- 把 Temp 定义为局部变量；
- 调用 swap()函数之前关中断，调用后再开中断；
- 用信号量(见 2.18 节：“互斥条件”)。

然而有些情况下，问题并非那么易解。应用程序中不可重入函数引起的错误很可能在测试时发现不了，直到产品到了现场，问题才出现。如果在多任务问题上还是新手，使用不可重入函数时，千万要当心。

如果中断发生在 swap()函数调用之前或调用之后，2 个任务中的 x，y 值都是正确的。

2.12 时间片轮番调度法

当2个或2个以上任务有同样的优先级时，内核允许1个任务运行事先确定的一段时间，这段时间叫做时间额度(quantum)，然后切换给另一个任务。这个过程叫做时间片轮番调度。内核在满足以下条件时，把CPU使用权交给下一个就绪态的任务：

- 当前任务已空闲；
- 当前任务在时间片还没结束时已经完成了；
- 时间片结束。

目前，μC/OS-II不支持时间片轮番调度法。应用程序中各任务的优先级必须互不相同。

2.13 任务优先级

每个任务都有其优先级。任务越重要，赋予的优先级应越高。就大多数内核而言，每个任务的优先级是由用户决定的。

2.14 静态优先级

应用程序执行过程中诸任务优先级不变，称之为静态优先级。在静态优先级系统中，诸任务以及它们的时间约束在程序编译时是已知的。

2.15 动态优先级

应用程序执行过程中，任务的优先级是可变的，称之为动态优先级。实时内核应当避免出现优先级反转问题。μC/OS-II具有这一功能。

2.16 优先级反转

使用实时内核，优先级反转问题是实时系统中出现得最多的问题。图F2.7解释优先级反转是如何出现的。图中任务1优先级高于任务2，任务2优先级高于任务3。

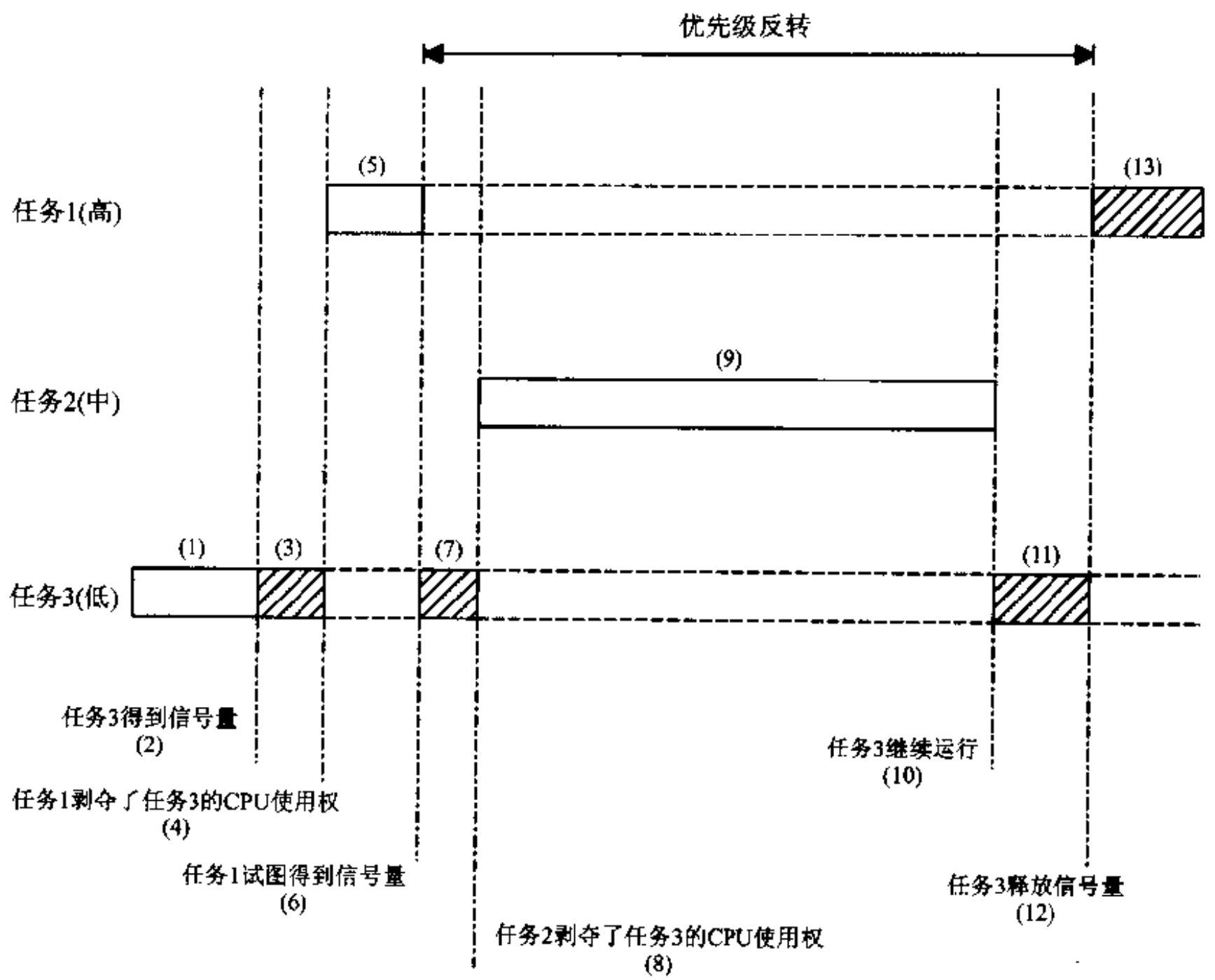


图 F2.7　优先级反转问题

F2.7(1)　任务 1 和任务 2 处于挂起状态，等待某一事件的发生，任务 3 正在运行。

F2.7(2)　此时，任务 3 要使用其共享资源。使用共享资源之前，必须首先得到该资源的信号量(见 2.18.04 节："信号量")。

F2.7(3)　任务 3 得到了该信号量，并开始使用该共享资源。

F2.7(4)　由于任务 1 优先级高，它等待的事件到来之后，剥夺了任务 3 的 CPU 使用权。

F2.7(5)

F2.7(6)　任务 1 开始运行。运行过程中，任务 1 也要使用任务 3 正在使用的资源。由于该资源的信号量还被任务 3 占用着，任务 1 只能进入挂起状态，等待任务 3 释放该信号量。

F2.7(7)

F2.7(8)　任务 3 得以继续运行。由于任务 2 的优先级高于任务 3，当任务 2 等待的事件发生后，任务 2 剥夺了任务 3 的 CPU 的使用权并开始运行。

F2.7(9)

F2.7(10)　任务 2 处理该处理的事件，直到处理完后，将 CPU 使用权还给任务 3。

F2.7(11)

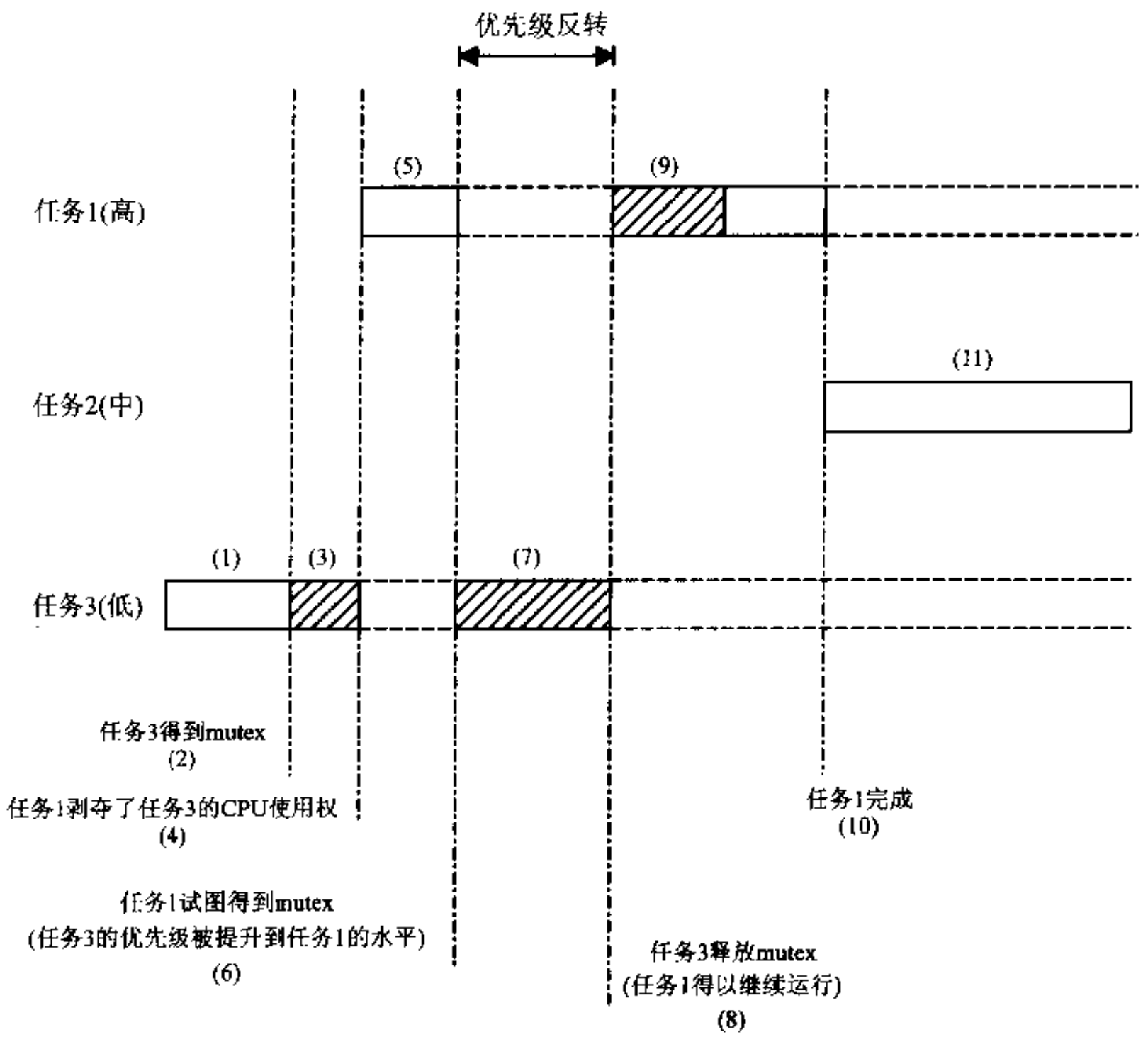
F2.7(12) 任务3继续运行，直到释放共享资源的信号量。此时，由于实时内核知道有个高优先级的任务在等待这个信号量，内核做任务切换，使任务1得以运行。

F2.7(13) 至此，任务1得到该信号量，并开始处理共享资源。

在这种情况下，任务1优先级实际上降到了任务3优先级的水平。因为任务1要等待，直等到任务3释放占用的共享资源。由于任务2剥夺了任务3的CPU使用权，使任务1的状况更加恶化，并使任务1增加了额外的延迟时间。任务1和任务2的优先级发生了反转。

纠正的方法可以是，在任务3使用共享资源时，提升任务3的优先级，任务完成后予以恢复。任务3的优先级必须升至最高，高于允许使用该资源的任何任务。多任务内核应允许动态改变任务的优先级，以避免出现优先级反转现象；然而，改变任务的优先级须花很多时间。如果任务3并没有先被任务1剥夺了CPU的使用权，又被任务2剥夺了CPU的使用权，在共享资源使用前花很多时间提升任务3的优先级，然后又在资源使用后花时间恢复任务3的优先级，则无形中浪费了很多CPU的时间。真正需要的是，为防止发生优先级反转，内核能自动变换任务的优先级，这叫做优先级继承(priority inheritance)；但μC/OS-II不支持优先级继承，一些商业内核有优先级继承功能。

图F2.8解释，如果内核支持优先级继承，在上述例子中会是怎样一个过程。



图F2.8 内核支持优先级继承

F2.8(1)

F2.8(2)　同先前例子，任务 3 正在运行，但此时，任务 3 申请互斥型信号量(mutex)，以获得共享资源的使用权。

F2.8(3)

F2.8(4)　任务 3 得到并开始使用共享资源。后来 CPU 的使用权被任务 1 剥夺。

F2.8(5)

F2.8(6)　任务 1 开始运行，任务 1 申请共享资源信号量 mutex。此时，内核知道该共享资源信号量 mutex 被任务 3 占用了，而任务 3 的优先级比任务 1 低，于是内核将任务 3 的优先级升至与任务 1 相同。

F2.8(7)　内核将任务 1 列入等待共享资源信号量 mutex 的任务列表中，然后回到任务 3 继续运行，任务 3 继续使用该共享资源。

F2.8(8)　任务 3 完成后，释放共享资源信号量 mutex。这时，内核恢复任务 3 本来的优先级，并查看等待共享资源信号量 mutex 的任务列表，看是否有任务在等这个 mutex。内核看到任务 1 在等这个 mutex，把信号量 mutex 交给任务 1。

F2.8(9)　任务 1 得以顺利运行。

F2.8(10)

F2.8(11)　任务 1 完成后，任务优先级在任务 1 与任务 3 之间的任务，例如任务 2，才能得到 CPU 的使用权，并开始运行。

注意：任务 2 在从图 F2.8(3)到图 F2.8(10)的任何一刻，都有可能进入就绪态，并不影响任务 1 和任务 3 的完成过程。在某种程度上，任务 2 和任务 3 之间也还是有不可避免的优先级反转。

2.17　任务优先级分配

给任务定(分配)优先级不是件简单的事，因为实时系统相当复杂。许多系统中，并非所有的任务都至关重要。不重要的任务自然优先级可以低一些。实时系统大多综合了软实时和硬实时 2 种需求。软实时系统只是要求任务执行得尽量快，并不要求在某一特定时间内完成；硬实时系统中，任务不但要执行无误，还要准时完成。

一项有意思的技术，称之为单调执行率调度法 RMS(Rate Monotonic Scheduling)，用于分配任务优先级。这种方法基于任务执行的次数(或者称为任务的执行率)，执行最频繁的任务优先级最高，见图 F2.9。

RMS 做了一系列假设：

- 所有任务都是周期性的；
- 任务间不需要同步，没有共享资源，没有任务间数据交换等问题；
- CPU 必须总是执行优先级最高且处于就绪态的任务，换句话说，须使用可剥夺型调度法。

给出一系列 n 值，表示系统中的不同任务数。要使所有的任务满足硬实时条件，必须使不等式(2.1)成立，这就是 RMS 定理：

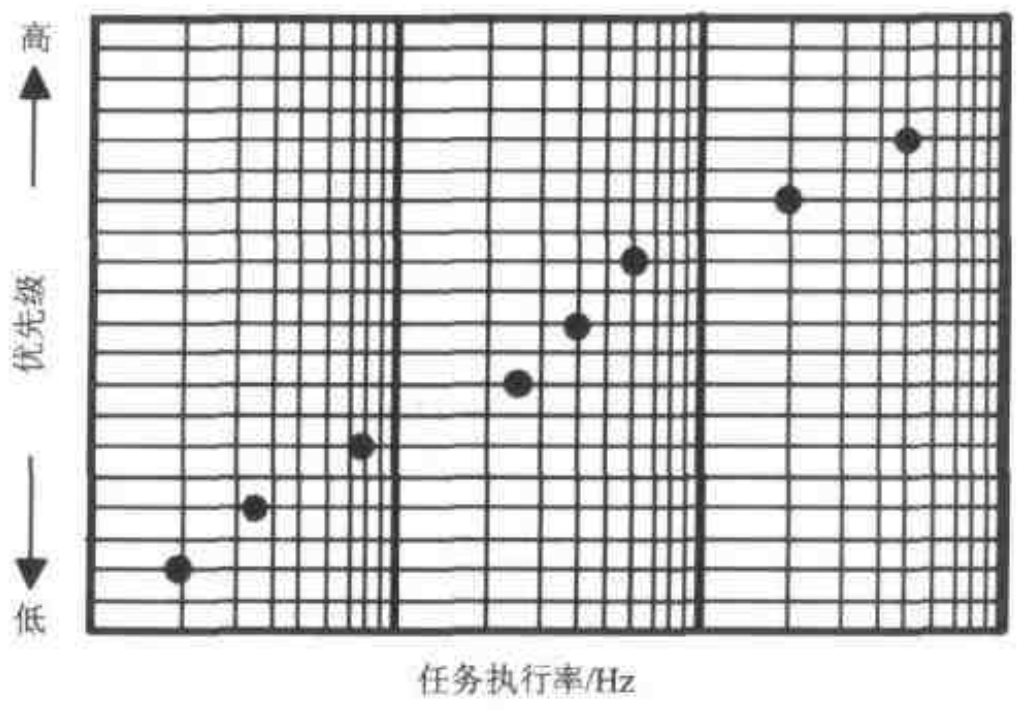


图 F2.9 基于任务执行率的优先级分配法

$$\sum_i \frac{E_i}{T_i} \leqslant n(2^{1/n}-1) \tag{2.1}$$

式(2.1)中 E_i 是任务 i 的最长执行时间，T_i 是任务 i 的执行周期。换句话说，E_i/T_i 是任务 i 所需的 CPU 时间。表 T2.1 给出 $n(2^{1/n}-1)$ 的值，n 是系统中的任务数。无穷多任务的上限是 ln 2 即 0.693。这就意味着，基于 RMS，如果要求任务都满足硬实时条件，那么所有有时间条件要求的任务 i 的总的 CPU 利用时间应小于 70%！

注意：这是指有时间条件要求的任务。系统中当然还可以有对时间没有什么要求的任务，使得 CPU 的利用率达到 100%。使 CPU 的利用率达到 100%并不好，因为那样的话程序就没有了修改的余地，也无法增加新功能了。作为系统设计的一条原则，CPU 的利用率应小于 60%～70%。

RMS 认为，最高执行率的任务具有最高的优先级；但在某些情况下，最高执行率的任务并非是最重要的任务。如果实际应用都以 RMS 为基准，也就没有优先级分配的问题可讨论了。然而讨论优先级分配的问题，RMS 无疑是一个有意思的起点。

表 T2.1 基于任务的 CPU 最高允许使用率

任务数	$n(2^{1/n}-1)$
1	1.000
2	0.828
3	0.779
4	0.756
5	0.743
.	.
.	.
.	.
∞	0.693

2.18　互斥条件

实现任务间通信最简便的办法是使用共享数据结构。特别是当所有的任务都在一个单一地址空间下，这种处理特别简便。如果能使用变量（如全局变量、指针、缓冲区、链表以及循环缓冲区等），使用共享数据结构通信就更为容易。虽然共享数据区法简化了任务间的信息交换，但是必须保证每个任务在处理共享数据时的排它性，以避免竞争和数据的破坏。与共享资源打交道时，使之满足互斥条件最一般的方法有：

- 关中断；
- 使用测试并置位指令；
- 禁止做任务切换；
- 利用信号量。

2.18.01　关中断和开中断

处理共享数据时保证互斥，最简便快捷的方法是关中断和开中断。如示意性代码程序清单 L2.3 所列。

程序清单 L2.3　关中断和开中断

```
Disable interrupts;                                          /*关中断*/
Access the resource (read/write from/to variables);          /*读/写变量*/
Reenable interrupts;                                         /*重新允许中断*/
```

μC/OS－II 在处理内部变量和数据结构时，使用的就是这种方法，即使不是全部，也是绝大部分过程都采用了这种方法。实际上 μC/OS－II 提供 2 个宏调用，允许在应用程序的 C 代码中关中断，然后再开中断：OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()（参见3.00节："临界段，OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()"）这 2 个宏调用的使用方法见程序清单 L2.4。

程序清单 L2.4　利用 μC/OS－II 宏调用关中断和开中断

```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
    .     /*在这里处理共享数据*/
    .
    OS_EXIT_CRITICAL();
}
```

必须十分小心,关中断的时间不能太长。因为它影响整个系统的中断响应时间,即中断延迟时间。当改变或复制某几个变量的值时,应考虑使用这种方法。这也是在中断服务子程序中处理共享变量或共享数据结构的惟一方法。在任何情况下,关中断的时间都要尽量短。

使用某种实时内核,一般地说,关中断的时间最长不超过内核本身的关中断时间,这样就不会影响系统中断延迟。当然须知道内核里中断关了多久。凡好的实时内核,厂商都会提供这方面的数据。总而言之,要想出售实时内核,时间特性最重要。

2.18.02 测试并置位操作

如果不使用实时内核,那么当2个任务共享一个资源时,须约定好,先测试一全局变量,如果该变量为0,允许该任务与共享资源打交道。为防止另一任务也要使用该资源,第1个得到资源的任务只需简单地将全程变量置为1。这通常称做测试并置位TAS(Test-and-Set)操作。TAS操作可能是微处理器的一条不会被中断的指令,否则应在程序中关中断做TAS操作再开中断,如程序清单L2.5所列。

程序清单L2.5 利用测试并置位处理共享资源

```
Disable interrupts;                          关中断
if ('Access Variable' is 0) {                如果资源未占用,标志为0
    Set variable to 1;                       置资源已占用,标志为1
    Reenable interrupts;                     重开中断
    Access the resource;                     处理该资源
    Disable interrupts;                      关中断
    Set the 'Access Variable' back to 0;     恢复资源为未占用,标志为0
    Reenable interrupts;                     重新开中断
} else {                                     否则
    Reenable interrupts;                     开中断
    /* 资源不可使用,以后再试 */
}
```

有些微处理器具有硬件的TAS指令(如Motorola 68000系列)。

2.18.03 禁止,然后允许任务切换

如果任务不与中断服务子程序共享变量或数据结构,可以使用"禁止,然后允许任务切换"操作(参见3.07节:"给调度器上锁和开锁")。如程序清单L2.6所列,以μC/OS-II的使用为例,2个或2个以上的任务可以共享数据而不发生竞争。注意,此时虽然任务切换是禁止了,但中断还是开着的。如果这时中断来了,中断服务子程序会在这一临界区内立即执行。中断服务子程序结束时,尽管有优先级高的任务已经进入就绪态,内核还是返回到原来被中断了的任务。由于ISR返回到被中断了的任务,内核的行为与不可剥夺型内核十分相像(至少在任务切换被上锁期间如此)。直到执行完给任务切换开锁函数OSSchedUnlock(),内核再看是否有优先级更高的任务被中断服务子程序激活而进入就绪态。如果有,则做任务切换。虽然这种方法是可行的,但应该尽量避免"禁止任务切换"等操作,因为内核最主要的功能就是做

任务的调度与协调。“禁止任务切换”显然与内核的初衷相违。应该使用下述方法。

程序清单 L2.6　用给任务切换上锁、然后开锁的方法实现数据共享

```
void Function (void)
{
    OSSchedLock();
    .
    .     /* 在这里处理共享数据(中断是开着的) */
    .
    OSSchedUnlock();
}
```

2.18.04　信号量

信号量(semaphores)* 是 20 世纪 60 年代中期 Edgser Dijkstra 发明的。信号量实际上是一种约定机制，在多任务内核中普遍使用。信号量用于：

- 控制共享资源的使用权(满足互斥条件)；
- 标志某事件的发生；
- 使 2 个任务的行为同步。

信号像一把钥匙，任务要运行下去，需先拿到这把钥匙。如果信号已被别的任务占用，那么该任务只得被挂起，直到信号被当前使用者释放。换句话说，申请信号的任务是在说：“把钥匙给我。如果谁正在用着，我只好等！”信号是只有 2 个值的变量，信号量是计数式的。只取 2 个值的信号，是只有 2 个值 0 和 1 的量，因此也称之为信号量。计数式信号量的值可以是 0～255，或 0～65 535，或 0～4 294 967 295，取决于信号量规约机制使用的是 8 位、16 位还是 32 位。到底是几位，实际上取决于所用内核的类型。根据信号量的值，内核跟踪那些等待信号量的任务。

一般地说，对信号量只能实施 3 种操作：初始化(initialize)，也可称做建立(create)；等信号(wait)，也可称做挂起(pend)；给信号(signal)或发信号(post)。信号量初始化时，要给信号量赋初值，等待信号量的任务表(waiting list)应清为空。

想得到信号量的任务，须执行“等待(wait)”操作。如果该信号量有效(信号量值大于 0)，则信号量值减 1，任务得以继续运行。如果信号量的值为 0，等待信号量的任务就被列入等待信号量任务表。多数内核允许定义等待超时。如果等待时间超过了某一设定值，该信号量还是无效，则等待信号量的任务进入就绪态，准备运行，并返回出错代码(指出发生了等待超时错误)。

任务以“发信号(signal)”操作释放信号量。如果没有任务等待信号量，那么信号量的值仅仅是简单地加 1；如果有任务等待该信号量，那么就会有一个任务进入就绪态，信号量的值也就不加 1。于是钥匙给了等待信号量的诸任务中的一个任务。至于给了哪个任务，要看内核

---

* 信号与信号量在英文中都叫做 semaphore，并不加以区分；而说它有 2 种类型：二进制型(binary)和计数器型(counting)。本书中的二进制型信号量，实际上是只取 2 个值 0 和 1 的信号量，实际上这个信号量只有 1 位，这种信号量翻译为信号更为贴切；而真正的二进制信号量通常指若干位的组合。本书中解释为事件标志的置位与清除。——译者注

是如何调度的。

收到信号量的任务可能是以下两者之一。

- 等待信号量任务中优先级最高的；
- 最早开始等待信号量的任务，即按先进先出 FIFO(First In First Out)的原则。

有些内核具有选择项，允许在信号量初始化时，选定上述 2 种方法中的一种；但 μC/OS-II只支持优先级法。

如果进入就绪态的任务比当前运行的任务优先级高(假设当前任务释放的信号量激活了比自己优先级高的任务)，则内核做任务切换(假设使用的是可剥夺型内核)，高优先级的任务开始运行，当前任务被挂起，直到又变成就绪态中优先级最高的任务。

程序清单 L2.7 示意在 μC/OS-II 中如何用信号量处理共享数据。要与同一共享数据打交道的任务，调用等待信号量函数 OSSemPend()。处理完共享数据后，再调用释放信号量函数 OSSemPost()。这 2 个函数将在以后的章节中描述。要注意的是，在使用信号量之前，一定要对该信号量做初始化。作为互斥条件，信号量初始化为 1。使用信号量处理共享数据不增加中断延迟时间。如果中断服务程序或当前任务激活了一个高优先级的任务，则高优先级的任务立即开始执行。

程序清单 L2.7　通过获得信号量处理共享数据

```
OS_EVENT *SharedDataSem;
void Function (void)
{
    INT8U err;
    OSSemPend(SharedDataSem, 0, &err);
    .
    .     /* 共享数据的处理在此进行(中断是开着的) */
    .
    OSSemPost(SharedDataSem);
}
```

当诸任务共享输入/输出设备时，信号量特别有用。可以想象，如果允许 2 个任务同时给打印机送数据会出现什么现象。打印机会打出相互交叉的 2 个任务的数据。例如任务 1 要打印"I am Task1!"，而任务 2 要打印"I am Task2!"，打印出来的结果可能是："I Ia amm T Tasask k1! 2!"。

在这种情况下，应使用信号量，并给信号量赋初值 1(用二进制信号量)。规则很简单，要想使用打印机的任务，先要得到该资源的信号量。

图 F2.10 表示 2 个任务竞争得到排它性打印机的使用权。图中信号量用一把钥匙表示，想使用打印机，先要得到这把钥匙。

上例中，每个任务都知道有个信号表示资源可不可以使用。要想使用该资源，需先得到这个信号。然而有些情况下，最好把信号量藏起来，各任务在与某一资源打交道时，并不知道实际上是在申请得到一个信号量。例如，多任务共享一个 RS-232C 外设接口时，各任务要送命令给接口另一端的设备，并接收该设备的回应。如图 F2.11 所示。

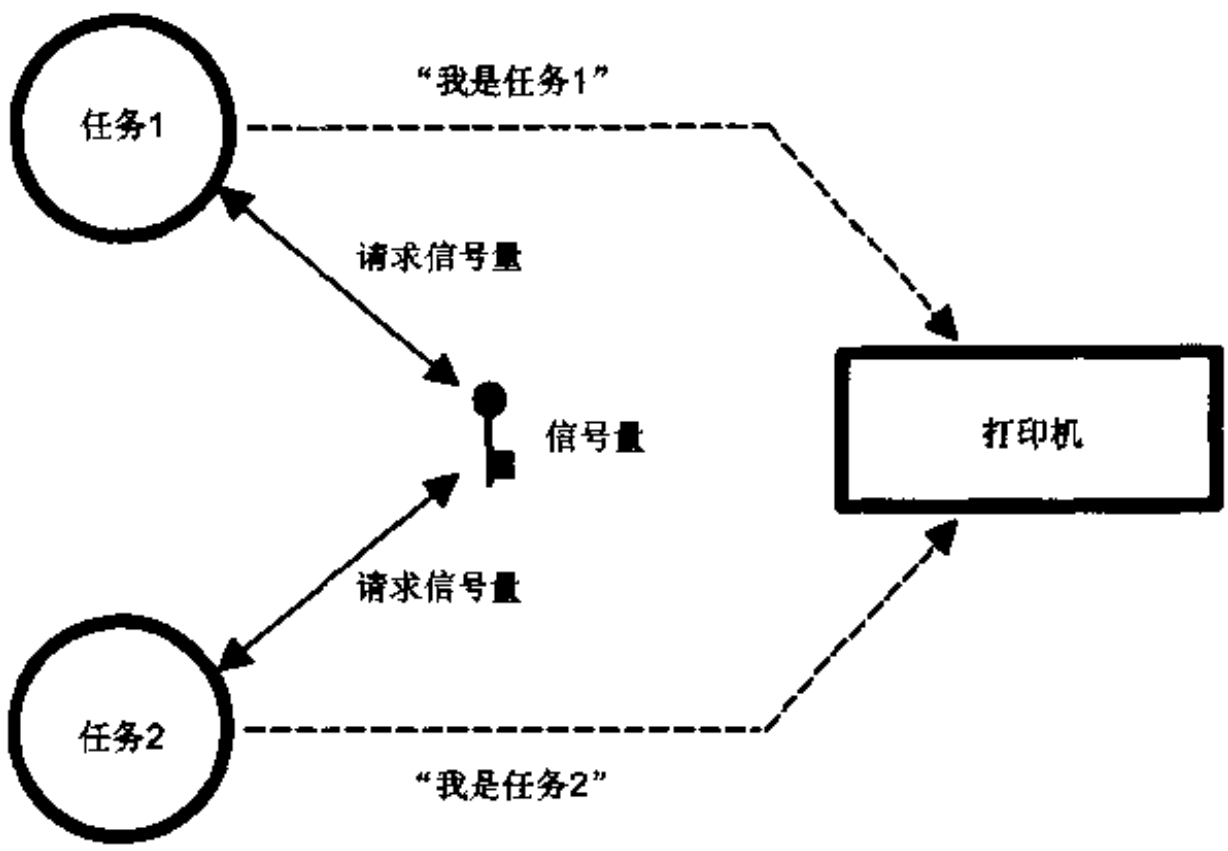


图 F2.10　用获取信号量的方法得到打印机的使用权

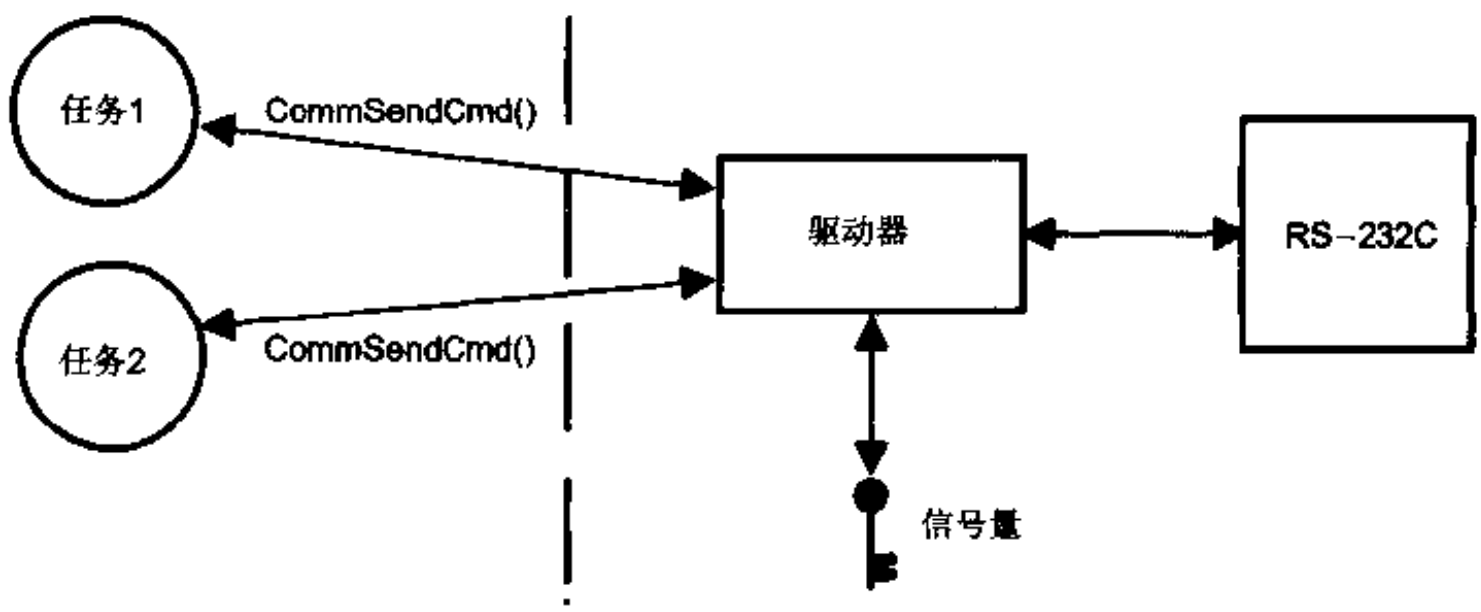


图 F2.11　在任务级看不到隐含的信号量

调用向串行口发送命令的函数 CommSendCmd()。该函数有 3 个形式参数：cmd 指向送出的 ASCII 码字符串命令；response 指向外设回应的字符串；timeout 指设定的时间间隔，如果超过这段时间外设还不响应，则返回超时错误信息。函数的示意代码如程序清单 L2.8 所列。

要向外设发送命令的任务，须调用上述函数。设信号量初值为 1，表示允许使用。初始化是在通信口驱动程序的初始化部分完成的。第 1 个调用 CommSendCmd()函数的任务申请并得到了信号量，开始向外设发送命令并等待响应。另一个任务也要发送命令，此时外设正“忙”，则第 2 个任务被挂起，直到该信号量重新被释放。第 2 个任务看起来像调用了一个普通函数一样，只不过这个函数在没有完成其相应功能时不返回。当第 1 个任务释放了信号量，第 2 个任务得到了该信号量时，第 2 个任务才能使用 RS-232 接口。

程序清单 L2.8　隐含的信号量

```
INT8U CommSendCmd(char *cmd, char *response, INT16U timeout)
{
    Acquire port's semaphore;
    Send command to device;
    Wait for response (with timeout);
    if (timed out) {
        Release semaphore;
        return (error code);
    } else {
        Release semaphore;
        return (no error);
    }
}
```

计数式信号量用于某资源可同时为几个任务所用。例如，用信号量管理缓冲区阵列(buffer pool)，如图 F2.12 所示。缓冲区阵列中共有 10 个缓冲区，任务通过调用申请缓冲区函数 BufReq()，向缓冲区管理方申请得到缓冲区使用权。当缓冲区使用权不再需要时，通过调用释放缓冲区函数 BufRel()，将缓冲区还给管理方。函数示意代码如程序清单 L2.9 所列。

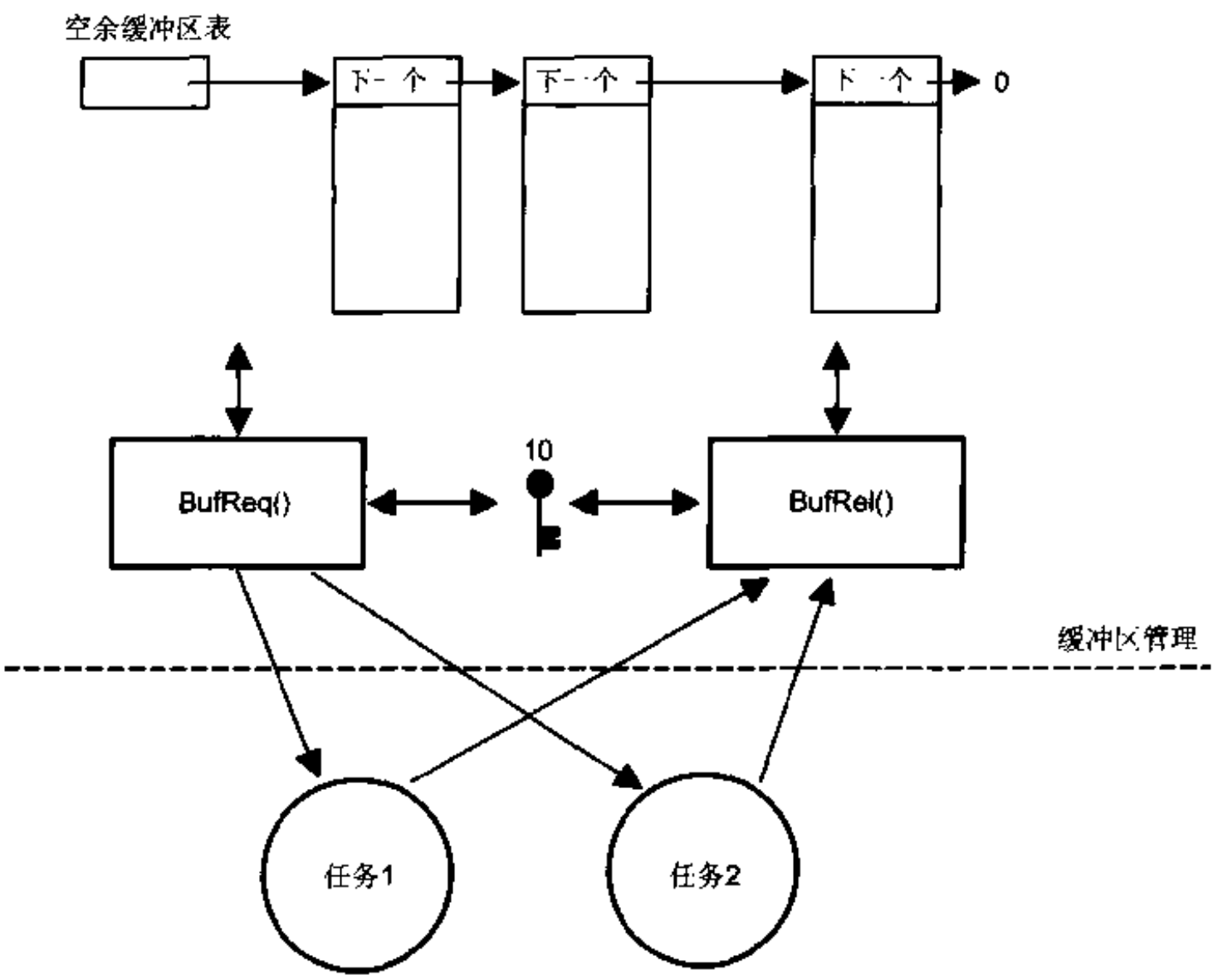


图 F2.12　计数式信号量的用法

程序清单 L2.9　用信号量管理缓冲区

```
BUF *BufReq(void)
{
    BUF *ptr;

    Acquire a semaphore;
    Disable interrupts;
    ptr          = BufFreeList;
    BufFreeList = ptr->BufNext;
    Enable interrupts;
    return (ptr);
}

void BufRel (BUF *ptr)
{
    Disable interrupts;
    ptr->BufNext = BufFreeList;
    BufFreeList  = ptr;
    Enable interrupts;
    Release semaphore;
}
```

缓冲区阵列管理方满足前 10 个申请缓冲区的任务，就好像有 10 把钥匙可以发给诸任务。当所有的钥匙都用完后，申请缓冲区的任务被挂起，直到信号量重新变为有效。缓冲区管理程序在处理链表指针时，为满足互斥条件，中断是关掉的(这一操作非常快)。任务使用完某一缓冲区，通过调用缓冲区释放函数 BufRel()，将缓冲区还给系统。系统先将该缓冲区指针插入空闲缓冲区链表中(linked list)，然后再给信号量加 1，或释放该信号量。这一过程隐含在缓冲区管理程序 BufReq()和 BufRel()之中，调用这 2 个函数的任务不用管函数内部的详细过程。

信号量常被用过了头。处理简单的共享变量也使用信号量，则是多余的。请求和释放信号量的过程是要花费相当的时间的。有时这种额外的负荷是不必要的，可能只须关中断、开中断来处理简单的共享变量，以提高效率(参见 2.18.01 小节："关中断和开中断")。假如 2 个任务共享一个 32 位的整数变量，那么一个任务给这个变量加 1 的同时，另一个任务给这个变量清 0。不管哪种操作，如果考虑到处理器执行操作的时间的长短，就不会使用信号量来满足互斥条件了。每个任务只需在操作执行前关中断，而操作执行后再开中断就可以了。然而，如果这个变量是浮点数，而相应的微处理器又没有硬件的浮点数处理器，那么浮点运算的时间会相当长，关中断时间长了会影响中断延迟时间。这种情况下就有必要使用信号量了。

2.19　死　锁

死锁也称做抱死(deadlock 或 deadly embrace)，指 2 个任务无限期地互相等待对方控制

着的资源。设任务 T1 正独享资源 R1，任务 T2 正独享资源 R2，而此时 T1 又要独享 R2，T2 也要独享 R1，于是哪个任务都无法继续执行了，发生了死锁。最简单的防止发生死锁的方法是，让每个任务都：

- 先得到全部需要的资源，再做下一步的工作；
- 用同样的顺序申请多个资源；
- 释放资源时，使用相反的顺序。

内核大多允许用户在申请信号量时定义等待超时，以此化解死锁。当等待时间超过了某一确定值，而信号量还是无效状态时，就会返回某种形式的出现超时错误的代码。这个出错代码告知该任务，不是得到了资源使用权，而是系统错误。死锁一般发生在大型多任务系统中，在嵌入式系统中不易出现（至少情况要好一些）。

2.20 同 步

可以利用信号量使某任务与中断服务同步（或者与另一个任务同步，这2个任务间没有数据交换）。如图 F2.13 所示。注意，图中用一面旗帜，或称做一个标志表示信号量。这个标志表示某一事件的发生（不再是一把用来保证互斥条件的钥匙），用来实现同步机制的信号量初始化成0。信号量用于这种类型的同步，称做单向同步（unilateral rendezvous）。一个任务做 I/O 操作，然后等信号回应。当 I/O 操作完成时，中断服务程序（或另外一个任务）发出信号，该任务得到信号后继续往下执行。

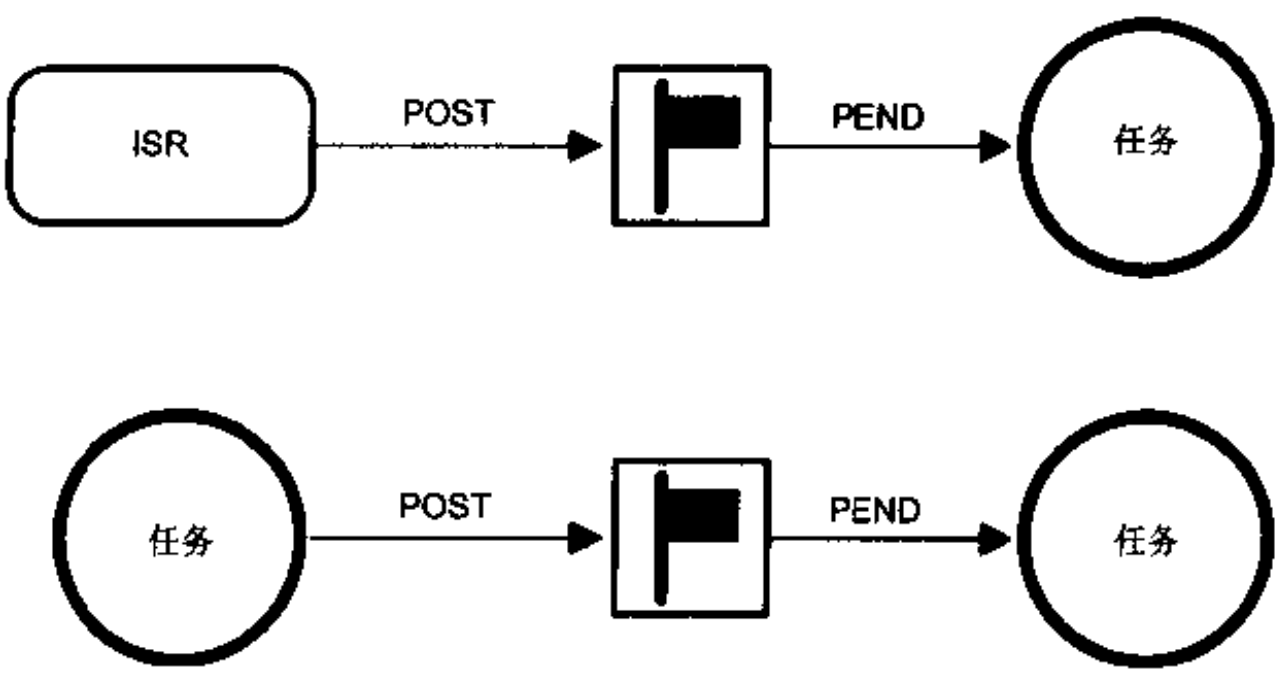


图 F2.13 用信号量使任务与中断服务同步

如果内核支持计数式信号量，那么信号量的值表示尚未得到处理的事件数。请注意，可能会有一个以上的任务在等待同一事件的发生，那么这种情况下内核会根据以下原则之一发信号给相应的任务：

- 发信号给等待事件发生的任务中优先级最高的任务；
- 发信号给最先开始等待事件发生的任务。

根据不同的应用，发信号以标志事件发生的中断服务或任务也可以是多个。

2 个任务可以用 2 个信号量同步它们的行为，如图 F2.14 所示。这叫做双向同步(bilateral rendezvous)。双向同步与单向同步类似，只是 2 个任务要相互同步。双向同步不可能在任务与 ISR 之间实施，因为 ISR 不可能等待一个信号量。

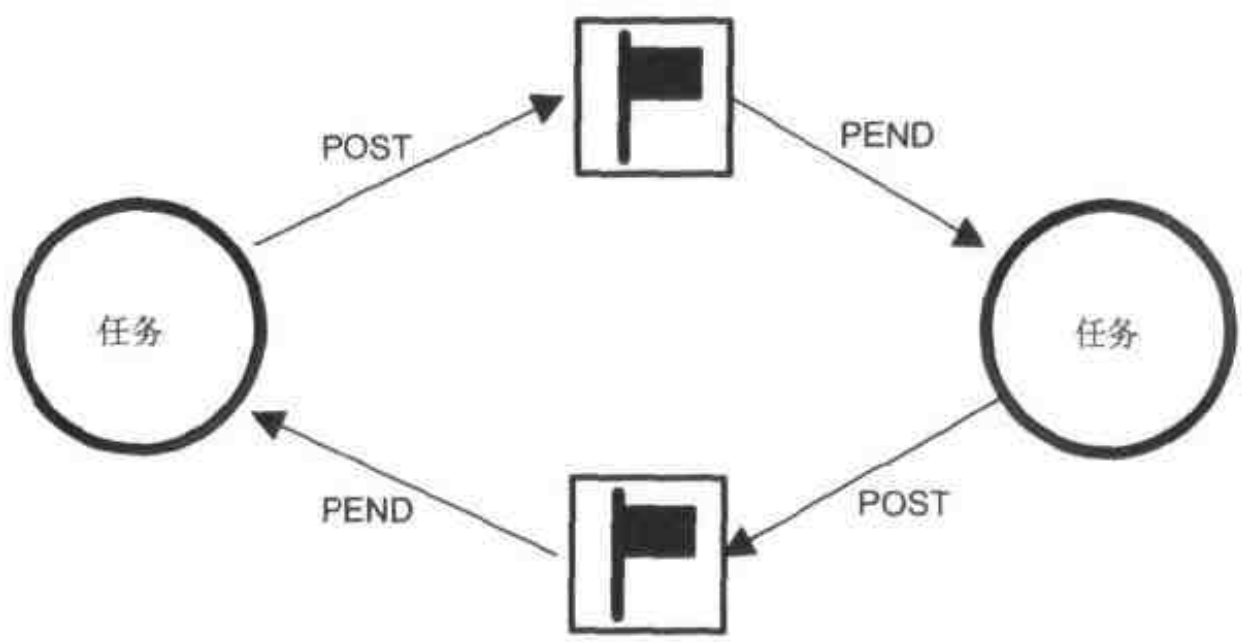


图 F2.14　2 个任务用信号量同步彼此的行为

程序清单 L2.10 所列为正在执行的 2 个任务的情况。

程序清单 L2.10　双向同步

```
Task1()
{
    for (;;) {
        Perform operation;
        Signal task #2;                                    (1)
        Wait for signal from task #2;                      (2)
        Continue operation;
    }
}

Task2()
{
    for (;;) {
        Perform operation;
        Signal task #1;                                    (3)
        Wait for signal from task #1;                      (4)
        Continue operation;
    }
}
```

L2.10(1)

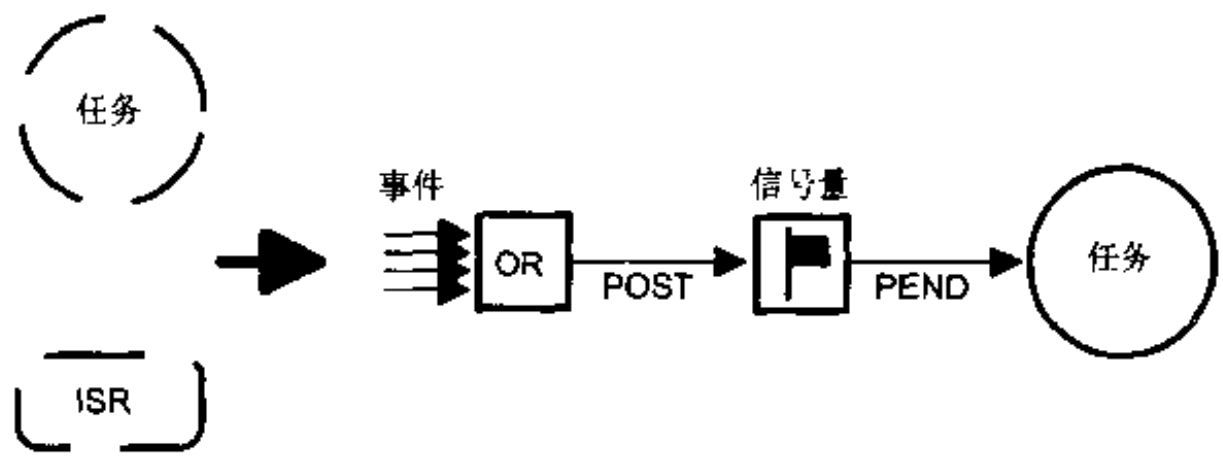
L2.10(2)　运行到某一处的第 1 个任务发信号给第 2 个任务，然后等待信号返回。

L2.10(3)

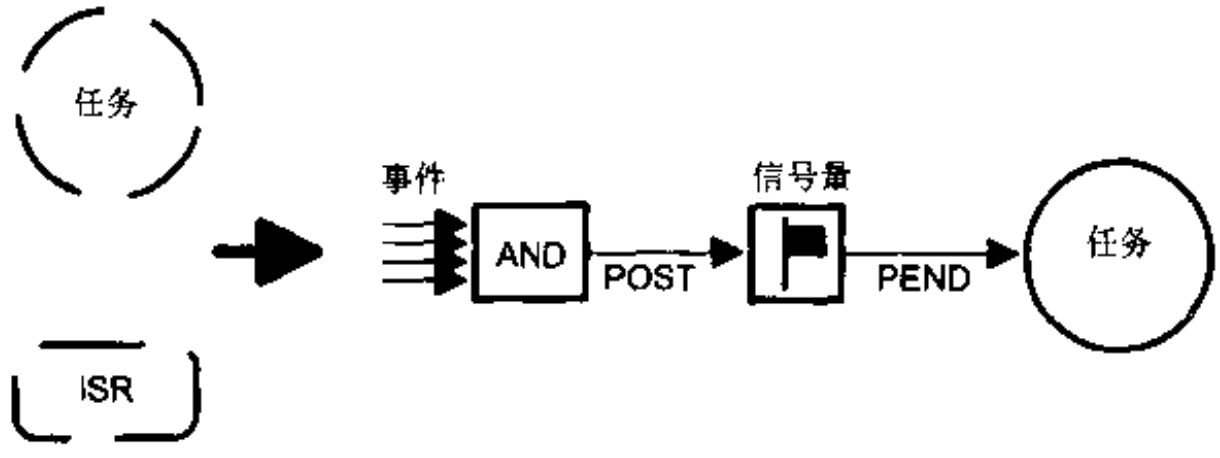
L2.10(4) 同样，当第2个任务运行到某一处时，发信号给第1个任务并等待返回信号。至此，2个任务实现了互相同步。

2.21 事件标志

当某任务要与多个事件同步时，须使用事件标志(event flag)。若任务需要与任何事件之一发生同步，可称为独立型同步(逻辑"或"关系)。任务也可以与若干事件都发生了同步，称之为关联型同步(逻辑"与"关系)。独立型及关联型同步如图F2.15所示。



"或"型同步



"与"型同步

图F2.15 独立型及关联型同步

可以用多个事件的组合发信号给多个任务，如图F2.16所示。典型的，8个、16个或32个事件可以组合在一起，取决于用的是哪种内核。每个事件占1位(bit)，以32位的情况为多。任务或中断服务可以给某一位置位或复位。当任务所需的事件都发生了，该任务继续执行。至于哪个任务该继续执行了，是在一组新的事件发生时确定的，也就是在事件位置位时做判断(执行SET操作时)。

许多内核，如μC/OS-II，支持事件标志，提供事件标志置位、事件标志清0和等待事件标志等服务。事件标志可以是独立型或关联型。

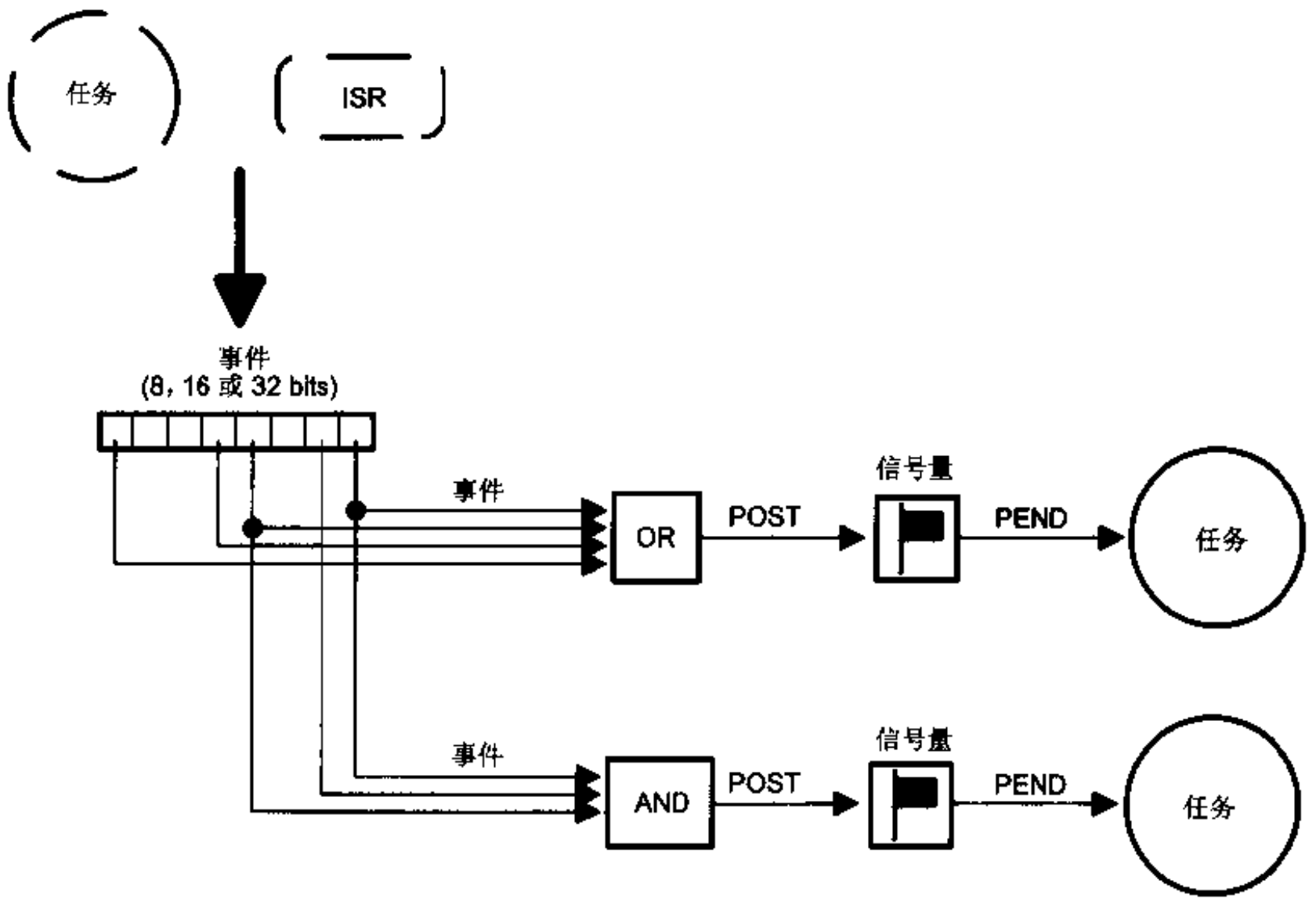


图 F2.16　事件标志

2.22　任务间通信

有时很需要任务间或中断服务与任务间的通信。这种信息传递称为任务间的通信(inter task communication)。任务间信息的传递有 2 个途径:通过全程变量,或发消息给另一个任务。

用全程变量时,必须保证每个任务或中断服务子程序独享该变量。中断服务中保证独享的惟一办法是关中断。如果 2 个任务共享某变量,各任务实现独享该变量的办法可以是关中断,再开中断,或使用信号量(如前面提到的)。请注意,任务只能通过全程变量与中断服务子程序通信,而任务并不知道什么时候全程变量被中断服务子程序修改了,除非中断服务子程序以信号量方式向任务发信号,或者是该任务以查询方式不断周期性地查询变量的值。要避免这种情况,可以考虑使用邮箱或消息队列。

2.23　消息邮箱

通过内核服务可以给任务发送消息。典型的消息邮箱(message mail box)也称做交换消息,即用一个指针型变量,一个任务或一个中断服务子程序通过内核服务,可以把一则消息(一

个指针)放到邮箱里去。同样,一个或多个任务通过内核服务,可以接收这则消息。发送消息的任务和接收消息的任务约定,该指针指向的内容就是那则消息。

每个邮箱有相应的正在等待消息的任务列表。要得到消息的任务会因为邮箱是空的,而被挂起,且被记录到等待消息的任务表中,直到接收到消息。一般地说,内核允许定义等待超时。等待消息的时间超过了,仍然没有接收到消息,该任务进入就绪态,并返回出错信息,报告等待超时错误。消息放入邮箱后,或者把消息传给等待消息的任务列表中优先级最高的任务(基于优先级),或者将消息传给最先开始等待消息的任务(基于先进先出)。μC/OS-II 只支持前者,即将消息传给等待消息的任务列表中优先级最高的任务。图 F2.17 示意把消息放入邮箱。用一个 I 字表示邮箱,旁边的小沙漏表示超时计时器,计时器旁边的数字表示定时器设定值,即任务最长可以等多少个时钟节拍(clock ticks)。关于时钟节拍以后会讲到。

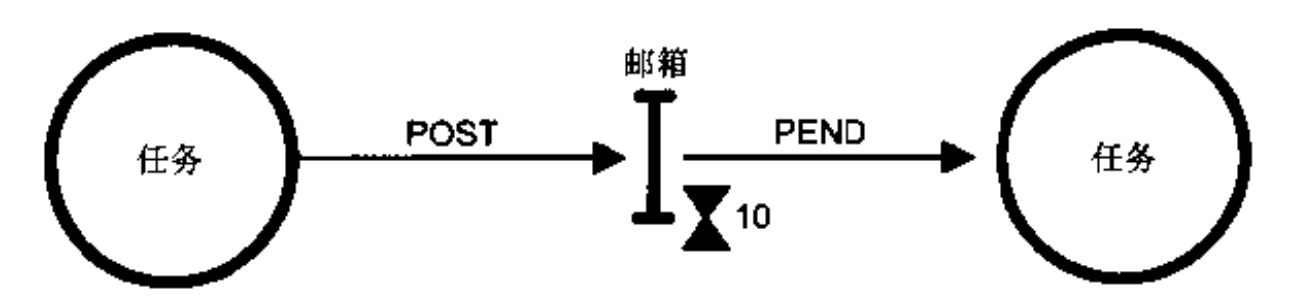


图 F2.17 消息邮箱

内核一般提供以下邮箱服务:

- 邮箱内消息内容的初始化,邮箱里最初可以有,也可以没有消息。
- 将消息放入邮箱(POST)。
- 等待消息进入邮箱(PEND)。
- 从邮箱中得到消息。如果邮箱里没有消息,则任务并不被挂起(accept);如果邮箱内有消息,就接收这则消息。

消息邮箱也可以当作只取 2 个值的信号量来用。邮箱里有消息,表示资源可以使用,而空邮箱表示资源已被其他任务占用。

2.24 消息队列

消息队列(message queue)用于给任务发消息。消息队列实际上是邮箱阵列。通过内核提供的服务,任务或中断服务子程序可以将一则消息(该消息的指针)放入消息队列。同样,一个或多个任务可以通过内核服务从消息队列中得到消息。发送和接收消息的任务约定,传递的消息实际上是传递的指针指向的内容。通常,先进入消息队列的消息先传给任务,也就是说,任务先得到的是最先进入消息队列的消息,即先进先出原则(FIFO);而 μC/OS-II 也允许使用后进先出方式(LIFO)。

像使用邮箱那样,当一个以上的任务要从消息队列接收消息时,每个消息队列有一张等待消息任务的等待列表(waiting list)。如果消息队列中没有消息,即消息队列为空,则等待消息的任务就被挂起,并放入等待消息任务列表中,直到有消息到来。通常,内核允许等待消息的任务定义等待超时。如果限定时间内任务没有收到消息,该任务就进入就绪态并开始运行,同

时返回出错代码，指出出现等待超时错误。一旦一则消息放入消息队列，该消息将传给等待消息任务列表中优先级最高的任务，或是最先进入等待消息任务列表的任务。μC/OS－II 只支持前一种机制，将消息传给等待消息任务列表中优先级最高的任务。

图 F2.18 示意，中断服务子程序如何将消息放入消息队列。图中 2 个 I 表示消息队列，"10"表示消息队列最多可以放 10 则消息，沙漏旁边的"0"表示任务没有定义超时，将永远等待下去，直至消息的到来。

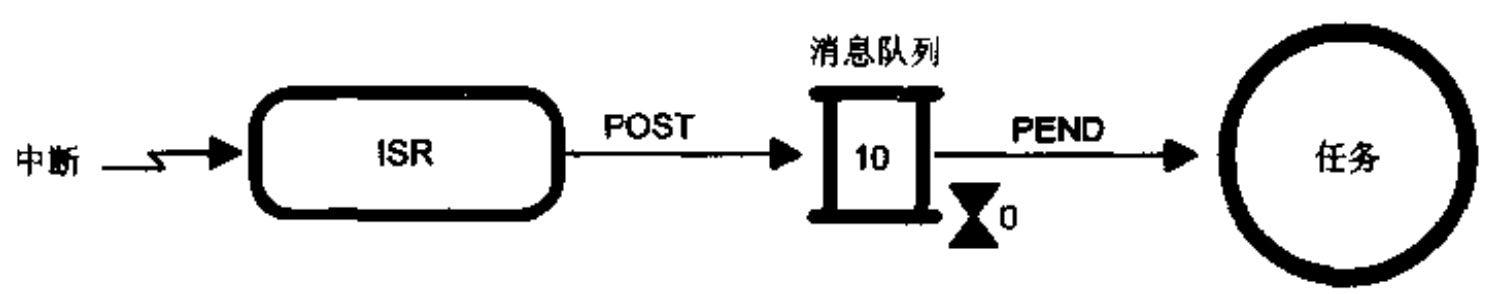


图 F2.18　消息队列

典型的，内核提供的消息队列服务如下：

- 消息队列初始化，队列初始化时总是清为空。
- 放一则消息到队列中去(POST)。
- 等待一则消息的到来(PEND)。
- 如果队列中有消息，则任务可以得到消息，但如果此时队列为空，内核并不将该任务挂起(accept)。如果有消息，则消息从队列中取走。

2.25　中　断

中断是一种硬件机制，用于通知 CPU"有个异步事件发生了"。中断一旦被识别，CPU 保存部分(或全部)现场(context)，即部分或全部寄存器的值，跳转到专门的子程序，称为中断服务子程序(ISR)。中断服务子程序做事件处理，处理完成后，程序回到：

- 在前/后台系统中，程序回到后台程序；
- 对不可剥夺型内核而言，程序回到被中断了的任务；
- 对可剥夺型内核而言，让进入就绪态的优先级最高的任务开始运行。

中断使得 CPU 可以在事件发生时才予以处理，而不必让微处理器连续不断地查询(polling)是否有事件发生。

通过 2 条特殊指令：关中断(disable interrupt)和开中断(enable interrupt)，可以让微处理器不响应或响应中断。在实时环境中，关中断的时间应尽量短。关中断影响中断延迟时间(见 2.26 节："中断延迟")。关中断时间太长，可能会引起中断丢失。微处理器一般允许中断嵌套，也就是说在中断服务期间，微处理器可以识别另一个更重要的中断，并服务于那个更重要的中断，如图 F2.19 所示。

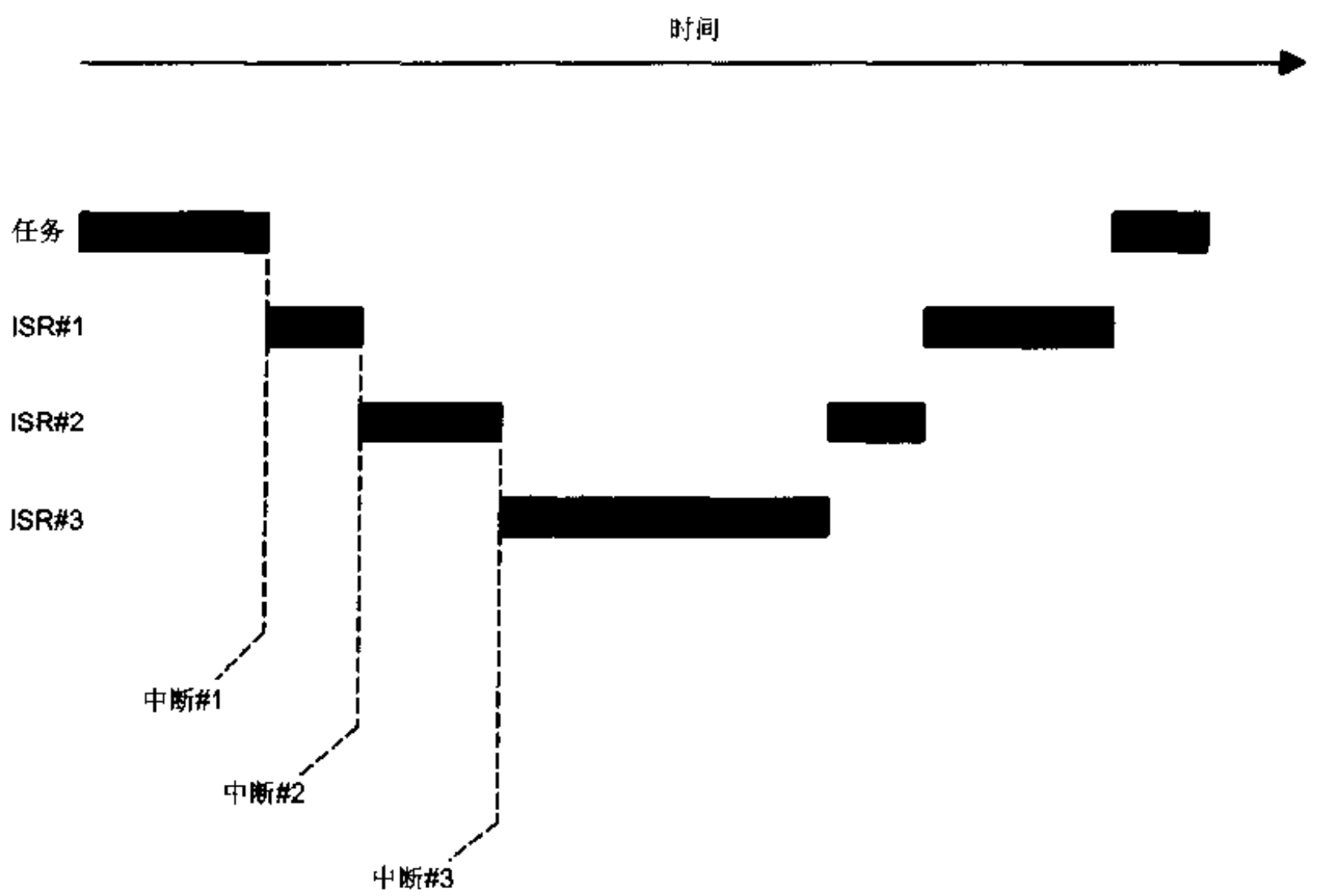


图 F2.19 中断嵌套

2.26 中断延迟

实时内核最重要的指标可能就是关中断的时间的长短。所有实时系统在进入临界区代码段之前，都要关中断；执行完临界代码之后，再开中断。关中断的时间越长，中断延迟就越长。中断延迟由表达式(2.2)给出。

中断延迟 = 关中断的最长时间 + 开始执行中断服务子程序第 1 条指令的时间 (2.2)

2.27 中断响应

中断响应定义为：从中断发生到开始执行用户的中断服务子程序代码来处理这个中断的时间。中断响应时间包括开始处理这个中断前的全部开销。典型的，执行用户代码之前，要保护现场，将 CPU 的各寄存器推入堆栈。这段时间将被记作中断响应时间。

对前/后台系统，保存寄存器以后立即执行用户代码，中断响应时间由式(2.3)给出。

中断响应时间 = 中断延迟 + 保存 CPU 内部寄存器的时间 (2.3)

对于不可剥夺型内核，微处理器保存内部寄存器以后，中断服务子程序代码立即全部得到执行。

不可剥夺型内核的中断响应时间由式(2.4)给出。

$$中断响应时间 = 中断延迟 + 保存\ CPU\ 内部寄存器的时间 \tag{2.4}$$

对于可剥夺型内核，则要先调用一个特定的函数。该函数通知内核即将开始中断服务，使得内核可以跟踪中断的嵌套。需要这个函数的原因，请参见 2.28 节："中断恢复时间"。对于 μC/OS－II，这个函数是 OSIntEnter()。

可剥夺型内核的中断响应时间由式(2.5)给出。

$$\begin{aligned}中断响应 = &中断延迟 + 保存\ CPU\ 内部寄存器的时间 + \\ &内核进入中断服务函数的执行时间\end{aligned} \tag{2.5}$$

中断响应是系统在最坏情况下的响应中断的时间。某系统 100 次中有 99 次在 50 μs 之内响应中断，只有 1 次响应中断的时间是 250 μs，只能认为中断响应时间是 250 μs。

2.28　中断恢复时间

对于可剥夺型内核，中断恢复时间(interrupt recovery)定义为：微处理器返回到被中断了的程序代码所需要的时间，或返回到更高优先级任务的时间。在前/后台系统中，中断恢复时间只包括恢复 CPU 内部寄存器值的时间和执行中断返回指令的时间。中断恢复时间由式(2.6)给出。

$$中断恢复时间 = 恢复\ CPU\ 内部寄存器值的时间 + 执行中断返回指令的时间 \tag{2.6}$$

同前/后台系统一样，不可剥夺型内核的中断恢复时间也只包括恢复 CPU 内部寄存器值的时间和执行中断返回指令的时间，如式(2.7)所示。

$$中断恢复时间 = 恢复\ CPU\ 内部寄存器值的时间 + 执行中断返回指令的时间 \tag{2.7}$$

对于可剥夺型内核，中断的恢复要复杂一些。典型的，在中断服务子程序的末尾，要调用一个由实时内核提供的函数。在 μC/OS－II 中，这个函数叫做 OSIntExit()。这个函数用于判断中断是否脱离了所有的中断嵌套。如果脱离了嵌套(已经可以返回到被中断了的任务级时)，由于中断服务子程序 ISR 的执行，内核要判断是否使得一个优先级更高的任务进入了就绪态。如果是，则要让这个优先级更高的任务开始运行。在这种情况下，被中断了的任务只有重新成为优先级最高的任务而进入就绪态时，才能继续运行。对于可剥夺型内核，中断恢复时间由式(2.8)给出。

$$\begin{aligned}中断恢复时间 = &判断是否有优先级更高的任务进入了就绪态的时间 + \\ &恢复优先级更高的任务的\ CPU\ 内部寄存器的时间 + \\ &执行中断返回指令的时间\end{aligned} \tag{2.8}$$

2.29 中断延迟、响应及恢复

图 F2.20～F2.22 示意前/后台系统、不可剥夺型内核及可剥夺型内核相应的中断延迟、响应及恢复过程。

注意：对于可剥夺型实时内核，中断返回函数将决定是返回到被中断的任务[参见图 F2.22 (A)]，还是运行因中断服务子程序而进入就绪态的优先级最高的任务[参见图 F2.22 (B)]。

在后一种情况下，中断恢复时间要稍长一些，因为内核要做任务切换。对于μC/OS-II，执行时间的不同大致可以知道。

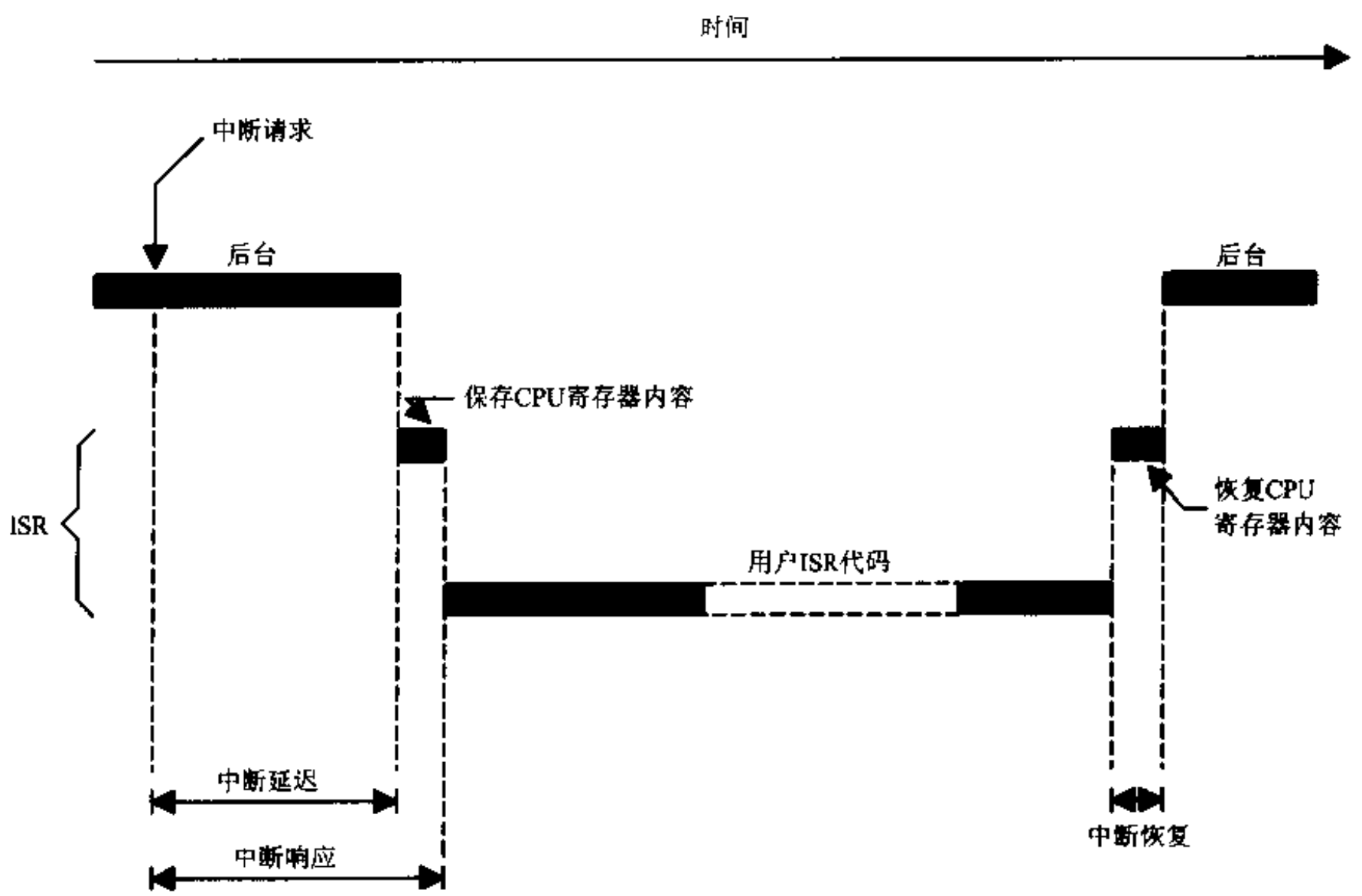


图 F2.20 中断延迟、响应及恢复(前/后台系统)

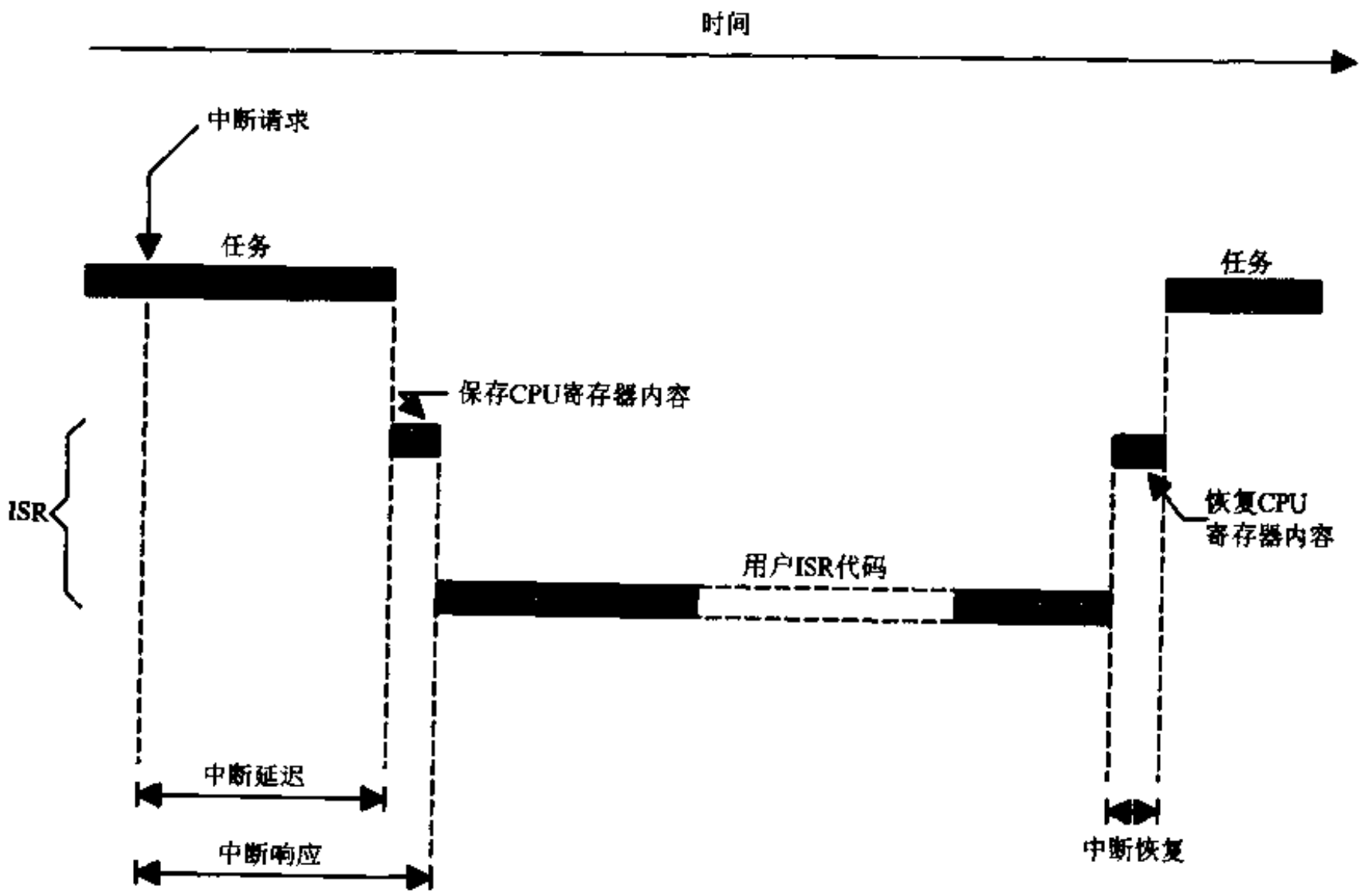


图 F2.21　中断延迟、响应及恢复(不可剥夺型内核)

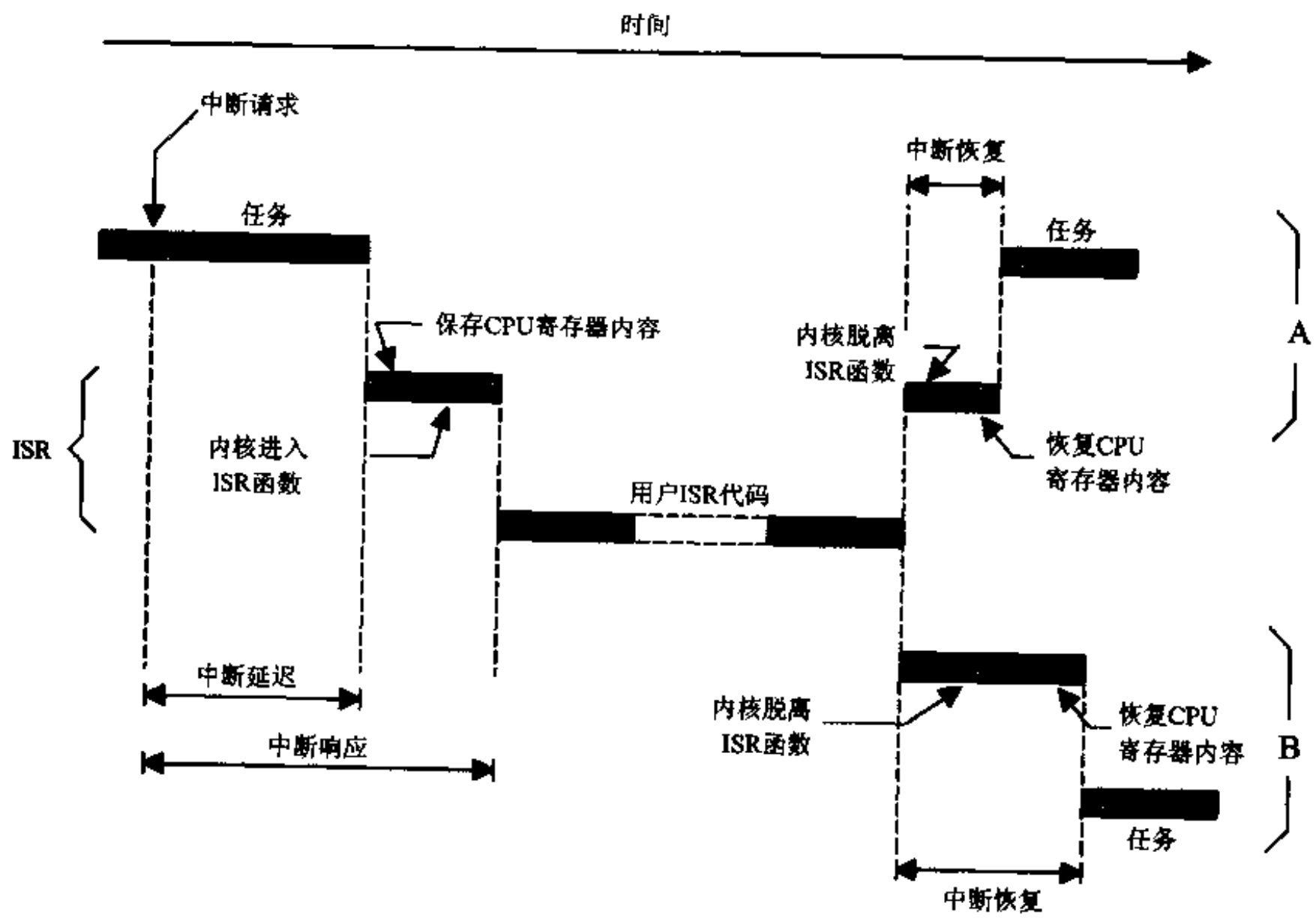


图 F2.22　中断延迟、响应及恢复(可剥夺型内核)

2.30 中断处理时间

虽然中断服务的处理时间应该尽可能短，但是对处理时间并没有绝对的限制。不能说中断服务时间必须全部短于 100 μs，500 μs 或 1 ms。如果中断服务是在任意给定的时间开始，且中断服务子程序代码是应用程序中最重要的代码，则中断服务需要多长时间，就应该给它多长时间。然而在大多数情况下，中断服务子程序应识别中断来源，从请求中断的设备取得数据或状态，并通知真正做该事件处理的任务。当然应该考虑到，通知一个任务去做事件处理所花的时间是否比处理这个事件所花的时间还多。在中断服务中，通知一个任务做时间处理(通过信号量、邮箱或消息队列)是需要一定时间的。如果事件处理需花的时间短于给一个任务发通知的时间，就应该考虑在中断服务子程序中做事件处理，并在中断服务子程序中开中断，以允许优先级更高的中断打入并优先得到服务。

2.31 非屏蔽中断

有时，中断服务必须尽可能地快，而内核引起的延时长得不可忍受。在这种情况下，可以使用非屏蔽中断(NMI)。绝大多数微处理器具有非屏蔽中断功能。通常非屏蔽中断留做紧急处理用，如断电时保存重要的信息；然而，如果应用程序没有这方面的要求，非屏蔽中断可用于时间要求最苛刻的中断服务。下列表达式给出如何确定中断延迟时间、中断响应时间及中断恢复时间。

中断延迟时间 = 指令执行时间中最长的时间 + 开始做非屏蔽中断服务的时间 (2.9)

中断响应时间 = 中断延迟时间 + 保存 CPU 寄存器的时间 (2.10)

中断恢复时间 = 恢复 CPU 寄存器的时间 + 执行中断返回指令的时间 (2.11)

在一项应用中，笔者将非屏蔽中断用于可能每 150 μs 发生 1 次的中断。中断处理时间 80 ~125 μs，所使用内核的关中断时间是 45 μs。可以看出，如果使用可屏蔽中断，中断响应会推迟 20 μs(125 μs+45 μs>150 μs)。

在非屏蔽中断的中断服务子程序中，不能使用内核提供的服务。因为非屏蔽中断是关不掉的，故不能在非屏蔽中断处理中处理临界区代码；然面向非屏蔽中断传送参数，或从非屏蔽中断获取参数，还是可以进行的。参数的传递必须使用全程变量，全程变量的位数必须是 1 次读或写能完成的，即不应该是 2 个分离的字节，需 2 次读或写才能完成。

非屏蔽中断可以用增加外部电路的方法禁止，如图 F2.23 所示。假定中断源和非屏蔽中断都是正逻辑，用一个简单的"与"门插在中断源和微处理器的非屏蔽中断输入端之间。向输出口(output port)写 0，就可将中断关闭。不一定要以这种关中断方式使用内核服务，但可以用这种方式在中断服务子程序和任务之间传递参数(大的、多字节的、一次读/写不能完成的变量)。

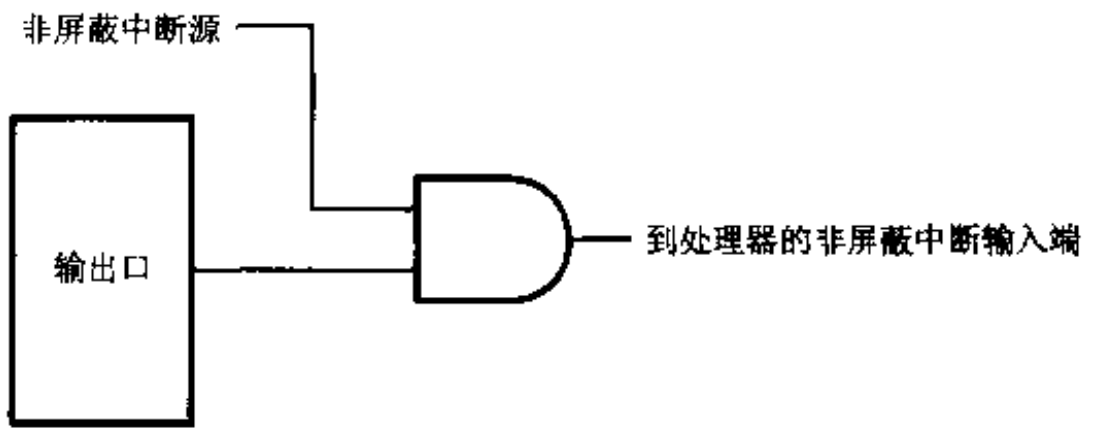


图 F2.23　非屏蔽中断的禁止

假定非屏蔽中断服务子程序每 40 次执行中，有 1 次要给任务发信号，如果非屏蔽中断 150 μs 执行一次，则每 6 ms(40×150 μs)给任务发一次信号。在非屏蔽中断服务子程序中，不能使用内核服务给任务发信号；但可以使用如图 F2.24 所示的中断机制，即用非屏蔽中断产生普通可屏蔽中断的机制。在这种情况下，非屏蔽中断通过某一输出口产生硬件中断(置输出口为有效电平)。由于非屏蔽中断服务通常具有最高的优先级，在非屏蔽中断服务过程中，不允许中断嵌套，普通中断一直要等到非屏蔽中断服务子程序运行结束后，才能被识别。在非屏蔽中断服务子程序完成以后，微处理器开始响应这个硬件中断。在这个中断服务子程序中，须清除中断源(置输出口为无效电平)，然后用信号量唤醒需要唤醒的任务。任务本身的运行时间和信号量的有效时间都接近 6 ms，实时性得到了满足。

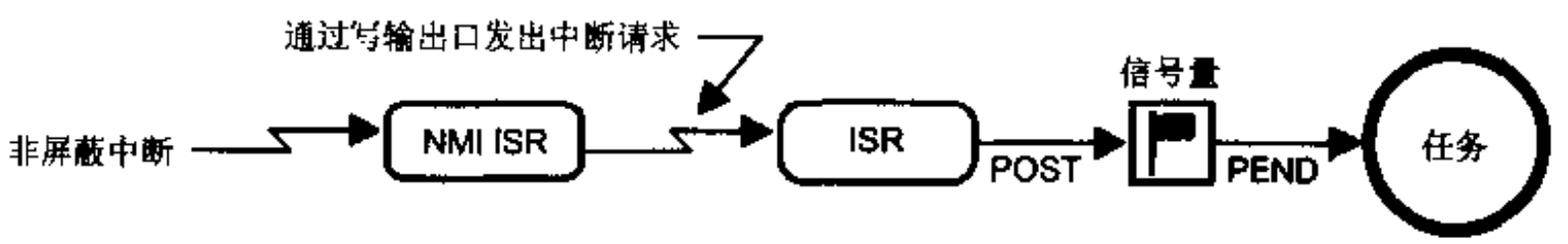


图 F2.24　非屏蔽中断产生普通可屏蔽中断

2.32　时钟节拍

时钟节拍(clock tick)是特定的周期性中断。这个中断可以看做是系统心脏的脉动。中断之间的时间间隔取决于不同的应用，一般为 10～200 ms。时钟的节拍式中断使得内核可以将任务延时若干个整数时钟节拍，以及当任务等待事件发生时，提供等待超时的依据。时钟节拍率越快，系统的额外开销就越大。

各种实时内核都有将任务延时若干个时钟节拍的功能，然而这并不意味着延时的精度是 1 个时钟节拍，只是在每个时钟节拍中断到来时，对任务延时做一次裁决而已。

图 F2.25～F2.27 示意任务将自身延迟 1 个时钟节拍的时序。阴影部分是各部分程序的执行时间。

注意：相应的程序运行时间是长短不一的，这反映了程序中含有循环和条件转移语句(if/else，switch，？：等语句)的典型情况。时钟节拍中断服务子程序的运行时间也是不一样的，尽管在图中画得有所夸大。

第 1 种情况：如图 F2.25 所示，优先级高的任务和中断服务超前于要求延时 1 个时钟节

拍的任务运行。由图可以看出，虽然该任务想要延时 20 ms，但由于其优先级的缘故，实际上每次延时多少是变化的。这种执行时间的变化引起了任务执行时间的抖动。

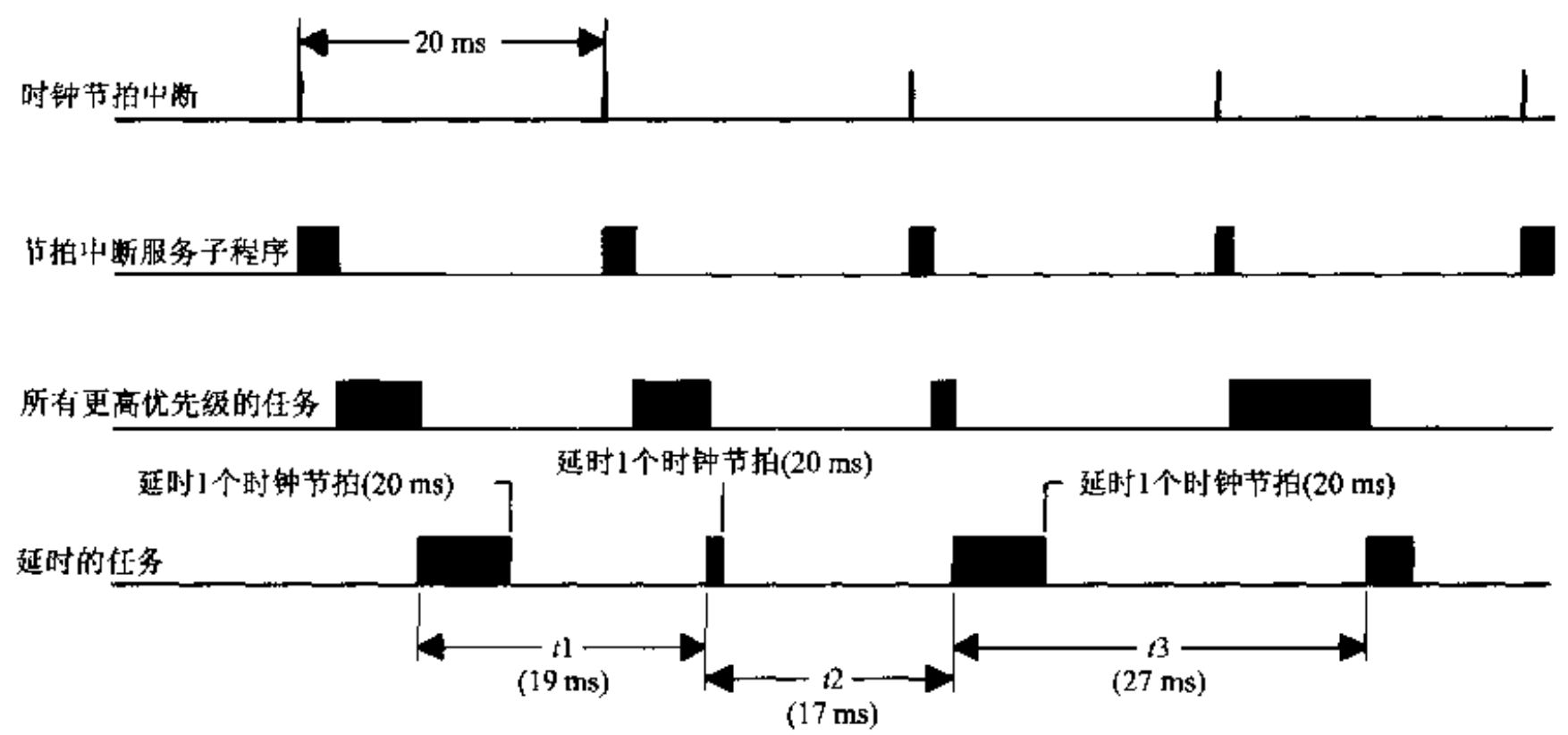


图 F2.25 将任务延时 1 个时钟节拍(第 1 种情况)

第 2 种情况：如图 F2.26 所示，所有高优先级的任务和中断服务的执行时间略微短于 1 个时钟节拍。如果任务将自己延时 1 个时钟节拍的请求刚好发生在下一个时钟节拍之前，则这个任务的再次执行几乎是立即开始的；因此，如果要求任务的延时至少为 1 个时钟节拍，则要多定义 1 个延时时钟节拍。换句话说，如果想要将一个任务至少延时 5 个时钟节拍，则应在程序中延时 6 个时钟节拍。

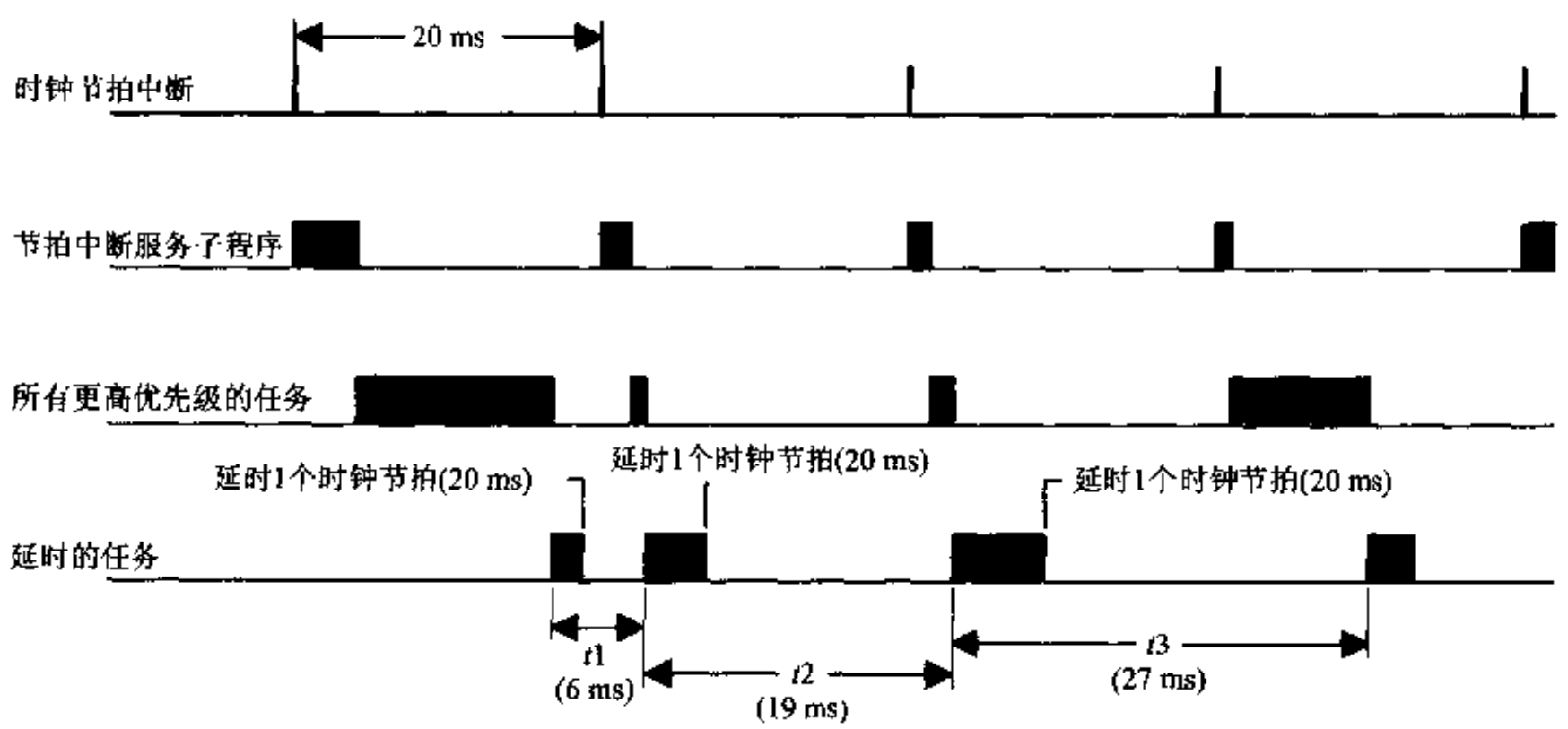


图 F2.26 将任务延时 1 个时钟节拍(第 2 种情况)

第 3 种情况：如图 F2.27 所示，所有高优先级的任务加上中断服务的执行时间长于 1 个时钟节拍。在这种情况下，拟延时 1 个时钟节拍的任务实际上在 2 个时钟节拍后开始运行，引起了延时时间超差。这种延时时间的超差在某些应用中或许是可以的，而在多数情况下是不可接受的。

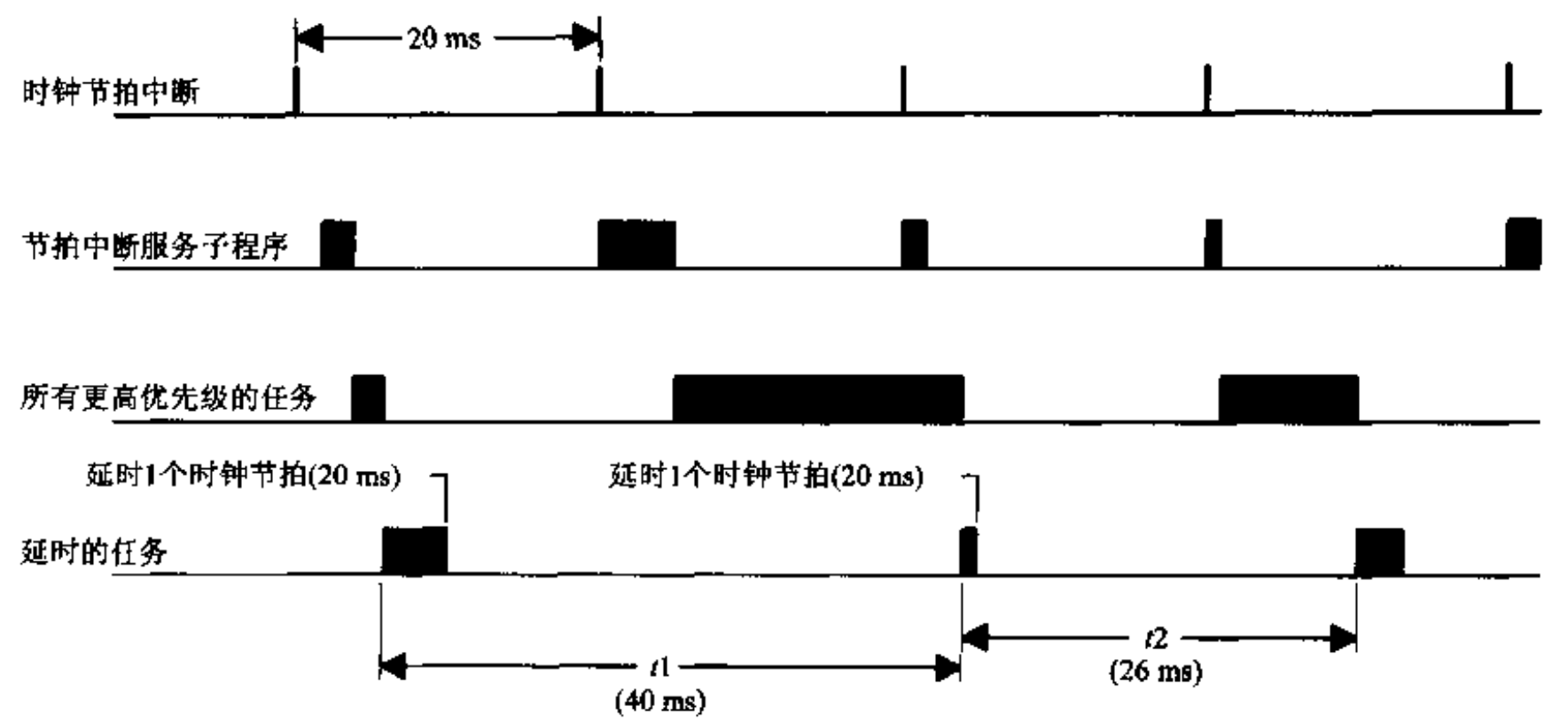


图 F2.27　将任务延时 1 个时钟节拍(第 3 种情况)

上述情况在所有的实时内核中都会出现，这与 CPU 负荷有关，也可能与系统设计不正确有关。以下是这类问题可能的解决方案：

- 增加微处理器的时钟频率；
- 增加时钟节拍的频率；
- 重新安排任务的优先级；
- 避免使用浮点运算(如果非使用不可，则尽量用单精度数)；
- 使用能较好地优化程序代码的编译器；
- 时间要求苛刻的代码用汇编语言写；
- 如果可能，则用同一家族且更快的微处理器做系统升级，如从 8086 向 80186 升级，从 68000 向 68020 升级等。

不管怎么样，抖动总是存在的。

2.33　对存储器的需求

如果设计是前/后台系统，那么对存储器容量的需求仅仅取决于应用程序代码；而使用多任务内核时的情况则很不一样，内核本身需要额外的代码空间(ROM)。内核的大小取决于多种因素，取决于内核的特性，1～100 KB 都是可能的。8 位 CPU 用的最小内核只提供任务调度、任务切换、信号量处理、延时及超时服务，约需要 1～3 KB 代码空间。代码空间总需求量由表达式(2.12)给出。

$$\text{总代码量} = \text{应用程序代码} + \text{内核代码} \tag{2.12}$$

因为每个任务都是独立运行的，必须给每个任务提供单独的栈空间(RAM)。应用程序设计人员决定分配给每个任务多少栈空间时，应该尽可能使之接近实际需求量(有时，这是相当困难的一件事)。决定栈空间的大小，不仅须计算任务本身的需求(局部变量、函数调用等

等），还须计算最多中断嵌套层数（保存寄存器、中断服务程序中的局部变量等）。根据不同的目标微处理器和内核的类型，任务栈和系统栈可以是分开的。系统栈专门用于处理中断级代码。这样做有许多好处，每个任务需要的栈空间可以大大减少。另一个内核应该具有的性能是，每个任务所需的栈空间大小可以分别定义（μC/OS-II 可以做到）；相反，有些内核要求每个任务所需的栈空间都相同。所有内核都需要额外的栈空间，以保证内部变量、数据结构及队列等的需求。如果内核不支持单独的中断用栈，则总的 RAM 需求由表达式(2.13)给出。

$$\text{RAM 总需求} = \text{应用程序的 RAM 需求} + (\text{任务栈需求} + \text{最多中断嵌套栈需求}) \times \text{任务数} \tag{2.13}$$

如果内核支持中断用栈分离，总 RAM 需求量由表达式(2.14)给出。

$$\text{RAM 总需求} = \text{应用程序的 RAM 需求} + \text{内核数据区的 RAM 需求} + \text{各任务栈需求之总和} + \text{最多中断嵌套栈需求} \tag{2.14}$$

除非有特别大的 RAM 空间可以使用，对栈空间的分配与使用要非常谨慎。为减少应用程序需要的 RAM 空间，对每个任务栈空间的使用都要非常谨慎，特别要注意以下几点：

- 定义函数和中断服务子程序中的局部变量，特别是定义大型数组和数据结构；
- 函数（子程序）的嵌套；
- 中断嵌套；
- 库函数需要的栈空间；
- 多变元的函数调用。

综上所述，多任务系统比前/后台系统需要更多的代码空间（ROM）和数据空间（RAM）。额外的代码空间取决于内核的大小，而 RAM 的用量取决于系统中的任务数。

2.34 使用实时内核的优、缺点

实时内核也称为实时操作系统或 RTOS。它的使用使得实时应用程序的设计和扩展变得容易，无需大的改动就可以增加新的功能。实际上，如果用户给系统增加一些低优先级的任务，那么用户系统对高优先级的任务的响应时间几乎不受影响。通过将应用程序分割成若干独立的任务，RTOS 使得应用程序的设计过程大为简化。使用可剥夺型内核时，所有时间要求苛刻的事件都得到了尽可能快捷、有效的处理。通过有效的服务，如信号量、邮箱、队列、延时及超时等，RTOS 使资源得到更好的利用。

如果应用项目对额外的需求可以承受，应该考虑使用实时内核。这些额外的需求是：内核的价格、额外的 ROM/RAM 开销及 2%～4%的 CPU 额外负荷。

另外一个因素是使用实时内核增加的价格成本。在一些应用中，价格就是一切，以至于无法考虑使用 RTOS。

当今有 150 个以上的 RTOS 商家，生产面向 8 位、16 位、32 位甚至 64 位的微处理器的 RTOS 产品。一些软件包是完整的操作系统，不仅包括实时内核，还包括输入/输出管理、视窗

系统(用于显示)、文件系统、网络、语言接口库、调试软件及交叉平台编译(cross-platform compilers)。开发中,RTOS 的成本从 70 $ 到远高于 30 000 $。RTOS 制造商还可能索取每个目标系统的版权使用费。像从 RTOS 商家买一个芯片安装到每一个产品上,然后一同出售,RTOS 商家称之为硅片软件(silicon software),每个产品的版权费从 5 $ 到超过 500 $。μC/OS-II 不是自由软件,用于商业目的时须取得许可证(见附录 B:"μC/OS-II 的使用许可证")。同如今的其他软件包一样,还须考虑软件维护费。这部分开销每年大约是 RTOS 价格的 15%。

2.35　实时系统小结

3 种类型的实时系统归纳于表 T2.2 中。这 3 种实时系统是:前/后台系统、不可剥夺型内核及可剥夺型内核。

表 T2.2　实时系统小结

	前/后台系统	不可剥夺型内核	可剥夺型内核
中断延时时间	MAX(最长指令时间,用户关中断时间)+中断向量跳转时间	MAX(最长指令时间,用户关中断时间,内核关中断时间)+中断向量跳转时间	MAX(最长指令时间,用户关中断时间,内核关中断时间)+中断向量跳转时间
中断响应时间	中断延时时间+保存 CPU 状态时间	中断延时时间+保存 CPU 状态时间	中断延时时间+保存 CPU 状态时间+内核 ISR 进入时间
中断恢复时间	后台状态恢复时间+中断返回时间	任务状态恢复时间+中断返回时间	寻找最高优先级任务的时间+最高优先级任务的恢复时间+中断返回时间
任务响应时间	后台	最长任务的执行时间+寻找最高优先级任务的时间+任务切换时间	寻找最高优先级任务的时间+任务切换时间
ROM 大小	应用程序代码	应用程序代码+内核代码	应用程序代码+内核代码
RAM 大小	应用程序代码	应用程序代码+内核 RAM+SUM(任务栈+MAX(ISR 栈))	应用程序代码+内核 RAM+SUM(任务栈+MAX(ISR 栈))
可获得的服务	应用程序代码必须提供	有	有

第 3 章

内核结构

本章描述 μC/OS - II 的结构概貌。将学习以下内容：

- μC/OS - II 是怎样处理临界段代码的；
- 什么是任务；
- μC/OS - II 是怎样得知用户任务的；
- 任务是怎样调度的；
- μC/OS - II 是怎样知道应用程序 CPU 利用率的；
- 怎样写中断服务子程序(ISR)；
- 什么是时钟节拍，μC/OS - II 是怎样处理时钟节拍的；
- μC/OS - II 是怎样初始化的；
- 怎样启动多任务。

本章还描述表 T3.1 中列出的应用功能函数。OSSchedLock()和 OSSchedUnlock()可以通过设 OS_CFG.H 文件中的 OS_SCHED_LOCK_EN 值为 0 而禁止掉，如表 T3.1 所列。

注意：表中其他函数不能从编译生成的代码中去掉，因为它们是 μC/OS - II 中必不可少的部分。

表 T3.1　OS_CFG.H 文件中的核心功能函数的配置常数

μC/OS - II 内核功能函数	OS_CFG.H 中置 1 允许相应函数
OS_ENTER_CRITICAL()	
OS_EXIT_CRITICAL()	
OSInit()	
OSStart()	
OSIntEnter()	
OSIntExit()	
OSSchedLock()	OS_SCHED_LOCK_EN
OSSchedUnlock()	OS_SCHED_LOCK_EN
OSVersion()	

图 F3.1 示意 μC/OS - II 的文件结构以及与硬件的关系。在应用程序中使用 μC/OS - II 时，需要用户提供的是应用软件和 μC/OS - II 的配置部分。书后所附的光盘中包括全部与处理器类型无关的源代码。处理器类型相关部分的源代码是针对 Intel 80x86 的，为实模式，以大模块编译。如果打算将 μC/OS - II 用到不同的处理器上，则可查看 μC/OS - II 的网站

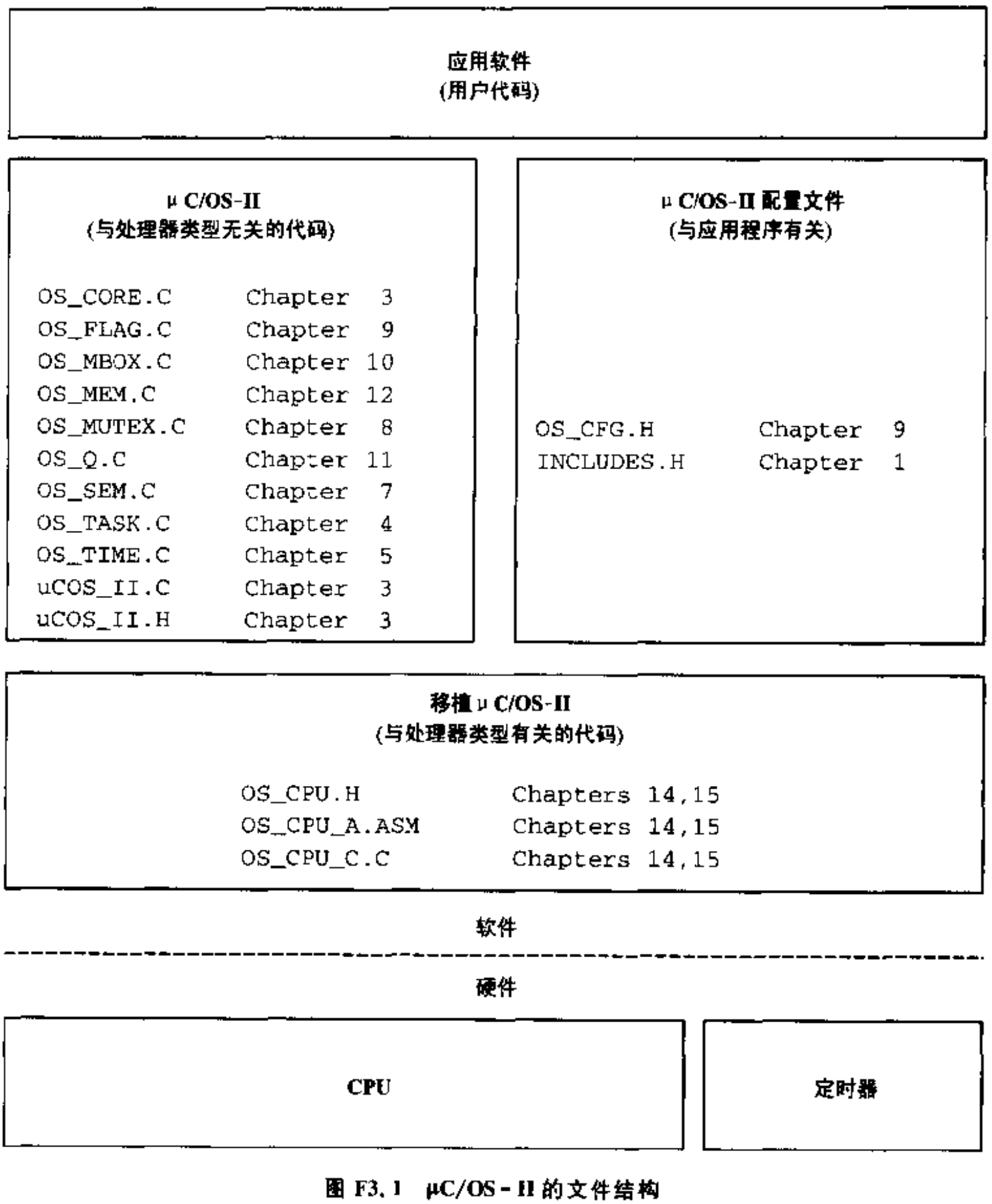


图 F3.1　μC/OS-II 的文件结构

www.uCOS-II.com，看是否有现成的、想用的处理器的移植代码。如果没有，可以自己写这一部分。

3.00　临界段，OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()

同其他内核一样，μC/OS-II 为了处理临界段(critical sections)代码，须关中断，处理完毕后，再开中断。关中断使得 μC/OS-II 能够避免同时有其他任务或中断服务进入临界段代码。关中断的时间是实时内核开发商应提供的最重要的指标之一，因为这个指标影响用户系

统对实时事件的响应特性。μC/OS-II 努力使关中断时间降至最短，但就使用 μC/OS-II 而言，关中断的时间很大程度上取决于微处理器的结构以及编译器所生成的代码质量。

微处理器一般都具有关中断/开中断指令，用户使用的 C 语言编译器必须具有某种机制，能够在 C 中直接实现关中断/开中断操作。有些 C 编译器允许在用户的 C 源代码中插入汇编语言的语句，使得插入微处理器指令来关中断/开中断很容易实现；而有些编译器把从 C 语言中关中断/开中断的操作放在语言的扩展部分，直接从 C 语言中可以关中断/开中断。

μC/OS-II 定义 2 个宏(macros)来关中断和开中断，以便避免不同 C 编译器厂商选择不同的方法来处理关中断和开中断。μC/OS-II 中的这 2 个宏调用分别是：OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()。因为这 2 个宏的定义取决于使用的微处理器，故在文件 OS_CPU.H 中可以找到相应的宏定义。每种微处理器都有自己的 OS_CPU.H 文件。

OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()总是成对使用的，把临界段代码封包起来，如以下代码所列：

```
{
    .
    .
    OS_ENTER_CRITICAL();
    /* μC/OS-II 临界段代码 */
    OS_EXIT_CRITICAL();
    .
    .
}
```

OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()也可以用来保护应用程序中的临界段代码；然而要特别小心，如果在调用一些如 OSTimeDel()之类的功能函数之前关中断，应用程序将会崩溃(死机，见第 5 章："时间管理")。这个问题是这样引起的：任务被挂起一段时间，直到挂起时间到，但由于中断关掉了，时钟节拍中断一直得不到服务。显然，所有的挂起类(PEND)调用都有这类问题，所以要特别小心。作为一条普遍适用的规则，调用 μC/OS-II 功能函数时，中断总应当是开着的。

OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 可以用不同的方法实现。具体用哪种方法，取决于用户打算移植到的处理器的性能以及所用的 C 编译器(见第 13 章："移植 μC/OS-II")。用定义(#define)常数 OS_CRITICAL_METHOD 可以选择具体使用哪种方法。该常数是在用户应用程序(即产品)的移植文件 OS_CPU.H 中定义的。

OS_CRITICAL_METHOD==1

第 1 种办法是以最简单的方式来实现这 2 个宏调用的：用处理器指令关中断，完成 OS_ENTER_CRITICAL()，用开中断完成 OS_EXIT_CRITICAL()。然而，这种办法有点小问题，如果调用 μC/OS-II 的功能函数时，中断是关掉的，则从 μC/OS-II 的函数返回时，中断就打开了。若调用 μC/OS-II 的函数之前已将中断关掉，那么用户往往希望从 μC/OS-II 的函数返回时，中断仍然是关掉的。在此情况下，这种实现方法就不妥当；但是对一些特定的

处理器或编译器，使用这种方法是惟一的选择。

OS_CRITICAL_METHOD==2

实现 OS_ENTER_CRITICAL()的第 2 种办法是在堆栈中保存中断的开/关状态，然后再关中断。在实现 OS_EXIT_CRITICAL()时，只需简单地从堆栈中弹出原来中断的开/关状态即可。利用这种机制，不论用户在调用 μC/OS-II 的函数之前中断是开着的还是关着的，函数的进入和返回状态都得到了保护。换句话说，如果调用前中断是打开的，则调用后仍是打开的；如果调用前中断是关掉的，则调用后仍是关掉的。关中断后调用 μC/OS-II 的函数要特别小心，因为关中断延长了应用程序的中断延迟时间。这 2 个宏调用的示意性代码如下：

```
#define OS_ENTER_CRITICAL()          \
        asm(" PUSH     PSW")         \
        asm(" DI")

#define OS_EXIT_CRITICAL()           \
        asm(" POP      PSW")
```

如以上代码所列，这里，假设用户使用的编译器允许直接在 C 代码行之间插入行汇编语句。请查阅所用编译器的说明文件。

PUSH PSW 指令将处理器状态字推入堆栈(也称做条件码寄存器或处理器标志)。DI 指令表示关中断。POP PSW 指令将中断标志从堆栈中弹出，恢复原来的状态。这里使用的指令只是为了解释之便，可能并非处理器的真实指令。

一些编译器对插入的行汇编代码优化得并不好，上述办法未必可行，因为编译器可能没有那么灵敏——知道堆栈指针的值因 PUSH 指令而改变了。特别是当使用的处理器有堆栈指针相对寻址模式时，编译器用这种模式对局部变量寻址，或者用栈指针加偏移量来传递参数。显然，如果 OS_ENTER_CRITICAL()宏调用改变了堆栈指针，所有的栈偏移量都可能出现偏差，并导致严重错误。

OS_CRITICAL_METHOD==3

一些编译器提供了扩展功能，用户可以得到当前处理器状态字的值，并保存在 C 函数的局部变量之中。这个变量可以用于恢复 PSW，如程序清单 L3.1 所列。

程序清单 L3.1　保护和恢复处理器状态字 PSW

```
void Some_uCOS_II_Service (arguments)
{
    OS_CPU_SR  cpu_sr;                                  (1)

    .
    .
    cpu_sr = get_processor_psw();                       (2)
    disable_interrupts();                               (3)
```

```
    .
    /* 临界段代码 */                                    (4)
    .
    set_processor_psw(cpu_sr);                          (5)
    .
}
```

L3.1(1) OS_CPU_SR 是在 μC/OS-II 中依赖于处理器的相关文件 OS_CPU.H 定义的。当应用程序选择这种进入临界段代码方式时，OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()将会多定义一个局部变量 cpu_sr。换句话说，如果使用这种方式保护自己的临界段代码，则应在应用程序的相关函数中多定义一个局部变量 cpu_sr。

L3.1(2) 为了进入临界段代码，调用一个编译器提供的函数，得到 PSW(条件码寄存器、处理器标志，或在用户的处理器中称做其他的名字)的值。把这个函数叫做 get_processor_psw()，是为了讨论方便。在用户使用的编译器中，该函数用的可能是其他名字。

L3.1(3) 自然会调用编译器提供的另一个函数(disable_interrupt())来关中断。

L3.1(4) 至此，可执行临界段代码了。

L3.1(5) 临界段代码执行完后，可以调用另一个编译器提供的功能扩展函数，重新开中断。为便于讨论，称之为 set_processor_psw()。该函数中有一个参数是原来的处理器状态字 psw。执行这个函数可恢复原来处理器中断的开/关状态。

由于笔者不知道编译器提供的函数的具体名称(没有标准的命名法)，μC/OS-II 中的宏调用按其功能可以写成：

```
#define OS_ENTER_CRITICAL()                 \
        cpu_sr = get_processor_psw();       \
        disable_interrupts();

#define OS_EXIT_CRITICAL()                  \
        set_processor_psw(cpu_sr);
```

3.01 任 务

任务通常是一个无限的循环，如程序清单 L3.2 所列。

程序清单 L3.2　任务是一个无限循环

```
void YourTask (void *pdata)                                    (1)
{
    for (;;) {                                                 (2)
        /* 用户代码 */
        调用 μC/OS-II 的某些功能函数:
        OSFlagPend();
        OSMboxPend();
        OSMutexPend();
        OSQPend();
        OSSemPend();
        OSTaskDel(OS_PRIO_SELF);
        OSTaskSuspend(OS_PRIO_SELF);
        OSTimeDly();
        OSTimeDlyHMSM();
        /* 用户代码 */
    }
}
```

L3.2(1)　返回参数类型必须定义成 void。当任务开始执行时，会有一个参数传递给用户任务代码。

注意：这个参数是一个指向 void 的指针，以允许用户应用程序向该任务传递任何类型的参数。

这个指针实际上是一部万能的车子，用于装载传递给任务的参数，这个参数可以是一个变量的地址、一个数据结构，或者，如果有必要，甚至可以是一个函数的入口地址。也可以建立许多相同的任务（见第 1 章的范例 1），所有任务都使用同一个函数（或者说是同一个任务程序代码）。

例如，可以将 4 个串行口安排成每个串行口都是一个单独的任务，而每个任务的代码实际上是相同的，并不需要将代码复制 4 次。可以建立一个任务，向这个任务传入一个指向某数据结构的指针变量，这个数据结构定义串行口的参数（波特率、I/O 口地址及中断向量号等）。

L3.2(2)　如果愿意，也可以使用 while (1)语句，实现无限的循环。

任务看起来如同其他 C 函数的代码，有返回值类型和参数，但任务函数永远不会返回。

不同的是，当任务完成后，任务可以自我删除，如程序清单 L3.3 所列。

注意：任务代码并非真的删除了，μC/OS-II 只是简单地不再理会这个任务。这个任务的代码也不会再运行了，即使任务调用了 OSTaskDel()，这个任务也绝不会返回。

程序清单 L3.3 任务完成后的自我删除

```
void YourTask (void *pdata)
{
    /* 用户代码 */
    OSTaskDel(OS_PRIO_SELF);
}
```

μC/OS-II可以管理多达64个任务,但目前版本的μC/OS-II有2个任务已经被系统占用了。建议用户不要使用优先级为0,1,2,3的任务,以及OS_LOWEST_PRIO-3,OS_LOWEST_PRIO-2,OS_LOWEST_PRIO-1及OS_LOWEST_PRIO,因为未来的μC/OS-II版本中,可能会用到这些任务;然而,如果要使应用程序尽量保持紧凑,也可以按照需要使用这些优先级,只是不要使用OS_LOWEST_PRIO。OS_LOWEST_PRIO是作为定义的常数,在OS_CFG.H文件中用定义常数语句"#define constant"定义的,因此可以有多达63个用户自己的任务。如果按笔者的建议,不使用最高4个和最低4个优先级,可以有56个用户自己的任务。

必须给每个任务赋以不同的优先级。优先级可以为0~OS_LOWEST_PRIO-2。优先级号越低,任务的优先级越高。μC/OS-II总是运行进入就绪态的优先级最高的任务。目前版本的μC/OS-II中,任务的优先级号就是任务编号(ID)。优先级号(任务的ID号)也被一些内核功能函数调用,如改变优先级函数OSTaskChangePrio(),以及任务删除函数OSTaskDel()。

为了使μC/OS-II能管理用户任务,必须在建立任务时,将任务的起始地址与其他参数一起传递给下面的2个函数之一:OSTaskCreate()或OSTaskCreateExt()。OSTaskCreateExt()是OSTaskCreate()的扩展,扩展了一些附加的功能。这2个函数的解释见第4章:"任务管理"。

3.02 任务状态

图F3.2是μC/OS-II控制下的任务状态转换图。在任一给定的时刻,任务的状态一定是在以下5种状态之一。

- 睡眠态(task dormant) 指任务驻留在程序空间(ROM或RAM),还没有交给μC/OS-II来管理。把任务交给μC/OS-II,是通过调用下述2个函数之一:OSTaskCreate()或OSTaskCreateExt()来实现的。这些调用只是用于告诉μC/OS-II,任务的起始地址在哪里;任务建立时,用户给任务赋予的优先级是多少;任务要使用多少栈空间等等。
- 就绪态 任务一旦建立,这个任务就进入了就绪态(task ready),准备运行。任务的建立可以是在多任务运行开始之前,也可以动态地由一个运行着的任务建立。如果多任务已经启动,且一个任务是被另一个任务建立的,而新建立的任务的优先级高于建立它的任务的优先级,则这个刚刚建立的任务将立即得到CPU的使用权。一个任务可以通过调用OSTaskDel()返回到睡眠态,或通过调用该函数让另一个任务进入睡眠态。

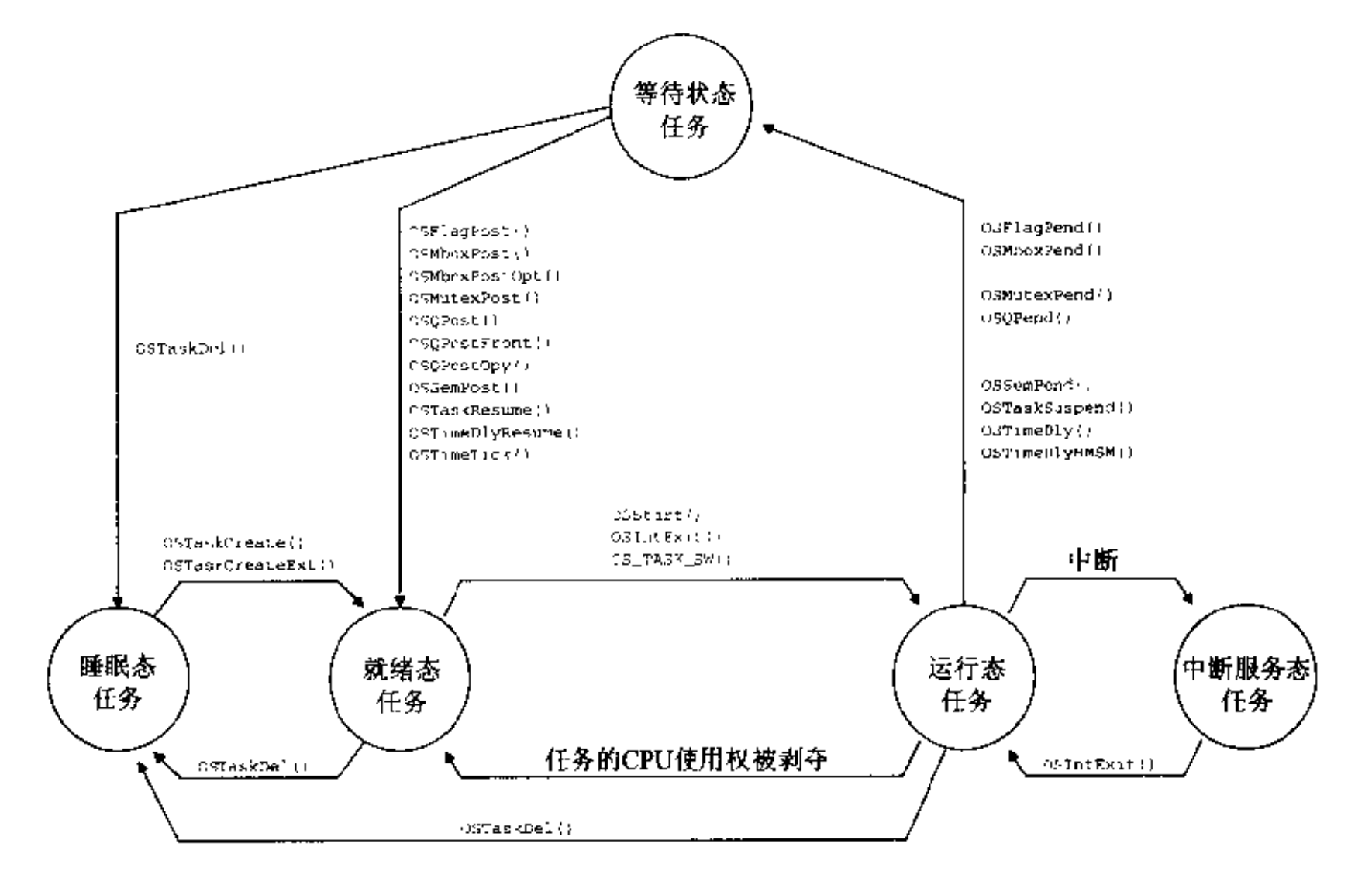


图 F3.2　任务的状态

- 运行态　调用 OSStart()可以启动多任务。OSStart()函数只能在启动时调用一次，该函数运行用户初始化代码中已经建立的、进入就绪态的优先级最高的任务。优先级最高的任务就这样进入了运行态(task running)。任何时刻只能有一个任务处于运行态。就绪的任务只有当所有优先级高于这个任务的任务都转为等待状态(task waiting)，或者是被删除了，才能进入运行态。
- 等待状态　正在运行的任务可以通过调用以下 2 个函数之一，将自身延迟一段时间。这 2 个函数是 OSTimeDly()或 OSTimeDlyHMSM()。这个任务于是进入等待状态(task waiting)，一直到函数中定义的延迟时间到。这 2 个函数会立即强制执行任务切换，让下一个优先级最高的、并进入了就绪态的任务运行。等待的时间过去以后，系统服务函数 OSTimeTick()使延迟了的任务进入就绪态(见 3.11 节："时钟节拍")。OSTimeTick()是 μC/OS-II 的一个内部函数，用户无需在应用程序代码中调用这个函数。

  正在运行的任务可能需要等待某一事件的发生，可以通过调用以下函数之一实现：OSFlagPend ( )，OSSemPend ( )，OSMutexPend ( )，OSMboxPend ( ) 或 OSQPend()。如果某事件并未发生，调用上述函数的任务就进入了等待状态(task waiting)，直到等待的事件发生了。当任务因等待事件被挂起(pend)时，下一个优先级最高的任务立即得到了 CPU 的使用权。当事件发生了或等待超时时，被挂起的任务就进入就绪态。事件发生的报告可能来自另一个任务，也可能来自中断服务子程序。
- 中断服务态　正在运行的任务是可以被中断的，除非该任务将中断关闭，或者 μC/OS-II将中断关闭。被中断了的任务于是进入了中断服务态(ISR running)。响应中断时，正在执行的任务被挂起，中断服务子程序控制了 CPU 的使用权。中断服务

子程序可能会报告一个或多个事件的发生，而使一个或多个任务进入就绪态。在这种情况下，从中断服务子程序返回之前，μC/OS-II 要判定，被中断的任务是否还是就绪态任务中优先级最高的。如果中断服务子程序使另一个优先级更高的任务进入了就绪态，则新进入就绪态的这个优先级更高的任务将得以运行；否则，原来被中断了的任务将继续运行。

当所有的任务都在等待事件发生或等待延迟时间的结束时，μC/OS-II 执行被称为空闲任务(idle task)的内部任务，即 OSTaskIdle()。

3.03 任务控制块(OS_TCB)

一旦任务建立，一个任务控制块 OS_TCB 就被赋值(程序清单 L3.4)。任务控制块是一个数据结构，当任务的 CPU 使用权被剥夺时，μC/OS-II 用它来保存该任务的状态。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_TCB 全部驻留在 RAM 中。读者将会注意到笔者在组织这个数据结构时，考虑到了各成员的逻辑分组。

程序清单 L3.4 μC/OS-II 的任务控制块

```
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 > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0) || (OS_SEM_EN > 0) || (OS_MUTEX_EN > 0)
    OS_EVENT        *OSTCBEventPtr;
# endif

# if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    void            *OSTCBMsg;
# endif

# if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
```

```
#if OS_TASK_DEL_EN > 0
    OS_FLAG_NODE    *OSTCBFlagNode;
#endif
    OS_FLAGS        OSTCBFlagsRdy;
#endif

    INT16U          OSTCBDly;
    INT8U           OSTCBStat;
    INT8U           OSTCBPrio;

    INT8U           OSTCBX;
    INT8U           OSTCBY;
    INT8U           OSTCBBitX;
    INT8U           OSTCBBitY;

#if OS_TASK_DEL_EN > 0
    BOOLEAN         OSTCBDelReq;
#endif
} OS_TCB;
```

.OSTCBStkPtr

指向当前任务堆栈栈顶的指针。μC/OS-II 允许每个任务有自己的堆栈，尤为重要的是，每个任务的堆栈的容量可以是任意的。有些商业内核要求所有任务堆栈的容量都一样，除非用户写一个复杂的接口函数来改变它。这种限制浪费了 RAM，当各任务需要的栈空间不同时，也得按任务中预期栈容量需求最多的来分配栈空间。OSTCBStkPtr 是 OS_TCB 数据结构中惟一一个能用汇编语言处置的变量(在任务切换的代码段 Context-switching 代码之中)。笔者决定把 .OSTCBStkPtr 放在数据结构的最前面，使得在汇编语言中处理这个变量时较为容易(在偏移量为 0 的位置)。

.OSTCBExtPtr

指向用户定义的任务控制块扩展。用户可以扩展任务控制块，而不必修改 μC/OS-II 的源代码。.OSTCBExtPtr 只在函数 OsTaskCreateExt()中使用，故使用时要将 OS_CFG.H 文件中的 OS_TASK_CREATE_EXT_EN 设为 1，以允许建立任务函数的扩展。然后，可以使用 .OSTCBExtPtr 指向一个数据结构，这个数据结构包含该任务的名字，以跟踪某个任务的执行时间，或者跟踪切换到某个任务的次数(见第 1 章的范例 3)。

注意：将这个扩展指针变量放在紧跟着堆栈指针的位置，目的是当需要在汇编语言中处理这个变量时，从数据结构的开始计算偏移量比较方便。

.OSTCBStkBottom

指向任务堆栈栈底的指针。如果微处理器的栈指针是递减的，即栈存储器从高地址向低地址方向分配，则 .OSTCBStkBottom 指向任务使用的栈空间的最低地址。类似地，如果微处

理器的堆栈是从低地址向高地址递增的，则 .OSTCBStkBottom 指向任务可以使用的堆栈空间的最高地址。函数 OSTaskStkChk() 要用到变量 .OSTCBStkBottom，在运行中检验栈空间的使用情况。用户可以用它来确定任务实际需要的栈空间。这个功能只有当用户在任务建立时允许使用 OSTaskCreateExt() 函数时，才能实现。这就要求将 OS_CFG.H 文件中的 OS_TASK_CREATE_EXT_EN 设为 1，以便允许该功能。

.OSTCBStkSize

存有栈中可容纳的指针元数目，而不是用字节(Byte)表示的栈容量总数。也就是说，如果栈中可以保存 1 000 个入口地址，每个地址宽度是 32 位的，则实际栈容量是 4 000 B。同样是 1 000 个入口地址，如果每个地址宽度是 16 位的，那么总栈容量只有 2 000 B。在函数 OSStakChk()中要用到 .OSTCBStkSize。同理，若使用该函数，则要将 OS_CFG.H 文件中的 OS_TASK_CREATE_EXT_EN 设为 1。

.OSTCBOpt

把"选择项"传给函数 OSTaskCreateExt()。只有当用户将 OS_CFG.H 文件中的 OS_TASK_CREATE_EXT_EN 设为 1 时，这个变量才有效。μC/OS-II 目前只支持 3 个选择项(见 μCOS_II.H)：OS_TASK_OPT_STK_CHK，OS_TASK_OPT_STK_CLR 和 OS_TASK_OPT_SAVE_FP。

OS_TASK_OPT_STK_CHK 用于告知 TaskCreateExt()，在任务建立时，任务栈检验功能得到了允许。μC/OS-II 不会自动进行堆栈检验，因为笔者不想过多地占用宝贵的 CPU 时间，只有当用户真正想用堆栈来检验功能时，才会进行。堆栈检验功能是由用户程序调用 OSTaskStkChk()函数启动的。

OS_TASK_OPT_STK_CLR 表示任务建立时，任务栈要清 0(μC/OS-II 向所有的栈空间写 0)。只有当用户需要有栈检验功能时，才需将栈清 0。若不定义 OS_TASK_OPT_STK_CLR，而后又建立和删除了任务，栈检验功能报告的栈使用情况将是错误的。任务一旦建立，就绝不会被删除，而初始化时，已将 RAM 清过 0，则 OS_TASK_OPT_STK_CLR 就无须再定义，这可以节省程序执行时间。传递了 OS_TASK_OPT_STK_CLR，将增加TaskCreateExt()函数的执行时间，因为要将栈空间清 0，栈容量越大，清 0 的时间越长。这里再次强调，堆栈检验不是自动允许的，需在用户应用程序代码中写上相应的调用。

OS_TASK_OPT_SAVE_FP 通知 TaskCreateExt()，任务要做浮点数运算。如果微处理器有硬件的浮点协处理器，则建立的任务在做任务调度切换时，浮点寄存器的内容须保存。

.OSTCBId

用于存储任务的识别码(ID)。这个变量现在没有使用，保留给将来扩展用。

.OSTCBNext 和 .OSTCBPrev

用于任务控制块 OS_TCB 双向链表的前/后链接，该链表在时钟节拍函数 OSTimeTick() 中使用。OSTimeTick()使用这个前/后链接的双向链表，来刷新各任务的任务延迟变量 .OSTCBDly。每个任务的任务控制块 OS_TCB 在任务建立时，链接到链表中，在任务删除

时，从链表中删除。双重链接的链表使任一成员都能被快速插入或删除。

.OSTCBEventPtr

指向事件控制块的指针。后面章节中会有所描述(见第 6 章："事件控制块")。

.OSTCBMsg

指向传递给任务的消息的指针。其用法将在后面章节中提到(见第 10 章和第 11 章)。

.OSTCBFlagNode

指向事件标志节点(flag node)的指针(参见第 9 章："事件标志组管理")，只有当 OSTaskDel()函数使用 .OSTCBFlagNode 删除等待事件标志组的任务时，才会用到；只有当 OS_CFG.H 文件中的 OS_FLAG_EN 置为 1 时，这个变元才会出现在 OS_TCB 中。

.OSTCBFlagRdy

当任务等待事件标志组时，.OSTCBFlagRdy 是使任务进入就绪态的事件标志(见第 9 章："事件标志组管理")。只有当 OS_CFG.H 文件中的 OS_FLAG_EN 置为 1 时，这个变元才会出现在 OS_TCB 中。

.OSTCBDly

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，须用到这个变量。这种等待是有超时限制的。在这种情况下，这个变量保存的值是任务允许等待事件发生的最多时钟节拍数。如果这个变量为 0，则表示任务不延时，或者表示等待事件发生的时间没有限制。

.OSTCBStat

任务的状态字。当 .OSTCBStat 等于 OS_STAT_READY 时，任务进入就绪态。μC/OS-II也可以给 .OSTCBStat 赋其他的值，在文件 uCOS_II.H 中有关于这个值的描述(见 OS_STAT_???)。

.OSTCBPrio

任务的优先级。高优先级任务的 .OSTCBPrio 值小。也就是说，这个值越小，任务的优先级越高。

.OSTCBX，.OSTCBY，.OSTCBBitX 及.OSTCBBitY

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(避免在运行中计算这些值)。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。这些值的算法见程序清单 L3.5。

程序清单 L3.5 任务控制块 OS_TCB 中几个成员的算法

```
OSTCBY          = priority >> 3;
OSTCBBitY       = OSMapTbl[priority >> 3];
OSTCBX          = priority & 0x07;
OSTCBBitX       = OSMapTbl[priority & 0x07];
```

.OSTCBDelReq

一个布尔量，用于表示该任务是否须删除，其用法将在后面的章节中描述（见第4章："任务管理"）。只有当 OS_CFG.H 文件中的 OS_FLAG_EN 置为1时，这个变元才会出现在 OS_TCB中。

读者可能已经注意到，OS_TCB 数据结构中的一些变元是用条件编译语句定义的。如果不需要使用 μC/OS-II 的全部功能，那么这种定义方法可以使用户减缩 μC/OS-II 对 RAM 的占用量。

应用程序中可以有的最多任务数（OS_MAX_TASKS）已经在文件 OS_CFG.H 中定义了。这个最多任务数也决定了分配给用户程序的任务控制块 OS_TCB 的数目。将 OS_MAX_TASKS的数目设置为用户应用程序实际需要的任务数，可减小 RAM 的需求量。所有的任务控制块 OS_TCB 都是放在任务控制块列表数组 OSTCBTbl[]中的。请注意，μC/OS-II分配给系统任务 OS_N_SYS_TASKS 若干个额外的任务控制块，供其内部使用（见文件 μCOS_II.H）。目前，一个用于空闲任务，另一个用于统计任务（如果 OS_TASK_STAT_EN设为1）。在 μC/OS-II 初始化时，如图 F3.3 所示，所有任务控制块 OS_TCB 都被链接成单向空任务链表。任务一旦建立，空任务控制块指针 OSTCBFreeList 指向的任务控制块便赋给了该任务，然后 OSTCBFreeList 的值调整为指向链表中下一个空的任务控制块。一旦任务被删除，任务控制块就还给空任务链表。

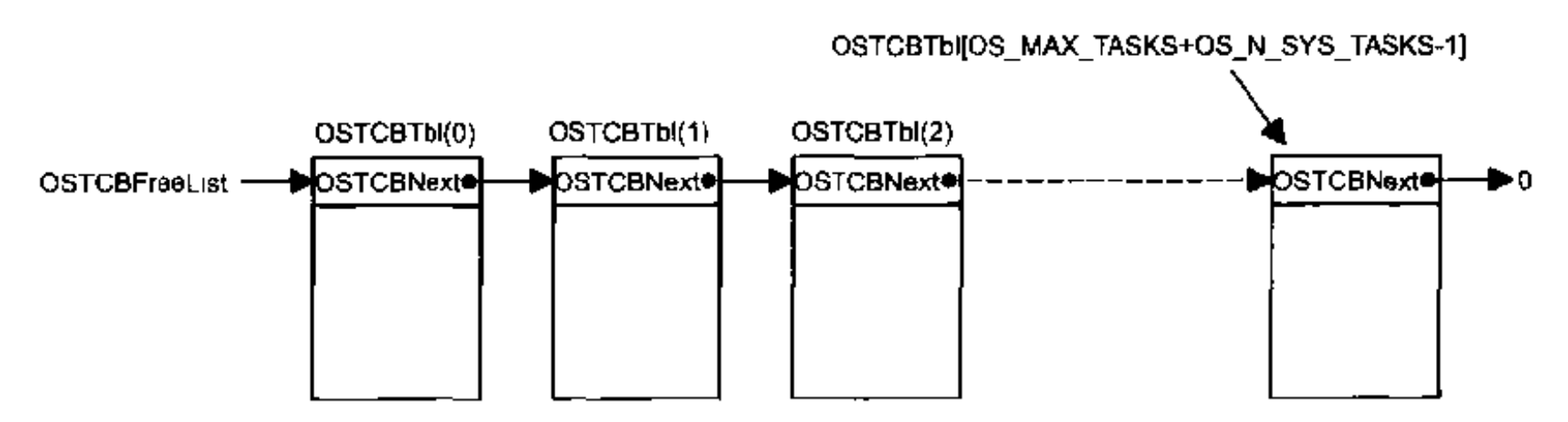


图 F3.3 空任务控制块列表（free OS_TCBs）

任务建立时，函数 OS_TCBInit()初始化任务控制块 OS_TCB（见程序清单 L3.6）。函数 OSTaskCreate()或函数 OSTaskCreateExt()调用任务控制块初始化函数 OS_TCBInit()（见第4章："任务管理"）。OS_TCBInit()接受7个参数：

prio　任务的优先级。

ptos　OSTaskInit()建立栈结构以后，ptos是指向栈顶的指针（见第13章："移植 μC/OS-II"）且保存在 OS_TCB 的 .OSTCBStkPtr 中。

pbos　指向栈底的指针，保存在 OS_TCB 的 .OSTCBStkBottom 变元中。

id　　任务标志符，保存在 .OSTCBId 中。

stk_size　　堆栈的容量，保存在 OS_TCB 的 .OSTCBStkSize 中。

pext　　OS_TCB 中的扩展指针 .OSTCBExtPtr 的值。

opt　　OS_TCB 的选择项，保存在 .OSTCBOpt 中。

程序清单 L3.6　任务控制块初始化函数 OS_TCBInit()

```
INT8U OS_TCBInit (INT8U  prio,      OS_STK *ptos,   OS_STK *pbos, INT16U id,
                  INT32U stk_size, void   *pext,   INT16U  opt)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif
    OS_TCB     *ptcb;


    OS_ENTER_CRITICAL();
    ptcb = OSTCBFreeList;                                                    (1)
    if (ptcb != (OS_TCB *)0) {                                               (2)
        OSTCBFreeList           = ptcb->OSTCBNext;
        OS_EXIT_CRITICAL();
        ptcb->OSTCBStkPtr       = ptos;                                      (3)
        ptcb->OSTCBPrio         = (INT8U)prio;
        ptcb->OSTCBStat         = OS_STAT_RDY;
        ptcb->OSTCBDly          = 0;
# if OS_TASK_CREATE_EXT_EN > 0
        ptcb->OSTCBExtPtr       = pext;                                      (4)
        ptcb->OSTCBStkSize      = stk_size;
        ptcb->OSTCBStkBottom    = pbos;
        ptcb->OSTCBOpt          = opt;
        ptcb->OSTCBId           = id;
# else
        pext                    = pext;
        stk_size                = stk_size;
        pbos                    = pbos;
        opt                     = opt;
        id                      = id;
# endif

# if OS_TASK_DEL_EN > 0
        ptcb->OSTCBDelReq       = OS_NO_ERR;                                 (5)
# endif
```

```
        ptcb->OSTCBY            = prio >> 3;                                    (6)
        ptcb->OSTCBBitY         = OSMapTbl[ptcb->OSTCBY];
        ptcb->OSTCBX            = prio & 0x07;
        ptcb->OSTCBBitX         = OSMapTbl[ptcb->OSTCBX];

#if OS_EVENT_EN > 0
        ptcb->OSTCBEventPtr     = (OS_EVENT *)0;                                (7)
#endif

#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0) && (OS_TASK_DEL_EN > 0)
        ptcb->OSTCBFlagNode     = (OS_FLAG_NODE *)0;                            (8)
#endif

#if OS_MBOX_EN || (OS_Q_EN && (OS_MAX_QS >= 2))
        ptcb->OSTCBMsg          = (void *)0;
#endif

#if OS_VERSION >= 204
        OSTCBInitHook(ptcb);                                                    (9)
#endif

        OSTaskCreateHook(ptcb);                                                 (10)

        OS_ENTER_CRITICAL();                                                    (11)
        OSTCBPrioTbl[prio]      = ptcb;                                         (12)
        ptcb->OSTCBNext         = OSTCBList;
        ptcb->OSTCBPrev         = (OS_TCB *)0;
        if (OSTCBList != (OS_TCB *)0) {
            OSTCBList->OSTCBPrev = ptcb;
        }
        OSTCBList               = ptcb;
        OSRdyGrp               |= ptcb->OSTCBBitY;                              (13)
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        OS_EXIT_CRITICAL();
        return (OS_NO_ERR);                                                     (14)
    }
    OS_EXIT_CRITICAL();
    return (OS_NO_MORE_TCB);
}
```

L3.6(1) OS_TCBInit()首先试图从OS_TCB缓冲池中得到一个任务控制块OS_TCB

L3.6(2)

L3.6(3)　如果缓冲池中有空余的 OS_TCB，这个 OS_TCB 就被初始化了。

注意：一旦分配到 OS_TCB，OS_TCBInit()，就可以重新开中断。因为此时要建立的任务已经占有了这个任务控制块，它不可能被同时建立的另一个任务夺取。OS_TCBInit()可以在中断开着的情况下，初始化 OS_TCB 中的其余变元。

L3.6(4)　如果用户允许了函数 OSTaskCreateExt()的代码生成(OS_CFG.H 文件中的 OS_TASK_CREATE_EN被置 1)，额外的变元就插入到 OS_TCB 中了。

L3.6(5)　OS_TCB 中是否出现 .OSTCBDelReq，取决于 OS_TASK_DEL_EN 是否被置为 1(见 OS_CFG.H)。换言之，如果不打算删除任务，可以从每一个 OS_TCB 中节省出一个布尔量空间。

L3.6(6)　为在任务调度时节省时间，OSTCBInit()对一些参数提前做了运算。笔者宁愿以存储空间换取执行时间。

L3.6(7)　如果不打算在应用程序中使用任何信号量、互斥型信号量 mutex、消息邮箱以及消息队列，OS_TCB 中的 .OSTCBEventPtr 变元就不会出现。

L3.6(8)　如果事件标志的使用得到允许(将 OS_CFG.H 文件中的 OS_FLAG_EN 置为 1)，则指向事件标志节点的指针被初始化为空指针。因为任务并没有等待事件标志，只是刚刚建立。

L3.6(9)　在 μC/OS-II V2.4 中，增加了对函数 OSTCBInitHook() 的调用。这个函数可以在处理器的移植文件中定义。该函数允许对 OS_TCB 加以扩展。例如，可以初始化和保存浮点寄存器、MMU 寄存器及与任务相关的内容；然而，这些额外的信息通常是保存在为应用程序分配的存储空间中的。

注意：当 OS_TCBInit()调用 OSTCBInitHook()时，中断是开着的。

L3.6(10)　OS_TCBInit() 然后调用 OSTaskCreateHook()。OSTaskCreateHook()是用户定义的函数。该函数使用户能够扩展函数 OSTaskCreate() 以及 OSTaskCreateExt() 的功能。当 OS_CPU_HOOKS_EN 置为 1 时，OSTaskCreateHook()可以在 OS_CPU.C 中定义；若 OS_CPU_HOOKS_EN 为 0，则可在任何其他地方定义。

注意：当 OS_TCBInit()调用 OSTCBCreateHook()时，中断是开着的。

读者可能已注意到，本来可以设计成只调用 2 个接口函数中的一个，OS-TCBInitExt()或 OSTaskCreateExt()。分成 2 个函数，可以使用户紧缩那些与 OS_TCB 相关的项，并通过 OSTCBInitHook()捆绑在 OS_TCB 上，而其他与任务初始化相关的项放在 OSTaskCreateHook()中。

L3.6(11)

L3.6(12)　当 OS_TCBInit()需要把 OS_TCB 插入到已经建立任务的双向链表中时，先要关中断。这个链表开始于 OSTCBList，新任务的 OS_TCB 总是插在表的开头。

L3.6(13)

L3.6(14)　最后，让该任务进入就绪态。OS_TCBInit()返回到调用函数[OSTaskCreate()或 OSTaskCreateExt()]。返回值表示已经分配到任务控制块，并且已经初始化了。

3.04 就绪表

每个任务被赋予不同的优先级等级，从0级到最低优先级（OS_LOWEST_PRIO，包括0和OS_LOWEST_PRIO在内（见文件OS_CFG.H）。当μC/OS-II初始化时，最低优先级OS_LOWEST_PRIO总是被赋给空闲任务idle task。

注意：最多任务数目（OS_MAX_TASKS和最低优先级数是没有关系的。应用程序可以只有10个任务，而仍然可以有32个优先级的级别（如果将最低优先级数设为31的话）。

每个就绪的任务都放在就绪表（ready list）中，就绪表中有2个变量，OSRdyGrp和OSRdyTbl[]。在OSRdyGrp中，任务按优先级分组，8个任务为一组。OSRdyGrp中的每一位表示8组任务中每一组是否有进入就绪态的任务。任务进入就绪态时，就绪表（OSRdyTbl[]中的相应元素的相应位也置为1。OSRdyGrp和OSRdyTbl[]之间的关系见图F3.4，是按以下规则给出的：

当OSRdyTbl[0]中的任何一位是1时，OSRdyGrp的第0位置1；
当OSRdyTbl[1]中的任何一位是1时，OSRdyGrp的第1位置1；
当OSRdyTbl[2]中的任何一位是1时，OSRdyGrp的第2位置1；
当OSRdyTbl[3]中的任何一位是1时，OSRdyGrp的第3位置1；
当OSRdyTbl[4]中的任何一位是1时，OSRdyGrp的第4位置1；
当OSRdyTbl[5]中的任何一位是1时，OSRdyGrp的第5位置1；
当OSRdyTbl[6]中的任何一位是1时，OSRdyGrp的第6位置1；
当OSRdyTbl[7]中的任何一位是1时，OSRdyGrp的第7位置1。

就绪表OSRdyTbl[]数组的大小取决于OS_LOWEST_PRIO（见文件OS_CFG.H）。当应用程序中任务数目比较少时，这种安排可以减小OS_LOWEST_PRIO的值，可以降低μC/OS-II对RAM（数据空间）的需求量。

为确定下一次该哪个优先级的任务运行了，μC/OS-II中的调度器总是将最低优先级的任务在就绪表中相应字节的相应位置1。

程序清单L3.7中的代码用于将任务放入就绪表。prio是任务的优先级。

程序清单L3.7 使任务进入就绪态

```
OSRdyGrp               |= OSMapTbl[prio >> 3];
OSRdyTbl[prio >> 3]    |= OSMapTbl[prio & 0x07];
```

从图F3.4可以看出，任务优先级的低3位用于确定任务在总就绪表OSRdyTbl[]中的所在位。接下去的3位用于确定是在OSRdyTbl[]数组的第几个元素。OSMapTbl[]是在ROM中的（见文件OS_CORE.C）屏蔽字，用于限制OSRdyTbl[]数组的元素下标为0～7，见表T3.2。

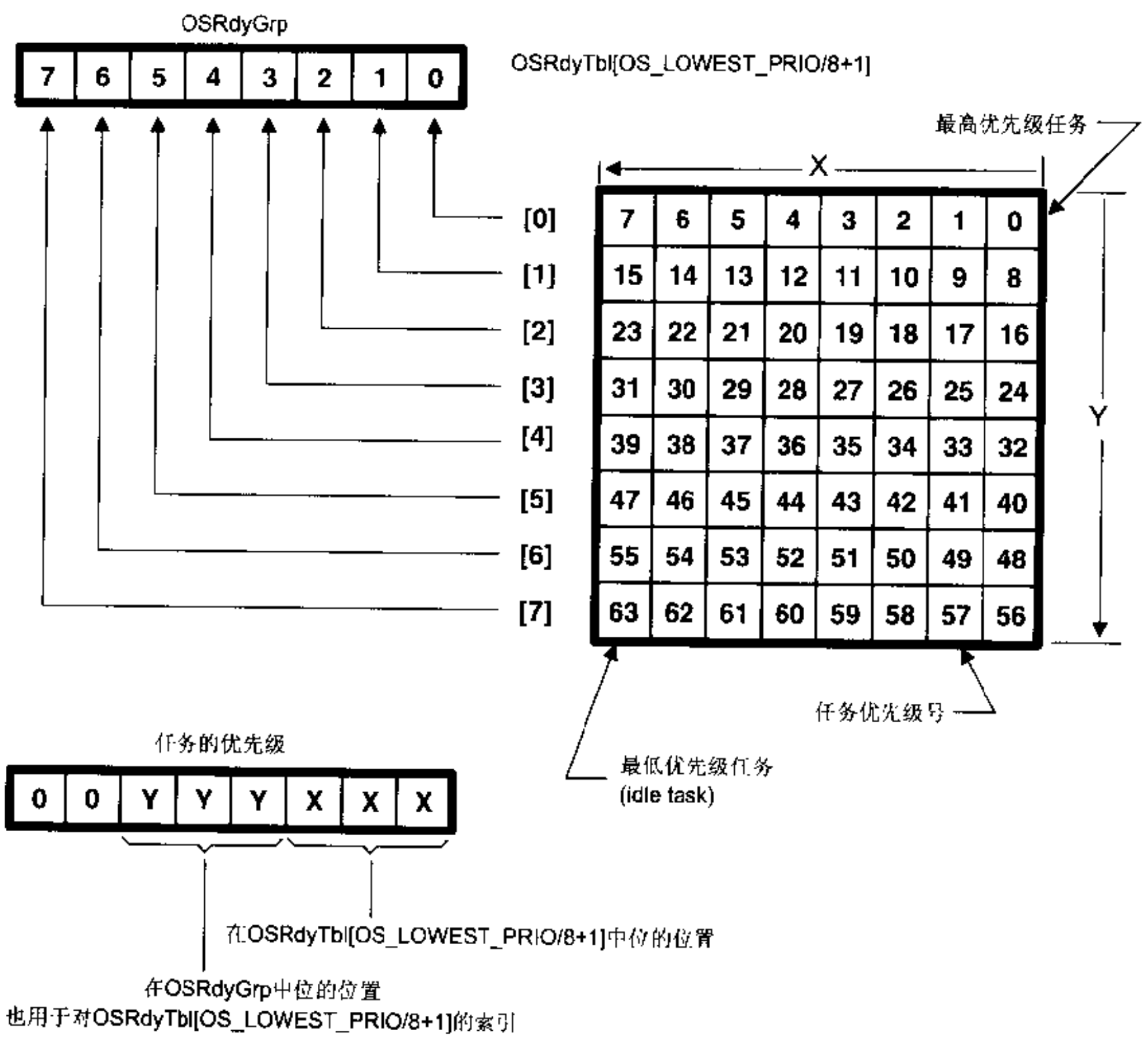


图 F3.4　μC/OS　II 的任务就绪表

表 T3.2　OSMapTbl[ ]的值

下　标	位掩码(二进制)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

如果要使一个任务脱离就绪态，则用程序清单 L3.8 中的代码做求反处理。

程序清单 L3.8 使任务脱离就绪态

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

以上代码将就绪任务表数组 OSRdyTbl[]中相应元素的相应位清 0；而对于 OSRdyGrp，只有当脱离就绪态的任务所在任务组中，全组任务一个都没有进入就绪态时，才将相应位清 0。也就是说，OSRdyTbl[prio≫3]所有的位都是 0 时，OSRdyGrp 的相应位才清 0。为了找到进入就绪态的优先级最高的任务，并不需要从 OSRdyTbl[0]开始扫描整个就绪任务表，只须查另外一张表，即优先级判定表 OSUnMapTbl[256]（见文件 OS_CORE.C）。OSRdyTbl[]中每个字节的 8 位代表这一组的 8 个任务哪些进入就绪态了，低位的优先级高于高位的优先级。利用这个字节为下标来查 OSUnMapTbl 这张表，返回的字节就是该组任务中就绪态任务中优先级最高的任务所在的位置。这个返回值为 0～7。确定进入就绪态的优先级最高的任务是用以下代码完成的，如程序清单 L3.9 所列。

程序清单 L3.9 找出进入就绪态的优先级最高的任务

```
y    = OSUnMapTbl[OSRdyGrp];
x    = OSUnMapTbl[OSRdyTbl[y]];
prio = y << 3 + x;
```

作为一个例子，如图 F3.5 所示，若 OSRdyGrp 的值为二进制 01101000，即 0x68，则查 OSUnMapTbl[OSRdyGrp]得到的值是 3，它相应于 OSRdyGrp 中的第 3 位 bit3，这里假设最右边的一位是第 0 位 bit0。类似地，如果 OSRdyTbl[3]的值是二进制 11100100，即 0xE4，则 OSUnMapTbl[OSRdyTbl[3]]的值是 2，即第 2 位 bit2。于是任务的优先级 prio 就等于 26，即(3×8－2)。利用这个优先级的值，查任务控制块优先级表 OSTCBPrioTbl[]，得到指向相应任务的任务控制块 OS_TCB。

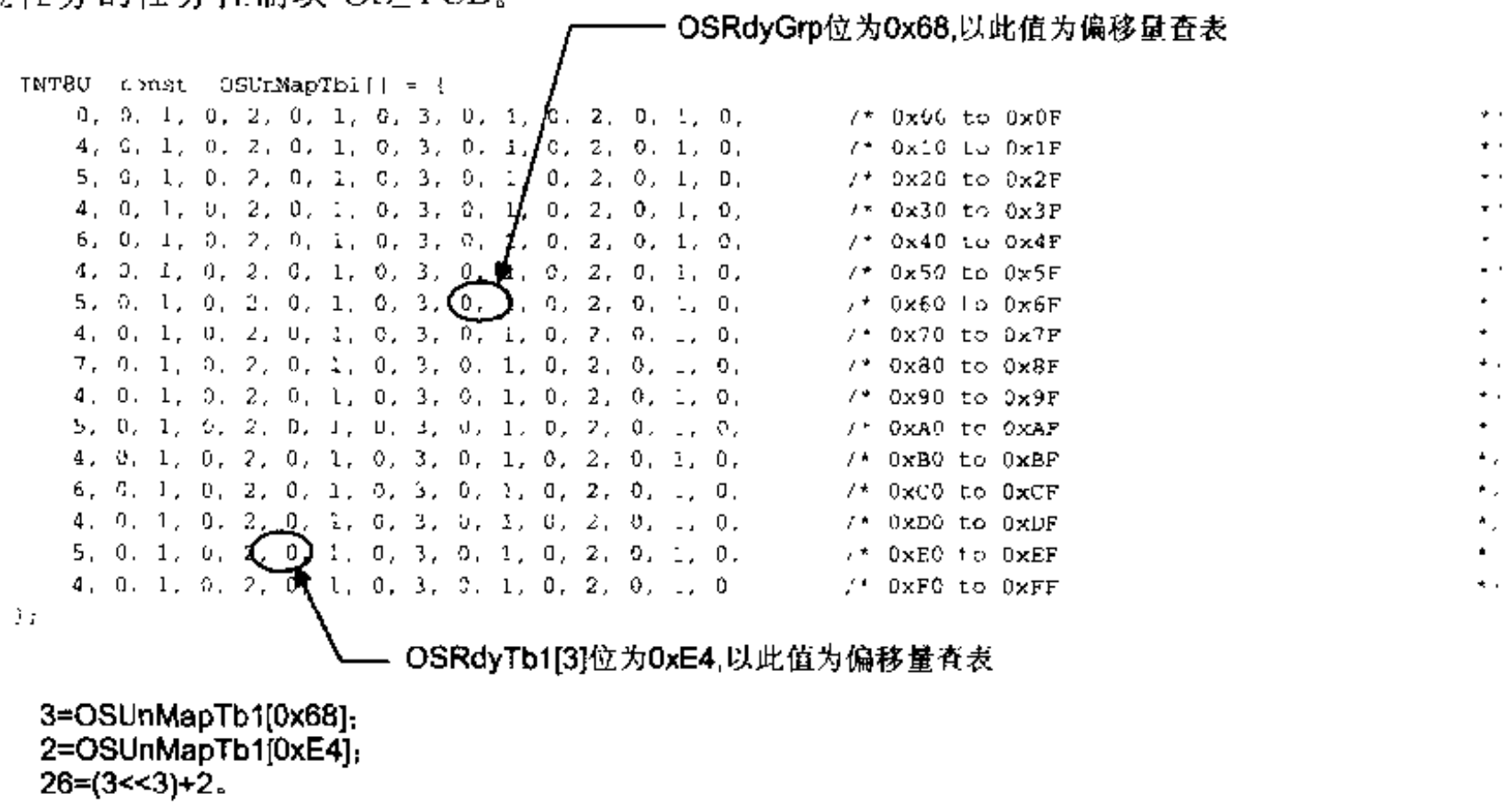


图 F3.5 找出优先级最高的就绪态任务

3.05　任务调度

μC/OS－Ⅱ总是运行进入就绪态任务中优先级最高的任务。确定哪个任务优先级最高，下面该哪个任务运行了，这一工作是由调度器(scheduler)完成的。任务级的调度是由函数OSSched()完成的。中断级的调度是由另一个函数OSIntExt()完成的。这个函数将在以后描述。OSSched()的代码如程序清单L3.10所列。μC/OS－Ⅱ任务调度(task scheduling)的执行时间是常数，与应用程序建立了多少个任务没有关系。

程序清单 L3.10　任务调度器(task scheduler)

```
void  OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      y;

    OS_ENTER_CRITICAL();
    if ((OSIntNesting == 0) && (OSLockNesting == 0)) {                          (1)
        y             = OSUnMapTbl[OSRdyGrp];                                   (2)
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {                                       (3)
            OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];                        (4)
            OSCtxSwCtr ++ ;                                                     (5)
            OS_TASK_SW();                                                       (6)
        }
    }
    OS_EXIT_CRITICAL();
}
```

L3.10(1)　如果调用来自中断服务子程序(OSIntNesting > 0)，或者由于至少调用了一次给任务调度上锁函数OSSchedLock()(OSLockNesting > 0)，任务调度函数OSSched()将退出，不做任务调度。

L3.10(2)　如果OS_Sched()不是在中断服务子程序中调用的，且调度是允许的，则任务调度函数将找出进入就绪态且优先级最高的任务。进入就绪态的任务在就绪任务表OSRdyTbl[]中相应的位置位。

L3.10(3)　找到优先级最高的任务之后，OSSched()检验这个优先级最高的任务是否是当前正在运行的任务，以避免不必要的任务调度，因为调度是很花时间的。注意，在μC/OS(V1.xx)中，是先得到OSTCBHighRdy(一个指针)，然后和

OSTCBCur（另一个指针）做比较。因为这个比较是2个指针型变量的比较，在8位和一些16位微处理器中，这种比较相对较慢；而在μC/OS-II中，是2个整数的比较。并且，除非实际需要做任务切换，在查任务控制块优先级表OSTCBPrioTbl[]时，不需要用指针变量来查OSTCBHighRdy。综合这2项改进，即用整数比较代替指针的比较，当需要任务切换时再查表，μC/OS-II比μC/OS在8位和一些16位微处理器上要更快一些。

L3.10(4) 为了实现任务切换，OSTCBHighRdy必须指向优先级最高的任务控制块OS_TCB。这是通过将以OSPrioHighRdy为下标的OSTCBPrioTbl[]数组中的元素赋给OSTCBHighRdy实现的。

L3.10(5) 接着，统计计数器OSCtxSwCtr（一个32位的变量）加1，以跟踪任务切换次数。这个计数器的使用，目的只是让用户知道每秒做了多少次任务切换。当然，用户还需每秒将OSCtxSwCtr的值存入另一个变量（例如，OSCtxPerSec），然后将OSCtxSwCtr清0。

L3.10(6) 最后，使用宏调用OS_TASK_SW()，完成实际上的任务切换。

任务切换很简单，由以下2步完成：将被挂起任务的处理器寄存器推入堆栈；然后将较高优先级任务的寄存器值从栈中恢复到寄存器中。在μC/OS-II中，就绪任务的栈结构总是看起来跟刚刚发生过中断一样，所有处理器的寄存器都保存在栈中。换句话说，μC/OS-II运行就绪态的任务所要做的一切，只是恢复所有的CPU寄存器并运行中断返回指令。为了做任务切换，运行OS_TASK_SW()，人为模仿了一次中断。多数微处理器由软中断指令或者指令陷阱TRAP来实现上述操作。中断服务子程序或陷阱处理（trap handler），也称做事故处理（exception handler），必须给汇编语言函数OSCtxSw()提供中断向量。OSCtxSw()除需OS_TCBHighRdy指向即将被挂起的任务，还需让当前任务控制块OSTCBCur指向即将被挂起的任务，参见第13章："移植μC/OS-II"。现在，只需知道OS_TASK_SW()挂起了正在执行的任务，而让CPU执行更重要的任务。

OS_Sched()的所有代码都属临界段代码。在寻找进入就绪态的优先级最高的任务过程中，为防止中断服务子程序把一个或几个任务的就绪位置位，中断是关掉的。为缩短切换时间，OS_Sched()全部代码都可以用汇编语言编写。为增加可读性、可移植性及将汇编语言代码最少化，OSSched()是用C语言编写的。

3.06 任务级的任务切换，OS_TASK_SW()

如前一节所讨论的，调度器确定更重要的任务该运行了，于是调用OS_TASK_SW()做任务切换。任务的context，其实就是CPU中的全部寄存器内容，context-switch就是任务切换。任务切换代码须恢复该任务在CPU使用权被剥夺时保存下来的全部寄存器的值，以便让这个任务能继续运行。

OS_TASK_SW()是宏调用，通常含有微处理器的软中断指令，因为μC/OS-II假定任务切换是靠中断级代码完成的。μC/OS-II需要的是一条处理器指令，其行为就像是硬件中断（所以称为软中断）。μC/OS-II使用宏定义，将与实际处理器相关的软件中断机制封装起来，

使之可以在多种处理器开发平台上移植。第 13 章："移植 μC/OS - II"讨论如何实现 OS_TASK_SW()。

图 F3.6 示意 μC/OS - II 在调用 OS_TASK_SW()之前一些变量和数据结构的状况。为便于讨论，构造了一个假想的 CPU，该 CPU 有 7 个寄存器：

1 个堆栈指针(SP)；

1 个程序计数器(PC)；

1 个处理器状态寄存器(PSW)；

4 个通用寄存器(R1，R2，R3 及 R4)。

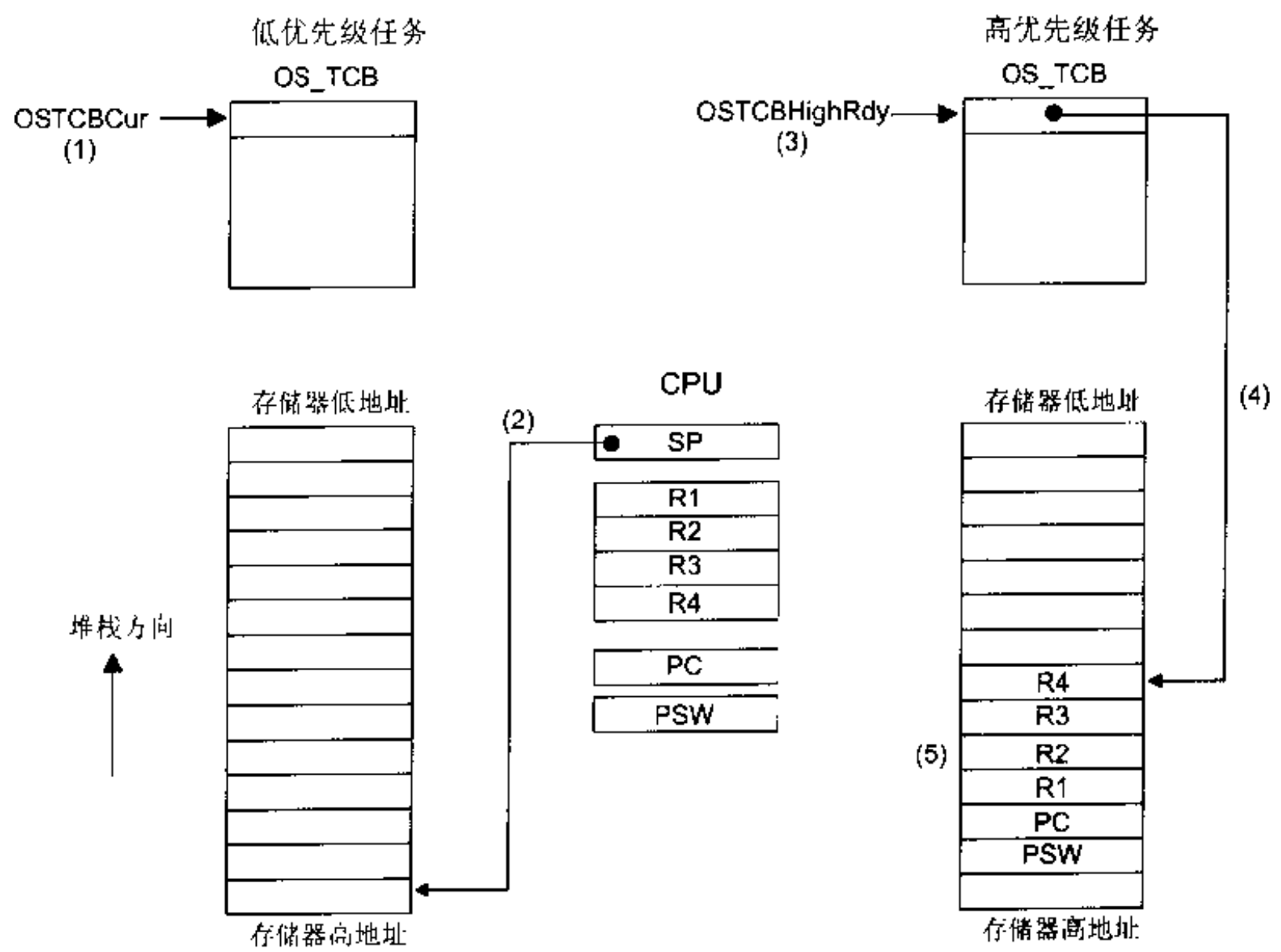


图 F3.6　μC/OS - II 调用 OS_TASK_SW()前的数据结构

F3.6(1)　OSTCBCur 指向即将被挂起的任务(低优先级的任务)。

F3.6(2)　CPU 的堆栈指针(SP 寄存器)指向即将被挂起任务的栈顶。

F3.6(3)　OSTCBHighRdy 指向任务切换后运行的任务的任务控制块 OS_TCB。

F3.6(4)　OS_TCB 中的 .OSTCBStkPtr 指向即将运行的任务的栈顶。

F3.6(5)　即将运行的任务的堆栈中存有准备装回到 CPU 寄存器中的一些值，不久就会看到，这些值是在上一次任务切换时保存下来的。当前，假设这些值是正确的。

图 F3.7 示意调用 OS_TASK_SW()以及保存被挂起任务的 CPU 寄存器之后，一些变量和数据结构的状况。

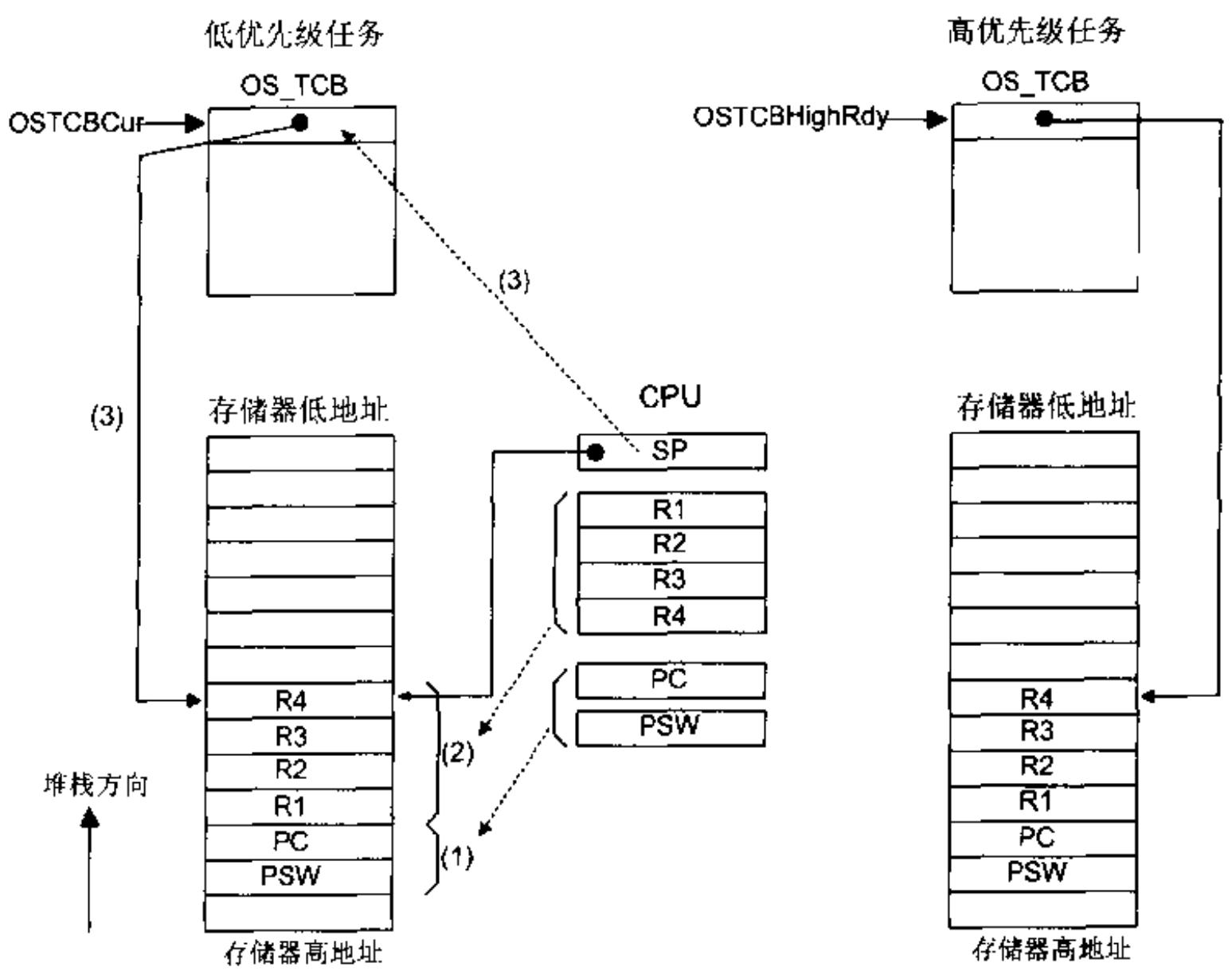


图 F3.7 保存当前任务 CPU 寄存器的值

F3.7(1) 调用含有软中断指令的函数 OS_TASK_SW()，强制处理器保存 PSW 和 PC 的当前值。

F3.7(2) 软中断的执行从保存通用寄存器开始，顺序是 R1，R2，R3，R4。

F3.7(3) 然后堆栈指针寄存器被保存在当前任务的 OS_TCB 中。此时，CPU 的 SP 寄存器和 OSTCBCur->OSTCBStkPtr 都指向当前任务堆栈的同一位置。

图 F3.8 示意执行任务切换最后一段代码之后变量和数据结构的状况。

F3.8(1) 由于新的当前任务是将要重新开始运行的任务，任务切换代码将 OSTCBHighRdy复制到 OSTCBCur。

F3.8(2) 从 OS_TCB 中找出将要重新开始运行的任务的堆栈指针（在 OSTCBHighRdy->OSTCBStkPtr 中），装入 CPU 的 SP 寄存器中。此时，SP 寄存器指向堆栈中存有寄存器 R4 值的位置。

F3.8(3) 按相反的方向从堆栈中弹出通用寄存器。

F3.8(4) 通过执行中断返回指令，PC 和 PSW 的值装回到 CPU 中。此时，由于程序计数器 PC 的值变了，重新开始运行的任务代码从 PC 指向的那一点开始运行，于是切换到新任务代码的过程完成了。

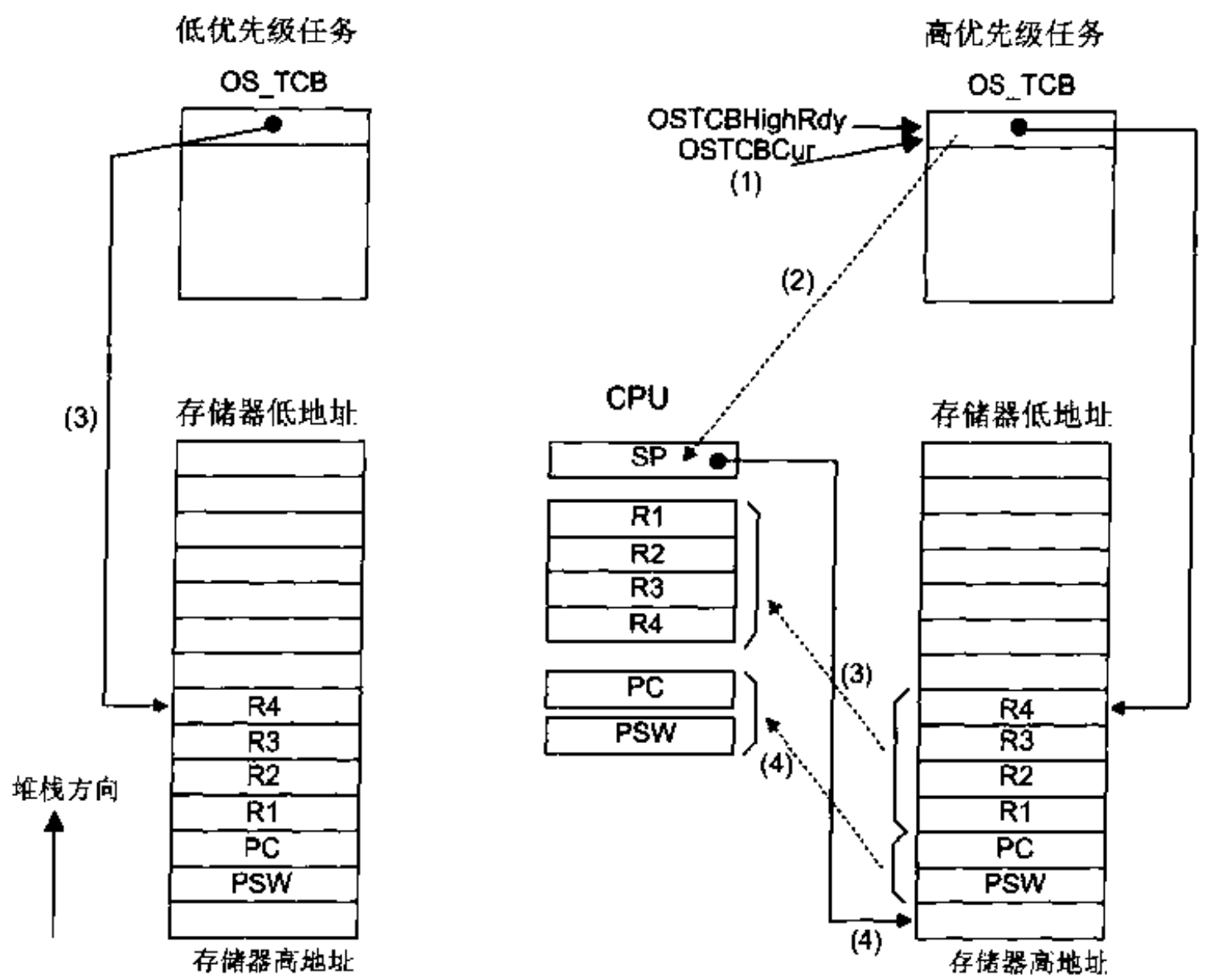


图 F3.8　重新装入要运行的任务

任务切换的示意性代码如程序清单 L3.11 所列。OSCtxSw()通常是用汇编语言写的，因为 C 编译器不能从 C 语言中直接处理 CPU 寄存器。第 14 章："μC/OS－II 在 80x86 上的移植"将讨论如何实现 OSCtxSw()以及其他一些 μC/OS－II 的函数。用的是一个真实的 CPU，Intel 80x86。

程序清单 L3.11　任务切换的示意性代码

```
void  OSCtxSw (void)
{
    将 R1,R2,R3 及 R4 推入当前堆栈；                          见 F3.6(2)
    OSTCBCur->OSTCBStkPtr = SP;                              见 F3.6(3)
    OSTCBCur              = OSTCBHighRdy;                    见 F3.7(1)
    SP                    = OSTCBHighRdy->OSTCBStkPtr;       见 F3.7(2)
    将 R4,R3,R2 及 R1 从新堆栈中弹出；                        见 F3.7(3)
    执行中断返回指令；                                        见 F3.7(4)
}
```

3.07 给调度器上锁和开锁

给调度器上锁函数 OSSchedlock()（程序清单 L3.12）用于禁止任务调度，直到任务完成后，调用给调度器开锁函数 OSSchedUnlock()为止（程序清单 L3.13）。调用 OSSchedlock()的任务，保持对 CPU 的使用权，尽管有个优先级更高的任务进入了就绪态。此时，中断仍然是可以识别的，中断服务也能得到（设中断是开着的）。OSSchedlock()和 OSSchedUnlock()必须成对地使用。变量 OSLockNesting 跟踪 OSSchedLock()函数被调用的次数，以允许嵌套的函数包含临界段代码，这段代码其他任务不得干预。μC/OS-II 允许嵌套深度达 255 层。当 OSLockNesting=0 时，调度重新得到允许。函数 OSSchedLock()和 OSSchedUnlock()的使用要非常谨慎，因为它们影响 μC/OS-II 对任务的正常管理。

程序清单 L3.12　给调度器上锁

```
void  OSSchedLock (void)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif


    if (OSRunning == TRUE) {                                        (1)
        OS_ENTER_CRITICAL();
        if (OSLockNesting < 255) {                                  (2)
            OSLockNesting++;
        }
        OS_EXIT_CRITICAL();
    }
}
```

L3.12(1)　如果多任务已经启动了（已经调用过 OSStart()），则给调度器上锁才有意义。

L3.12(2)　在给嵌套层数 OSLockNesting 加 1 之前，须确认嵌套层数没有超界。

调用 OSSchedLock()之后，用户应用程序不得调用可能会使当前任务挂起的系统功能函数。也就是说，用户应用程序不得调用 OSFlagPend()，OSMboxPend()，OSMutexPend()，OSQPend()，OSSemPend()，OSTaskSuspend(OS_PRIO_SELF)，OSTimeDly()或者 OSTimeDlyHMSM()，直到 OSLockNesting 回 0 为止。因为 OSSchedLock()给调度器上了锁，不让其他任务运行，用户锁住了系统。

程序清单 L3.13　给调度器开锁

```
void  OSSchedUnlock (void)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif


    if (OSRunning == TRUE) {                                          (1)
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {                                      (2)
            OSLockNesting--;                                          (3)
            if ((OSLockNesting == 0) && (OSIntNesting == 0)) {        (4)
                OS_EXIT_CRITICAL();
                OS_Sched();                                           (5)
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}
```

L3.13(1)　如果多任务已经启动了(已经调用过 OSStart())，则给调度器开锁才有意义。

L3.13(2)　确认 OSLockNesting 还没有回 0。若已回 0，说明调用OSSchedUnlock()的次数太多了。换言之，调用 OSSchedLock()的次数与调用 OSSchedUnlock()的次数不一致。

L3.13(3)　OSLockNesting 减 1。

L3.13(4)

L3.13(5)　所有嵌套的函数执行完后，希望允许调度器运行，于是在任务级调用一次 OSSchedUnlock()。因为在调度器上锁期间，可能有什么事件发生了，并使一个更高优先级的任务进入就绪态。

3.08　空闲任务

μC/OS-II 总要建立一个空闲任务 (idle task)，这个任务在没有其他任务进入就绪态时投入运行。这个空闲任务[OSTaskIdle()]永远设为最低优先级，即 OS_LOWEST_PRIO。空闲任务的代码如程序清单 L3.14 所列。空闲任务不可能被应用软件删除。

程序清单 L3.14 μC/OS-II 的空闲任务

```
void  OS_TaskIdle (void *pdata)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif


    pdata = pdata;
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtr ++ ;                                          (1)
        OS_EXIT_CRITICAL();
        OSTaskIdleHook();                                       (2)
    }
}
```

L3.14(1) 空闲任务 OSTaskIdle() 在不停地给一个 32 位的名叫 OSIdleCtr 的计数器加 1，统计任务(见 3.09 节："统计任务")用这个计数器确定当前应用软件实际消耗了多少 CPU 时间。在计数器加 1 前、后，中断是先关掉，再开启的。因为 8 位及大多数 16 位微处理器的 32 位数加 1 需要多条指令，以防止高优先级的任务或中断服务子程序从中打入。

L3.14(2) OSTaskIdle() 调用 OSTaskIdleHook()，可以在这个函数中写入任何用户代码。可以借助 OSTaskIdleHook()，让 CPU 执行 STOP 指令，从而进入低功耗模式。当应用系统由电池供电时，这种方式特别有用。OSTaskIdle()是永远处于就绪态的，故不要在 OSTaskIdleHook()中调用任何可以使任务挂起的 PEND 函数、OSTimeDly???()函数以及 OSTaskSuspend()函数。

3.09 统计任务

μC/OS-II 有一个统计运行时间的任务，叫做 OSTaskStat()。如果将系统配置常数 OS_TASK_STAT_EN(见文件 OS_CFG.H)设为 1，这个任务就会建立。一旦得到了允许，OSTaskStat()运行 1 次/秒(见文件 OS_CORE.C)，计算当前的 CPU 利用率。换句话说，OSTaskStat()告诉用户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%。

如果应用程序打算使用统计任务，那么必须在初始化时建立的第 1 个也是惟一的任务中调用统计任务初始化函数 OSStatInit()(见文件 OS_CORE.C)。换句话说，在调用系统启动函数 OSStart()之前，用户初始代码中必须先建立一个任务，在这个任务中调用系统统计初始化函数 OSStatInit()，然后再建立应用程序中的其他任务。程序清单 L3.15 是统计任务的示

意性代码。

程序清单 L3.15　初始化统计任务

```
void main (void)
{
    OSInit();                                   /* 初始化 μC/OS-II */                    (1)
    /* 安装 μC/OS-II 的任务切换向量 */
    /* 创建用户起始任务(为了方便讨论,这里以 TaskStart()作为起始任务)*/                    (2)
    OSStart();                                  /* 开始多任务调度 */                     (3)
}


void TaskStart (void *pdata)
{
    /* 安装并启动 μC/OS-II 的时钟节拍 */                                                 (4)
    OSStatInit();                               /* 初始化统计任务 */                     (5)
    /* 创建用户应用程序任务 */
    for (;;) {
        /* 这里是 TaskStart()的代码! */
    }
}
```

因为用户的应用程序必须先建立一个起始任务 TaskStart()，当主程序 main()调用系统启动函数 OSStart()时，μC/OS-II 只有 3 个须管理的任务：TaskStart()，OS_TaskIdle()及 OS_TaskStat()。

注意：任务 TaskStart()的名称是无所谓的，叫什么名字都可以。因为 μC/OS-II 已经将空闲任务的优先级设为最低，即 OS_LOWEST_PRIO，统计任务的优先级设为次低，OS_LOWEST_PRIO−1。启动任务 TaskStart()总是优先级最高的任务。

图 F3.9 解释初始化统计任务时的流程。

F3.9(1)　必须首先调用的是 μC/OS-II 中的系统初始化函数 OSInit()，该函数初始化 μC/OS-II。

F3.9(2)　设置执行任务切换的中断向量。有些处理器(例如 Motorola 的 MC68HC11)不需要设置中断向量，中断向量在 ROM 中已经有了。

F3.9(3)　必须调用 OSTaskCreate()或者 OSTaskCreateExt()，以建立 TaskStart()。

F3.9(4)　进入多任务的条件准备好后，调用系统启动函数 OSStart()。这个函数将使 TaskStart()开始执行，因为 TaskStart()是优先级最高的任务。

F3.9(5)　TaskStart()负责初始化和启动时钟节拍。在第 1 个任务开始执行的时候启动时钟节拍是必要的，因为用户并不希望在多任务还没有开始时就接收到时钟节拍中断。

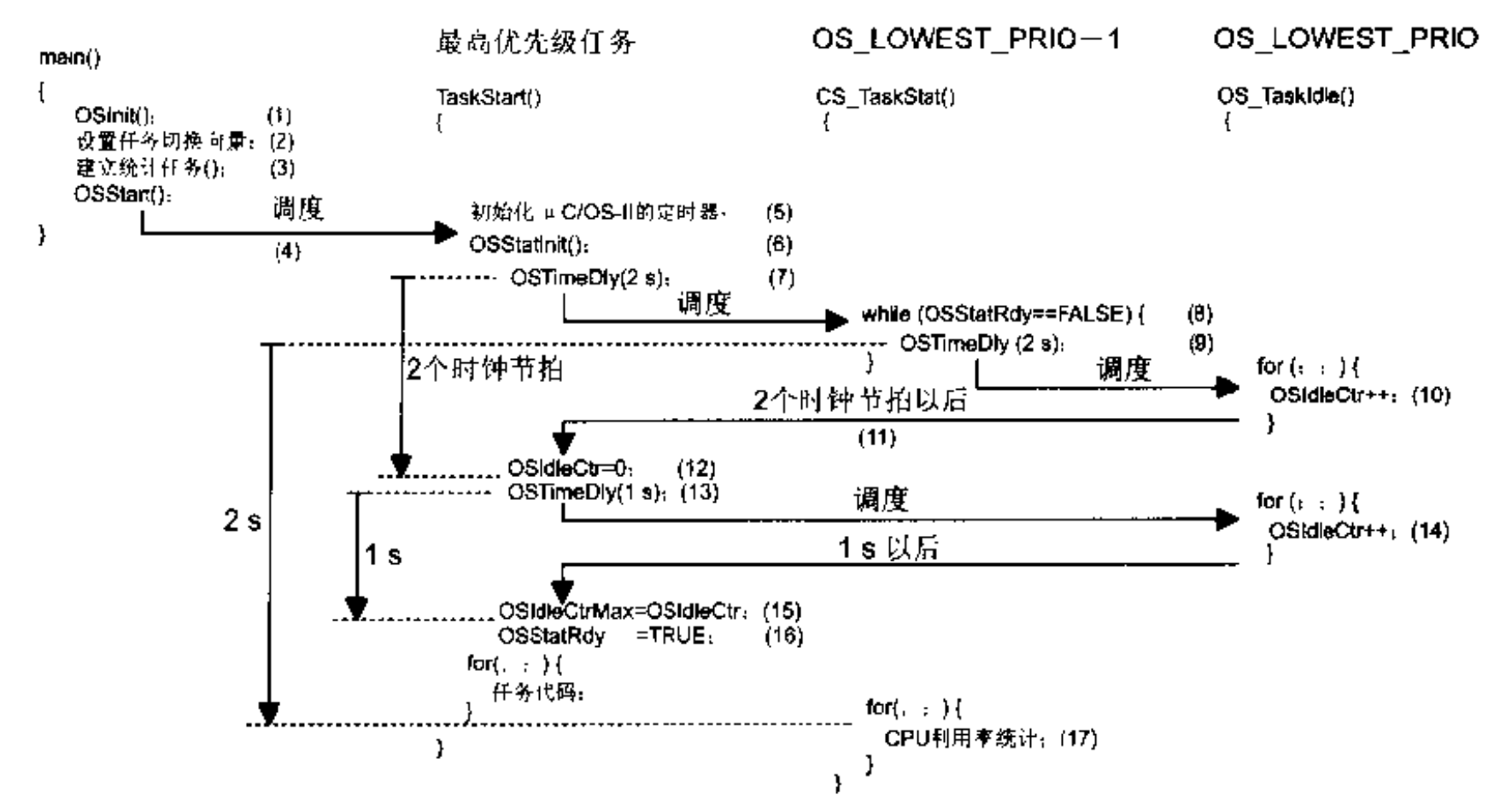


图 F3.9 统计任务的初始化

F3.9(6) TaskStart()调用统计初始化函数 OSStatInit()。统计初始化函数OSStatInit()测定在没有其他应用任务运行时，空闲计数器(OSIdleCtr)的计数速率。当奔腾II微处理器以 333 MHz 运行时，加 1 操作可使该计数器的值达到 15 000 000次/秒。OSIdleCtr 的值离 32 位计数器的溢出极限值4 294 967 296 还差得很远。按照当前微处理器发展的速度，距离 OSIdleCtr 溢出的那一天不会太久。如果出现了溢出问题，则可以在 OSTaskIdleHook()中引入一些软件延时。因为在 OSTaskIdle()中的确是并不执行什么有用的代码，丢弃一些 CPU 周期也没有什么关系。

F3.9(7) 系统统计任务初始化函数 OSStatInit()调用时间延迟函数 OSTimeDly()，将自身延时 2 个时钟节拍，以停止自身的运行。目的是使 OSStatInit()与时钟节拍同步。μC/OS-II 然后选下一个优先级最高的进入就绪态的任务运行，这恰好是统计任务 OSTaskStat()。

F3.9(8) OSTaskStat()的代码将在后面讨论，但粗看一下，OSTaskStat()一开始要做的第 1 件事就是，查看统计任务就绪标志是否为"假"(FALSE)。如果是，也要延时 2 个时钟节拍。

F3.9(9) 一定会是这样，因为标志 OSStatRdy 已被 OSInit()函数初始化为"假"了，所以实际上 OSTaskStat 也将自己置为休眠态(sleep)2 个时钟节拍。于是任务切换到空闲任务，OSTaskIdle()开始运行，这是惟一一个就绪态任务了。

F3.9(10) CPU 处于空闲任务 OSTaskIdle 中，直到 TaskStart()的延迟 2 个时钟节拍时间到。

F3.9(11)

F3.9(12) 2 个时钟节拍之后，TaskStart()恢复运行 OSStatInit()，空闲计数器OSIdleCtr 被清 0。

F3.9(13)　OSStatInit()将自身延时整整 1 s。因为没有其他进入就绪态的任务，因此 OSTaskIdle()又获得了 CPU 的使用权。

F3.9(14)　其间，OSIdleCtr 不停地加 1。

F3.9(15)　1 s 后，TaskStart()继续运行，还是在 OSStatInit()中。空闲计数器将 1 s 内计数的值存入空闲计数器最大值 OSIdleCtrMax 中。

F3.9(16)

F3.9(17)　OSStatInit()将统计任务就绪标志 OSStatRdy 设为"真"(TURE)，以此允许 2 个时钟节拍以后 OSTaskStat()开始计算 CPU 的利用率。

统计任务的初始化函数 OSStatInit()的代码如程序清单 L3.16 所列。

程序清单 L3.16　初始化统计任务

```
void  OSStatInit (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif



    OSTimeDly(2);
    OS_ENTER_CRITICAL();
    OSIdleCtr     = 0L;
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC);
    OS_ENTER_CRITICAL();
    OSIdleCtrMax  = OSIdleCtr;
    OSStatRdy     = TRUE;
    OS_EXIT_CRITICAL();
}
```

统计任务函数 OS_TaskStat()的代码如程序清单 L3.17 所列。

程序清单 L3.17　统计任务

```
void  OS_TaskStat (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT32U     run;
    INT32U     max;
    INT8S      usage;
```

```
    pdata = pdata;
    while (OSStatRdy == FALSE) {                                          (1)
        OSTimeDly(2 * OS_TICKS_PER_SEC);
    }
    max = OSIdleCtrMax / 100L;                                            (2)
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtrRun = OSIdleCtr;                                         (3)
        run          = OSIdleCtr;
        OSIdleCtr    = 0L;
        OS_EXIT_CRITICAL();
        if (max > 0L) {
            usage = (INT8S)(100L - run / max);                            (4)
            if (usage >= 0) {
                OSCPUUsage = usage;
            } else {
                OSCPUUsage = 0;
            }
        } else {
            OSCPUUsage = 0;
            max        = OSIdleCtrMax / 100L;
        }
        OSTaskStatHook();                                                 (5)
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

L3.17(1) 在前面一段中，已经讨论了为什么要等待统计任务就绪标志 OSStatRdy 变为"真"。这个任务执行 1 次/秒，以确定应用程序中的所有任务消耗了多少 CPU 时间。当应用程序代码加入后，运行空闲任务的 CPU 时间就少了，OSIdleCtr就不会像没有什么应用任务运行时那样有那么多计数。要知道，OSIdleCtr 的最大计数值是 OSStatInit()在初始化时，保存在空闲计数器最大值 OSIdleCtrMax 中的。

L3.17(3) 每过 1 s，空闲计数器的值就被复制到全局变量 OSIdleCtrRun 中。于是，这个变量就记录了刚刚过去的 1 s 内空闲计数器的最大值。这个变量在 μC/OS-II的其他任何地方都没有用，但在应用程序中可以监视或显示它。空闲计数器然后清 0，以用于下一次测量。

L3.17(4) CPU 利用率(表达式[3.1])是保存在变量 OSCPUUsage[L3.14(2)]中的。

$$\text{OSCPUUsage}(\%) = 100 \times \left(1 - \frac{\text{OSIdleCtr}}{\text{OSIdleCtrMax}}\right) \tag{3.1}$$

L3.17(2)　表达式应该重写，OSIdleCtr/OSIdleCtrMax 永远为 0，因为是整数运算。新的表达式为：

$$\text{OSCPUUsage}(\%) = \left(100 - \frac{100 \times \text{OSIdleCtr}}{\text{OSIdleCtrMax}}\right) \tag{3.2}$$

将 OSIdleCtr 乘以 100，限制了 OSIdleCtr 的最大值，特别是当处理器特别快时。换言之，为使 OSIdleCtr 与 100 的乘积不溢出，OSIdleCtr 的值任何时候都不能大于 42 949 672。对于高速处理器，OSIdleCtr 很快就会达到这个值。为解决这个潜在的问题，只需将 OSIdleCtrMax 除以 100，如表达式 3.3 所示。

$$\text{OSCPUUsage}(\%) = \left[100 - \frac{\text{OSIdleCtr}}{\left(\frac{\text{OSIdleCtrMax}}{100}\right)}\right] \tag{3.3}$$

局部变量 max 就这样预先算好了，等于 OSIdleCtrMax 除以 100。

L3.17(5)　一旦上述计算完成，函数 OSTaskStat()即调用统计任务外界接入函数 OSTaskStatHook()。这是一个可以由用户定义的函数，这个函数能使统计任务得到扩展。这样，用户可以计算并显示所有任务的总的执行时间、每个任务执行时间的百分比以及其他信息等(见第 1 章的范例 3)。

3.10　μC/OS－II 中的中断

μC/OS－II 中，中断服务子程序要用汇编语言来写。然而，如果用户使用的 C 语言编译器支持在线汇编语言，则可以直接将中断服务子程序代码放在 C 语言的程序文件中。

中断服务子程序的示意性代码如程序清单 L3.18 所列。

程序清单 L3.18　μC/OS－II 中的中断服务子程序

```
用户中断服务子程序：
    保存全部 CPU 寄存器；                                    (1)
    调用 OSIntEnter()或 OSIntNesting 直接加 1；              (2)
    if (OSIntNesting == 1) {                                 (3)
        OSTCBCur->OSTCBStkPtr = SP;                          (4)
    }
    清中断源；                                               (5)
    重新开中断；                                             (6)
    执行用户代码做中断服务；                                 (7)
    调用 OSIntExit();                                        (8)
    恢复所有 CPU 寄存器；                                    (9)
    执行中断返回指令；                                       (10)
```

L3.18(1) 用户代码应该将全部CPU寄存器推入当前任务栈。

注意：有些微处理器，例如Motorola 68020（及68020以上的处理器），做中断服务时使用另外的栈。μC/OS-II可以用在这类微处理器中，当任务切换时，寄存器是保存在被中断了的任务的堆栈中的。

L3.18(2) μC/OS-II需要知道正在做中断服务，故应该调用OSIntEnter()，或者将全局变量OSIntNesting直接加1。可以直接给OSIntNesting加1，这样比调用OSIntEnter()快得多，使用这种方法更快捷。

有些处理器，如Motorola 68020，在中断服务的一开始就允许中断嵌套。这类处理器中断服务子程序的开始处会有所不同。不打算在这里讨论这种情况，可以从www.uCOS-II.com网站上下载CPU32的移植范例，看看这种情况该如何处理。

L3.18(3)

L3.18(4) 查看是否为中断的第1层，如果是，立即将堆栈指针保存到这个任务的任务控制块OS_TCB中。应注意到，自μC/OS-IIV2.04以上版本，增加了这2行代码。如手中有μC/OS-II V2.04或更早的版本，应在所有中断服务子程序中加上这2行。

L3.18(5) 如果决定重新允许中断，则必须清中断源；否则会重新进入这一段中断服务子程序。

L3.18(6) 如果允许中断嵌套，则可以重新开中断。μC/OS-II允许中断嵌套，因为μC/OS-II跟踪嵌套层数OSIntNesting。

L3.18(7) 上述2步完成后，可以开始服务于请求中断的设备了。这一段完全取决于应用。

L3.18(8) 调用脱离中断函数OSIntExit()，则标志着中断服务子程序的终结，OSIntExit()将中断嵌套层数计数器减1。当嵌套计数器减到0时，所有嵌套的中断就都完成了。此时μC/OS-II须判定是否有优先级更高的任务被中断服务子程序（或任一层嵌套的中断）唤醒。如果有优先级更高的任务进入了就绪态，μC/OS-II就返回到更高优先级的任务，而不是返回到被中断了的任务。

L3.18(9) 如果被中断了的任务仍然是最重要的任务，则OSIntExit()从中断服务子程序返回到原来的任务。

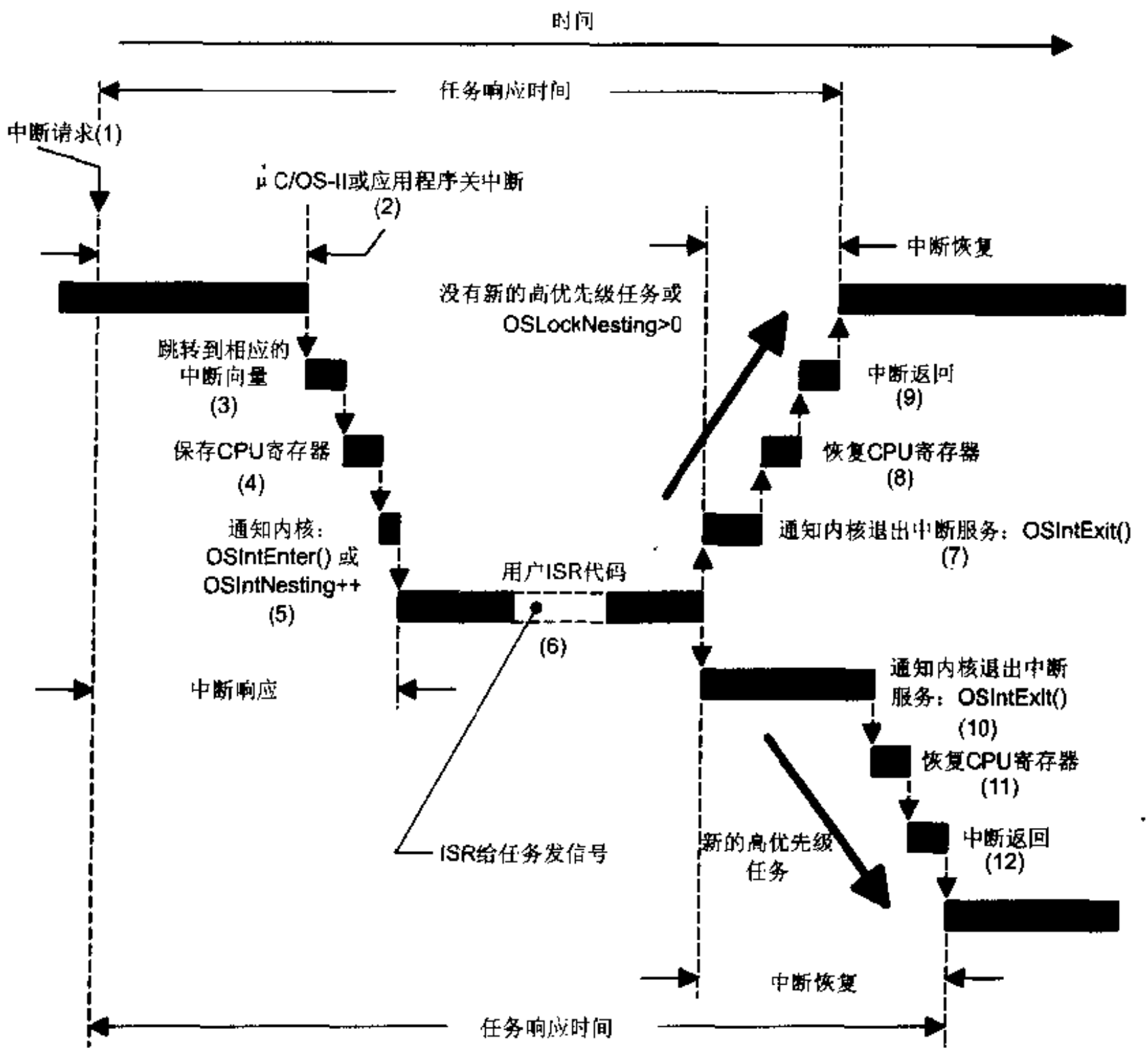
L3.18(10) 保存的寄存器值是在这个时候恢复的，然后执行中断返回指令。

注意：如果调度被禁止了（OSIntNesting > 0），μC/OS-II将返回到被中断了的任务。

以上描述的详细解释如图F3.10所示。

F3.10(1) 中断到来，但还不能被CPU识别。也许是因为中断被μC/OS-II或用户应用程序关了，或者是因为CPU还没执行完当前指令。

F3.10(2)



注：在(5)中，以V2.51算法做移植时，应加上：OSTCBCur->OSTCVStkPtr=SP。

图 F3.10　中断服务

F3.10(3)　一旦 CPU 响应了这个中断，CPU 的中断向量（至少大多数微处理器是如此）跳转到中断服务子程序。

F3.10(4)　中断服务子程序保存 CPU 的全部寄存器（也叫做 CPU context）。

F3.10(5)　保存完 CPU 寄存器之后，中断服务子程序通知 μC/OS－II 进入中断服务子程序。做法是调用 OSIntEnter()，或者给 OSIntNesting 加 1。还应将堆栈指针保存到当前任务控制块 OS_TCB 中。

F3.10(6)　用户中断服务代码开始执行。中断服务所做的事应尽可能地少，应把大部分工作留给任务去做。中断服务子程序通知某任务去做事的手段是调用以下函数之一：OSFlagPost()，OSMboxPost()，OSQPost()，OSQPostFront()，或者 OSSemPost()。中断发生并由上述函数发出消息时，接收消息的任务可能是，也可能不是挂起在邮箱、队列或信号量上的任务。

F3.10(7)　中断服务完成后，须调用 OSIntExit()。从时序图上可以看出，对被中断了的

任务来说，如果没有高优先级的任务被中断服务子程序激活而进入就绪态，那么 OSIntExit()只占用很短的运行时间。

F3.10(8)

F3.10(9) 在这种情况下，CPU 寄存器只是被简单地恢复并执行中断返回指令。

F3.10(10) 如果中断服务子程序使一个高优先级的任务进入了就绪态，则 OSIntExit()将占用较长的运行时间，因为这时要做任务切换。

F3.10(11)

F3.10(12) 新任务的 CPU 寄存器内容要被重新装入，并执行中断返回指令。

进入中断函数 OSIntEnter()的代码如程序清单 L3.19 所列，从中断服务中退出函数 OSIntExit()的代码如程序清单 L3.20 所列。如前所述，OSIntEnter()所做的事是非常少的。

程序清单 L3.19　通知 μC/OS-II 中断服务子程序 ISR 开始

```
void  OSIntEnter (void)
{
    if (OSRunning == TRUE) {
        if (OSIntNesting < 255) {
            OSIntNesting++;
        }
    }
}
```

程序清单 L3.20　通知 μC/OS-II 离开中断服务子程序 ISR

```
void  OSIntExit (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif

    OS_ENTER_CRITICAL();
    if (OSRunning == TRUE) {
        if (OSIntNesting > 0) {                                          (1)
            OSIntNesting--;
        }
        if ((OSIntNesting == 0) && (OSLockNesting == 0)) {
            OSIntExitY    = OSUnMapTbl[OSRdyGrp];                        (2)
            OSPrioHighRdy = (INT8U)((OSIntExitY << 3)
                          + OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
            if (OSPrioHighRdy != OSPrioCur) {
                OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
                OSCtxSwCtr++;
```

```
            OSIntCtxSw();                                                    (3)
        }
    }
  }
  OS_EXIT_CRITICAL();
}
```

OSIntExit()看起来非常像 OSSched()，但有 3 点不同：

L3.20(1)　OSIntExit()使中断嵌套层数减 1，而重新调度的条件是：中断嵌套层数计数器和锁定嵌套计数器(OSLockNesting)二者都必须是 0。

L3.20(2)　OSRdyTbl[]所需的检索值 Y 是保存在全局变量 OSIntExitY 中的。因为在早于 V2.51 版本的 μC/OS-II 中，OS_IntCtxSw()须考虑局部变量和返回地址；而在 μC/OS-IIV2.51 版本中，OS_IntCtxSw()不需要考虑这些。然而，笔者决定保留局部变量 OSIntExitY，以保持和以前的移植版本兼容。

L3.20(3)　如果须做任务切换，OSIntExit()将调用 OSIntCtxSw()，而不是调用 OS_TASK_SW()，如同在 OSSched()函数中那样。

应调用中断切换函数 OSIntCtxSw()，而不应调用任务切换函数 OS_TASK_SW()。因为中断服务子程序已经将 CPU 寄存器存入到中断了的任务的堆栈中，而无需再做第 2 次。OSIntCtxSw()是如何实现的，将在第 13 章："移植 μC/OS-II"中描述。

有些微处理器，像 Motorola 68HC11，中断发生时，CPU 寄存器是自动入栈的，且若要允许中断嵌套，在中断服务子程序中须重新开中断，这可以视作一个优点。确实，如果中断服务子程序执行得非常快，无须通知任务，已经进入了中断服务，只要不在中断服务期间开中断，也无须调用 OSIntEnter()或 OSIntNesting 加 1。程序清单 L3.21 中的示意代码说明了这种情况。一个任务和这个中断服务子程序通信的惟一方法是通过全局变量。

程序清单 L3.21　Motorola 68HC11 中的中断服务子程序

```
M68HC11_ISR:                    /* 快速中断服务子程序，不准开中断 */
    所有寄存器被 CPU 自动保存；
    执行用户代码，以响应中断；
```

3.11　时钟节拍

μC/OS-II 需要提供周期性信号源，用于实现时间延时和确认超时。节拍率应为 10～100 次/秒，或者说 10～100 Hz。时钟节拍率越高，系统的额外负荷就越重。时钟节拍的实际频率取决于应用程序的精度。时钟节拍源可以是专门的硬件定时器，也可以是来自 50/60 Hz

交流电源的信号。

必须在多任务系统启动以后，也就是在调用 OSStart()之后，再开启时钟节拍器。换句话说，调用 OSStart()之后应做的第一件事是初始化定时器中断。通常容易犯的错误是，将允许时钟节拍器中断放在系统初始化函数 OSInit()之后，在启动多任务系统启动函数OSStart()之前，如程序清单 L3.22 所列。

程序清单 L3.22 启动时钟节拍器的不正确做法

```
void main (void)
{
    .
    .
    OSInit();                    /* 初始化 μC/OS-II                                  */
    .
    .
    /* 应用程序初始化代码...                                                         */
    /* ... 通过调用 OSTaskCreate()创建至少一个任务                                   */
    .
    .
    允许时钟节拍(TICKER)中断;    /* 千万不要在这里允许时钟节拍中断!!!               */
    .
    .
    OSStart();                   /* 开始多任务调度                                   */
}
```

μC/OS-II 中的时钟节拍服务是通过在中断服务子程序中调用 OSTimeTick()实现的。OSTimeTick()跟踪所有任务的定时器以及超时时限，时钟节拍中断服从 3.10 节："μC/OS-II中的中断"中描述的所有规则。时钟节拍中断服务子程序的示意性代码如程序清单L3.23所列。这段代码必须用汇编语言编写，因为在 C 语言里不能直接处理 CPU 的寄存器。节拍中断总是需要的，这部分程序在移植代码中提供。

程序清单 L3.23 时钟节拍中断服务子程序示意性代码

```
void OSTickISR (void)
{
    保存处理器寄存器的值;
    调用 OSIntEnter(),或是将 OSIntNesting 加 1;
    if (OSIntNesting == 1) {
        OSTCBCur->OSTCBStkPtr = SP;
    }
    调用 OSTimeTick();
    清发出中断设备的中断;
    重新允许中断(可选用);
```

```
    调用 OSIntExit();
    恢复处理器寄存器的值;
    执行中断返回指令;
}
```

时钟节拍函数（OSTimeTick()）的代码如程序清单 1.3.24 所列

程序清单 1.3.24　时钟节拍函数 OSTimTick() 的节拍服务

```
void  OSTimeTick (void)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR   cpu_sr;
# endif
    OS_TCB      *ptcb;



    OSTimeTickHook();                                                    (1)
# if OS_TIME_GET_SET_EN ) 0
    OS_ENTER_CRITICAL();
    OSTime ++ ;                                                          (2)
    OS_EXIT_CRITICAL();
# endif
    if (OSRunning == TRUE) {
        ptcb = OSTCBList;                                                (3)
        while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {                        (4)
            OS_ENTER_CRITICAL();
            if (ptcb->OSTCBDly != 0) {
                if ( -- ptcb->OSTCBDly == 0) {
                    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == 0x00) {   (5)
                        OSRdyGrp               |= ptcb->OSTCBBitY;       (6)
                        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                    } else {
                        ptcb->OSTCBDly = 1;
                    }
                }
            }
            ptcb = ptcb->OSTCBNext;
            OS_EXIT_CRITICAL();
        }
    }
}
```

L3.24(1) OSTimeTick()以调用由用户定义的时钟节拍外连函数 OSTimeTickHook()开始。这个外连函数可以将时钟节拍函数 OSTimeTick()扩展。笔者决定首先调用 OSTimeTickHook(),是打算在时钟节拍中断服务一开始,给用户一个可以做点儿什么的机会.因为用户可能会有时间要求苛刻的工作要做。OSTimeTick()中工作量较大的是,给每个用户任务控制块 OS_TCB 中的时间延时项 OSTCBDly 减 1(如果该项不为 0 的话)。

L3.24(2) OSTimeTick() 还计算自从系统上电以来的时钟节拍数,这是一个无符号 32 位变量,名为 OSTime。

注意:在给 OSTime 加 1 之前,关了中断。因为对有些处理器而言,32 位数的加 1 通常需要多条指令。

L3.24(3)

L3.24(4) OSTimeTick()从 OSTCBList 开始,沿着 OS_TCB 链表做,一直做到空闲任务。

L3.24(6) 当某任务的任务控制块中的时间延时项 OSTCBDly 减为 0 时,这个任务就进入了就绪态。

L3.24(5) 而确实被任务挂起函数 OSTaskSuspend()挂起的任务则不会进入就绪态。

OSTimeTick()的执行时间正比于应用程序中建立的任务数;然而,执行时间仍然是可以确定的。

中断服务子程序似乎就须写这么长。如果不喜欢将中断服务程序写这么长,可以从任务级调用 OSTimeTick(),如程序清单 L3.25 所列。这样,须建立一个高于应用程序中所有其他任务优先级的任务。时钟节拍中断服务子程序利用信号量或邮箱,给这个高优先级的任务发信号。

程序清单 L3.25 用时钟节拍任务 TickTask()做时钟节拍服务

```
void TickTask (void *pdata)
{
    pdata = pdata;
    for (;;) {
        OSMboxPend(...);     /* 等待从时钟节拍中断服务程序发来的信号 */
        OSTimeTick();
        OS_Sched();
    }
}
```

当然需要先建立一个邮箱(初始化成 NULL),用于给上述任务发信号,告知时钟节拍中断已经发生了(程序清单 L3.26)。

程序清单 L3.26　用时钟节拍中断服务函数 OSTickISR()做节拍服务

```
void OSTickISR (void)
{
    保存处理器寄存器的值;
    调用 OSIntEnter(),或是将 OSIntNesting 加 1;
    if (OSIntNesting == 1) {
        OSTCBCur->OSTCBStkPtr = SP;
    }

    发哑消息(dummy) (即(void *)1) 给时钟节拍邮箱;
    调用 OSIntExit();
    恢复处理器寄存器的值;
    执行中断返回指令;
}
```

3.12　μC/OS－II 初始化

在调用 μC/OS－II 的任何其他服务之前，μC/OS－II 要求首先调用系统初始化函数 OSIint()。OSIint()初始化 μC/OS－II 所有的变量和数据结构(见 OS_CORE.C)。OSInit()建立空闲任务 OS_TaskIdle()，这个任务总是处于就绪态。空闲任务 OS_TaskIdle()的优先级总是设成最低，即 OS_LOWEST_PRIO。如果统计任务允许 OS_TASK_STAT_EN 和任务建立扩展都设为 1，则 OSInit()还须建立统计任务 OS_TaskStat()，并且使其进入就绪态。OS_TaskStat的优先级总是设为 OS_LOWEST_PRIO －1。

图 F3.11 表示调用 OSInit()之后，μC/OS－II 的变量和数据结构之间的关系。其解释假设，在 OS_CFG.H 中，#define 语句的定义如下：

- OS_TASK_STAT_EN 设为 1；
- OS_FLG_EN 设为 1；
- OS_LOWEST_PRIO 设为 63。
- 最多任务数 OS_MAX_TASKS 设为 62。

F3.11(1)　注意：以上 2 个任务的任务控制块(OS_TCBs)是用双向链表链接在一起的。

F3.11(2)　OSTCBList 指向这个链表的起始处。当建立一个任务时，这个任务总是被放在这个链表的开始处。换句话说，OSTCBList 总是指向最后建立的任务。

F3.11(3)　链表的终点指向空指针 NULL(也就是 0)。

F3.11(4)　因为这 2 个任务都处于就绪态，在就绪任务表 OSRdyTbl[]中的相应位是设为 1 的。还有，因为这 2 个任务的相应位在 OSRdyTbl[]的同一行上，即属于同一组，故 OSRdyGrp 中只有 1 位是设为 1 的。

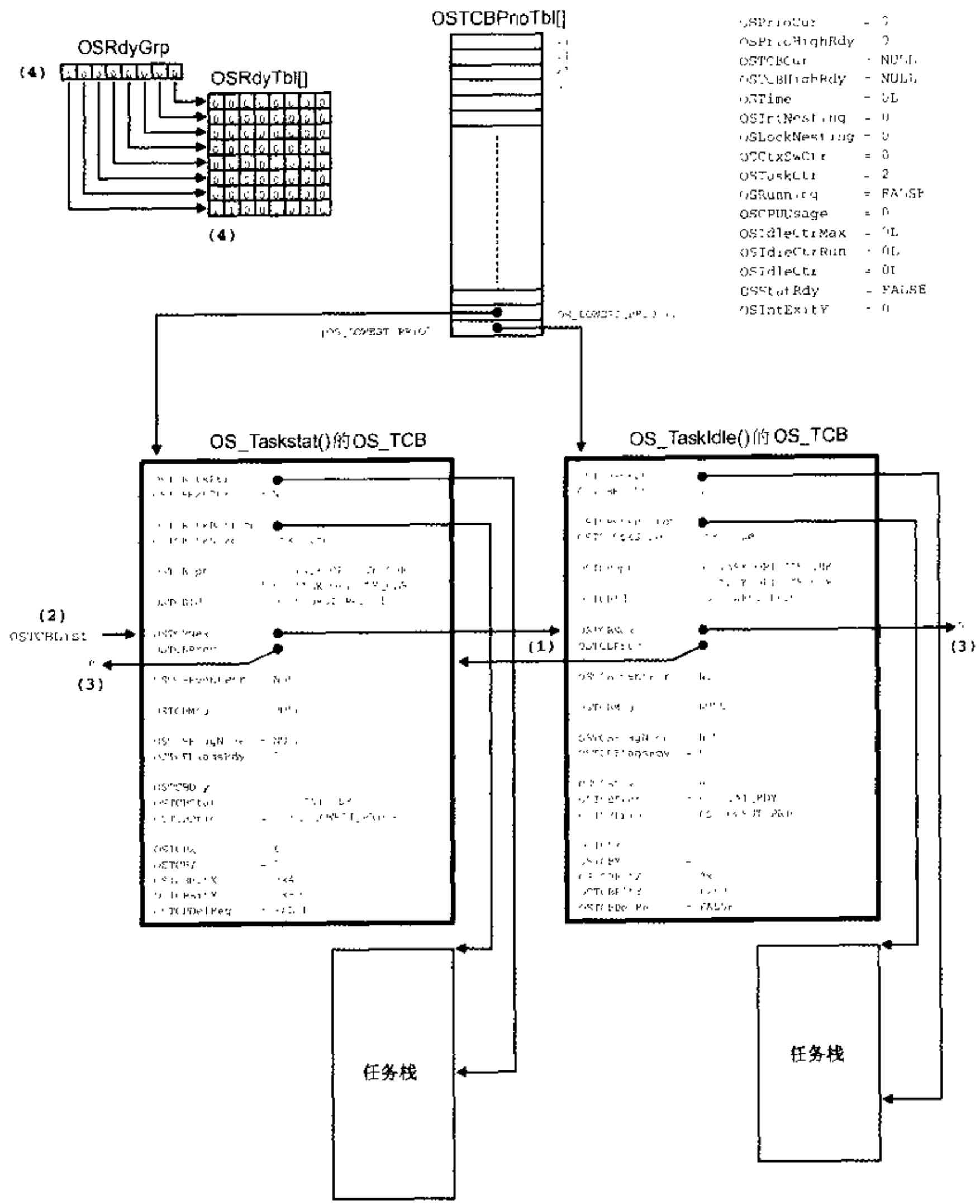


图 F3.11 调用 OSInit()之后的变量与数据结构

μC/OS-II 还初始化了 5 个空的数据结构缓冲区，如图 F3.12 所示，每个缓冲区都是单向链表，允许 μC/OS-II 从缓冲区中迅速取得或释放一个缓冲区中的元素

调用 OSInit()以后，任务控制块缓冲池中有 OS_MAX_TASKS 个任务控制块，事件控制块缓冲池中有 OS_MAX_EVENTS 个事件控制块，消息队列缓冲池 OS_Q 中有 OS_MAX_QS

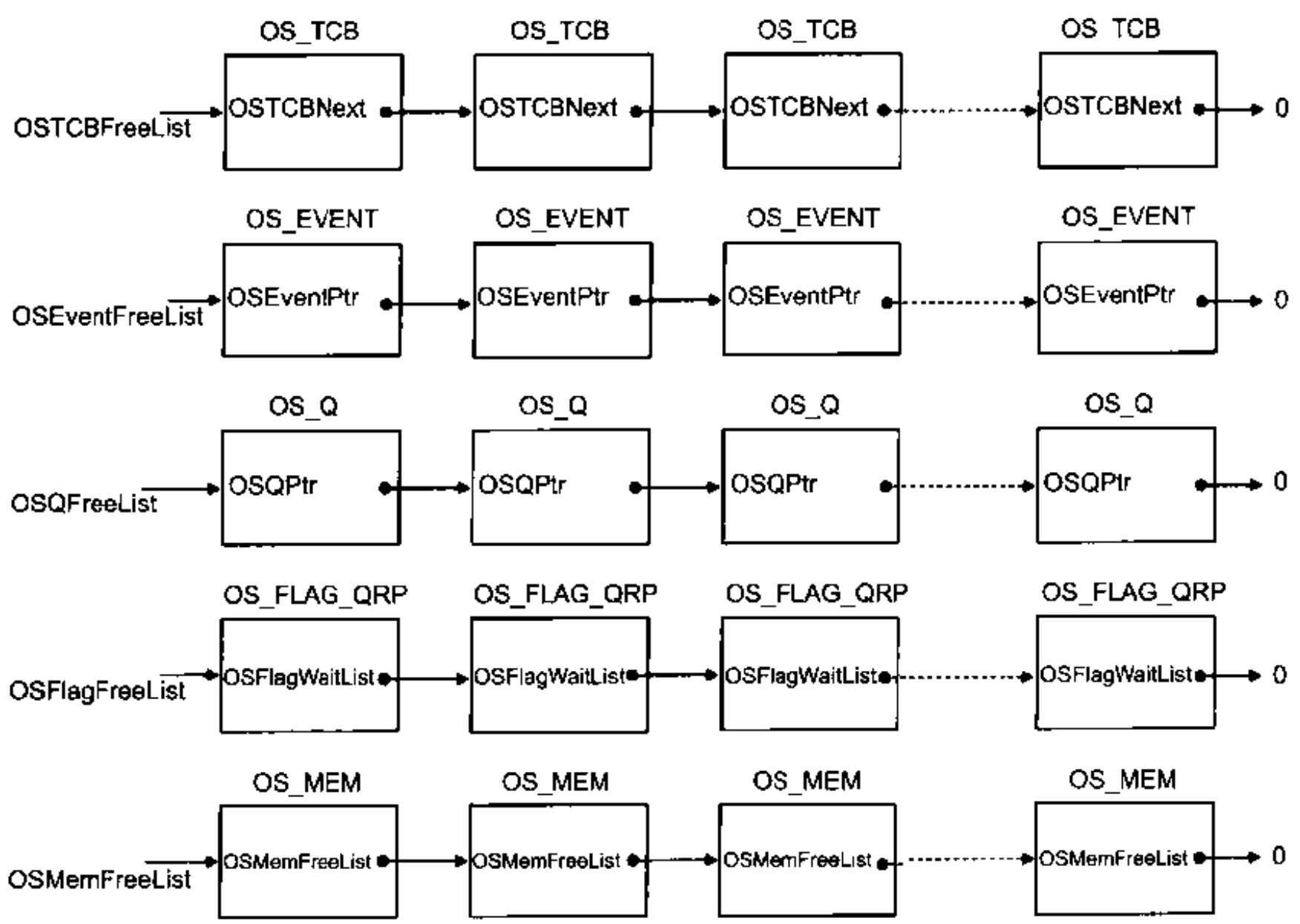


图 F3.12　空闲缓冲池

个消息队列控制块，OS_FLAG_GRP 缓冲池中有 OS_MAX_FLAGS 个标志控制块。最后，OS_MEM缓冲池中有 OS_MAX_MEM_PART 个存储控制块。每个空闲缓冲池以空指针结束。当然，如果所有列表中的指针全部指向 NULL，缓冲池全空，缓冲池的容量在 OS_CFG.H中定义。

3.13　μC/OS－II 的启动

多任务的启动是通过调用 OSStart()实现的。然而，启动 μC/OS－II 之前，至少须建立一个应用任务，如程序清单 L3.27 所列。

程序清单 L3.27　初始化和启动 μC/OS－II

```
void main (void)
{
    OSInit();                /* 初始化 μC/OS-II */
    .
    .
    通过调用 OSTaskCreate()或 OSTaskCreateExt()创建至少一个任务；
```

```
    .
    .
    OSStart();          /* 开始多任务调度！OSStart()永远不会返回 */
}
```

OSStart()的代码如程序清单 L3.28 所列。

程序清单 L3.28　启动多任务

```
void OSStart (void)
{
    INT8U y;
    INT8U x;

    if (OSRunning == FALSE) {
        y             = OSUnMapTbl[OSRdyGrp];
        x             = OSUnMapTbl[OSRdyTbl[y]];
        OSPrioHighRdy = (INT8U)((y << 3) + x);
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];                    (1)
        OSTCBCur      = OSTCBHighRdy;
        OSStartHighRdy();                                               (2)
    }
}
```

L3.28(1)　当调用 OSStart()时，OSStart()从任务就绪表中找出用户建立的优先级最高任务的任务控制块。

L3.28(2)　OSStart()调用高优先级就绪任务启动函数 OSStartHighRdy()（见汇编语言文件 OS_CPU_A.ASM，这个文件与选择的微处理器有关（见第 13 章："移植 μC/OS-II"））。实质上，函数 OSStartHighRdy()是将任务栈中保存的值弹回到 CPU 寄存器中，然后执行一条中断返回指令，中断返回指令强制执行该任务代码。

注意：OSStartHighRdy()将永远不返回到 OSStart()。

多任务启动以后，变量与数据结构中的内容如图 F3.13 所示。这里假设建立的任务优先级为 6。

注意：OSTaskCtr 指出已经建立了 3 个任务。

OSRunning 已设为"真"，指出多任务已经开始；OSPrioCur 和 OSPrioHighRdy 存放的是应用任务的优先级；OSTCBCur 和 OSTCBHighRdy 二者都指向任务的任务控制块。

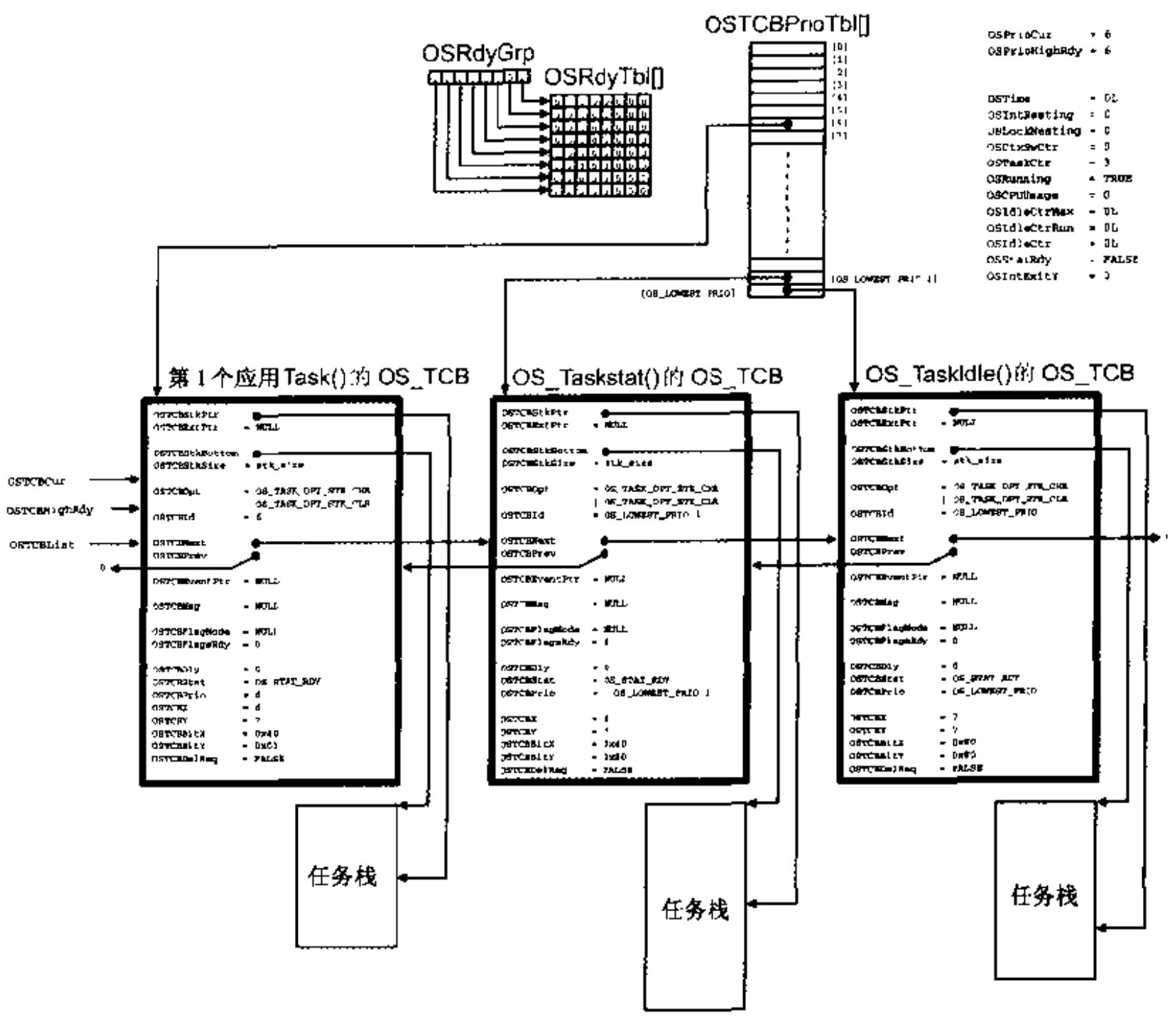


图 F3.13　调用 OSStart()以后的变量与数据结构

3.14　获取当前 μC/OS－II 的版本号

应用程序调用函数 OSVersion()(程序清单 L3.29)，可以得到当前 μC/OS－II 的版本号。OSVersion()函数返回的值是版本号乘以 100。换言之，200 表示版本号 2.00。

程序清单 L3.29　得到 μC/OS－II 当前版本号

```
INT16U OSVersion (void)
{
    return (OS_VERSION);
}
```

为找到 μC/OS－II 的最新版本以及做版本升级，可以查看 μC/OS－II 的正式网站：http://www.uCOS-II.com。

第 4 章

任务管理

在前面的章节中，曾叙述过任务可以是一个无限的循环，也可以在一次执行完毕后被删除掉。这里要注意的是，任务代码并不是被真正地删除了，而只是 μC/OS－II 不再理会该任务代码，所以该任务代码不会再运行。任务看起来与任何 C 函数一样，具有一个返回类型和一个参数，只是它决不返回。任务的返回类型必须定义成 void 型。本章所提到的函数可以在 OS_TASK.C 文件中找到。任务必须是以下 2 种结构之一：

```
void YourTask (void *pdata)
{
    for (;;) {
        /* 用户代码 */
        调用 μC/OS－II 的服务例程之一：
            OSFlagPend();
            OSMboxPend();
            OSMutexPend();
            OSQPend();
            OSSemPend();
            OSTaskSuspend(OS_PRIO_SELF);
            OSTimeDly();
            OSTimeDlyHMSM();
        /* 用户代码 */
    }
}
```

或

```
void YourTask (void *pdata)
{
    /* 用户代码 */
    OSTaskDel(OS_PRIO_SELF);
}
```

本章内容包括如何在应用程序中建立任务、删除任务、改变任务的优先级及挂起和恢复任务，以及如何获得有关任务的信息。

虽然 μC/OS-II 可以管理多达 64 个任务，但建议保留 4 个最高优先级和 4 个最低优先级的任务，供以后 μC/OS-II 的版本使用。然而，目前 μC/OS-II 实际上只使用了 2 个优先级别：OS_LOWEST_PRIO 和 OS_LOWEST_PRIO－1（参看文件 OS_CFG.H）。即使这样，留给用户使用的任务仍可多达 56 个。优先级的值越低，表示任务的优先级越高。在当前的 μC/OS-II版本中，任务的优先级数也可作为任务的标志符使用。

4.00　建立任务，OSTaskCreate()

如果想让 μC/OS-II 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 2 个函数之一来建立任务：OSTaskCreate() 或 OSTaskCreateExt()。OSTaskCreate()与 μC/OS 是向下兼容的，OSTaskCreateExt()是 OSTaskCreate()的扩展，提供了一些附加的功能。用 2 个函数中的任何一个都可以建立任务。任务可以在多任务调度开始前建立，也可以在其他任务的执行过程中建立。在开始多任务调度（调用 OSStart()）前，必须建立至少一个任务。任务不能由中断服务程序（ISR）建立。

OSTaskCreate()的代码如程序清单 L4.1 所列。从中可以知道，OSTaskCreate()需要 4 个参数：task 是指向任务代码的指针；pdata 是任务开始执行时，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（参看 4.02 节："任务堆栈"）；prio 是分配给任务的优先级。

程序清单 L4.1　OSTaskCreate()

```
INT8U OSTaskCreate (void ( *task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    void       *psp;
    INT8U      err;

#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO){                                          (1)
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {                             (2)
        OSTCBPrioTbl[prio] = (OS_TCB *)1;                                (3)
        OS_EXIT_CRITICAL();                                              (4)
        psp = (void *)OSTaskStkInit(task, pdata, ptos, 0);               (5)
        err = OS_TCBInit(prio, psp, (void *)0, 0, 0, (void *)0, 0);      (6)
```

```
        if (err == OS_NO_ERR) {                                    (7)
            OS_ENTER_CRITICAL();
            OSTaskCtr++;                                           (8)
            OS_EXIT_CRITICAL();
            if (OSRunning == TRUE) {                               (9)
                OS_Sched();                                        (10)
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;                      (11)
            OS_EXIT_CRITICAL();
        }
        return (err);
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
```

L4.1(1) 如果常数 OS_ARG_CHK_EN (参看文件 OS_CFG.H)被设置为1,那么OSTaskCreate()会检测分配给任务的优先级是否是有效的。任务的优先级必须为0~OS_LOWEST_PRIO,包括0和OS_LOWEST_PRIO。请注意,优先级OS_LOWEST_PRIO是为μC/OS-II空闲任务保留的,但不必担心会因此出错。因为在执行系统初始化函数OSInit()时,已经把优先级OS_LOWEST_PRIO分配给了空闲任务,所以在应用程序中不能调用OSTaskCreate()函数来创建优先级为OS_LOWEST_PRIO的任务。如果试图这样做,OSTaskCreate()会返回代码出错信息OS_PRIO_EXIST。

L4.1(2) OSTaskCreate()须确保拟定义的优先级没有被其他任务占用。在μC/OS-II中,每个任务的优先级必须各不相同。

L4.1(3) 如果拟分配的优先级是空闲的,那么μC/OS-II通过在OSTCBPrioTbl[]中放置一个非空指针,保留该优先级。

L4.1(4) 这样就使得OSTaskCreate()在设置任务数据结构的其他部分时,能重新开中断。因为此时该优先级已经被占用,所以,即使其他任务并行地调用OSTaskCreate(),也无法再把该优先级分配给别的任务了。

L4.1(5) OSTaskCreate()调用OSTaskStkInit()。它负责建立任务的堆栈。该函数是与处理器的硬件体系相关的函数,可以在OS_CPU_C.C文件中找到。有关实现OSTaskStkInit()的细节,可参看第13章:"移植μC/OS-II"。如果有人已经在读者打算用的处理器上成功地移植了μC/OS-II,而又能得到其源代码,就不必深究该函数的实现细节了。OSTaskStkInit()函数返回新的堆栈的栈顶(psp),并保存在任务控制块OS_TCB中。

注意:应将传递给OSTaskStkInit()函数的第4个参数opt清0,因为OSTaskCreate()与OSTaskCreateExt()不同,不支持在任务的建立过程中设

置选项，所以没有任何选项可以通过 opt 参数传递给 OSTaskStkInit()。

μC/OS-II 支持的处理器的堆栈，既可从上(高地址)往下(低地址)递减，也可从下往上递增。调用 OSTaskCreate()时，必须知道堆栈是递增的，还是递减的(参见所用处理器的 OS_CPU.H 中的 OS_STACK_GROWTH)。因为必须把堆栈的栈顶传递给 OSTaskCreate()，而栈顶可能是堆栈的最高地址(堆栈从上往下递减)，也可能是最低地址(堆栈从下往上递减)。

L4.1(6) 一旦 OSTaskStkInit()函数完成了堆栈的建立，OSTaskCreate()就调用 OS_TCBInit()，从空闲的 OS_TCB 缓冲池中获得并初始化一个任务控制块 OS_TCB。OS_TCBInit()的代码可参见 3.03 节："任务控制块(OS_TCB)"，它是在 OS_CORE.C 文件中定义的，而不是在 OS_TASK.C 文件中定义的。

L4.1(7)

L4.1(8) 从 OS_TCBInit()返回后，OSTaskCreate()会检测返回的代码。如果成功，就给任务计数器 OSTaskCtr 加 1。OSTaskCtr 用于跟踪已经建立了的任务数目。

L4.1(11) 如果 OS_TCBInit()返回任务控制块初始化失败，OSTaskCreate()会通过设置 OSTCBPrioTbl[prio]的入口为 0，而放弃该任务的优先级。

L4.1(9)

L4.1(10) 如果 OSTaskCreate()函数是在某个任务的执行过程中被调用的(OSRunning 已置为 True)，则任务调度函数会被调用，以判断新建立的任务是否比原来的任务有更高的优先级。如果新任务的优先级更高，内核会进行一次从旧任务到新任务的任务切换。如果该任务是在多任务调度开始之前(用户建立任务时还没有调用过 OSStart())，则任务调度函数不会起作用。

4.01 建立任务，OSTaskCreateExt()

用 OSTaskCreateExt()函数来建立任务会更加灵活，但是会增加一些额外的开销。OSTaskCreateExt()函数的代码如程序清单 L4.2 所列。

程序清单 L4.2 OSTaskCreateExt()

```
INT8U  OSTaskCreateExt (void      (*task)(void *pd),
                        void      *pdata,
                        OS_STK    *ptos,
                        INT8U      prio,
                        INT16U     id,
                        OS_STK    *pbos,
                        INT32U     stk_size,
                        void      *pext,
                        INT16U     opt)
{
```

```
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_STK    *psp;
    INT8U      err;


#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO) {                                          (1)
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {                              (2)
        OSTCBPrioTbl[prio] = (OS_TCB *)1;                                 (3)

        OS_EXIT_CRITICAL();                                               (4)

        if (((opt & OS_TASK_OPT_STK_CHK) != 0x0000) ||                    (5)
            ((opt & OS_TASK_OPT_STK_CLR) != 0x0000)) {
            #if OS_STK_GROWTH == 1
            (void)memset(pbos, 0, stk_size * sizeof(OS_STK));
            #else
            (void)memset(ptos, 0, stk_size * sizeof(OS_STK));
            #endif
        }

        psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, opt);            (6)
        err = OS_TCBInit(prio, psp, pbos, id, stk_size, pext, opt);       (7)
        if (err == OS_NO_ERR) {                                           (8)
            OS_ENTER_CRITICAL();                                          (9)
            OSTaskCtr++;
            OS_EXIT_CRITICAL();
            if (OSRunning == TRUE) {                                      (10)
                OS_Sched();                                               (11)
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;                             (12)
            OS_EXIT_CRITICAL();
        }
        return (err);
```

```
    }
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
```

可以看到，OSTaskCreateExt()需要 9 个参数。前 4 个参数(task、pdata、ptos 及 prio)与 OSTaskCreate()的 4 个参数完全相同，先后顺序也一样。这样做的目的是，使用户能够更容易地将应用程序从 OSTaskCreate()移植到 OSTaskCreateExt()上。

id　此参数为要建立的任务创建一个特殊的标志符。该参数以后在 μC/OS - II 升级版本中可能会用到，但在目前的 μC/OS - II 中没有使用。这个标志符可以扩展 μC/OS - II 的功能，使它的可以执行的任务数超过目前的 63 个。但在这里，只需简单地将任务的 id 值设置成与任务的优先级一样就可以了。

pbos　指向任务堆栈栈底的指针，用于堆栈的检验。

stk_size　用于指定堆栈的容量。也就是说，如果堆栈的入口宽度为 4 B，那么 stk_size 为 1 000，是指堆栈有 4 000 B。该参数与 pbos 一样，也用于堆栈的检验。

pext　指向用户附加的数据域的指针，用来扩展任务的任务控制块 OS_TCB。例如，可以给每个任务取一个名字(参看第 1 章的范例 3)；可以在任务切换过程中，将浮点寄存器的内容储存到这个附加数据域中(参看第 1 章的范例 4)；也可将某 I/O 口地址的值储存到这个附加数据域中，用于在任务切换过程中触发示波器，等等。

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 0，任务是否要进行浮点操作，等等。uCOS_II.H 文件中有一个包含所有的可能选项(OS_TASK_OPT_STK_CHK、OS_TASK_OPT_STK_CLR 以及 OS_TASK_OPT_SAVE_FP)的常数表。每个选项占用 opt 的 1 位，并可以通过该位的置位来选定(在使用时，只需将以上的 OS_TASK_OPT_??? 选项常数进行位"或"(OR)操作就可以了)。

L4.2(1)　OSTaskCreateExt()开始先检测分配给任务的优先级是否有效。任务的优先级必须为 0～OS_LOWEST_PRIO，包括 0 和 OS_LOWEST_PRIO。请再次注意优先级 OS_LOWEST_PRIO 是为 μC/OS - II 空闲任务保留的。因为在执行系统初始化函数 OSInit()时，已经把优先级 OS_LOWEST_PRIO 分配给空闲任务，所以在应用程序中不能调用 OSTaskCreateExt()函数来创建优先级为 OS_LOWEST_PRIO 的任务。假如试图这样做，OSTaskCreateExt()会返回代码 OS_PRIO_EXIST。

L4.2(2)　OSTaskCreateExt()须确认指定的优先级上是否有已经建立了的任务。μC/OS - II 中，任何 2 个任务的优先级都不能相同。

L4.2(3)　如果拟分配的优先级是空闲的，那么 μC/OS - II 通过在 OSTCBPrioTbl[]中放置一个非空指针，占用该优先级。

L4.2(4) 这样,使得OSTaskCreateExt()在设定任务数据结构的其他部分时,可以重新开中断。

L4.2(5) 为了检验任务的堆栈(参看4.03节:"堆栈检验,OSTaskStkChk()"),必须在opt参数中置OS_TASK_OPT_STK_CHK标志为1。检验堆栈还要求在任务建立时,堆栈中存储的值都是0(堆栈是清0的)。为了在任务建立时将堆栈清0,需要在opt参数中置OS_TASK_OPT_STK_CLR为1。当以上2个标志都设置好以后,OSTaskCreateExt()才能将堆栈清0。

注意:在这里调用了memset()函数,因为这是一个标准的ANSI函数,编译器商家会使之最优化。

L4.2(6) OSTaskCreateExt()调用OSTaskStkInit(),它负责初始化任务的堆栈。该函数是与处理器的硬件体系相关的函数,可以在OS_CPU_C.C文件中找到。有关实现OSTaskStkInit()的细节,可参看第13章:"移植μC/OS-II"。如果已经有人在读者打算用的处理器上成功地移植了μC/OS-II,而又能得到其源代码,就不必深究该函数的实现细节了。OSTaskStkInit()函数返回新的堆栈栈顶(psp),并保存在任务控制块OS_TCB中。μC/OS-II支持的处理器的堆栈,既可以从上(高地址)往下(低地址)递减,也可以从下往上递增(参看4.02节:"任务堆栈")。在调用OSTaskCreateExt()时,必须知道堆栈是递增的,还是递减的(见所用处理器的OS_CPU.H文件),因为必须把堆栈的栈顶传递给OSTaskCreateExt(),而栈顶可能是堆栈的最低地址(OS_STK_GROWTH为0时),也可能是最高地址(当OS_STK_GROWTH为1时)。

L4.2(7) 且OSTaskStkInit()函数完成了堆栈的初始化,OSTaskCreateExt()就调用OS_TCBInit(),从空闲的OS_TCB缓冲池中获得并初始化一个任务控制块OS_TCB。OS_TCBInit()的代码请参见3.03节:"任务控制块(OS_TCB)"。

L4.2(8)

L4.2(9) 从OS_TCBInit()返回后,OSTaskCreateExt()会检测返回的代码。如果成功,就给任务计数器OSTaskCtr加1。OSTaskCtr用于跟踪已建立的任务数目。

L4.2(12) 如果OS_TCBInit()返回失败,那么函数OSTaskCreateExt()会通过设置OSTCBPrioTbl[prio]的入口为0,而放弃该任务的优先级。

L4.2(10)

L4.2(11) 如果OSTaskCreateExt()函数是在某个任务的执行过程中被调用的(OSRunning已置为True),那么多任务调度函数会被调用,以判断新建立的任务是否比原来任务的优先级高。建立一个优先级更高的任务,会使内核进行一次从旧任务到新任务的任务切换。如果在多任务调度开始之前(还没有调用过OSStart())建立新任务,则任务调度函数不会起作用。

4.02 任务堆栈

每个任务都有自己的堆栈空间。堆栈必须声明为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(在编译时分配),也可以动态分配堆栈空间(在运行时分配)。静态堆栈声明如程序清单 L4.3 和 L4.4 所列。这 2 种声明方式都应放置在函数的外面。

程序清单 L4.3　静态堆栈

```
static OS_STK  MyTaskStack[stack_size];
```

或

程序清单 L4.4　静态堆栈

```
OS_STK  MyTaskStack[stack_size];
```

可以用 C 编译器提供的 malloc()函数,动态地分配堆栈空间,如程序清单 L4.5 所列。在动态分配中,要时刻注意内存碎片问题。特别是当反复地建立和删除任务时,内存堆中可能会出现大量的内存碎片,导致没有足够大的一块连续内存区域可用做任务堆栈,这时 malloc()便无法成功地为任务分配堆栈空间。

程序清单 L4.5　用 malloc()为任务分配堆栈空间

```
OS_STK   *pstk;



pstk = (OS_STK *)malloc(stack_size);
if (pstk != (OS_STK *)0) {              /* 确认 malloc()能得到足够的内存空间 */
    Create the task;
}
```

图 F4.1 表示一块能被 malloc()动态分配的 3 KB 的内存碎片。

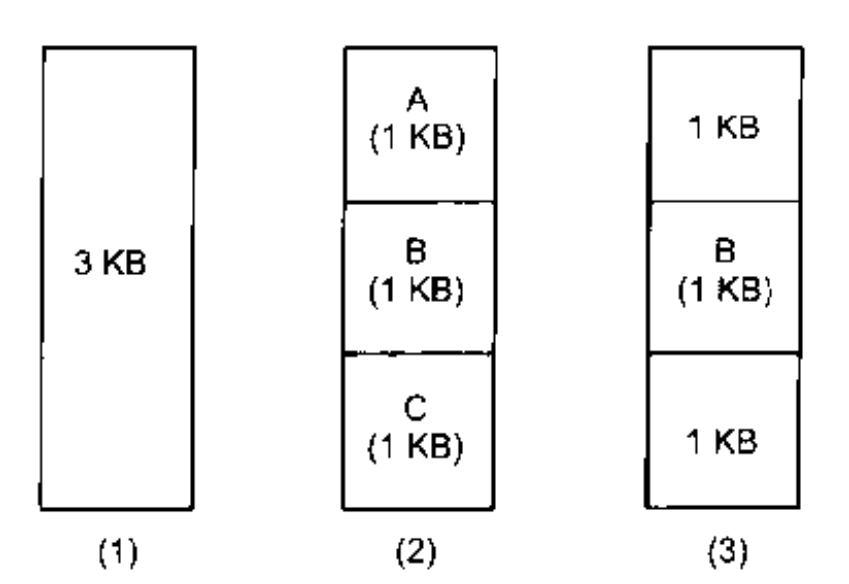


图 F4.1　内存碎片

F4.1(1)　为了讨论问题方便,假定要建立 3 个任务(任务 A,B 和 C),每个任务需要 1 KB 的空间。

F4.1(2) 设第1个1 KB给任务A，第2个1 KB给任务B，第3个1 KB给任务C。

F4.1(3) 应用程序删除任务A和任务C，用free()函数释放内存到内存堆中。现在，内存堆虽有2 KB的自由内存空间，但却是不连续的，所以不能建立另一个需要2 KB内存的任务（任务D）。但是，如果从未删除任何任务，那么使用malloc()分配内存空间就是可行的。

μC/OS-II支持的处理器的堆栈既可以是递减的，也可以是递增的。在调用函数OSTaskCreate()或OSTaskCreateExt()时，必须知道堆栈是递减的，还是递增的，因为必须把堆栈的栈顶地址传递给以上2个函数。当OS_CPU.H文件中的OS_STK_GROWTH为0时，需将堆栈的最低地址传递给任务创建函数，如程序清单L4.6所列。

程序清单L4.6　堆栈从下往上递增

```
OS_STK  TaskStack[TASK_STACK_SIZE];

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

当OS_CPU.H文件中的OS_STK_GROWTH置为1时，需将堆栈的最高内存地址传递给任务创建函数，如程序清单L4.7所列。

程序清单L4.7　堆栈从上往下递减

```
OS_STK  TaskStack[TASK_STACK_SIZE];

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

这个问题会影响代码的可移植性。如果想将代码从支持往下递减堆栈的处理器中移植到支持往上递增堆栈的处理器中，需使代码同时适应以上2种情况。在这种特殊情况下，程序清单L4.6和L4.7可重新写成如程序清单L4.8所列的形式。

程序清单L4.8　对递减和递增的堆栈都提供支持

```
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
```

任务所需的堆栈的容量是由应用程序确定的。在确定堆栈容量时，必须考虑到任务调用的所有函数的嵌套情况、任务调用的所有函数为局部变量分配的所有内存的数目，以及所有可

能的中断服务子程序嵌套对堆栈的需求。另外，堆栈必须能保存 CPU 所有的寄存器。

4.03　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栈空间，从而减少应用程序代码所需的 RAM（内存）数量。μC/OS－II 具有的 OSTaskStkChk()函数可以提供这种有价值的信息。

要使用 μC/OS－II 的堆栈检验功能，须做以下几件事情：

- 在 OS_CFG.H 文件中置 OS_TASK_CREATE_EXT 为 1。
- 用 OSTaskCreateExt()建立任务，并赋予该任务比实际需要多一些的栈空间。可以在任何任务中调用 OSTaskStkChk()函数，对任何以 OSTaskCreateExt()建立的任务进行堆栈检验。
- 在 OSTaskCreateExt()中，将参数 opt 设置为：
  OS_TASK_OPT_STK_CHK＋OS_TASK_OPT_STK_CLR

  注意：如果程序启动代码已将所有的 RAM 清 0，并且从未删除过任何任务，那么就不必设置选项 OS_TASK_OPT_STK_CLR。这样就会缩短 OSTaskCreateExt()函数的执行时间。
- 把需要进行堆栈检验的任务的优先级作为 OSTaskStkChk()的参数并调用。可以检验任何任务的堆栈，并不是只能检验正在运行的任务。

图 F4.2 所示为堆栈检验。

F4.2(1)　图 F4.2 中，假定堆栈是从高地址内存向低地址内存变化（OS_STK_GROWTH 被置为 1），但以下的讨论也同样适用于从下往上递增的堆栈。μC/OS－II 通过查看堆栈本身的内容，决定堆栈方向。只有内核或是任务发出堆栈检验的命令时，堆栈检验才会执行，它不会不断地去检验任务堆栈的使用情况。

F4.2(2)　在堆栈检验时，μC/OS－II 要求在任务建立时，堆栈中存储的必须是 0。

F4.2(3)

F4.2(4)　另外，μC/OS－II 还须知道堆栈栈底（BOS）的位置和分配给任务的堆栈的容量。在任务建立时，这 2 个值储存在任务的任务控制块 OS_TCB 中，但该任务只能是由 OSTaskCreateExt()函数建立的。

F4.2(5)　OSTaskStkChk()顺着堆栈的栈底开始计算空闲的堆栈空间大小，具体实现方法是统计储存值为 0 的连续堆栈入口的数目，直到发现储存值不为 0 的堆栈入口。

注意：堆栈入口的储存值在进行检验时，使用的数据类型是堆栈型（参见 OS_CPU.H 中的 OS_STK）。换句话说，如果堆栈的入口有 32 位宽，对 0 值的比较也是按 32 位完成的。

F4.2(6)

F4.2(8)　已使用了的堆栈空间数目是指，从在 OSTaskCreateExt()中定义的堆栈容量中减去储存值为 0 的连续堆栈入口的数目以后的大小。OSTaskStkChk()实

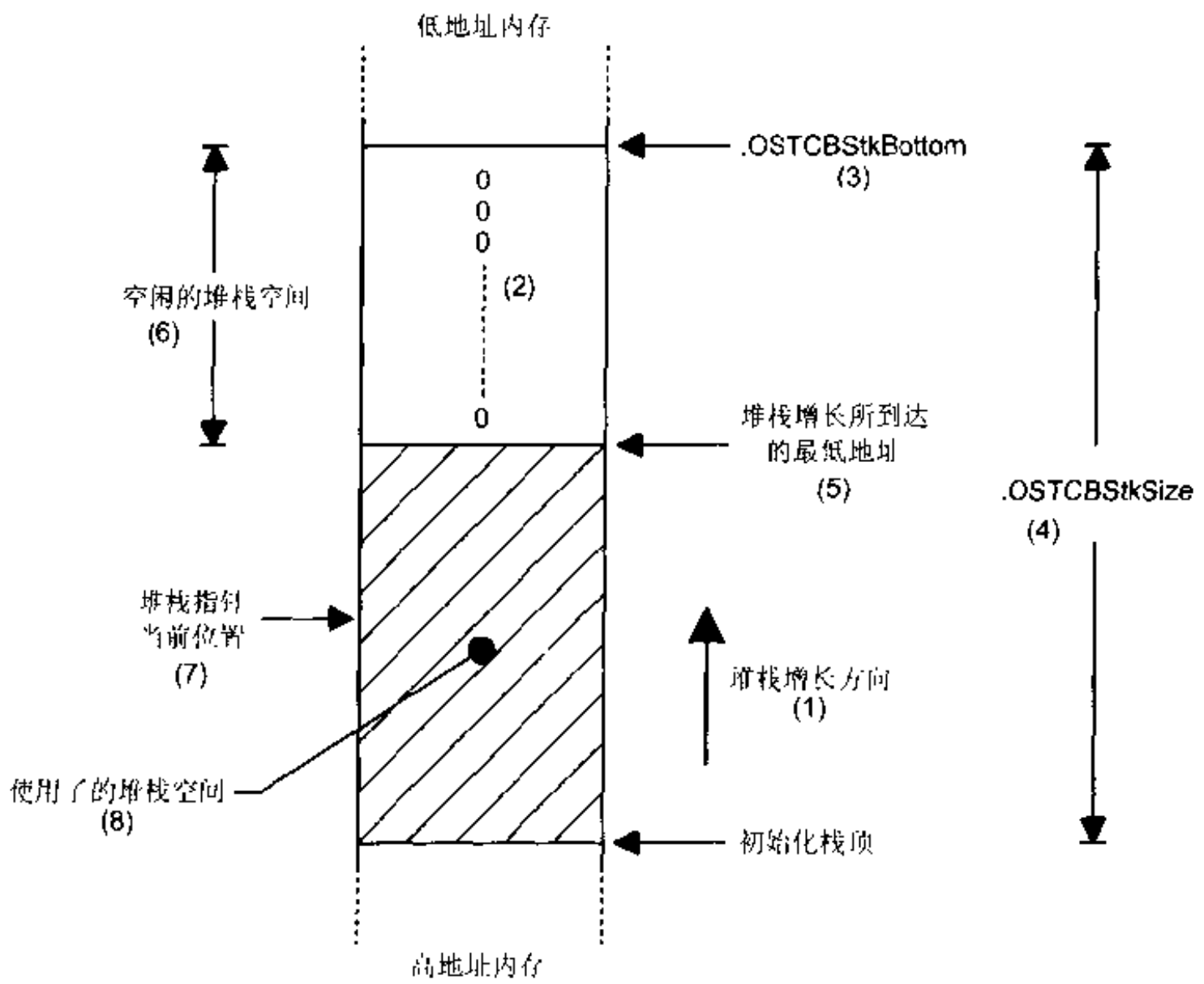


图 F4.2 堆栈检验

际上把空闲堆栈的字节数和已用堆栈的字节数放置在 OS_STK_DATA 数据结构中(参见 μC/OS_II.H)

F4.2(7) 注意:在某个给定的时间,被检验的任务的堆栈指针可能会指向最初的堆栈栈顶(TOS)与堆栈最深处之间的任何位置。

F4.2(5) 每次在调用 OSTaskStkChk()时,也可能会因为任务还没触及堆栈的最深处而得到不同的堆栈的空闲空间数。

应使自己的应用程序运行足够长的时间,并且经历最坏的堆栈使用情况,这样才能得到正确的数目。一旦 OSTaskStkChk()提供给最坏情况下的堆栈需求,就可以重新设置堆栈的最终大小了。为了适应系统以后的升级和扩展,应该多分配 10%~100%的堆栈空间。在堆栈检验中,所得到的只是一个大致的堆栈使用情况,并不能说明堆栈使用的全部实际情况。

OSTaskStkChk()函数的代码如程序清单 14.9 所列。OS_STK_DATA 数据结构(参见 μC/OS_II.H)用来保存有关任务堆栈的信息。决定使用一个数据结构来做此事,是有 2 方面的原因的:第 1,把 OSTaskStkChk()当作查询类型的函数,并且使所有的查询类型的函数用同样的方法返回值,即返回查询数据到某个数据结构中;第 2,在数据结构中传递数据的效率高,且可以在不改变 OSTaskStkChk()的 API(应用程序编程接口)的条件下,为该数据结构增加其他域。现在,OS_STK_DATA 只包含 2 个域:OSFree 和 OSUsed。从代码中可看到,可以通过指定要执行堆栈检验的任务的优先级,调用 OSTaskStkChk()。

程序清单 L4.9　堆栈检验函数

```
INT8U  OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    OS_TCB      *ptcb;
    OS_STK      *pchk;
    INT32U       free;
    INT32U       size;


#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {                    (1)
        return (OS_PRIO_INVALID);
    }
#endif
    pdata->OSFree = 0;
    pdata->OSUsed = 0;
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                                             (2)
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb == (OS_TCB *)0) {                                              (3)
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);
    }
    if ((ptcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) == 0) {                      (4)
        OS_EXIT_CRITICAL();
        return (OS_TASK_OPT_ERR);
    }
    free = 0;                                                               (5)
    size = ptcb->OSTCBStkSize;
    pchk = ptcb->OSTCBStkBottom;
    OS_EXIT_CRITICAL();
#if OS_STK_GROWTH == 1
    while (*pchk++ == (OS_STK)0) {
        free++;
    }
#else
    while (*pchk-- == (OS_STK)0) {
```

```
        free++;
    }
#endif
    pdata->OSFree = free * sizeof(OS_STK);                                    (6)
    pdata->OSUsed = (size - free) * sizeof(OS_STK);
    return (OS_NO_ERR);
}
```

L4.9(1) 假如 OS_CFG 中的常数 OS_ARG_CHK_EN 被设置为 1，OSTaskStkChk()会检验定义的优先级是否在有效的范围之内。

L4.9(2) 如果定义了 OS_PRIO_SELF，就表明用户想知道当前任务的堆栈信息。

L4.9(3) 显然，要执行堆栈检验的任务必须存在。只要检验到 OSTCBPrioTbl[]里存在一个非空指针，就可以确定任务存在。

L4.9(4) 要执行堆栈检验，必须已用 OSTaskCreateExt()建立了任务，并且已经传递了选项 OS_TASK_OPT_STK_CHK。如果调用函数 OSTaskStkChk()对用 OSTaskCreate()（而不是 OSTaskCreateExt()）建立的任务进行堆栈检验，那么因为参数 opt（会传递给 OS_TCBInit()）为 0，所以检验会失败。

L4.9(5) 如果所有的条件都满足了，OSTaskStkChk()就会像前面描述的那样，从堆栈栈底开始统计堆栈的空闲空间，直到发现一个储存值非 0 的堆栈入口。

L4.9(6) 储存在 OS_STK_DATA 中的信息被确定下来。

注意：函数所确定的是堆栈的实际空闲字节数和已被占用的字节数，而不是堆栈的总字节数。当然，堆栈的实际大小（用字节表示）就是这 2 项之和。

4.04 删除任务，OSTaskDel()

有时需要删除任务。删除任务，是说任务将返回并处于休眠状态（参见 3.02 节："任务状态"），并不是说任务的代码被删除了，只是任务的代码不再被 μC/OS-II 调用。通过调用 OSTaskDel()，就可以完成删除任务的功能（如程序清单 L4.10 所列）。

程序清单 L4.10 删除任务

```
INT8U  OSTaskDel (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR       cpu_sr;
#endif

#if OS_EVENT_EN > 0
    OS_EVENT        *pevent;
#endif
```

```
#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
    OS_FLAG_NODE  *pnode;
#endif
    OS_TCB        *ptcb;
    BOOLEAN        self;


    if (OSIntNesting > 0) {                                                    (1)
        return (OS_TASK_DEL_ISR);
    }
#if OS_ARG_CHK_EN > 0
    if (prio == OS_IDLE_PRIO) {                                                (2)
        return (OS_TASK_DEL_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {                      (3)
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                                                (4)
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {                                                 (5)
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {            (6)
            OSRdyGrp &= ~ptcb->OSTCBBitY;
        }
#if OS_EVENT_EN > 0
        pevent = ptcb->OSTCBEventPtr;                                          (7)
        if (pevent != (OS_EVENT *)0) {
            if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
            }
        }
#endif
#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
        pnode = ptcb->OSTCBFlagNode;                                           (8)
        if (pnode != (OS_FLAG_NODE *)0) {
            OS_FlagUnlink(pnode);
        }
#endif
```

```
    ptcb->OSTCBDly  = 0;                                          (9)
    ptcb->OSTCBStat = OS_STAT_RDY;                                (10)
    if (OSLockNesting < 255) {                                    (11)
        OSLockNesting++;
    }
    OS_EXIT_CRITICAL();                                           (12)
    OS_Dummy();                                                   (13)
    OS_ENTER_CRITICAL();
    if (OSLockNesting > 0) {                                      (14)
        OSLockNesting--;
    }
    OSTaskDelHook(ptcb);                                          (15)
    OSTaskCtr--;                                                  (16)
    OSTCBPrioTbl[prio] = (OS_TCB *)0;                             (17)
    if (ptcb->OSTCBPrev == (OS_TCB *)0) {                         (18)
        ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
        OSTCBList                  = ptcb->OSTCBNext;
    } else {
        ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;
        ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;
    }
    ptcb->OSTCBNext = OSTCBFreeList;                              (19)
    OSTCBFreeList   = ptcb;
    OS_EXIT_CRITICAL();
    OS_Sched();                                                   (20)
    return (OS_NO_ERR);
  }
  OS_EXIT_CRITICAL();
  return (OS_TASK_DEL_ERR);
}
```

L4.10(1) OSTaskDel()应确保不是在中断服务子程序ISR中试图删除一个任务，因为这是不允许的。

L4.10(2) OSTaskDel()还应确保所要删除的任务不是空闲任务，因为这也是不允许的。

L4.10(3) 可以删除统计任务(OS_LOWEST_PRIO-1)和所有具有更高优先级的任务(也就是说，其他任务的优先级数值相比统计任务的优先级数值应该小一些)。

L4.10(4) 调用该函数的任务可以通过定义参数OS_PRIO_SELF删除自己。

L4.10(5) OSTaskDel()应确保被删除的任务是存在的。如果指定的参数是OS_PRIO_SELF，这一判断过程自然是可以通过的。笔者没有为这种情况单独写一段代码，因为这样只会增加代码，并延长程序的执行时间。假如指

定的参数是OS_PRIO_SELF，OSTaskDel()须做的工作只是取得当前任务的优先级。任务优先级被保存在任务的任务控制块 OS_TCB 中。

所有的条件都满足后，OSTaskDel()就会从所有可能的 μC/OS - II 的数据结构中去掉任务的任务控制块 OS_TCB。OSTaskDel()分 2 步完成该去除操作，以缩短中断响应时间。

L4.10(6)　首先，如果任务处于就绪表中，就会从就绪表中被去除。

L4.10(7)　如果任务处于互斥型信号量、邮箱、消息队列或信号量的等待表中，它就从自己所处的表中被去除。

L4.10(8)　如果任务处于事件标志的等待表中，就会从此表中被去除。

L4.10(9)　接着，OSTaskDel()将任务的时钟节拍延迟数清 0，以确保自己重新开中断时，中断服务子程序 ISR 不会使该任务就绪。

L4.10(10)　OSTaskDel()置任务的 .OSTCBStat 标志为 OS_STAT_RDY。

注意：OSTaskDel()并不试图使任务处于就绪状态，而只是阻止其他任务或中断服务子程序 ISR 让该任务重新开始执行(避免其他任务或 ISR 调用 OSTaskResume())。

这种情况是可能发生的，因为 OSTaskDel()会重新开中断，而 ISR 可以让更高优先级的任务处于就绪状态，这就有可能会使打算删除的任务重新开始执行。若不设置任务的 .OSTCBStat 标志为 OS_STAT_RDY，也可以清除 OS_STAT_SUSPEND 位，达到相同的目的(这样代码可能显得更清楚、更容易理解一些)，但这样做会使得处理时间稍长一些。

L4.10(11)　至此，被删除的任务就不会被其他的任务或中断服务子程序置于就绪状态了。因为该任务已从就绪任务表中被删除了，它不是在等待什么事件的发生，也不是在等待延时时间到，不能重新开始执行。为了达到删除任务的目的，任务被置于休眠状态。正因为任务处于休眠状态，OSTaskDel()须防止任务调度程序在删除过程中切换到其他的任务中去。因为如果当前任务正在被删除，它是不可能被再次调度的。

L4.10(12)　OSTaskDel()重新开中断，以缩短中断响应时间。这样，OSTaskDel()就能处理中断了；但由于给 OSLockNesting 加了 1，ISR 执行完后，会返回到被中断了的任务，从而继续任务的删除工作。

注意：OSTaskDel()此时还没有完全完成删除任务的工作，因为它还需要从任务控制块 TCB 链中释放该任务的任务控制块 OS_TCB，并将 OS_TCB退回到空闲 OS_TCB 表中。

L4.10(13)　另外需要注意的是，在调用 OS_EXIT_CRITICAL()函数后，马上调用了 OS_Dummy()，该函数并不会进行任何实质性的操作。这样做的目的，只是想保证处理器在中断开着的情况下至少执行一条指令。对于许多处理器来说，执行完开中断指令后，CPU 会再强制执行下一条指令，然后才开中断。也就是说，直到执行完下一条指令后，才会真正开中断。Intel 80x86 和 Zilog Z - 80 处理器就是如此工作的。开中断后马上关中断，就等于从来没

开过中断，这只会延长中断的响应时间；因此，调用 OS_Dummy()，确保在再次禁止中断之前，至少执行了一条调用指令和一条返回指令。当然，可以用宏定义将 OS_Dummy()定义为一个空操作指令，这样调用 OS_Dummy()就等于执行了一个空操作指令，会使 OSTaskDel()的执行时间稍微缩短一点；但笔者认为，这种宏定义是没有价值的，因为它会增加移植 μC/OS-II 的工作量。

L4.10(14) OSTaskDel()可以继续执行删除任务的操作了。在 OSTaskDel()重新关中断后，它通过把锁定嵌套计数器(OSLockNesting)减1，重新允许任务调度。

L4.10(15) OSTaskDel()调用自定义的删除任务接口函数 OSTaskDelHook()，可以在这里删除或释放自定义的 TCB 附加数据域。

L4.10(16) OSTaskDel()给 μC/OS-II 的任务计数器减1，表示 μC/OS-II 管理的任务减少了1个。

L4.10(17) OSTaskDel()简单地将指向被删除任务的任务控制块 OS_TCB 的指针置为 NULL，就从优先级表中把 OS_TCB 去除了。

L4.10(18) OSTaskDel()会从以 OSTCBList 开头的 OS_TCB 双向链表中把被删除任务的任务控制块 OS_TCB 去掉。

注意：在这里没有必要检验 ptcb->OSTCBNext == 0 的情况。因为 OSTaskDel()不能删除空闲任务，而恰恰是空闲任务处于链表的末端。

L4.10(19) 被删除任务的任务控制块 OS_TCB 被退回到空闲 OS_TCB 表中，以供建立其他任务时使用。

L4.10(20) 调用任务调度程序，查看在 OSTaskDel()重新开中断时[L4.10(12)]，中断服务子程序 ISR 是否曾使更高优先级的任务进入了就绪状态。

4.05 请求删除任务，OSTaskDelReq()

有时候，任务会占用一些内存缓冲区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。这会导致存储器漏洞，而这是任何嵌入式系统都无法接受的。在这种情况下，须想办法让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。可以通过 OSTaskDelReq()函数完成该功能。发出删除任务请求的任务和打算被删除的任务都须调用 OSTaskDelReq()函数。发出删除任务请求的任务的代码如程序清单 L4.11 所列。

程序清单 L4.11 请求删除其他任务的任务的代码

```
void RequestorTask (void *pdata)
{
    INT8U err;
```

```
    pdata = pdata;
    for (;;) {
        /* 应用程序代码 */
        if ('TaskToBeDeleted()'需要被删除) {                                    (1)
            while (OSTaskDelReq(TASK_TO_DEL_PRIO) != OS_TASK_NOT_EXIST) {      (2)
                OSTimeDly(1);                                                   (3)
            }
        }
        /* 应用程序代码 */                                                      (4)
    }
}
```

L4.11(1)　发出删除任务请求的任务须决定，在什么情况下发出删除任务的请求。也就是说，应用程序须决定删除任务的条件。

L4.11(2)　如果任务的确需要被删除，则可以通过传递被删除任务的优先级，调用 OSTaskDelReq()函数。如果要被删除的任务不存在(任务已被删除或是还没建立)，OSTaskDelReq()会返回 OS_TASK_NOT_EXIST，即该任务不存在。如果 OSTaskDelReq()的返回值为 OS_NO_ERR，则表明请求已被接受，但任务还没被删除。用户可能希望发出删除任务请求的任务在等到要被删除的任务确实删除了自己以后，才继续进行下面的工作。

L4.11(3)　这时用户可以像笔者一样，通过让发出删除任务请求的任务延迟一定时间，达到这个目的。这里只延迟了 1 个时钟节拍，如果需要，用户可以延迟得更长一些。

L4.11(4)　当要被删除的任务完全删除自己后，程序清单 L4.11(2)中返回的值会成为 OS_TASK_NOT_EXIST，此时循环结束。

需要删除自己的任务的代码如程序清单 L4.12 所列。在任务控制块 OS_TCB 中存有一个标志，任务通过查询这个标志的值，确认自己是否需要被删除。通过调用 OSTaskDelReq((OS_PRIO_SELF)，可以得到这个标志的值。

程序清单 L4.12　需要删除自己的任务

```
void TaskToBeDeleted (void *pdata)
{
    INT8U err;


    pdata = pdata;
    for (;;) {
        /* 应用程序代码 */
        if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ) {                    (1)
```

```
            释放所有占用的资源;                                          (2)
            释放所有动态内存;
            OSTaskDel(OS_PRIO_SELF);                                    (3)
        } else {
            /* 应用程序代码 */
        }
    }
}
```

L4.12(1) 当 OSTaskDelReq()返回给调用者的值为 OS_TASK_DEL_REQ 时，表明已经有另外的任务请求删除该任务了。

L4.12(2)

L4.12(3) 在这种情况下，被删除的任务会释放它所拥有的所用资源，并且调用 OSTaskDel(OS_PRIO_SELF)删除自己。前面曾提到，任务的代码没有被真正删除，而只是 μC/OS-II 不再理会该任务代码。换句话说，就是任务的代码不会再运行了；但是，可以通过调用 OSTaskCreate()或 OSTaskCreateExt()函数重新建立该任务。

OSTaskDelReq()的代码如程序清单 L4.13 所列。通常 OSTaskDelReq()须检查临界条件。

程序清单 L4.13 OSTaskDelReq()

```
INT8U  OSTaskDelReq (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    stat;
    INT8U      err;
    OS_TCB    *ptcb;


#if OS_ARG_CHK_EN > 0
    if (prio == OS_IDLE_PRIO) {                                         (1)
        return (OS_TASK_DEL_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {               (2)
        return (OS_PRIO_INVALID);
    }
#endif
    if (prio == OS_PRIO_SELF) {                                         (3)
```

```
        OS_ENTER_CRITICAL();
        stat = OSTCBCur->OSTCBDelReq;
        OS_EXIT_CRITICAL();
        return (stat);
    }
    OS_ENTER_CRITICAL();
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {                                          (4)
        ptcb->OSTCBDelReq = OS_TASK_DEL_REQ;                            (5)
        err               = OS_NO_ERR;
    } else {
        err               = OS_TASK_NOT_EXIST;                          (6)
    }
    OS_EXIT_CRITICAL();
    return (err);
}
```

L4.13(1)　如果需要被删除的任务是空闲任务，那么 OSTaskDelReq()会报错并返回。

L4.13(2)　须保证调用者请求删除的任务的优先级是有效的。

L4.13(3)　如果调用者就是被删除的任务本身，存储在 OS_TCB 中的标志将会作为返回值返回。

L4.13(4)

L4.13(5)　如果用优先级而不是用 OS_PRIO_SELF 指定任务，并且任务是存在的，那么 OSTaskDelReq()就会设置任务的内部标志。

L4.13(6)　如果任务不存在，OSTaskDelReq()则会返回 OS_TASK_NOT_EXIST，表明该任务可能已经删除自己了。

4.06　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，可以通过调用函数 OSTaskChangePrio()，改变任务的优先级。换句话说，μC/OS-II 允许动态地改变任务的优先级。OSTaskChangePrio()的代码如程序清单 L4.14 所列。

程序清单 L4.14　OSTaskChangePrio()

```
INT8U  OSTaskChangePrio (INT8U oldprio, INT8U newprio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR       cpu_sr;
#endif
```

```
#if OS_EVENT_EN > 0
    OS_EVENT      *pevent;
#endif

    OS_TCB        *ptcb;
    INT8U          x;
    INT8U          y;
    INT8U          bitx;
    INT8U          bity;


#if OS_ARG_CHK_EN > 0
    if ((oldprio >= OS_LOWEST_PRIO && oldprio != OS_PRIO_SELF) ||                    (1)
        newprio >= OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[newprio] != (OS_TCB *)0) {                                      (2)
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    } else {
        OSTCBPrioTbl[newprio] = (OS_TCB *)1;                                         (3)
        OS_EXIT_CRITICAL();
        y    = newprio >> 3;                                                         (4)
        bity = OSMapTbl[y];
        x    = newprio & 0x07;
        bitx = OSMapTbl[x];
        OS_ENTER_CRITICAL();
        if (oldprio == OS_PRIO_SELF) {                                               (5)
            oldprio = OSTCBCur->OSTCBPrio;
        }
        ptcb = OSTCBPrioTbl[oldprio];
        if (ptcb != (OS_TCB *)0) {                                                   (6)
            OSTCBPrioTbl[oldprio] = (OS_TCB *)0;                                     (7)
            if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {                (8)
                if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {          (9)
                    OSRdyGrp &= ~ptcb->OSTCBBitY;
                }
                OSRdyGrp    |= bity;                                                 (10)
                OSRdyTbl[y] |= bitx;
```

```
#if OS_EVENT_EN > 0
            } else {
                pevent = ptcb->OSTCBEventPtr;
                if (pevent != (OS_EVENT *)0) {                                  (11)
                    if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                        pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
                    }
                    pevent->OSEventGrp     |= bity;                             (12)
                    pevent->OSEventTbl[y]  |= bitx;
                }
#endif
            }
            OSTCBPrioTbl[newprio] = ptcb;                                       (13)
            ptcb->OSTCBPrio       = newprio;                                    (14)
            ptcb->OSTCBY          = y;                                          (15)
            ptcb->OSTCBX          = x;
            ptcb->OSTCBBitY       = bity;
            ptcb->OSTCBBitX       = bitx;
            OS_EXIT_CRITICAL();
            OS_Sched();                                                         (16)
            return (OS_NO_ERR);
        } else {
            OSTCBPrioTbl[newprio] = (OS_TCB *)0;                                (17)
            OS_EXIT_CRITICAL();
            return (OS_PRIO_ERR);
        }
    }
}
```

L4.14(1)　不能改变空闲任务的优先级，但可以改变调用本函数的任务或者其他任务的优先级。为了改变调用本函数的任务的优先级，可以将参数 old priority 指定为该任务当前的优先级或 OS_PRIO_SELF，而 OSTaskChangePrio()会确定该任务的优先级。当然，还必须指定任务的新(想要的)优先级。

L4.14(2)　因为 μC/OS-II 不允许多个任务具有相同的优先级，所以 OSTaskChangePrio()需要检验新优先级是否空闲。

L4.14(3)　如果新优先级是空闲的，那么 μC/OS-II 会通过将一个非 0 指针储存到 OSTCBPrioTbl[newprio]中，保留这个优先级。如此就使得函数 OSTaskChangePrio()可以重新开中断。因为此时其他任务已经不可能建立拥有该优先级的任务了，也不能通过定义相同的新优先级来调用函数 OSTaskChangePrio()。

L4.14(4)　OSTaskChangePrio()会预先计算新优先级任务的任务控制块 OS_TCB 中的

某些值。使用这些值可将任务放入就绪表或从该表中移除任务(参见3.04:"就绪表(ready list)")。

L4.14(5) OSTaskChangePrio()检验当前任务是否想要改变自己的优先级。

L4.14(6) OSTaskChangePrio()检查想要改变优先级的任务是否存在。很明显,如果要改变优先级的任务就是当前任务,这个测试一定会成功。

L4.14(17) 如果OSTaskChangePrio()想要改变优先级的任务不存在,那么它必须将预留的新优先级放回到优先级表OSTCBPrioTbl[]中,并返回给调用者一个错误信息。

L4.14(7) OSTaskChangePrio()可以通过插入NULL指针,将指向当前任务OS_TCB的指针从优先级表中移除。这就使得当前任务的旧的优先级空闲,并可以被其他任务重新使用。

L4.14(8) 检验OSTaskChangePrio()想要改变优先级的任务是否已经就绪。

L4.14(9)

L4.14(10) 如果该任务处于就绪状态,那么必须在当前的优先级下,从就绪表中移除该任务,然后在新的优先级下,将该任务插入到就绪表中。需要注意的是,OSTaskChangePrio()是利用预先计算的值[L4.14(4)]将任务插入就绪表中的。

L4.14(11) 如果任务未就绪,那么它可能会正在等待一个信号量、一个互斥型信号量、一封邮件或是一个消息队列。如果OSTCBEventPtr非空(不等于NULL),OSTaskChangePrio()就会知道任务正在等待以上的某件事。

L4.14(12) 如果任务正在等待某一事件的发生,OSTaskChangePrio()必须将任务从事件控制块(参见第6章:"事件控制块")的等待队列(在旧的优先级下)中移除,并在新的优先级下将事件插入到等待队列中。任务也可能正在等待延时时间到(参见第5章:"时间管理"),或是被挂起(参见4.07节:"挂起任务,OSTaskSuspend()")。在这些情况下,L4.14(8)~L4.14(12)这几行可以省略。

L4.14(13) OSTaskChangePrio()将指向任务的OS_TCB的指针存到OSTCBPrioTbl[]中。

L4.14(14)

L4.14(15) 新的优先级保存在OS_TCB中,预先计算的值也保存在OS_TCB中。

L4.14(16) OSTaskChangePrio()完成了关键性的步骤后,在新的优先级高于旧的优先级或新的优先级高于调用本函数的任务的优先级的情况下,任务调度函数就会被调用。

4.07 挂起任务,OSTaskSuspend()

有时将任务挂起是很有用的。挂起任务可通过调用OSTaskSuspend()函数来完成。被挂起的任务只能通过调用OSTaskResume()函数来恢复。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操作,并且要继续等待延迟时间到,任务才能转入就绪状态。任务可以挂起自己或者其他任务。

OSTaskSuspend()函数的代码如程序清单 1.4.17 所列

程序清单 1.4.15　OSTaskSuspend()

```
INT8U  OSTaskSuspend (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN     self;
    OS_TCB     *ptcb;


#if OS_ARG_CHK_EN > 0
    if (prio == OS_IDLE_PRIO) {                                          (1)
        return (OS_TASK_SUSPEND_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {                (2)
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                                          (3)
        prio = OSTCBCur->OSTCBPrio;
        self = TRUE;
    } else if (prio == OSTCBCur->OSTCBPrio) {                            (4)
        self = TRUE;
    } else {
        self = FALSE;
    }
    ptcb = OSTCBPrioTbl[prio];                                           (5)
    if (ptcb == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_TASK_SUSPEND_PRIO);
    }
    if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {          (6)
        OSRdyGrp &= ~ptcb->OSTCBBitY;
    }
    ptcb->OSTCBStat |= OS_STAT_SUSPEND;                                  (7)
    OS_EXIT_CRITICAL();
    if (self == TRUE) {
        OS_Sched();                                                      (8)
```

```
    }
    return (OS_NO_ERR);
}
```

L4.15(1) OSTaskSuspend()要确保应用程序不是挂起空闲任务。

L4.15(2) 检查定义的优先级是否有效。记住,最大的、有效的优先级数值(最低的优先级)是 OS_LOWEST_PRIO。

注意:可以挂起统计任务(statistic)。

可能用户已经注意到了,第1个测试[L4.15(1)]在L4.15(2)中被重复了。这样做是为了能与 μC/OS 兼容。第1个测试可以删除并节省程序处理的时间,但这样做的意义不大,所以笔者决定留下它。

L4.15(3) OSTaskSuspend()检验是否通过定义 OS_PRIO_SELF 挂起来调用本函数的任务本身。在这种情况下,OSTaskSuspend()会从当前任务的任务控制块 OS_TCB中获得当前任务的优先级。

L4.15(4) 也可以通过指定优先级,挂起调用本函数的任务。在这2种情况下,任务调度程序都需要被调用。这就是为什么要定义局部变量 self 的原因。该变量在适当情况下会被测试。若没有挂起调用本函数的任务,OSTaskSus-pend()就没有必要运行任务调度程序,因为调用该函数的任务正在挂起一个优先级较低的任务。

L4.15(5) OSTaskSuspend()检验要挂起的任务是否存在。

L4.15(6) 如果该任务存在,它就会从就绪表中被去掉。

注意:要被挂起的任务有可能没有在就绪表中,因为它有可能在等待事件的发生或被延迟。

在这种情况下,要挂起的任务在 OSRdyTbl[]中对应的位已被清除(为0)。再次清除该位,要比先检验该位是否被清除了,若没被清除,然后再清除要快得多;所以,没有检验该位而直接将其清0。

L4.15(7) OSTaskSuspend()可以在任务的 OS_TCB 中设置 OS_STAT_SUSPEND 标志了,以表明任务现在被挂起了。

L4.15(8) 仅在被挂起的任务就是调用本函数的任务本身的情况下,OSTaskSuspend()才调用任务调度程序。

4.08 恢复任务,OSTaskResume()

在上一节中曾经提到,被挂起的任务只有通过调用函数 OSTaskResume()才能恢复。OSTaskResume()函数的代码如程序清单 L4.16 所列。

程序清单 L4.16　OSTaskResume()

```
INT8U  OSTaskResume (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR   cpu_sr;
#endif
    OS_TCB      *ptcb;


#if OS_ARG_CHK_EN > 0
    if (prio >= OS_LOWEST_PRIO) {                                          (1)
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb == (OS_TCB *)0) {                                             (2)
        OS_EXIT_CRITICAL();
        return (OS_TASK_RESUME_PRIO);
    }
    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) != 0x00) {                     (3)
        if (((ptcb->OSTCBStat &= ~OS_STAT_SUSPEND) == OS_STAT_RDY) &&      (4)
            (ptcb->OSTCBDly    == 0)) {                                    (5)
            OSRdyGrp               |= ptcb->OSTCBBitY;                     (6)
            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
            OS_EXIT_CRITICAL();
            OS_Sched();                                                    (7)
        } else {
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_TASK_NOT_SUSPENDED);
}
```

L4.16(1)　因为 OSTaskSuspend()不能挂起空闲任务，所以必须确认应用程序不是在恢复空闲任务。

注意：这个测试也可以确保不是在恢复优先级为 OS_PRIO_SELF 的任务(OS_PRIO_SELF 被定义为 0xFF，它总是比 OS_LOWEST_PRIO 大)。

恢复当前任务是没有任何意义的，因为当前任务肯定没有被挂起。

L4.16(2)

L4.16(3) 要恢复的任务必须是存在的，因为须操作它的任务控制块 OS_TCB，并且该任务必须是已被挂起的。

L4.16(4) OSTaskResume()是通过清除 OSTCBStat 域中的 OS_STAT_SUSPEND 位而取消挂起的。

L4.16(5) 要使任务处于就绪状态，OS_TCBDly 域必须为 0。这是因为在 OSTCBStat 中，没有任何标志表明任务正在等待延迟时间到。

L4.16(6) 只有当以上 2 个条件都满足的时候，任务才处于就绪状态。

L4.16(7) 任务调度程序会检查被恢复的任务拥有的优先级是否比调用本函数的任务的优先级高。

4.09 获得任务的信息，OSTaskQuery()

应用程序可以通过调用 OSTaskQuery()，来获得自身或其他应用任务的信息。实际上，OSTaskQuery()获得的是指定任务的任务控制块 OS_TCB 中内容的拷贝。能访问的 OS_TCB的数据域的多少，取决于应用程序的配置(参见 OS_CFG.H)。由于 μC/OS-II 是可裁剪的，它只包括应用程序所要求的属性和功能。

要调用 OSTaskQuery()，应用程序必须为 OS_TCB 分配存储空间，如程序清单 L4.17 所列。这个 OS_TCB 的数据空间与 μC/OS-II 分配给任务控制块 OS_TCB 的数据空间完全不同。在调用了 OSTaskQuery()后，这个 OS_TCB 包含了指定任务的任务控制块 OS_TCB 的副本。必须十分小心地处理 OS_TCB 中指向其他 OS_TCB 的指针(OSTCBNext 与 OSTCBPrev)，不要试图改变这些指针。一般来说，本函数只用于了解任务正在干什么 — 本函数实际是一个很有用的调试工具。

程序清单 L4.17 获得任务的信息

```
void MyTask (void *pdata)
{
    OS_TCB  MyTaskData;

    pdata = pdata;
    for (;;) {
        /* 用户代码                                  */
        err = OSTaskQuery(10, &MyTaskData);
        /* 检测错误代码..                            */
        /* 用户代码                                  */
    }
}
```

OSTaskQuery()的代码如程序清单 L4.18 所列。

程序清单 L4.18　OSTaskQuery()

```
INT8U  OSTaskQuery (INT8U prio, OS_TCB *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR   cpu_sr;
#endif
    OS_TCB      *ptcb;


#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {                    (1)
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                                             (2)
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb == (OS_TCB *)0) {                                              (3)
        OS_EXIT_CRITICAL();
        return (OS_PRIO_ERR);
    }
    memcpy(pdata, ptcb, sizeof(OS_TCB));                                    (4)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L4.18(1)　注意：允许查询所有的任务，包括空闲任务；但需特别注意的是，不要改变 OSTCBNext 与 OSTCBPrev 的指向。

L4.18(2)

L4.18(3)　通常，OSTaskQuery()需要检验用户想知道的是否是当前任务的有关信息，以及该任务是否已经建立了。

L4.18(4)　所有的域是通过赋值语句一次性复制的，而不是一个域一个域地复制。相比一个域一个域地复制或是相同结构复制，使用 memcpy()函数要快得多，因为编译器大多能够产生内存复制指令。

第 5 章

时间管理

在 3.11 节："时钟节拍"中曾提到，μC/OS－II（像大部分内核一样）要求提供定时中断，以实现延时与超时控制等功能。这个定时中断叫做时钟节拍，它应该发生 10～100 次/秒。时钟节拍的实际频率是由应用程序根据需要指定的。时钟节拍的频率越高，系统的负荷就越重。

在 3.10 节："μC/OS－II 中的中断"中，讨论了时钟的中断服务子程序 ISR 和时钟节拍函数 OSTimeTick——该函数用于通知 μC/OS－II，发生了时钟节拍中断。本章主要讲述 5 个可以处理时间问题的功能函数：

- OSTimeDly()；
- OSTimeDlyHMSM()；
- OSTimeDlyResume()；
- OSTimeGet()；
- OSTimeSet()。

本章所提到的这些函数，可以在 OS_TIME.C 文件中找到。

必须通过设置 OS_CFG.H 中的一些配置常量，才能使一些时间管理函数被允许使用。在表 T5.1 中，明确说明了一些函数的使用是由 OS_CFG.H 中配置常量的数值决定的。

表 T5.1　OS_CFG.H 中与时间管理相关的配置常量

μC/OS－II 的时间管理函数	在 OS_CFG.H 中，置 1 允许相应函数
OSTimeDly()	
OSTimeDlyHMSM()	OS_TIME_DLY_HMSM_EN
OSTimeDlyResume()	OS_TIME_DLY_RESUME_EN
OSTimeGet()	OS_TIME_GET_SET_EN
OSTimeSet()	OS_TIME_GET_SET_EN

5.00　任务延时函数，OSTimeDly()

μC/OS－II 提供了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，这个功能函数就是 OSTimeDly()。这段时间的长短是由指定的时钟节拍的数目来确定的。调用该函数会使 μC/OS－II 进行一次任务调度，从而去执行下一个优先级最高的就绪态任务。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，它就会立即进入就绪状态。

注意：只有当该任务在所有就绪任务中具有最高的优先级时，它才会立即运行。

程序清单 L5.1 所列为任务延时函数 OSTimeDly()的代码。应用程序是通过提供延时的时钟节拍数——一个 1～65 535 的数，调用该函数的。指定 0 值不进行任何延时。

程序清单 L5.1　OSTimeDly()

```
void OSTimeDly (INT16U ticks)
{
    if (ticks > 0) {                                                    (1)
        OS_ENTER_CRITICAL();
        if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {   (2)
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBDly = ticks;                                     (3)
        OS_EXIT_CRITICAL();
        OSSched();                                                      (4)
    }
}
```

L5.1(1)　假如指定一个 0 值，则表示不想对任务延时，函数会立即返回到调用该函数的任务。

L5.1(2)　指定一个非 0 值，会引起 OSTimeDly()从就绪表中移除当前任务。

L5.1(3)　延时节拍数会被保存在当前任务的任务控制块 OS_TCB 中，并且每隔 1 个时钟节拍，OSTimeTick()就会将这个延时节拍数减 1。用户一定注意到，调用该函数的任务并没有被放置到任何等待列表中。实际只要在 OSTCBDLY 中有一个非 0 值，就足够使 OSTimeTick()知道任务正在被延时。

L5.1(4)　既然任务已经不再处于就绪状态，任务调度程序会执行下一个优先级最高的就绪任务。

清楚地认识 0～1 个节拍之间的延时过程是非常重要的。换句话说，想延时 1 个时钟节拍，而实际上是在 0～1 个节拍之间结束延时。即使处理器的负荷不是很重，这种情况依然是存在的。图 F5.1 详细说明了整个过程。

F5.1(1)　系统每隔 10 ms 发生 1 次时钟节拍中断。

F5.1(2)　假如没有执行其他的中断并且此时的中断是开着的，时钟节拍中断服务子程序 ISR 就会被调用。

F5.1(3)　也许有好几个高优先级的任务(HPT)在等待延时期满，它们会接着执行。

F5.1(4)　低优先级任务(LPT)会得到执行的机会，该任务执行完后，马上调用图中所示的 OSTimeDly(1)。μC/OS-II 会使该任务处于休眠状态，直至下一个节拍的到来。

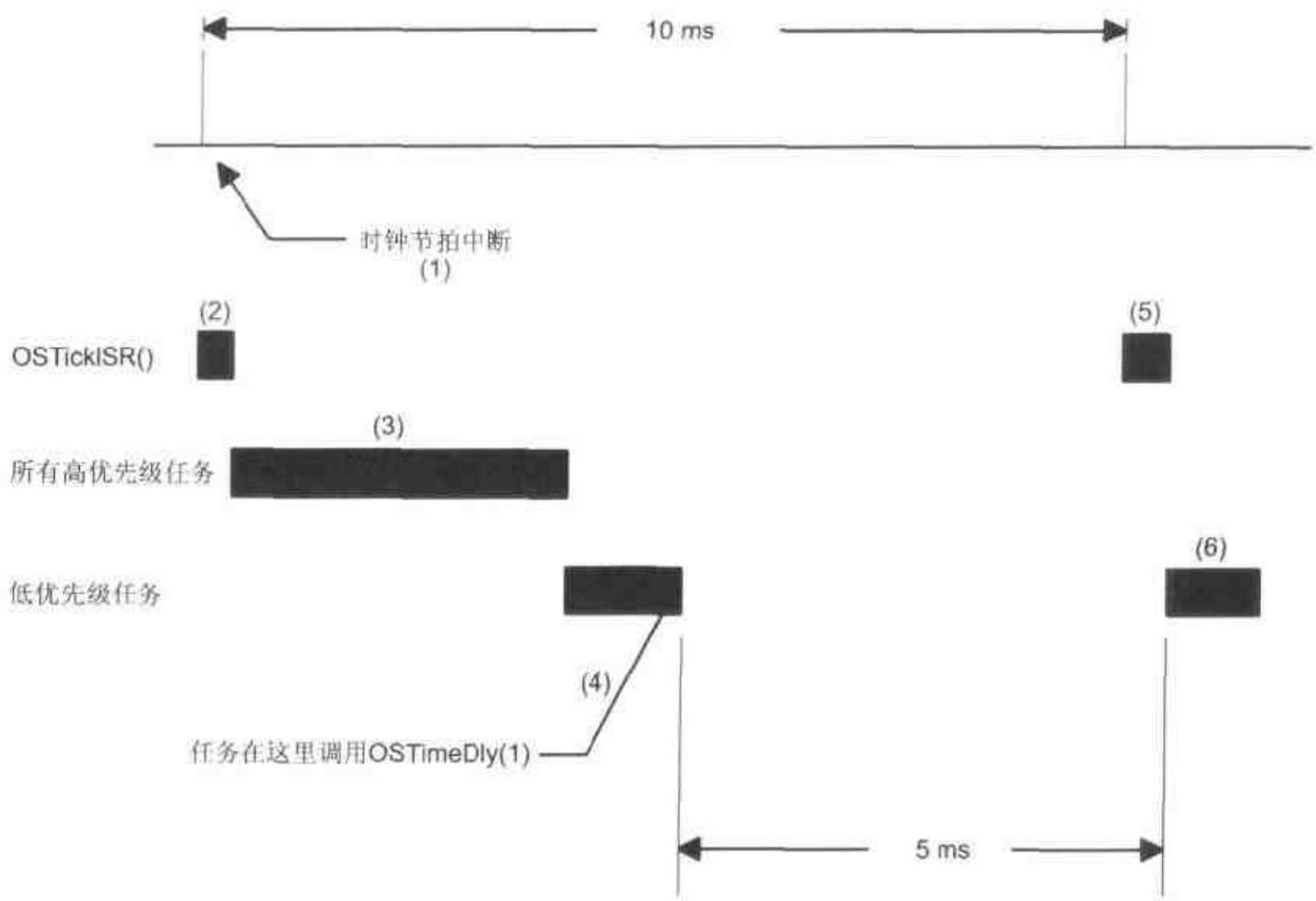


图 F5.1 延时详解

F5.1(5)

F5.1(6) 当下一个节拍到来后，时钟节拍中断服务子程序会被调用，但是这一次由于没有高优先级的任务需要执行，μC/OS-II会立即执行申请延时1个时钟节拍的任务。正如所看到的，该任务实际的延时少于1个节拍。在负荷很重的系统中，任务甚至有可能会在时钟中断即将发生时调用OSTimeDly(1)。在这种情况下，任务几乎就没有得到任何延时，因为任务马上又被重新调度了。如果应用程序至少需要延时1个节拍，必须要调用OSTimeDly(2)，指定延时为2个节拍。

5.01 按时、分、秒延时函数，OSTimeDlyHMSM()

OSTimeDly()虽是一个非常有用的函数，但应用程序须知道延时时间对应的时钟节拍的数目。可以使用全局常数OS_TICKS_PER_SEC(参看OS_CFG.H)声明一些定义，从而将时间转换成时钟段，如下所示：

```
#define  OS_TIME_100mS   (INT16U)((INT32U)OS_TICKS_PER_SEC * 100L / 1000L)
#define  OS_TIME_500mS   (INT16U)((INT32U)OS_TICKS_PER_SEC * 500L / 1000L)
#define  OS_TIME_2S      (INT16U)(OS_TICKS_PER_SEC * 2)
```

但这种处理方法有时显得比较愚笨。增加了 OSTimeDlyHMSM() 函数后，就可以按[小]时(h)、分(m)、秒(s)及毫秒(ms)来定义时间了，会显得更自然些。与 OSTimeDly() 一样，调用 OSTimeDlyHMSM() 函数也会使 μC/OS-II 进行一次任务调度，并且执行下一个优先级最高的就绪态任务。任务调用 OSTimeDlyHMSM() 后，一旦规定的时间期满，或者有其他任务通过调用 OSTimeDlyResume() 取消了对该任务的延时(参见 5.02："恢复延时的任务，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只有最高的优先级时，它才会开始运行。程序清单 L5.2 所列为 OSTimeDlyHMSM() 的代码。从中可以看出，应用程序是通过用[小]时、分、秒及毫秒指定延时调用该函数的。在实际应用中，应避免使任务延时过长的时间，因为从任务中获得一些反馈行为(减少计数器，清除 LED 等等)经常是很有意义的；但是，如果确实需要长时间延时，μC/OS-II 可以将任务延时长达 256 h(接近 11 天)。

程序清单 L5.2　OSTimeDlyHMSM()

```
INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U milli)
{
    INT32U ticks;
    INT16U loops;


    if (hours > 0 || minutes > 0 || seconds > 0 || milli > 0) {                    (1)
        if (minutes > 59) {
            return (OS_TIME_INVALID_MINUTES);
        }
        if (seconds > 59) {
            return (OS_TIME_INVALID_SECONDS);
        }
        If (milli > 999) {
            return (OS_TIME_INVALID_MILLI);
        }
        ticks = (INT32U)hours    * 3600L * OS_TICKS_PER_SEC                         (2)
              + (INT32U)minutes  *   60L * OS_TICKS_PER_SEC
              + (INT32U)seconds  *         OS_TICKS_PER_SEC
              + OS_TICKS_PER_SEC * ((INT32U)milli
              + 500L/OS_TICKS_PER_SEC) / 1000L;                                     (3)
        loops = ticks / 65536L;                                                     (4)
        ticks = ticks % 65536L;                                                     (5)
        OSTimeDly(ticks);                                                           (6)
        while (loops > 0) {                                                         (7)
            OSTimeDly(32768);                                                       (8)
            OSTimeDly(32768);
            loops--;
```

```
        }
        return (OS_NO_ERR);
    } else {
        return (OS_TIME_ZERO_DLY);                                    (9)
    }
}
```

L5.2(1) OSTimeDlyHMSM()一开始须检验是否为参数定义了有效的值。

L5.2(9) 与 OSTimeDly()一样，假如没有定义延时，OSTimeDlyHMSM()会立即返回调用它的函数。

因为 μC/OS-II 只能识别节拍数，而节拍总数是从指定的时间中计算出来的；所以很明显，程序清单 L5.2 中的程序并不是十分有效的。只是想通过这种方法告诉大家一个公式，这样就可以知道怎样计算总的节拍数了。真正有意义的只是 OS_TICKS_PER_SEC。

L5.2(3) 方程的这一部分计算出了与指定的毫秒数最接近的时钟节拍数。500/OS_TICKS_PER_SECOND的值基本上与 0.5 个节拍对应的毫秒数相同。例如，若将时钟频率(OS_TICKS_PER_SEC)设置成 100 Hz(10 ms)，那么 4 ms 的延时不会产生任何延时，而 5 ms 的延时却会引起 10 ms 的延时。

L5.2(4)

L5.2(5) μC/OS-II 支持的最长延时只有 65 535 个节拍。要想支持更长时间的延时，如 L5.2(2)所列，OSTimeDlyHMSM()须确定指定的延时对应的时钟节拍总数中含有多少个完整的 65 535 个节拍的延时，以及余下的节拍数为多少。例如，如果 OS_TICKS_PER_SEC 的值为 100，若想延时 15 min，那么 OSTimeDlyHMSM()会延时 15×60×100=90 000 个时钟。这个延时会被分割成 2 次32 768个节拍的延时(因为只能延时 65 535 个节拍，而不是 65 536 个节拍)和 1 次 24 464 个节拍的延时。

L5.2(6)

L5.2(7)

L5.2(8) 在这种情况下，OSTimeDlyHMSM()首先考虑剩下的节拍，然后是超过 65 535 的节拍数(2 个 32 768 个节拍延时)。

OSTimeDlyHMSM()的具体实现方法，对于调用 OSTimeDlyHMSM()而延时超过 65 535个节拍的任务，不能恢复(参见 5.02："恢复延时的任务，OSTimeDlyResume()")。也就是说，如果时钟节拍的频率是 100 Hz，则不能让调用 OSTimeDlyHMSM(0,10,55,350)的任务或延迟时间更长的任务来结束延时。

5.02　恢复延时的任务，OSTimeDlyResume()

延时的任务可以不等待延时期满，而是通过其他任务取消延时而使自己处于就绪态。这可以通过指定要恢复的任务的优先级来调用 OSTimeDlyResume()函数而完成。实际上，OSTimeDlyResume()也可以唤醒正在等待事件(参见第 7～11 章)的任务，但并不推荐使用这种方法。在这种情况下，等待事件发生的任务会把它当作等待超时处理。OSTimeDlyResume()的代码如程序清单 L5.3 所列。

程序清单 L5.3　恢复延时的任务

```
INT8U  OSTimeDlyResume (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR   cpu_sr;
#endif
    OS_TCB      *ptcb;

    if (prio >= OS_LOWEST_PRIO) {                                          (1)
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    ptcb = (OS_TCB *)OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {                                             (2)
        if (ptcb->OSTCBDly != 0) {                                         (3)
            ptcb->OSTCBDly = 0;                                            (4)
            if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY) {      (5)
                OSRdyGrp               |= ptcb->OSTCBBitY;                 (6)
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                OS_EXIT_CRITICAL();
                OS_Sched();                                                (7)
            } else {
                OS_EXIT_CRITICAL();
            }
            return (OS_NO_ERR);
        } else {
            OS_EXIT_CRITICAL();
            return (OS_TIME_NOT_DLY);
        }
    }
    OS_EXIT_CRITICAL();
    return (OS_TASK_NOT_EXIST);
}
```

L5.3(1) OSTimeDlyResume()须确保指定的任务优先级有效。

L5.3(2) OSTimeDlyResume()须确认要结束延时的任务是确实存在的。

L5.3(3) 如果任务存在,OSTimeDlyResume()会检验任务是否在等待延时期满。只要任务控制块 OS_TCB 的 .OSTCBDly 域中包含非 0 值,就表明任务正在等待延时期满,因为任务调用了 OSTimeDly(),OSTimeDlyHMSM()或其他在后面章节中所描述的 PEND 函数。

L5.3(4) 可以通过使 .OSTCBDly 为 0 而取消延时。

L5.3(5) 延时的任务有可能已被挂起了。然而任务只有在没有被挂起的情况下,才能处于就绪状态。

L5.3(6) 当上面的条件都满足后,任务就会被放在就绪表中。

L5.3(7) 这时,OSTimeDlyResume()会调用任务调度程序,看被恢复的任务是否拥有比当前任务更高的优先级。如果是,就会引起任务切换。

注意:任务有可能是通过暂时等待信号量、互斥型信号量、邮箱或消息队列来延时自己的(参见第 7~11 章)。可以简单地通过控制信号量、互斥型信号量、邮箱或消息队列,来恢复这样的任务。这种情况存在的惟一问题是,它要求用户分配事件控制块(参见 6.00 节:"将任务置于等待事件的任务列表"),因此应用程序会多占用一些 RAM。

5.03 系统时间,OSTimeGet()和 OSTimeSet()

无论时钟节拍何时发生,μC/OS-II 都会将一个 32 位的计数器加 1。这个计数器在调用 OSStart()初始化多任务时,被置为 0,并且在 4 294 967 295 个节拍执行完一遍时,会重新从 0 开始计数。

当时钟节拍的频率等于 100 Hz 时,这个 32 位的计数器每隔 497 天就重新开始计数。可以通过调用 OSTimeGet(),获得该计数器的当前值;也可以通过调用 OSTimeSet(),改变该计数器的值。

OSTimeGet()和 OSTimeSet()2 个函数的代码如程序清单 L5.4 所列。

注意:在访问 OSTime 时中断是关掉的。这是因为在大多数 8 位处理器上增加和拷贝一个 32 位的数,都需要数条指令,这些指令一般需要一次执行完毕,而不能被中断等因素打断。

程序清单 L5.4 得到和改变系统时间

```
INT32U  OSTimeGet (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT32U     ticks;

    OS_ENTER_CRITICAL();
```

```
    ticks = OSTime;
    OS_EXIT_CRITICAL();
    return (ticks);
}

void  OSTimeSet (INT32U ticks)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif

    OS_ENTER_CRITICAL();
    OSTime = ticks;
    OS_EXIT_CRITICAL();
}
```

第 6 章

事件控制块

图 F6.1 示意任务和中断服务子程序之间是如何进行通信的。任务或中断服务子程序可以通过事件控制块 ECB(Event Control Blocks)向另外的任务发信号。这里，信号(signal)被看成是事件(event)。这也说明为什么前面把用于通信的数据结构叫做事件控制块。

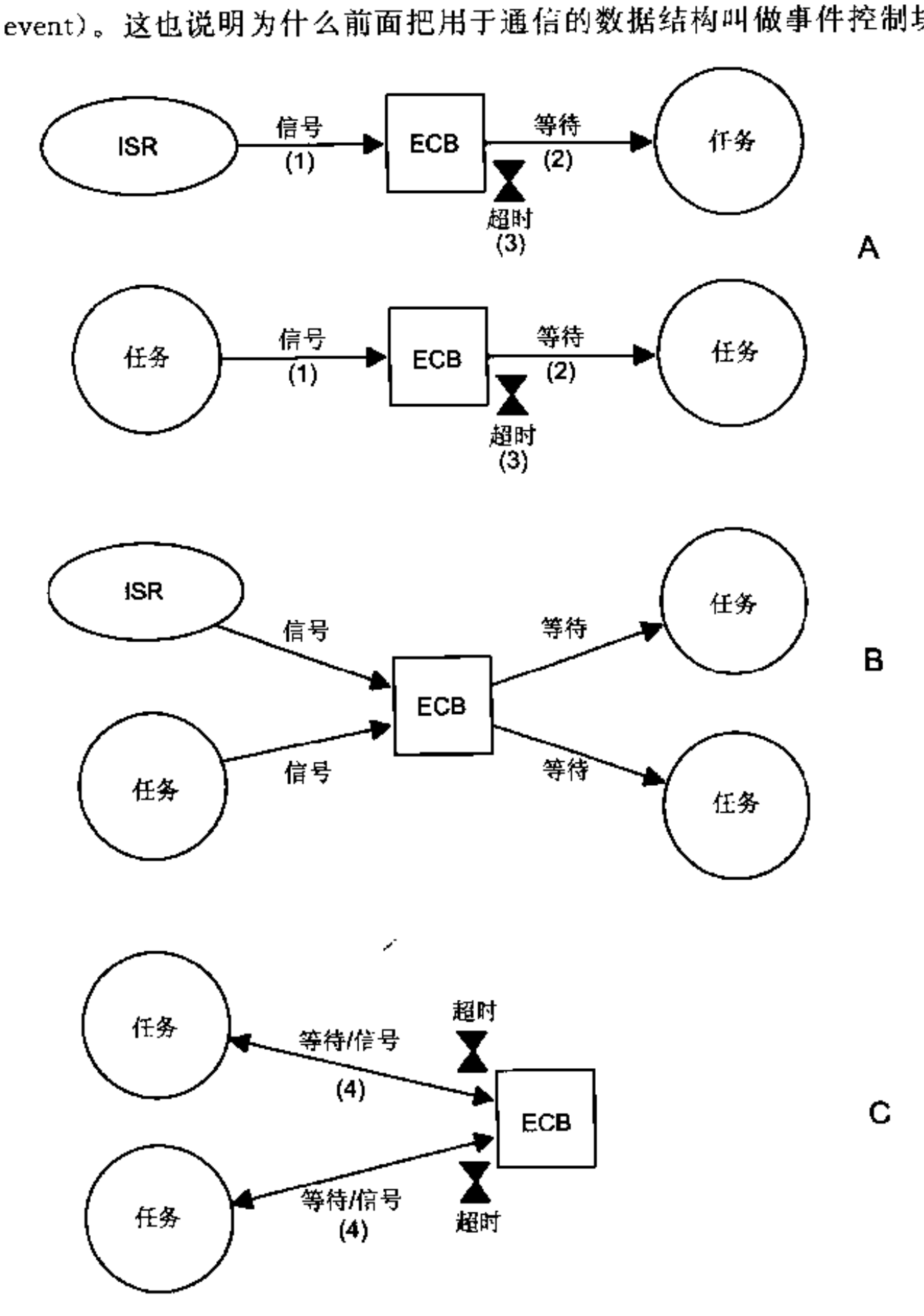


图 F6.1 事件控制块的使用

F6.1A(1)　任务或中断服务子程序可以给事件控制块 ECB 发信号。

F6.1A(2)　只有任务可以等待另一个任务或中断服务子程序通过事件控制块 ECB 给它发送信号，而中断服务子程序是不能等待事件控制块 ECB 给它发送信号的。

F6.1A(3)　处于等待状态的任务可以指定一个最长等待时间，以防止因等待的事件没有发生而无限期地等下去。

F6.1B　多个任务可以同时等待同一事件的发生。在这种情况下，当该事件发生后，所有等待该事件的任务中，只有优先级最高的任务得到该事件并进入就绪状态，准备执行。

事件控制块 ECB 可以是信号量的、消息邮箱的或者消息队列的等等，这些将在以后的章节详尽阐述。

F6.1C(4)　当事件控制块 ECB 是一个信号量时，任务可以等待它，也可以给它发送消息。

事件控制块 ECB 是用于实现以下功能函数的基本数据结构，如"信号量管理"(第 7 章)、"互斥型信号量管理"(第 8 章)、"消息邮箱管理"(第 10 章)及"消息队列管理"(第 11 章)等。

μC/OS-II 通过 uCOS_II.H 中定义的 OS_EVENT 数据结构，维护一个事件控制块 ECB 的所有信息。该结构中除了包含事件本身的定义外(如用于信号量的计数器、用于互斥型信号量的位，用于指向消息邮箱的指针，或者用于指向消息队列的指针数组等)，还定义了等待该事件的所有任务的列表。每个信号量、互斥型信号量、消息邮箱及消息队列都应分配到一个事件控制块 ECB。

事件控制块 ECB 的数据结构定义见程序清单 L6.1 和图 F6.2。

程序清单 L6.1　事件控制块 ECB 的数据结构

```
typedef struct {
    INT8U      OSEventType;                    /* 事件类型 */
    INT8U      OSEventGrp;                     /* 等待任务所在的组 */
    INT16U     OSEventCnt;                     /* 计数器(当事件是信号量时) */
    void      *OSEventPtr;                     /* 指向消息或者消息队列的指针 */
    INT8U      OSEventTbl[OS_EVENT_TBL_SIZE];  /* 等待任务列表 */
} OS_EVENT;
```

.OSEventType

定义了事件的具体类型。它可以是信号量(OS_EVENT_TYPE_SEM)、互斥型信号量(OS_EVENT_TYPE_MUTEX)、邮箱(OS_EVENT_TYPE_MBOX)或消息队列(OS_EVENT_TYPE_Q)中的一种。用户须根据该域的具体值调用 μC/OS-II 的相应功能函数，以保证对其进行操作的正确性。.OSEventType 是数据结构的第 1 个域(第 1 个字节)，这便于实时检测，以确定该指针是指向 ECB，还是事件标志(详见第 9 章："事件标志组管理")。

.OSEventPtr

只有在所定义的事件是消息邮箱或者消息队列时才使用。当所定义的事件是邮箱时，它

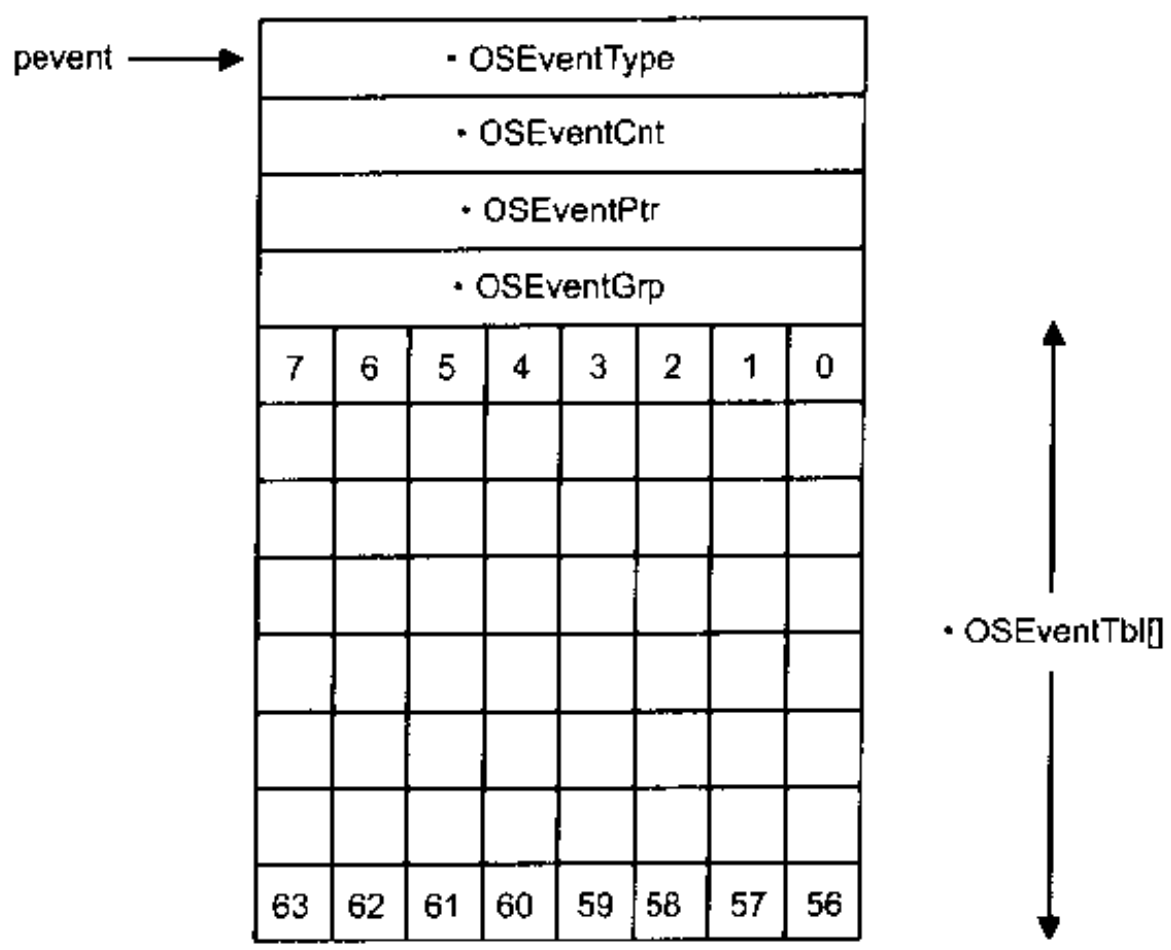


图 F6.2 事件控制块(ECB)

指向一个消息；而当所定义的事件是消息队列时，它指向一个数据结构。（详见第10章："消息邮箱管理"和第11章："消息队列管理"。）

.OSEventTbl[] 和 .OSEventGrp

分别与 OSRdyTbl[]和 OSRdyGrp 相似，只不过前两者包含的是等待某事件的任务，而后两者包含的是系统中处于就绪状态的任务(见3.04节："就绪表(ready list)")。

.OSEventCnt

当事件控制块用于信号量时，.OSEventCnt 是用于信号量的计数器(见第7章："信号量管理")；当事件控制块用于互斥型信号量时，.OSEventCnt 用于互斥型信号量和优先级继承优先级(PIP)的计数器(详见第8章："互斥型信号量管理")。

每个等待事件发生的任务都被列入到该事件控制块中的等待事件的任务列表中。该列表包括 .OSEventGrp 和 .OSEventTbl[]2 个域。变量前面的点(.)说明该变量是数据结构的一个域。在这里，所有任务的优先级被分成8组(每组8个优先级)，分别对应 .OSEventGrp 中的8位。.OSEventGrp 中的每一位用来指示是否有任务处于等待该事件的状态。当某组中有任务处于等待该事件的状态时，.OSEventGrp 中对应的位就被置1。相应地，该任务在 .OSEventTbl[] 中的对应位也被置1。.OSEventTbl[]数组的大小由系统中任务的最低优先级决定，这个值由头文件 uCOS_II.H 中的 OS_LOWEST_PRIO 常数定义。这样，在任务比较少的情况下，可减少 μC/OS-II 对系统 RAM(数据空间)的占用量。

当一个事件发生后，该事件的等待事件列表中优先级最高的任务（即在 .OSEventTbl[] 中所有被置 1 的位中优先级的数值最小的任务）得到该事件。图 F6.3 给出了 .OSEventGrp 和 .OSEventTbl[] 之间的对应关系。该关系可以描述为：

当 .OSEventTbl[0] 中的任何一位为 1 时，.OSEventGrp 中的第 0 位为 1。

当 .OSEventTbl[1] 中的任何一位为 1 时，.OSEventGrp 中的第 1 位为 1。

当 .OSEventTbl[2] 中的任何一位为 1 时，.OSEventGrp 中的第 2 位为 1。

当 .OSEventTbl[3] 中的任何一位为 1 时，.OSEventGrp 中的第 3 位为 1。

当 .OSEventTbl[4] 中的任何一位为 1 时，.OSEventGrp 中的第 4 位为 1。

当 .OSEventTbl[5] 中的任何一位为 1 时，.OSEventGrp 中的第 5 位为 1。

当 .OSEventTbl[6] 中的任何一位为 1 时，.OSEventGrp 中的第 6 位为 1。

当 .OSEventTbl_7] 中的任何一位为 1 时，.OSEventGrp 中的第 7 位为 1。

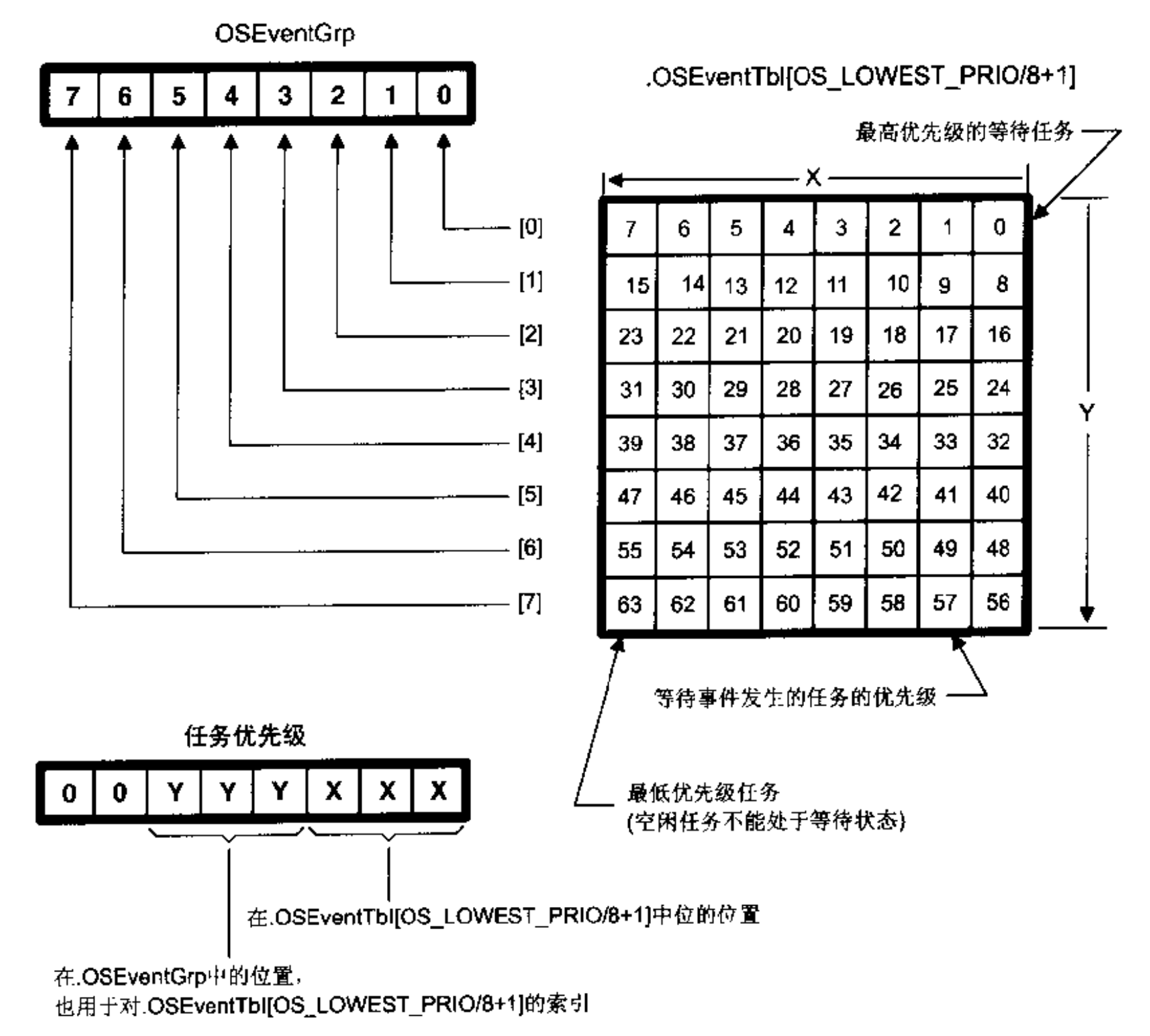


图 F6.3　事件的等待任务列表

6.00 将任务置于等待事件的任务列表

下面的代码示意将一个任务放到等待事件的任务列表中。

程序清单 L6.2 将一个任务插入到等待事件的任务列表中

```
pevent->OSEventGrp              |= OSMapTbl[prio >>3];
pevent->OSEventTbl[prio >> 3]  |= OSMapTbl[prio & 0x07];
```

其中，prio 是任务的优先级；pevent 是指向事件控制块的指针。

从程序清单 L6.2 可以看出，将一个任务插入到等待事件任务列表中所需的时间是常数，与表中现有多少个任务无关。从图 F6.3 中可以看出该算法的原理：任务优先级的最低 3 位决定了该任务在相应的 .OSEventTbl[] 中的位置，紧接着的高 3 位则决定了该任务优先级在 .OSEventTbl[] 中的字节索引。该算法中用到的查找表 .OSMapTbl[]（定义在文件 OS_CORE.C中），一般在 ROM 中实现。表 T6.1 列出了 .OSMapTbl[] 中的字节索引和位屏蔽码之间的对应关系。

表 T6.1 OSMapTbl[] 中的内容

索 引	位掩码(二进制)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

6.01 从等待事件的任务列表中使任务脱离等待状态

从等待任务列表中使任务脱离等待状态的算法则正好相反，如程序清单 L6.3 所列。

程序清单 L6.3 从等待事件的任务列表中使任务脱离等待状态

```
if ((pevent->OSEventTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0) {
    pevent->OSEventGrp &= ~OSMapTbl[prio >> 3];
}
```

该代码清除了任务在 .OSEventTbl[] 中的相应位，并且，如果其所在的组中不再有处于等待该事件的任务(.OSEventTbl[prio>>3]为 0)，则 .OSEventGrp 中的相应位也清除了。

6.02　在等待事件的任务列表中查找优先级最高的任务

从等待任务列表中查找处于等待状态的优先级最高的任务的代码如程序清单 L6.4 所列。为了提高执行速度，这里又一次使用了查表法。为了找到优先级最高的任务，不打算一次一位地扫描整个等待事件任务列表 .OSEventTbl[]。

程序清单 L6.4　在等待事件的任务列表中查找优先级最高的任务

```
y    = OSUnMapTbl[pevent->OSEventGrp];                    (1)
x    = OSUnMapTbl[pevent->OSEventTbl[y]];                 (2)
prio = (y << 3) + x;                                      (3)
```

L6.4(1)　使用 .OSEventGrp 作为 OSUnMapTbl[] (见程序清单 L6.5)的索引，可快速确定 .OSEventTbl[]中优先级最高的任务所在的组。OSUnMapTbl[]返回优先级最高的任务所在的组的位置(0～7 的一个数)，这个值与 .OSEventTbl[] 的 y 坐标相对应(如图 F6.3 所示)。

L6.4(2)　知道优先级最高的任务所在的组后，可通过再次查 OSUnMapTbl[]表，以确定其在该组中的具体位置。此次使用该组在 .OSEventTbl[]中的值作为索引号查找，也会得到一个 0～7 的值，这个值与 .OSEventTbl[] 的 x 坐标相对应(如图 F6.3 所示)。

L6.4(3)　结合前 2 项操作，即可确定优先级最高的任务的优先级序号值，其值范围为 0～63。

程序清单 L6.5　OSUnMapTbl[]

```
INT8U  const  OSUnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x00～0x0F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x10～0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x20～0x2F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x30～0x3F */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x40～0x4F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x50～0x5F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x60～0x6F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x70～0x7F */
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x80～0x8F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0x90～0x9F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0xA0～0xAF */
```

```
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0xB0~0xBF */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0xC0~0xCF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0xD0~0xDF */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,        /* 0xE0~0xEF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0         /* 0xF0~0xFF */
};
```

举例来说，如图 F6.4 所示，如果 .OSEventGrp 的值是 11001000(二进制)，即 0xC8，而对应的 OSUnMapTbl[.OSEventGrp] 值为 3，则说明优先级最高的任务所在的组是第 3 组。这正是 .OSEventTbl[]的索引号，即在.OSEventTbl[]表的第 0～7 B 中，从第 3 B 开始出现第 1 个不为 0 的值。

注意：这里假设各位的位置定义从右边开始，即第 0 位为最右边的位。

与上面所述类似，如果 .OSEventTbl[3] 的值是 00010000(二进制)，即 0x10，OSUnMapTbl[.OSEventTbl[3]]的值为 4(第 4 位)，则处于等待事件状态的任务中最高优先级是 3×8+4=28，相应关系如图 F6.3 所示。

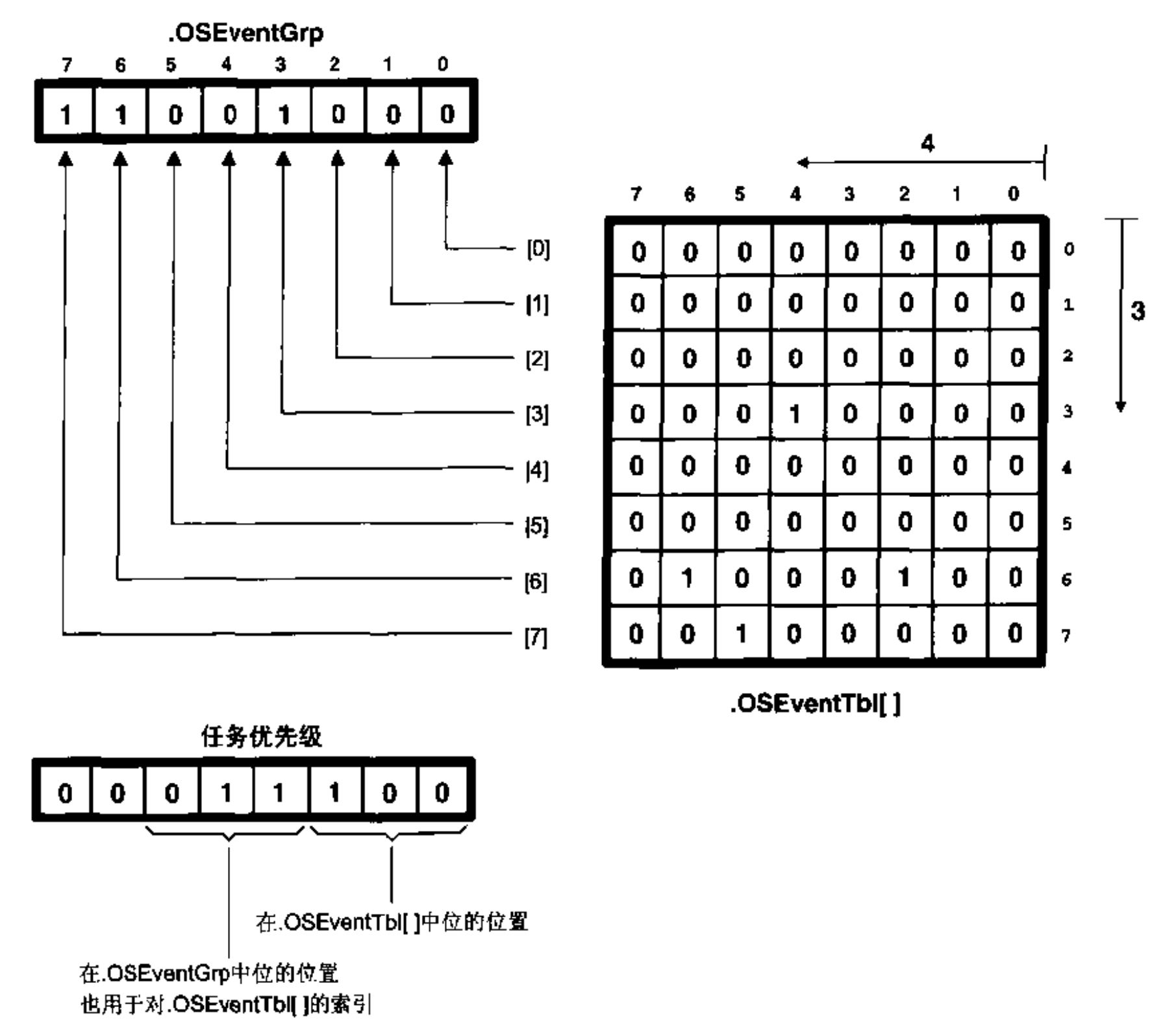


图 F6.4 等待任务列表范例

6.03　空余事件控制块链表

事件控制块 ECB 的总数由应用程序所需要的信号量、互斥型信号量、邮箱及消息队列的总数决定。该值由文件 OS_CFG.H 中的"#define OS_MAX_EVENTS"语句定义。在调用 OSInit()时(见 3.12 节："μC/OS-II 初始化")，所有事件控制块 ECB 被链接成一个单向链表——空余事件控制块链表(图 F6.5)。每当建立一个信号量、互斥型信号量、邮箱或者消息队列时，就从该链表中取出一个空余事件控制块，并对它进行初始化。调用删除信号量、互斥型信号量、邮箱及消息队列的函数 OS???Del()，可将事件控制块放回到空余事件控制块链表中。

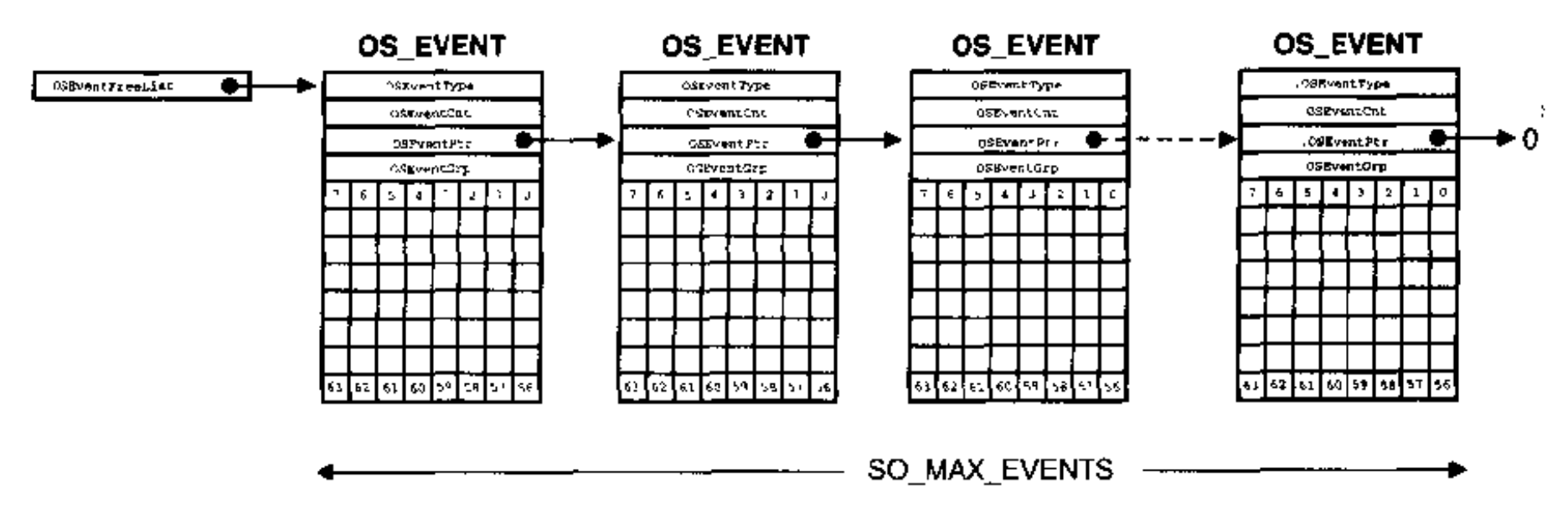


图 F6.5　空余事件控制块链表

对事件控制块进行的操作一般包括：

- 初始化一个事件控制块；
- 使一个任务进入就绪态；
- 使一个任务进入等待某事件的状态；
- 因为等待超时而使一个任务进入就绪态。

为了避免代码重复和缩短程序代码长度，以上操作分别由 4 个功能函数实现：

OS_EventWaitListInit()，OS_EventTaskRdy()，OS_EventWait()，OS_EventTO()。

6.04　初始化一个事件控制块，OS_EventWaitListInit()

程序清单 L6.6 是函数 OS_EventWaitListInit()的源代码。当建立一个信号量、邮箱或者消息队列时，相应的建立函数 OSSemCreate()，OSMutexCreate()，OSMboxCreate()或者 OSQCreate()通过调用 OS_EventWaitListInit()，对事件控制块中的等待任务列表进行初始化。该函数初始化一个空的等待任务列表，当初始化完成时，表中没有任何等待事件的任务。函数 OS_EventWaitListInit()传递一个指针变量给事件控制块。该指针变量就是创建信号量、互斥型信号量、邮箱或消息队列时分配的事件控制块的指针 pevent。这部分代码用条件编译来实现，以避免 for 循环的系统开销。

程序清单 L6.6 初始化 ECB 块的等待任务列表

```
void  OS_EventWaitListInit (OS_EVENT *pevent)
{
    INT8U   *ptbl;

    pevent->OSEventGrp = 0x00;
    ptbl               = &pevent->OSEventTbl[0];
# if OS_EVENT_TBL_SIZE > 0
    *ptbl++            = 0x00;
# endif

# if OS_EVENT_TBL_SIZE > 1
    *ptbl++            = 0x00;
# endif

# if OS_EVENT_TBL_SIZE > 2
    *ptbl++            = 0x00;
# endif

# if OS_EVENT_TBL_SIZE > 3
    *ptbl++            = 0x00;
# endif

# if OS_EVENT_TBL_SIZE > 4
    *ptbl++            = 0x00;
# endif

# if OS_EVENT_TBL_SIZE > 5
    *ptbl++            = 0x00;
# endif

# if OS_EVENT_TBL_SIZE > 6
    *ptbl++            = 0x00;
# endif

# if OS_EVENT_TBL_SIZE > 7
    *ptbl              = 0x00;
# endif
}
```

6.05　使一个任务进入就绪态，OS_EventTaskRdy()

程序清单 L6.7 是函数 OS_EventTaskRdy()的源代码。当某个事件发生了，要将等待该事件任务列表中的最高优先级的任务 HPT(Highest Priority Task)置于就绪态时，信号量、互斥型信号量、消息邮箱或消息队列所对应的 POST 函数调用 OS_EventTaskRdy()，以实现该操作。换句话说，该函数从等待任务队列中使最高优先级任务脱离等待状态，并把该任务置于就绪态。

程序清单 L6.7　使一个任务进入就绪态

```
INT8U  OS_EventTaskRdy (OS_EVENT *pevent, void *msg, INT8U msk)
{
    OS_TCB *ptcb;
    INT8U   x;
    INT8U   y;
    INT8U   bitx;
    INT8U   bity;
    INT8U   prio;


    y     = OSUnMapTbl[pevent->OSEventGrp];                          (1)
    bity  = OSMapTbl[y];                                             (2)
    x     = OSUnMapTbl[pevent->OSEventTbl[y]];                       (3)
    bitx  = OSMapTbl[x];                                             (4)
    prio  = (INT8U)((y << 3) + x);                                   (5)
    if ((pevent->OSEventTbl[y] &= ~bitx) == 0x00) {                  (6)
        pevent->OSEventGrp &= ~bity;
    }
    ptcb                    = OSTCBPrioTbl[prio];                    (7)
    ptcb->OSTCBDly          = 0;                                     (8)
    ptcb->OSTCBEventPtr     = (OS_EVENT *)0;                         (9)
#if ((OS_Q_EN >0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    ptcb->OSTCBMsg          = msg;                                   (10)
#else
    msg                     = msg;
#endif
    ptcb->OSTCBStat       &= ~msk;                                   (11)
    if (ptcb->OSTCBStat == OS_STAT_RDY) {                            (12)
        OSRdyGrp          |= bity;                                   (13)
        OSRdyTbl[y]       |= bitx;
```

```
    }
    return (prio);                                                    (14)
}
```

L6.7(1) OS_EventTaskRdy() 首先确定 HPT 任务在 .OSEventTbl[] 中的字节索引，其结果是一个 0～OS_LOWEST_PRIO/8 + 1 的数。

L6.7(2) 利用该索引得到优先级最高的任务(HPT)在 .OSEventGrp 中的位屏蔽码(位屏蔽码见表 T6.1)。

L6.7(3)

L6.7(4) OS_EventTaskRdy() 函数判断最高优先级任务 HPT 在 .OSEventTbl[] 中相应位的位置，其结果是一个 0～OS_LOWEST_PRIO/8 + 1 的数，并判断最高优先级任务 HPT 在 .OSEventTbl[] 中相应的位屏蔽码(位屏蔽码见表 T6.1)。

L6.7(5) 结合 x 和 y 索引，可确定进入就绪态的任务的优先级。

L6.7(6) 这时，可从等待事件的任务列表中删除该任务。该段代码与程序清单 L6.3 略有不同，但其工作原理是一样的。

L6.7(7) 就绪任务的任务控制块 TCB 中有须改动的状态参数。知道了任务的优先级，就可找到指向该任务的任务控制块 TCB 的指针。

L6.7(8) 因为最高优先级任务 HPT 不再等待该事件的发生，必须停止 OSTimeTick() 函数对 .OSTCBDly 域的递减操作，所以 OS_EventTaskRdy() 直接将 .OSTCBDly域清 0。

L6.7(9) 因为最高优先级任务 HPT 不再等待该事件的发生，所以 OS_EventTaskRdy()函数将其任务控制块中指向事件控制块的指针指向 NULL。

L6.7(10) 如果 OS_EventTaskRdy()是由消息邮箱或消息队列的 POST 函数调用的，那么该函数还要将相应的消息作为参数传递给最高优先级任务 HPT，并放在它的任务控制块中。

L6.7(11) 当调用 OS_EventTaskRdy()时，位屏蔽码 msk 作为参数传递给它。该参数用于将 .OSTCBStat 中的位清 0，它与所发生事件的类型相对应(OS_STAT_SEM，OS_STAT_MUTEX，OS_STAT_MBOX 或 OS_STAT_Q)。

L6.7(12)

L6.7(13) 如果 .OSTCBStat 指示该任务已处于就绪态，则 OS_EventTaskRdy()将最高优先级任务 HPT 插入到 μC/OS-II 的就绪任务列表中。

注意：最高优先级任务 HPT 得到该事件后不一定进入就绪态，也许该任务已经由于其他原因挂起了。

(见 4.07 节："挂起任务，OSTaskSuspend()"和 4.08 节："恢复任务，OSTaskResume()"。)

L6.7(14) OS_EventTaskRdy()函数返回就绪态任务的优先级。

注意：OS_EventTaskRdy()函数要在关中断的情况下调用。

6.06　使一个任务进入等待某事件发生状态，OS_EventTaskWait()

程序清单 L6.8 是 OS_EventTaskWait() 函数的源代码。当某个任务须等待一个事件的发生时，信号量、互斥型信号量、邮箱及消息队列会通过相应的 PEND 函数调用函数 OSEventTaskWait()，使当前任务从就绪任务表中脱离就绪态，并放到相应事件的事件控制块 ECB 的等待任务表中。

程序清单 L6.8　使一个任务进入等待状态

```
void  OS_EventTaskWait (OS_EVENT *pevent)
{
    OSTCBCur->OSTCBEventPtr = pevent;                                    (1)
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0x00) {  (2)
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;         (3)
    pevent->OSEventGrp                   |= OSTCBCur->OSTCBBitY;
}
```

L6.8(1)　将指向事件控制块 ECB 的指针放到任务的任务控制块 TCB 中，建立任务与事件控制块之间的链接。

L6.8(2)　将任务从就绪任务表中删除。

L6.8(3)　把该任务放到事件控制块 ECB 的等待事件的任务列表中。

6.07　由于等待超时而将任务置为就绪态，OS_EventTO()

程序清单 L6.9 是 OS_EventTO() 函数的源代码。如果在预先指定的等待时限内任务等待的事件没有发生，那么 OSTimeTick() 函数会因为等待超时而将任务的状态置为就绪态。在这种情况下，信号量、互斥型信号量、邮箱及消息队列会通过 PEND 函数调用 OS_EventTO() 函数，以完成这项工作。

程序清单 L6.9　因为等待超时而将任务置为就绪态

```
void  OS_EventTO (OS_EVENT *pevent)
{
    if ((pevent->OSEventTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0x00) {   (1)
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
```

```
    }
    OSTCBCur->OSTCBStat      = OS_STAT_RDY;                          (2)
    OSTCBCur->OSTCBEventPtr  = (OS_EVENT *)0;                        (3)
}
```

L6.9(1) OS_EventTO()函数必须从事件控制块 ECB 中的等待任务列表中将该任务删除。该代码与程序清单 L6.3 略有不同，但其工作原理是一样的。

L6.9(2) 该任务被置为就绪状态。

L6.9(3) 从任务控制块 TCB 中将指向事件控制块 ECB 的指针删除。

注意：调用 OS_EventTO()也应当先关中断。

第 7 章
信号量管理

μC/OS-II 中的信号量由 2 部分组成：一部分是 16 位的无符号整型信号量的计数值（0～65 535）；另一部分是由等待该信号量的任务组成的等待任务表。μC/OS-II 提供了 6 个对信号量进行操作的函数：OSSemAccept()，OSSemCreate()，OSSemDel()，OSSemPend()，OSSemPost()及 OSSemQuery()。

须在 OS_CFG.H 中将配置常数置为 1，这样 μC/OS-II 才能支持信号量。表 T7.1 列出了哪些函数是可以根据 OS_CFG.H 中配置常数的值而确定是否编译的。当 OS_SEM_EN 设为 0 时，所有的信号量函数都不能使用。需要使用这些函数时，只需将配置常数置为 1。因为只要 μC/OS-II 支持信号量，就必须有 OSSemCreate()，OSSemPend()及 OSSemPost()函数，所以它们不能被单独禁止。

表 T7.1　OS_CFG.H 中的信号量配置常数

μC/OS-II 中的信号量函数	在 OS_CFG.H 中置 1 允许相应函数
OSSemAccept()	OS_SEM_ACCEPT_EN
OSSemCreate()	
OSSemDel()	OS_SEM_DEL_EN
OSSemPend()	
OSSemPost()	
OSSemQuery()	OS_SEM_QUERY_EN

图 F7.1 描述了任务、中断服务子程序和信号量之间的关系。图中用钥匙或旗帜表示信号量。

如果信号量用于对共享资源的访问，那么信号量就用钥匙符号。符号旁边的数字 N 代表可用资源数。对于二值信号量，N 值就是 1。如果信号量用于表示某事件的发生，那么就用旗帜符号。这时的数字 N 代表事件已经发生的次数。小沙漏表示 OSSemPend()定义的超时时限。

从图 F7.1 可以看出，OSSemAccept()，OSSemPost()及 OSSemQuery()函数可以由任务或者中断服务子程序调用，而 OSSemDel()和 OSSemPend()函数只能由任务程序调用。

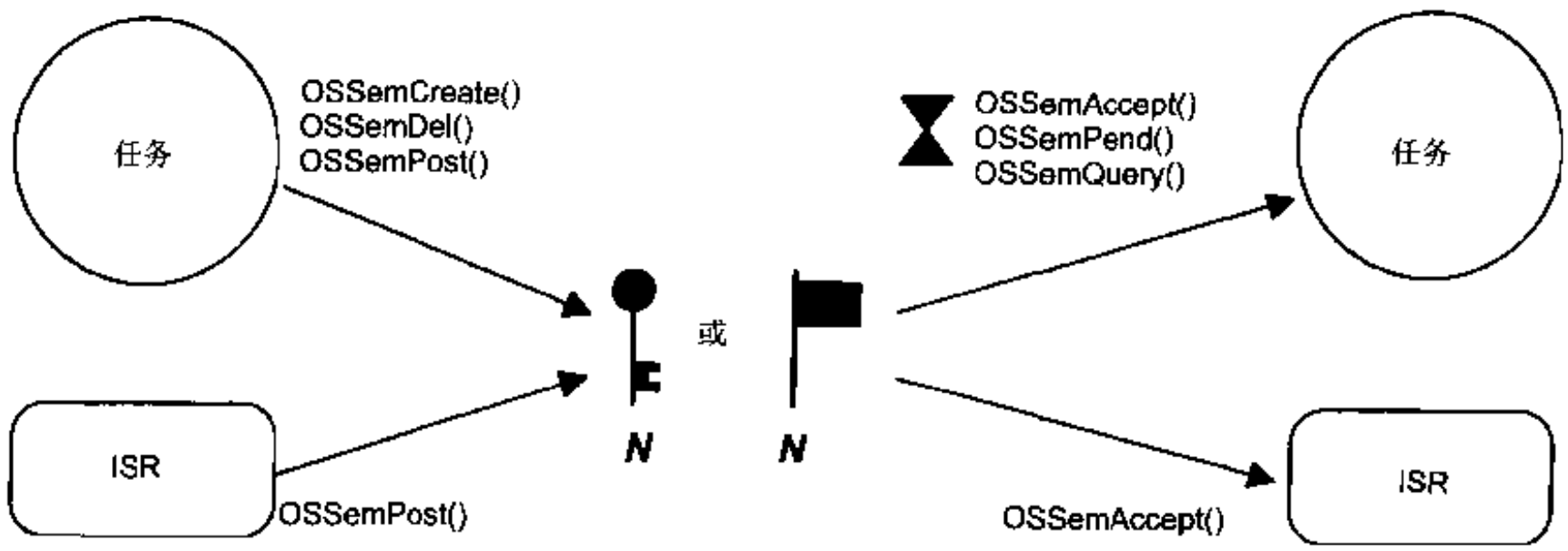


图 F7.1 任务、中断服务子程序和信号量之间的关系

7.00 建立一个信号量，OSSemCreate()

在使用一个信号量之前，首先须建立该信号量。可以调用OSSemCreate()函数来建立信号量，并对信号量赋予初始计数值。该初始值为0～65 535的一个数。如果信号量是用来表示一个或者多个事件发生的，那么该信号量的初始值通常赋为0；如果信号量用于对共享资源的访问，那么该信号量的初始值应赋为1（例如，把它当作二值信号量使用）；如果信号量用来表示允许任务访问n个相同的资源，那么该信号量的初始值应赋为n，并把该信号量作为一个可计数的信号量使用。

OSSemCreate()函数的源代码如程序清单L7.1所列。

程序清单L7.1 建立一个信号量

```
OS_EVENT    *OSSemCreate (INT16U cnt)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;                                                    (1)
#endif
    OS_EVENT    *pevent;


    if (OSIntNesting > 0) {                                                 (2)
        return ((OS_EVENT *)0);
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                               (3)
    if (OSEventFreeList != (OS_EVENT *)0) {                                 (4)
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;          (5)
    }
```

```
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                                    (6)
        pevent->OSEventType = OS_EVENT_TYPE_SEM;                      (7)
        pevent->OSEventCnt  = cnt;                                    (8)
        pevent->OSEventPtr  = (void *)0;                              (9)
        OS_EventWaitListInit(pevent);                                 (10)
    }
    return (pevent);                                                  (11)
}
```

L7.1(1)　如果关中断的方法 OS_CRITICAL_METHOD 等于 3，则定义局部变量 cpu_sr。

L7.1(2)　中断服务子程序不能调用 OSSemCreate()函数，所以 OSSemCreate()函数首先判定不是被中断服务子程序所调用。所有建立信号量的工作必须在任务级代码中或者多任务启动前完成。

L7.1(3)　OSSemCreate()函数试图从空余事件控制块链表中获得一个事件控制块 ECB（见图 F6.5）。

L7.1(4)

L7.1(5)　对空余事件控制块链表的指针进行适当的调整，使它指向下一个空余的事件控制块。

L7.1(6)

L7.1(7)　如果这时有事件控制块 ECB 可用，就将该事件控制块 ECB 的事件类型设置成信号量 OS_EVENT_TYPE_SEM。其他信号量操作函数 OSSem???()通过检查该域，以保证所操作的事件控制块类型的正确性（类型为信号量）。这样可以防止调用 OSSemPost()函数对一个用做邮箱的事件控制块 ECB 进行操作（见第 10 章："消息邮箱管理"）。

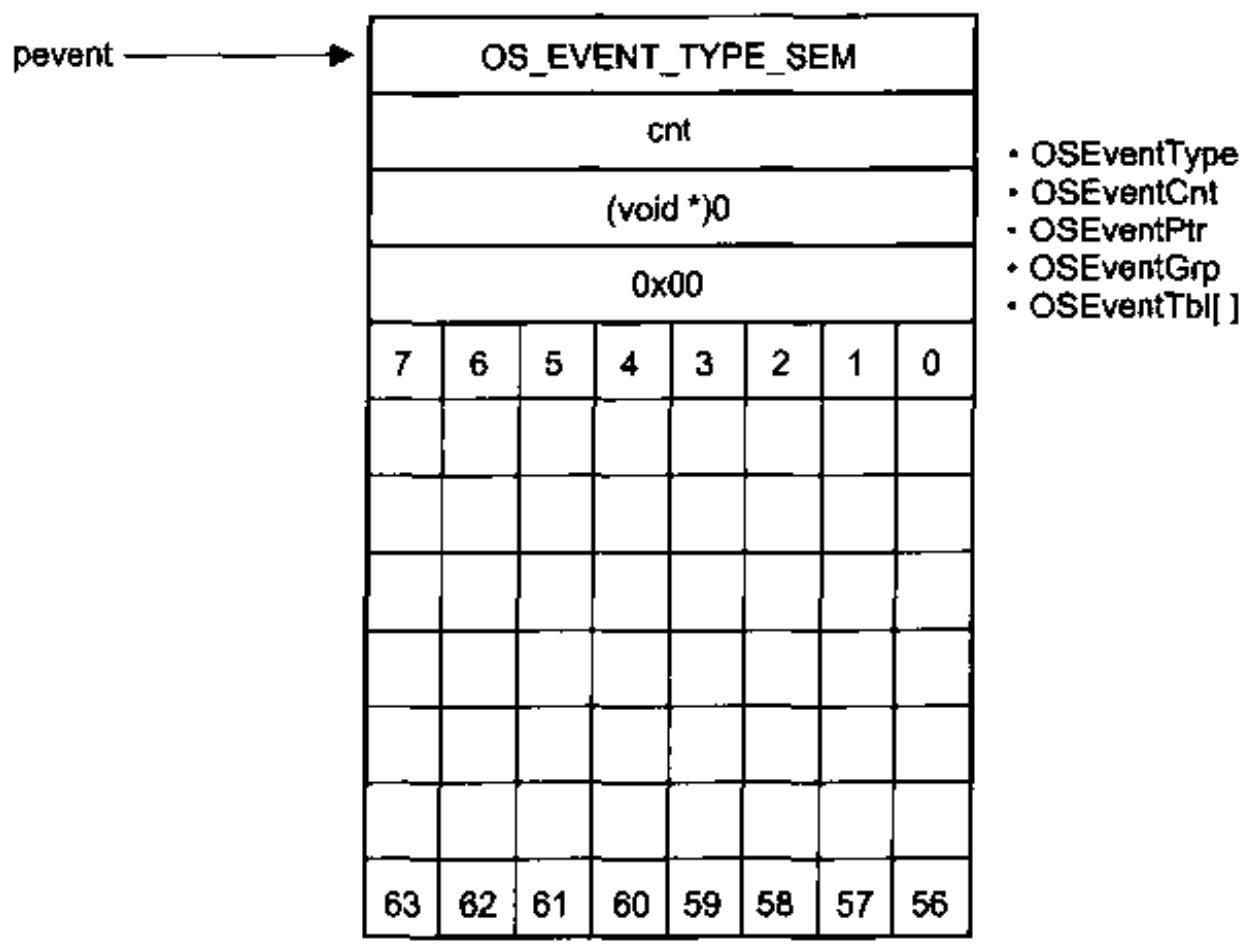
L7.1(8)　将信号量的初始值存入事件控制块 ECB 中。

L7.1(9)　将 .OSEventPtr 初始化为指向 NULL，因为它已经不再属于空余事件控制块链表。

L7.1(10)　调用 OS_EventWaitListInit()函数，对事件控制块的等待任务列表进行初始化（见 6.04 节："初始化一个事件控制块，OS_EventWaitListInit()"）。因为信号量正在被初始化，这时没有任何任务等待该信号量，所以，函数 OS_EventWaitListInit()将 .OSEventGrp 和 .OSEventTbl[]清 0。

L7.1(11)　最后 OSSemCreate()函数返回给调用函数一个指向事件控制块 ECB 的指针。以后，OSSemAccept()，OSSemDel()，OSSemPend()，OSSemPost()及 OSSemQuery()函数对信号量的所有操作都是通过该指针完成的；因此，这个指针实际上是该信号量的句柄（handle）。如果系统中没有可用的事件控制块，OSSemCreate()函数将返回一个指向 NULL 的空指针。必须养成良好的习惯，在函数返回时，检查和确认返回值的正确性。传递 NULL 指针并不意味着操作失败，因为，μC/OS-II 会检查参数的合理性（虽然只是在 OS_ARG_CHK_EN置为 1 时如此）。

图 F7.2 为 OSSemCreate()函数返回之前事件控制块 ECB 的内容。



注：表中所有值均初始化为0。

图 F7.2 OSSemCreate()函数返回之前的事件控制块 ECB

7.01 删除一个信号量,OSSemDel()

程序清单 L7.2 是删除一个信号量的函数 OSSemDel()的源代码。当文件 OS_CFG.H 中的 OS_SEM_DEL_EN 置为 1 时,该代码才被编译。使用此函数时需特别注意,多个任务可能试图操作已经删除的信号量。总之,在删除信号量之前,必须首先删除操作该信号量的所有任务。

程序清单 L7.2 删除一个信号量

```
OS_EVENT   *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    tasks_waiting;
    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_DEL_ISR;
        return (pevent);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                   (2)
```

```
        *err = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                    (3)
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                  (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:
            if (tasks_waiting == FALSE) {                              (5)
                pevent->OSEventType = OS_EVENT_TYPE_UNUSED;            (6)
                pevent->OSEventPtr  = OSEventFreeList;                 (7)
                OSEventFreeList     = pevent;
                OS_EXIT_CRITICAL();
                *err = OS_NO_ERR;
                return ((OS_EVENT *)0);                                (8)
            } else {
                OS_EXIT_CRITICAL();
                *err = OS_ERR_TASK_WAITING;
                return (pevent);
            }
        case OS_DEL_ALWAYS:                                            (9)
            while (pevent->OSEventGrp != 0x00) {                       (10)
                OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM);
            }
            pevent->OSEventType = OS_EVENT_TYPE_UNUSED;                (11)
            pevent->OSEventPtr  = OSEventFreeList;                     (12)
            OSEventFreeList     = pevent;
            OS_EXIT_CRITICAL();
            if (tasks_waiting == TRUE) {
                OS_Sched();                                            (13)
            }
            *err = OS_NO_ERR;
            return ((OS_EVENT *)0);                                    (14)
        default:
```

```
        OS_EXIT_CRITICAL();
        *err = OS_ERR_INVALID_OPT;
        return (pevent);
    }
}
```

L7.2(1) 中断服务子程序是不能调用 OSSemDel()函数的，所以 OSSemDel()函数首先确认不被中断服务子程序所调用。

L7.2(2)

L7.2(3) OSSemDel()函数验证 pevent，确认它为非空指针，并且指向某一用于信号量的事件控制块 ECB。

L7.2(4) OSSemDel()函数判定是否有任务在等待该信号量，并根据判定结果对标志变量 tasks_waiting 进行设置。

根据 opt 参数的值，OSSemDel()对信号量进行相应的删除操作。当没有任务等待信号量时，OSSemDel()删除信号量（当 opt == OS_DEL_NO_PEND 时）；或者，尽管有任务在等待，OSSemDel() 函数亦直接删除信号量（当 opt == OS_DEL_ALWAYS 时）。

L7.2(5)

L7.2(6)

L7.2(7) 当 opt 被置为 OS_DEL_NO_PEND，并且没有任务在等待该信号量时，OSSemDel()函数将事件控制块 ECB 标志为未使用，并将它退回到空余事件控制链表中。此操作允许该事件控制块用于创建另一信号量（或者其他使用事件控制块 ECB 的通信机制）。

L7.2(8) OSSemDel()函数返回一个指向 NULL 的空指针，因为，此后该信号量就不能再通过原先的指针访问了。当指定 opt 为 OS_DEL_NO_PEND 时，若有任务在等待该信号量，则不能删除该信号量，此时 OSSemDel()函数将返回"出错"代码(OS_ERR_TASK_WAITING)。

L7.2(9)

L7.2(10) 当 opt 被设定为 OS_DEL_ALWAYS 时，所有等待该信号的任务都将进入就绪态，每个任务都得到了该信号量。当然，这样可能会导致致命的后果，因为，采用信号量是为了防止对资源的多重访问。

L7.2(11)

L7.2(12) 当所有的等待任务就绪后，OSSemDel()函数将事件控制块 ECB 标志为未使用，并把它退回到空余事件控制链表中。

L7.2(13) 当有任务在等待该信号量时，任务调度才被调用。

L7.2(14) 再次注意，OSSemDel()函数返回一指向 NULL 的空指针，因为，此后该信号量就不能再通过原先的指针访问了。

7.02　等待一个信号量，OSSemPend()

等待一个信号量函数 OSSemPend() 的源代码如程序清单 L7.3 所列。

程序清单 L7.3　等待一个信号量

```
void  OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_PEND_ISR;
        return;
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                   (2)
        *err  = OS_ERR_PEVENT_NULL;
        return;
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                  (3)
        *err = OS_ERR_EVENT_TYPE;
        return;
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventCnt > 0) {                                    (4)
        pevent->OSEventCnt--;                                        (5)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;                                            (6)
        return;
    }
    OSTCBCur->OSTCBStat |= OS_STAT_SEM;                              (7)
    OSTCBCur->OSTCBDly  = timeout;                                   (8)
    OS_EventTaskWait(pevent);                                        (9)
    OS_EXIT_CRITICAL();
    OS_Sched();                                                      (10)
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStat & OS_STAT_SEM) {                         (11)
```

```
        OS_EventTO(pevent);                                          (12)
        OS_EXIT_CRITICAL();
        *err = OS_TIMEOUT;                                           (13)
        return;
    }
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                         (14)
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
```

L7.3(1) OSSemPend()函数检查是否是中断服务子程序调用了该函数。因为中断服务子程序不能等待，所以由中断服务子程序调用OSSemPend()函数就没有意义。相反，可以在中断服务子程序中调用OSSemAccept()函数（见7.04节："无等待地请求一个信号量，OSSemAccept()"）。

L7.3(2)

L7.3(3) 如果OS_ARG_CHK_EN设置为1，OSSemPend()函数检查pevent，以确定其为非空指针，并且指向由OSSemCreate()函数建立的事件控制块ECB。

L7.3(4)

L7.3(5)

L7.3(6) 如果信号量本来就是有效的（信号量的计数值非0），则信号量的计数值递减，OSSemPend()函数将"无错"代码（OS_NO_ERR）返回给调用它的函数。如果OSSemPend()函数用于等待一共享资源的信号量，那么通过该返回值，可以知道代码是否正确运行，并可进一步访问该共享资源。这也是OSSemPend()函数执行得最快的情况。

L7.3(6) 如果此时信号量无效（计数器的值是0），OSSemPend()函数需进一步检查调用它的函数是否为中断服务子程序。因为中断服务子程序是不能等待的，所以由中断服务子程序调用OSSemPend()函数没有意义。相反，应该调用OSSemAccept()（见7.04节："无等待地请求一个信号量，OSSemAccept()"）。这里加入这些代码，是为了以防万一。

如果信号量的计数值为0，则调用OSSemPend()函数的任务须进入睡眠状态，等待另一个任务（或者中断服务子程序）发出该信号量（见7.03节："发出一个信号量，OSSemPost()"）。OSSemPend()允许定义一个最长等待时间作为它的参数（超时时限），这样可以避免该任务无限期地等待信号量。如果该参数值是一个大于0的值，那么该任务将一直等到信号量有效或者是等待超时；如果该参数值为0，该任务将无限期地一直等待下去。

L7.3(7) OSSemPend()函数通过将任务控制块中的状态标志，OSTCBStat置1，使任务进入睡眠状态。

L7.3(8) 最长等待时限也同时置入到任务控制块TCB中，该值在OSTimeTick()函数中被逐次递减。

注意：OSTimeTick()对每个任务的任务控制块TCB中的，OSTCBDly域

做递减操作(只要该域不为 0)(见 3.11 节:"时钟节拍")。

L7.3(9)　真正将任务置入睡眠状态的操作在 OS_EventTaskWait() 函数中执行(见 6.06 节:"使一个任务进入等待某事件发生状态,OS_EventTaskWait()")。

L7.3(10)　由于得不到信号量,当前任务就不能再处于就绪态了,于是调用任务调度函数,让下一个优先级最高的任务运行。这样,调用 OSSemPend()函数的任务被挂起了,一直到信号量出现,才能再继续运行。在此之前,该任务并不知道自己被挂起了。

L7.3(11)　当信号量有效(或者等待超时)时,OSSemPend()函数在调用 OS_Sched()函数之后又继续运行。在此之后,OSSemPend()须检查任务控制块中的状态标志,判定该任务是否仍处于等待信号量的状态。如果是,则说明该任务并没有被 OSSemPost() 函数发出的信号量唤醒;而实际上该任务是因等待超时,而被 TimeTick() 函数置为就绪状态的。

L7.3(12)

L7.3(13)　这种情况下,OSSemPend() 函数调用 OS_EventTO() 函数,将任务从等待任务列表中删除,并返回给调用它的任务一个"超时"错误代码。如果任务的任务控制块 TCB 中的 OS_STAT_SEM 标志位没有置 1,则 OSSemPost()已发出信号量(见 7.03 节:"发出一个信号量,OSSemPost()"),并且调用 OSSemPend()的任务已经得到了该信号量。

L7.3(14)　将指向信号量 ECB 的指针从该任务的任务控制块中删除。

7.03　发出一个信号量,OSSemPost()

程序清单 L7.4 是发出一个信号量的函数 OSSemPost()的源代码。

程序清单 L7.4　发出一个信号量

```
INT8U  OSSemPost (OS_EVENT *pevent)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif

# if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                          (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                         (2)
        return (OS_ERR_EVENT_TYPE);
    }
# endif
```

```
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                    (3)
        OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM);                 (4)
        OS_EXIT_CRITICAL();
        OS_Sched();                                                      (5)
        return (OS_NO_ERR);
    }
    if (pevent->OSEventCnt < 65535) {
        pevent->OSEventCnt++;                                            (6)
        OS_EXIT_CRITICAL();
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_SEM_OVF);                                                 (7)
}
```

L7.4(1)

L7.4(2) 如果OS_ARG_CHK_EN置1，则OSSemPost()函数首先检查参数指针pevent是否为非NULL指针，并且检查pevent指向的任务控制块是否是由OSSemCreate()函数建立的。

L7.4(3) 接着检查是否有任务在等待该信号量。如果该事件控制块中的.OSEventGrp域不是0，则说明有任务正在等待该信号量。

L7.4(4)

L7.4(5) OS_EventTaskRdy()函数把优先级最高的任务从等待任务列表中去除（见6.05节："使一个任务进入就绪态，OS_EventTaskRdy()"），并使它进入就绪态。然后，调用OSSched()任务调度函数，检查该任务是否是系统中的最高优先级的就绪态任务。如果是优先级最高的就绪态任务，这时就要进行任务切换（如果OSSemPost()函数是在任务中调用的），准备执行该就绪态任务。换句话说，因为OSSemPost()函数使更重要的任务进入了就绪态，故调用OSSemPost()函数的任务就不能再继续运行了，μC/OS-II就是这样做的。如果就绪任务不是优先级最高的任务，OSSched()直接返回，调用OSSemPost()的任务得以继续执行。

L7.4(6)

L7.4(7) 如果这时没有任务在等待该信号量，则该信号量的计数值就简单地加1。μC/OS-II中计数信号量采用16位变量的形式，OSSemPost()函数保证信号量不会溢出。

特别注意：当中断服务子程序调用OSSemPost()函数时，不会发生上面的任务切换。因为，任务切换须等到中断嵌套的最外层中断服务子程序调用OSIntExit()函数后，才能进行（见3.10节："μC/OS-II中的中断"）。

7.04　无等待地请求一个信号量，OSSemAccept()

当一个任务请求一个信号量时，如果该信号量暂时无效，也可以让该任务简单地返回，而不是进入睡眠等待状态。这种情况下的操作是由 OSSemAccept() 函数完成的，其源代码见程序清单 L7.5。

程序清单 L7.5　无等待地请求一个信号量

```
INT16U  OSSemAccept (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT16U     cnt;

#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                  (1)
        return (0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                 (2)
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
    cnt = pevent->OSEventCnt;                                       (3)
    if (cnt > 0) {                                                  (4)
        pevent->OSEventCnt--;                                       (5)
    }
    OS_EXIT_CRITICAL();
    return (cnt);                                                   (6)
}
```

L7.5(1)

L7.5(2)　如果 OS_CFG.H 中的 OS_ARG_CHK_EN 置为 1，则 OSSemAccept() 函数首先检查参数指针 pevent 是否为非 NULL 指针，并且检查 pevent 指向的事件控制块是否是由 OSSemCreate() 函数建立的。

L7.5(3)

L7.5(4)　OSSemAccept() 函数从该信号量的事件控制块中取出当前计数值，并检查该信号量是否有效(计数值是否为非 0 值)。

L7.5(5)　如果信号量有效，则将信号量的计数值减 1。

L7.5(6) 最后将信号量的原有计数值返回给调用函数。

调用函数需对该返回值进行检查。如果该值是0，则说明该信号量无效；如果该值大于0，则说明该信号量有效，同时该值也暗示着该信号量当前可用的资源数。应该注意的是，这些可用资源中，已经被该调用函数自身占用了一个（该计数值已经被减1）。

中断服务子程序可以使用OSSemAccept()函数；但是，并不推荐任务和中断服务子程序共享信号量。因为，信号量一般用于任务级。如果确实要在任务和中断服务子程序中传递信号量，则中断服务子程序只能发送信号量。

7.05 查询一个信号量的当前状态，OSSemQuery()

在应用程序中，随时可以调用函数OSSemQuery()，来查询一个信号量的当前状态（程序清单L7.6）。该函数有2个参数：一个是指向信号量对应事件控制块的指针pevent，该指针是在建立信号量时，由OSSemCreate()函数返回的；另一个是指向用于记录信号量信息的数据结构OS_SEM_DATA（见文件uCOS_II.H）的指针pdata。因此，调用该函数前，必须先定义该结构变量，用于存储信号量的有关信息。在这里，之所以使用一个新的数据结构，是因为调用函数应只关心与特定信号量有关的信息，而非OS_EVENT数据结构包含的全部信息。OS_SEM_DATA数据结构只包含信号量当前计数值（.OSCnt）和等待任务列表（.OSEventTbl[]和.OSEventGrp）；而OS_EVENT中还包含了另外的2个域（.OSEventType和.OSEventPtr）。

程序清单L7.6 查询一个信号量的当前状态

```
INT8U  OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif

    INT8U     *psrc;
    INT8U     *pdest;

#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                    (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                   (2)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
```

```
    OS_ENTER_CRITICAL();
    pdata->OSEventGrp = pevent->OSEventGrp;                                    (3)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
#if OS_EVENT_TBL_SIZE > 0
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 1
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 4
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *pdest            = *psrc;
#endif

    pdata->OSCnt      = pevent->OSEventCnt;                                    (4)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L7.6(1)

L7.6(2) 如果OS_CFG.H文件中的OS_ARG_CHK_EN置为1，则OSSemQuery()函数首先检查参数指针pevent是否为非NULL指针，并且检查pevent指向的事件控制块是否是由OSSemCreate()函数建立的。

L7.6(3) OSSemQuery()将OS_EVENT结构中的等待任务列表复制到OS_SEM_DATA数据结构中。此处，使用条件编译而不使用循环语句，目的是使程序执行得更快。

L7.6(4) OSSemQuery()函数将OS_EVENT结构中的信号量当前计数值复制到OS_SEM_DATA结构中。

第 8 章

互斥型信号量管理

任务可以用互斥型信号量(mutual exclusion semaphores)实现对共享资源的独占式处理。互斥型信号量也称做 mutex。mutex 是二值信号量，除具有 μC/OS-II 中普通信号量的机制外，还具有其他一些特性。

mutex 可以在应用程序代码中用于降解优先级反转问题，见 2.16 节。当高优先级的任务需要使用某共享资源，而该资源已被一个低优先级的任务占用时，就会发生优先级反转。为了降解优先级反转，内核可以将低优先级任务的优先级提升到高于那个高优先级的任务，直到低优先级的任务使用完占用的共享资源。

为了互斥型信号量得以实现，实时内核应能够支持相同优先级下的多任务。不幸的是，μC/OS-II 不允许多任务处于同一优先级；然而，这个问题还是有办法解决的。如果把占有 mutex 的低优先级任务的优先级提高到略高于等待 mutex 的高优先级任务的优先级，情况会怎样呢？

用以下的例子来说明 μC/OS-II 中互斥型信号量的机理。在程序清单 L8.1 中，有 3 个任务要使用某共享资源。为得到该资源，每个任务都可能因得不到互斥型信号量 ResourceMutex而被挂起(pend)。任务 1 的优先级最高(10)，任务 2 中等(15)，任务 3 最低(20)，没有被占用的、略高于最高优先级任务的优先级(9)，保留给"优先级继承优先级"PIP (Priority Inheritance Priority)。

程序清单 L8.1　mutex 应用实例

```
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);
     --------应用程序初始化--------
    OSStart();                                                          (4)
}


void TaskPrio10 (void *pdata)                                   /* 任务1 */
{
    INT8U err;


    pdata = pdata;
    while (1) {
         -------- 应用程序代码 --------
        OSMutexPend(ResourceMutex, 0, &err);
         -------- 占用共享资源 --------
        OSMutexPost(ResourceMutex);
         -------- 应用程序代码 --------
    }
}


void TaskPrio15 (void *pdata)                                   /* 任务2 */
{
    INT8U err;


    pdata = pdata;
    while (1) {
         -------- 应用程序代码 --------
        OSMutexPend(ResourceMutex, 0, &err);
         -------- 占用共享资源 --------
        OSMutexPost(ResourceMutex);
         -------- 应用程序代码 --------
    }
}


void TaskPrio20 (void *pdata)                                   /* 任务3 */
{
    INT8U err;
```

```
    pdata = pdata;
    while (1) {
        -------- 应用程序代码 --------
        OSMutexPend(ResourceMutex, 0, &err);
        -------- 占用共享资源 --------
        OSMutexPost(ResourceMutex);
        -------- 应用程序代码 --------
    }
}
```

L8.1(1)

L8.1(2)　如 main() 中所示，初始化 μC/OS-II，调用 OSMutexCreate()，建立互斥型信号量。

注意：调用 OSMutexCreate() 时，传递了 PIP 参数，也就是 9。

L8.1(3)

L8.1(4)　3 个任务都建立了，启动 μC/OS-II。

假设应用程序运行了一些时候，在某一时刻，任务 3 首先申请到了 mutex 并开始同共享资源打交道。过了一会儿，任务 3 的 CPU 使用权被任务 1 剥夺了。任务 1 需要使用共享资源，于是申请 mutex(通过调用 OSMutexPend())。在这种情况下，OSMutexPend()注意到高优先级的任务要用这个共享资源，于是将任务 3 的优先级升至 9，并强制任务调度回到任务 3。任务 3 继续运行处理共享资源，试图能尽快释放该共享资源。当共享资源使用完时，任务 3 调用 OSMutexPost()，释放 mutex。OSMutexPost()注意到原来占用这个 mutex 的低优先级的任务优先级被抬高了，于是将任务 3 的优先级恢复到原来的水平。OSMutexPost()还注意到有个高优先级的任务，即任务 1，需要使用这个共享资源，于是将共享资源交给任务 1，并做任务切换，让任务 1 运行。

μC/OS-II 的互斥型信号量由 3 个元素组成：1 个标志，指示 mutex 是否可以使用(0 或 1)；1 个优先级，准备一旦高优先级的任务需要这个 mutex，赋给占有 mutex 的任务；1 个等待该 mutex 的任务列表。

μC/OS-II 对于互斥型信号量提供 6 种服务：OSMutexCreate()，OSMutexDel()，OSMutexPend()，OSMutexPost()，OSMutexAccept() 以及 OSMutexQuery()。

为允许 μC/OS-II 中的互斥型信号量功能，应置 OS_CFG.H 文件中的相应常数为 1，基于 CONFIG.H 中配置常数的数值，决定哪些函数须编译。相应的配置常数列在表 T8.1 中。

注意：如果 OS_MUTEX_EN 设为 0，所有 mutex 服务都是禁止的。要得到表 T8.1 中列出的特别功能(服务)，应当将相应配置常数置为 1。OSMutexCreate()，OSMutexPend()及 OSMutexPost()不能被单个地禁止掉，而另外几项功能是可以的。这是因为只要允许了 μC/OS-II中的互斥型信号量管理，上述 3 个函数总是要用到的。

表 T8.1 OS_CFG.H 中的 mutex 配置常数

μC/OS-II 中的 mutex 函数	在 OS_CFG.H 中置 1 允许相应函数
OSMutexAccept()	OS_MUTEX_ACCEPT_EN
OSMutexCreate()	
OSMutexDel()	OS_MUTEX_DEL_EN
OSMutexPend()	
OSMutexPost()	
OSMutexQuery()	OS_MUTEX_QUERY_EN

图 F8.1 是解释任务与互斥型信号量之间关系的流程图。mutex 只能供任务使用。注意图中代表 mutex 符号的是一把钥匙，钥匙符表明 mutex 是用于处理共享资源的。

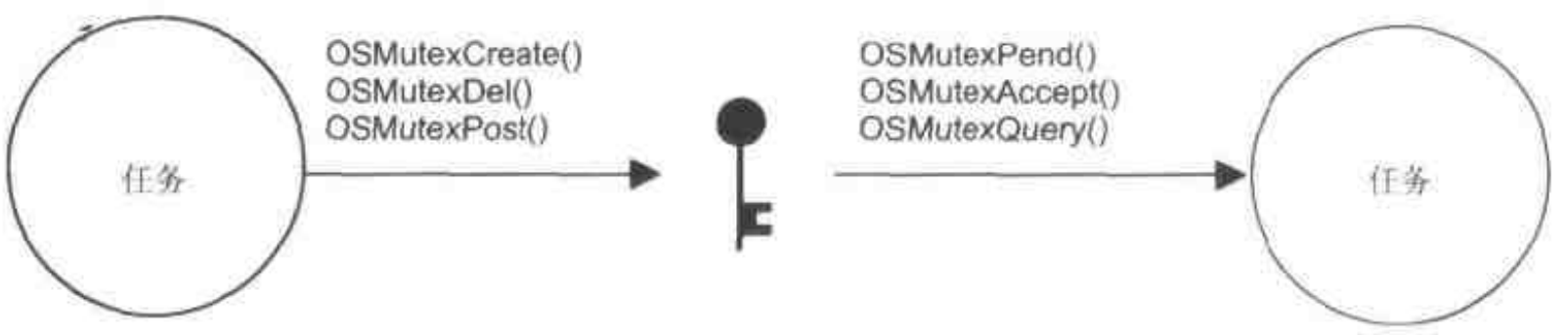


图 F8.1 任务与互斥型信号量之间的关系

8.00 建立一个互斥型信号量，OSMutexCreate()

在使用 mutex 之前，必须建立它。建立 mutex 是通过调用函数 OSMutexCreate()来完成的。mutex 的初值已经设为 1，表示资源是可以使用的。程序清单 L8.2 列出的是建立 mutex 的代码。

程序清单 L8.2 建立一个互斥型信号量

```
OS_EVENT *OSMutexCreate (INT8U prio, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_EVENT *pevent;


    if (OSIntNesting > 0) {                                                    (1)
        *err = OS_ERR_CREATE_ISR;
        return ((OS_EVENT *)0);
```

```
    }
#if OS_ARG_CHK_EN
    if (prio >= OS_LOWEST_PRIO) {                                    (2)
        *err = OS_PRIO_INVALID;
        return ((OS_EVENT *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] != (OS_TCB *)0) {                         (3)
        *err = OS_PRIO_EXIST;
        OS_EXIT_CRITICAL();
        return ((OS_EVENT *)0);
    }
    OSTCBPrioTbl[prio] = (OS_TCB *)1;                                (4)
    pevent             = OSEventFreeList;                            (5)
    if (pevent == (OS_EVENT *)0) {
        OSTCBPrioTbl[prio] = (OS_TCB *)0;
        OS_EXIT_CRITICAL();
        *err               = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    OSEventFreeList        = (OS_EVENT *)OSEventFreeList->OSEventPtr;  (6)
    OS_EXIT_CRITICAL();
    pevent->OSEventType    = OS_EVENT_TYPE_MUTEX;                    (7)
    pevent->OSEventCnt     = (prio << 8) | OS_MUTEX_AVAILABLE;       (8)
    pevent->OSEventPtr     = (void *)0;                              (9)
    OSEventWaitListInit(pevent);                                     (10)
    *err                   = OS_NO_ERR;
    return (pevent);                                                 (11)
}
}
```

L8.2(1)　OSMutexCreate() 从确认不是被中断服务子程序调用开始。因为不允许在中断服务子程序中调用该函数。

L8.2(2)　基于应用程序在 OS_CFG.H 文件中定义的最低任务优先级的值，函数 OSMutexCreate()判定 PIP 参数值是否在有效值范围内。

L8.2(3)　OSMutexCreate()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。

L8.2(4)　如果该指针可以使用，则 OSMutexCreate()通过给 OSTCBPrioTbl[prio]赋以非空指针，将这个优先级保留下来，以防止使用这个优先级建立别的任务，或者其他 mutex 使用这个优先级。

L8.2(5) OSMutexCreate()试图从空余事件控制块列表中得到一个事件控制块 ECB。

L8.2(6) 空余事件控制块列表调整到指向下一个空余控制块。

L8.2(7) 如果有空余事件控制块可以使用，则事件控制块 ECB 的类型被设置为 OS_EVENT_TYPE_MUTEX。μC/OS-II 中的其他服务会检验这个参数并确认这个 ECB 的类型是否正确。这种检验可以防止在调用 OSMutexPost()时，使用了诸如为消息邮箱建立的事件控制块。

L8.2(8) OSMutexCreate()置 mutex 的值为有效，同时将 PIP 保存起来。

值得注意的是，事件计数器 OSEventCnt 在此用法不同。特别是，这里 OSEventCnt 的高 8 位用于保存 PIP 的值；而低 8 位在资源无任务占用时的值为 0xFF，有任务占用时为占用 mutex 的任务的优先级。这种设置避免了 OS_EVENT 数据结构增加额外的空间，减少了 μC/OS-II 对 RAM 空间的占用量。

L8.2(9) 由于 mutex 正在被初始化，故没有等待这个 mutex 的任务。

L8.2(10) 调用 OSEventWaitListInit()，初始化等待任务列表。

L8.2(11) OSMutexCreate()返回指向 ECB 的指针。这个指针在有关 mutex 的一系列调用与操作（OSMutexPend()、OSMutexPost()、OSMutexAccept()、OSMutexDel()以及 OSMutexQuery()）中是必须使用的。这个指针实际上是处理 mutex 的句柄。如果没有空余的 ECB 可以使用，则 OSMutexCreate()将返回空指针 NULL。

图 F8.2 示意 ECB 在 OSMutexCreate()函数返回前的数据结构。

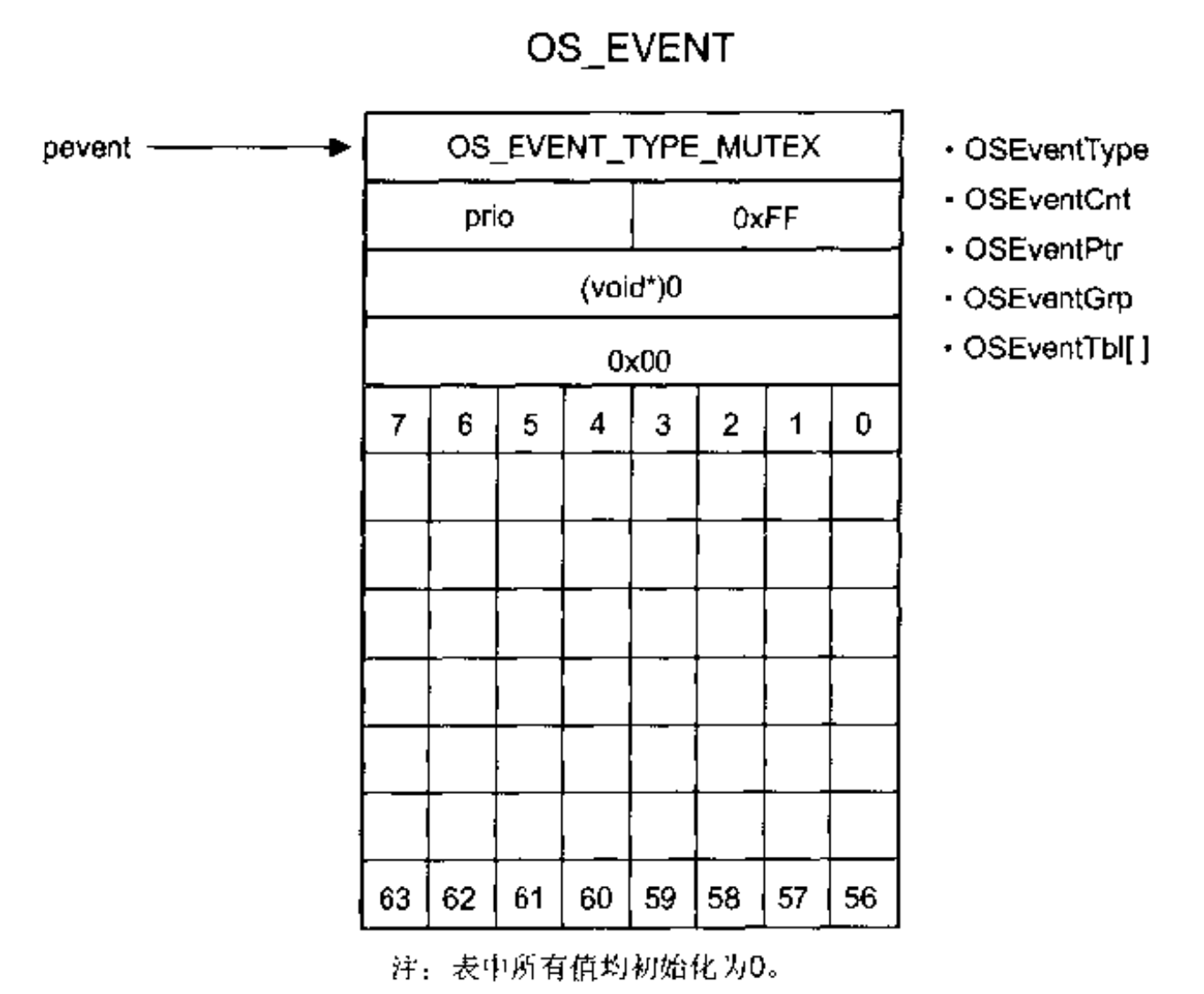


图 F8.2 ECB 在 OSMutexCreate()函数返回前的数据结构

8.01　删除一个互斥型信号量，OSMutexDel()

程序清单 L8.3 列出了删除一个 mutex 的代码。只有当 OS_CFG.H 文件中的 OS_MUTEX_DEL_EN这一项置 1 时，该项服务才有效。使用这项功能很危险，因为多任务可能试图使用一个已经删除了的 mutex。使用这个函数时要格外小心。一般说来，删除一个 mutex 之前，应先删除可能用到该 mutex 的所有任务。

程序清单 L8.3　删除一个互斥型信号量

```
OS_EVENT    *OSMutexDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    tasks_waiting;


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_DEL_ISR;
        return (pevent);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                                   (2)
        *err  = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {                (3)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:
```

```
        if (tasks_waiting == FALSE) {                                   (5)
            pevent->OSEventType = OS_EVENT_TYPE_UNUSED;                 (6)
            pevent->OSEventPtr  = OSEventFreeList;                      (7)
            OSEventFreeList     = pevent;
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
            return ((OS_EVENT *)0);                                     (8)
        } else {
            OS_EXIT_CRITICAL();
            *err = OS_ERR_TASK_WAITING;
            return (pevent);
        }

    case OS_DEL_ALWAYS:                                                 (9)
        while (pevent->OSEventGrp != 0x00) {                            (10)
            OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
        }
        pevent->OSEventType = OS_EVENT_TYPE_UNUSED;                     (11)
        pevent->OSEventPtr  = OSEventFreeList;                          (12)
        OSEventFreeList     = pevent;
        OS_EXIT_CRITICAL();
        if (tasks_waiting == TRUE) {                                    (13)
            OS_Sched();
        }
        *err = OS_NO_ERR;
        return ((OS_EVENT *)0);                                         (14)

    default:
        OS_EXIT_CRITICAL();
        *err = OS_ERR_INVALID_OPT;
        return (pevent);
}
}
```

L8.3(1) OSMutexDel()确认这个函数不是由中断服务子程序调用的，因为这是不允许的。

L8.3(2)

L8.3(3) 检验传递的参数 pevent 不应该是一个 NULL 指针，pevent 应该指向一个 mutex。

L8.3(4) OSMutexDel()看是否有任务在等待该 mutex，并设定相应标志 task_waiting。

基于调用时设定的选项(opt)，OSMutexDel()或者在没有任务因这个 mutex 而挂起的情况下删除它(opt = OS_DEL_NO_PEND)；或者强行删除，不管是否有任务在等待这个 mutex(opt = OS_DEL_ALWAYS)。

L8.3(5)

L8.3(6)

L8.3(7)　当 opt 设置成 OS_DEL_NO_PEND，并且没有任务在等待这个 mutex 时，OSMutexDel()将 ECB 标明为没有使用，并被送回到空余事件控制块列表中。这种处理允许以这个 ECB 来创建另一个 mutex，或者用于任何基于 ECB 的其他对象。请注意，OSMutexDel()返回了一个空指针 NULL [L8.3(8)]，因为此时的 mutex 已经不能够再通过原来的指针处置了。

L8.3(9)

L8.3(10)　当 opt 设为 OS_DEL_ALWAYS 时，所有等待 mutex 的任务都被置为就绪态，每个任务都认为自己得到了 mutex。当然这是一种很危险的状态，因为使用 mutex 的全部意义就在于防止对资源的多重占有。这里再次强调，在删除 mutex 之前，应先删除所有会用到这个 mutex 的任务。

L8.3(11)

L8.3(12)　所有挂起的任务都进入就绪态以后，OSMutexDel()将 ECB 标明为没有被占用，并被送回到空余事件控制块列表中。

L8.3(13)　只有当有任务在等待这个 mutex 时，才调用调度函数。

L8.3(14)　注意：OSMutexDel()返回一个空指针 NULL，因为此时的 mutex 已经不能够再通过原来的指针处置了。

8.02　等待一个互斥型信号量(挂起)，OSMutexPend()

等待 mutex 的程序代码如程序清单 L8.4 所列。

程序清单 L8.4　等待一个互斥型信号量

```
void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      pip;
    INT8U      mprio;
    BOOLEAN    rdy;
    OS_TCB    *ptcb;

    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_PEND_ISR;
        return;
    }
#if OS_ARG_CHK_EN
```

```
    if (pevent == (OS_EVENT *)0) {                                          (2)
        *err  = OS_ERR_PEVENT_NULL;
        return;
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {                       (3)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return;
    }
#endif                                                                      (4)
if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
    pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;                            (5)
    pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;                              (6)
    pevent->OSEventPtr  = (void *)OSTCBCur;                                 (7)
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return;
}

pip  = (INT8U)(pevent->OSEventCnt >> 8);                                    (8)
mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);                (9)
ptcb  = (OS_TCB *)(pevent->OSEventPtr);                                     (10)

if (ptcb->OSTCBPrio != pip && mprio > OSTCBCur->OSTCBPrio) {                (11)
    if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {               (12)
                                                                            (13)

        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
            OSRdyGrp &= ~ptcb->OSTCBBitY;
        }
        rdy = TRUE;                                                         (14)
    } else {
        rdy = FALSE;                                                        (15)
    }
    ptcb->OSTCBPrio = pip;                                                  (16)
    ptcb->OSTCBY    = ptcb->OSTCBPrio >> 3;
    ptcb->OSTCBBitY = OSMapTbl[ptcb->OSTCBY];
    ptcb->OSTCBX    = ptcb->OSTCBPrio & 0x07;
    ptcb->OSTCBBitX = OSMapTbl[ptcb->OSTCBX];
    if (rdy == TRUE) {                                                      (17)
```

```
            OSRdyGrp               |= ptcb->OSTCBBitY;
            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        }
        OSTCBPrioTbl[pip] = (OS_TCB *)ptcb;
    }
    OSTCBCur->OSTCBStat |= OS_STAT_MUTEX;                                   (18)
    OSTCBCur->OSTCBDly  = timeout;                                          (19)
    OS_EventTaskWait(pevent);                                               (20)
    OS_EXIT_CRITICAL();
    OS_Sched();                                                             (21)
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStat & OS_STAT_MUTEX) {                              (22)
        OS_EventTO(pevent);                                                 (23)
        OS_EXIT_CRITICAL();
        *err = OS_TIMEOUT;                                                  (24)
        return;
    }
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                                (25)
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
```

L8.4(1)　如同所有 μC/OS-II 对于 pend 的调用，OSMutexPend()不能在中断服务子程序中调用。这里 OSMutexPend()首先检验调用条件。

L8.4(2)

L8.4(3)　假定配置常数 OS_ARG_CHK_EN 是设定为 1 的，OSMutexPend()确认 pevent 句柄不是空指针 NULL，而是指向 OSMutexCreate()函数已经建立的事件控制块 ECB。

L8.4(4)

L8.4(5)

L8.4(6)　如果 .OSEventCnt 的低 8 位等于 0xFF(等于 OS_MUTEX_AVAILABLE)，这个 mutex 有效。OSMutexPend()将 mutex 赋予调用该函数的任务，且 OSMutexPend()置 .OSEventCnt 的低 8 位等于调用该函数的任务的优先级。

L8.4(7)　OSMutexPend()让 .OSEventPtr 指向调用该函数任务的任务控制块 TCB，然后返回。至此，调用该函数的任务可以开始与共享资源打交道了，因为返回出错代码为没有错误(OS_NO_ERR)。显然，如果想得到 mutex，这是最好的情况。这也是 OSMutexPend()执行最快的情况和正常的情况。

如果 mutex 被另外的任务占用，那么调用任务应进入休眠态，直到占用 mutex 的任务释放了 mutex(见 OSMutexPost())。OSMutexPend()允许定义一个超时时限值，作为参数之一传递给该函数。这种功能可以避免无限等待 mutex。如果传入的值是一个非 0 值，则 OSMutexPend()将任务挂起，直到

mutex 可以使用或定义的超时时间到。

注意：定义超时时限值为 0，表示任务将无限期地等待 mutex。

L8.4(8)

L8.4(9)

L8.4(10) 在调用让任务进入休眠态程序之前，OSMutexPend()提取 mutex 中的 PIP、占用 mutex 任务的优先级以及指向占用 mutex 任务的任务控制块指针。

L8.4(11) 如果占用 mutex 的任务的优先级比调用 OSMutexPend()函数任务的优先级低（数值大），占用 mutex 的任务的优先级就被提升到 mutex 的优先级继承优先级 PIP。意图在于让占用 mutex 的任务从速释放 mutex。

L8.4(12) OSMutexPend()确认占用 mutex 的任务是否进入就绪态。

L8.4(13)

L8.4(14) 如果该任务处于就绪态，那么这个任务已不是处在它原来优先级上的就绪态，是以置 rdy 标志，指明占用 mutex 的任务进入了就绪态，可以运行。

L8.4(15) 如果任务没有进入就绪态，rdy 置位与否就要看情况了。置位这个标志是为了表明，需要让这个任务在新的、较高的优先级（PIP）上进入就绪态。

L8.4(16) OSMutexPend()在优先级的继承优先级（PIP）上处理任务控制块。读者可能注意到，为了节省处理时间，在建立 mutex 时，已经在 OS_EVENT 的数据结构中保存了这些信息；然而，保存就意味着每个 OS_EVENT 又多占了若干 RAM。

L8.4(17) 根据这些信息以及 rdy 标志的状态，确定占用 mutex 的任务是否应在 PIP 优先级上进入就绪态。

L8.4(18) 为让调用本函数的任务进入休眠态，OSMutexPend()将该任务的任务控制块中的状态标志置位，以标明任务因等待 mutex 而挂起。

L8.4(19) 等待超时参数也保存在任务控制块中，这个参数由 OSTimeTick()做递减操作。或许还记得，如果该参数不为 0，OSTimeTick()每个时钟节拍将已建立任务的 .OSTCBDly 项数值减 1。

L8.4(20) 让任务进入休眠态的工作实际上是 OS_EventTaskWait()做的。

L8.4(21) 因为调用本函数的任务已经不再处于就绪态，调用调度函数，以运行下一个优先级最高的进入就绪态的任务。

当 mutex 被释放（或者等待超时）时，调用 OSMutexPend()的任务又变成优先级最高的任务了，OS_Sched()函数返回。

L8.4(22) OSMutexPend()检查任务控制块的状态标志，看是否还是置位，还在等待 mutex。如果任务还在等待 mutex，说明一定是没有从 OSMutexPost()函数调用中得到 mutex，而是由于 OSTimeTick()函数进入就绪态，出现等待超时。

L8.4(23)

L8.4(24) 在这种情况下，通过调用 OS_EventTO()函数，将任务从等待 mutex 的任务列表中删除，并将出错代码返回给任务，指明出现了等待超时错误。

如果任务控制块中的状态标志 OS_STAT_MUTEX 位不再是置位状态，

说明 mutex 已经被释放了，调用 OSMutexPend( )函数的任务可以断定自己已经得到了 mutex。

L8.5(25)　释放事件控制块指针。

8.03　释放一个互斥型信号量，OSMutexPost( )

释放互斥型信号量(signal a mutex)的代码如程序清单 L8.5 所列。

程序清单 L8.5　释放一个互斥型信号量

```
INT8U OSMutexPost (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      pip;
    INT8U      prio;


    if (OSIntNesting > 0) {                                          (1)
        return (OS_ERR_POST_ISR);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                                   (2)
        return (OS_ERR_PEVENT_NULL);
    }
#endif
    OS_ENTER_CRITICAL();
    pip  = (INT8U)(pevent->OSEventCnt >> 8);
    prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {                (3)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (OSTCBCur->OSTCBPrio != pip ||
        OSTCBCur->OSTCBPrio != prio) {                               (4)
        OS_EXIT_CRITICAL();
        return (OS_ERR_NOT_MUTEX_OWNER);
    }
#endif
```

```
    if (OSTCBCur->OSTCBPrio == pip) {                                    (5)

                                                                         (6)
        if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBPrio            = prio;
        OSTCBCur->OSTCBY               = prio >> 3;
        OSTCBCur->OSTCBBitY            = OSMapTbl[OSTCBCur->OSTCBY];
        OSTCBCur->OSTCBX               = prio & 0x07;
        OSTCBCur->OSTCBBitX            = OSMapTbl[OSTCBCur->OSTCBX];
        OSRdyGrp                      |= OSTCBCur->OSTCBBitY;
        OSRdyTbl[OSTCBCur->OSTCBY]    |= OSTCBCur->OSTCBBitX;
        OSTCBPrioTbl[prio]             = (OS_TCB *)OSTCBCur;
    }
    OSTCBPrioTbl[pip] = (OS_TCB *)1;
    if (pevent->OSEventGrp != 0x00) {                                    (7)
                                                                         (8)
        prio                 = OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
        pevent->OSEventCnt  &= 0xFF00;                                   (9)
        pevent->OSEventCnt  |= prio;
        pevent->OSEventPtr   = OSTCBPrioTbl[prio];
        OS_EXIT_CRITICAL();
        OS_Sched();                                                      (10)
        return (OS_NO_ERR);
    }
    pevent->OSEventCnt |= 0x00FF;                                        (11)
    pevent->OSEventPtr = (void *)0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L8.5(1) 互斥型信号量只能用于任务，故须检测一下，以确认OSMutexPost()不是被中断服务程序调用。

L8.5(2)

L8.5(3) 设配置常数OS_ARG_CHK_EN是置1的，OSMutexPost()确定pevent的句柄不是空指针（指向NULL），而是指向OSMutexCreate()函数已经建立了的事件控制块ECB。

L8.5(4) OSMutexPost()确认，释放mutex的任务确实是占用mutex的任务。占用mutex的任务的优先级或者是被提升到PIP（OSMutexPend()函数已经将该任务的优先级升高），或仍是保存在mutex之中的优先级。

L8.5(5) OSMutexPost()查看占用mutex的任务的优先级是否已经升到了PIP，因为有个高优先级的任务也需要这个mutex。在这种情况下，占用mutex的任务的优先级被降到了原来的优先级。原来的优先级是从.OSEventCnt的低8位中得到的。

L8.5(6) 将调用本函数的任务从任务就绪表中PIP位置上删去，放回到任务就绪表原来的优先级位置上。

注意：原任务优先级的任务控制块将被重新处理。

L8.5(7) 查看是否有正在等待mutex的任务。ECB中的.OSEventGrp不为0，说明有任务在等待mutex。

L8.5(8) OS_EventTaskRdy()将最高优先级的任务从等待mutex的任务列表中删除(见6.05节："使一个任务进入就绪态.OS_EventTaskRdy()")，于是这个任务就进入了就绪态。

L8.5(9) 新的占用mutex任务的优先级被保存在mutex的ECB中。

L8.5(10) 调用OS_Sched()，查看这个进入就绪态的任务是否是当前就绪态任务中优先级最高的。如果是，则引起任务切换，这个就绪态的任务于是得以继续运行；如果进入就绪态的任务不是优先级最高的任务，则OS_Sched()返回，继续执行调用OSMutexPost()的任务。

L8.5(11) 如果没有等待mutex的任务，则.OSEventCnt的低8位被置为0xFF，表明mutex有效，立即可用。

8.04 无等待地获取互斥型信号量(任务不挂起)，OSMutexAccept()

如果要得到mutex，而当前mutex无效，那么不让任务进入休眠态也是可以的。这种功能可以通过调用OSMutexAccept()来实现。这个函数的代码如程序清单L8.6所列。

程序清单L8.6 无等待地获取互斥型信号量

```
INT8U OSMutexAccept (OS_EVENT *pevent, INT8U *err)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_PEND_ISR;
        return (0);
```

```
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {
        *err  = OS_ERR_PEVENT_NULL;
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
                                                                                    (2)
    if ((pevent->OSEventCnt &OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;                                (3)
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
        pevent->OSEventPtr  = (void *)OSTCBCur;                                     (4)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (1);
    }
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (0);
}
```

L8.6(1) 同其他调用一样，如果 OS_CFG.H 文件中的 OS_ARG_CHK_EN 项置 1，则 OSMutexAccept()开始于确认该函数不被中断服务子程序调用，并且做参数定义域值检查。

L8.6(2) OSMutexAccept()检查 mutex 是否有效(.OSEventCnt 的低 8 位为 0xFF)。

L8.6(3)

L8.6(4) 如果 mutex 有效，那么 OSMutexAccept()把调用该函数任务的优先级写到.OSEventCnt的低 8 位，并将 mutex 的事件控制块 ECB 链接到该任务的任务控制块，于是该任务得到了 mutex。

调用 OSMutexAccept()的代码应检查返回值。返回值为 0，表明 mutex 无效，不能使用；返回值为 1，表明 mutex 已经得到，可以使用相应的共享资源。

8.05　获取互斥型信号量的当前状态，OSMutexQuery()

OSMutexQuery()使应用程序看到用于互斥型信号量 mutex 的事件控制块的当前状况。这个函数的代码如程序清单 L8.7 所列。

程序清单 L8.7　获取 mutex 的当前状态

```
INT8U OSMutexQuery (OS_EVENT *pevent, OS_MUTEX_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    INT8U        *psrc;
    INT8U        *pdest;

    if (OSIntNesting > 0) {                                          (1)
        return (OS_ERR_QUERY_ISR);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                                   (2)
        return (OS_ERR_PEVENT_NULL);
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {                (3)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    pdata->OSMutexPIP    = (INT8U)(pevent->OSEventCnt >> 8);         (4)
    pdata->OSOwnerPrio   = (INT8U)(pevent->OSEventCnt & 0x00FF);
    if (pdata->OSOwnerPrio == 0xFF) {
        pdata->OSValue   = 1;                                        (5)
    } else {
        pdata->OSValue   = 0;                                        (6)
    }
    pdata->OSEventGrp    = pevent->OSEventGrp;                       (7)
    psrc                 = &pevent->OSEventTbl[0];
    pdest                = &pdata->OSEventTbl[0];
#if OS_EVENT_TBL_SIZE > 0
```

```
    *pdest ++          =  *psrc ++ ;
#endif

#if OS_EVENT_TBL_SIZE > 1
    *pdest ++          =  *psrc ++ ;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *pdest ++          =  *psrc ++ ;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *pdest ++          =  *psrc ++ ;
#endif

#if OS_EVENT_TBL_SIZE > 4
    *pdest ++          =  *psrc ++ ;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *pdest ++          =  *psrc ++ ;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *pdest ++          =  *psrc ++ ;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *pdest             =  *psrc;
#endif
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

OSMutexQuery()接受 2 个代入参数：pevent 是指向 mutex 的指针，这个指针是建立 mutex时 OSMutexCreate()函数返回的值；pdata 是指向数据结构 OS_MUTEX_DATA 的指针（见 uCOS_II.H），该数据结构存有 mutex 的有用信息。故应用程序中应分配一个类型为 OS_MUTEX_DATA 的变量，存储 mutex 的相关信息。笔者决定使用一个新的数据结构，因为本函数的调用者应该只关心与 mutex 有关的数据，而不是通用的 OS_EVENT 数据结构中的全部数据。OS_MUTEX_DATA 包括 mutex 的优先级继承优先级 PIP（.OSMutexPIP）、占用 mutex 的任务的优先级（.OSMutexPrio）及 mutex 的值（.OSMutexValue）。该值为 1，表示 mutex 有效，可以使用；如果为 0，则 mutex 无效，已被别的任务占用。

注意：若 mutex 没被任何任务占用，则.OSMutexPrio 的值为 0xFF。OS_MUTEX_DATA 还包含等待 mutex 的任务列表(.OSEventTbl[]和 .OSEventGrp)。

L8.7(1)　如同所有 mutex 的调用，OSMutexQuery()查看调用是否来自中断服务子程序。

L8.7(2)

L8.7(3)　若配置常数 OS_ARG_CHK_EN 为 1，则 OSMutexQuery() 确认句柄 pevent 不是空指针 NULL。OSMutexCreate()已经建立了 mutex，指针已指向相关的 ECB。

L8.7(4)　OSMutexQuery()将相应的参数传给 OS_MUTE_DATA，mutex 的 PIP 由 .OSEnentCnt的高 8 位得到。

L8.7(5)　从.OSEventCnt 的低 8 位得到 mutex 的值。如果 mutex 有效(低 8 位是 0xFF)，将 mutex 的值设为 1。

L8.7(6)　否则 mutex 的值为 0(由于被任务占用，mutex 无效)。

L8.7(7)　将等待 mutex 任务列表复制到 OS_MUTEX_DATA 中。为使性能更好一些，没有使用 for 循环语句，而使用了行代码条件编译。

第 9 章

事件标志组管理

μC/OS－II 的事件标志组(event flag)由 2 部分组成：一是用来保存当前事件组中各事件状态的一些标志位；二是等待这些标志位置位或清除的任务列表。μC/OS－II 提供了 6 个函数，完成事件标志组的各种功能，这些函数是：OSFlagAccept()，OSFlagCreate()，OSFlagDel()，OSFlagPend()，OSFlagPost()及 OSFlagQuery()。

为使 μC/OS－II 在编译后生成具有事件标志组功能的代码，必须在 OS_CFG.H 中选定事件标志组控制选项。表 T9.1 中列出了 OS_CFG.H 文件中的事件标志组控制开关量选项。如果 OS_FLAG_EN 选项被禁止，则表 T9.1 中的所有选项也都默认为禁止。要允许这些选项，只需简单地将这些选项重新定义为 1 即可。OSFlagCreate()，OSFlagPend()及 OSFlagPost()是事件标志组最基本的功能，不能单独禁止。

表 T9.1　OS_CFG.H 中事件标志组的配置常量

μC/OS－II 事件标志组函数	OS_CFG.H 中置为 1 允许相应函数
OSFlagAccept()	OS_FLAG_ACCEPT_EN
OSFlagCreate()	
OSFlagDel()	OS_FLAG_DEL_EN
OSFlagPend()	
OSFlagPost()	
OSFlagQuery()	OS_FLAG_QUERY_EN

图 F9.1 展示了任务、中断及事件标志组之间的联系。这里用了一个 8 位的序列表示事件标志组，实际上事件标志组可以是 8 位，16 位或 32 位的。时钟沙漏表示 OSFlagPend() 支持超时机制。

从图 F9.1 可以看到，任务或中断都可以调用函数 OSFlagAccept()，OSFlagPost()或者 OSFlagQuery()；但只有任务才可以调用函数 OSFlagCreate()，OSFlagDel()及 OSFlagPend()。

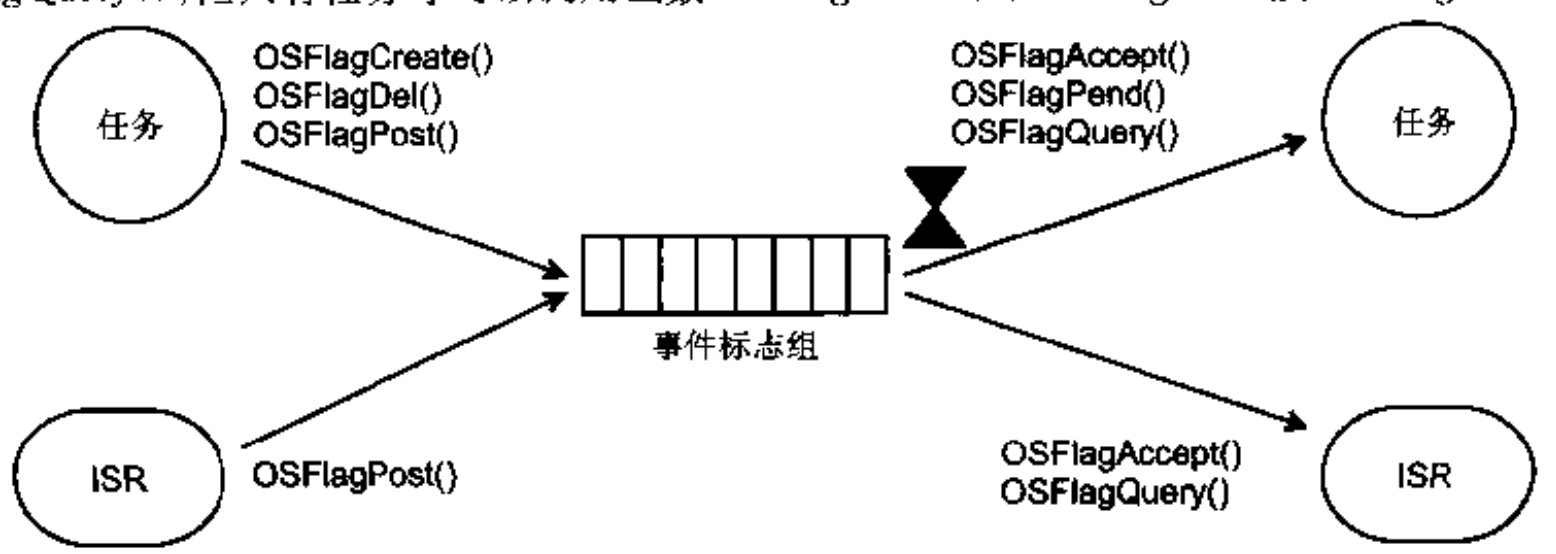


图 F9.1　μC/OS－II 事件标志组功能

9.00　深入事件标志组

在 OS_FLAG_GRP 数据结构中可以看到，μC/OS - II 的事件标志组对应的数据结构包含 3 个成员，如程序清单 L9.1 所列。

程序清单 L9.1　事件标志组数据结构

```
typedef struct {
    INT8U      OSFlagType;                                    (1)
    void      *OSFlagWaitList;                                (2)
    OS_FLAGS   OSFlagFlags;                                   (3)
} OS_FLAG_GRP;
```

L9.1(1)　.OSFlagType 变量用来检验指针的类型是否是指向事件标志组的指针。这个变量是事件标志组数据结构的第 1 个成员，这样便于 μC/OS - II 检验指针指向的数据类型。

例如，如果把一个指向事件标志组的指针作为参数传递给 OSSemPend() 函数，μC/OS - II 将返回一个指向错误数据结构类型的出错代码。

注意：事件控制块 ECB(Event Control Block)的第一个字节也是一个标志该事件控制块类型(信号量、mutex、邮箱或消息队列)的变量。

L9.1(2)　.OSFlagWaitList 包含了一个等待事件标志组的任务列表。

L9.2(3)　.OSFlagFlags 包含了一系列表明当前事件标志状态的位。这些位的数目在编译时由 OS_CFG.H 文件中的 OS_FLAGS 常量决定，可以为 8 位、16 位或 32 位。

这里使用的等待事件标志组的任务列表和 μC/OS - II 里用到的其他等待任务列表不同。这里的任务列表是一个双向链表，如图 F9.2 所示，使用了 3 个数据结构：OS_FLAG_GRP、上面刚刚提到的任务控制块 OS_TCB 以及 OS_FLAG_NODE。这个数据结构用来记录任务在等待哪些事件标志位及等待的方式("与"或者"或")。在这里，大量使用了指针。

当一个任务开始等待某些事件标志位时，就建立一个 OS_FLAG_NODE 数据结构。当这些等待的事件标志位发生后，这个数据结构被删除。换句话说，当调用 OSFlagPend()时，建立 OS_FLAG_NODE。

OS_FLAG_NODE 数据结构如程序清单 L9.2 所列。

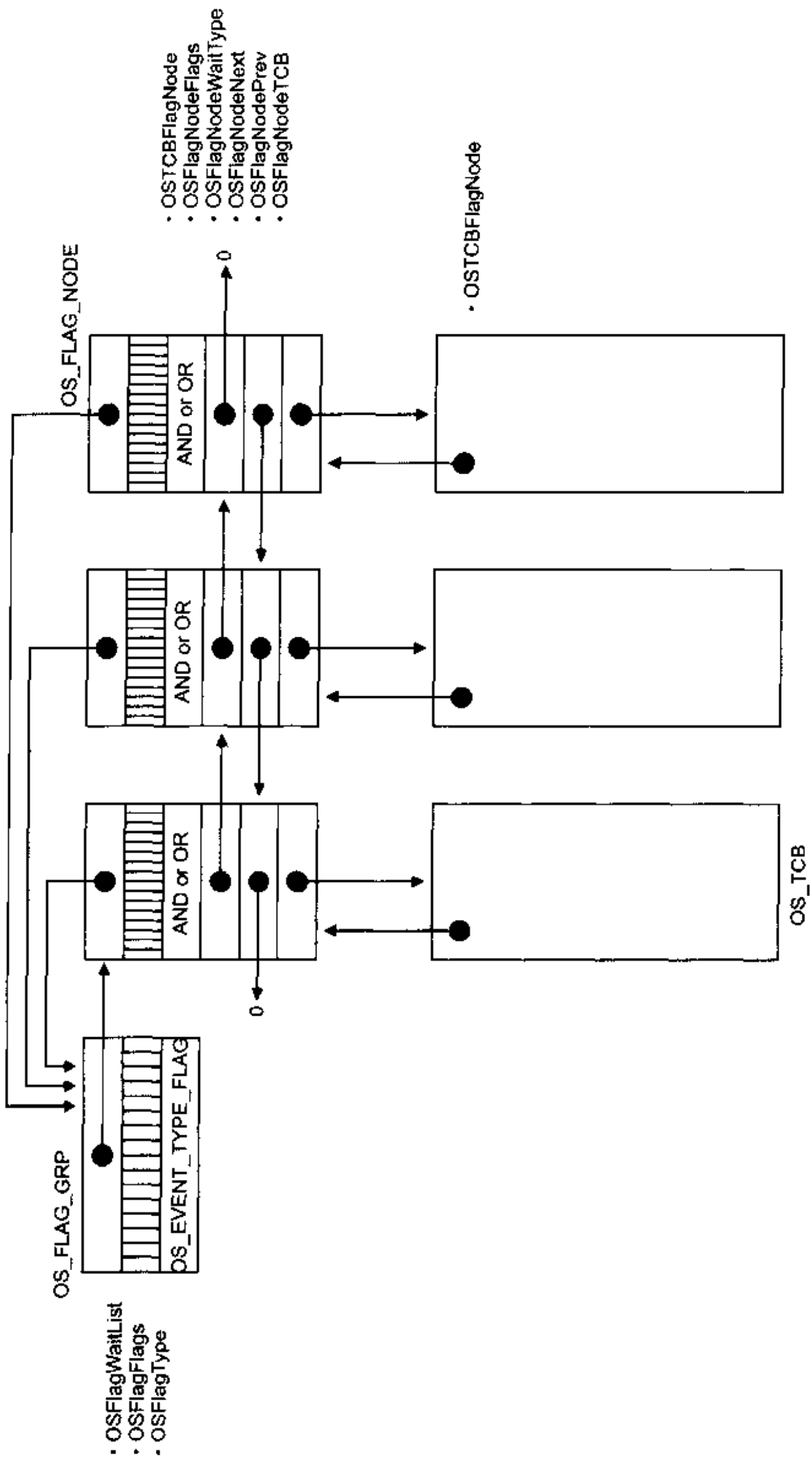


图 F9.2 事件标志组、事件标志节点及任务控制块之间的关系

程序清单 L9.2　OS_FLAG_NODE 数据结构

```
typedef struct {
    void        *OSFlagNodeNext;                                    (1)
    void        *OSFlagNodePrev;
    void        *OSFlagNodeTCB;                                     (2)
    void        *OSFlagNodeFlagGrp;                                 (3)
    OS_FLAGS     OSFlagNodeFlags;                                   (4)
    INT8U        OSFlagNodeWaitType;                                (5)
} OS_FLAG_NODE;
```

L9.2(1)　.OSFlagNodeNext 和 .OSFlagNodePrev 用来构建双向OS_FLAG_NODE数据结构链表。这种双向链表可以很方便地向链表中插入或从链表中删除一个OS_FLAG_NODE 结构。

L9.2(2)　.OSFlagNodeTCB 指针指向某个等待事件标志组中的事件标志的任务控制块。换句话说，通过这个指针，可以知道哪一个任务在等待事件标志组中的事件。

L9.2(3)　.OSFlagNodeFlagGrp 是一个反向指向事件标志组的指针。在删除一个OS_FLAG_NODE数据结构或通过 OSTaskDel()删除一个仍然在等待事件标志的任务时，将用到这个指针。

L9.2(4)　.OSFlagNodeFlags 用来指明任务等待事件标志组中的哪些事件标志。例如，任务可能调用 OSFlagPend()，并指明等待事件标志组中的事件标志 0、4、6 及 7。在这种情形下，OSFlagNodeFlags 的值为 0xD1。根据由 OS_FLAGS 指定的事件标志组的位数，OSFlagNodeFlags 可能为 8 位、16 位或者 32 位。因为 μC/OS-II 及其移植是以源代码的形式提供的，可以很方便地根据实际产品的需要更改事件标志组的位数。使用 8 位的事件标志组，可以减少程序占用的 RAM 和 ROM 空间；但为了程序的可移植性，推荐将 OS_FLAGS 定义为 INT32U 类型。

L9.2(5)　OS_FLAG_NODE 数据结构的最后一个变量是 OSFlagNodeWaitType。这个变量指明任务是等待事件标志组中的所有事件标志都发生(AND－"与"型)，还是任何一个事件标志发生(OR－"或"型)。OSFlagNodeWaitType 可以为下面的值：

```
OS_FLAG_WAIT_CLR_ALL
OS_FLAG_WAIT_CLR_AND

OS_FLAG_WAIT_CLR_ANY
OS_FLAG_WAIT_CLR_OR

OS_FLAG_WAIT_SET_ALL
```

```
OS_FLAG_WAIT_SET_AND

OS_FLAG_WAIT_SET_ANY
OS_FLAG_WAIT_SET_OR
```

注意:AND 和 ALL 的意义是相同的,两者都可以使用。

这里推荐使用 OS_FLAG_WAIT_???_ALL 的方式,因为这种定义方法更为清晰;而OS_FLAG_WAIT_???_AND 也是没有任何问题的。类似地,OR 和 ANY 的意义也是相同的,并且推荐使用 OS_FLAG_WAIT_???_ANY 的定义方法。另外,任务等待事件可以是等待事件置起(为 1),也可以是等待事件清除(为 0)。

9.01 建立一个事件标志组,OSFlagCreate()

建立事件标志组的代码列在程序清单 L9.3 中。

程序清单 L9.3 建立一个事件标志组

```
OS_FLAG_GRP   *OSFlagCreate (OS_FLAGS flags, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR     cpu_sr;
#endif
    OS_FLAG_GRP *pgrp;



    if (OSIntNesting > 0) {                                                    (1)
        *err = OS_ERR_CREATE_ISR;
        return ((OS_FLAG_GRP *)0);
    }
    OS_ENTER_CRITICAL();
    pgrp = OSFlagFreeList;                                                     (2)
    if (pgrp != (OS_FLAG_GRP *)0) {                                            (3)
                                                                               (4)
        OSFlagFreeList        = (OS_FLAG_GRP *)OSFlagFreeList->OSFlagWaitList;
        pgrp->OSFlagType      = OS_EVENT_TYPE_FLAG;                            (5)
        pgrp->OSFlagFlags     = flags;                                         (6)
        pgrp->OSFlagWaitList  = (void *)0;                                     (7)
        OS_EXIT_CRITICAL();
        *err                  = OS_NO_ERR;
    } else {
```

```
        OS_EXIT_CRITICAL();
        *err                    = OS_FLAG_GRP_DEPLETED;
    }
    return (pgrp);                                                    (8)
}
```

L9.3(1)　不允许在中断中调用 OSFlagCreate()函数，这里首先做一个检查。

L9.3(2)　OSFlagCreate()从系统的空闲事件标志组链表中取得一个空闲的事件标志组。

L9.3(3)　如果返回一个 NULL 的指针，则说明系统已经没有空闲的事件标志组可以分配。

L9.3(4)　分配了一个空闲的事件标志组后，调整系统的空闲事件标志组链表指针。系统空闲事件标志组链表中的总的事件标志组数目由 OS_CFG.H 文件中的 OS_MAX_FLAGS常数定义。

L9.3(5)　OSFlagCreate()函数初始化事件标志组数据结构的各个变量。OS_EVENT_TYPE_FLAG指明这个数据结构是一个事件标志组。因为这个变量是数据结构的第 1 个变量，所以它相对事件标志组数据结构指针的偏移量是 0。在 μC/OS-II 中，事件标志组数据结构和事件控制块(信号量、邮箱、消息队列及互斥型信号量需要使用事件控制块)的第一个字节都是一个表明自身类型的变量。这里检查这个变量是否是事件标志组类型，可以保证对正确的事件标志组数据结构进行操作。

L9.3(6)　OSFlagCreate()使用传入的参数 flags，初始化 OSFlagNodeFlags。通常，会将 OSFlagNodeFlags 初始化为全 0；但如果任务是等待事件清除，也可以将 OSFlagNodeFlags初始化为全 1。

L9.3(7)　因为刚刚建立事件标志组，还没有任务等待这个事件标志组，所以等待任务链表指针初始化为 NULL。

L9.3(8)　OSFlagCreate()返回刚刚建立的事件标志组的指针。如果没有空闲的事件标志组，OSFlagCreate()将返回 NULL 指针。

图 F9.3 所示为调用 OSFlagCreate()后事件标志组的数据结构。

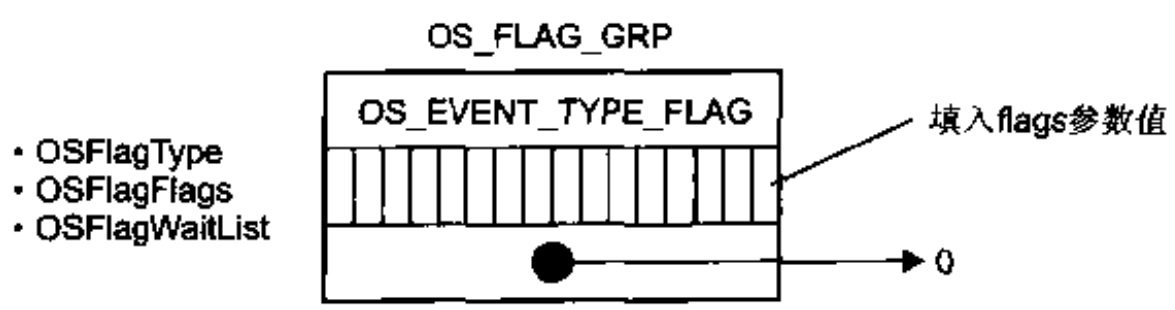


图 F9.3　调用 OSFlagCreate()后事件标志组的数据结构

9.02 删除一个事件标志组，OSFlagDel()

删除一个事件标志组的程序如程序清单 L9.4 所列。

程序清单 L9.4 删除事件标志组

```
OS_FLAG_GRP   *OSFlagDel (OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR       cpu_sr;
#endif
    BOOLEAN         tasks_waiting;
    OS_FLAG_NODE    *pnode;

    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_DEL_ISR;
        return (pgrp);
    }
#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                                  (2)
        *err = OS_FLAG_INVALID_PGRP;
        return (pgrp);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {                    (3)
        *err = OS_ERR_EVENT_TYPE;
        return (pgrp);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pgrp->OSFlagWaitList != (void *)0) {                         (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:                                         (5)
            if (tasks_waiting == FALSE) {
                pgrp->OSFlagType      = OS_EVENT_TYPE_UNUSED;
                pgrp->OSFlagWaitList  = (void *)OSFlagFreeList;      (6)
                OSFlagFreeList        = pgrp;
                OS_EXIT_CRITICAL();
```

```
            *err                = OS_NO_ERR;
            return ((OS_FLAG_GRP *)0);                                  (7)
        } else {
            OS_EXIT_CRITICAL();
            *err                = OS_ERR_TASK_WAITING;
            return (pgrp);
        }

    case OS_DEL_ALWAYS:                                                 (8)
        pnode = pgrp->OSFlagWaitList;
        while (pnode != (OS_FLAG_NODE *)0) {                            (9)
            OS_FlagTaskRdy(pnode, (OS_FLAGS)0);
            pnode = pnode->OSFlagNodeNext;
        }
        pgrp->OSFlagType        = OS_EVENT_TYPE_UNUSED;
        pgrp->OSFlagWaitList    = (void *)OSFlagFreeList;               (10)
        OSFlagFreeList          = pgrp;
        OS_EXIT_CRITICAL();
        if (tasks_waiting == TRUE) {                                    (11)
            OS_Sched();
        }
        *err = OS_NO_ERR;
        return ((OS_FLAG_GRP *)0);                                      (12)

    default:
        OS_EXIT_CRITICAL();
        *err = OS_ERR_INVALID_OPT;
        return (pgrp);
    }
}
```

使用这个函数需要注意，多任务系统中可能出现一个任务试图访问已经被删除的事件标志组。一般地说，在删除一个事件标志组之前，必须先删除所有可能用到这个事件标志组的任务。

L9.4(1)　不允许在中断服务子程序中调用 OSFlagDel()函数。这里首先做一个检查。

L9.4(2)

L9.4(3)　检查传递给 OSFlagDel()函数的参数是否正确。首先，pgrp 指针必须是一个非 NULL 指针，并且指向一个事件标志组。这些检查函数参数的代码可以通过编译选项控制，如果常量 OS_ARG_CHK_EN 被定义成 0，这些代码就不会被编译到执行代码中。这样可以减少程序的代码量。

L9.4(4) OSFlagDel()检查是否有任务在等待这个即将被删除的事件标志组，并将检查的结果保存在局部布尔型变量 tasks_waiting 中。

如果调用时选择不同的选项，那么 OSFlagDel()函数可以只在没有任务等待事件标志组的情况下，才删除事件标志组(选项为 OS_DEL_NO_PEND)；也可以不管是否有等待的任务，立即删除事件标志组(选项为 OS_DEL_ALWAYS)。

L9.4(5)

L9.4(6) 当调用选项是 OS_DEL_NO_PEND 并且没有等待事件标志组的任务时，OSFlagDel()把这个事件标志组标记为未使用，并且将这个事件标志组放回到系统空闲事件标志组链表中。这样可以允许再次分配和使用这个事件标志组。

L9.4(7) OSFlagDel()函数返回一个 NULL 指针。因为此时事件标志组已经被放回到系统空闲事件标志组链表中，不能再使用原来的指针访问这个事件标志组了。

L9.4(8)

L9.4(9) 当调用选项是 OS_DEL_ALWAYS 时，所有等待这个事件标志组的任务都会被标志为就绪态，认为任务等待的事件都发生了。将在讨论 OSFlagPost()函数时，进一步了解这里用到的 OS_FlagTaskRdy()函数。

L9.4(10) 在把所有等待该事件标志组的任务标志为就绪态以后，OSFlagDel()把这个事件标志组标志为未使用，并且将这个事件标志组放回到系统空闲事件标志组链表中。

L9.4(11) 如果根据前面得到的布尔变量 tasks_waiting 确定还有任务等待事件标志组，那么在删除该事件标志组后，将进行任务调度。

L9.4(12) OSFlagDel()函数返回一个 NULL 指针。因为此时事件标志组已经被放回到系统空闲事件标志组链表中，不能再使用原来的指针来访问这个事件标志组。

9.03 等待事件标志组的事件标志位，OSFlagPend()

等待一个事件标志组中的事件标志位的代码列在程序清单 L9.5 中。

程序清单 L9.5 等待事件标志组的事件标志位

```
OS_FLAGS  OSFlagPend (OS_FLAG_GRP *pgrp, OS_FLAGS flags,
                      INT8U wait_type, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR     cpu_sr;
#endif
    OS_FLAG_NODE  node;
    OS_FLAGS      flags_cur;
```

```
    OS_FLAGS        flags_rdy;
    BOOLEAN         consume;


    if (OSIntNesting > 0) {                                                 (1)
        *err = OS_ERR_PEND_ISR;
        return ((OS_FLAGS)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                                         (2)
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {                           (3)
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    if (wait_type & OS_FLAG_CONSUME) {                                      (4)
        wait_type &= ~OS_FLAG_CONSUME;
        consume   = TRUE;
    } else {
        consume   = FALSE;
    }
    OS_ENTER_CRITICAL();
    switch (wait_type) {                                                    (5)
        case OS_FLAG_WAIT_SET_ALL:
            flags_rdy = pgrp->OSFlagFlags & flags;                          (6)
            if (flags_rdy == flags) {                                       (7)
                if (consume == TRUE) {                                      (8)
                    pgrp->OSFlagFlags &= ~flags_rdy;                        (9)
                }
                flags_cur = pgrp->OSFlagFlags;                              (10)
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
                return (flags_cur);                                         (11)
            } else {                                                        (12)
                OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                OS_EXIT_CRITICAL();
            }
            break;

        case OS_FLAG_WAIT_SET_ANY:
```

```
            flags_rdy = pgrp->OSFlagFlags & flags;                          (13)
            if (flags_rdy != (OS_FLAGS)0) {                                 (14)
                if (consume == TRUE) {                                      (15)
                    pgrp->OSFlagFlags &= ~flags_rdy;                        (16)
                }
                flags_cur = pgrp->OSFlagFlags;                              (17)
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
                return (flags_cur);                                         (18)
            } else {                                                        (19)
                OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                OS_EXIT_CRITICAL();
            }
            break;

# if OS_FLAG_WAIT_CLR_EN > 0
        case OS_FLAG_WAIT_CLR_ALL:
            flags_rdy = ~pgrp->OSFlagFlags & flags;
            if (flags_rdy == flags) {
                if (consume == TRUE) {
                    pgrp->OSFlagFlags |= flags_rdy;
                }
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
                return (flags_cur);
            } else {
                OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                OS_EXIT_CRITICAL();
            }
            break;

        case OS_FLAG_WAIT_CLR_ANY:
            flags_rdy = ~pgrp->OSFlagFlags & flags;
            if (flags_rdy != (OS_FLAGS)0) {
                if (consume == TRUE) {
                    pgrp->OSFlagFlags |= flags_rdy;
                }
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
                return (flags_cur);
```

```
            } else {
                OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                OS_EXIT_CRITICAL();
            }
            break;
#endif

        default:
            OS_EXIT_CRITICAL();
            flags_cur = (OS_FLAGS)0;
            *err      = OS_FLAG_ERR_WAIT_TYPE;
            return (flags_cur);
    }
    OS_Sched();                                                          (20)
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStat & OS_STAT_FLAG) {                            (21)
        OS_FlagUnlink(&node);                                            (22)
        OSTCBCur->OSTCBStat = OS_STAT_RDY;
        OS_EXIT_CRITICAL();
        flags_cur           = (OS_FLAGS)0;
        *err                = OS_TIMEOUT;
    } else {
        if (consume == TRUE) {                                           (23)
            switch (wait_type) {
                case OS_FLAG_WAIT_SET_ALL:
                case OS_FLAG_WAIT_SET_ANY:                               (24)
                    pgrp->OSFlagFlags &= ~OSTCBCur->OSTCBFlagsRdy;
                    break;

                case OS_FLAG_WAIT_CLR_ALL:
                case OS_FLAG_WAIT_CLR_ANY:
                    pgrp->OSFlagFlags |= OSTCBCur->OSTCBFlagsRdy;
                    break;
            }
        }
        flags_cur = pgrp->OSFlagFlags;                                   (25)
        OS_EXIT_CRITICAL();
        *err      = OS_NO_ERR;
    }
    return (flags_cur);
}
```

L9.5(1) 在 μC/OS-II 中,不允许在中断服务子程序中调用本函数。这里首先检查一下是否是在中断中调用的。

L9.5(2)

L9.5(3) 假定 OS_ARG_CHK_EN 常量被定义为 1,即允许检查函数参数。函数 OSFlagPend()将检查 pgrp 是否是一个值为 NULL 的空指针,并且检查 pgrp 是否指向一个由 OSFlagCreate()函数建立的事件标志组数据结构。

OSFlagPend()函数允许指定在任务等待的事件发生后,重新置起或是清除相应的事件标志位。这是通过在调用 OSFlagPend()函数时将一个常量 OS_FLAG_CONSUME和参数 wait_type 相"加"(或者相"或")来实现的。例如,希望等待事件标志组的 BIT0 位置位,而此时事件标志组的 BIT0 位已经置位了,那么如果在调用 OSFlagPend()时,把参数 wait_type 加上 OS_FLAG_CONSUME,就能清除这个事件标志位。如下所示:

```
OSFlagPend(OSFlagMyGrp,
           (OS_FLAGS)0x01,
           FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME,
           0,
           &err);
```

L9.5(4) 因为对事件标志组中的事件标志位的处理是在后面的代码中进行的,这里仅仅把对事件标志位的处理方法保存在一个局部布尔变量 consume 中。

L9.5(5) OSFlagPend() 根据不同的等待事件标志组的方式进行不同的处理,这里可能有 4 种不同的处理方式:

- 等待所有指定的事件标志组中的事件标志位置 1;
- 等待任意一个指定的事件标志组中的事件标志位置 1;
- 等待所有指定的事件标志组中的事件标志位清 0;
- 等待任意一个指定的事件标志组中的事件标志位清 0。

除了等待事件标志位清 0 和置起的不同外,后面 2 种方式的处理与前面 2 种基本相同;所以,这里只讨论前面 2 种方式。实际上,为了节约代码空间,完全可以只使用前面 2 种方式。通过设定 OS_CFG.H 文件中的常量 OS_FLAG_WAIT_CLR_EN为 0,可以在编译时忽略后 2 种方式的代码。

等待所有指定的事件标志组中的事件标志位置 1:

L9.5(6) 当 wait_type 为 OS_FLAG_WAIT_SET_ALL 或 OS_FLAG_WAIT_SET_AND 时,OSFlagPend()函数取出事件标志组中由 flags 参数指定的事件标志位。

L9.5(7) 如果取出的事件标志位的状态恰好完全符合预期的状态,说明任务需要等待的事件标志位都已经置位。这种情况下,PEND 函数立即返回调用函数。

L9.5(8)

L9.5(9) 在返回前,需要根据前面的布尔变量 consume 决定,是否需要对指定的事件标

志位进行清 0 操作。

L9.5(10)

L9.5(11)　获取事件标志组的新的事件标志状态值，并返回调用函数。

L9.5(12)　如果指定的事件标志位没有全部置位，那么此时调用本函数的任务将被挂起，直到指定的事件标志位全部置位或者指定的等待超时时间到。在 4 种不同的方式里都有这样的处理。为了节省代码空间，这里编写了一个函数 OS_FlagBlock()来完成此事。这个函数完成挂起一个等待事件标志组的任务的细节处理。

等待任意一个指定的事件标志组中的事件标志位置 1：

L9.5(13)　当 wait_type 为 OS_FLAG_WAIT_SET_ANY 或 OS_FLAG_WAIT_SET_OR 时，OSFlagPend()函数取出事件标志组中由 flags 参数指定的事件标志位。

L9.5(14)　如果指定的事件标志位中任意一个已经置位，则等待操作立刻结束，返回调用函数。

L9.5(15)

L9.5(16)　在返回前，需要根据前面的布尔变量 consume 决定，是否需要对指定的事件标志位进行清除操作。

L9.5(17)

L9.5(18)　获取事件标志组的新的状态值，并返回调用函数。

L9.5(19)　如果所有指定的事件标志位都没有置位，则调用 OSFlagPend()的任务将被挂起，直到任意一个指定的事件标志位置位或者等待的超时时间到。

前面提到，如果期望的事件标志位没有置位，任务将被挂起，直到事件标志位置位或者等待超时。OS_FlagBlock()函数（其代码如程序清单 L9.6 所列）完成这个处理，并将调用 OSFlagPend()的任务添加到事件标志组的等待队列中。这个函数的处理过程如图 F9.4 所示。

程序清单 L9.6　添加一个任务到事件标志组等待任务链表中

```
static void OS_FlagBlock (OS_FLAG_GRP      *pgrp,
                          OS_FLAG_NODE     *pnode,
                          OS_FLAGS          flags,
                          INT8U             wait_type,
                          INT16U            timeout)
{
    OS_FLAG_NODE   *pnode_next;

    OSTCBCur->OSTCBStat       |= OS_STAT_FLAG;                    (1)
    OSTCBCur->OSTCBDly         = timeout;
#if OS_TASK_DEL_EN > 0
    OSTCBCur->OSTCBFlagNode    = pnode;                           (2)
```

```
#endif
    pnode->OSFlagNodeFlags        = flags;                                  (3)
    pnode->OSFlagNodeWaitType     = wait_type;
    pnode->OSFlagNodeTCB          = (void *)OSTCBCur;                       (4)
    pnode->OSFlagNodeNext         = pgrp->OSFlagWaitList;                   (5)
    pnode->OSFlagNodePrev         = (void *)0;                              (6)
    pnode->OSFlagNodeFlagGrp      = (void *)pgrp;                           (7)
    pnode_next                    = pgrp->OSFlagWaitList;
    if (pnode_next != (void *)0) {
        pnode_next->OSFlagNodePrev = pnode;                                 (8)
    }
    pgrp->OSFlagWaitList = (void *)pnode;                                   (9)
                                                                           (10)
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
}
```

L9.6(1)

F9.4(1) OS_FlagBlock()首先设置任务控制块的相应变量。注意到OS_FLAG_NODE数据结构是存放在调用OSFlagPend()的任务堆栈中的(见OSFlagPend()函数代码,程序清单L9.5)。这就不需要维持一个全局的空闲OS_FLAG_NODE链表,因为可以很方便地在任务堆栈中得到这样大的数据结构空间。这同时也意味着,任务堆栈必须有足够的空间,以保证可以存放OS_FLAG_NODE数据结构。

L9.5(2)

F9.4(2) 将OS_FLAG_NODE数据结构保存到任务控制块的一个链接中,前题是常量OS_TASK_DEL_EN必须被置为1。这样在调用OSTaskDel()时,可以根据这个链接把要删除的任务从对应的事件标志组的等待任务列表中删除。当然,这个删除操作是由其他任务完成的。

L9.6(3)

F9.4(3) OS_FlagBlock()保存任务等待事件标志组的指定事件标志位和等待方式的信息。

L9.6(4)

F9.4(4) 把任务控制块指针保存到OS_FLAG_NODE的任务控制块链接中。

L9.6(5)

F9.4(5) 事件标志组对应的所有OS_FLAG_NODE都链接在一起,保存在事件标志组的等待任务链表中。

L9.6(6)

F9.4(6) 新增加的OS_FLAG_NODE被添加到双向链表的开始端,这样处理起来比较

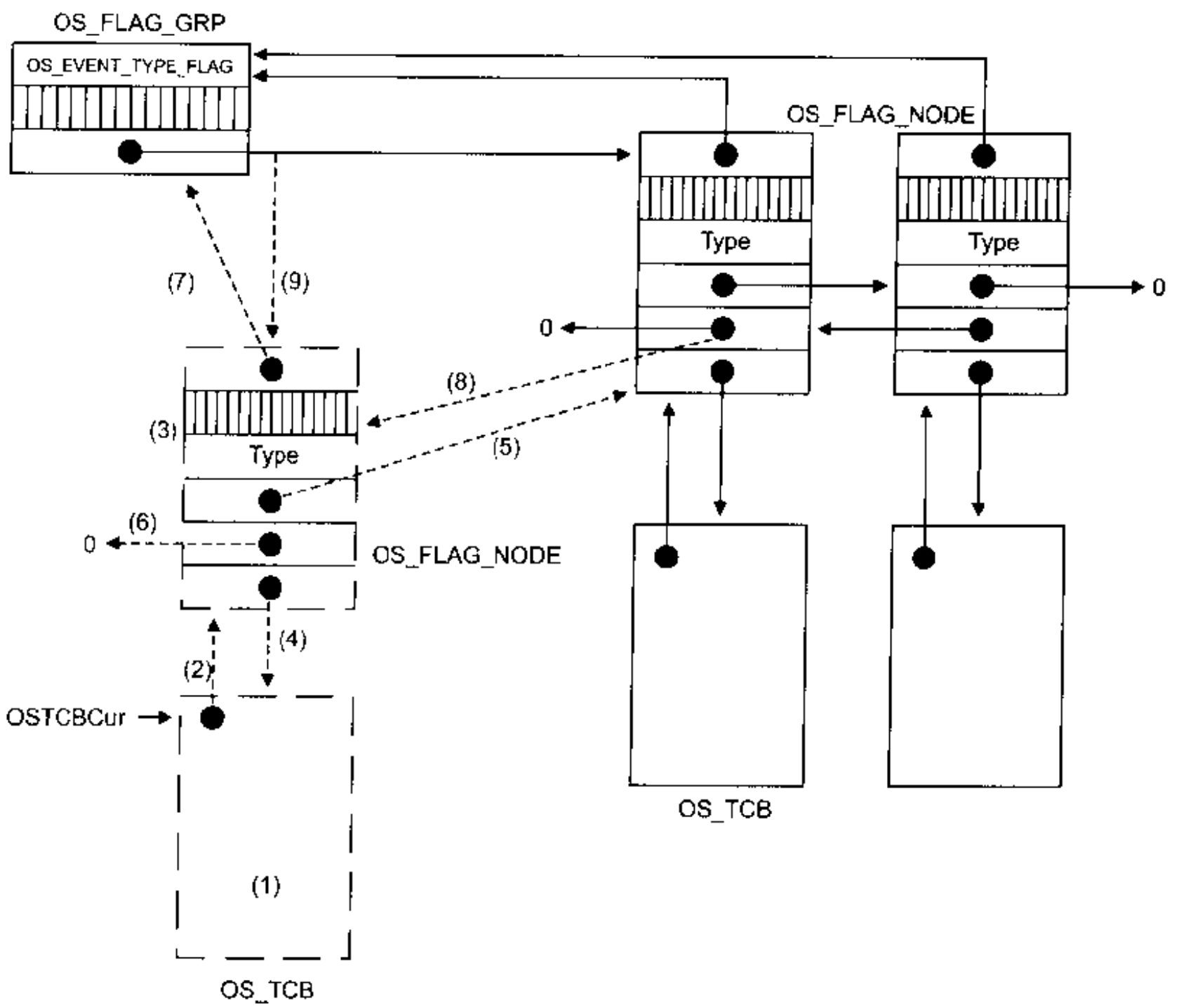


图 F9.4　添加当前任务到事件标志组等待任务链表中

简单容易

L9.5(7)

F9.4(7)　事件标志组的指针被反向链接到 OS_FLAG_NODE 的事件标志组指针中。当删除一个任务时，需要根据这个链接把被删除的任务从对应的事件标志组的等待任务列表中删除。

L9.6(8)

F9.4(8)　OS_FlagBlock()函数把前面一个 OS_FLAG_NODE 指针链接到新添加的 OS_FLAG_NODE。

L9.6(9)

F9.4(9)

L9.6(10)　等待任务列表的起始指针被更新为新添加的 OS_FLAG_NODE，调用任务也不再处于就绪态。

注意：在进行挂起任务处理时，中断关掉的。

L9.5(20)　OS_FlagBlock()返回时，会进行任务调度。这是因为调用该函数的任务由于

等待的事件标志没有发生而不能继续运行。

L9.5(21) 当这个任务恢复运行时，首先检查是因为什么原因而恢复运行的。如果任务控制块的状态表明，该任务还在等待事件标志组的事件标志，那么这个任务一定是因为等待事件标志超时而恢复运行的。

L9.5(22) 在这种情况下，调用OS_FlagUnlink()函数，把这个OS_FLAG_NODE从事件标志组的等待任务链表中删除，并且返回一个出错代码，说明发生了等待超时。OS_FlagUnlink()函数的代码并没有列出来，这段代码只是简单地将一个OS_FLAG_NODE从一个双向链表中删除。书后所附的光盘中的代码有注释，可以帮助理解。

L9.5(23)

L9.5(24) 如果任务恢复运行不是因为等待超时，那么一定是任务等待的事件标志按照预期的方式产生了。此时，根据调用OSFlagPend()时传入的是否须清除事件标志的参数，对事件标志进行相应的置位或清0操作。

L9.5(25) OSFlagPend()获取事件标志组的当前事件标志状态，并返回给调用任务。

9.04 置位或清0事件标志组中的事件标志，OSFlagPost()

置位或清0事件标志组中的事件标志通过调用OSFlagPost()函数来完成。这个函数的代码列在程序清单L9.7中。

程序清单L9.7 置位或清0事件标志组中的事件标志

```
OS_FLAGS  OSFlagPost (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR        cpu_sr;
#endif
    OS_FLAG_NODE    *pnode;
    BOOLEAN          sched;
    OS_FLAGS         flags_cur;
    OS_FLAGS         flags_rdy;


#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                                          (1)
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {                            (2)
```

```
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    OS_ENTER_CRITICAL();
    switch (opt) {                                                          (3)
        case OS_FLAG_CLR:
            pgrp->OSFlagFlags &= ~flags;                                    (4)
            break;

        case OS_FLAG_SET:
            pgrp->OSFlagFlags |= flags;                                     (5)
            break;

        default:
            OS_EXIT_CRITICAL();
            *err = OS_FLAG_INVALID_OPT;
            return ((OS_FLAGS)0);
    }
    sched = FALSE;                                                          (6)
    pnode = pgrp->OSFlagWaitList;
    while (pnode != (OS_FLAG_NODE *)0) {                                    (7)
        switch (pnode->OSFlagNodeWaitType) {
            case OS_FLAG_WAIT_SET_ALL:                                      (8)
                flags_rdy = pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                if (flags_rdy == pnode->OSFlagNodeFlags) {                  (9)
                    if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {        (10)
                        sched = TRUE;                                      (11)
                    }
                }
                break;

            case OS_FLAG_WAIT_SET_ANY:
                flags_rdy = pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                if (flags_rdy != (OS_FLAGS)0)
                    if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
                        sched = TRUE;
                    }
                }
                break;

#if OS_FLAG_WAIT_CLR_EN > 0
```

```
            case OS_FLAG_WAIT_CLR_ALL:
                flags_rdy = ~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                if (flags_rdy == pnode->OSFlagNodeFlags)
                    if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
                        sched = TRUE;
                    }
                }
                break;

            case OS_FLAG_WAIT_CLR_ANY:
                flags_rdy = ~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                if (flags_rdy != (OS_FLAGS)0) {
                    if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
                        sched = TRUE;
                    }
                }
                break;
#endif
        }
        pnode = pnode->OSFlagNodeNext;                                  (12)
    }
    OS_EXIT_CRITICAL();
    if (sched == TRUE) {                                                (13)
        OS_Sched();                                                     (14)
    }
    OS_ENTER_CRITICAL();
    flags_cur = pgrp->OSFlagFlags;                                      (15)
    OS_EXIT_CRITICAL();
    *err      = OS_NO_ERR;
    return (flags_cur);                                                 (16)
}
```

L9.7(1)

L9.7(2) 假定 OS_ARG_CHK_EN 常量被定义为 1，即允许检查函数参数。函数 OSFlagPost()将检查 pgrp 是否是一个值为 NULL 的空指针，并且检查 pgrp 是否指向一个由 OSFlagCreate()建立的事件标志组数据类型的结构。

L9.7(3)

L9.7(4)

L9.7(5) 根据调用 OSFlagPost()函数时传入的参数 opt，OSFlagPost()将置位(opt 为 OS_FLAG_SET)或清 0(opt 为 OS_FLAG_CLR)，由另一个参数 flag 指定事件标志。如果 opt 参数不是这 2 个值之一，OSFlagPost()将立刻结束并返回一个出错代码。

L9.7(6)　首先假定对事件标志的操作不会导致一个更高优先级的任务进入就绪态，将一个局部布尔变量 sched 置为 FALSE。如果后来发现了一个更高优先级的任务由于对事件标志的操作而进入了就绪态，则再把这个变量改为 TRUE。

L9.7(7)　如果有任务等待这个事件标志组，就遍历事件标志组的等待任务链表进行处理。

L9.7(15)

L9.7(16)　如果等待任务链表是空的，那么 OSFlagPost()函数将获取当前事件标志组的事件标志状态，并返回调用函数。

L9.7(8)　若等待任务链表非空，那么 OSFlagPost()函数将遍历所有 OS_FLAG_NODE，以检查新设定的事件标志是否满足某个任务所期待的条件。每个任务都可能等待下面 4 种情形中的一种：

- 所有指定的事件标志位都置位；
- 任意一个指定的事件标志位置位；
- 所有指定的事件标志位都清 0；
- 任意一个指定的事件标志位清 0。

L9.7(9)

L9.7(10)　后面 2 种情形可以通过设置常量 OS_FLAG_WAIT_CLR_EN 为 0，使得在编译时忽略相关的代码。如果不需要使用等待事件标志位清除的功能，或者希望减少最终产品的程序代码，则可以选择这 2 种做法。如果一个任务等待的事件标志位条件得到满足，那么这个任务将被标志为进入就绪态。该处理是通过调用函数 OS_FlagTaskRdy()来完成的(参见程序清单 L9.8)。由于 4 种情形的处理都非常相似，这里只详细讨论第 1 种情形的处理。

L9.7(11)　这里任务等待的事件标志已经满足，任务进入就绪态，所以需要立即进行任务调度。但是，并不是每检查一个 OS_FLAG_NODE，就进行一次任务调度，而是在遍历完全部等待任务后，进行一次总的调度，所以这里将是否需要进行调度的信息保存在一个局部布尔变量 sched 中。

L9.7(12)　通过双向链表得到下一个 OS_FLAG_NODE 的指针。

当遍历等待任务链表时，中断是关闭的。这意味着 OSFlagPost()函数有可能导致中断的长时间关闭，特别是在 OSFlagPost()函数可能引起多个任务进入就绪态时。在这种情况下，该函数的执行时间仍然是有限的，而且是可以预先确定的。

L9.7(13)

L9.7(14)　遍历完等待任务链表后，检查 sched 变量，判断是否需要进行任务调度。这可能导致一个刚刚接收到预期的事件标志而进入就绪态的更高优先级的任务开始运行。

L9.7(15)

L9.7(16)　OSFlagPost()返回当前事件标志组的事件标志状态。

前面提到，程序清单 L9.8 是使一个等待事件标志的任务进入就绪态的程序代码。

程序清单 L9.8 使等待事件标志的任务进入就绪态

```
static  BOOLEAN  OS_FlagTaskRdy (OS_FLAG_NODE *pnode, OS_FLAGS flags_rdy)
{
    OS_TCB   *ptcb;
    BOOLEAN   sched;


    ptcb                = (OS_TCB *)pnode->OSFlagNodeTCB;  /* Point to TCB of waiting task*/
    ptcb->OSTCBDly      = 0;
    ptcb->OSTCBFlagsRdy = flags_rdy;
    ptcb->OSTCBStat    &= ~OS_STAT_FLAG;
    if (ptcb->OSTCBStat == OS_STAT_RDY) {                  /* Put task into ready list*/
        OSRdyGrp               |= ptcb->OSTCBBitY;
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        sched                  = TRUE;
    } else {
        sched                  = FALSE;
    }
    OS_FlagUnlink(pnode);
    return (sched);
}
```

L9.8(4) 该处理在 μC/OS-II 中是一个标准的过程。这里惟一的不同在于，事件标志组中当一个任务等待的事件标志发生后，为该任务建立的 OS_FLAG_NODE 数据结构就没有用处了；所以这里把这个任务的 OS_FLAG_NODE 数据结构从等待任务链表中删除掉，同时还会把这个 OS_FLAG_NODE 数据结构指针从该任务的事件控制块中删除掉（参见 6.05 节："使一个任务进入就绪态，OS_EventTaskRdy()"）。

L9.8(1)

L9.8(2)

L9.8(3) 注意：即使任务等待的事件标志都发生了，任务已经从事件标志组的等待任务链表中被删除了，但是这个任务可能由于其他的原因而不能进入就绪态。

这里一个局部布尔变量 sched 返回给调用函数，用来通知调用函数，这个任务究竟是否进入了就绪态。

把一个 OS_FLAG_NODE 从任务等待链表中删除的工作由OS_FlagUnlink()来完成，这部分的代码如程序清单 L9.9 所列。图 F9.5 显示了一个即将被删除的 OS_FLAG_NODE 在任务等待链表中可能出现的 4 种位置情况。从双向链表中删除一个节点，需要注意调整很多的指针。

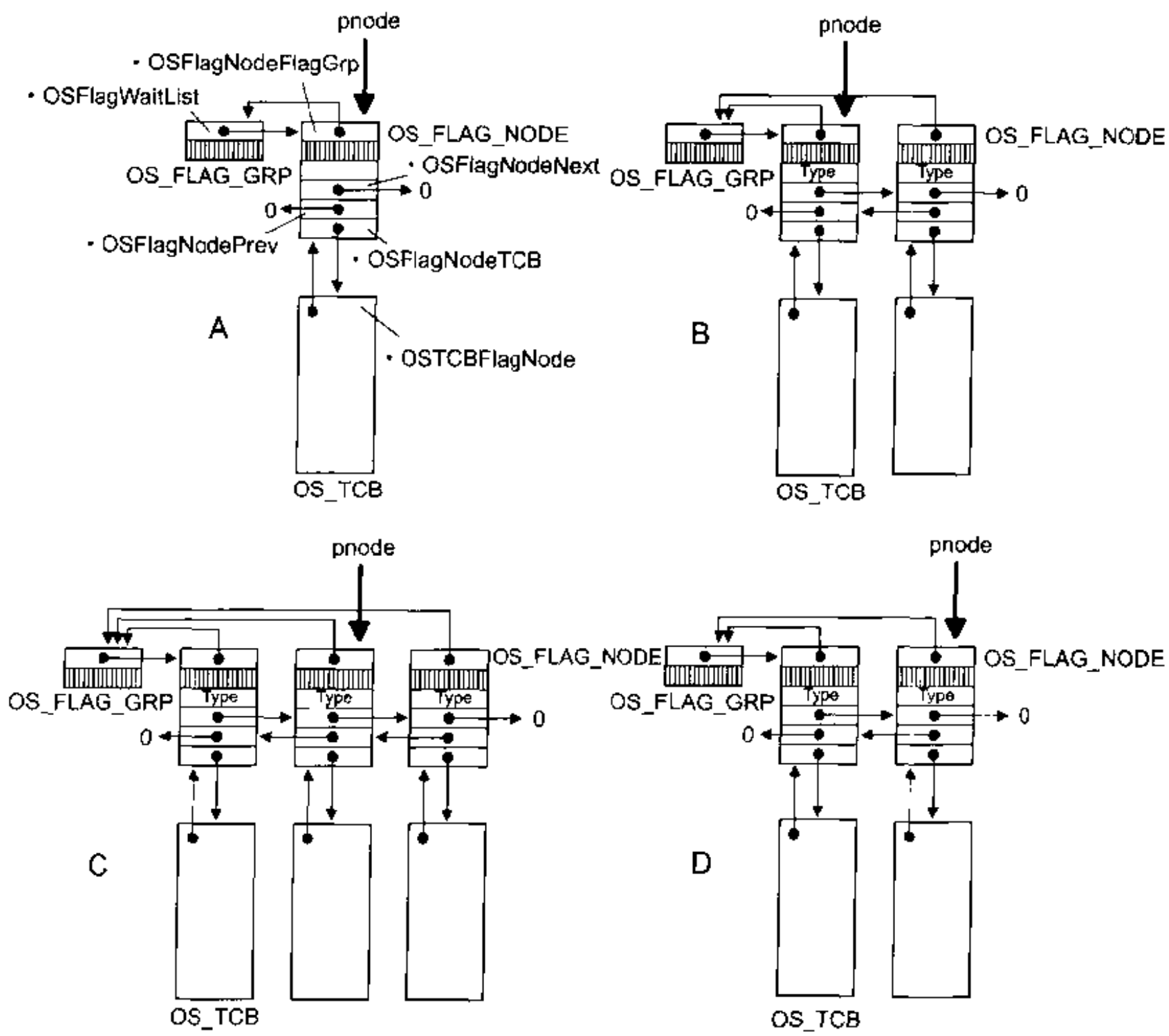


图 F9.5 从任务等待链表中删除 OS_FLAG_NODE

程序清单 L9.9 从任务等待链表中删除 OS_FLAG_NODE

```
void  OS_FlagUnlink (OS_FLAG_NODE *pnode)
{
#if OS_TASK_DEL_EN > 0
    OS_TCB        *ptcb;
#endif
    OS_FLAG_GRP   *pgrp;
    OS_FLAG_NODE  *pnode_prev;
    OS_FLAG_NODE  *pnode_next;


    pnode_prev = pnode->OSFlagNodePrev;                                    (1)
    pnode_next = pnode->OSFlagNodeNext;                                    (2)
```

```
    if (pnode_prev == (OS_FLAG_NODE *)0) {                                  (3)
        pgrp                    = pnode->OSFlagNodeFlagGrp;                 (4)
        pgrp->OSFlagWaitList    = (void *)pnode_next;                       (5)
        if (pnode_next != (OS_FLAG_NODE *)0) {                              (6)
            pnode_next->OSFlagNodePrev = (OS_FLAG_NODE *)0;                 (7)
        }
    } else {
        pnode_prev->OSFlagNodeNext = pnode_next;                            (8)
        if (pnode_next != (OS_FLAG_NODE *)0) {                              (9)
            pnode_next->OSFlagNodePrev = pnode_prev;                        (10)
        }
    }
#if OS_TASK_DEL_EN > 0
    ptcb                = (OS_TCB *)pnode->OSFlagNodeTCB;                   (11)
    ptcb->OSTCBFlagNode = (void *)0;                                        (12)
#endif
}
```

L9.9(1)

L9.9(2) OS_FlagUnlink()首先定义了2个局部指针:pnode_next和pnode_prev。这2个指针分别指向链表中的后一个和前一个OS_FLAG_NODE数据结构(这里把每一个这样的数据结构叫做链表的一个节点)。

L9.9(3)

F9.5(A,B) 检查指向前一个节点的指针是否为NULL,判断即将被删除的这个节点是否是图F9.6中的前2种情形。在这2种情形下,即将被删除的节点是整个链表的第1个节点。

L9.9(4)

L9.9(5) 如果即将被删除的节点是链表的第1个节点,那么在该节点删除后,链表的表头指针应该指向下一个节点。

L9.9(6)

L9.9(7)

F9.5(B) 如果被删除的节点确实是链表的第1个节点,那么紧接着的一个节点将成为链表的第1个节点。这个新的头节点的"前一个节点指针"将被更新为NULL。

L9.9(8)

F9.5(C,D) 如果被删除的节点不是链表中的第1个节点,那么这个节点的前一个节点的"后一个节点指针"将指向即将被删除的节点的后一个节点。

L9.9(9)

L9.9(10) 同样,这个将被删除的节点的后一个节点的"前一个节点指针"也将被更新为该即将被删除节点的前一个节点。

L9.9(11)

L9.9(12)　在所有的 4 种情况下，都会把任务控制块中的 OSTCBFlagNode 指针重新赋值为 NULL。因为任务得到了预期的事件标志，则 OSFlagPend()函数也将退出，建立的 OS_FLAG_NODE 数据结构也将不复存在。

图 F9.6～F9.9 显示了在上述 4 种情形下删除一个 OS_FLAG_NODE 数据结构前、后整个链表的情况。图中括号标注的数字可以对照程序清单 L9.9，可以看到 OS_FlagUnlink()最多时更新了 3 个指针。

因为 OS_FLAG_NODE 数据结构是在任务的堆栈中分配空间的(由 OSFlagPend()函数分配)，因此它会自动消失，其占用的空间会返还给堆栈，以便重新使用。对于等待事件标志的任务来说，它甚至根本不知道 OS_FLAG_NODE 的存在。

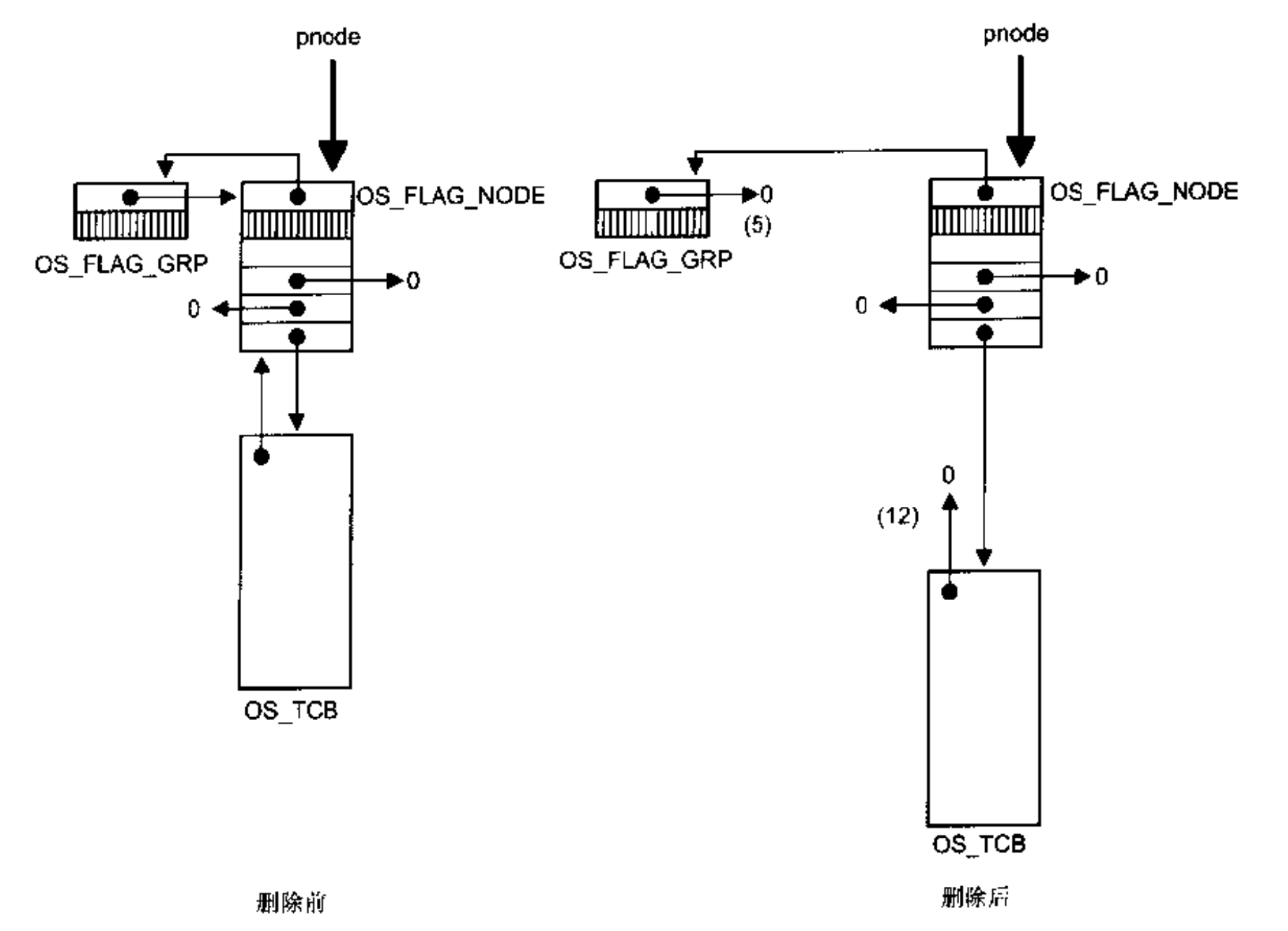


图 F9.6　从任务等待链表中删除 OS_FLAG_NODE，情形 A

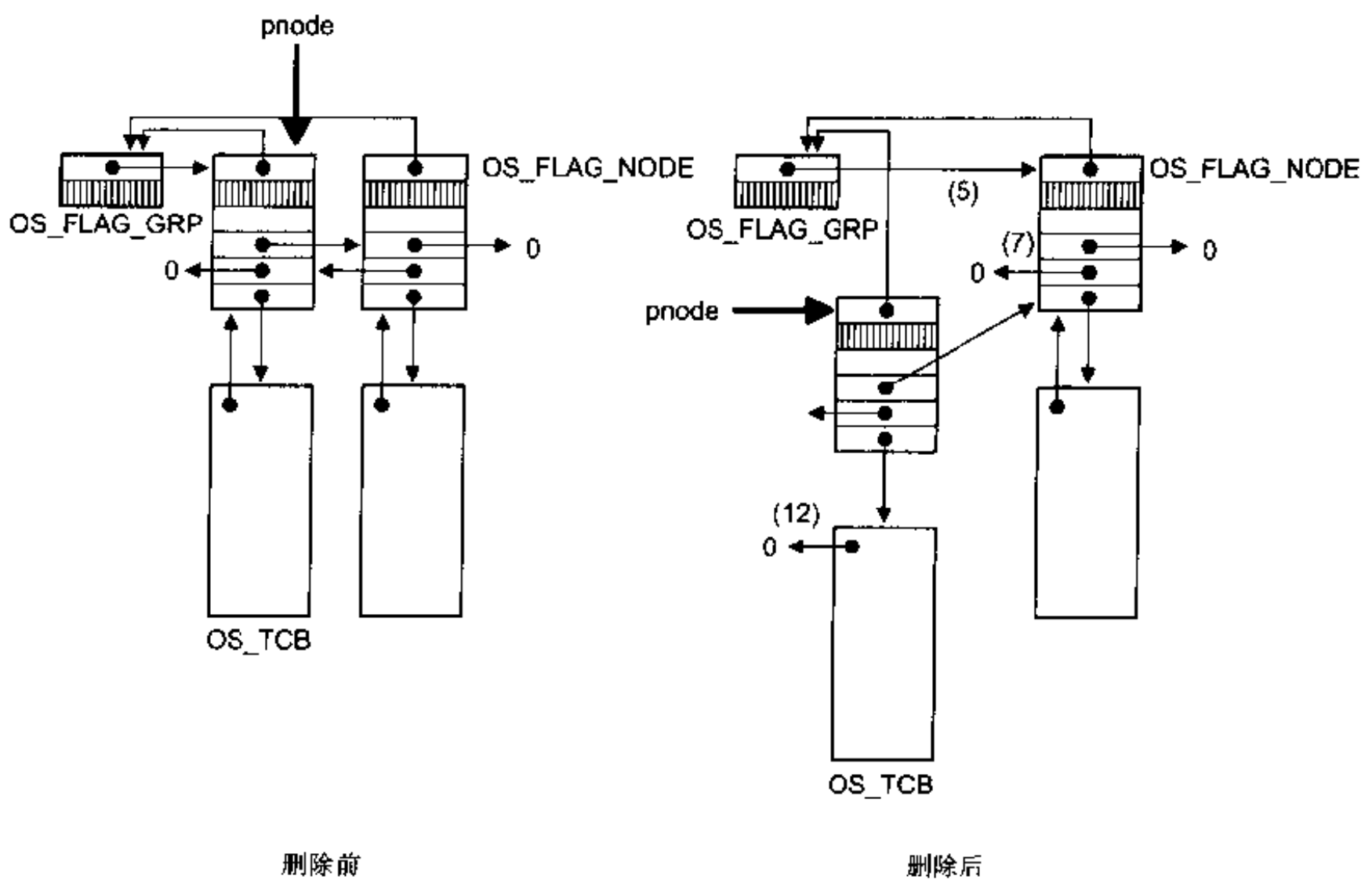


图 F9.7 从任务等待链表中删除 OS_FLAG_NODE，情形 B

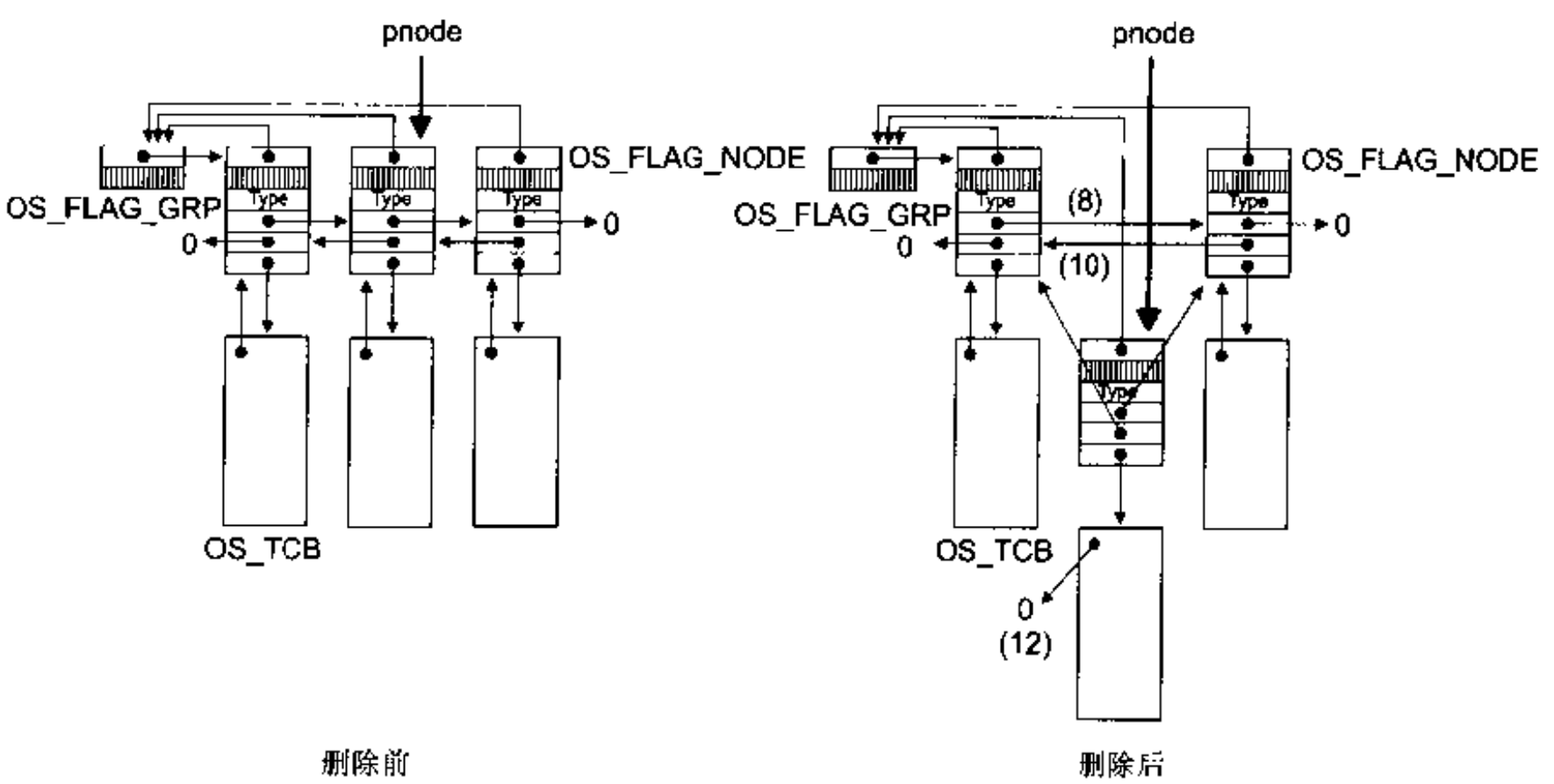


图 F9.8 从任务等待链表中删除 OS_FLAG_NODE，情形 C

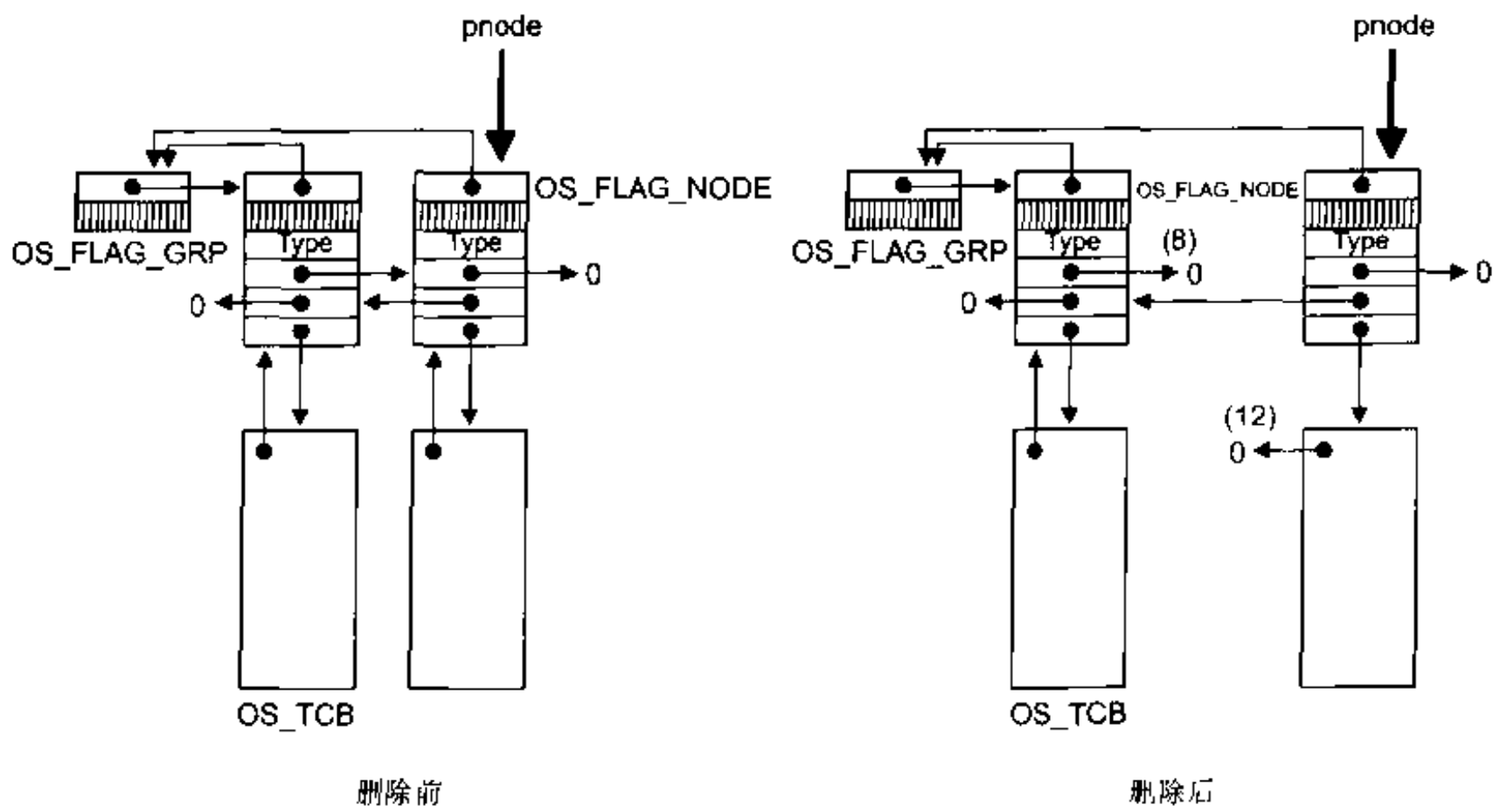


图 F9.9　从任务等待链表中删除 OS_FLAG_NODE，情形 D

9.05　无等待地获得事件标志组中的事件标志，OSFlagAccept()

无等待地获得事件标志组中的事件标志的函数代码列在程序清单 L9.10 中。这个函数同 OSFlagPend()非常相似，不同点仅在于，如果等待的事件标志没有发生，那么调用的任务并不挂起，以等待这些事件标志发生。这些不同点具体表现为以下 2 方面：

- OSFlagAccept() 可以在中断服务子程序中调用。
- 如果等待的事件标志没有发生，那么该函数并不停止任务的运行以等待事件标志，而是简单地返回一个出错代码。调用函数应判断 OSFlagAccept()的返回值，以了解是否得到了期望的事件标志。

程序清单 L9.10　无等待地获得事件标志组中的事件标志

```
OS_FLAGS  OSFlagAccept (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR     cpu_sr;
#endif
    OS_FLAGS      flags_cur;
    OS_FLAGS      flags_rdy;
    BOOLEAN       consume;
```

```
#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    if (wait_type & OS_FLAG_CONSUME) {
        wait_type &= ~OS_FLAG_CONSUME;
        consume    = TRUE;
    } else {
        consume    = FALSE;
    }

    OS_ENTER_CRITICAL();
    switch (wait_type) {
        case OS_FLAG_WAIT_SET_ALL:
            flags_rdy = pgrp->OSFlagFlags & flags;
            if (flags_rdy == flags) {
                if (consume == TRUE) {
                    pgrp->OSFlagFlags &= ~flags_rdy;
                }
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
            } else {
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_FLAG_ERR_NOT_RDY;
            }
            break;

        case OS_FLAG_WAIT_SET_ANY:
            flags_rdy = pgrp->OSFlagFlags & flags;
            if (flags_rdy != (OS_FLAGS)0) {
                if (consume == TRUE) {
                    pgrp->OSFlagFlags &= ~flags_rdy;
                }
                flags_cur = pgrp->OSFlagFlags;
```

```
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
            } else {
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_FLAG_ERR_NOT_RDY;
            }
            break;

# if OS_FLAG_WAIT_CLR_EN > 0
        case OS_FLAG_WAIT_CLR_ALL:
            flags_rdy = ~pgrp->OSFlagFlags & flags;
            if (flags_rdy == flags) {
                if (consume == TRUE) {
                    pgrp->OSFlagFlags |= flags_rdy;
                }
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
            } else {
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_FLAG_ERR_NOT_RDY;
            }
            break;

        case OS_FLAG_WAIT_CLR_ANY:
            flags_rdy = ~pgrp->OSFlagFlags & flags;
            if (flags_rdy != (OS_FLAGS)0) {
                if (consume == TRUE) {
                    pgrp->OSFlagFlags |= flags_rdy;
                }
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
            } else {
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_FLAG_ERR_NOT_RDY;
            }
            break;
# endif
```

```
        default:
            OS_EXIT_CRITICAL();
            flags_cur = (OS_FLAGS)0;
            *err      = OS_FLAG_ERR_WAIT_TYPE;
            break;
    }
    return (flags_cur);
}
```

9.06 查询事件标志组的状态，OSFlagQuery()

OSFlagQuery() 允许程序代码得到当前事件标志组的事件标志状态。这个函数的代码列在程序清单 L9.11 中。

程序清单 L9.11 查询事件标志组的状态

```
OS_FLAGS  OSFlagQuery (OS_FLAG_GRP *pgrp, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_FLAGS   flags;


#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                                    (1)
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {                      (2)
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    OS_ENTER_CRITICAL();
    flags = pgrp->OSFlagFlags;                                         (3)
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (flags);                                                    (4)
}
```

调用 OSFlagQuery()时需要 2 个参数：pgrp 为需要查询的事件标志组的指针，该指针应该通过 OSFlagCreate()函数建立；err 为一个指向返回错误信息字节的指针，通过该指针指向的地址的内容，可以了解 OSFlagQuery()函数的返回信息。

L9.11(1)

L9.11(2)　如果 OS_CFG.H 文件中将常量 OS_ARG_CHK_EN 置为 1，OSFlagQuery()将对函数的参数进行检查

L9.11(3)

L9.11(4)　如果函数运行过程中没有其他错误，则 OSFlagQuery() 获取当前事件标志组的事件标志的状态，并返回给调用函数。

第 10 章

消息邮箱管理

消息邮箱(简称邮箱)是 μC/OS-II 中的另一种通信机制,可以使一个任务或者中断服务子程序向另一个任务发送一个指针型的变量。通常该指针指向一个包含了"消息"的特定数据结构。

μC/OS-II 提供了 6 种对邮箱的操作:OSMboxCreate(),OSMboxPend(),OSMboxPost(),OSMboxPostOpt(),OSMboxAccept()及 OSMboxQuery()函数。

为了在 μC/OS-II 中使用邮箱,必须将 OS_CFG.H 文件中的相应配置常数置为 1。表 T10.1 列出了基于 OS_CFG.H 文件中配置常数的值而确定是否编译的函数。当 OS_MBOX_EN 被置为 0 时,所有的邮箱函数都被屏蔽。当要使用表 T10.1 中的相应函数时,只需将其后列出的配置常数置为 1。因为 μC/OS-II 支持消息邮箱管理必须有函数(OSMboxCreate()和 OSMboxPend(),所以它们不能单独被屏蔽掉,并且必须允许函数 OSMboxPost()和 OSMboxPostOpt()中的一个。

表 T10.1　OS_CFG.H 中的邮箱配置常数

μC/OS-II 邮箱函数	OS_CFG.H 中置 1 允许相应函数
OSMboxAccept()	OS_MBOX_ACCEPT_EN
OSMboxCreate()	
OSMboxDel()	OS_MBOX_DEL_EN
OSMboxPend()	
OSMboxPost()	OS_MBOX_POST_EN
OSMboxPostOpt()	OS_MBOX_POST_OPT_EN
OSMboxQuery()	OS_MBOX_QUERY_EN

图 F10.1 描述了任务、中断服务子程序和邮箱之间的关系。图中用符号"I"表示邮箱,沙漏表示 OSMboxPend()所定义的超时时限。邮箱包含的内容是一个指向一条消息的指针,指针所指向的内容可以给定。一个邮箱只能包含一个这样的指针(邮箱为满时),或者一个指向 NULL 的指针(邮箱为空时)。

从图 F10.1 可以看出,任务或者中断服务子程序都可以调用函数 OSMboxPost()或者 OSMboxPostOpt(),而只有任务可以调用 OSMboxDel(),OSMboxPend()及 OSMboxQuery()函数。

应用程序可以使用任意多个邮箱,其最大数目由 OS_CFG.H 文件中的配置常数 OS_MAX_EVENTS 设定。

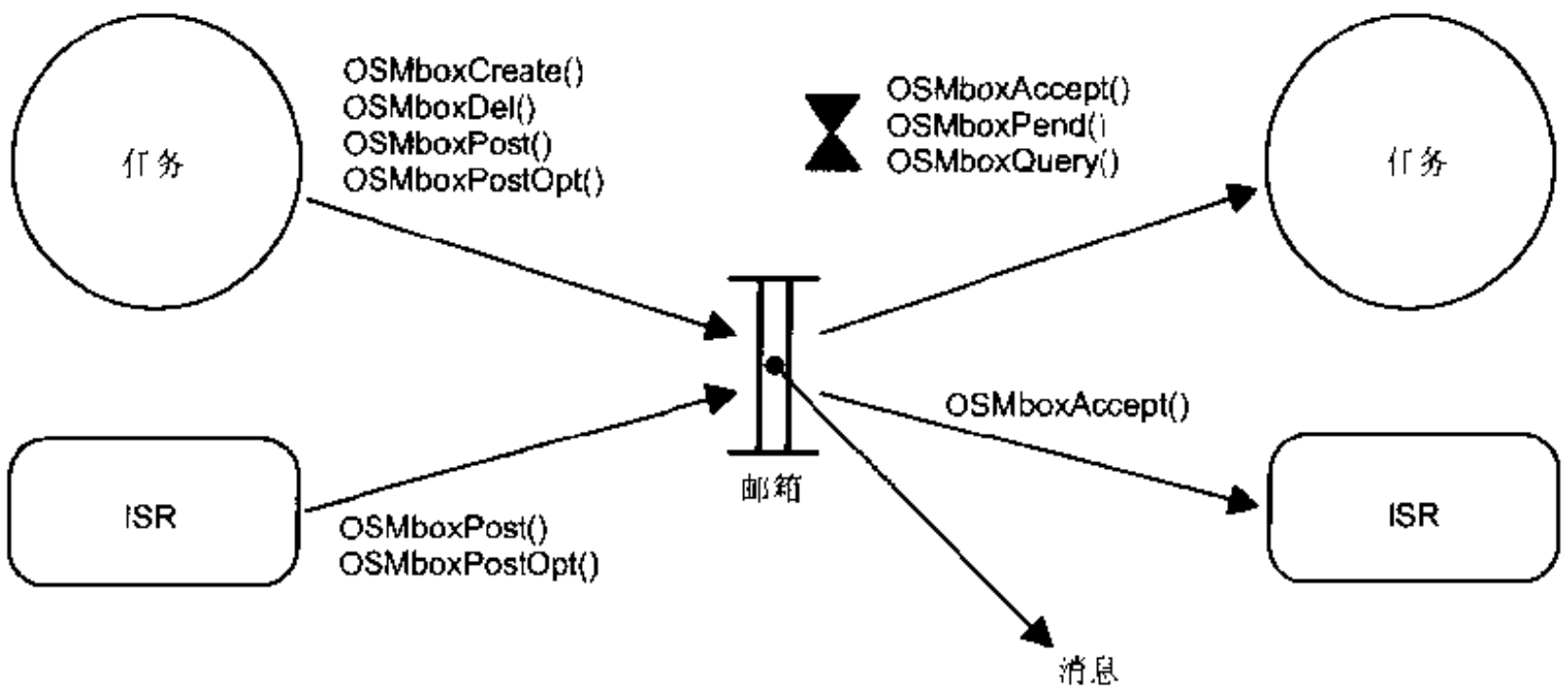


图 F10.1 任务、中断服务子程序和邮箱之间的关系

10.00 建立一个邮箱,OSMboxCreate()

使用邮箱之前,必须先建立邮箱。该操作可以通过调用OSMboxCreate()函数来完成,并且须定义指针的初始值。一般情况下,这个初始值是NULL;但也可以初始化一个邮箱,使其在最开始就包含一条消息。如果使用邮箱的目的是通知一个事件的发生(发送一条消息),那么就要初始化该邮箱为空,即NULL,因为在开始时(很有可能)事件还没有发生。如果用邮箱共享某些资源,那么就要初始化该邮箱为一个非NULL的指针。在这种情况下,邮箱被当成一个二值信号量使用。

程序清单L10.1是建立一个邮箱的源代码。

程序清单L10.1 建立一个邮箱

```
OS_EVENT    *OSMboxCreate (void *msg)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;                                                    (1)
# endif
    OS_EVENT    *pevent;


    if (OSIntNesting > 0) {                                                 (2)
        return ((OS_EVENT *)0);
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                               (3)
    if (OSEventFreeList != (OS_EVENT *)0) {                                 (4)
```

```
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;          (5)
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                                          (6)
        pevent->OSEventType = OS_EVENT_TYPE_MBOX;                           (7)
        pevent->OSEventCnt  = 0;                                            (8)
        pevent->OSEventPtr  = msg;                                          (9)
        OS_EventWaitListInit(pevent);                                       (10)
    }
    return (pevent);                                                        (11)
}
```

L10.1(1) 如果关中断的方法 OS_CRITICAL_METHOD 等于 3,就要在该函数中多定义一个局部变量 cpu_sr。

L10.1(2) 中断服务子程序不能调用 OSMboxCreate()函数,所以 OSMboxCreate()函数首先确定不是被中断服务子程序调用。所有建立工作必须在任务级代码中或者多任务启动前完成。

L10.1(3) OSMboxCreate()函数试图从空余事件控制块链表中获得一个事件控制块(见图 F6.5)。

L10.1(4)

L10.1(5) 对空余事件控制链表的指针进行适当的调整,使它指向下一个空余的事件控制块。

L10.1(6)

L10.1(7) 如果这时有事件控制块 ECB 可用,就将该事件控制块 ECB 的事件类型设置成邮箱型 OS_EVENT_TYPE_MBOX。其他的邮箱类函数 OSMbox???()通过检查该域,保证所操作的事件控制块类型的正确性(类型为邮箱)。这样可以防止调用 OSMboxPost()函数对一个用做消息队列的事件控制块 ECB 进行操作。

L10.1(8) 初始化 .OSEventCnt 域为 0,因为邮箱不再使用该域。

L10.1(9) 将消息的初始值存入事件控制块 ECB 中。

L10.1(10) 调用 OS_EventWaitListInit()函数,对事件控制块的等待任务列表进行初始化(见 6.04 节:"初始化一个事件控制块,OS_EventWaitListInit()")。因为邮箱正在被初始化,这时没有任何任务等待该邮箱,故 OS_EventWaitListInit()函数将 .OSEventGrp 和 .OSEventTbl[]域清 0。

L10.1(11) OSMboxCreate()函数返回给调用函数一个指向事件控制块 ECB 的指针。以后,OSMboxAccept(),OSMboxDel(),OSMboxPend(),OSMboxPost(),OSMboxPostOpt()及 OSMboxQuery()函数对邮箱的所有操作都是通过该指针完成的;因此,这个指针实际上就是该邮箱的句柄。如果系统中没有可用的事件控制块,OSMboxCreate()函数将返回一个 NULL 指针。应养成良好的习惯,检查和确认函数返回值确实是想得到的值。而传递 NULL 指针

并不意味着调用失败，因为，当 OS_ARG_CHK_EN 置为 1 时，μC/OS－II 会检查参数的合理性。OSMboxCreate() 函数返回之前，事件控制块的内容如图 F10.2 所示。

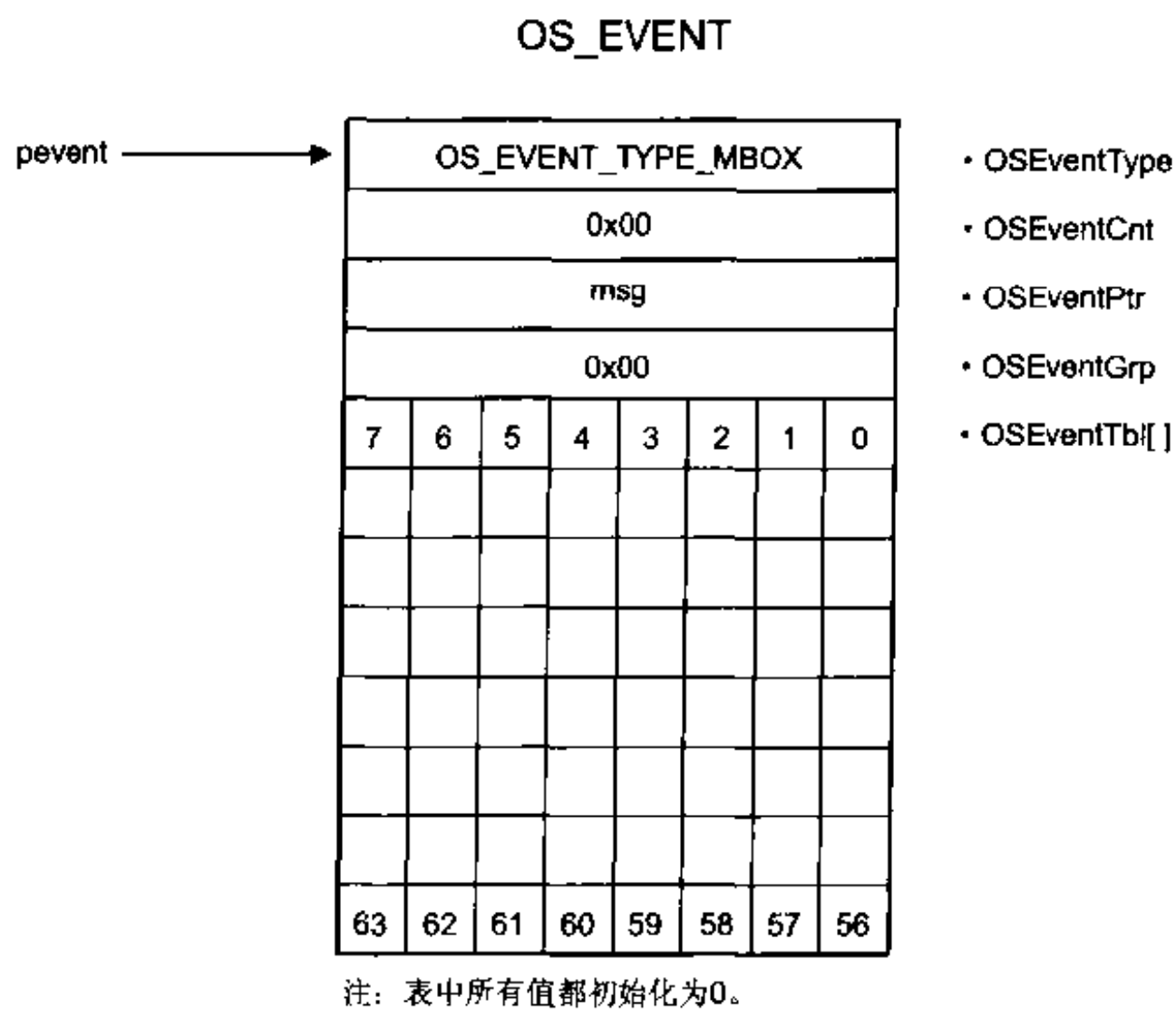


图 F10.2　OSMboxCreate()函数返回之前的事件控制块数据结构

10.01　删除一个邮箱，OSMboxDel()

程序清单 L10.2 是 OSMboxDel()函数的源代码。当将 OS_CFG.H 文件中的 OS_MBOX_DEL_EN设为 1 时，该段代码才被编译。使用此函数时需特别注意，多个任务可能试图操作已经删除的邮箱。总之，在删除邮箱之前，必须首先删除可能操作该邮箱的所有任务。

程序清单 L10.2　删除一个邮箱

```
OS_EVENT    *OSMboxDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    tasks_waiting;
```

```
    if (OSIntNesting > 0) {                                                    (1)
        *err = OS_ERR_DEL_ISR;
        return (pevent);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                             (2)
        *err  = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                           (3)
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                          (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:
            if (tasks_waiting == FALSE) {
                pevent->OSEventType = OS_EVENT_TYPE_UNUSED;                    (5)
                pevent->OSEventPtr  = OSEventFreeList;                         (6)
                OSEventFreeList     = pevent;                                  (7)
                OS_EXIT_CRITICAL();
                *err = OS_NO_ERR;
                return ((OS_EVENT *)0);                                        (8)
            } else {
                OS_EXIT_CRITICAL();
                *err = OS_ERR_TASK_WAITING;
                return (pevent);
            }

        case OS_DEL_ALWAYS:
            while (pevent->OSEventGrp != 0x00) {                               (9)
                OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MBOX);              (10)
            }
            pevent->OSEventType = OS_EVENT_TYPE_UNUSED;                        (11)
            pevent->OSEventPtr  = OSEventFreeList;                             (12)
```

```
            OSEventFreeList        = pevent;
            OS_EXIT_CRITICAL();
            if (tasks_waiting == TRUE) {
                OS_Sched();                                                    (13)
            }
            *err = OS_NO_ERR;
            return ((OS_EVENT *)0);                                            (14)

        default:
            OS_EXIT_CRITICAL();
            *err = OS_ERR_INVALID_OPT;
            return (pevent);
    }
}
```

L10.2(1)　中断服务子程序不能调用 OSMboxDel()函数，所以 OSMboxDel()函数首先确定不是被中断服务子程序调用。

L10.2(2)

L10.2(3)　OSMboxDel()函数验证 pevent，确定它为非 NULL 指针，并且指向某一用于邮箱的事件控制块。

L10.2(4)　OSMboxDel()函数确定是否有任务在等待该邮箱，并根据判断结果对标志变量 tasks_waiting 进位设置。

根据 opt 参数的值，OSMboxDel()函数对邮箱进行相应的删除操作。当没有任务等待邮箱时，OSMboxDel()函数删除该邮箱（opt == OS_DEL_NO_PEND）；或者，尽管有任务在等待，OSMboxDel()函数亦直接删除该邮箱（opt==(OS_DEL_ALWAYS)。

L10.2(5)

L10.2(6)

L10.2(7)　当 opt 设置为 OS_DEL_NO_PEND，并且没有等待该邮箱的任务时，OSMboxDel()函数将事件控制块标志为未使用，并把它返还给空余事件控制块链表。此操作允许创建另一个邮箱（或者其他使用事件控制块 ECB 的通信机制）。

L10.2(8)　OSMboxDel()函数返回 - NULL 指针，因为，此后该邮箱就不能再通过原先的指针访问了。可以通过以下方式调用 OSMboxDel()函数。

```
MbxPtr = (OSMboxDel(MbxPtr, opt, &err);
```

这种方式允许调用函数修改指向邮箱的指针。当指定 opt 为 OS_DEL_NO_PEND 时，若有任务在等待该邮箱，则不能删除该邮箱，且此时 OSMboxDel()函数将返回“出错”代码（OS_ERR_TASK_WAITING）。

L10.2(9)

L10.2(10) 当 opt 设定为 OS_DEL_ALWAYS 时，所有等待该信号的任务都进入就绪态，每个任务都以为自己得到了 NULL 信息。每个任务必须检查返回的指针，看其是否是非 NULL。另外，当每一个任务都进入了就绪态时，中断是关掉的。当然，这也延长了系统的中断响应时间。

L10.2(11)

L10.2(12) 当所有的等待任务就绪后，OSMboxDel()函数将事件控制块标志为未使用，并把它返还给空余事件控制块链表。

L10.2(13) 当有任务在等待该邮箱时，任务调度函数才被调用。

L10.2(14) 请再次注意，OSMboxDel()函数返回一个 NULL 指针，因为，此后该邮箱就不能再通过原先的指针访问了。

10.02 等待邮箱中的消息，OSMboxPend()

程序清单 L10.3 是 OSMboxPend()函数的源代码。

程序清单 L10.3 等待邮箱中的消息，OSMboxPend()

```
void   *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    void         *msg;


    if (OSIntNesting > 0) {                                                    (1)
        *err = OS_ERR_PEND_ISR;
        return ((void *)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                             (2)
        *err  = OS_ERR_PEVENT_NULL;
        return ((void *)0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                           (3)
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
#endif
    OS_ENTER_CRITICAL();
```

```
    msg = pevent->OSEventPtr;                                          (4)
    if (msg != (void *)0) {
        pevent->OSEventPtr = (void *)0;                                (5)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (msg);                                                  (6)
    }
    OSTCBCur->OSTCBStat |= OS_STAT_MBOX;                               (7)
    OSTCBCur->OSTCBDly  = timeout;                                     (8)
    OS_EventTaskWait(pevent);                                          (9)
    OS_EXIT_CRITICAL();
    OS_Sched();                                                        (10)
    OS_ENTER_CRITICAL();
    msg = OSTCBCur->OSTCBMsg;
    if (msg != (void *)0) {                                            (11)
        OSTCBCur->OSTCBMsg      = (void *)0;
        OSTCBCur->OSTCBStat     = OS_STAT_RDY;
        OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
        OS_EXIT_CRITICAL();
        *err                    = OS_NO_ERR;
        return (msg);                                                  (12)
    }
    OS_EventTO(pevent);                                                (13)
    OS_EXIT_CRITICAL();
    *err = OS_TIMEOUT;
    return ((void *)0);                                                (14)
}
```

L10.3(1)　OSMboxPend()函数检查是否是中断服务子程序调用了该函数。中断服务子程序不能等待，所以由中断服务子程序调用 OSMboxPend()函数就没有意义。中断服务子程序中可以调用 OSMboxAccept()函数(见 10.05 节："无等待地从邮箱中得到一则消息，OSMboxAccept()")。

L10.3(2)

L10.3(3)　如果 OS_CFG.H 文件中的 OS_ARG_CHK_EN 设置为 1，则 OSMboxPend()函数检查 pevent，以确定其为非 NULL 指针，并且指向由 OSMboxCreate()函数建立的事件控制块。

L10.3(4)

L10.3(5)

L10.3(6)　如果邮箱中有消息(非 NULL 指针)，那么从邮箱中取出该消息，返回给调用函数，并将 NULL 指针存入邮箱中。同时，OSMboxPend() 函数将"无错"代码(OS_NO_ERR)返回给它的调用函数。这个返回结果表示，已由另一个任

务或者中断服务子程序将信息发送到邮箱中。这也是运行 OSMboxPend()函数最快的情况。

如果邮箱为空，则调用 OSMboxPend()函数的任务要进入睡眠状态，等待另一个任务(或者中断服务子程序)通过邮箱发送消息(见 10.04 节："向邮箱发送一则消息，OSMboxPost()")。OSMboxPend()允许定义一个最长等待时间作为它的参数(超时时限)，这样可以避免该任务无限期地等待邮箱的消息。

如果该参数值是一个大于 0 的值，那么 OSMboxPend()函数挂起该任务，该任务将一直等到邮箱获得消息或者等待超时。如果该参数值为 0，则该任务将无限期地等待下去。

L10.3(7) OSMboxPend()函数通过将任务控制块中的状态标志 .OSTCBStat 置 1，把任务置于睡眠态，等待邮箱消息将其唤醒。

L10.3(8) 等待时间也同时置入任务控制块中，该值在 OSTimeTick()函数中被逐次减 1。

注意：OSTimeTick() 函数对每个任务的任务控制块的 .OSTCBDly 域做减 1 操作(只要该域不为 0)(见 3.11 节："时钟节拍")。

L10.3(9) 真正将任务置于睡眠状态的操作在 OS_EventTaskWait()函数中实现(见 6.06节："使一个任务进入等待某事件发生状态，OS_EventTaskWait()")。

L10.3(10) 因为当前任务已经不再是就绪态了，所以任务调度函数将下一个最高优先级的任务调入，准备运行。就当前任务来说，在消息到来之前，调用 OSMboxPend()函数的任务并不知道自己被挂起了。直到邮箱收到了消息(或者等待超时)，OSMboxPend()函数接着从 OS_Sched()函数的后面继续运行。

L10.3(11) OS_Sched()函数返回后，OSMboxPend()函数要检查 OSMboxPost()函数是否将消息存入任务控制块中。

L10.3(12) 如果是，那么该次函数调用成功，对应的消息返回给调用函数。

L10.3(13) 如果没有获得消息，那么 OS_Sched()函数则由于等待超时而返回。OS_EventTO()函数将调用该函数的任务从邮箱的任务等待列表中删除。

L10.3(14) 因为没有消息可返回，返回指针被设置为 NULL。调用函数必须检查返回指针或者返回代码的内容，以确定获得的消息是否有效。

10.03 向邮箱发送一则消息，OSMboxPost()

向邮箱发送一则消息的代码如程序清单 L10.4 所列。

程序清单 L10.4 向邮箱发送一则消息，OSMboxPost()

```
INT8U  OSMboxPost (OS_EVENT *pevent, void *msg)
{
#if OS_CRITICAL_METHOD == 3
```

```
    OS_CPU_SR  cpu_sr;
#endif


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                          (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (msg == (void *)0) {
        return (OS_ERR_POST_NULL_PTR);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                       (2)
        OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);                         (3)
        OS_EXIT_CRITICAL();
        OS_Sched();                                                         (4)
        return (OS_NO_ERR);
    }
    if (pevent->OSEventPtr != (void *)0) {                                  (5)
        OS_EXIT_CRITICAL();
        return (OS_MBOX_FULL);
    }
    pevent->OSEventPtr = msg;                                               (6)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L10.4(1)　如果 OS_ARG_CHK_EN 设置为 1，则 OSMboxPost() 函数首先检查 pevent 是否为非空指针，消息参数是否也为非空指针，最后确定事件控制块的类型为邮箱型。

L10.4(2)　OSMboxPost()函数检查是否有任务在等待该邮箱中的消息。如果该事件控制块中的 .OSEventGrp 域非 0，就表明有任务在等待该消息。

L10.4(3)　调用 OS_EventTaskRdy()函数，将其中的最高优先级任务从等待列表中删除(见 6.05 节："使一个任务进入就绪态：OS_EventTaskRdy()")，并加入到系统的就绪任务列表中，准备运行。

L10.4(4)　调用 OS_Sched()函数，检查该任务是否是系统中就绪任务中优先级最高的。如果是，则执行任务切换(仅当 OSMboxPost()函数是由任务调用时)，该就绪任务得以执行。如果该任务不是优先级最高的任务，则 OS_Sched()返回，

OSMboxPost()的调用函数继续运行。

L10.4(5) 此时,应该没有任务在等待该邮箱中的消息。OSMboxPost()函数随后检查邮箱中是否为没有消息。因为邮箱只能保存一则消息,如果邮箱中已存有消息,则将返回"邮箱已满"的错误代码。

L10.4(6) 如果没有任何任务等待该邮箱中的消息,则指向消息的指针就被保存到邮箱中。这样,下一个调用 OSMboxPend()函数的任务就可以立刻得到该消息了。

注意:如果从中断服务子程序中调用 OSMboxPost()函数,那么,这时并不发生任务的切换。因为,中断服务子程序引起的任务切换只发生在中断嵌套的最外层和中断服务子程序调用 OSIntExit()函数时(见3.10节:"μC/OS-II 中的中断")。

10.04 向邮箱发送一则消息,OSMboxPostOpt()

也可以使用另一个功能更强的函数 OSMboxPostOpt()向邮箱中发送消息。为了与μC/OS-II以前的版本兼容,此处保留了2个向邮箱发送消息的函数。OSMboxPostOpt()是新加的函数,可以替代原有的 OSMboxPost()函数。另外,OSMboxPostOpt()函数可以向等待邮箱的所有任务发送消息(广播)。向邮箱发送一则消息的代码如程序清单 L10.5 所列。

程序清单 L10.5 向邮箱发送一则消息,OSMboxPostOpt()

```
INT8U  OSMboxPostOpt (OS_EVENT *pevent, void *msg, INT8U opt)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                          (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (msg == (void *)0) {
        return (OS_ERR_POST_NULL_PTR);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                       (2)
```

```
        if ((opt & OS_POST_OPT_BROADCAST) != 0x00) {                    (3)
            while (pevent->OSEventGrp != 0x00) {                        (4)
                OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);             (5)
            }
        } else {
            OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);                 (6)
        }
        OS_EXIT_CRITICAL();
        OS_Sched();                                                     (7)
        return (OS_NO_ERR);
    }
    if (pevent->OSEventPtr != (void *)0) {                              (8)
        OS_EXIT_CRITICAL();
        return (OS_MBOX_FULL);
    }
    pevent->OSEventPtr = msg;                                           (9)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L10.5(1) 如果 OS_ARG_CHK_EN 设置为 1，则 OSMboxPostOpt()函数首先检查 pevent是否为非空指针，消息参数是否也为非空指针，最后确认事件控制块的类型的确为邮箱型。

L10.5(2) OSMboxPost()函数检查是否有任务在等待该邮箱中的消息。如果该事件控制块中的 .OSEventGrp 域为非 0，就表明有任务在等待消息。

L10.5(3)

L10.5(4)

L10.5(5) 如果 opt 参数中的 OS_POST_OPT_BROADCAST 位置 1，则等待任务都得到了该信息。OS_EventTaskRdy()函数将所有等待任务从等待列表中删除(见 6.05 节："使一个任务进入就绪态，OS_EventTaskRdy()")。

注意：这里禁止中断的时间与等待邮箱中消息的任务数是成正比的。

L10.5(6) 如果没有请求广播，那么只有最高优先级的任务进入就绪态，准备运行。OS_EventTaskRdy()函数将最高优先级任务从等待列表中删除。

L10.5(7) 调用 OS_Sched()函数，检查该任务是否是系统中优先级最高的就绪态任务。如果是，则执行任务切换(仅当 OSMboxPostOpt()函数是由任务调用时)，该就绪任务得以执行。如果该任务不是优先级最高的任务，则 OS_Sched()返回，调用 OSMboxPostOpt()的函数继续运行。

L10.5(8) 如果没有任务在等待该邮箱中的消息，那么发送的消息应放入邮箱中。OSMboxPostOpt()函数将检查邮箱中是否没有消息，因为邮箱中只能保存一则消息。如果向已存有消息的邮箱中发送另一则消息，则将返回"邮箱已满"的错误代码。

L10.5(9) OSMboxPostOpt()函数将消息保存在邮箱中。

注意：如果 OSMboxPostOpt()函数是从中断服务子程序中调用的，那么，这时并不发生任务的切换。因为，中断服务子程序引起的任务切换只发生在中断嵌套的最外层和中断服务子程序对 OSIntExit()函数的调用时(见3.10节："μC/OS-II 中的中断")。

10.05 无等待地从邮箱中得到一则消息，OSMboxAccept()

如果邮箱为空，则应用程序也可以从邮箱中得到消息，而不必使任务进入睡眠状态。这种情况下的操作是由 OSMboxAccept()函数完成的，其源代码见程序清单 L10.6。

程序清单 L10.6 无等待地从邮箱中得到消息

```
void    *OSMboxAccept (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    void         *msg;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                    (1)
        return ((void *)0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                  (2)
        return ((void *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    msg                 = pevent->OSEventPtr;                         (3)
    pevent->OSEventPtr = (void *)0;                                   (4)
    OS_EXIT_CRITICAL();
    return (msg);                                                     (5)
}
```

L10.6(1)

L10.6(2) 若 OS_CFG.H 文件中的 OS_ARG_CHK_EN 设置为1，则 OSMboxAccept()函数首先检查 pevent 是否为非空指针，并且 pevent 所指向的事件控制块是否是由 OSMboxCreate()函数建立的。

L10.6(3) OSMboxAccept() 函数得到邮箱中的当前内容，并判断是否有消息可用(为非空指针)。

L10.6(4) 如果邮箱中有消息，则把邮箱清空。即使邮箱内已经是一 NULL 指针了，也做一次清空的操作。这样做是出于性能方面的考虑。

L10.6(5) 将邮箱中原有的内容返回，传给 OSMboxAccept()的调用函数。

OSMboxAccept()函数的调用函数必须检查返回值。如果该值是 NULL，则说明邮箱是空的，没有可用的消息；如果该值是非 NULL 值，则说明邮箱中有消息可用。中断服务子程序在试图得到一则消息时，应该使用 OSMboxAccept()函数，而不能使用 OSMboxPend()函数。

OSMboxAccept()函数的另一个用途是，清空邮箱中现有的内容。

10.06 查询一个邮箱的状态，OSMboxQuery()

OSMboxQuery()函数使应用程序可以随时查询一个邮箱的当前状态。程序清单 L10.7 是该函数的源代码。OSMboxQuery()函数需要 2 个参数：一个是指向邮箱的指针 pevent，该指针是在建立该邮箱时，由 OSMboxCreate()函数返回的；另一个是指向用来保存有关邮箱信息的 OS_MBOX_DATA(见 uCOS_II.H)数据结构的指针 pdata。在调用 OSMboxCreate()函数之前，必须先定义该结构变量，用来保存有关邮箱的信息。之所以定义一个新的数据结构，是因为这里关心的只是和特定邮箱有关的内容，而非整个 OS_EVENT 数据结构的内容。OS_MBOX_DATA只包含邮箱中的消息指针(.OSMsg)和该邮箱现有的等待任务列表(.OSEventTbl[]和.OSEventGrp)，而 OS_EVENT 中还包含了另外 2 个域(.OSEventCnt 和.OSEventType)。

程序清单 L10.7 查询邮箱的状态

```
INT8U  OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    INT8U        *psrc;
    INT8U        *pdest;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                          (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                        (2)
        return (OS_ERR_EVENT_TYPE);
```

```
    }
#endif
    OS_ENTER_CRITICAL();
    pdata->OSEventGrp = pevent->OSEventGrp;                                    (3)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];

#if OS_EVENT_TBL_SIZE > 0
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 1
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 4
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *pdest            = *psrc;
#endif
    pdata->OSMsg = pevent->OSEventPtr;                                         (4)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L10.7(1)

L10.7(2)　同前面的所有函数一样，若 OS_ARG_CHK_EN 设置为 1，则 OSSemQuery()函数首先检查参数指针 pevent 是否为非空指针，并且 pevent 指向的事件控制块是否为邮箱类型。

L10.7(3)　OSMboxQuery() 函数复制等待任务列表。此处，之所以使用条件编译产生的内嵌代码，而不使用循环语句，原因是这样做代码执行速度更快。

L10.7(4)　将邮箱中的当前消息从 OS_EVENT 数据结构复制到 OS_MBOX_DATA 数据结构中。

10.07　用邮箱作为二值信号量

一个邮箱可以被用做二值信号量。首先，在初始化时，将邮箱设置为一个非空指针（如 void *1）。这样，一个任务可以调用 OSMboxPend()函数请求一个信号量，然后通过调用 OSMboxPost()函数释放一个信号量。程序清单 L10.8 说明了这个过程。如果只需要二值信号量和邮箱，那么使用这种方法可以节省代码空间。这时，可以将 OS_MBOX_EN 置为 1，将 OS_SEM_EN 设为 0，只使用邮箱，而不同时使用邮箱和信号量。

程序清单 L10.8　用邮箱作为二值信号量

```
OS_EVENT *MboxSem;



void Task1 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSMboxPend(MboxSem, 0, &err);          /* 获得对资源的访问权 */
        .
        .    /* 任务获得信号量，对资源进行访问 */
        .
        OSMboxPost(MboxSem, (void *)1);        /* 释放对资源的访问权 */
    }
}
```

10.08　用邮箱实现延时，而不使用 OSTimeDly()

邮箱的等待超时功能可以用来模仿 OSTimeDly()函数的延时，如程序清单 L10.9 所列。如果在指定的时间段 TIMEOUT 内没有消息到来，Task1()函数将继续执行。实质上，这与

OSTimeDly(TIMEOUT)功能很相似；但是，如果Task2()在指定的时间结束之前，向该邮箱发送了一个“哑”(dummy)消息，那么Task1()就会提前开始继续执行。这与调用函数OSTimeDlyResume()的功能是一样的。

注意：这里忽略了对返回消息的检查，因为此时并不需要从另一任务或者中断服务子程序得到消息。

程序清单 L10.9 用邮箱实现延时

```
OS_EVENT *MboxTimeDly;


void Task1 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSMboxPend(MboxTimeDly, TIMEOUT, &err);    /* 延时该任务 */
        .
        .     /* 延时结束后执行的代码 */
        .
    }
}


void Task2 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSMboxPost(MboxTimeDly, (void *)1);        /* 取消任务1的延时 */
        .
        .
    }
}
```

第 11 章

消息队列管理

消息队列(简称队列)是 μC/OS - II 中的另一种通信机制。它允许一个任务或者中断服务子程序向另一个任务发送以指针方式定义的变量或其他任务。因具体的应用不同,每个指针指向的包含了消息的数据结构的变量类型也有所不同。μC/OS - II 提供了 9 个对消息队列进行操作的函数:OSQCreate()、OSQDel()、OSQPend()、OSQPost()、OSQPostFront()、OSQPostOpt()、OSQAccept()、OSQFlush()和 OSQQuery()。

为了使用 μC/OS - II 的消息队列功能,需要在 OS_CFG.H 文件中设置配置常量。特别的,如表 T11.1 列出的被编译的功能以及在 OS_CFG.H 文件中相应的配置常量的值。应该注意到的是,当 OS_Q_EN 和 OS_MAX_QS 常量设置为 0 时,不能使用任何一种队列功能。也就是说,当要使用某个特定的功能时,只需简单地将相应的配置常量设置为 1。与其他功能不同的是,OSQCreate()和 OSQPend()函数不能单独禁用。这是因为在使用 μC/OS - II 的消息队列管理时,这 2 个函数是必需的。此外,至少要调用 OSQPost()、OSQPostFront()及 OSQPostOpt()功能函数中的 1 个函数。

表 T11.1　OS_CFG.H 文件中的消息队列配置常量

μC/OS - II 消息队列函数	OS_CFG.H 中置 1 允许相应函数
OSQAccept()	OS_Q_ACCEPT_EN
OSQCreate()	
OSQDel()	OS_Q_DEL_EN
OSQFlush()	OS_Q_FLUSH_EN
OSQPend()	
OSQPost()	OS_Q_POST_EN
OSQPostFront()	OS_Q_POST_FRONT_EN
OSQPostOpt()	OS_Q_POST_OPT_EN
OSQQuery()	OS_Q_QUERY_EN

图 F11.1 是任务、中断服务子程序及消息队列之间的关系的流程图。其中,消息队列的符号很像是带有多个入口的邮箱。实际上,可以将消息队列看做是多个邮箱组成的数组,只是它们共用一个等待任务列表。图中的沙漏代表一个在调用 OSQPend()函数时定义的超时功能。每个指针所指向的数据结构类型是由具体的应用程序决定的。*N* 代表消息队列中总的可以保存的消息数。当调用 OSQPend()或者 OSQAccept()之前,调用 *N* 次 OSQPost()、OSQPostFront()、或者 OSQPostOpt(),就会把消息队列填满。

就像图中表示的那样,任务或者中断服务子程序都可以调用函数 OSQPost()、OSQPostFront()、OSQPostOpt()、OSQFlush()、或 OSQAccept();但是,只有任务才能调用 OSQDel()、OSQPend()及 OSQQuery()。

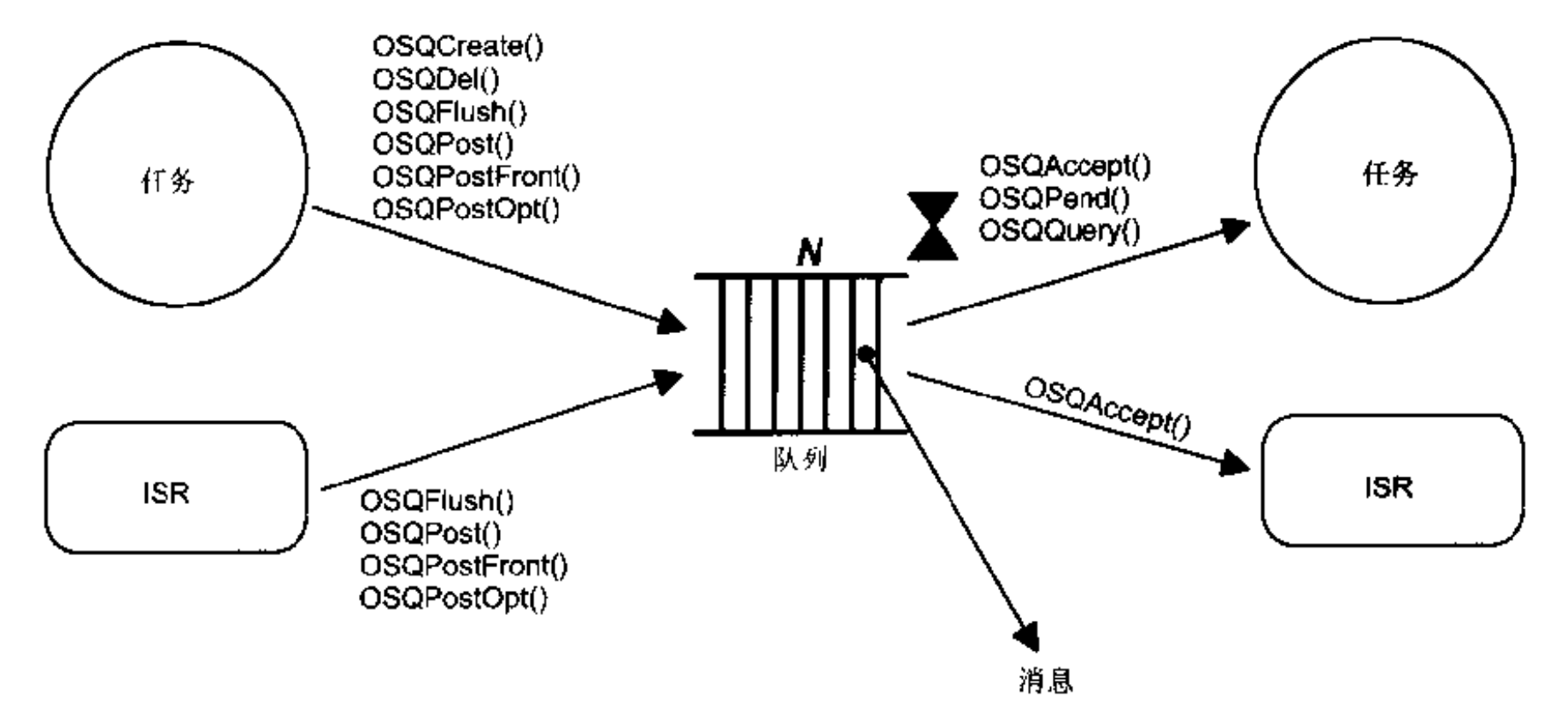


图 F11.1 任务、中断服务子程序及消息队列之间的关系

图 F11.2 是实现消息队列所需要的各种数据结构。

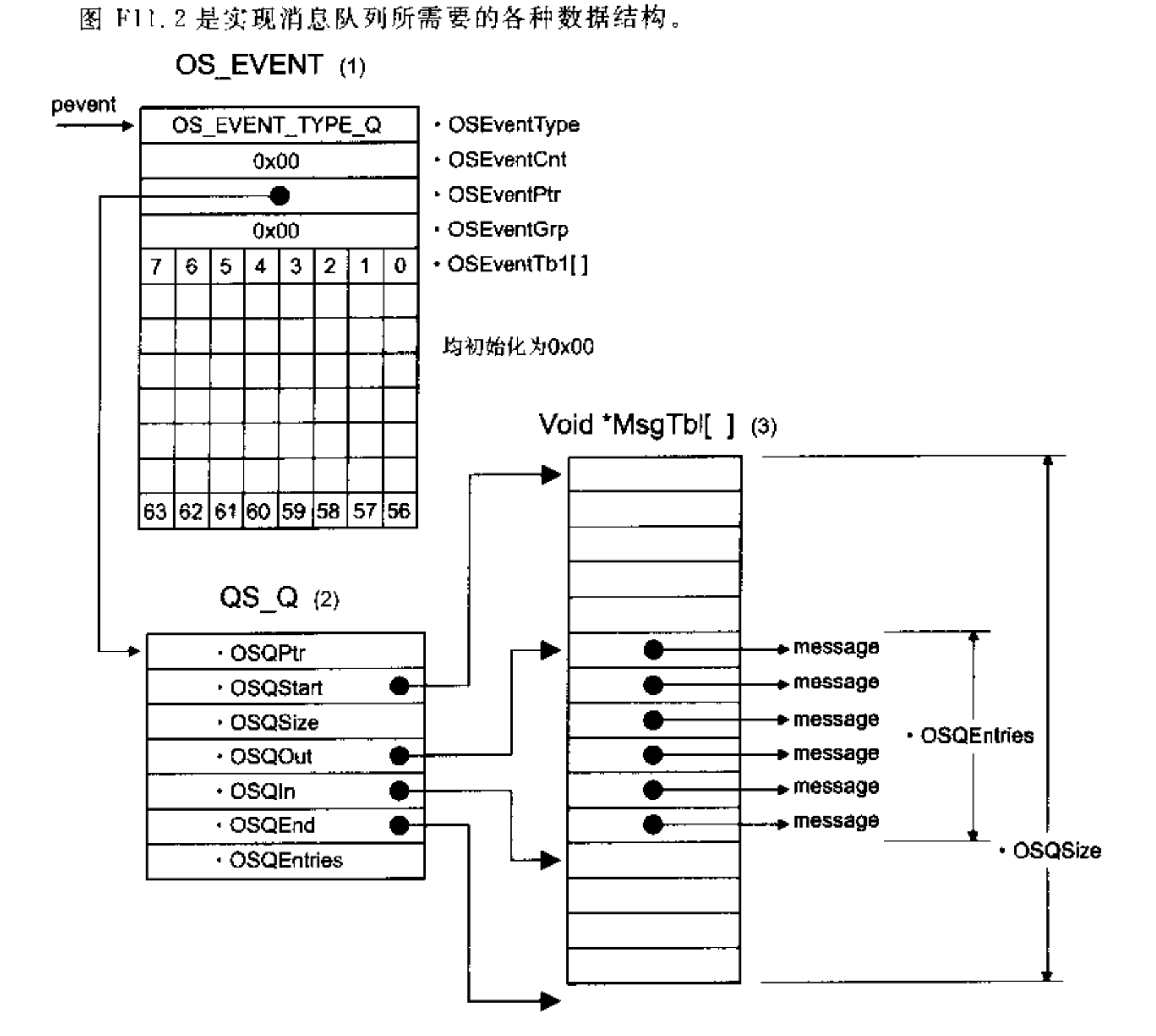


图 F11.2 用于消息队列的数据结构

F11.2(1)　这里也需要事件控制块 ECB 记录等待任务列表，而且，事件控制块可以使多个消息队列的操作与信号量函数、互斥型信号量及邮箱的函数使用相同的代码。

F11.2(2)　当建立消息队列时，一个队列控制块（如 OS_Q 结构，见 OS_Q.C 文件）也同时分配给这个控制块，并通过 OS_EVENT 中的 .OSEventPtr 域链接到对应的事件控制块 ECB。

F11.2(3)　在建立一个消息队列之前，必须先定义一个含有与消息队列最大消息数相同的指针数组。换句话说，这个数组中的单元数与队列中的单元数一致。数组的起始地址、数组的大小以及数组中的元素数目作为参数传递给 OSQCreate()函数。事实上，并不是必须使用指针数组结构，也不必占用连续的存储空间。

文件 OS_CFG.H 中的配置常量 OS_MAX_QS 定义了在 μC/OS-II 中可以使用的最大消息队列数，这个值应该大于 0。μC/OS-II 在初始化时建立一个空闲的队列控制块链表，如图 F11.3 所示。

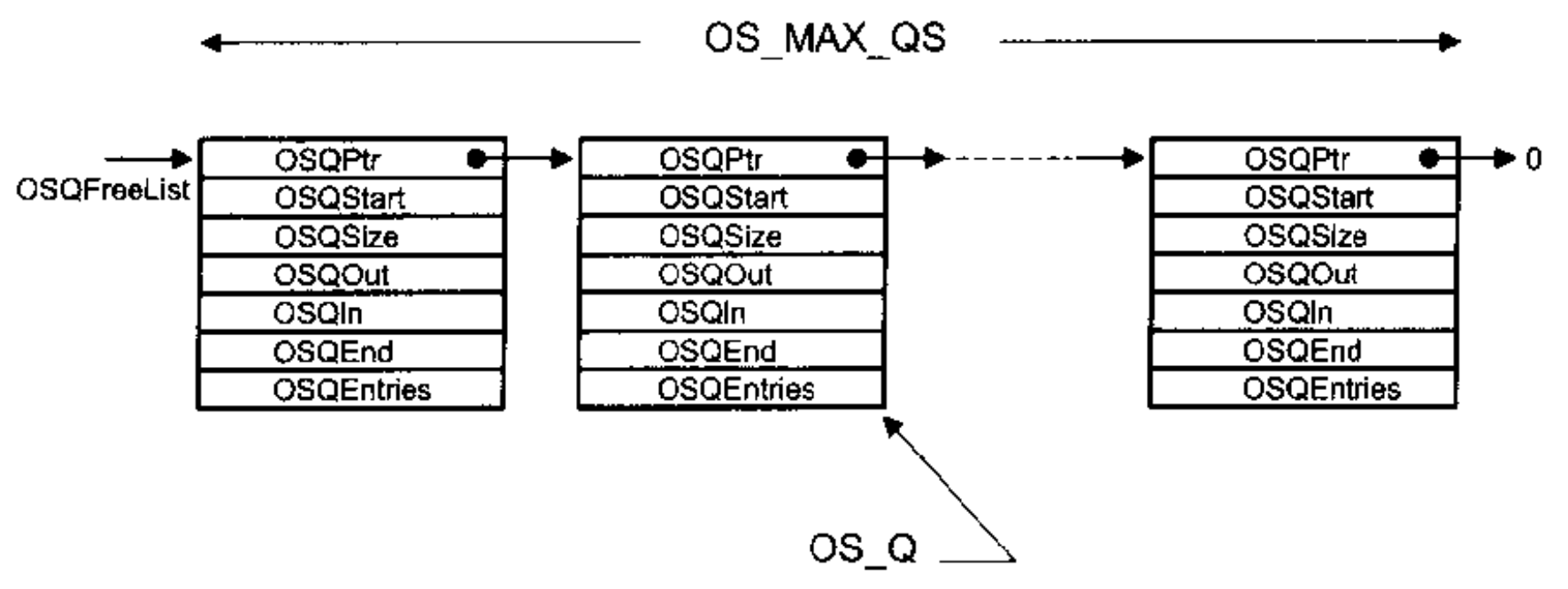


图 F11.3　空闲队列控制块链表

队列控制块是一个用于维护消息队列信息的数据结构，包含了以下的一些域。这里，仍然在各变量前加一个“.”，表示它们是数据结构中的一个域，而非一个简单的变量。

.OSQPtr

在空闲队列控制块中，链接所有的队列控制块。一旦建立了消息队列，该域就不再有用了。

.OSQStart

指向消息队列的指针数组的起始地址的指针。应用程序在使用消息队列之前，必须先定义该数组。

.OSQEnd

指向消息队列结束单元的下一个地址的指针。该指针使得消息队列构成一个循环的缓冲区。

.OSQIn

指向消息队列中插入下一条消息的位置的指针。当 .OSQIn 和 .OSQEnd 相等时，.OSQIn被调整到指向消息队列的起始单元。

.OSQOut

指向消息队列中下一个取出消息的位置的指针。当 .OSQOut 和 .OSQEnd 相等时，

.OSQOut被调整为指向消息队列的起始单元。.OSQOut 通常也用于插入一则消息(参见 OSQPostFront()和 OSQPostOpt())。

.OSQSize

消息队列中可容纳的总的消息数。该值是在建立消息队列时,由应用程序定义的。在 μC/OS-II 中,该值最大可以是 65 535。

.OSQEntries

消息队列中当前的消息数。当消息队列为空时,该值为 0;当消息队列填满以后,该值与 .OSQSize值相等。在消息队列刚刚建立时,该值为 0。

消息队列最根本的部分是一个循环缓冲区,如图 F11.4 所示。

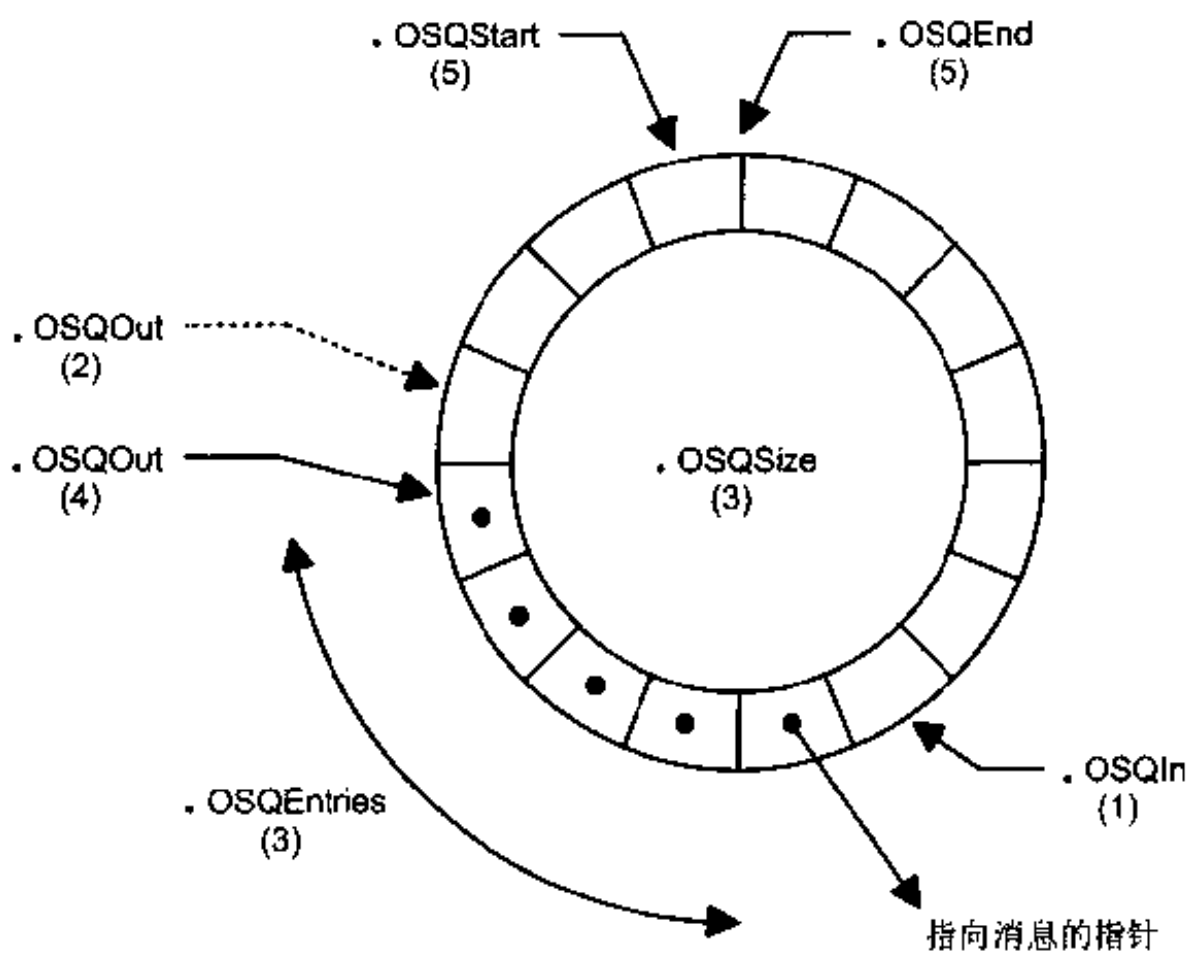


图 F11.4 消息队列是一个由指针组成的循环缓冲区

F11.4(1)

F11.4(3) 其中的每个单元包含一个指针。当队列未满时,.OSQIn 指向下一个存放消息的地址单元。如果在 .OSQIn 指向的单元插入新的指向消息的指针,就构成先进先出(FIFO)的队列,这一任务由 OSQPost()完成。

F11.4(2) 如果在 .OSQOut 指向单元的下一个单元插入新的指针,μC/OS-II 就构成后进先出(LIFO)的队列(参见 OSQPostFront()和 OSQPostOpt())。

F11.4(4) 当 .OSQEntries 和 .OSQSize 相等时,说明队列已满。消息指针总是从 .OSQOut指向的单元取出。

F11.4(5) 指针 .OSQStart 和 .OSQEnd 作为简单的标记,定义了消息指针数组的头尾,以便在 .OSQIn 和 .OSQOut 到达队列的边缘时,进行边界检查和必要的指针调整,实现循环功能。

11.00　建立一个消息队列,OSQCreate()

在使用一个消息队列(或者是一个队列)之前,首先需要建立它。可以通过调用函数OSQCreate()函数建立一个队列,并且要赋值给它2个参数:指向消息数组的指针和数组的大小。该指针数组必须声明为 void 类型,如下所示:

void *MyArray0[Msg[SIZE];

同时将 MyArray0[Msg[]的地址及数组的大小传递给 OSQCreate()。这样,这个消息队列就被初始化为空,即其中不包含任何消息。

程序清单 L11.1 是建立一个消息队列的源代码。

程序清单 L11.1　建立一个消息队列

```
OS_EVENT    *OSQCreate (void **start, INT16U size)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;                                                    (1)
#endif
    OS_EVENT    *pevent;
    OS_Q        *pq;


    if (OSIntNesting > 0) {                                                 (2)
        return ((OS_EVENT *)0);
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                               (3)
    if (OSEventFreeList != (OS_EVENT *)0) {
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                                          (4)
        OS_ENTER_CRITICAL();
        pq = OSQFreeList;
        if (pq != (OS_Q *)0) {
            OSQFreeList          = OSQFreeList->OSQPtr;
            OS_EXIT_CRITICAL();
            pq->OSQStart         = start;                                   (5)
            pq->OSQEnd           = &start[size];
```

```
        pq->OSQIn           = start;
        pq->OSQOut          = start;
        pq->OSQSize         = size;
        pq->OSQEntries      = 0;
        pevent->OSEventType = OS_EVENT_TYPE_Q;                          (6)
        pevent->OSEventCnt  = 0;
        pevent->OSEventPtr  = pq;
        OS_EventWaitListInit(pevent);                                   (7)
    } else {
        pevent->OSEventPtr = (void *)OSEventFreeList;                   (8)
        OSEventFreeList    = pevent;
        OS_EXIT_CRITICAL();
        pevent = (OS_EVENT *)0;
    }
    return (pevent);                                                    (9)
}
```

L11.1(1) 分配名为 cpu_sr 的局部变量，用于支持 OS_CRITICAL_METHOD 方法 #3。

L11.1(2) 只有在确认没有从中断服务子程序中调用此程序后，OSQCreate()才能启动。这是因为不允许在中断服务子程序中调用 OSQCreate()。所有 μC/OS-II 内核对象都必须在任务级代码中建立，或者是在多任务启动之前建立。

L11.1(3) OSQCreate()试图从空余事件控制块 ECB 链表中取得一个事件控制块 ECB (见图 F6.5)，并对剩下的空余事件控制块链表的指针做相应的调整。

L11.1(4) 如果有事件控制块可以使用，OSQCreate()就会试图从空闲队列控制块链表(见图 F11.3)中得到一个队列控制模块(OS_Q)，并且对链表做出相应的调整。

L11.1(5)

L11.1(6) 如果空闲队列链表中有控制块可用，与其他事件控制模块 ECB 一样，要对其设置和初始化。

注意：将 .OSEventType 域设成 OS_EVENT_TYPE_Q 类型，是为了在以后的消息队列函数中可以检测事件控制模块 ECB 类型的合法性。

L11.1(7) 等待列表清 0，意味着当前没有任务在等待消息队列。

L11.1(8) 如果有事件控制模块 ECB 可用，但不是队列控制模块类型，那么事件控制模块 ECB 就要被返还给空余控制块链表，而建立一个队列的请求也无法完成，除非有空闲队列控制模块可用。

L11.1(9) 如果消息队列建立成功，则 OSQCreate()就会返回一个指针；否则，就返回空指针。消息队列成功建立之后返回的这个指针用于以后对消息队列的操作；因此，该指针可以看作是相应消息队列的句柄。

11.01　删除一个消息队列,OSQDel()

程序清单 L11.2 列出了删除一个消息队列的源代码。只有在 OS_CFG.H 文件中将 OS_Q_DEL_EN设置为 1,以下的代码才会被编译。使用此功能应该特别小心,多任务可能会试图访问一个删除了的消息队列。一般地说,在删除一个消息队列之前,应该首先删除所有能用到这个消息队列的任务。

程序清单 L11.2　删除一个消息队列

```
OS_EVENT   *OSQDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    tasks_waiting;
    OS_Q       *pq;


    if (OSIntNesting > 0) {                                                    (1)
        *err = OS_ERR_DEL_ISR;
        return ((OS_EVENT *)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                             (2)
        *err = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                              (3)
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                          (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:
```

```
        if (tasks_waiting == FALSE) {
            pq                  = pevent->OSEventPtr;                          (5)
            pq->OSQPtr          = OSQFreeList;
            OSQFreeList         = pq;
            pevent->OSEventType = OS_EVENT_TYPE_UNUSED;                        (6)
            pevent->OSEventPtr  = OSEventFreeList;                             (7)
            OSEventFreeList     = pevent;
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
            return ((OS_EVENT *)0);                                            (8)
        } else {
            OS_EXIT_CRITICAL();
            *err = OS_ERR_TASK_WAITING;
            return (pevent);
        }

    case OS_DEL_ALWAYS:
        while (pevent->OSEventGrp != 0x00) {                                   (9)
            OS_EventTaskRdy(pevent, (void *)0, OS_STAT_Q);                     (10)
        }
        pq                  = pevent->OSEventPtr;                              (11)
        pq->OSQPtr          = OSQFreeList;
        OSQFreeList         = pq;
        pevent->OSEventType = OS_EVENT_TYPE_UNUSED;                            (12)
        pevent->OSEventPtr  = OSEventFreeList;                                 (13)
        OSEventFreeList     = pevent;
        OS_EXIT_CRITICAL();
        if (tasks_waiting == TRUE) {
            OS_Sched();                                                        (14)
        }
        *err = OS_NO_ERR;
        return ((OS_EVENT *)0);                                                (15)

    default:
        OS_EXIT_CRITICAL();
        *err = OS_ERR_INVALID_OPT;
        return (pevent);
    }
}
```

1.11.2(1) 在启动OSQDel()之前，首先要确认此功能不是在中断服务子程序中调用的，因为这是不允许的。

L11.2(2)

L11.2(3) 如果OS_ARG_CHK_EN(见OS_CFG.H文件)设置为1,则OSQDel()确认pevent不是一个空指针,而是队列建立时得到的指向队列控制模块的指针。

L11.2(4) OSQDel()确定是否有任务在等待消息队列的消息。标志tasks_waiting也被相应地赋值。根据在调用中定义的选项属性(如opt)的不同,OSQDel()执行不同的操作。当opt值为OS_DEL_NO_PEND时,OSQDel()仅在无任务等待队列消息时才删除队列;而opt置为OS_DEL_ALWAYS时,OSQDel()不管有没有任务在等待队列的消息,立即删除队列。

L11.2(5) 当opt被设置为OS_DEL_NO_PEND,并且队列中没有任务正在等待时,就可以通过把队列控制模块返还给空闲链表,来启动OSQDel()。

L11.2(6)

L11.2(7) OSQDel()将事件控制模块ECB标记为未使用状态,同时这个事件控制模块ECB也被返还给空余控制块链表,以便用于建立其他消息队列(或者是其他基于事件控制模块的函数)。

L11.2(8) 注意:OSQDel()返回了空指针,此时,这个队列已经不能再使用原来的指针来访问了。应该按照以下方式调用OSQDel():

```
QPtr = OSQDel(QPtr, opt, &err);
```

若有任一任务正在等待队列中的消息,OSQDel()就会返回一出错代码(如OS_ERR_TASK_WAITING)。这是因为,定义了OS_DEL_NO_PEND就表明:在有任务等待消息队列中消息的情况下,不得删除消息队列。

L11.2(9)

L11.2(10) 当opt被置为OS_DEL_ALWAYS时,所有等待消息队列的任务都会进入就绪态。每个任务都会认为接收到了一则消息,而事实上并没有任何消息发出,任务应该去检测返回的指针,确认不是空指针。同时,应该注意到,当任务进入就绪态时,中断是关掉的。当然,这样就增加了系统的中断延迟时间。

L11.2(11) OSQDel()将队列控制模块返还给空余控制块链表。

L11.2(12)

L11.2(13) 在所有的挂起任务进入就绪态之后,OSQDel()把事件控制模块ECB标记为未使用,这个事件控制模块ECB也同时被返回给空余控制块链表。

L11.2(14) 只有当有任务在等待队列中的消息时,调度器才会起作用。

L11.2(15) 再一次强调,OSQDel()返回了空指针,因为此时消息队列已经不能再通过原来的指针来访问了。

11.02 等待消息队列中的消息,OSQPend()

程序清单L11.3是等待消息队列中的消息的源代码。

程序清单 1.11.3 等待消息队列中的消息

```
void   *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    void        *msg;
    OS_Q        *pq;


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_PEND_ISR;
        return ((void *)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                   (2)
        *err  = OS_ERR_PEVENT_NULL;
        return ((void *)0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (3)
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {                                        (4)
        msg = *pq->OSQOut++;                                         (5)
        pq->OSQEntries--;                                            (6)
        if (pq->OSQOut == pq->OSQEnd) {                              (7)
            pq->OSQOut = pq->OSQStart;                               (8)
        }
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (msg);                                                (9)
    }
    OSTCBCur->OSTCBStat |= OS_STAT_Q;                                (10)
    OSTCBCur->OSTCBDly  = timeout;                                   (11)
    OS_EventTaskWait(pevent);                                        (12)
    OS_EXIT_CRITICAL();
    OS_Sched();                                                      (13)
    OS_ENTER_CRITICAL();
```

```
    msg = OSTCBCur->OSTCBMsg;                                          (14)
    if (msg != (void *)0) {
        OSTCBCur->OSTCBMsg      = (void *)0;                           (15)
        OSTCBCur->OSTCBStat     = OS_STAT_RDY;
        OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
        OS_EXIT_CRITICAL();
        *err                    = OS_NO_ERR;
        return (msg);
    }
    OS_EventTO(pevent);                                                (16)
    OS_EXIT_CRITICAL();
    *err = OS_TIMEOUT;
    return ((void *)0);                                                (17)
}
```

L11.3(1)　不能在中断服务子程中调用 OSQPend()，因为中断服务子程序不能等待。可以调用 OSQAccept()(参见 11.06 节："无等待地从消息队列中获得消息，OSQAccept()")。

L11.3(2)

L11.3(3)　如果 OS_ARG_CHK_EN(见 OS_CFG.H)置为 1，则 OSQPend()确定 pevent 不是空指针，并且 pevent 指向的事件控制模块 ECB 也已经由 OSQCreate()函数建立。

L11.3(4)

L11.3(5)　当 .OSQEntries 大于 0 时，表明已有消息可用。此时，OSQPend()得到事件控制模块的 .OSQOut 域所指向的消息，将指向消息的指针复制到 msg 变量中，并让 .OSQOut 指针指向队列中的下一个单元。

L11.3(6)　OSQPend() 将队列中的有效消息数减 1，因为先前的操作削减了一则消息(删除了一则最旧的消息)。

L11.3(7)

L11.3(8)　因为消息队列是一个循环的缓冲区，OSQPend()函数需要检查 .OSQOut 是否超出了队列中的最末一个单元。当发生这种越界时，就要将 .OSQOut 重新调整到指向队列的起始单元。

L11.3(9)　从队列中抽出的消息被返回给 OSQPend()的调用者。这是调用 OSQPend()函数时所期望的，也是执行 OSQPend()函数最快的路径。

如果消息队列是空的，则调用本函数的任务被挂起，直到有别的任务(或者中断服务子程序)通过消息队列发送一则消息(参见 11.04 节："向消息队列发送一则消息(LIFO)，OSQPostFront()")。OSQPend()允许定义一个超时时限(定义为整型值)作为一个参量(timeout)。此特性可以有效地避免无限期地等待消息队列中的消息。如果这个超时时限非 0，OSQPend()就挂起这个任务，直到队列接收到一则消息或者等待超时。需要注意的是，定

义超时时限值为0，表示任务将无限期地等待消息的到来。

L11.3(10) OSQPend()设置任务的任务控制块TCB中的状态标志，以此表明等待消息队列消息的任务被挂起。

L11.3(11) 超时值被保存在任务控制块TCB中。这样，超时时限值就可以由函数OSTimeTick()做减1操作。如同第3章中描述的（见3.11节："时钟节拍"），如果任务的.OSTCBDly域非0，则OSTimeTick()函数会在每个时钟节拍中给已建立任务的.OSTCBDly域减1。

L11.3(12) 实际上，让任务进入睡眠状态的工作是由OS_EventTaskWait()函数完成的（参见6.06节："使一个任务进入等待某事件发生状态，OS_EventTaskWait()"）。

L11.3(13) 由于正在调用的任务不再是就绪态，调度器会安排下一个进入就绪态的优先级最高的任务运行。在用户程序中，调用OSQPend()函数的任务并不知道消息没有到来之前自己被挂起了。当队列接收到了一则消息（或者是等待超时）时，OSQPend()函数就会在调用OS_Sched()之后恢复执行。

L11.3(14) 当OS_Sched()返回后，OSQPend()会检查OSQPost()是否已将消息放在任务的任务控制块TCB中了。

L11.3(15) 如果消息确实存在，那么调用就是成功的，同时这则消息就会被返回给调用者。

L11.3(16) 如果没有接收到消息，那么OSQPend()一定是由于等待超时返回的，于是调用OS_EventTO()函数，把该任务从等待消息队列消息的任务列表中删除。

L11.3(17) 注意：返回了空指针NULL，因为没有收到消息。调用本函数的任务或者检查返回的指针指向的消息，或者检查返回的出错代码，以确定是否接收到了合法的消息。

11.03 向消息队列发送一则消息(FIFO)，OSQPost()

程序清单L11.4列出了向消息队列发送一则消息的源代码。

程序清单L11.4 向消息队列发送一则消息(FIFO)，OSQPost()

```
INT8U  OSQPost (OS_EVENT *pevent, void *msg)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    OS_Q         *pq;
```

```
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                  (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (msg == (void *)0) {                                         (2)
        return (OS_ERR_POST_NULL_PTR);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                   (3)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                               (4)
        OS_EventTaskRdy(pevent, msg, OS_STAT_Q);                    (5)
        OS_EXIT_CRITICAL();
        OS_Sched();                                                 (6)
        return (OS_NO_ERR);
    }
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries >= pq->OSQSize) {                            (7)
        OS_EXIT_CRITICAL();
        return (OS_Q_FULL);
    }
    *pq->OSQIn++ = msg;                                             (8)
    pq->OSQEntries++;                                               (9)
    if (pq->OSQIn == pq->OSQEnd) {                                  (10)
        pq->OSQIn = pq->OSQStart;
    }
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L11.4(1)

L11.4(2)

L11.4(3)　如果 OS_CFG.H 文件中的 OS_ARG_CHK_EN 置为 1，则 OSQPost()检查，以确定 3 件事：pevent 为非空指针；发送的消息为非空指针；事件控制模块 ECB 是消息队列类型。

L11.4(4)　OSQPost()函数检查是否有任务在等待该消息队列中的消息。当事件控制块 ECB 的 .OSEventGrp 域为非 0 值时，说明该消息队列的等待任务列表中有等待消息的任务。

L11.4(5)　调用 OSEventTaskRdy()函数（见 6.05 节："使一个任务进入就绪态，OSEventTaskRdy()"），从列表中取出最高优先级的任务，并将它置于就

绪态。

L11.4(6) 调用函数 OSSched()进行任务调度。如果上面进入就绪态的任务是就绪态任务中优先级最高的任务，且 OSQPost()函数是被任务级调用的，就执行任务切换，让优先级最高的任务运行；如果不是，则 OSSched()函数返回，让调用 OSQPost()函数的任务继续执行。

L11.4(7) 如果没有任务等待该消息队列中的消息，而且此时消息队列未满，则指向该消息的指针被插入到消息队列中。如果消息队列中的消息已满，函数就会返回一个出错代码。

L11.4(8)

L11.4(9) 如果没有任务等待消息队列的消息，并且消息队列未满，那么收到的消息就会被插入到下一个空缓冲区中(FIFO)，同时队列中的消息数加1。

L11.4(10) OSQPost() 调整环形缓冲区的指针，为下一次消息发送做准备。

注意：如果 OSQPost()函数是由中断服务子程序调用的，那么即使产生了更高优先级的任务，也不会在调用 OSSched()函数时发生任务切换。一直要等到中断嵌套的最外层中断服务子程序调用 OSIntExit()函数时，才能进行任务调度(见3.10节："μC/OS-II 中的中断")。

11.04 向消息队列发送一则消息(LIFO)，OSQPostFront()

OSQPostFront()函数同 OSQPost()基本上是一样的，只是在插入新的消息到消息队列中时，使用 .OSQOut，而不是 .OSQIn，作为指向下一个插入消息的单元指针。程序清单L11.5是它的源代码。

程序清单 L11.5 向消息队列发送一则消息(LIFO)，OSQPostFront()

```
INT8U  OSQPostFront (OS_EVENT *pevent, void *msg)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    OS_Q         *pq;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {
        return (OS_ERR_PEVENT_NULL);
    }
    if (msg == (void *)0) {
        return (OS_ERR_POST_NULL_PTR);
```

```
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {
        OS_EventTaskRdy(pevent, msg, OS_STAT_Q);
        OS_EXIT_CRITICAL();
        OS_Sched();
        return (OS_NO_ERR);
    }
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries >= pq->OSQSize) {
        OS_EXIT_CRITICAL();
        return (OS_Q_FULL);
    }
    if (pq->OSQOut == pq->OSQStart) {                                    (1)
        pq->OSQOut = pq->OSQEnd;                                         (2)
    }
    pq->OSQOut--;                                                        (3)
    *pq->OSQOut = msg;
    pq->OSQEntries++;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L11.5(1)

L11.5(2)　值得注意的是，.OSQOut 指针指向的是已经插入了消息指针的单元，所以在插入新的消息指针之前，必须先将 .OSQOut 指针在消息队列中前移一个单元。如果 .OSQOut 指针指向的当前单元是队列中的第一个单元，这时再前移就会发生越界。实际上该“前移”意味着指针指向了队列的末尾。

L11.5(3)　然而，由于 .OSQEnd 指向的是消息队列中最后一个单元的下一个单元，因此 .OSQOut 必须被调整到指向队列的有效范围内。因为 OSQPend() 函数取出的消息是由 OSQPostFront() 函数刚刚插入的，这样，OSQPostFront() 函数就实现了一个后进先出的消息队列。

11.05　向消息队列发送一则消息(FIFO 或 LIFO)，OSQPostOpt()

可以调用名为 OSQPostOpt() 的函数，以可替代的、更灵活的方式给消息队列发送消息。

这里给出了3个发送消息给消息队列的函数，为的是兼容 μC/OS-II 以前的版本。其中，OSQPostOpt()是新函数。通过调用该函数，可以完全替代 OSQPost()和 OSQPost-Front()函数。此外，OSQPostOpt() 允许发送消息给所有消息队列中等待消息的任务(广播方式)。程序清单 L11.6 列出了这个函数的源代码。

程序清单 L11.6　向消息队列发送一则消息(广播、FIFO 或 LIFO)，OSQPostOpt()

```
INT8U  OSQPostOpt (OS_EVENT *pevent, void *msg, INT8U opt)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    OS_Q         *pq;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                          (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (msg == (void *)0) {                                                 (2)
        return (OS_ERR_POST_NULL_PTR);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                           (3)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                       (4)
        if ((opt & OS_POST_OPT_BROADCAST) != 0x00) {                        (5)
            while (pevent->OSEventGrp != 0x00) {                            (6)
                OS_EventTaskRdy(pevent, msg, OS_STAT_Q);
            }
        } else {
            OS_EventTaskRdy(pevent, msg, OS_STAT_Q);                        (7)
        }
        OS_EXIT_CRITICAL();
        OS_Sched();                                                         (8)
        return (OS_NO_ERR);
    }
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries >= pq->OSQSize) {                                    (9)
        OS_EXIT_CRITICAL();
        return (OS_Q_FULL);
```

```
    }
    if ((opt & OS_POST_OPT_FRONT) != 0x00) {                                    (10)
        if (pq->OSQOut == pq->OSQStart) {                                       (11)
            pq->OSQOut = pq->OSQEnd;
        }
        pq->OSQOut--;
        *pq->OSQOut = msg;
    } else {
        *pq->OSQIn++ = msg;                                                     (12)
        if (pq->OSQIn == pq->OSQEnd)
            pq->OSQIn = pq->OSQStart;
        }
    }
    pq->OSQEntries++;                                                           (13)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L11.6(1)

L11.6(2)

L11.6(3)　如果 OS_CFG.H 文件中的 OS_ARG_CHK_EN 置为 1，则 OSQPost() 检查，以确定 3 件事：pevent 为非空指针；发送的消息为非空指针；事件控制块 ECB 是消息队列类型。

L11.6(4)　OSQPost()函数检查是否有任务在等待该消息队列中的消息。当事件控制块的 .OSEventGrp 域为非 0 值时，该消息队列的等待消息的任务列表中有等待着的任务。

L11.6(5)

L11.6(6)　如果将 opt 参数中的 OS_POST_OPT_BROADCAST 位置为 1，则所有正在等待消息的任务都能接收到这则消息，并且被 OS_EventTaskRdy()从等待列表中删除(见 6.05 节："使一个任务进入就绪态，OS_EventTaskRdy()")。

注意：关中断的时间与等待消息队列消息的任务数目成比例。

L11.6(7)　如果不是广播方式，则只有等待消息的任务中优先级最高的任务能够进入就绪态。然后，OS_EventTaskRdy()从等待列表中把等待消息的任务中优先级最高的任务删除。

L11.6(8)　调用函数 OS_Sched()进行任务调度。如果上面取出的任务是进入就绪态的任务中优先级最高的任务，而且 OSQPost()函数是由任务调用的，就执行任务切换，该最高优先级的任务运行；如果不是，则 OS_Sched()函数直接返回，调用 OSQPost()函数的任务继续运行。

L11.6(9)　如果没有任务等待该消息队列中的消息，而且此时消息队列未满，则指向该消息的指针被插入到消息队列中；如果消息队列已满，函数就会返回一个出

错代码。

L11.6(10) OSQPostOpt()检查 opt 参数的设置，以确定调用本函数的任务要求按先进先出还是后进先出(设置 OS_POST_OPT_FRONT)来处理消息。

L11.6(11) 如果选择的是后进先出，则 OSQPostOpt()函数仿真 OSQPostFront()的功能。

L11.6(12) 如果选择的是先进先出，则 OSQPostOpt()函数仿真 OSQPost()的功能。

L11.6(13) 无论哪种情况，队列中的消息数都增加1。

如果 OSQPostOpt()函数是由中断服务子程序调用的，那么不会发生任务切换。一直要等到中断嵌套的最外层中断服务子程序调用 OSIntExit()函数时，才能进行任务切换(见3.10节："μC/OS-II 中的中断")。

11.06 无等待地从消息队列中获得消息，OSQAccept()

如果试图从消息队列中获得一则消息，而此时消息队列为空，又不想让任务因得不到消息而被挂起，则可以通过调用 OSQAccept()函数来实现。程序清单 L11.7 列出了这个函数的源代码。

程序清单 L11.7 无等待地从消息队列中获得消息，OSQAccept()

```
void   *OSQAccept (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    void         *msg;
    OS_Q         *pq;

#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                    (1)
        return ((void *)0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                     (2)
        return ((void *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {                                         (3)
```

```
        msg = *pq->OSQOut++;                                    (4)
        pq->OSQEntries--;                                       (5)
        if (pq->OSQOut == pq->OSQEnd) {                         (6)
            pq->OSQOut = pq->OSQStart;
        }
    } else {
        msg = (void *)0;                                        (7)
    }
    OS_EXIT_CRITICAL();
    return (msg);
}
```

L11.7(1)

L11.7(2)　如果 OS_CFG.H 文件中的 OS_ARG_CHK_EN 置为 1，则 OSQAccept()函数首先验证 pevent 不是空指针，而是指向由 OSQCreate()函数建立的事件控制块。

L11.7(3)　OSQAccept()通过查看队列控制模块中的 .OSQEntries 域，来确定队列中是否有消息。

L11.7(4)

L11.7(5)　如果有消息，则队列中最早进入消息队列的消息被取出(先进先出)，并被复制到局部指针变量 msg 中。同时，将队列中的消息数减 1。

L11.7(6)　OSQAccept()让 .OSQOut 指针指向下一个缓冲区，循环队列的指针就这样被调整了。

L11.7(7)　如果队列中没有消息，那么局部变量指针就被置为 NULL。

调用 OSQAccept() 函数的代码应检查 OSQAccept()函数的返回值。如果返回的是空指针 NULL，则说明消息队列是空的，没有消息。空指针对应用程序没有用，因为按照约定，空指针是无效指针。非空指针说明，已经从消息队列中成功地获得了一则消息。中断服务子程序可以调用 OSQAccept()。

11.07　清空消息队列，OSQFlush()

OSQFlush()函数允许清空一个消息队列中的所有消息，以重新开始使用。程序清单 L11.8 是该函数的源代码。

程序清单 L11.8　清空消息队列

```
INT8U  OSQFlush(OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
```

```
    OS_CPU_SR    cpu_sr;
#endif
    OS_Q         *pq;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                          (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                           (2)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    pq                 = (OS_Q *)pevent->OSEventPtr;                        (3)
    pq->OSQIn          = pq->OSQStart;
    pq->OSQOut         = pq->OSQStart;
    pq->OSQEntries     = 0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L11.8(1)

L11.8(2) 如果 OS_CFG.H 文件中的 OS_ARG_CHK_EN 置为 1，则 OSQFlush() 函数首先验证 pevent 不是空指针，pevent 指向的事件控制块 ECB 已经由 OSQCreate() 函数建立。

L11.8(3) 将队列的插入指针 IN 和取出指针 OUT 复位，使它们都指向消息队列起始端，同时，将队列中的消息数 .OSQEntries 清 0。这里，没有必要检查该消息队列的等待任务列表是否为空，这样做只会占用更多的处理时间。换句话说，只要有等待队列消息的任务，.OSQEntries 就一定是 0。惟一不同的是，指针 .OSQIn 和 .OSQOut 此时可以指向消息队列数组中的任何缓冲区，不一定是起始处。也没有必要用空指针 NULL 填满消息队列。

11.08 获取消息队列的状态，OSQQuery()

OSQQuery() 函数使用户可以查询一个消息队列的当前状态。程序清单 L11.9 是该函数的源代码。

OSQQuery() 需要 2 个参数：一个是指向消息队列的指针 pevent，它是在建立一个消息队列时，由 OSQCreate() 函数返回的；另一个是指向 OS_Q_DATA(见 uCOS_II.H 文件)数据结构的指针 pdata，该数据结构包含了有关消息队列的信息。应用程序在调用 OSQQuery()

函数之前，必须先定义数据结构类型为 OS_Q_DATA 的变量。OS_Q_DATA 数据结构包含下面的几个域：

.OSMsg

如果消息队列中有消息，那么它包含指针 .OSQOut 所指向的队列单元中的内容；如果消息队列是空的，那么 .OSMsg 是一个 NULL 指针。

.OSNMsgs

消息队列中的消息数目(.OSQEntries 的拷贝)。

.OSQSize

消息队列的总容量。

.OSEventTbl[]

.OSEventGrp

消息队列的等待任务列表。通过它们，OSQQuery() 的调用函数可以知道有多少任务在等待该消息队列的消息。

程序清单 L11.9　获取消息队列的状态

```
INT8U  OSQQuery (OS_EVENT *pevent, OS_Q_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    OS_Q         *pq;
    INT8U        *psrc;
    INT8U        *pdest;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                          (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                           (2)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    pdata->OSEventGrp = pevent->OSEventGrp;                                 (3)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
#if OS_EVENT_TBL_SIZE > 0
    *pdest++          = *psrc++;
#endif
```

```
    #if OS_EVENT_TBL_SIZE > 1
        *pdest++          = *psrc++;
    #endif

    #if OS_EVENT_TBL_SIZE > 2
        *pdest++          = *psrc++;
    #endif

    #if OS_EVENT_TBL_SIZE > 3
        *pdest++          = *psrc++;
    #endif

    #if OS_EVENT_TBL_SIZE > 4
        *pdest++          = *psrc++;
    #endif

    #if OS_EVENT_TBL_SIZE > 5
        *pdest++          = *psrc++;
    #endif

    #if OS_EVENT_TBL_SIZE > 6
        *pdest++          = *psrc++;
    #endif

    #if OS_EVENT_TBL_SIZE > 7
        *pdest            = *psrc;
    #endif
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {                                          (4)
        pdata->OSMsg = *pq->OSQOut;
    } else {
        pdata->OSMsg = (void *)0;
    }
    pdata->OSNMsgs = pq->OSQEntries;                                   (5)
    pdata->OSQSize = pq->OSQSize;                                      (6)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L11.9(1)

L11.9(2) 同前面提到的函数一样，如果OS_CFG.H文件中的OS_ARG_CHK_EN置为1，那么OSQQuery()函数首先确认pevent不是空指针，pevent是指向消

息队列类型的事件控制块 ECB。

L11.9(3)　OSQQuery()函数复制等待任务列表。

注意：为了获得更优的性能，笔者在这里使用条件编译，生成相应数目的指令行，而没有使用循环语句。

L11.9(4)　如果消息队列中有消息，则提取出（并不是删除）队列中最早进入队列的消息，并将其复制到 .OSMsg 中。也就是说，OSQQuery()并没有改动 .OSQOut 指针。如果队列中没有消息，那么 .OSMsg 就会包含一个空指针。

L11.9(5)

L11.9(6)　消息队列中的消息数和消息队列的容量大小被分别放置在 .OSNMsgs 中和 OS_Q_DATA 结构的 .OSQSize 域中。

11.09　使用消息队列读取模拟量的值

在控制类应用中，经常要频繁地读取模拟量的值。这时，可先建立一个名为OSTimeDly()的任务（见 5.00 节："任务延时函数，OSTimeDly()"），并且给出希望的采样周期。

如图 F11.5 所示，可以用一个带有超时时限的消息队列来实现 A/D 采样。超时时限就是采样周期。当没有其他任务向该消息队列发送消息时，采样任务因为等待超时而退出等待状态，并恢复运行。这就模仿了 OSTimeDly()函数的功能。

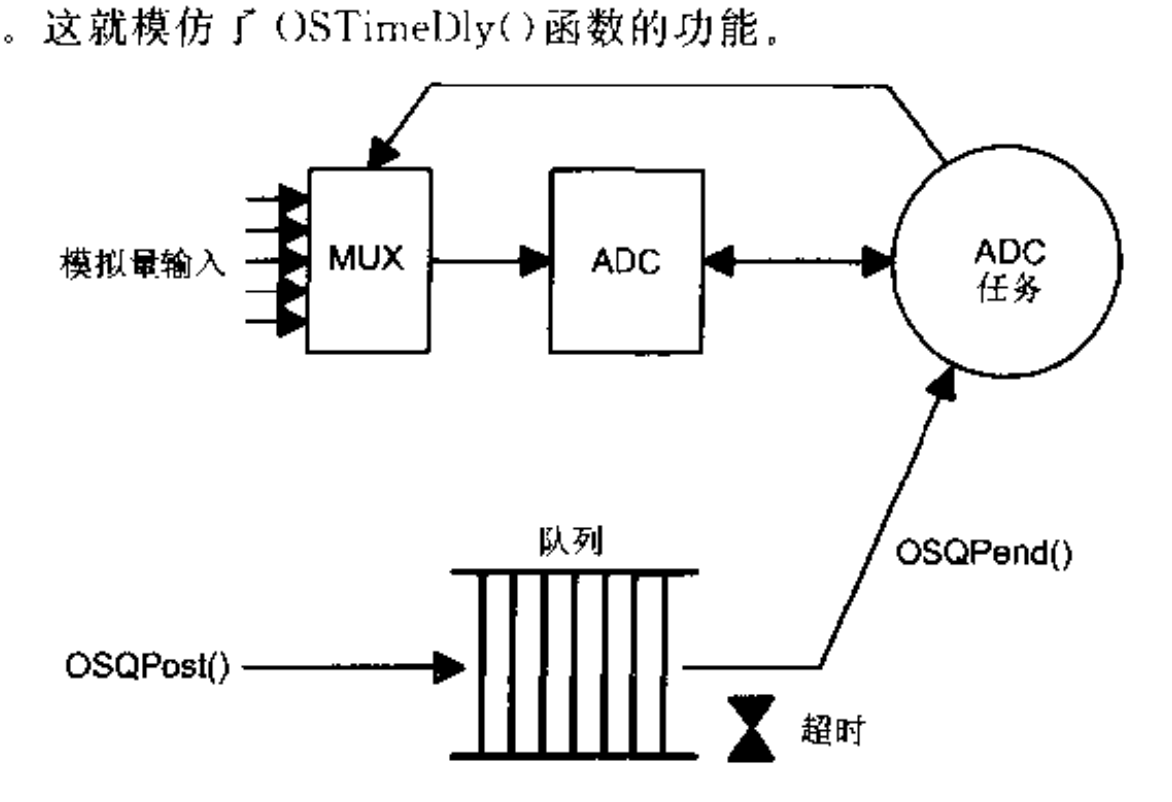


图 F11.5　读取模拟量输入

也许会有疑问，既然 OSTimeDly()函数能完成这项工作，为什么还要使用消息队列呢？这是因为，借助消息队列，可以让其他任务向消息队列发送消息，以终止 A/D 采样任务的等待状态，使其马上执行一次 A/D 采样。此外，还可以通过消息队列通知 A/D 采样程序，具体对哪个通道进行采样，告诉它增加采样频率等等，从而使得应用更加智能化。换句话说，可以告诉 A/D 采样程序："现在马上读取通道 3 的输入值！"，然后该采样任务将重新开始在消息队列中的等待，准备开始一次新的扫描过程。

11.10 消息队列用做计数型信号量

在消息队列初始化时，可以将消息队列中的多个指针设为非 NULL 值［(void *1)］，实现计数型信号量的功能。这里，初始化为非 NULL 值的指针数就是可用的资源数。系统中的任务可以通过 OSQPend()来请求信号量，然后通过调用 OSQPost()来释放信号量，如程序清单 L11.10 所列。如果系统中只使用了计数型信号量和消息队列，那么使用这种方法可以有效地节省代码空间（然后就无需信号量函数了）。这时将 OS_SEM_EN 设为 0，就可以不使用信号量了，而只使用消息队列。值得注意的是，这种方法消耗的是消息队列指针指向的数据类型的变量，原来可以用信号量来管理的每个共享资源都需要占用一个队列控制块。也就是说，为了节省代码空间，而牺牲了 RAM 空间。另外，对消息队列的操作比对信号量的操作要慢。因此，当用计数型信号量管理的共享资源很多时，这种方法的效率是非常低的（因为需要一个大的指针数组）。

程序清单 L11.10　消息队列用做计数型信号量

```
OS_EVENT *QSem;
void     *QMsgTbl[N_RESOURCES]


void main (void)
{
    OSInit();
    .
    .
    QSem = OSQCreate(&QMsgTbl[0], N_RESOURCES);
    for (i = 0; i < N_RESOURCES; i++) {
        OSQPost(QSem, (void *)1);
    }
    .
    .
    OSTaskCreate(Task1, .., .., ..);
    .
    .
    OSStart();
}


void Task1 (void *pdata)
{
    INT8U err;
```

```
    for (;;) {
        OSQPend(&QSem, 0, &err);        /* 申请使用共享资源 */
        .
        .    /* 任务得到信号量,开始使用共享资源          */
        .
        OSMQPost(QSem, (void *)1);      /* 释放共享资源     */
    }
}
```

第 12 章

内存管理

在 ANSI C 中，可以用 malloc() 和 free() 2 个函数动态地分配内存和释放内存；但是，在嵌入式实时操作系统中，调用 malloc() 和 free() 却是危险的，因为多次调用这 2 个函数，会把原来很大的一块连续内存区域逐渐地分割成许多非常小而且彼此又不相邻的内存块，也就是内存碎片。由于这些碎片的大量存在，使得程序到后来连一段非常小的连续内存也分配不到。另外，由于内存管理算法上的原因，malloc() 和 free() 函数的执行时间是不确定的。

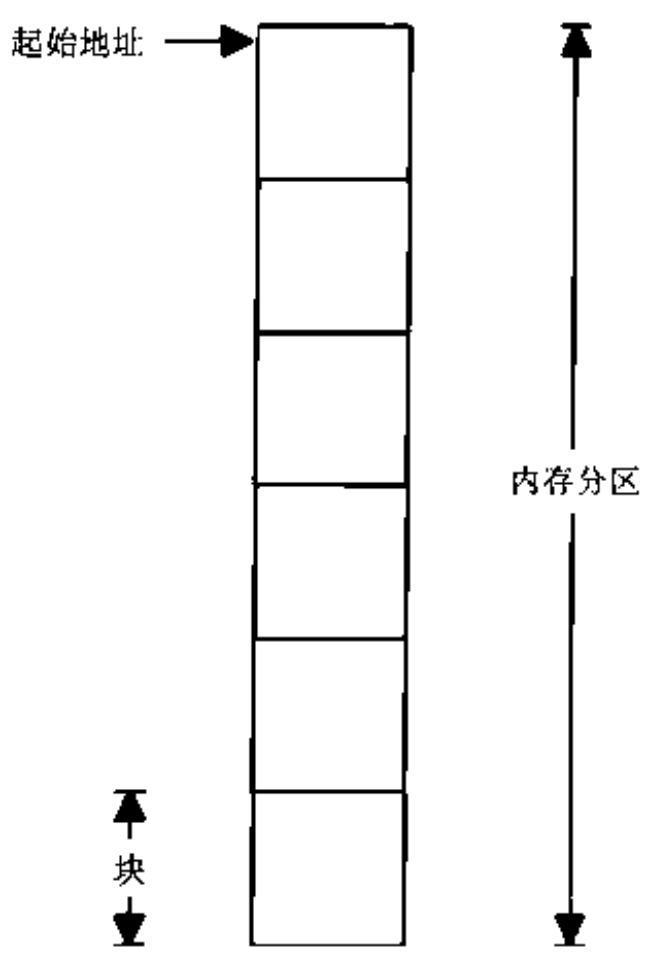


图 F12.1　内存分区

在 μC/OS-II 中，操作系统把连续的大块内存按分区来管理。每个分区中包含整数个大小相同的内存块，如图 F12.1 所示。利用这种机制，μC/OS-II对 malloc()和 free()函数进行了改进，使得它们可以得到和释放固定大小的内存块。这样，malloc()和 free()函数的执行时间就是确定的了。

如图 F12.2 所示，在一个系统中可以有多个内存分区，这样，应用程序就可以从不同的内存分区中得到不同大小的内存块；但是，特定的内存块在释放时，必须重新放回到它以前所属于的内存分区。显然，采用这样的内存管理算法，上面的内存碎片问题就得到了解决。

在调用 μC/OS-II 的相关内存管理函数之前，必须在 OS_CFG.H 文件中设置相应的配置常数。从表 T12.1 中可以看出，OSMemQuery() 函数的编译与否，是由 OS_CFG.H 文件中的配置常数 OS_MEM_QUERY_EN 决定的。应当注意到，在开关量 OS_MEM_EN 被设为 0 的时候，所有的内存管理函数都是禁止的。为了调用表 T12.1 中的功能函数，只需将相应的配置常数置为 1 就可以了。OSMemCreate()、OSMemGet() 及 OSMemPut()这 3 个函数并不像其他函数那样可以被单独禁止，这是因为只要使用 μC/OS-II 内存管理功能函数，就一定会用到这些函数。

表 T12.1　OS_CFG.H 与内存管理相关的配置常数

μC/OS-II 中的内存管理函数	在 OS_CFG.H 中，配置常数为 1，表示允许使用
OSMemCreate()	
OSMemGet()	
OSMemPut()	
OSMemQuery()	OS_MEM_QUERY_EN

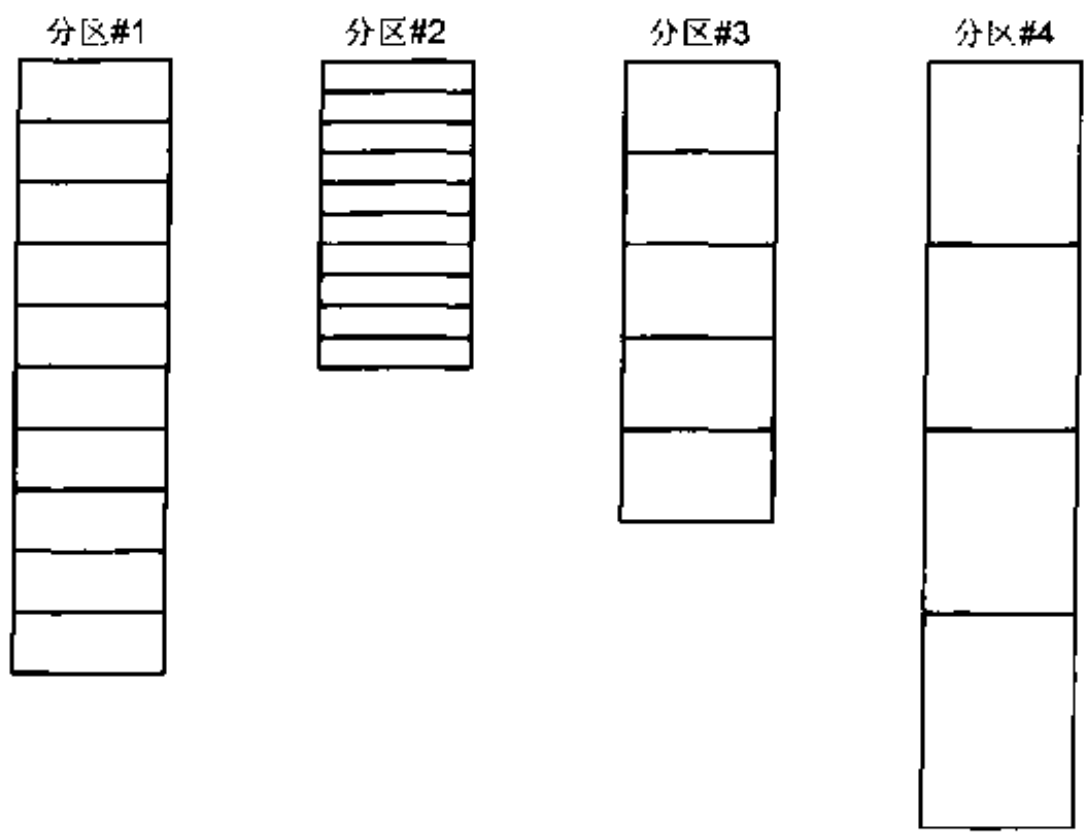


图 F12.2　多个内存分区

12.00　内存控制块

为了便于内存的管理，在 μC/OS-II 中使用内存控制块(memory control blocks)的数据结构跟踪每一个内存分区，系统中的每个内存分区都有它自己的内存控制块。程序清单 L12.1是内存控制块的定义。

程序清单 L12.1　内存控制块的数据结构

```
typedef struct {
    void     *OSMemAddr;
    void     *OSMemFreeList;
    INT32U    OSMemBlkSize;
    INT32U    OSMemNBlks;
    INT32U    OSMemNFree;
} OS_MEM;
```

.OSMemAddr

指向内存分区起始地址的指针。它在建立内存分区(见 12.01 节："建立一个内存分区，OSMemCreate()")时初始化，在此之后就不能更改了。

.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，具体含义须根据该内存分区是否已经建立来决定(见 12.01 节)。

.OSMemBlkSize

内存分区中内存块的大小，是建立该内存分区时定义的(见 12.01 节)。

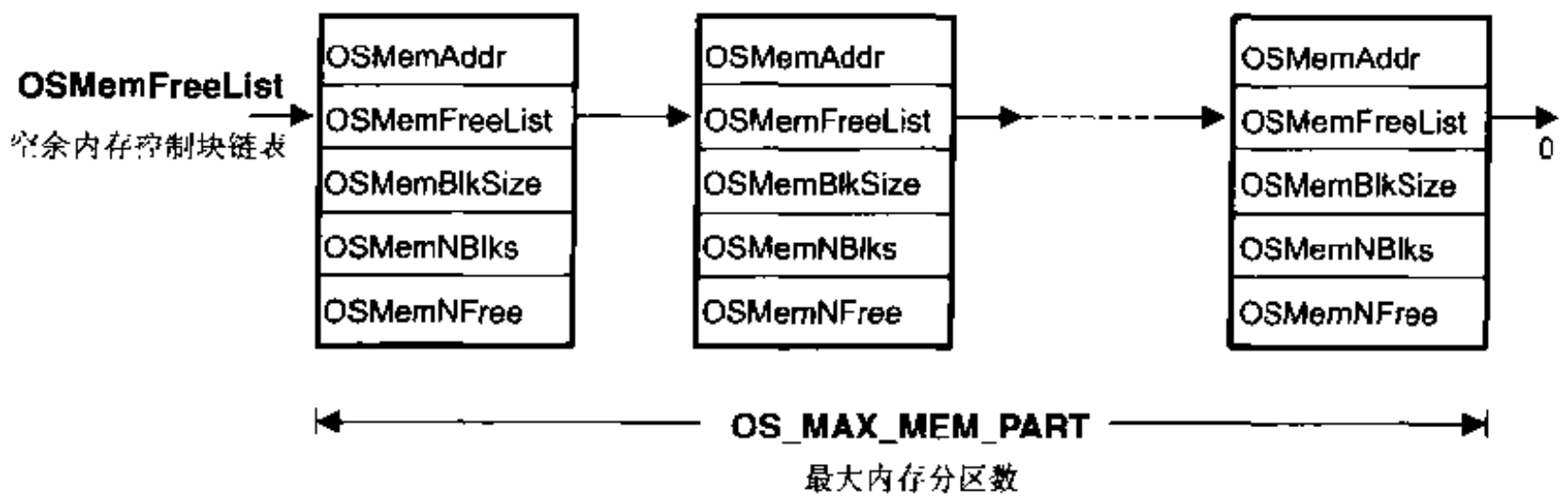
.OSMemNBlks

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(见12.01节)。

.OSMemNFree

内存分区中当前可以获得的空余内存块数量。

若要在μC/OS-II中使用内存管理，则需要在OS_CFG.H文件中将开关量OS_MEM_EN设置为1。这样μC/OS-II在启动时就会对内存管理器进行初始化(由OSInit()调用OSMemInit()函数来实现)。该初始化主要建立一个图F12.3所示的内存控制块链表，其中的常数OS_MAX_MEM_PART(见OS_CFG.H)定义了最大的内存分区数，该常数值至少应为2。



图F12.3 空余内存控制块链表

从图中可以看出，内存控制块的域OSMemFreeList的作用是，将空余的内存控制块链接成内存控制块链表。

12.01 建立一个内存分区，OSMemCreate()

在使用一个内存分区之前，必须先建立该内存分区。这个操作可以通过调用函数OSMemCreate()来完成。程序清单L12.2说明了如何建立一个含有100个内存块并且每个内存块大小为32 B的内存分区。

程序清单L12.2 建立一个内存分区

```
OS_MEM *CommTxBuf;
INT8U  CommTxPart[100][32];


void main (void)
{
    INT8U err;
```

```
    OSInit();
    .
    .
    CommTxBuf = OSMemCreate(CommTxPart, 100, 32, &err);
    .
    .
    OSStart();
}
```

程序清单 L12.3 是 OSMemCreate()函数的源代码。该函数共有 4 个参数:内存分区的起始地址、分区内的内存块总数、每个内存块的字节数及一个指向出错信息代码的指针。如果 OSMemCreate()操作失败,它将返回一个 NULL 指针;否则,它将返回一个指向内存控制块的指针。对内存管理的其他操作,像 OSMemGet(),OSMemPut()及 OSMemQuery()函数等(见 12.02～12.04 节),都需通过该指针进行。

程序清单 L12.3　OSMemCreate()

```
OS_MEM *OSMemCreate (void *addr, INT32U nblks, INT32U blksize, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_MEM      *pmem;
    INT8U       *pblk;
    void        **plink;
    INT32U       i;


#if OS_ARG_CHK_EN > 0
    if (addr == (void *)0) {                                          (1)
        *err = OS_MEM_INVALID_ADDR;
        return ((OS_MEM *)0);
    }
    if (nblks < 2) {                                                  (2)
        *err = OS_MEM_INVALID_BLKS;
        return ((OS_MEM *)0);
    }
    if (blksize < sizeof(void *)) {                                   (3)
        *err = OS_MEM_INVALID_SIZE;
        return ((OS_MEM *)0);
    }
#endif
    OS_ENTER_CRITICAL();
```

```
    pmem = OSMemFreeList;                                                    (4)
    if (OSMemFreeList != (OS_MEM *)0) {
        OSMemFreeList = (OS_MEM *)OSMemFreeList->OSMemFreeList;
    }
    OS_EXIT_CRITICAL();
    if (pmem == (OS_MEM *)0) {                                               (5)
        *err = OS_MEM_INVALID_PART;
        return ((OS_MEM *)0);
    }
    plink = (void **)addr;                                                   (6)
    pblk  = (INT8U *)addr + blksize;
    for (i = 0; i < (nblks - 1); i++) {
        *plink = (void *)pblk;
        plink = (void **)pblk;
        pblk  = pblk + blksize;
    }
    *plink              = (void *)0;
    pmem->OSMemAddr     = addr;                                              (7)
    pmem->OSMemFreeList = addr;
    pmem->OSMemNFree    = nblks;
    pmem->OSMemNBlks    = nblks;
    pmem->OSMemBlkSize  = blksize;
    *err                = OS_NO_ERR;
    return (pmem);                                                           (8)
}
```

L12.3(1) 定义的内存分区的起始地址必须是有效的。

L12.3(2) 每个内存分区必须含有至少2个内存块。

L12.3(3) 每个内存块至少须容得下一个指针，因为同一分区中的所有空余内存块是由指针链串联起来的。

L12.3(4) OSMemCreate()从系统中的空余内存控制块链表中取得一个内存控制块。该内存控制块包含相应内存分区的运行状态信息(run-time information)。

L12.3(5) OSMemCreate()必须在有空余内存控制块可用的情况下，才能建立一个内存分区。

L12.3(6) 当上述条件均得到满足时，所要建立的内存分区内的所有内存块链接成一个单向链表。这里之所以用单向链表，是因为在单向链表中插入和删除元素都是从链表的顶端开始向下执行的。

L12.3(7) 在对应的内存控制块中填写与该内存分区相关的信息。

L12.3(8) 完成上述各操作后，OSMemCreate()返回指向该内存控制块的指针。该指针在以后对该内存分区的操作中使用。

图 F12.4 是 OSMemCreate()函数完成后，内存控制块及对应的内存分区与分区内的内存块之间的关系。

在程序运行期间，经过多次的内存分配和释放后，同一分区内的各内存块之间的链接顺序会发生很大的变化。

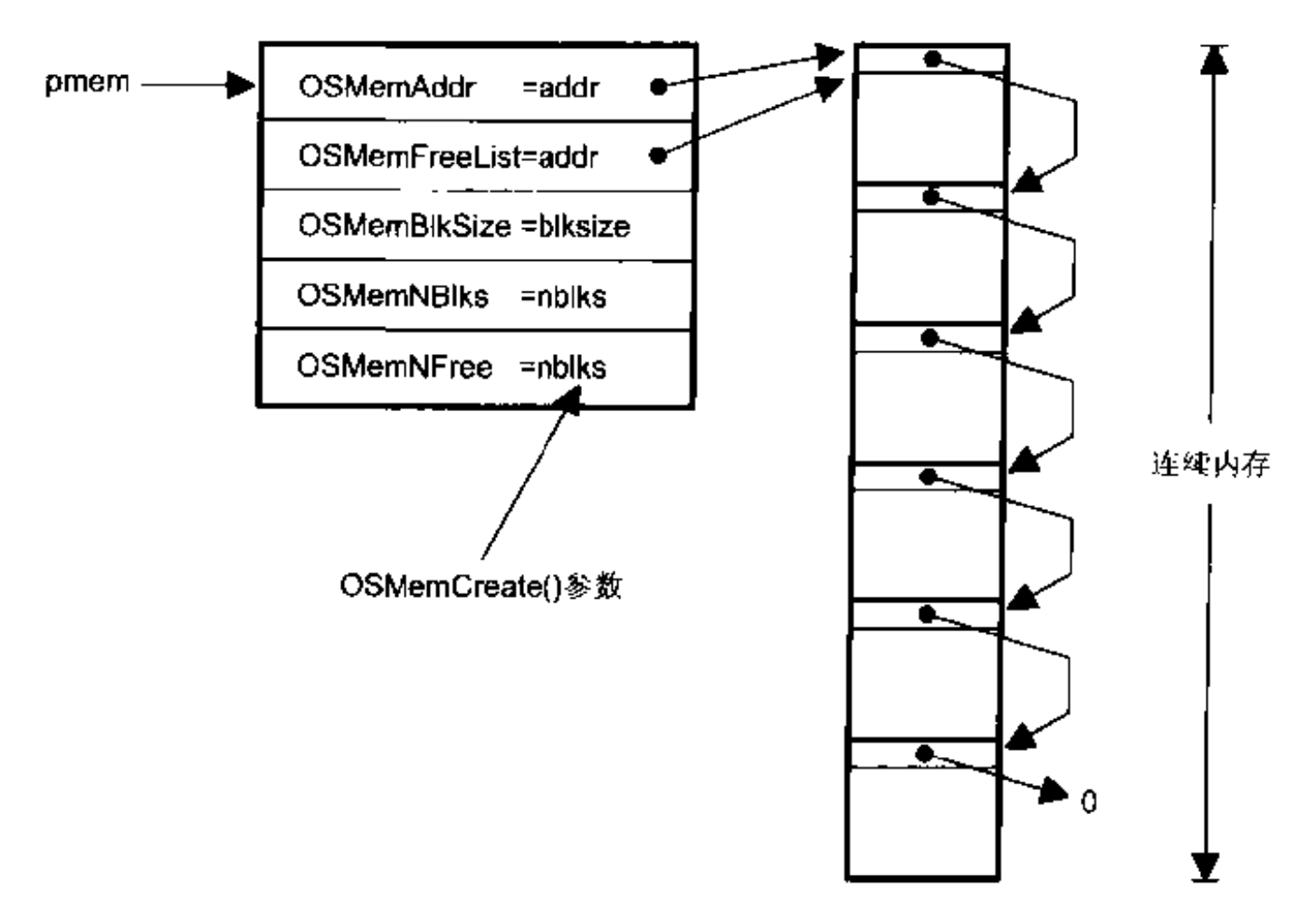


图 F12.4　由 OSMemCreate()函数建立的内存分区

12.02　分配一个内存块，OSMemGet()

应用程序可以调用 OSMemGet()函数，从已经建立的内存分区中申请一个内存块。该函数的惟一参数是指向特定内存分区的指针。该指针是在建立内存分区时，由 OSMemCreate()函数返回的。显然，应用程序必须知道内存块的大小，并且在使用时不能超过其容量。例如，如果一个内存分区内的内存块为 32 B，那么，应用程序最多只能使用该内存块中的 32 B。当应用程序不再使用这个内存块后，必须及时将其释放，重新放回到相应的内存分区中（参见 12.03 节："释放一个内存块，OSMemPut()"）。程序清单 L12.4 是 OSMemGet()函数的源代码。

程序清单 L12.4　OSMemGet()

```
void   *OSMemGet (OS_MEM *pmem, INT8U *err)                    (1)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    void         *pblk;
```

```
#if OS_ARG_CHK_EN > 0
    if (pmem == (OS_MEM *)0) {                                          (2)
        *err = OS_MEM_INVALID_PMEM;
        return ((OS_MEM *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree > 0) {                                         (3)
        pblk                = pmem->OSMemFreeList;                      (4)
        pmem->OSMemFreeList = *(void **)pblk;                           (5)
        pmem->OSMemNFree--;                                             (6)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (pblk);                                                  (7)
    }
    OS_EXIT_CRITICAL();
    *err = OS_MEM_NO_FREE_BLKS;
    return ((void *)0);
}
```

L12.4(1) 传递给 OSMemGet()函数的参数中的指针 pmem，它指向希望从中分配到内存块的内存分区。

L12.4(2) 如果允许参数检验(OS_CFG.H 文件中的配置常量 OS_ARG_CHK_EN 应设置为 1)，那么 OSMemGet()函数须确保代入的指针 pmem 不是空指针。但遗憾的是，OSMemGet()却无法知道非空指针是否一定指向了一个合法的内存分区(因为 pmem 可以指向任何地址)。

L12.4(3) OSMemGet()要检查内存分区中是否有空余的内存块。

L12.4(4) 如果有空余的内存块，那么其中第 1 个内存块会从空余内存块链表中被删除。

L12.4(5)

L12.4(6) 空余内存块链表会被更新，包括将链表头指针后移 1 个元素和将空余内存块数减 1。

L12.4(7) 将指向已分配的内存块的指针返回给用户程序。

注意：可以在中断服务子程序中调用 OSMemGet()，因为在暂时没有内存块可用的情况下，OSMemGet()不会等待，而是立即返回 NULL 指针。

12.03　释放一个内存块，OSMemPut()

当应用程序不再使用某一个内存块时，必须及时地把它释放，并放回到相应的内存分区中。这个操作由 OSMemPut()函数完成。必须注意的是，OSMemPut()并不知道该内存块是属于哪个内存分区的。也就是说，如果用户程序从一个包含 32 B 内存块的分区中分配了一个内存块，那么用完后，千万不能把它返还给一个包含 120 B 内存块的内存分区。因为，当应用程序下一次申请 120 B 分区中的一个内存块时，它会只得到 32 B 的可用空间，其它 88 B 属于其他的任务，这就有可能使系统崩溃。

程序清单 L12.5 是 OSMemPut()函数的源代码。

程序清单 L12.5　OSMemPut()

```
INT8U  OSMemPut (OS_MEM   *pmem, void *pblk)                          (1)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


#if OS_ARG_CHK_EN > 0
    if (pmem == (OS_MEM *)0) {                                        (2)
        return (OS_MEM_INVALID_PMEM);
    }
    if (pblk == (void *)0) {
        return (OS_MEM_INVALID_PBLK);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree >= pmem->OSMemNBlks) {                       (3)
        OS_EXIT_CRITICAL();
        return (OS_MEM_FULL);
    }
    *(void **)pblk      = pmem->OSMemFreeList;                        (4)
    pmem->OSMemFreeList = pblk;
    pmem->OSMemNFree++;                                               (5)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

L12.5(1)　传给 OSMemPut()的第 1 个参数 pmem 是指向内存控制块的，pblk 指向由

pmem 管理的内存块。

L12.5(2) OSMemPut()函数必须确保传递给它的参数指针是非空的;但是遗憾的是,OSMemPut()并不知道要释放的内存块是否属于此内存分区。因此,应用程序必须保证把内存块释放到合理的内存分区中。

L12.5(3) OSMemPut()须检查内存分区是否已满。如果已满,则说明系统在分配和释放内存时出现了错误。

L12.5(4) 如果未满,则要释放的内存块被插入到该分区的空余内存块链表中。

L12.5(5) 将分区中空余内存块总数加1。

12.04 查询一个内存分区的状态,OSMemQuery()

在 μC/OS-II 中,可以使用 OSMemQuery()函数来查询一个特定内存分区的有关信息。通过该函数可以知道特定内存分区中内存块的大小、可用内存块数目及已经使用了的内存块数目等信息。所有这些信息都存放在一个叫 OS_MEM_DATA 的数据结构中,如程序清单 L12.6 所列。程序清单 L12.7 是 OSMemQuery()函数的源代码。

程序清单 L12.6 OS_MEM_DATA 数据结构

```
typedef struct {
    void    *OSAddr;        /* 指向内存分区首地址的指针 */
    void    *OSFreeList;    /* 指向空余内存块链表首地址的指针 */
    INT32U  OSBlkSize;      /* 每个内存块所含的字节数 */
    INT32U  OSNBlks;        /* 内存分区总的内存块数 */
    INT32U  OSNFree;        /* 空余内存块总数 */
    INT32U  OSNUsed;        /* 正在使用的内存块总数 */
} OS_MEM_DATA;
```

程序清单 L12.7 OSMemQuery()

```
INT8U  OSMemQuery (OS_MEM *pmem, OS_MEM_DATA *pdata)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif

# if OS_ARG_CHK_EN > 0
    if (pmem == (OS_MEM *)0) {                                          (1)
        return (OS_MEM_INVALID_PMEM);
    }
    if (pdata == (OS_MEM_DATA *)0) {
        return (OS_MEM_INVALID_PDATA);
```

```
    }
#endif
    OS_ENTER_CRITICAL();
    pdata->OSAddr      = pmem->OSMemAddr;                                  (2)
    pdata->OSFreeList  = pmem->OSMemFreeList;
    pdata->OSBlkSize   = pmem->OSMemBlkSize;
    pdata->OSNBlks     = pmem->OSMemNBlks;
    pdata->OSNFree     = pmem->OSMemNFree;
    OS_EXIT_CRITICAL();
    pdata->OSNUsed     = pdata->OSNBlks - pdata->OSNFree;                  (3)
    return (OS_NO_ERR);
}
```

L12.7(1)　与通常一样，首先须确保传递给函数的参数不是空指针。

L12.7(2)　将指定内存分区的信息全部复制到 OS_MEM_DATA 定义的变量的相应域中。在此之前，代码首先禁止了外部中断，以防止复制过程中某些变量值被修改。

L12.7(3)　由于正在使用的内存块数是由 OS_MEM_DATA 中的局部变量计算得到的，因此，这部分代码可以放在临界段代码之外。

12.05　使用内存分区

图 F12.5 是一个使用 μC/OS II 中的动态分配内存功能并利用它进行消息传递（见第 11 章："消息队列管理"）的例子。程序清单 L12.8 是这个例子中 2 个任务执行过程的示意性代码，其中一些重要代码的标号与图 F12.5 中括号内用数字标志的操作是相对应的。

第 1 个任务读取并检查模拟量输入的值（如气压、温度及电压等）。如果该值超过了一定的阈值，就向第 2 个任务发送一则消息。该消息中含有出错时的时间信息、出错的通道号以及其他出错代码等可以想到的任何可能的信息。

在这个例子中，对错误做集中处理，即任何任务和中断服务子程序都可以向该任务发送出错消息。错误处理程序则负责在显示设备上显示出错信息，在磁盘上登记出错记录或者启动另一个任务对错误进行纠正等。

程序清单 L12.8　内存分配的例子 —— 扫描模拟量输入和报告出错

```
AnalogInputTask()
{
    for (;;) {
        for (所有的模拟量输入) {
            读入模拟量输入值;                                              (1)
            if (模拟量超过阈值) {
```

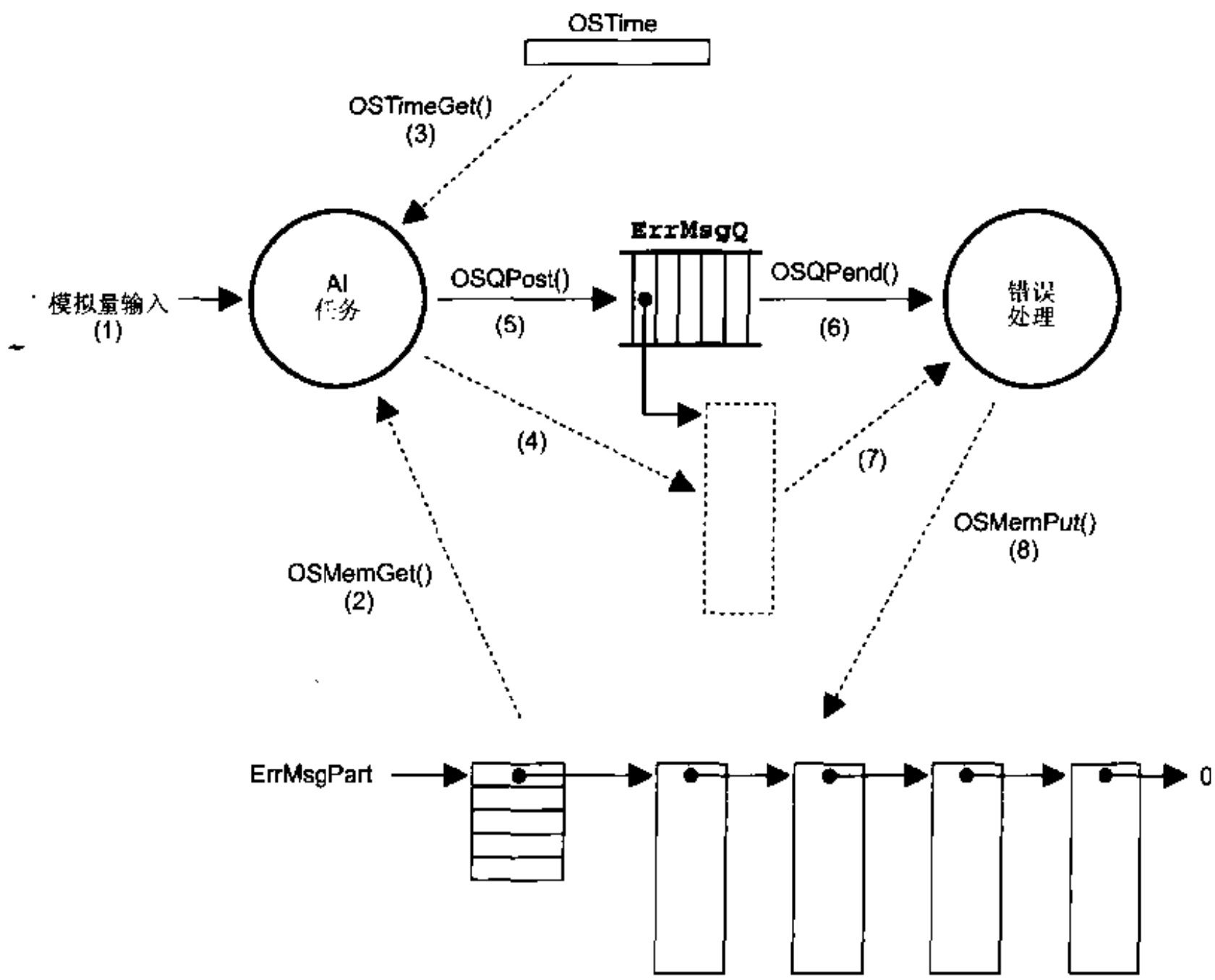


图 F12.5 使用动态内存分配

```
            得到一个内存块；                                                    (2)
            得到当前系统时间(以时钟节拍为单位)；                                (3)
            将下列各项存入内存块；                                              (4)
                系统时间(时间戳)；
                超过阀值的通道号；
                错误代码；
                错误等级；
                等等.
            向错误处理队列发送出错消息；                                        (5)
                (一个指向包含上述各项的内存块的指针)
        }
    }
    延时任务，直到再次对模拟量进行采样时为止；
  }
}
```

```
ErrorHandlerTask()
{
    for (;;) {
        等待错误处理队列的消息；                                    (6)
            (得到指向包含有关错误数据的内存块的指针)
        读入消息，并根据消息的内容执行相应的操作；                  (7)
        将内存块放回到相应的内存分区中；                            (8)
    }
}
```

12.06　等待内存分区中的一个内存块

有时，在内存分区暂时没有可用的空余内存块的情况下，让一个申请内存块的任务等待也是可行的；但是，μC/OS-II 本身在内存管理上并不支持这项功能。如果确实需要，则可以通过为特定内存分区增加计数型信号量的方法，实现这种功能（见第 7 章："信号量管理"）。应用程序为了申请分配内存块，首先要得到相应的信号量，然后才能调用 OSMemGet()函数。如果须释放内存块，只要将内存块释放到相应的内存分区中，并发送一个信号量即可。整个过程见程序清单 L12.9。

程序清单 L12.9　等待从一个内存分区中分配内存块

```
OS_EVENT    *SemaphorePtr;                                          (1)
OS_MEM      *PartitionPtr;
INT8U        Partition[100][32];
OS_STK       TaskStk[1000];


void main (void)
{
    INT8U err;

    OSInit();                                                       (2)
    .
    .
    SemaphorePtr = OSSemCreate(100);                                (3)
    PartitionPtr = OSMemCreate(Partition, 100, 32, &err);           (4)
    .
    OSTaskCreate(Task, (void *)0, &TaskStk[999], &err);             (5)
    .
    OSStart();                                                      (6)
}
```

```
void Task (void *pdata)
{
    INT8U   err;
    INT8U  *pblock;

    for (;;) {
        OSSemPend(SemaphorePtr, 0, &err);                          (7)
        pblock = OSMemGet(PartitionPtr, &err);                     (8)
        .
        .   /*使用内存块*/
        .
        OSMemPut(PartitionPtr, pblock);                            (9)
        OSSemPost(SemaphorePtr);                                   (10)
    }
}
```

L12.9(1) 程序代码首先定义了程序中使用到的各个变量。该例中,直接用数字定义了各个变量的值。实际应用中,建议将这些数值定义成常数。

L12.9(2)

L12.9(3) 当系统复位时,μC/OS-II 调用 OSInit()进行系统初始化,然后用内存分区中总的内存块数初始化并建立一个信号量。

L12.9(4) 紧接着建立内存分区和相应的要访问该分区的任务。

L12.9(5) 至此,已经清楚如何增加其他的任务了。显然,如果系统中只有一个任务使用动态内存块,就没有必要使用信号量了。这种情况不需要保证内存资源的互斥。事实上,除非要实现多任务共享内存,否则连内存分区都不需要。

L12.9(6) 调用 OSStart(),启动多任务。

L12.9(7)

L12.9(8) 当一个任务运行时,只有在信号量有效时,才有可能得到内存块。一旦信号量有效了,就可以申请内存块并使用它,而没有必要对 OSSemPend()返回的错误代码进行检查。因为在这里,只有当一个内存块被其它任务释放并放回到内存分区后,μC/OS-II 才会返回到该任务,使其继续执行。同理,对 OSMemGet()返回的错误代码也无须做进一步的检查——为了使任务能够得到内存块并继续执行,在内存分区中至少要有一个内存块是可用的。

L12.9(9)

L12.9(10) 当一个任务不再使用某内存块时,只需简单地将它放回到内存分区中,并发出信号量。

第 13 章

移植 μC/OS－II

这一章介绍如何将 μC/OS－II 移植到不同的处理器上。所谓移植，就是使一个实时内核能在其他的微处理器或微控制器上运行。为了方便移植，大部分 μC/OS－II 的代码是用 C 语言编写的；但是仍需要用 C 语言和汇编语言编写一些与处理器硬件相关的代码，这是因为 μC/OS－II 在读/写处理器寄存器时，只能通过汇编语言来实现。由于 μC/OS－II 在设计之初就已经充分考虑了可移植性，所以 μC/OS－II 的移植相对来说是比较容易的。如果已经在使用的处理器上成功地移植了 μC/OS－II，也得到了相关的代码，那就不必阅读这一章了。当然，本章介绍的内容将有助于了解 μC/OS－II 中与处理器相关部分的代码，有助于学习如何在不同的微处理器上移植 μC/OS－II。

要使 μC/OS－II 正常运行，处理器必须满足以下要求：

- 处理器的 C 编译器能产生可重入型代码；
- 处理器支持中断，并且能产生定时中断(通常为 10～100 Hz)；
- 用 C 语言就可以开/关中断；
- 处理器能支持一定数量的数据存储硬件堆栈(可能是几千字节)；
- 处理器有将堆栈指针以及其他 CPU 寄存器的内容读出、并存储到堆栈或内存中去的指令。

像 Motorola 6805 系列的处理器就不能满足上面的后 2 项要求，所以 μC/OS－II 不能在这类处理器上运行。

图 F13.1 示意 μC/OS－II 的结构以及与硬件的关系。使用 μC/OS－II 时，应提供应用软件和 μC/OS－II 的配置代码。本书和所附的光盘中包含了所有与处理器无关的代码以及基于 Intel 80x86 实模式在大模式下编译的与处理器相关的代码。如果打算在其他处理器上使用 μC/OS－II，最好能找到一个现成的移植范例。如果没有，则只好自己编写了。可以在 μC/OS－II的官方网站 www.uCOS－II.com 中查找移植范例；也可以阅读其他处理器的移植代码，学习他人的移植经验。

如果已经了解处理器和 C 编译器的技术细节，那么移植 μC/OS－II 的工作实际上是非常容易的。

根据处理器的不同，一个移植实例可能需要编写或改写 50～300 行的代码，需要的时间从几个小时到一星期不等。然而，最容易做的还是修改现成的移植代码，该代码与拟选用的 CPU 相同或类似。

表 T13.1 列出的函数等是需要修改的，其中最后的数字表示难易程度(1 表示容易；2 表示适中；3 表示比较复杂)。

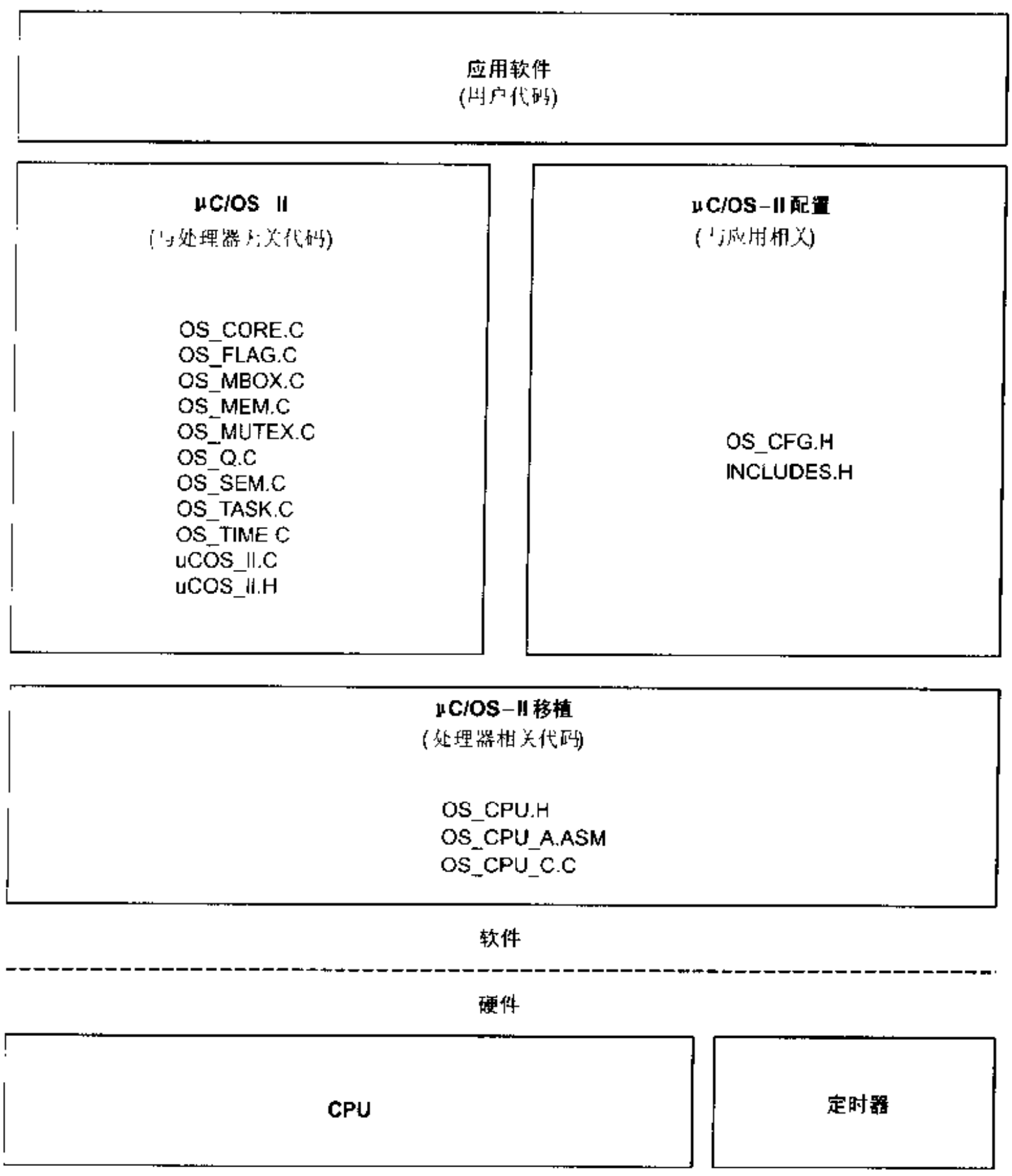


图 F13.1 μC/OS-II 硬件/软件体系结构

一旦代码移植完毕，下一步工作就是测试。测试一个像 μC/OS-II 这样的多任务实时内核其实并不复杂，甚至可以在没有应用程序的情况下测试。换句话说，就是让内核自己测试自己。这样做有2点好处：第1，避免使本来就复杂的事情更加复杂化；第2，如果出现问题，可以知道问题是出在内核代码中，而不是在应用程序中。刚开始时，可以运行一些简单的任务和时钟节拍中断服务子程序。一旦多任务调度成功地运行了，再添加应用程序的任务就非常简单了。

表 13.1　移植需要修改的部分

名　称	类　型	所在文件	C 语言还是汇编？	复杂程度
BOOLEAN	数据类型	OS_CPU.H	C 语言	1
INT8U	数据类型	OS_CPU.H	C 语言	1
INT8S	数据类型	OS_CPU.H	C 语言	1
INT16U	数据类型	OS_CPU.H	C 语言	1
INT16S	数据类型	OS_CPU.H	C 语言	1
INT32U	数据类型	OS_CPU.H	C 语言	1
INT32S	数据类型	OS_CPU.H	C 语言	1
FP32	数据类型	OS_CPU.H	C 语言	1
FP64	数据类型	OS_CPU.H	C 语言	1
OS_STK	数据类型	OS_CPU.H	C 语言	2
OS_CPU_SR	数据类型	OS_CPU.H	C 语言	2
OS_CRITICAL_METHOD	宏定义	OS_CPU.H	C 语言	1
OS_STK_GROWTH	宏定义	OS_CPU.H	C 语言	1
OS_ENTER_CRITICAL()	宏	OS_CPU.H	C 语言	3
OS_EXIT_CRITICAL()	宏	OS_CPU.H	C 语言	3
OSStartHighRdy()	函　数	OS_CPU_A.ASM	汇　编	2
OSCtxSw()	函　数	OS_CPU_A.ASM	汇　编	3
OSIntCtxSw()	函　数	OS_CPU_A.ASM	汇　编	3
OSTickISR()	函　数	OS_CPU_A.ASM	汇　编	3
OSTaskStkInit()	函　数	OS_CPU_C.C	C 语言	3
OSInitHookBegin()	函　数	OS_CPU_C.C	C 语言	1
OSInitHookEnd()	函　数	OS_CPU_C.C	C 语言	1
OSTaskCreateHook()	函　数	OS_CPU_C.C	C 语言	1
OSTaskDelHook()	函　数	OS_CPU_C.C	C 语言	1
OSTaskSwHook()	函　数	OS_CPU_C.C	C 语言	1
OSTaskStatHook()	函　数	OS_CPU_C.C	C 语言	1
OSTCBInitHook()	函　数	OS_CPU_C.C	C 语言	1
OSTimeTickHook()	函　数	OS_CPU_C.C	C 语言	1
OSTaskIdleHook()	函　数	OS_CPU_C.C	C 语言	1

13.00 开发工具

如前所述，因为 μC/OS-II 绝大部分是用标准的 ANSI C 编写的，移植 μC/OS-II 需要标准的 C 交叉编译器，并且是针对所使用的 CPU 的；而且因为 μC/OS-II 是一个可剥夺型内核，只能通过 C 编译器来产生可重入型代码。

同时 C 编译器还应支持汇编语言程序，因为有一些对 CPU 寄存器的操作只能通过汇编语言实现。实际上有很多 C 编译器是不允许直接用 C 语言操作（读/写）CPU 内部寄存器的，只能通过插入汇编语句来完成。

绝大部分为嵌入式系统设计的 C 编译器都包括汇编器、链接器及定位器。链接器用来将不同的模块（编译过或汇编过的文件）链接成目标文件；定位器则允许将代码和数据放置在目标处理器的指定内存空间中。

所用的 C 编译器还须提供一种机制，能在 C 语言中开中断和关中断。一些编译器允许在 C 语言源代码中插入汇编语句，这就使得插入相应的处理器指令开中断和关中断变得很容易了。还有一些编译器实际上包括了语言扩展功能，可以直接在 C 语言中开中断和关中断，这就更容易了。

13.01 目录和文件

本书所附的光盘中提供了 μC/OS-II 的安装程序，可在硬盘上安装 μC/OS-II 和移植范例代码（Intel 80x86 实模式，大模式下编译）。笔者设计了一个目录结构，使得用户更容易找到目标处理器的相关文件。如果想增加一个其他处理器的移植范例，则可以考虑采取同样的方法（包括目录的建立和文件的命名等等）。

所有的移植范例都应放在硬盘的 \SOFTWARE\uCOS-II 目录下（可以自己指定硬盘根目录）。微处理器或微控制器的移植源代码一定能在以下 2 个或 3 个文件中找到：OS_CPU.H，OS_CPU_C.C 及 OS_CPU_A.ASM。汇编语言文件 OS_CPU_A.ASM 不一定需要，因为某些 C 编译器允许在 C 语言中直接插入汇编语言语句，所以，可以将所需的汇编语言代码直接放到 OS_CPU_C.C 文件中。

放置移植范例的目录取决于所用的处理器。表 T13.2 说明了放置不同移植范例的目录结构。

注意：各个目录虽然针对完全不同的目标处理器，但都包括了相同的文件名。

例如：使用 Paradigm C 编译器（参见 www.DevTools.com）编译的 μC/OS-II 的移植代码应该放在 PARADIGM 子目录下；使用 Borland C 编译器（4.5 版本）（参见 www.Borland.com）编译的 μC/OS-II 的移植代码应该放在 BC45 子目录下；同样，使用 COSMIC 编译器（参见 www.Cosmic-US.com）编译的 Motorola 公司的 68HC11 处理器的 μC/OS-II 移植代码应该放在 COSMIC 子目录下，如表 T13.2 所列。

表 T13.2　移植范例的目录结构

Intel/AMD 80186	\SOFTWARE\uCOS-II\Ix86L\PARADIGM
	\OS_CPU.H \OS_CPU_A.ASM \OS_CPU_C.C
	\SOFTWARE\uCOS-II\Ix86L\BC45
	\OS_CPU.H \OS_CPU_A.ASM \OS_CPU_C.C
Motorola 68HC11	\SOFTWARE\uCOS-II\68HC11\COSMIC
	\OS_CPU.H \OS_CPU_A.ASM \OS_CPU_C.C

13.02　INCLUDES.H

在第 1 章中曾经提到，INCLUDES.H 是一个主头文件，它出现在每个 .C 文件的第1 行。

```
#include "includes.h"
```

INCLUDES.H 文件使得工程项目中的每个 .C 文件无需分别考虑它实际上需要哪些头文件。使用 INCLUDES.H 文件的惟一缺点是，它可能包含一些与当前要编译的 .C 文件实际上不相干的头文件。这意味着每个文件的编译时间都可能会增加；但由于它增强了代码的可移植性，所以还是决定使用这一方法。

可以通过重新编辑 INCLUDES.H，增加自己的头文件，但是头文件必须添加在头文件列表的最后。虽然 INCLUDES.H 文件与移植的过程没有关系，但是因为每一个 μC/OS-II 的 C 文件都要使用它，所以还是在此说明一下。

13.03　OS_CPU.H

OS_CPU.H 包括了用 #define 语句定义的、与处理器相关的常数、宏以及类型。OS_CPU.H的大体结构如程序清单 L13.1 所列。

程序清单 L13.1 OS_CPU.H

```
/*
*********************************************************************
*                              数据类型
*                          （与编译器有关）
*********************************************************************
*/

typedef unsigned  char  BOOLEAN;                                        (1)
typedef unsigned  char  INT8U;          /* 无符号 8 位整数          */
typedef signed    char  INT8S;          /* 有符号 8 位整数          */
typedef unsigned  int   INT16U;         /* 无符号 16 位整数         */
typedef signed    int   INT16S;         /* 有符号 16 位整数         */
typedef unsigned  long  INT32U;         /* 无符号 32 位整数         */
typedef signed    long  INT32S;         /* 有符号 32 位整数         */
typedef float           FP32;           /* 单精度浮点数             */   (2)
typedef double          FP64;           /* 双精度浮点数             */

typedef unsigned  int   OS_STK;         /* 堆栈入口宽度为 16 位     */   (3)
typedef unsigned  short OS_CPU_SR;      /* 定义 CPU 状态寄存器的宽度为 16 位 */   (4)
*********************************************************************
*                          与处理器有关的代码
*********************************************************************
*/
#define  OS_CRITICAL_METHOD    ??                                       (5)

#if      OS_CRITICAL_METHOD == 1
#define  OS_ENTER_CRITICAL()  ????                                      (6)
#define  OS_EXIT_CRITICAL()   ????
#endif

#if      OS_CRITICAL_METHOD == 2
#define  OS_ENTER_CRITICAL()  ????                                      (7)
#define  OS_EXIT_CRITICAL()   ????
#endif

#if      OS_CRITICAL_METHOD == 3
#define  OS_ENTER_CRITICAL()  ????                                      (8)
#define  OS_EXIT_CRITICAL()   ????
#endif
#define  OS_STK_GROWTH        1   /* 定义堆栈的方向:1=向下递减,0=向上递增 */   (9)

#define  OS_TASK_SW()         ???                                       (10)
```

13.03.01　与编译器相关的数据类型

因为不同的微处理器有不同的字长，所以 μC/OS-II 的移植包括了一系列的数据类型定义，以确保其可移植性。尤其是，μC/OS-II 代码从不使用 C 语言中的 short、int 及 long 等数据类型，因为它们是与编译器相关的，是不可移植的。相反，定义的整型数据结构等既是可移植的，又很直观。

为了完成数据类型的定义，应该参考所使用的 C 编译器的文档，从中找到其使用的数据类型的定义，从而使 μC/OS-II 得到正确的数据类型定义。

L13.1(1)　在此定义了可移植且直观的整型数据类型。举例来说，INT16U 数据类型总是代表 16 位的无符号整型数。这样，μC/OS-II 和应用程序就可以断定，声明为该数据类型变量的范围都是 0～65 535。将 μC/OS-II 移植到 32 位的处理器上，就意味着 INT16U 实际被声明为无符号短整数，而不是无符号整数，但是，μC/OS-II 处理的仍然是 INT16U。用户要做的只是查阅编译器文档，找出对应于 μC/OS-II 的标准 C 的相应数据类型。

L13.1(2)　为方便起见，虽然 μC/OS-II 中并不使用浮点数，但还是定义了浮点数据类型。

L13.1(3)　必须将任务堆栈的数据类型告诉 μC/OS-II。这是通过为 OS_STK 声明恰当的 C 数据类型来实现的。如果处理器的堆栈是 32 位的，那么就应该将 OS_STK声明为：

```
typedef  INT32U  OS_STK;
```

这个例子中定义 OS_STK 按 32 位无符号数 INT32U 类型处理。当建立任务声明任务栈类型时，必须永远使用 OS_STK 作为堆栈的数据类型。

L13.1(4)　如果使用 OS_CRITICAL_METHOD = 3(见下一节)，那么还须声明 CPU 状态寄存器(PSW)的数据类型。PSW 也称为处理器标志或状态寄存器。如果所用 CPU 的 PSW 是 16 位宽度，那么只须做如下声明：

```
typedef  INT16U  OS_CPU_SR;
```

13.03.02　OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()

本节与 3.09 节："临界段，OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()"的内容非常相似，只是增减了一些内容。之所以这样重复说明，是想避免读者前后翻查。与所有的实时内核一样，μC/OS-II 需要先关中断，再处置临界段代码，并且在处置完毕后重新开中断。这就使得 μC/OS-II 能够保护临界段代码免受多任务或中断服务子程序的破坏。

通常每个处理器都会提供一定的汇编指令来开/关中断，C 编译器必须有一定的机制直接从 C 语言中执行这些操作。有些编译器允许在 C 源代码中插入行汇编语句，这就使得插入处理器指令来开中断和关中断变得很容易。有些其他的编译器提供语言扩展功能，可以直接从 C 语言中开/关中断。

为了隐藏编译器厂商提供的不同实现方法，以增加可移植性，μC/OS-II 定义了 2 个宏，用来关/开中断：OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()(参见 L13.1(5)～(8))。

实际上，OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()总是成对使用的，并且分别加在临界段代码的前面和后面，如程序清单 L13.2 所列。

程序清单 L13.2 临界代码的使用

```
{
    .
    .
    .
    OS_ENTER_CRITICAL();
    /* μC/OS-II 临界代码段 */
    OS_EXIT_CRITICAL();
    .
    .
    .
}
```

应用程序可以调用 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()，以保护临界代码；但是，在使用这种方法时需特别小心，因为如果在调用像 OSTimeDly()之类的功能函数(见第 5 章："时间管理")之前就关掉了中断，应用程序就会崩溃。发生这种情况的原因是，任务被挂起直到延迟时间到，而中断是关掉的。OSTimeDly() 实际上是依靠时钟节拍实现中断的，而因为中断是关掉的，程序不可能获得时钟节拍中断。很明显，所有的 PEND 调用都会涉及到这个问题，需十分小心。一个通用的办法是，应该在中断允许的情况下调用 μC/OS-II 的系统功能函数。

OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()可以使用 3 种不同的方式实现，但只需要其中一种方式就可以了。即便如此，还是在 OS_CPU.H 文件中给出 3 种不同的方法(见程序清单 L13.1)。使用何种实现方法，主要取决于使用的处理器类型以及 C 编译器的特性。可以在 OS_CPU.H 文件中通过定义 OS_CRITICAL_METHOD 常数，选择使用不同的实现方法。必须使用 OS_CPU.H 文件中给出的定义 OS_CRITICAL_METHOD 的方法，因为设置为 3 时，μC/OS-II 要多定义一个名为 cpu_sr 的局部变量。

实现方法 1 OS_CRITICAL_METHOD == 1

实现 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()这 2 个宏的第 1 种方法，也是最简单的方法是，在 OS_ENTER_CRITICAL()中调用处理器指令关中断，以及在 OS_EXIT_CRITICAL()中调用相应处理器指令开中断；但是，在这个过程中还存在着小小的问题：如果在禁止中断的情况下调用 μC/OS-II 函数，那么从 μC/OS-II 函数返回时，中断可能会变成允许的了。而实际上如果调用 μC/OS-II 的功能函数之前中断是关掉的，就表明再从μC/OS-II函数返回时，希望中断还是关掉的。在这种情况下，仅靠这种方法可能是不适宜的，但是有一些 CPU 或者 C 编译器只能如此处理。方法 1 实现的方式如程序清单 L13.3 所列。在此，假定处理器和 C 编译器可以使用 disable_int() 和 enable_int()这 2 个函数关/开中

断。这里函数的名字是为了看起来直观而无须解释，当然用户使用的 C 编译器可以用其他函数名。

程序清单 L13.3　实现方法 1

```
#define  OS_ENTER_CRITICAL()  disable_int()      /* 关中断 */
#define  OS_EXIT_CRITICAL()   enable_int()       /* 开中断 */
```

实现方法 2　OS_CRITICAL_METHOD == 2

第 2 种方法是：执行 OS_ENTER_CRITICAL()时，先将中断状态保存到堆栈中，然后关中断；而当执行 OS_EXIT_CRITICAL()时，再从堆栈中恢复原来的中断开/关状态。如果用这种方法，那么不管用户是在中断禁止，还是中断允许的情况下调用 μC/OS - II 的功能函数，调用后都不会改变中断状态。换句话说，如果调用前中断是开着的，则调用后中断还是开着的；调用前中断是关掉的，则调用后中断仍然是关掉的。如果在中断禁止时调用 μC/OS - II 的功能函数，则需特别当心，因为实际上是在延长应用程序的中断响应时间。程序清单 L13.4 是实现方法 2 的示意性代码。

程序清单 L13.4　实现方法 2

```
#define OS_ENTER_CRITICAL()        \
        asm(" PUSH    PSW");       \
        asm(" DI");
#define OS_EXIT_CRITICAL()         \
        asm(" POP     PSW");
```

在此，假设所使用的 C 编译器支持用 asm()插入行汇编代码，如程序清单 L13.4 所列。用户需要参阅自己使用的 C 编译器文档，以确认是否有这种功能。

PUSH PSW 语句将处理器状态寄存器 PSW（也称为条件码寄存器或者处理器标志寄存器）压入堆栈；而 DI 语句实现禁止中断；最后，POP PSW 语句恢复中断原来的状态。这里使用的汇编指令并不针对某个具体的 CPU，只是为了说明实现方法。

一些编译器对插入的行汇编代码优化得并不好，上述办法未必可行，因为编译器可能没有那么机敏——知道堆栈指针的值因 PUSH 指令而改变了。特别是，当用户使用的处理器有堆栈指针相对寻址模式时，编译器用这种模式对局部变量寻址，或者用栈指针加偏移量传递参数。显然，如果 OS_ENTER_CRITICAL()宏调用改变了堆栈指针，那么所有的栈偏移量都可能会出现偏差并导致严重错误。

实现方法 3　OS_CRITICAL_METHOD == 3

某些编译器提供了扩展功能，用户可以得到当前处理器状态字的值，并将其保存在 C 函数的局部变量之中。这个变量可以用于恢复 PSW，如程序清单 L13.5 所列。

程序清单 L13.5 保存和恢复 PSW

```
void Some_uCOS_II_Service (arguments)
{
    OS_CPU_SR  cpu_sr                                          (1)

    .
    cpu_sr = get_processor_psw();                              (2)
    disable_interrupts();                                      (3)
    .
    /* 临界段代码 */                                            (4)
    .
    set_processor_psw(cpu_sr);                                 (5)
}
```

L13.5(1) OS_CPU_SR 是 μC/OS-II 的一种数据类型，在与处理器有关的头文件(OS_CPU.H)中定义。选用这种实现方式以后，OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()2 个函数就认为有了 cpu_sr 这个局部变量。换句话说，如果想使用第 3 种实现方式保护临界段代码，就必须在用户程序中的每个用到这 2 个宏的函数中都定义 cpu_sr。当然，不必在 μC/OS-II 的文件里定义这个局部变量，因为 μC/OS-II 的相关文件中已经定义了这个局部变量。

L13.5(2) 为进入临界段代码，调用一个编译器提供的函数，得到 PSW(条件码寄存器、处理器标志或在用户的处理器中这个寄存器叫什么其他的名字)的值。把这个函数叫做 get_processor_psw()是为了讨论方便，在用户使用的编译器中的函数可能是其他名字。

L13.5(3) 自然会调用 C 编译器提供的另外一个函数 disable_interrupts()，用来关中断。

L13.5(4) 至此，可执行临界段代码了。

L13.5(5) 临界段代码执行完以后，可以调用另一个编译器提供的功能扩展函数，重新开中断。为便于讨论，笔者称之为 set_processor_psw()。该函数中有一个参数是原来的处理器状态字 PSW，执行这个函数可恢复原来处理器中断的开/关状态。

由于笔者不知道编译器所提供函数的具体名称(没有标准的命名法)，μC/OS-II 中的宏调用按其功能可以写成，如程序清单 L13.6 所列。

程序清单 L13.6 实现方法 3

```
#define OS_ENTER_CRITICAL()              \
        cpu_sr = get_processor_psw();    \
        disable_interrupts();

#define OS_EXIT_CRITICAL()               \
        set_processor_psw(cpu_sr);
```

13.03.03　OS_STK_GROWTH

绝大多数微处理器和微控制器的堆栈是从上往下递减的，但是也有某些处理器使用的是相反的方式。

L13.1(9)　μC/OS - II 被设计成对 2 种情况都可以处理，只要再用配置常数 OS_STK_GROWTH指定堆栈的方向就可以了：
置 OS_STK_GROWTH 为 0，表示堆栈从下(低地址)往上(高地址)递增；
置 OS_STK_GROWTH 为 1，表示堆栈从上(高地址)往下(低地址)递减。

之所以这样处理，是出于 2 个原因：首先，OSInit()需要知道，当用 OS_TaskIdle() 和 OS_TaskStat()函数建立任务时，堆栈的顶端地址在哪里；其次，在调用 OSTaskStkChk() 时，μC/OS - II需要知道堆栈的底端地址在哪里，从而得到堆栈的使用情况。

13.03.04　OS_TASK_SW()

L13.1(10)　OS_TASK_SW()是一个宏，是在 μC/OS - II 从低优先级任务切换到高优先级任务时须用到的。OS_TASK_SW()总是在任务级代码中被调用。另一个函数 OSIntExit()用在中断服务子程序 ISR 中。当中断服务子程序使更高优先级任务进入就绪态时，OSIntExit()完成任务切换功能。任务切换只是简单地将处理器的寄存器保存到将被挂起的任务的堆栈中，并且从堆栈中恢复要运行的更高优先级的任务。

在 μC/OS - II 中，处于就绪态任务的堆栈结构看起来就像刚刚发生过中断一样，所有的寄存器都保存在堆栈中。换句话说，μC/OS - II 要运行处于就绪态的任务必须要做的事就是，从任务堆栈中恢复处理器所有的寄存器，并且执行中断返回指令。为了任务调度，可以通过执行 OS_TASK_SW()模仿中断的产生。绝大多数处理器会提供软中断或指令陷阱(TRAP)来完成这项功能。中断服务子程序或指令陷阱处理函数(也叫做异常处理函数)的中断向量地址必须指向汇编语言函数 OSCtxSw()(参看 13.04.02 小节："OSTaskCreateHook()")。

例如，在 Intel 或者 AMD80x86 处理器上可以使用 INT 指令(如程序清单 L13.7 所列)，但是中断向量必须指向 OSCtxSw()。用户必须知道自己所使用的编译器和处理器是如何实现让中断向量指向 OSCtxSw()的。

程序清单 L13.7　实现方法 3(续)

```
#define  OS_TASK_SW()          asm  INT  080H
```

Motorola 68HC11 处理器使用的是 SWI 指令。这里，SWI 就自然是 OSCtxSw()。还有，Motorola 680x0 CPU32 可能会使用 16 个指令陷阱中的 1 个，选中的指令陷阱向量处理最终还是 OSCtxSw()。

一些处理器，如 Zilog Z80，并不提供软中断机制。在这种情况下，需要想办法将堆栈结构设置成与软中断发生后的堆栈结构一样。在 OS_TASK_SW()函数中调用 OSCtxSw()，而不

是将某个中断向量指向 OSCtxSw()。实际上 μC/OS 已经被移植到了 Z80 处理器上，μC/OS-II也同样是可以的。

13.04 OS_CPU_C.C

μC/OS-II 的移植范例要求用户编写 10 个简单的 C 函数：

OSTaskStkInit();
OSTaskCreateHook();
OSTaskDelHook();
OSTaskSwHook();
OSTaskIdleHook();
OSTaskStatHook();
OSTimeTickHook();
OSInitHookBegin();
OSInitHookEnd();
OSTCBInitHook()。

惟一必要的函数是 OSTaskStkInit()，其他 9 个函数必须声明，但并不一定要包含任何代码。这些函数的原型放在本章的末尾。

13.04.01 OSTaskStkInit()

OSTaskCreate()和 OSTaskCreateExt()通过调用 OSTaskStkInit()，初始化任务的栈结构；因此，堆栈看起来就像中断刚发生过一样，所有寄存器都保存在堆栈中。OSTaskStkInit()的示意性代码如程序清单 L13.8 所列。

程序清单 L13.8 OSTaskStkInit()的示意性代码

```
OS_STK *OSTaskStkInit (void   (*task)(void *pd),
                       void    *pdata,
                       OS_STK  *ptos,
                       INT16U   opt);
{
    模拟带参数(pdata)的函数调用;                                    (1)
    模拟 ISR 向量;                                                  (2)
    按照预先设计的寄存器值初始化堆栈结构;                           (3)
    返回栈顶指针给调用该函数的函数;                                 (4)
}
```

图 F13.2 显示了 OSTaskStkInit() 在建立任务时，任务栈应该初始化成何种形式。

注意：在这里假定堆栈是从上往下递减的。下面的讨论同样适用于以相反方向从下往上递增的堆栈结构。

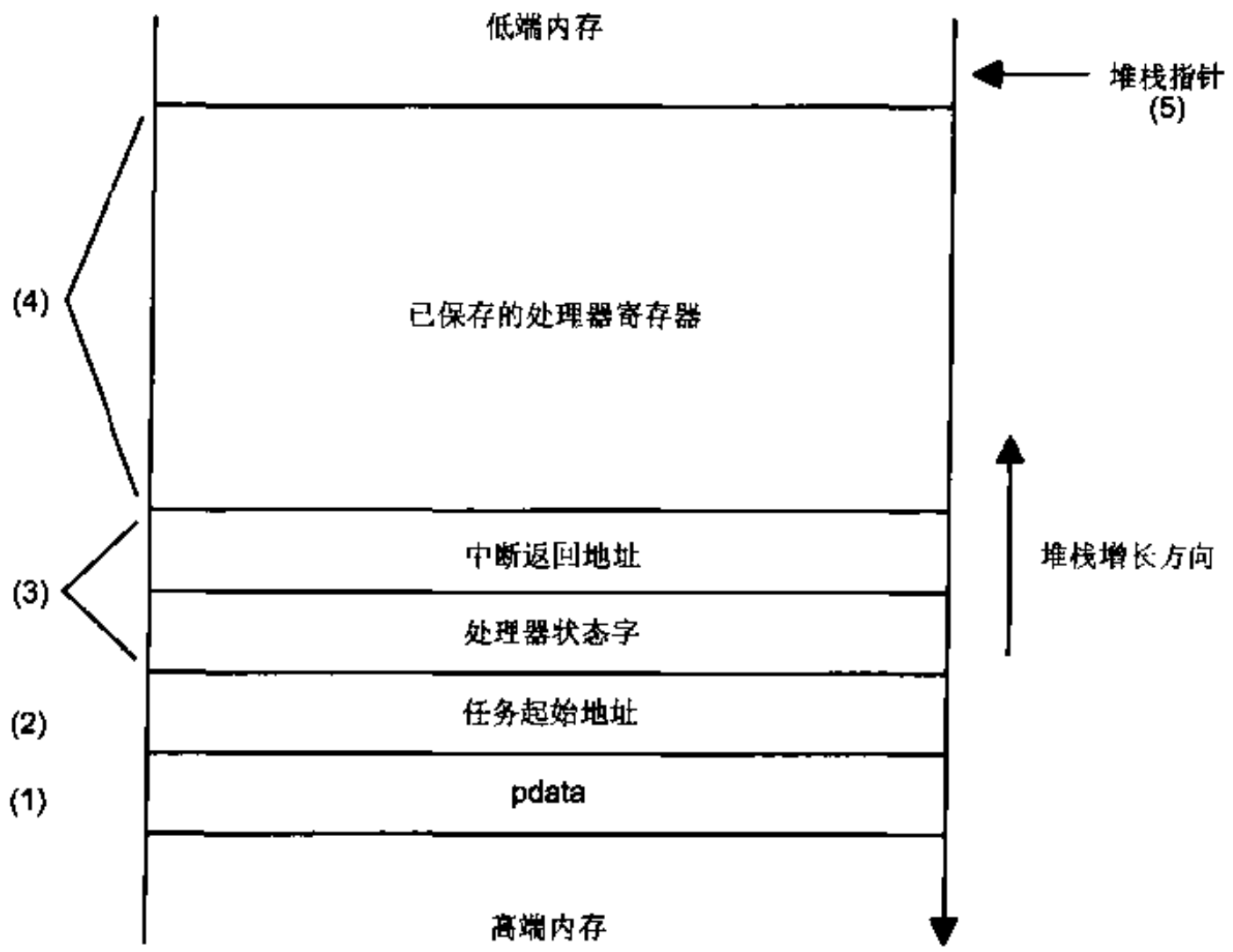


图 F13.2　初始化堆栈结构，pdata 参数传递给堆栈

程序清单 L13.9 给出了 OSTaskCreate()，OSTaskCreateExt()及 OSTaskStkInit() 的函数原型，其中参数 task，pdata，ptos 及 opt 是调用 OSTaskStkInit()函数时须传递过去的参数。因为只有 OSTaskCreate()函数不支持附加的 opt 选项，因此，当 OSTaskCreate()调用 OSTaskStkInit()时，将 opt 设置为 0x0000。

程序清单 L13.9　函数原型

```
INT8U  OSTaskCreate (void   (*task)(void *pd),
                     Void    *pdata,
                     OS_STK  *ptos,
                     INT8U    prio)
INT8U  OSTaskCreateExt (void    (*task)(void *pd),
                        void     *pdata,
                        OS_STK   *ptos,
                        INT8U     prio,
                        INT16U    id,
                        OS_STK   *pbos,
                        INT32U    stk_size,
                        void     *pext,
                        INT16U    opt)
```

```
OS_STK *OSTaskStkInit (void   (*task)(void *pd),
                       void    *pdata,
                       OS_STK  *ptos,
                       INT16U   opt);
```

回顾一下，在 μC/OS-II 中，任务是一个无限的循环，其他部分看起来与别的 C 函数没有什么不同。

当 μC/OS-II 让任务开始执行时，任务就会收到一个参数，好像是被其他任务调用了，如程序清单 L13.10 所列。

程序清单 L13.10　任务代码

```
void MyTask (void *pdata)
{
    /* 用"pdata"参数做某些操作 */
    for (;;) {
        /* 任务代码 */
    }
}
```

如果是从其他函数中调用 MyTask()，那么 C 编译器就会先将调用 MyTask()的函数的返回地址保存到堆栈中，再将参数保存到堆栈中。OSTaskStkInit()需要模仿编译器的这种动作。实际上有些编译器会将 pdata 参数放在一个或多个寄存器中传递，后面会讨论这类情况。

F13.2(1)

L13.8(1)　假定 pdata 会被编译器保存到堆栈中，OSTaskStkInit()就会模仿编译器的这种行为，将 pdata 保存到堆栈中。

F13.2(2)

L13.8(1)　与 C 函数的调用不一样的是，调用者的返回地址是未知的，因为该函数根本就没有被调用，只是为了建立一个任务的堆栈结构，好像这段任务代码被调用过一样。OSTaskStkInit()函数仅仅知道任务的起始地址(是作为参数传递过来的)。换言之，也不需要知道返回地址，因为任务并不再返回。

F13.2(3)

L13.8(2)　这时，OSTaskStkInit()需将处理器的寄存器保存到堆栈中。当处理器识别并开始执行中断时，它会自动地完成该过程。一些处理器会将所有的寄存器都推入堆栈，而另一些处理器只将部分寄存器推入堆栈。一般而言，处理器至少需将程序计数器的值(中断返回地址)和处理器的状态字存入堆栈。很明显，处理器是按一定的顺序将寄存器推入堆栈的，而用户在将寄存器推入堆栈时，也就必须依照这一顺序。

F13.2(4)

L13.8(3)　OSTaskStkInit()需要将剩下的处理器寄存器保存到堆栈中。入栈的顺序取

决于处理器是否允许按不同顺序操作。有些处理器用一条指令就可以一次将全部寄存器都保存起来，有些则要几条指令。必须严格按照处理器要求的入栈顺序完成这一过程。例如，Intel 80x86 有 PUSHA 指令，可将 8 个寄存器推入堆栈；对 Motorola 68HC11 处理器而言，在中断响应期间，所有的寄存器都会按一定顺序自动地保存到堆栈中。所以在初始化堆栈时，也必须符合这一顺序。

F13.2(5)

L13.8(4)　在初始化堆栈以后，OSTaskStkInit()应当返回堆栈指针所指向的地址。OSTaskCreate()或 OSTaskCreateExt()得到这个地址，并且保存在任务控制块中。处理器的文档应该告诉用户，堆栈指针是指向下一个可以使用的堆栈空间，还是指向上次入栈的数据。例如 80x86 就是指向上次存储的数据，而 Motorola 的 68HC11 处理器是指向下一个可以使用的栈空间。

现在是讨论这个问题的时候了：如果用户的 C 编译器用寄存器传递 pdata 参数，而不使用堆栈传递参数，该怎么办？请看图 F13.3。

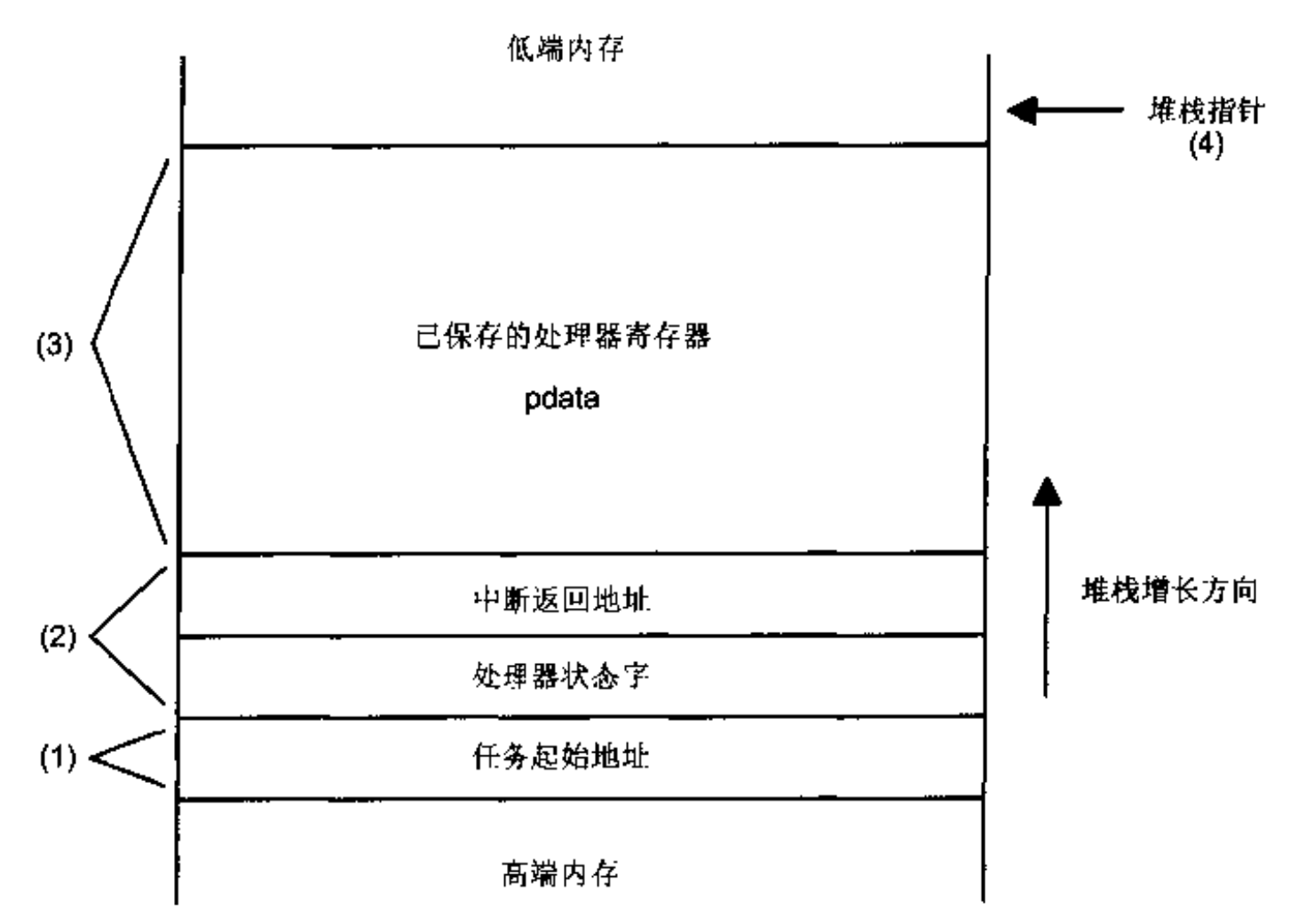


图 F13.3　堆栈初始化(pdata 通过寄存器传递)

F13.3(1)　同上一种情况类似，OSTaskStkInit()模仿调用应用程序任务代码的情况，将任务起始地址保存在堆栈中。

F13.3(2)　同样地，OSTaskStkInit()将寄存器保存到堆栈中。当处理器识别并开始执行中断时，自动地将寄存器推入堆栈中。一些处理器会将所有的寄存器都推入堆栈，而另外一些处理器只将部分寄存器推入堆栈。一般而言，处理器至少需将程序计数器的值(中断返回地址)和处理器的状态字存入堆栈。很明

显，处理器是按一定的顺序将寄存器推入堆栈的，而用户在将寄存器推入堆栈时，也必须严格依照同样的顺序。

F13.3(3) OSTaskStkInit()会将其余的处理器寄存器保存到堆栈。入栈的顺序取决于处理器是否允许按不同顺序操作。有些处理器可以使用一条指令将很多寄存器推入堆栈，有些则需要几条指令。这时，应该模拟出这些指令。因为编译器是使用寄存器传递参数的(至少部分参数是通过寄存器传递的)，所以应从相应文档中弄清楚 pdata 是通过哪个寄存器传递的，并且将该寄存器也推入堆栈。

F13.3(4) 在初始化堆栈后，OSTaskStkInit()函数应当返回堆栈指针所指向的地址。OSTaskCreate()或 OSTaskCreateExt()得到这个地址并且保存在任务控制块中。再次强调，应该清楚堆栈指针是指向下一个空的堆栈空间，还是指向上次推入堆栈的值。

13.04.02 OSTaskCreateHook()

每当添加任务时，OS_TCBInit()函数都会调用 OSTaskCreateHook() 函数。该函数允许扩展 μC/OS-II 的功能。当 μC/OS-II 设置完任务控制块(OS_TCB)初始化的绝大部分工作后，但在任务控制块被链接到相应的任务链中之前，以及在该任务就绪运行之前，μC/OS-II 会调用 OSTaskCreateHook()。该函数被调用时中断是开着的。

当 OSTaskCreateHook()被调用时，它会收到指向刚刚建立任务的任务控制块的指针。这样，OSTaskCreateHook()就可访问任务控制块结构所有的成员了。当用 OSTaskCreate()建立任务时，OSTaskCreateHook()的功能是有限的；但当使用 OSTaskCreateExt()建立任务时，会得到 OS_TCB 中的扩展指针(OSTCBExtPtr)。该指针可用来访问任务的附加数据，如浮点寄存器、MMU 寄存器、任务计数器以及调试信息。可以检查 OS_TCBInit()，看都做了哪些工作。第 15 章将说明如何使用这个函数。

关于 OS_CPU_HOOKS_EN 的注解

只有当 OS_CFG.H 中的 OS_CPU_HOOKS_EN 置为 1 时，才会生成 OS_CPU_C.C 文件中的相应 OS???Hook()函数。这些在本节和以下各节描述的 HOOK 函数的代码中，OS???Hook()函数是始终需要的，"#define constant OS_CPU_HOOKS_EN"并不意味着 OS???Hook()函数是否被调用。OS_CPU_HOOKS_EN 只是表明 OS???Hook()函数是在 OS_CPU_C.C 文件中(OS_CPU_HOOKS_EN == 1)，还是在其他文件中(OS_CPU_HOOKS_EN == 0)。这个特点使得用户可以在自己的移植实例中，在 OS_CPU_C.C文件之外定义自己的 HOOK 函数。显然，使用该移植范例其他源文件的用户，应将 OS_CPU_HOOKS_EN 设置为 0，以防止链接时出现重复定义错误。如果不需要使用 HOOK 函数这种机制来扩展 μC/OS-II 的功能，可以将这些 HOOK 函数写成空函数。需要再次强调的是，μC/OS-II 需要这些 HOOK 函数(这些函数必须在某处定义)。

13.04.03 OSTaskDelHook()

在任务从就绪列表或等待列表(若任务在等待某事件发生)中被删除后，OSTaskDel()就

会调用 OSTaskDelHook()。该函数在将任务从 μC/OS-II 的内部有效任务链表中删除之前被调用。当调用 OSTaskDelHook()时，它会收到一个指向正在被删除任务的任务控制块的指针，这样它就可以访问该任务控制块所有的结构成员了。OSTaskDelHook()可以用来检验 TCB 扩展部分是否已建立了(一个非空的指针)，并进行一些清 0 等操作。OSTaskDelHook()被调用时，中断是被关掉的，所以该函数的代码太长会影响中断响应时间。

在第 15 章，用户可以研究 OSTaskDel()函数，看在调用 OSTaskDelHook()之前，都做了哪些工作。

13.04.04　OSTaskSwHook()

做任务切换时，会调用 OSTaskSwHook()函数。不管任务切换是通过 OSCtxSw()函数实现的，还是通过 OSIntCtxSw()函数(见 13.05 节："OS_CPU_A.ASM")实现的，都会调用该函数。OSTaskSwHook()可以直接访问 OSTCBCur 和 OSTCBHighRdy 这 2 个全局变量。OSTCBCur 指向将被切换出去的任务的任务控制块，而 OSTCBHighRdy 指向新任务的任务控制块。

注意：在调用 OSTaskSwHook() 期间，中断一直是关掉的。因为附加的代码会影响中断的响应时间，所以应尽量使这部分代码减至最少。

OSTaskSwHook()没有任何参数，也不返回任何值。第 15 章将详细讲解如何使用这个函数。

13.04.05　OSTaskStatHook()

OSTaskStatHook()函数每秒都会被统计任务 OSTaskStat()调用一次。可以用 OSTaskStatHook()扩展统计任务的功能。例如，可以跟踪并显示每个任务的执行时间、每个任务所用的 CPU 份额以及每个任务执行的频率等等。

OSTaskStatHook() 没有任何参数，也不返回任何值。第 1 章的范例 3 介绍了应该如何使用 OSTaskStatHook()。

13.04.06　OSTimeTickHook()

OSTaskTimeHook()函数在每个时钟节拍都会被 OSTimeTick()调用。实际上，OSTimeTickHook()是在 μC/OS-II 真正处理时钟节拍之前被调用的，以便于用户能先处理应急的事务。

OSTimeTickHook()没有任何参数，也不返回任何值。

13.04.07　OSTCBInitHook()

OS_TCBInit()函数在调用 OSTaskCreateHook()之前，会先调用 OSTCBInitHook()。这样做的原因在于，用户可以在 OSTCBInitHook()函数中做一些与初始化控制块 OS_TCB 有关的处理；在 OSTaskCreateHook() 中做一些与初始化任务有关的处理。是否用 OSTCBInitHook()和 OSTaskCreateHook()函数，完全取决于用户。同 OSTaskCreateHook()一样，OSTCBInitHook() 会收到指向新添加任务的任务控制块的指针，而这个新添加任务的任务控制块绝大部分已经初始化完成，但是还没有链接到已经建立任务的链表中。详见

OS_TCBInit()。

13.04.08 OSTaskIdleHook()

很多微处理器都允许执行相应的指令，将CPU置于低功耗模式。而当接收到中断信号时，CPU就会退出低功耗模式。OSTaskIdle()函数可调用OSTaskIdleHook()函数，实现CPU的这种低功耗模式，如程序清单L13.11所列。

程序清单L13.11 OSTaskIdleHook()的使用

```
void  OS_TaskIdle (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


    pdata = pdata;
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtr++;                                                    (1)
        OS_EXIT_CRITICAL();
        OSTaskIdleHook();                                               (2)
    }
}

void  OSTaskIdleHook (void)
{
    asm(" STOP");                                                       (3)
    /* 收到中断并完成中断服务 */                                         (4)
}
```

L13.11(1) 我们知道，当没有任何其他任务运行时，就会运行OS_TaskIdle()函数。OS_TaskIdle()运行时给空闲计数器OSIdleCtr加1。

L13.11(2) OS_TaskIdle()调用在OS_CPU_C.C文件中声明的OSTaskIdleHook()函数。

L13.11(3) OSTaskIdleHook()调用CPU的特定指令，使CPU进入低功耗模式。为了便于阐述清楚，假设编译器支持插入行汇编，并且CPU的低功耗指令是STOP指令。若编译器不支持插入行汇编，那么可将OSTaskIdleHook()函数在OS_CPU_A.ASM文件中用汇编语句实现，别忘了子程序返回指令。

同样，使CPU进入低功耗的指令也可以用其他名字。

L13.11(4) 当中断发生时，CPU退出低功耗模式，执行中断服务程序ISR。ISR会向更高优先级的任务发信号，更高优先级的任务会开始执行，因为ISR结束时会

调用 OSInitExit()。当所有其他任务再次等待事件发生时，μC/OS-II 在中断服务完成后，又会立刻切换到空闲任务。这样 OSTaskIdleHook() 又会在 OS_TaskIdle() 中被调用，同样的处理过程又会重复。

其实可以在 OSTaskIdleHook()函数中让 LED 闪烁，以表示 CPU 的状态。例如，暗淡的 LED 表示 CPU 很忙；而明亮的 LED 表示 CPU 负荷比较轻。

13.04.09　OSInitHookBegin()

进入 OSInit()函数后，OSInitHookBegin() 就会立即被调用。添加这个函数的原因在于，想把与 OS 有关的初始化代码也放在 OSInit() 函数中。这个函数使得用户可以将自己特定的代码也放在 OSInit()函数中。这样代码会简洁明了。

13.04.10　OSInitHookEnd()

OSInitHookEnd()与 OSInitHookBegin()函数相似，只是它在 OSInit()函数返回之前被调用。添加这个函数的原因与添加 OSInitHookBegin()的原因是相同的。可以在第 15 章看到 OSInitHookEnd()的例子。

13.05　OS_CPU_A.ASM

- μC/OS-II 的移植实例要求用户编写 4 个简单的汇编语言函数：

  OSStartHighRdy()；

  OSCtxSw()；

  OSIntCtxSw()；

  OSTickISR()。

如果编译器支持插入行汇编代码，就可以将所有与处理器相关的代码放到 OS_CPU_C.C 文件中，而不必再有单独的汇编语言文件。

13.05.01　OSStartHighRdy()

OSStart()函数调用 OSStartHighRdy()来使就绪态任务中优先级最高的任务开始运行，这个函数的示意性代码如程序清单 L13.12 所列。用户应该将它转换成汇编语言代码。

程序清单 L13.12　OSStartHighRdy()函数的示意性代码

```
void OSStartHighRdy (void)
{
    调用用户定义的 OSTaskSwHook();                                    (1)
    OSRunning = TRUE;                                                 (2)
    得到将要恢复运行任务的堆栈指针：                                  (3)
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
```

```
    从新任务堆栈中恢复处理器的所有寄存器;                                    (4)
    执行中断返回指令;                                                        (5)
}
```

L13.12(1) OSStartHighRdy()必须调用OSTaskSwHooK()函数;但OSStartHighRdy()函数只是做了任务切换工作的一半——只是完成了高优先级任务寄存器的恢复,而并没有保存当前任务的寄存器。OSTaskSwHooK()函数必须检查OSRunning标志位,以确定OSTaskSwHooK()函数是被OSStartHighRdy()函数调用的(OSRunning是FALSE),还是在正常的任务切换之中(OSRunning是TRUE)被调用的。

L13.12(2) OSStartHighRdy()调用完OSTaskSwHooK()后,将OSRunning标志位设置为TRUE,这时更高优先级任务的寄存器还没有被恢复。或许注意到,应该将这2条语句放在OSStart()中,而不应放在OSStartHighRdy()中;但不幸的是,刚开始实现OSStart()时,并没有意识到这一点。如果现在更改OSStart(),将会有很多移植不能正常工作,所以仍然将这些声明放在函数OSStartHighRdy()中。

L13.12(3) OSStartHighRdy()需得到最高优先级任务的堆栈指针。OSStartHighRdy()假设OSTCBHighRdy指向最高优先级任务的任务控制块。为了简单起见,堆栈指针总是储存在任务控制块的开头。换句话说,也就是需要恢复的任务堆栈指针总是储存在任务控制块的偏移地址为0的位置。

L13.12(4) 在μC/OS-II中,处于就绪状态任务的堆栈结构看起来就像刚发生过中断一样,所有的寄存器都保存在堆栈中。要想运行最高优先级任务,需将所有处理器寄存器按顺序从任务堆栈中恢复出来,并且执行中断返回指令。在这一步中,OSStartHighRdy()从堆栈中取出所有寄存器。最重要的一点是,必须按照OSTaskStkInit()中相反的顺序弹出寄存器(见13.04.01小节:"OSTaskStkInit()")。

L13.12(5) OSStartHighRdy()函数中最后一个步骤就是执行中断返回指令,使得CPU从堆栈里面恢复程序指针以及CPU的标志寄存器(也称为状态寄存器)。该操作使得CPU开始恢复执行最高优先级任务的第1条指令。

切记,在调用OSStart()函数之前,要已经建立了至少1个应用任务(见OSTaskCreate()和OSTaskCreateExt()函数)。

13.05.02 OSCtxSw()

任务级的切换是通过执行软中断指令,或者依据处理器的不同,执行TRAP(陷阱)指令来实现的。中断服务子程序、陷阱或异常处理的向量地址必须指向OSCtxSw()。

如果当前任务调用μC/OS-II提供的功能函数,并使得更高优先级任务进入了就绪态,μC/OS-II就会借助上面提到的向量地址找到OSCtxSw()。在系统服务调用的最后,

μC/OS-II会调用 OSSched()，并由此推断出当前任务不再是需要运行的最重要的任务了。OSSched()先将最高优先级任务的地址装载到 OSTCBHighRdy，再通过调用 OS_TASK_SW()执行软中断或陷阱指令。

注意：变量 OSTCBCur 已经包含了指向当前任务的任务控制块(OS_TCB)的指针。软中断(或陷阱)指令会强制将处理器的一些寄存器(通常是返回地址和处理器状态字)保存到当前任务的堆栈中，并使处理器执行 OSCtxSw()。

OSCtxSw()的示意性代码如程序清单 L13.13 所列，这些代码必须用汇编语言编写，因为用户不能直接在 C 语言中访问 CPU 寄存器。

注意：在执行 OSCtxSw()和用户定义的函数 OSTaskSwHook()的过程中，中断是关掉的。当 OS_TASK_SW()调用 OSCtxSw()时，OSCtxSw()认为处理器的程序计数器以及标志寄存器(或者状态寄存器)已经被 OS_TASK_SW()的软中断指令压入堆栈中了。

程序清单 L13.13　OSCtxSw()的示意性代码

```
void OSCtxSw (void)
{
    保存处理器寄存器;                                                    (1)
    在当前任务的任务控制块中保存当前任务的堆栈指针:                      (2)
        OSTCBCur->OSTCBStkPtr = Stack pointer;
    OSTaskSwHook();                                                      (3)
    OSTCBCur  = OSTCBHighRdy;                                            (4)
    OSPrioCur = OSPrioHighRdy;                                           (5)
    得到将要重新开始运行的任务的堆栈指针:                                (6)
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    从新任务的任务堆栈中恢复处理器所有寄存器的值:                        (7)
    执行中断返回指令;                                                    (8)
}
```

L13.13(1)　OSCtxSW()按照同 OSTaskInit() 函数一样的顺序将处理器的寄存器保存到堆栈之中(除了软中断中已经被保存的寄存器)。

L13.13(2)　当将要被挂起的任务的所有 CPU 寄存器都已经入栈后，OSCtxSW() 将堆栈指针保存在当前任务的任务控制块(OS_TCB)中。

L13.13(3)　当用户的移植需要扩展任务切换函数的功能时，OSCtxSW()调用函数 OSTaskSwHook()。

注意：无论 OSTaskSwHook()是在 OS_CPU_C.C 文件中声明的，还是在其他地方声明的，总是要被调用的。

L13.13(4)　OSCtxSW()此时需要将指向当前任务的指针指向要恢复运行的任务。换句话说，新的任务取代了当前任务。

L13.13(5)　OSCtxSW()将新任务的优先级复制给当前任务的优先级。

L13.13(6)　从新任务的任务控制块中得到新任务的堆栈指针。

L13.13(7)　OSCtxSW()恢复将运行任务的 CPU 寄存器。必须按照这些寄存器保存时

相反的顺序恢复它们。举个例子，如果处理器有4个寄存器R1、R2、R3及R4，而且按照R1、R2、R3、R4的顺序保存它们，那么恢复时就应该是从R4开始，到R1结束。

L13.13(8) 因为高优先级任务的程序计数器（或者还有状态寄存器）仍然在堆栈之中，此时一个中断返回指令使得程序指针和状态寄存器从堆栈中弹出到CPU中，从而使用户任务恢复运行。

可以在www.uCOS-II.com网站上看到基于80x86 CPU的任务切换的范例。

13.05.03 OSTickISR()

μC/OS-II要求用户提供一个周期性的时钟源，来实现时间的延迟和超时功能。时钟节拍应该每秒发生10～100次/秒。为了完成该任务，可以使用硬件定时器，也可以从交流电中获得50/60 Hz的时钟频率。

必须在开始多任务后，即调用OSStart()后，启动时钟节拍中断；但由于OSStart()函数不会返回，用户无法实现这一操作。然而，可以在OSStart()运行后，μC/OS-II启动运行的第1个任务中初始化节拍中断。这个任务是调用OSStart()之前建立的任务中优先级最高的任务。通常容易犯的错误是，在调用OSInit()和OSStart()之间打开了时钟节拍中断（如程序清单L13.14所列）。产生问题的原因在于，在μC/OS-II开始第1个任务之前，时钟节拍中断服务程序就会执行，而此时的μC/OS-II处于未知状态，当然应用程序就会崩溃。

程序清单L13.14 在不正确的位置启动时钟节拍中断

```
void main (void)
{
    .
    .
    OSInit();                    /* 初始化 μCOS-II          */
    .
    .
    /* 应用程序初始化代码...                                */
    /* ... 调用OSTaskCreate()建立至少1个任务
    .
    .
    开时钟节拍中断;              /* 千万不要在这里开中断!!! */
    .
    .
    OSStart();                   /* 开始多任务              */
}
```

时钟节拍中断服务子程序的示意性代码如程序清单L13.15所列。这些代码必须用汇编语言编写，因为在C语言中不能直接访问CPU寄存器。

程序清单 L13.15　时钟节拍 ISR 的示意性代码

```
void OSTickISR (void)
{
    保存处理器寄存器；                                          (1)
    调用 OSIntEnter()或者直接给 OSIntNesting 加 1；             (2)
    if (OSIntNesting == 1) {                                    (3)
        OSTCBCur->OSTCBStkPtr = Stack Pointer;
    }
    给产生中断的设备清中断；                                    (4)
    重新允许中断(可选)；                                        (5)
    OSTimeTick();                                               (6)
    OSIntExit();                                                (7)
    恢复处理器寄存器；                                          (8)
    执行中断返回指令；                                          (9)
}
```

L13.15(1)　时钟节拍中断服务子程序(ISR)同其他中断服务子程序一样，都要先将 CPU 寄存器保存到当前任务的堆栈中。当然，必须按照与 OSTaskInit()函数相同的顺序保存寄存器。

L13.15(2)　此时中断已经禁止，可以直接给 OSIntNesting(中断嵌套次数寄存器)加 1，而不用担心数据被其他中断服务破坏。过去一直推荐使用 OSIntEnter()函数处理中断嵌套层数的增加。在 ISR 开始之前做更多处理的原因是，限制中断嵌套的层数。在 OSIntEnter()函数中增加了边界检查功能，以保证中断嵌套的层数不超过 255 层。如果用户不可能达到如此的嵌套深度，也可以直接给 OSIntNesting 加 1，而不使用边界检查功能；但是如果想使系统更加安全和稳定，则简单调用 OSIntEnter()函数就可以了。当然，调用函数 OSIntEnter()会给中断服务子程序增加额外负担，这一点取决于用户的需求。

L13.15(3)　时钟节拍中断服务子程序检查 OSIntNesting 的数值。如果 OSIntNesting 等于 1，则需要在当前任务的任务控制块(OS_TCB)中保存堆栈指针。这个步骤是在 V2.51 版本中新添加的，虽然这使得 ISR 程序变得稍微复杂一些，但是却使得移植与编译器更加无关。

L13.15(4)　这里取决于中断源，有些中断设备需要在中断响应时清中断

L13.15(5)　此处可以开中断，用来响应更高优先级的中断。当然这个步骤是可选的，如果用户认为中断嵌套占用了过多的堆栈空间，则可以不允许中断嵌套。

L13.15(6)　OSTickISR()必须调用 OSTimeTick()来维持 μC/OS II 内部的定时。定时器允许任务被挂起一定的时间，或者允许 PEND 类型调用的超时功能。

L13.15(7)　因为执行了这个中断服务子程序，在此必须调用 OSIntExit()。大概还记得，OSIntExit()决定是否因为这个中断服务程序的执行，使得更高优先级的任务就绪。如果更高级别的任务就绪，OSIntExit()就不会返回到被中断的

任务，而是切换到更高优先级的任务。

L13.15(8) 如果没有更高优先级的任务进入就绪态，OSIntExit()就会返回，此时只需简单地恢复刚才保存的CPU寄存器就可以了。当然要用相反的顺序。

L13.15(9) OSTickISR()应该执行中断返回指令，返回被中断的任务。

13.05.04 OSIntCtxSw()

OSIntExit()通过调用OSIntCtxSw()，在ISR中执行任务切换功能。因为OSIntCtxSw()是在ISR中被调用的，所以假定所有的处理器寄存器都被正确地保存到了被中断任务的堆栈之中(参见13.05.03小节："OSTickISR()")。

OSIntCtxSw()的示意性代码如程序清单L13.16所列。这些代码必须用汇编语言编写，因为在C语言中不能直接访问CPU寄存器。如果编译器支持插入汇编语言代码，就可以将OSIntCtxSw()代码放在OS_CPU_C.C文件中，而不放在OS_CPU_A.ASM文件中。

注意：这些原型代码是为V2.51或者更高的版本而编写的，因为在V2.51版本以前，OSIntCtxSw()的代码还要多一些。如果使用的版本是V2.51以前的版本，那么推荐用户更改自己的移植代码，以符合L13.16中的算法。

OSIntCtxSw()函数中的绝大多数代码同OSCtxSw()函数是一样的。区别只是，因为ISR已经保存了CPU的寄存器，而不再需要再在OSIntCtxSw()函数中保存CPU的寄存器。实际上如果需要，则可以跳转到OSCtxSw()中相同的代码，以减少代码量。一般地，知道如何实现OSCtxSw()以后，也就知道如何实现OSIntCtxSw()了。

程序清单L13.16 OSIntCtxSw()的示意性代码(V2.51或以上版本)

```
void OSIntCtxSw (void)
{
    调用用户定义的 OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    得到将要重新执行的任务的堆栈指针：
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    从新任务堆栈中恢复所有处理器寄存器；
    执行中断返回指令；
}
```

L13.17中列出了μC/OS-II V2.51以前版本OSIntCtxSw()函数的实现的示意性代码。应该注意的地方是，在调用OSTaskSwHook()之前多出来的2行代码：L13.17(1)和L13.17(2)。在这些版本的移植代码上，ISR程序并没有像L13.15(3)中那样在OS_TCB中保存中断任务的堆栈指针；因此，OSIntCtxSw()需在多出的2行代码中完成这个工作。但是因为堆栈指针已改变了原来指向的地址(当OSIntCtxSw()开始执行时，OSInitExit()和OSIntCtxSw()的返回地址已经放到堆栈中了)，所以应该对堆栈指针做相应的调整，解决的办法是给堆栈指针加上一定的偏移量，而该偏移量取决于不同的编译器选项。就是因为这个偏移量的值，笔者收到了很多用户的电子邮件，比预期的要多很多。而其中一个非常聪明的用户

是 Nicolas Pinault，是他给出了以上的方法，可以避免调整堆栈指针。也是因为 Nicolas Pinault 的工作，μC/OS - II 不再依赖于编译器的选项。再次感谢 Nicolas！

程序清单 L13.17　OSIntCtxSw()的示意性代码(V2.51 之前版本)

```
void OSIntCtxSw (void)
{
    调整堆栈指针：                                                    (1)
        OSIntExit();
        OSIntCtxSw();
    在当前任务的任务控制块中保存当前任务的堆栈指针：                  (2)
        OSTCBCur->OSTCBStkPtr = Stack Pointer;
    调用用户定义的 OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    得到将要重新执行的任务的堆栈指针：
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    从新任务的堆栈中恢复所有处理器寄存器；
    执行中断返回指令；
}
```

13.06　测试移植代码

当用户为自己的处理器做完 μC/OS - II 的移植后，紧接着的工作就是验证移植的 μC/OS - II是否正常工作，而这可能是移植中最复杂的一步。应该首先不加任何应用代码来测试移植好的 μC/OS - II，也就是说，应该首先测试内核自身的运行状况。有 2 个原因促使这样做：首先，用户不希望将事情复杂化；其次，如果有些部分没有正常工作，可以明白是移植本身的问题，而不是应用代码产生的问题。如果已经将 2 个基本的任务和节拍中断运行起来，那么接下来添加应用任务是非常简单的。

可以使用各种不同的技术测试自己的移植工作，取决于用户个人在嵌入式系统方面的经验和对处理器的理解。这里笔者通过 4 个步骤测试移植代码：

- 确保 C 编译器、汇编编译器及链接器正常工作；
- 验证 OSTaskStkInit()和 OSStartHighRdy()函数；
- 验证 OSCtxSw()函数；
- 验证 OSIntCtxSw()和 OSTickISR()函数。

13.06.01　确保 C 编译器、汇编编译器及链接器正常工作

当修改完需要根据 CPU 更改的文件后，紧接着要把这些文件和 μC/OS - II 中与处理器无关的文件一同编译和链接。显然，这个步骤取决于使用的编译器。用户须熟悉自己所使用的编译器系统。

通常这样设置测试的目录：

\SOFTWARE\uCOS-II\processor\compiler\TEST

这里，processor 指的是做移植的处理器或者微控制器的型号名称；compiler 指的是使用的编译器。

表 T13.3 列出了要用到的文件。在 TEST 目录下，至少应该有 3 个文件：TEST.C，INCLUDES.H及 OS_CFG.H。根据使用处理器的不同，可能需要设置中断向量表等，放在 VECTOR.C 文件中，当然也可以叫其他名字。

表 T13.3 测试移植需要的文件

目 录	文 件
\SOFTWARE\uCOS-II\processor\compiler\TEST	TEST.C
	OS_CFG.H
	INCLUDES.H
	VECTORS.C
	MAKEFILE 或 IDEproject 文件
\SOFTWARE\uCOS-II\processor\compiler	OS_CPU_A.ASM
	OS_CPU_C.C
	OS_CPU.H
\SOFTWARE\uCOS-II\SOURCE	OS_CORE.C
	OS_FLAG.C
	OS_MBOX.C
	OS_MEM.C
	OS_MUTEX.C
	OS_Q.C
	OS_SEM.C
	OS_TASK.C
	OS_TIME.C
	uCOS_II.C
	uCOS_II.H

TEST 目录下还应该包括用来指定编译器、汇编器及链接器的 MAKEFILE 文件，以建立用户应用工程。当然使用 MAKEFILE 的前提是，使用有 make 功能的编译系统。如果编译器提供集成开发环境(IDE)，就不再需要 MAKEFILE 文件了；但是会有与该 IDE 相关的项目工程文件。

程序清单 L13.18 列出了典型的 INCLUDES.H 文件包含的内容。因为 OSTaskCreateExt()函数使用标准 C 函数 memset()初始化任务堆栈，所以应该包含 STRING.H 文件。其他标准的 C 语言的头文件如 STDIO.H、CTYPE.H 以及 STDLIB.H 等，实际上 μC/OS-II 并不使用，也可以包含进去，以备用户的应用代码可能用到。

程序清单 L13.18　典型的 INCLUDES.H

```
#include    <stdio.h>
#include    <string.h>
#include    <ctype.h>
#include    <stdlib.h>

#include    "os_cpu.h"
#include    "os_cfg.h"
#include    "ucos_ii.h"
```

程序清单 L13.19 列出了 OS_CFG.H 文件的内容。OS_CFG.H 文件用来确定允许 μC/OS - II的各种功能。

可以在本书光盘的\SOFTWARE\uCOS - II\EX1_x86L\BC45\SOURCE 目录下找到相似的例子作为起点，而不必重新键入 OS_CFG.H 文件。

程序清单 L13.19　OS_CFG.H 允许所有 μC/OS - II 的功能函数

```
/* ------------------------各种定义------------------------ */
#define OS_ARG_CHK_EN          1     /* 允许(1)或者禁止(0)变量检查                    */

#define OS_CPU_HOOKS_EN        1     /* 在处理器移植文件中允许使用 μC/OS - II 的接口函数  */

#define OS_LOWEST_PRIO        63     /* 定义任务的最低优先级...                       */
                                     /* ... 不得大于 63!                              */

#define OS_MAX_EVENTS         20     /* 应用中最多事件控制块的数目...                 */
                                     /* ... 必须大于 0                                */
#define OS_MAX_FLAGS          10     /* 应用中最多事件标志组的数目...                 */
                                     /* ... 必须大于 0                                */
#define OS_MAX_MEM_PART       10     /* 最多内存块的数目...                           */
                                     /* ... 必须大于 0                                */
#define OS_MAX_QS             10     /* 应用中最多队列控制块的数目...                 */
                                     /* ... 必须大于 0                                */
#define OS_MAX_TASKS          63     /* 应用中最多任务数目...                         */
                                     /* ... 必须大于或者等于 2                        */

#define OS_SCHED_LOCK_EN       1     /* 允许生成 OSSchedLock() 和 OSSchedUnlock()代码  */

#define OS_TASK_IDLE_STK_SIZE 512    /* 空闲任务堆栈容量(#按照 OS_STK 的宽度的数目)   */

#define OS_TASK_STAT_EN        1     /* 允许(1) 或者禁止(0) 统计任务                  */
#define OS_TASK_STAT_STK_SIZE 512    /* 统计任务堆栈的容量(#按照 OS_STK 的宽度计)     */
```

```
#define OS_TICKS_PER_SEC        200     /* 设置每秒时钟节拍数目                          */

/* ------------------------ 事件标志------------------------ */
#define OS_FLAG_EN              1       /* 允许(1)或者禁止(0) 产生事件标志代码           */
#define OS_FLAG_WAIT_CLR_EN     1       /* 允许生成 Wait on Clear 事件标志代码           */
#define OS_FLAG_ACCEPT_EN       1       /* 允许生成 OSFlagAccept() 函数代码              */
#define OS_FLAG_DEL_EN          1       /* 允许生成 OSFlagDel() 函数代码                 */
#define OS_FLAG_QUERY_EN        1       /* 允许生成 OSFlagQuery() 函数代码               */

/* ------------------------ 消息邮箱------------------------ */
#define OS_MBOX_EN              1       /* 使能 (1) 或者禁止(0) 产生邮箱代码             */
#define OS_MBOX_ACCEPT_EN       1       /* 允许生成 OSMboxAccept()函数代码               */
#define OS_MBOX_DEL_EN          1       /* 允许生成 OSMboxDel()函数代码                  */
#define OS_MBOX_POST_EN         1       /* 允许生成 OSMboxPost()函数代码                 */
#define OS_MBOX_POST_OPT_EN     1       /* 允许生成 OSMboxPostOpt()函数代码              */
#define OS_MBOX_QUERY_EN        1       /* 允许生成 OSMboxQuery()函数代码                */

/* ------------------------ 内存管理------------------------ */
#define OS_MEM_EN               1       /* 允许 (1) 或者禁止(0) 产生内存关系代码         */
#define OS_MEM_QUERY_EN         1       /* 允许生成 OSMemQuery()函数代码                 */

/* ------------------------ 互斥型信号量------------------------ */
#define OS_MUTEX_EN             1       /* 允许(1) 或者禁止(0) 产生互斥型信号量代码      */
#define OS_MUTEX_ACCEPT_EN      1       /* 允许生成 OSMutexAccept()函数代码              */
#define OS_MUTEX_DEL_EN         1       /* 允许生成 OSMutexDel()函数代码                 */
#define OS_MUTEX_QUERY_EN       1       /* 允许生成 OSMutexQuery()函数代码               */

/* ------------------------ 消息队列------------------------ */
#define OS_Q_EN                 1       /* 允许(1) 或者禁止 (0) 产生消息队列代码         */
#define OS_Q_ACCEPT_EN          1       /* 允许生成 OSQAccept()函数代码                  */
#define OS_Q_DEL_EN             1       /* 允许生成 OSQDel()函数代码                     */
#define OS_Q_FLUSH_EN           1       /* 允许生成 OSQFlush()函数代码                   */
#define OS_Q_POST_EN            1       /* 允许生成 OSQPost()函数代码                    */
#define OS_Q_POST_FRONT_EN      1       /* 允许生成 OSQPostFront()函数代码               */
#define OS_Q_POST_OPT_EN        1       /* 允许生成 OSQPostOpt()函数代码                 */
#define OS_Q_QUERY_EN           1       /* 允许生成 OSQQuery()函数代码                   */

/* ------------------------ 信号量------------------------ */
#define OS_SEM_EN               1       /* 允许(1) 或者禁止(0) 产生信号量相关代码        */
#define OS_SEM_ACCEPT_EN        1       /* 允许生成 OSSemAccept()函数代码                */
#define OS_SEM_DEL_EN           1       /* 允许生成 OSSemDel()函数代码                   */
```

```
#define OS_SEM_QUERY_EN          1     /* 允许生成 OSSemQuery()函数代码                    */

/* ------------------ 任务管理 ------------------ */
#define OS_TASK_CHANGE_PRIO_EN   1     /* 允许生成 OSTaskChangePrio()函数代码              */
#define OS_TASK_CREATE_EN        1     /* 允许生成 OSTaskCreate()函数代码                  */
#define OS_TASK_CREATE_EXT_EN    1     /* 允许生成 OSTaskCreateExt()函数代码               */
#define OS_TASK_DEL_EN           1     /* 允许生成 OSTaskDel()函数代码                     */
#define OS_TASK_SUSPEND_EN       1     /* 允许生成 OSTaskSuspend()函数和
                                       /* OSTaskResume()函数代码                           */
#define OS_TASK_QUERY_EN         1     /* 允许生成 OSTaskQuery()函数代码                   */

/* ------------------ 时间管理 ------------------ */
#define OS_TIME_DLY_HMSM_EN      1     /* 允许生成 OSTimeDlyHMSM()函数代码                 */
#define OS_TIME_DLY_RESUME_EN    1     /* 允许生成 OSTimeDlyResume()函数代码               */
#define OS_TIME_GET_SET_EN       1     /* 允许生成 OSTimeGet() 和 OSTimeSet()函数代码      */
typedef INT16U          OS_FLAGS;      /* 事件标志的数据类型(8 位,16 位或者 32 位)         */
```

程序清单 L13.20 给出了一个简单的 TEST.C 程序应该包括的内容。作为第 1 步，没有必要添加特别复杂的代码，因为只是想验证是否可以编译出正确的代码。此时，需用户来解决各种编译器、汇编器及链接器的错误。可能会遇到一些警告，需自己判断这些警告是否足以成为编译和链接的严重问题。

程序清单 L13.20　第 1 步，最小的 TEST.C

```
#include    "includes.h"

void  main (void)
{
    OSInit();
    OSStart();
}
```

13.06.02　验证 OSTaskStkInit()和 OSStartHighRdy()函数

如果上一步的测试完成了，就可以编译移植好的 μC/OS-II 代码，真正的测试工作就可以开始了。如这一小节标题所示，这一步要验证的是 OSTaskStkInit()和 OSStartHighRdy()函数。

使用源代码调试器测试

如果拥有源码级的调试工具，那么这一步将会很快完成。我们假设用户熟悉自己使用的调试器。

首先，修改 OS_CFG.H 文件，设置 OS_TASK_STAT_EN 为 0，以禁止统计任务。因为

用户的 TEST.C (见程序清单 L13.20)并没有添加任何应用任务,所以惟一的任务是 μC/OS-II 的空闲任务:OS_TaskIdle()。可以单步执行,直到 μC/OS-II 切换到 OS_TaskIdle()。

应该将代码下载到调试器,单步执行 main()程序。跳过 OSInit()函数,单步进入 OSStart()函数(见程序清单 L13.21)。一直单步运行到调用 OSStartHighRdy()(这是 OSStart()函数的最后一句),然后单步进入 OSStartHighRdy()。这时,编译器应该切换到汇编语言模式下,因为 OSStartHighRdy()是用汇编语句实现的。OSStartHighRdy()会开始运行第1个任务;而此时并没有任何应用任务,只有 OS_TaskIdle()可以运行。

程序清单 L13.21 OSStart()

```
void  OSStart (void)
{
    INT8U y;
    INT8U x;


    if (OSRunning == FALSE) {
        y             = OSUnMapTbl[OSRdyGrp];
        x             = OSUnMapTbl[OSRdyTbl[y]];
        OSPrioHighRdy = (INT8U)((y << 3) + x);
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
        OSTCBCur      = OSTCBHighRdy;
        OSStartHighRdy();
    }
}
```

继续单步执行,同时检查是否出错。实际上,OSStartHighRdy()会将 OSTaskStkInit()推入堆栈的 CPU 寄存器,并按照相反的顺序弹出。如果这一点不正确,堆栈指针就会出错。这时应该校正 OSTaskStkInit()函数(见 OS_CPU_C.C)。OSStartHighRdy()的最后一条语句会从中断中返回。一旦执行这条语句,调试器就应该指向 OS_TaskIdle()的第1条指令。而如果这一步没有发生,那么可能是因为没有将正确的任务起始地址放在任务堆栈中。这时很可能须修改 OSTaskStkInit()函数。如果调试器在 OS_TaskIdle()的循环中运行,且在无限循环中已经执行几次,那么就已经验证了 OSTaskStkInit()和 OSStartHighRdy()函数是成功的。

运行/不运行测试法(Go/No Go)

如果没有源代码级的调试器,但是在目标系统上有发光二极管(LED),则可以使用运行/不运行的测试方法。也就是先关闭 LED,如果 OSTaskStkInit()和 OSStartHighRdy()函数工作正常,再由 OS_TaskIdle()函数点亮 LED。实际上,因为 LED 亮和灭的速度特别快,用户只会看到 LED 一直是亮的。这时使用示波器可以观察到,LED 大约是以 50%的占空比闪烁的。

为了进行测试,应该临时修改3个文件:OS_CFG.H,OS_CPU_C.C 及 TEST.C。在

OS_CFG.H中，须设 OS_TASK_STAT_EN 为 0，以禁止统计任务；在 TEST.C 中，须添加代码，以关闭 LED(见程序清单 L13.22)；而在 OS_CPU_C.C 中，须修改 OSTaskIdleHook()，以不断翻转 LED(示意性代码见程序清单 L13.23)。

接下来，就是将代码下载到用户的目标系统中进行测试。如果 LED 闪烁不正常，则说明 OSTaskStkInit()和 OSStartHighRdy()函数不能正常工作。因为使用的调试工具太简单，功能有限了。此时只能非常细心地检查程序代码，直到找出错在哪里。

程序清单 L13.22　在 TEST.C 中更改 main()

```
#include    "includes.h"


void  main (void)
{
    OSInit();
    关 LED;
    OSStart();
}
```

程序清单 L13.23　在 OS_CPU_C.C 中修改 OSTaskIdleHook()

```
void  OSTaskIdleHook (void)
{
    if (LED is ON) {                /* 翻转 LED */
        关 LED;
    } else {
        开 LED;
    }
}
```

13.06.03　验证 OSCtxSw()函数

如果上一步测试成功，这一步代码验证就比较容易了。因为已经知道由 OSTaskStkInit()初始化的堆栈结构是正确的。在这一步测试中，添加了一个应用程序，并不断切换到空闲任务。在这之前，应该保证已经正确设置了软中断向量或指令陷阱 TRAP 向量，使之指向 OSCtxSw()函数。

使用源代码调试器测试

如程序清单 L13.24 所列，可以从修改 TEST.C 中的 main()函数开始。为了便于讨论，假设处理器的堆栈是由高地址向低地址方向递减的，而且测试任务有 100 个(字节、字、或者长字，取决于处理器)堆栈空间就足够了。当然，可以根据自己处理器的情况修改这部分代码。

程序清单 L13.24 使用调试器测试 OSCtxSw()

```
#include    "includes.h"
OS_STK  TestTaskStk[100];

void  main (void)
{
    OSInit();
    OSTaskCreate(TestTask, (void *)0, &TestTaskStk[99], 0);                    (1)
    OSStart();
}


void  TestTask (void *pdata)                                                   (2)
{
    pdata = pdata;
    while (1) {
        OSTimeDly(1);                                                          (3)
    }
}
```

L13.24(1) 建立一个高优先级的任务。这里用了优先级0，但是也可以使用其他任何小于OS_LOWEST_PRIO(见OS_CFG.H)的优先级。

L13.24(2) 因为在13.06.02小节："验证OSTaskStkInit()和OSStartHighRdy()函数"中，OSStartHighRdy()函数已经经过了验证，所以μC/OS-II应该将TestTask()作为第1个任务开始执行，而不再用空闲任务。可以单步执行，一直到TestTask()函数的开始。

L13.24(3) TestTask()进入无限循环，循环中调用OSTimeDly(1)。换句话说，TestTask()不做任何事情，只是简单地等待一段时间。因为并没有允许中断，也没有打开时钟节拍，所以OSTimeDly(1)不会返回到TestTask()。

现在可以单步执行进入OSTimeDly()函数。OSTimeDly()接着调用OS_Sched()，而OS_Sched()调用汇编语言编写的OSCtxSw()函数。多数情况下，这种调用是使用TRAP或者软中断机制完成的。换句话说，如果正确设置了TRAP或者软中断，CPU将会开始执行OSCtxSw()。可以单步进入OSCtxSw()的代码，看到TestTask()任务的寄存器已经保存到它的任务堆栈，而OS_TaskIdle()的寄存器正在被调入CPU。当从中断(软中断或者TRAP)中返回时，应该是在OS_TaskIdle()任务中。

如果OSCtxSw()并没有将用户带入OS_TaskIdle()，则应当查明原因，纠正OSCtxSw()函数中的错误。

运行/不运行测试法

同样，修改TEST.C中的main()函数(如程序清单L13.25所列)。设堆栈方向同样是递

减的，有 100 个存储单元可供任务使用。

程序清单 L13.25　使用 LED 测试 OSCtxSw()

```
#include    "includes.h"

OS_STK  TestTaskStk[100];

void  main (void)
{
    OSInit();
    关闭 LED;                                                    (1)
    OSTaskCreate(TestTask, (void *)0, &TestTaskStk[99], 0);      (2)
    OSStart();
}

void  TestTask (void *pdata)                                     (3)
{
    pdata = pdata;
    while (1) {
        OSTimeDly(1);                                            (4)
    }
}
```

L13.25(1)　在运行其他程序之前，应该关掉 LED。如果测试失败，则希望 LED 仍然是关着的。所谓希望 LED 仍然是关着的，是因为处理器运行崩溃以后，LED 可能会是亮的。然而，如果 OSCtxSw()代码是正确的，那么 LED 应该以很高的频率闪烁，这可以借助示波器观察到。

L13.25(2)　建立更高级别的任务。这里使用了优先级 0，也可以使用其他任何小于 OS_LOWEST_PRIO(见 OS_CFG.H)的优先级。

L13.25(3)　因为在 13.06.02 小节："验证 OSTaskStkInit() 和 OSStartHighRdy()函数"中，OSStartHighRdy()已经经过了验证，作为第 1 个任务，μC/OS - II 应该执行 TestTask()，而不再用空闲任务。可以单步运行，一直到 TestTask()函数的开始。

L13.25(4)　TestTask()进入无限循环，循环中调用 OSTimeDly(1)。也就是 TestTask()不做任何事情，只是简单地等待一段时间。因为没有开中断，也没有让时钟节拍开始，所以 OSTimeDly(1)不会返回到 TestTask()。当调用 OSTimeDly(1)时，会发生任务切换。如果 OSCtxSw()代码正确，LED 会快速闪烁。实际上，由于闪烁速度很快，看起来像是 LED 一直亮着，故只能用示波器观察。如果 LED 不闪烁，或者是不亮，则应查明原因，并对 OSCtxSw()做必要的修正。

13.06.04 验证 OSIntCtxSw()和 OSTickISR()函数

OSIntCtxSw()很像 OSCtxSw()，且比 OSCtxSw()简单。实际上，OSIntCtxSw()中的多数代码可从 OSCtxSw()中得到。

在此测试之前，应该保证时钟中断向量指向了时钟节拍中断服务子程序，然后，初始化时钟节拍并开中断。

程序从更改 TEST.C 中的 main()函数开始，如程序清单 L13.26 所列。

程序清单 L13.26　测试 OSIntCtxSw()和 OSTickISR()

```
#include    "includes.h"

OS_STK  TestTaskStk[100];

void  main (void)
{
    OSInit();
    关 LED;                                                          (1)
    设置时钟节拍中断向量;                                            (2)
    OSTaskCreate(TestTask, (void *)0, &TestTaskStk[99], 0);          (3)
    OSStart();
}

void  TestTask (void *pdata)                                         (4)
{
    BOOLEAN  led_state;

    pdata = pdata;
    初始化时钟节拍中断(定时器);                                      (5)
    使能中断;                                                        (6)
    led_state = FALSE;
    打开 LED;                                                        (7)
    while (1) {
        OSTimeDly(1);                                                (8)
        if (led_state == FALSE) {                                    (9)
            led_state = TRUE;
            开 LED;
        } else {
            led_state = FALSE;
            关 LED;
```

```
        }
    }
}
```

L13.26(1)　无论是否使用源代码级的调试器，对这一步测试来说，使用 LED(或者其他显示设备)显示都是非常有用的。在运行其他代码之前，先关闭 LED。

L13.26(2)　必须设置时钟节拍中断向量。此时应该参照编译器文档和处理器手册，看应该如何设置中断向量。

注意：有些处理器不支持程序运行中更改中断向量表(如 Motorola 的 68HC11，是假定向量已经放在 ROM 中)。

时钟节拍中断应该指向 OSTickISR()。

L13.26(3)　建立更高优先级的任务。这里用了优先级 0，也可以使用其他任何数值小于 OS_LOWEST_PRIO 的优先级(见 OS_CFG.H)。

L13.26(4)　因为在 13.06.02 小节："验证 OSTaskStkInit() 和 OSStartHighRdy()函数"中，OSStartHighRdy()能正常工作，所以 μC/OS-II 应将 TestTask()作为第 1 个任务开始执行。

L13.26(5)　当进入 TestTask()任务时，应该初始化产生时钟中断的相关设备(典型的如定时器)，以产生一定频率的时钟节拍中断。推荐使用 10 Hz 左右的节拍，这样可以使得 LED 以 5 Hz 的频率闪烁。节拍的速度应该与在 OS_CFG.H 中设置的 OS_TICKS_PER_SEC 吻合。

L13.26(6)　现在可以开中断了，允许时钟中断调用 OSTickISR()。

L13.26(7)　点亮 LED，表明进入了 TestTask()。

L13.26(8)　调用 OSTimeDly()，使 OSCtxSw()将任务切换到空闲任务。空闲任务一直运行，直到接收到时钟节拍中断。时钟节拍中断调用 OSTickISR()，继而调用 OSTimeTick()。OSTimeTick()将 TestTask()的 .OSTCBDly 计数器递减到 0，使该任务进入就绪态。而当 OSTickISR()完成并调用 OSIntExit()函数时，OSIntExit()将会注意到有更重要的任务，TestTask()已处于就绪态等待运行。这时 ISR 就不会再返回到空闲任务，而是做任务切换，回到 TestTask()任务。当然，以上是假设 OSIntCtxSw()和 OSTickISR()这 2 个函数都能正常工作。

L13.26(9)　如果 OSIntCtxSw()正常工作，而且已经将时钟节拍频率设置为 10 Hz，那么用户应该看到 LED 以 5 Hz 的频率闪烁。

如果 LED 不闪烁，而用户是在用调试器调试代码，则可以在 OSTickISR()函数中设置断点，看发生了什么。

建议单独运行 ISR 程序，而不调用 OSIntExit()。在这种情况下，可以看到 ISR 使 LED 闪烁。

如果此时 LED 能闪烁，则说明问题出在 OSIntCtxSw()中，而 OSIntCtxSw()源自 OSCtxSw()，所以问题可能出在 OSTickISR()。

至此，移植的 μC/OS-II 已经正常工作了，可以添加应用任务了。希望读者能从中得到乐趣。

OSCtxSw()

void OSCtxSw(void)

所属文件	调用者	
OS_CPU_A.ASM	OS_TASK_SW()	必 须

该函数用于在任务级完成任务切换。一般情况下,通过软中断指令(也叫做指令陷阱TRAP)调用该函数。该函数的示意性代码如下:

```
void  OSCtxSw (void)
{
    保存处理器的寄存器;
    把当前任务的堆栈指针存入当前任务的任务控制块 OS_TCB:
        OSTCBCur->OSTCBStkPtr = Stack pointer;
    OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    得到要恢复运行的任务的堆栈指针:
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    从新任务堆栈中恢复处理器的寄存器;
    执行中断返回指令;
}
```

参 数

无。

返回值

无。

注意/警告

1. 调用该函数时,中断是关掉的。
2. 有些编译器允许在C语言程序中直接建立软中断或指令陷阱子程序,这样,本函数就可放在OS_CPU_C.C文件中。有些编译器要求的函数声明法会有所不同,这种情况下可在INCLUDES.H文件中,用#define语句定义常数OS_ISR_PROTO_EXT,以实现用其他方法定义函数原型。即并非一定要用void OSCtxSw(void)定义函数原型。

范 例

无。

OSInitHookBegin()

void OSInitHookBegin(void)

所属文件	调用者	OS_CPU_C.C 文件中的开关量
OS_CPU_C.C	OSInit()	OS_CPU_HOOKS_EN = 1

OSInit()在一开始就会调用该函数。该函数使得用户可以完成一些 CPU(或者其他外设)的初始化，并使之成为 OSInit()函数的一部分。例如，可以在 OSInitHookBegin()函数中对 I/O 端口进行初始化。该函数可以将其他初始化代码加到 OSInit()函数中去。换句话说，移植后，μC/OS-II 的使用者不必知道还有哪些其他的初始化操作。

参　数

无。

返回值

无。

注意/警告

无。

范　例

无。

OSInitHookEnd()

void OSInitHookEnd(void)

所属文件	调用者	OS_CPU_C.C 文件中的开关量
OS_CPU_C.C	OSInit()	OS_CPU_HOOKS_EN == 1

OSInit()在结束时会调用该函数。该函数使得用户可以完成一些CPU(或者其他外设)的初始化。例如，可以在OSInitHookEnd()里对I/O端口进行初始化。该函数可以将初始化全都放在一起，移植后，μC/OS-II的使用者不必知道还有哪些其他的初始化操作。

参　数

无。

返回值

无。

注意/警告

无。

范　例

无。

OSIntCtxSw()

void OSIntCtxSw(void)

所属文件	调用者	
OS_CPU_A.ASM	OSIntExit()	必　须

该函数为 OSIntExit()所调用。当 OSIntExit()判定，由于中断服务的结果，该让更高优先级的任务运行时，就会调用该函数做任务切换。该函数的示意性代码如下：

```
void  OSIntCtxSw (void)
{
    OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    得到要恢复任务的堆栈指针：
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    从新任务栈中恢复全部 CPU 寄存器；
    执行中断返回指令；
}
```

参　数

无。

返回值

无。

注意/警告

调用该函数时中断是关掉的。

范　例

无。

OSStartHighRdy()

void OSStartHighRdy(void)

所属文件	调用者	
OS_CPU_A.ASM	OSStart()	必 须

OSStart()调用该函数，使得在OSStart()之前定义的任务中优先级最高的任务开始运行。这个函数的示意性代码如下：

```
void OSStartHighRdy (void)
{
    OSTaskSwHook();
    OSRunning = TRUE;
    得到要恢复任务的堆栈指针：
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;

    从新任务栈中恢复全部CPU寄存器；
    执行中断返回指令；
}
void OSStartHighRdy (void)
```

参 数

无。

返回值

无。

注意/警告

调用该函数时中断是关掉的。

范 例

无。

OSTaskCreateHook( )

void OSTaskCreateHook(OS_TCB *ptcb)

所属文件	调用者	OS_CPU_C.C 文件中的开关量
OS_CPU_C.C	OSTaskCreate() OSTaskCreateExt()	OS_CPU_HOOKS_EN == 1

无论是否建立了任务，在分配并初始化了任务控制块 TCB、且任务栈结构也已经初始化完毕后，就可以调用该函数。OSTaskCreateHook()使用户可以按自己的方式来扩展任务、建立函数的功能。例如可以初始化并存储与任务相关的浮点寄存器、内存管理单元 MMU 的寄存器，以及做一些与拟建立的任务相关的事。用户应注意，OSTaskCreateHook()是在调用另一个名为 OSTCBInitHook()的接口函数后立即被调用的。换句话说，这 2 个函数都可初始化任务控制块；然而，应该用 OSTCBInitHook()做与任务控制块相关的事，用 OSTaskCreateHook()做与任务相关的事。还可以通过使用 OSTaskCreateHook()触发示波器或逻辑分析仪，以及设置断点等。

参　数

ptcb　指向建立任务的任务控制块的指针。

返回值

无。

注意/警告

当调用该函数时，中断是开着的，因此，可能需要调用 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()，以保护 OSTaskCreateHook()函数中的临界段代码。

范　例

该范例假定是用 OSTaskCreateExt()建立任务的。因为它希望在任务控制块 OS_TCB 中有 .OSTCBExtPtr 域。该域有一个指针变量，指向用于保存浮点寄存器内容的存储区。

```
void OSTaskCreateHook (OS_TCB *ptcb)
{
    if (ptcb->OSTCBExtPtr != (void *)0) {
        /* 储存浮点寄存器的内容到.. */
        /* ..TCB 的扩展域中          */
    }
}
```

OSTaskDelHook()

void OSTaskDelHook(OS_TCB *ptcb)

所属文件	调用者	OS_CPU_C.C 文件中的开关量
OS_CPU_C.C	OSTaskDel()	OS_CPU_HOOKS_EN == 1

当调用 OSTaskDel()函数删除任务时，会用到该函数。这样就可以处置建立任务接口函数 OSTaskCreateHook()分配给这个拟删除任务的内存。OSTaskDelHook()是在被删除任务的任务控制块 TCB 从 TCB 链表中移除之前调用的。还可以用 OSTaskDelHook()* 触发示波器或逻辑分析仪，以及设置断点。

参 数

ptcb 指向拟删除任务的任务控制块的指针。

返回值

无。

注意/警告

该函数在被调用时，中断是关掉的。因此，应将该函数中的代码减至最少，以缩短中断的响应时间。

范 例

```
void OSTaskDelHook (OS_TCB *ptcb)
{
    /* 输出信号触发示波器 */
}
```

* 原文为 OSTaskCreateHook()。——译者注

OSTaskIdleHook( )

void OSTaskIdleHook(void)

所属文件	调用者	OS_CPU_C.C 文件中的开关量
OS_CPU_C.C	OSTaskIdle()	OS_CPU_HOOKS_EN==1

空闲任务 OS_TaskIdle()调用该函数。当没有其他高优先级的任务运行时，会运行空闲任务。对于电池充电的产品，为了省电，在没有其他任务运行时，OSTaskIdleHook()可以用来使 CPU 进入低功耗模式。

参　数

无。

返回值

无。

注意/警告

该函数在被调用时，中断是开着的。

范　例

```
void OSTaskIdleHook (void)
{
    /* 使 CPU 进入低功耗模式 */
}
```

OSTaskStatHook()

void OSTaskStatHook(void)

所属文件	调用者	OS_CPUC.C 文件中的开关量
OS_CPU_C.C	OSTaskStat()	OS_CPU_HOOKS_EN==1

该函数每秒都会被 μC/OS-II 的统计任务调用。OSTaskStatHook()允许用户加入自己的统计功能。

参 数

无。

返回值

无。

注意/警告

统计任务大概在调用 OSStart()函数 5 s 之后开始执行。

注意:当 OS_TASK_STAT_EN 或者 OS_TASK_CREATE_EXT_EN 被置为 0 时,该函数不会被调用。

范 例

```
void OSTaskStatHook (void)
{
    /* 计算所有任务总的执行时间 */
    /* 计算每个任务的执行时间在总时间内所占的百分比 */
}
```

OSTaskStkInit()

```
OS_STK *OSTaskStkInit(void( *task)(void *pd),
                      void *pdata,
                      OS_STK *ptos,
                      INT16U opt);
```

所属文件	调用者	
OS_CPU_C.C	OSTaskCreate()或者 OSTaskCreateExt()	必　须

OSTaskCreate()或者 OSTaskCreateExt()调用该函数初始化任务的堆栈结构。一般地说，为了使堆栈结构看起来像刚刚发生过中断一样，CPU 寄存器已经保存在堆栈中。这个函数的示意性代码如下：

```
OS_STK * OSTaskStkInit (void   ( *task)(void *pd),
                        void    *pdata,
                        OS_STK  *ptos,
                        INT16U   opt);
{
    模拟带参数(如 pdata)的函数调用;
    模拟 ISR 向量;
    按照预先设计的寄存器值初始化堆栈结构;
    返回新的栈顶指针给调用该函数的函数;
}
```

参　数

task　指向任务代码的指针(定义任务的入口地址)。

pdata　指向用户数据区的指针，任务第 1 次执行时会被传递给任务。有时编译器会使用寄存器传递 pdata，而有时会使用堆栈来传递。应参考 C 编译器手册，确认使用何种方式。

ptos　指向堆栈顶部的指针。ptos 指向一个空任务堆栈的入口。若 OS_STK_GROWTH 置为 1，则 ptos 指向堆栈的最高有效地址；如果 OS_STK_GROWTH 设为 0，则 ptos 指向堆栈的最低有效地址。

opt　可选参数。可以用来改变 OSTaskStkInit()的行为。参见 uCOS_II.H 文件中的 OS_TASK_OPT_???。

返回值

指向新的堆栈顶部的指针。

注意/警告

当该函数被调用时，中断是开着的。

范　例

无。

OSTaskSwHook( )

void OSTaskSwHook(void)

所属文件	调用者	OS_CPU_C.C 文件中的开关量
OS_CPU_C.C	OSCtxSw()和 OSIntCtxSw()	OS_CPU_HOOKS_EN==1

执行任务切换时调用该函数。全局变量 OSTCBHighRdy 指向得到 CPU 的任务的任务控制块 TCB，而 OSTCBCur 指向被挂起任务的任务控制块 TCB。OSTaskSwHook()在保存好任务的寄存器和指向当前任务 TCB 的堆栈指针后，马上被调用。可以用该函数保存或恢复浮点寄存器，或者 MMU 寄存器的内容，以跟踪任务的执行时间和任务切换的次数等等。OSTaskSwHook()函数也被 OSStartHighRdy()调用，因此应该在 OSTaskSwHook()中检查 OSRunning 标志。如果任务不是被挂起，则无须做什么(见范例)。

参　数

无。

返回值

无。

注意/警告

该函数在被调用时，中断是禁止的；因此，应将该函数中的代码减至最少，因为其影响中断响应时间。

范　例

```
void OSTaskSwHook (void)
{
    If (OSRunning == TRUE) {
    /* 将浮点寄存器的内容储存到当前任务的 TCB 扩展域中 */
    }
    /* 用新任务的 TCB 扩展域中的值恢复浮点寄存器的内容 */
}
```

OSTCBInitHook()

void OSTCBInitHook(void)

所属文件	调用者	OS_CPU_C.C 文件中的开关量
OS_CPU_C.C	OS_TCBInit()	OS_CPU_HOOKS_EN == 1

无论是否建立了任务，在分配并初始化了任务控制块 TCB、且任务栈结构也已经初始化完毕后，就可以调用该函数。OSTCBInitHook()使用户可以按自己的方式扩展建立任务控制块的函数。比如可以初始化和保存浮点寄存器、MMU 寄存器，以及做一些与拟建立的任务相关的事，例如，初始化分配给每个任务的内存。应该注意，调用 OSTCBInitHook()函数会在 OSTaskCreate()之前，也就是说，这 2 个函数都可以用来初始化 TCB；但是，应该尽量用 OSTCBInitHook()处理与 TCB 相关的事项，而用 OSTaskCreate()函数处理与任务相关的事项。

参 数

ptcb 指向所建立任务的任务控制块 TCB 的指针。

返回值

无。

注意/警告

调用该函数时，中断是开着的。应调用 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()，以保护 OSTCBInitHook()中的临界段代码。

范 例

这个例子假设已经使用 OSTaskCreateExt()函数建立了一个任务，而且该函数认为这个任务的任务控制块 TCB 中有一个 .OSTCBExtPtr 域是指向保存浮点寄存器内容的存储区的。

```
void OSTCBInitHook (OS_TCB *ptcb)
{
    if (ptcb->OSTCBExtPtr != (void *)0) {
        /* 保存浮点寄存器的内容.. */
        /* .. 到扩展的 TCB 中      */
    }
}
```

OSTickISR()

void OSTickISR(void)

所属文件	调用者	
OS_CPU_A.ASM	Tick Interrupt	必　须

发生节拍中断时，CPU 的中断向量会指向该 ISR。节拍中断服务程序的示意性代码为：

```
void  OSTickISR (void)
{
    保存处理器寄存器；
    调用 OSIntEnter() 或者递增 OSIntNesting；
    if (OSIntNesting == 1)
       OSTCBCur->OSTCBStkPtr = Stack Pointer;
    }
    清除设备中断标志位；
    重新使能中断(可选)；
    OSTimeTick();
    OSIntExit();
    恢复处理器寄存器；
    执行中断返回；
}
```

参　数

无。

返回值

无。

注意/警告

1. 一般情况下，产生调用 OSTickISR()函数时钟节拍的设备，应设置成每隔 10～100 ms 产生 1 次中断。
2. 某些编译器支持使用 C 语言添加中断服务程序，这样可以把该函数放在 OS_CPU_C.C 文件中。在某些情况下，编译器还要求声明不同的函数原型。这时，可在 INCLUDES.H 中用"#define constant OS_ISR_PROTO_EXT"声明。这样，可声明不同的 OSTickISR()函数。也就是说，不一定非要使用"void OSTickISR(void)"的原型函数。

范　例

无。

OSTimeTickHook()

void OSTimeTickHook(void)

所属文件	调用者	OS_CPU_C.C 文件中的开关量
OS_CPU_C.C	OSTimeTick()	OS_CPU_HOOKS_EN == 1

只要发生时钟节拍，该函数就会被 OSTimeTick()调用。一旦进入 OSTimeTick()，就会马上调用 OSTimeTickHook()，以允许执行应用程序中的与时间密切相关的代码。还可以通过使用该函数，触发示波器或逻辑分析仪来调试，或者为仿真器设置断点。

参 数

无。

返回值

无。

注意/警告

OSTimeTick()通常是被 ISR 调用的，所以时钟节拍 ISR 的执行时间会因在该函数中提供的代码而增加。当 OSTimeTick()被调用时，中断可以是禁止的，也可以是允许的，主要取决于该处理器上的移植是怎样进行的。如果中断是禁止的，则该函数将会影响到中断响应时间。

范 例

```
void OSTimeTickHook (void)
{
    /* 触发示波器 */
}
```

第 14 章

μC/OS－II 在 80x86 上的移植

实模式，在大模式下编译，支持仿真浮点数运算

本章介绍如何将 μC/OS－II 移植到 Intel 80x86 系列的 CPU 上。所介绍的代码移植是针对 80x86 的实模式的，且使用 Borland C++ V4.51 编译工具在大模式下编译和链接。本移植假定应用程序不做浮点数数学运算。如果做，则使用 Borland 浮点运算仿真库。换句话说，假定是移植到嵌入式 80186，80286，80386 或者普通 8086 类型的处理器，这些处理器内部没有浮点处理单元，只能基于整数运算。本移植也适用于普通 8086 处理器，但需正确地用 PUSH/POP 指令替换 PUSHA/POPA 指令。

Intel 80x86 系列包括 80186，80286，80386，80486，Pentiums™（所有型号）以及 Celeron，也包括 AMD，NEC（V 系列）等生产的多数 80x86 处理器。80x86 CPU 每年的产量有数百万颗，大部分用于个人计算机，但用于嵌入式系统的数量也在不断增加。预计到 2005 年，最快的处理器将达到 10 GHz。

大多数支持 80x86（实模式）的 C 编译器都提供了不同的内存使用模式。每种模式都有各自的程序和数据容量，适用于不同规模的应用程序。在大模式下，应用程序和数据的最大寻址空间为 1 MB。虽然寻址空间只有 1 MB，却需要 32 位的指针变量。下一节将介绍为什么 32 位指针只用到了其中的 20 位来寻址（1 MB）。

图 F14.1 显示了工作在实模式下的 80x86 处理器的编程模式。所有的寄存器都是 16 位，在任务切换时，须保存所有寄存器内容。如图 F14.1 所示，没有浮点寄存器，浮点运算是使用 Borland 编译库，通过整数寄存器仿真来实现的。

80x86 提供了一种特殊的机制，使得用 16 位寄存器可以寻址 1 MB 地址空间，这就是存储器分段的方法。内存的物理地址用段地址寄存器和偏移量寄存器共同表示。计算方法是：段地址寄存器的内容左移 4 位（乘以 16），再加上偏移量寄存器（其他 5 个寄存器中的 1 个，即 BP，SP，SI，DI 或 IP）的内容，产生可寻址 1 MB 的 20 位物理地址。图 F14.2 表明了寄存器是如何组合的。段寄存器可以指向一个内存块，称为一段。一个 16 位的段寄存器可以表示 65 536个不同的段，因此可以寻址 1 048 576 B。由于偏移量寄存器也是 16 位的，所以单个段不能超过 64 KB。实际操作中，应用程序是由许多小于 64 KB 的段组成的。

代码段地址寄存器（CS）指向当前程序运行的码段基地址，堆栈段寄存器（SS）指向程序堆栈段基地址，数据段寄存器（DS）指向程序数据区基地址，附加段寄存器（ES）指向附加数据存储区基地址。每次 CPU 寻址时，段寄存器中的某一个会被自动选用，加上偏移量寄存器的内

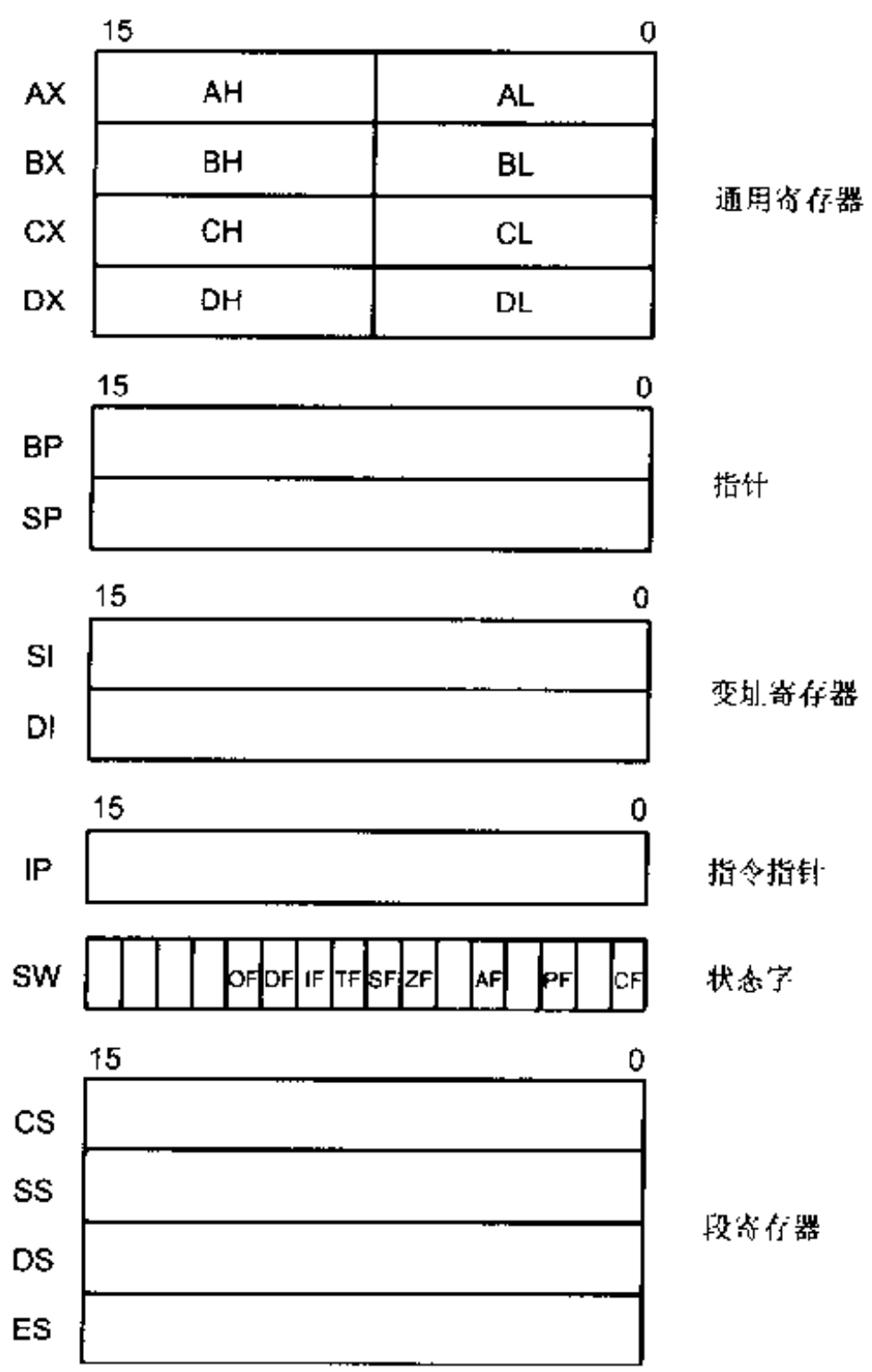


图 F14.1 80x86 实模式内部寄存器图

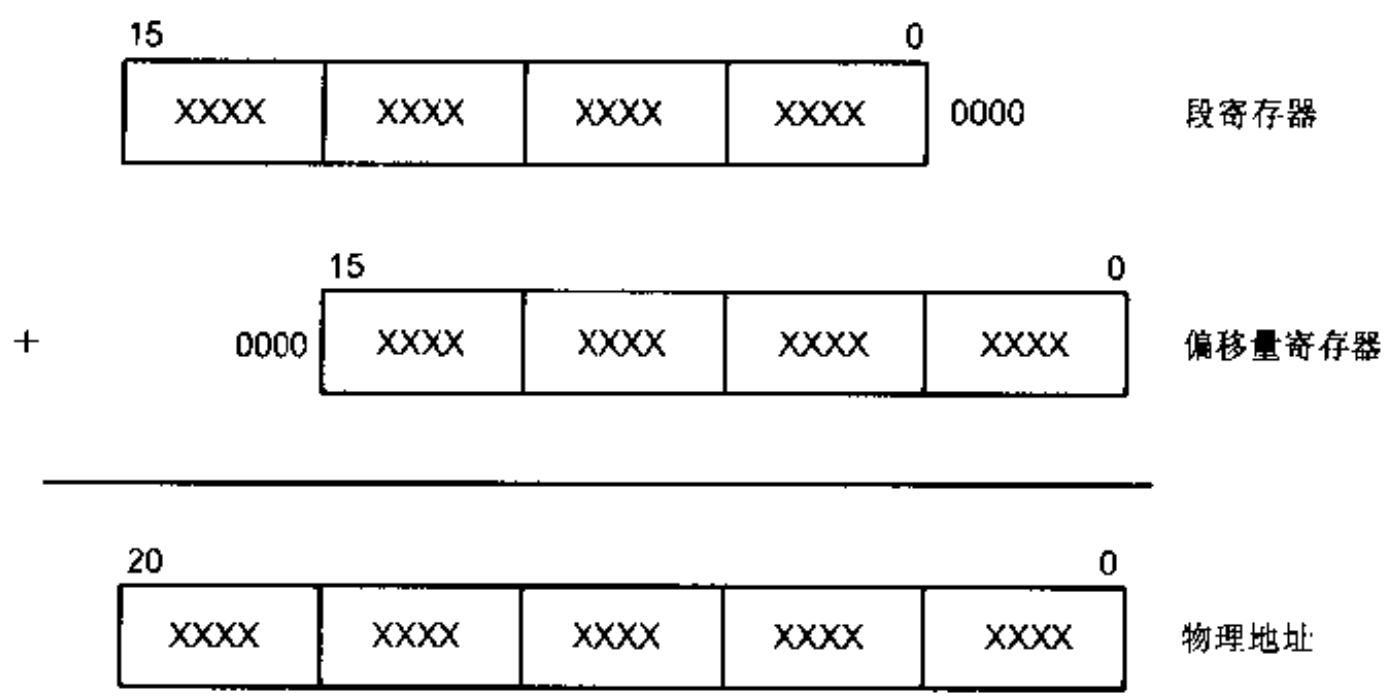


图 F14.2 用段地址寄存器和偏移量寄存器寻址

容，作为物理地址。文献中会经常出现用段地址:偏移量表示地址的方法，例如 1000:00FF 表示物理地址 0x100FF。

14.00 开发工具

采用 Borland C/C++ V4.51 的 C 编译器和 Borland Turbo Assembler 汇编器，完成程序的移植和测试。这个编译器可产生可重入型代码，同时支持在 C 语言程序中嵌入汇编语句。编译器带浮点运算仿真库，可以仿真具备浮点运算能力的 80x86 处理器上的硬件浮点运算单元。编译完成后，程序可在 PC 机上运行。本书代码的测试是在一台 300 MHz 的 Pentium II计算机上完成的，操作系统是 Microsoft Windows 2000。实际上编译器生成的是 DOS 下的可执行文件，在 Windows 的 DOS 窗口中运行。

笔者曾考虑换个编译器，因为一些读者抱怨找不到 Borland 工具，使他们很难编译书中提供的范例代码。现在 Borland 提供了一款类似的编译器和汇编器，也能编译范例中的代码，仅需 70 $(2002 年前后)。Borland 称之为"Turbo C++ Suite for DOS"，可通过访问 Borland 公司的网站 www.Borland.com，根据相应链接，网上订购。

也可以访问 Paradigm 网站(www.DevTools.com)，得到专业级的 80x86 开发工具。它不仅包括了与 Borland 兼容的编译器和汇编器，还包括集成开发环境 IDE，能在嵌入式系统开发过程中由用户定位应用代码，实现源代码级的调试，还有其他一些功能。Paradigm 称他们的软件包为 Paradigm C++ Professional Real。

还可以修改本章提供的移植代码，使之适合别的 80x86 编译器，只要这些编译器可以生成实模式代码。如果开发环境不同，则很可能需要修改一下编译器和汇编器的设置。

表 T14.1 是命令行提供的 Borland C/C++编译器版本 V4.51 的编译选项。这些设置可用来编译移植代码以及第 1 章提供的范例代码。

表 T14.1 编译移植和范例的编译选项

选 项	描 述
-1	产生 80186 代码
-B	若有在线汇编语句，则编译并调用汇编程序处理
-c	编译生成 .OBJ 文件，不链接
-G	生成速度最优化代码
-I	编译器包含文件路径为 C:\BC45\INCLUDE
-k-	采用标准栈结构
-L	编译器库文件路径为 C:\BC45\LIB
-ml	大模式下编译
-N-	不检查堆栈溢出
-n..\obj	存放目标文件路径为 ..\OBJ
-O	优化跳转语句

续表 T14.1

选 项	描 述
-Ob	删除无用代码
-Oe	分配寄存器给全局变量，并分析变量作用期
-Og	全程优化，整个函数内消除重复子表达式
-Oi	扩展公共内联函数
-Ol	循环优化
-Om	恒定代码转移，将不变代码转移到循环体外
-Op	复制传播，即复制相同的常量、变量及表达式
-Ov	归纳变量，削减计算强度
-v	打开源码调试
-vi	内联函数展开
-wpro	错误报告：无原型函数调用
-Z	寄存器优化

表 T14.2 是命令行提供的 Borland Turbo Assembler V4.0 汇编器的选项。这些设置用于汇编移植代码的 OS_CPU_A.ASM 文件。

表 T14.2 汇编 .ASM 文件时使用的汇编选项

选 项	描 述
/MX	全局变量和标号区分大小写
/ZI	目标文件包含全部调试信息
/O	生成代码覆盖同名文件

14.01 目录和文件

安装 μC/OS-II 时，附带光盘中的安装程序将把针对 Intel 80x86（实模式，大模式下编译）的移植代码安装到 \SOFTWARE\uCOS-II\1x86L\BC45 目录下。目录名称代表 Intel 80x86 实模式，大模式 L 下编译，放在 BC45 目录下。移植部分的代码可在下述文件中找到：OS_CPU.H、OS_CPU_C.C 及 OS_CPU_A.ASM。

14.02 INCLUDES.H 文件

INCLUDES.H 是主头文件，在所有后缀名为.C 的文件的开始处都包含 INCLUDES.H 文件。使用 INCLUDES.H 的好处是，项目中所有的 .C 文件都只须包含这一个头文件，而不用具体考虑哪个 .C 文件须包含哪几个头文件，程序简洁，可读性强；缺点是，.C 文件可能会

包含一些并不需要的头文件，额外地增加了编译时间。与代码的可移植性相比，多花一些时间编译还是可以接受的。用户可以改写 INCLUDES.H 文件，增加自己的头文件，但必须加在文件末尾。程序清单 L14.1 是为 80x86 编写的 INCLUDES.H 文件的内容。

INCLUDES.H 不涉及真正移植部分的内容，但由于在编译移植代码时要用到它，所以仍在此说明。

程序清单 L14.1　INCLUDES.H

```
#include    <stdio.h>
#include    <string.h>
#include    <ctype.h>
#include    <stdlib.h>
#include    <conio.h>
#include    <dos.h>
#include    <math.h>
#include    <setjmp.h>

#include    "os_cpu.h"
#include    "os_cfg.h"
#include    "ucos_ii.h"
#include    "pc.h"
```

14.03　OS_CPU.H 文件

OS_CPU.H 文件中有与处理器相关的常量、宏及结构体的定义。程序清单 L14.2 是为 80x86 编写的 OS_CPU.H 文件的内容。

OS_CPU_GLOBALS 和 OS_CPU_EXT 允许用户声明针对本移植的全局变量。

程序清单 L14.2　OS_CPU.H

```
#ifdef  OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT  extern
#endif

typedef  unsigned  char  BOOLEAN;                                    (1)
typedef  unsigned  char  INT8U;
typedef  signed    char  INT8S;
typedef  unsigned  int   INT16U;
typedef  signed    int   INT16S;
```

```
typedef unsigned long  INT32U;
typedef signed   long  INT32S;
typedef float          FP32;                                    (2)
typedef double         FP64;

typedef unsigned int   OS_STK;                                  (3)
typedef unsigned short OS_CPU_SR;                               (4)

#define  BYTE          INT8S                                    (5)
#define  UBYTE         INT8U
#define  WORD          INT16S
#define  UWORD         INT16U
#define  LONG          INT32S
#define  ULONG         INT32U
```

14.03.01 OS_CPU.H,数据类型

L14.2(1) 如果参考 Borland 编译器的文档，则会发现 int 型是 16 位，long 型是 32 位。

L14.2(2) 浮点数据类型也在其中，虽然 μC/OS-II 不使用浮点数。

L14.2(3) 实模式运行的 80x86 处理器堆栈入口为 16 位宽，OS_STK 就是据此声明的。所有的任务堆栈必须以 OS_STK 为数据类型。

L14.2(4) 实模式下运行的 80x86 处理器的状态寄存器(也称为处理器标志)也是 16 位宽。

OS_CPU_SR 数据类型只在关中断方式 OS_CRITICAL_METHOD 为 3 时使用，在该移植中不使用。

包含了 OS_CPU_SR 数据类型，是为了用户在使用不同编译器时，可能会用到 OS_CRITICAL_METHOD 为 3。

L14.2(5) 包含了该数据类型，是为了与老版本 μC/OS V1.xx 的程序向下兼容。如果不使用 μC/OS V1.xx 的程序，这些不是必需的(可以删掉这些行)。

14.03.02 OS_CPU.H,OS_ENTER_CRITICAL()及 OS_EXIT_CRITICAL()

程序清单 L14.2 OS_CPU.H(续)

```
#define  OS_CRITICAL_METHOD     2                               (6)

#if      OS_CRITICAL_METHOD == 1
#define  OS_ENTER_CRITICAL()  asm  CLI                          (7)
#define  OS_EXIT_CRITICAL()   asm  STI
#endif

#if      OS_CRITICAL_METHOD == 2
#define  OS_ENTER_CRITICAL()  asm  {PUSHF; CLI}                 (8)
#define  OS_EXIT_CRITICAL()   asm  POPF
```

```
#endif

#if      OS_CRITICAL_METHOD == 3
#define  OS_ENTER_CRITICAL()  (cpu_sr = OSCPUSaveSR())                    (9)
#define  OS_EXIT_CRITICAL()   (OSCPURestoreSR(cpu_sr))
#endif

#if      OS_CRITICAL_METHOD == 3                                          (10)
OS_CPU_SR OSCPUSaveSR(void);
void      OSCPURestoreSR(OS_CPU_SR cpu_sr);
#endif
```

L14.2(6) 与别的实时内核一样，μC/OS-II 需要关中断来访问临界段代码，完成后再开中断。因为 Borland 编译器支持交叉汇编语言，这使得开/关中断非常容易。μC/OS-II 定义了 2 条宏，分别关中断和开中断：OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()。笔者实际上允许用户使用 3 种方法中的 1 种来开/关中断。针对该移植，首选的是第 2 种方法，因为它是编译器直接支持的。

OS_CRITICAL_METHOD = 1

L14.2(7) 实现这 2 个宏的第 1 种方法，也是最简单的方法。执行处理器指令，与 OS_ENTER_CRITICAL() 关中断对应的是 CLI 指令，与 OS_EXIT_CRITICAL()开中断对应的是 STI 指令。

OS_CRITICAL_METHOD = 2

L14.2(8) 实现 OS_ENTER_CRITICAL()的第 2 种方法。将原中断状态保存到堆栈中，然后关中断。这一步在 80x86 上是通过执行 PUSHF 指令和 CLI 指令实现的。OS_EXIT_CRITICAL()只需简单地执行 POPF 指令，以恢复处理器中 SW 寄存器原来的内容。

OS_CRITICAL_METHOD = 3

L14.2(9) 实现 OS_ENTER_CRITICAL()的第 3 种方法。写一个函数，将 CPU 的状态寄存器保存到局部变量中。OS_EXIT_CRITICAL()调用另一个函数，从这个局部变量中恢复状态寄存器。本移植中未包含这些代码，但如果熟悉汇编语言，则很容易写出这些函数。

L14.2(10) 推荐分别在 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()中调用相应的 OSCPUSaveSR()和 OSCPURestoreSR()。应该在 OS_CPU_A.ASM 中增加这 2 个函数。

14.03.03 OS_CPU.H,堆栈方向

程序清单 L14.2 OS_CPU.H(续)

```
#define  OS_STK_GROWTH        1                                    (11)
```

L14.2(11) 80x86 处理器的堆栈从内存高地址向低地址递减,故 OS_STK_GROWTH 必须置为 1。

14.03.04 OS_CPU.H 和 OS_TASK_SW()

程序清单 L14.2 OS_CPU.H(续)

```
#define  uCOS                 0x80                                 (12)

#define  OS_TASK_SW()         asm  INT  uCOS                       (13)
```

L14.2(13) 为了实现任务切换,OS_TASK_SW()需要模仿一次中断过程。80x86 有 256 个软中断可供选用。中断服务子程序 ISR(也称为异常处理)的入口地址必须指向汇编函数 OSCtxSw()(见 OS_CPU_A.ASM 文件)。这里需要确认 0x80 中断向量指向 OSCtxSw()。

L14.2(12) 笔者在 PC 机上做了测试,本章的代码用到了向量号为 128(0x80)的中断,因为此中断是提供给用户使用的。实际上,最初的 PC 将 0x80~0xF0 的中断给 BASIC 解释程序使用,后来几乎再没有内植 BASIC 解释程序的 PC 机了,所以用这些中断向量是安全的。类似地,可用的中断号还有 0x4B~0x5B, 0x5D~0x66 以及 0x68~0x6F。如果用的不是 PC,而是其他嵌入式系统,如 80186 处理器,则可能有更多的中断资源可供选用。

14.03.05 OS_CPU.H,时钟节拍频率

实时系统中时钟节拍的频率应该设置为 10~100 Hz。通常(但不是必须)为了方便计算,设为整数。不幸的是,在 PC 中,系统默认的时钟节拍频率是 18.206 48 Hz,这对于计算和设置都不方便。本章中,将 PC 的时钟节拍频率从标准的 18.206 48 Hz 改为 200 Hz(间隔 5 ms)。这样做的原因有 3 个:

- 200 Hz 近似 18.206 48 Hz 的 11 倍,可以经过 11 次延时,再调用 DOS 中断。在 DOS 中,时钟节拍处理程序要求每隔 54.93 ms 进行一次有关系统维护的操作。
- 设定的间隔 5 ms 对于时间延迟和超时设置很有用。若 PC 机处理器是 80386,那么时钟节拍最快也只能到 200 Hz;而如果是 Pentium II 处理器,则很容易达到 200 Hz 以上。
- 虽然将时钟节拍频率设定为 20 Hz 或者 100 Hz 也是可以的,但是这样将难以产生 18.206 48 Hz 的 DOS 中断。这就是选择整数倍频,即选择 200 Hz 的原因。当然也可以选择 22 倍频,这样就是 400 Hz(间隔 2.5 ms)。在一台更快的 PC 上,使用这种时钟

节拍频率甚至更高的频率都不会有问题。

程序清单 L14.2 OS_CPU.H(续)

```
OS_CPU_EXT  INT8U  OSTickDOSCtr;                                    (14)
```

L14.2(14) 在这里声明了一个 8 位变量 OSTickDOSCtr,用来保存时钟节拍发生的次数。每发生 11 次,调用 1 次 DOS 的时钟节拍函数,从而实现与 DOS 时钟的同步。OSTickDOSCtr 在 OS_CPU_A.ASM 中使用,是专门为 PC 环境而定义的。如果在其他非 PC 的嵌入式系统中运行 μC/OS-II,就不必使用这种方法,直接设定时钟节拍频率就行了。

14.03.06 OS_CPU.H,浮点仿真

如前面所述,Borland 编译器提供了一个浮点仿真库。但是该库不具备可重入性。

程序清单 L14.2 OS_CPU.H(续)

```
void      OSTaskStkInit_FPE_x86(OS_STK **pptos, OS_STK **ppbos, INT32U *psize);      (15)
```

L14.2(15) 增加了一个函数,使用户可以预处理任务的堆栈。这样使 Borland C 认为只有一个任务在使用浮点仿真库,于是让浮点仿真库具备可重入性。该函数的详细说明见 14.04.02 小节:"浮点仿真任务栈初始化函数 OSTaskStkInit_FPE_x86()"。

14.04 OS_CPU_C.C 文件

μC/OS-II 的移植须改写 10 个非常简单的 C 函数:

OSTaskStkInit();

OSTaskCreateHook();

OSTaskDelHook();

OSTaskSwHook();

OSTaskIdleHook();

OSTaskStatHook();

OSTimeTickHook();

OSInitHookBegin();

OSInitHookEnd();

OSTCBInitHook()。

只有 OSTaskStkInit()函数在 μC/OS-II 中是必要的。另外 9 个函数必须声明,但可不包含任何代码。在本移植中就是这么做的。用 #define 语句定义的常数 OS_CPU_HOOKS_EN(见 OS_CFG.H)应该置为 1。

14.04.01 任务堆栈初始化函数 OSTaskStkInit()

该函数由 OSTaskCreate()或 OSTaskCreateExt()调用,用来初始化任务堆栈。将堆栈的结构初始化成看起来好像刚刚发生过中断一样,处理器的所有寄存器被推入堆栈。图 F14.3 给出了 OSTaskStkInit()初始化后的堆栈结构。

注意:图中示意的不是调用 OSTaskStkInit()函数的任务的堆栈结构,而是新建立任务的堆栈结构。

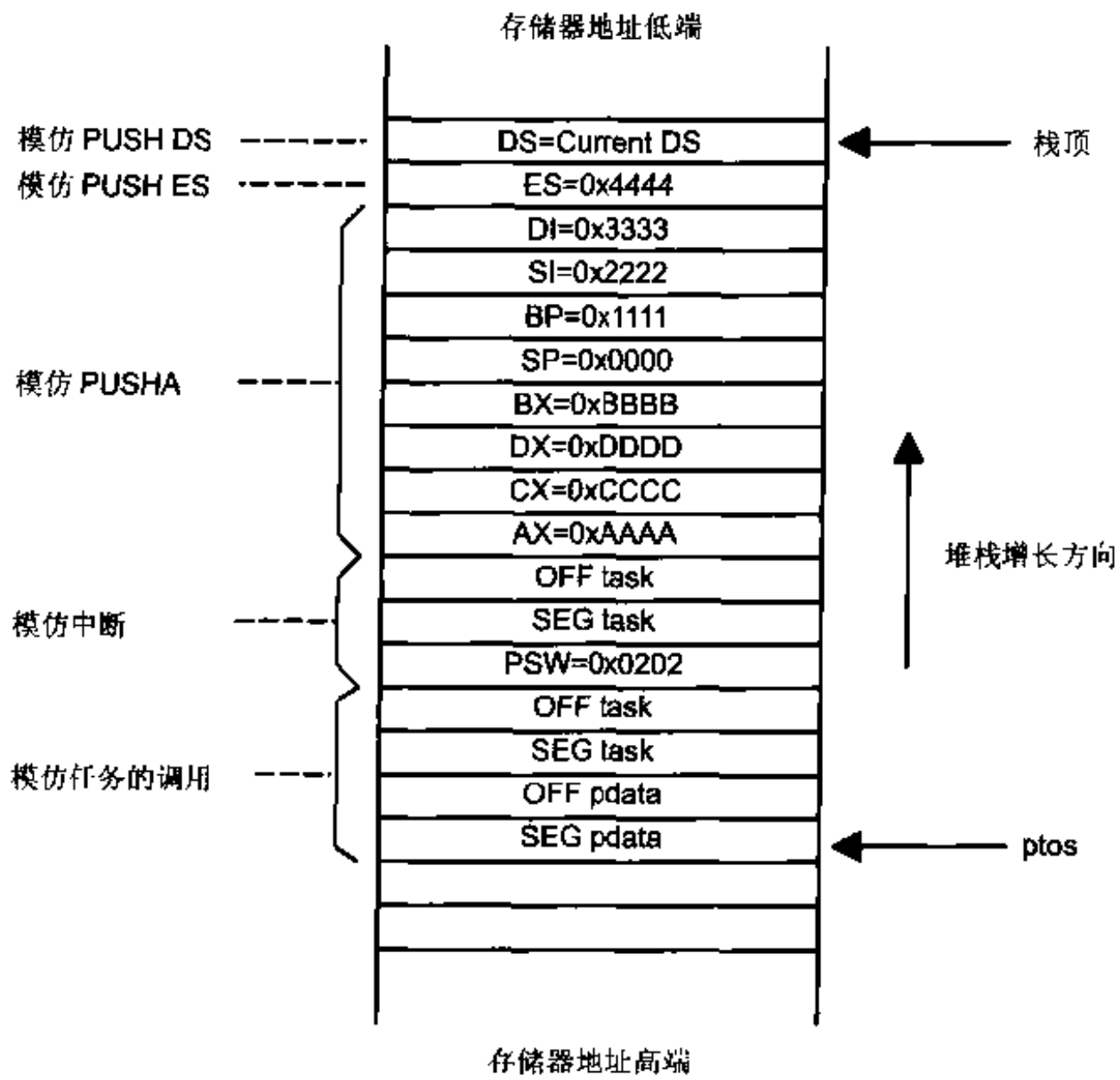


图 F14.3 传递了 pdata 参数的堆栈初始化结构

当调用 OSTaskCreate()或 OSTaskCreateExt()建立一个新任务时,须传递的参数是:任务代码的起始地址(task)、参数指针(pdata)、任务堆栈顶端的地址(ptos)及任务的优先级(prio)。OSTaskCreateExt()还需要一些其他参数,但与 OSTaskStkInit()没有关系。为了合理地初始化堆栈结构,OSTaskStkInit()(程序清单 L14.3)只需要以上提到的前 3 个参数(task,pdata 及 ptos)。

程序清单 L14.3 OS_CPU_C.C,OSTaskStkInit()

```
OS_STK   *OSTaskStkInit (void     (*task)(void *pd),
                         void      *pdata,
                         OS_STK    *ptos,
                         INT16U     opt)
```

```
{
    INT16U *stk;



    opt     = opt;
    stk     = (INT16U *)ptos;                                    (1)
    *stk--  = (INT16U)FP_SEG(pdata);                             (2)
    *stk--  = (INT16U)FP_OFF(pdata);
    *stk--  = (INT16U)FP_SEG(task);                              (3)
    *stk--  = (INT16U)FP_OFF(task);
    *stk--  = (INT16U)0x0202;                                    (4)
    *stk--  = (INT16U)FP_SEG(task);
    *stk--  = (INT16U)FP_OFF(task);
    *stk--  = (INT16U)0xAAAA;                                    (5)
    *stk--  = (INT16U)0xCCCC;
    *stk--  = (INT16U)0xDDDD;
    *stk--  = (INT16U)0xBBBB;
    *stk--  = (INT16U)0x0000;
    *stk--  = (INT16U)0x1111;
    *stk--  = (INT16U)0x2222;
    *stk--  = (INT16U)0x3333;
    *stk--  = (INT16U)0x4444;
    *stk    = _DS;                                               (6)
    return ((OS_STK *)stk);                                      (7)
}
```

L14.3(1)　由于 80x86 堆栈是 16 位宽的(以字为单位),OSTaskStkInit()将建立并初始化一个局部指针变量。该指针变量指向以字为单位的内存区,同时 μC/OS-II要求指针 ptos 指向空堆栈的入口。

L14.3(2)　Borland C/C++编译器用堆栈而不是寄存器来传递参数 pdata,此时参数 pdata 的段地址和偏移量都将按次序保存在堆栈中。

L14.3(3)　堆栈中紧接着是任务函数的起始地址。理论上,此处应该为任务的返回地址;但在 μC/OS-II 中,任务函数必须为无限循环结构,不能有返回点,所以这里保存的内容实际上无关紧要。

L14.3(4)　返回地址下面是状态字(SW),设置状态字也是为了模拟中断发生后的堆栈结构。堆栈中的 SW 初始化为 0x0202,这将使任务启动后中断是开着的;如果设为 0x0002,则任务启动后将关中断。μC/OS-II 不能设置成一些任务启动后中断是开着的,而另一些任务启动后中断是关掉的。也就是说,要么所有的任务启动时中断都是开着的,要么都是关掉的。如果确实需要突破上述限制,则可以在用 OSTaskCreateExt()函数建立任务时,通过参数 pdata 或者 opt 向任务传递启动时想要的中断状态。然而,目前没有这样做。如果选择

任务启动后中断是关掉的，那么每个任务运行时都需要重新开中断。在这种情况下，还要修改 OSTaskIdle() 和 OSTaskStat() 函数，运行时开中断。如果以上任何一个环节出了问题，系统就会崩溃，所以推荐设置 SW 为 0x0202，在任务启动时开中断。

L14.3(5) 堆栈中还须留出各个寄存器的空间。注意寄存器在堆栈中的位置要和运行指令 PUSHA，PUSH ES 及 PUSH DS 后压入堆栈的次序相同。上述指令在每次进入中断服务子程序时都会被调用。AX，BX，CX，DX，SP，BP，SI 及 DI 的次序是与指令 PUSHA 的压栈次序相同的。如果使用没有 PUSHA 指令的 8086 处理器，就要使用多个 PUSH 指令压入上述寄存器，且顺序要与 PUSHA 相同。在程序清单中，每个寄存器被初始化为不同的值，仅仅是为了调试方便。

L14.3(6) Borland 编译器支持虚拟寄存器变量操作，可以用 _DS 关键字取得 CPU 中 DS 寄存器的值，程序清单中用 _DS 直接把 DS 寄存器复制到堆栈中。

L14.3(7) 堆栈初始化工作结束后，OSTaskStkInit() 函数返回新的堆栈栈顶指针，OSTaskCreate() 或 OSTaskCreateExt() 将指针保存在任务的控制块 OS_TCB 中。

14.04.02 浮点仿真任务栈初始化函数 OSTaskStkInit_FPE_x86()

当使用浮点仿真库时（见 Borland 文档），用 Borland 编译器编译生成的代码堆栈被安排成如图 F14.4 所示的结构。编译器假设程序运行在单线程或者说单任务环境下。

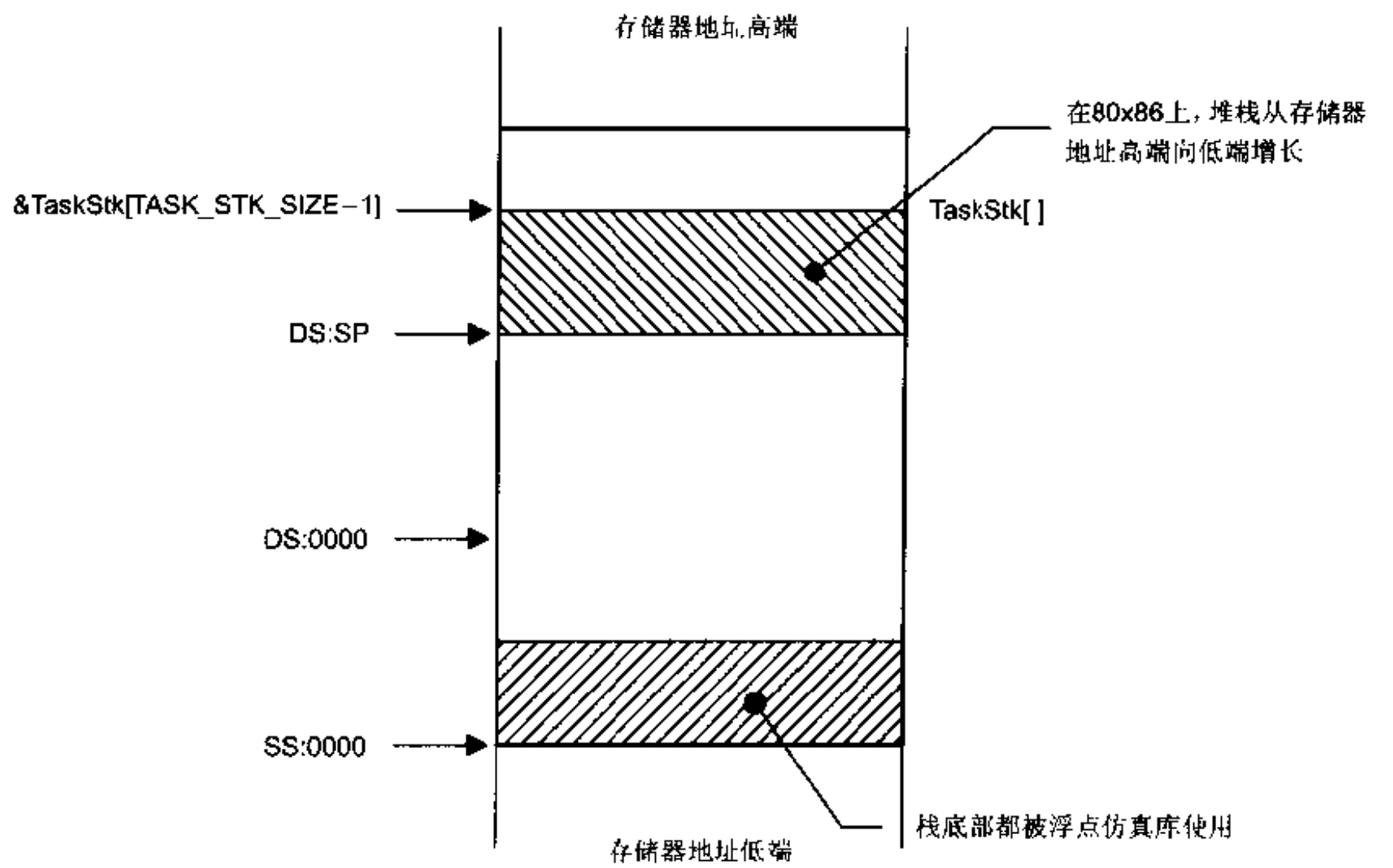


图 F14.4 Borland 浮点仿真堆栈

Borland C 浮点仿真库(FPE)占用从 SS:0x0000 开始的大约 300 B,保存浮点仿真变量。这只适用于大模式。为达到此目的,在调用 OSTaskCreate()或者 OSTaskCreateExt()建立任何一个用到浮点运算库的任务之前,必须调用专门的函数 OSTaskStkInit_FPE_x86(),对其堆栈结构进行正确的初始化。该函数适用于 Borland V3.x 和 Borland V4.5x 编译器,所以在使用别的编译器时,最好不要把 OSTaskStkInit_FPE_x86()函数包含进去。

浮点仿真库根据当前的 SS 寄存器值,将它的数据保存到上述空间中。这里认为从 SS:0x0000 开始往上的一些空间是保留给浮点操作用的。

μC/OS-II 的任务堆栈通常用以下方式静态分配:

```
OS_STK Task1Stk[TASK_STK_SIZE];     /* 任务 1 栈 */
OS_TSK Task2Stk[TASK_STK_SIZE];     /* 任务 2 栈 */
```

当 μC/OS-II 建立一个任务时,堆栈空间的高地址如下所示,被传递给 OSTaskCreate()或者 OSTaskCreateExt()函数:

```
OSTaskCreate(Task1, (void *)0, &Task1Stk[TASK_STK_SIZE-1], prio1);
OSTaskCreate(Task2, (void *)0, &Task2Stk[TASK_STK_SIZE-1], prio2);
```

任务 1 Task1()的堆栈从 DS:&Task1Stk[TASK_STK_SIZE-1]开始,任务 2 Task2()的堆栈从 DS:&Task2Stk[TASK_STK_SIZE-1]开始。μC/OS-II 完成初始化后,堆栈的栈顶地址(TOS)被保存到任务控制块 OS_TCB 中,

用上面代码建立的 2 个任务的堆栈如图 F14.5 所示。可以看出,2 个任务在同一段中,更为重要的是,它们共享相同的段地址,因为 2 个栈是从同一数据段分配到的。当 μC/OS-II 在任务切换时,会将堆栈的 DS 寄存器的值赋给 SS 寄存器。这会引起一个问题,因为 2 个任务共享了同一块浮点仿真变量区。

即使用了信号量来保护对数据段开始处的操作,这一区域还是会被浮点仿真库覆盖。用一个信号量虽然可以保证这些浮点变量资源的互斥性,但不能防止数据段被覆盖。甚至只有一个使用浮点变量的 μC/OS-II 任务,浮点库也会覆盖数据段。深层次的系统行为取决于哪些数据被覆盖了。一般地说,数据段的覆盖会引发系统崩溃。

类似的情形还发生于从堆空间中分配栈空间时,因为不知道内存的哪一部分可能被覆盖。典型的是,由于浮点仿真库覆盖了从堆空间分配到的内存块的开头,使得堆被冲毁。

为解决这一问题,以下给出了 OSTaskStkInit_FPE_x86()函数,如程序清单 L14.4 所列。该函数应在建立任务之前调用。这一函数主要规格化堆栈,使每一个栈都从 SS:0x0000 处开始,并为即将建立的任务保留空间,且正确地初始化浮点仿真变量。

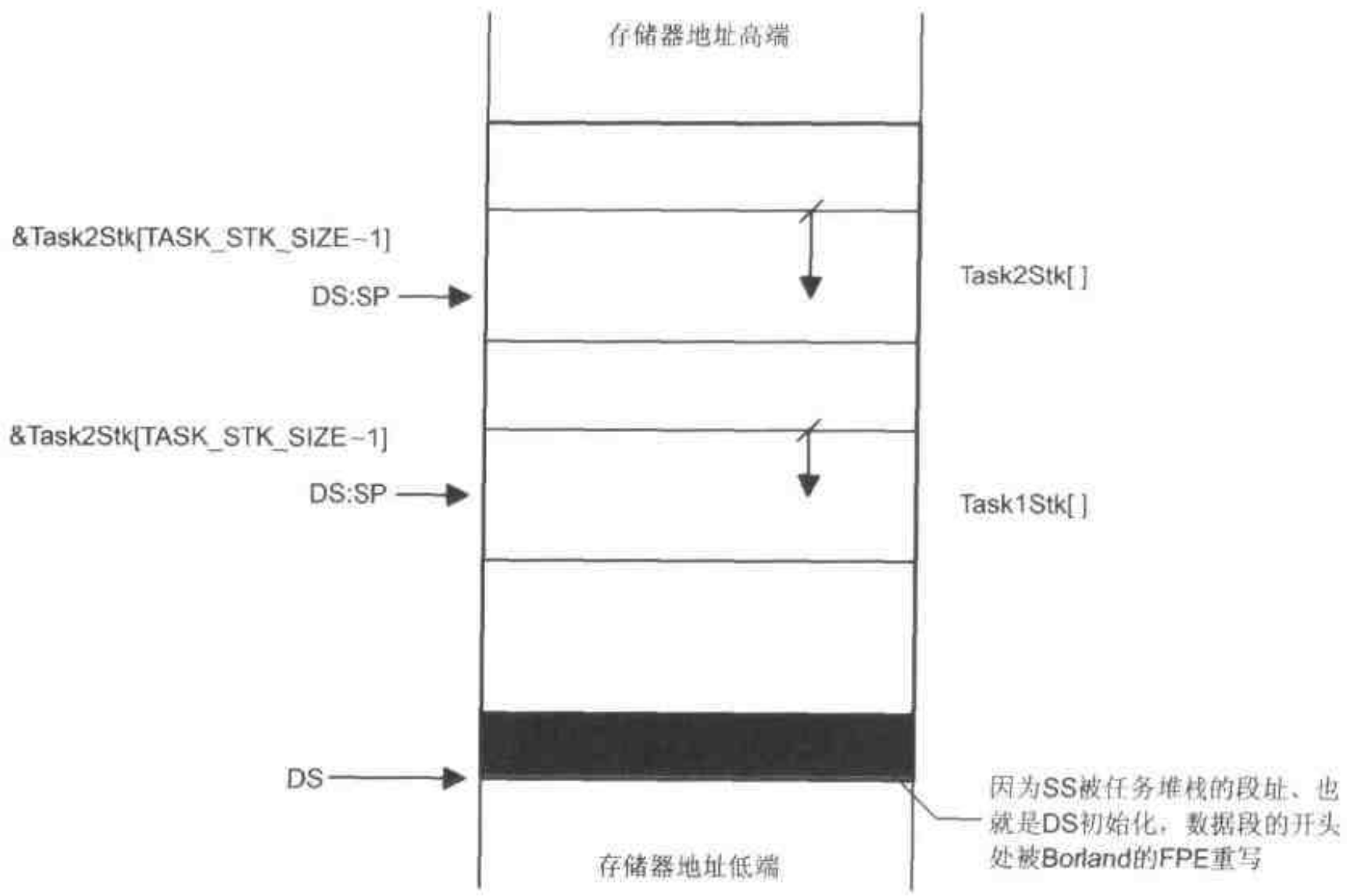


图 F14.5 Borland 浮点仿真堆栈

程序清单 L14.4 OS_CPU_C.C,OSTaskStkInit_FPE_x86()

```
void  OSTaskStkInit_FPE_x86 (OS_STK **pptos,
                             OS_STK **ppbos,
                             INT32U  *psize)
{
    INT32U   lin_tos;
    INT32U   lin_bos;
    INT16U   seg;
    INT16U   off;
    INT32U   bytes;


seg       = FP_SEG( *pptos);                                    (1)
off       = FP_OFF( *pptos);
lin_tos   = ((INT32U)seg << 4) + (INT32U)off;                   (2)
bytes     = *psize * sizeof(OS_STK);                            (3)
lin_bos   = (lin_tos - bytes + 15) & 0xFFFFFFF0L;               (4)

seg       = (INT16U)(lin_bos >> 4);                             (5)
*ppbos    = (OS_STK *)MK_FP(seg, 0x0000);                       (6)
```

```
    memcpy( *ppbos, MK_FP(_SS, 0), 384);                          (7)
    bytes     = bytes - 16;                                       (8)
    *pptos    = (OS_STK *)MK_FP(seg, (INT16U)bytes);              (9)
    *ppbos    = (OS_STK *)MK_FP(seg, 384);                        (10)
    bytes     = bytes - 384;                                      (11)
    *psize    = bytes / sizeof(OS_STK);                           (12)
}
```

从上述代码可以看到，需要给 OSTaskStkInit_FPE_x86()函数传递 3 个参数。

pptos　一个指向任务栈顶指针(TOS)的指针(指向指针的指针)。当建立任务时，任务的栈顶指针 TOS 会被传递给 OSTaskCreate()或者 OSTaskCreateExt()函数。从数据空间分配到的栈的位置由 DS 寄存器值和一个偏移量确定。由于 OSTaskStkInit_FPE_x86()函数规格化 TOS，所以指向初始 TOS 的指针被传递给这个函数，使得 TOS 值可以修改。

ppbos　一个指向任务栈底指针(BOS)的指针(指向指针的指针)。任务的栈底指针(BOS)不传递给 OSTaskCreate()函数，但会传递给 OSTaskCreateExt()函数。换句话说，ppbos 对 OSTaskCreateExt()函数是必需的。栈底常常不是在 DS:0000处，而是在 DS 寄存器加一个偏移量处。因为 OSTaskStkInit_FPE_x86()函数规格化 BOS，所以指向初始 BOS 的指针被传递给这个函数，使得 BOS 值可以修改。

psize　一个指向变量的指针，这个变量是堆栈的容量。堆栈的容量参数对于函数 OSTaskCreate()函数不是必需的，但是对于 OSTaskCreateExt()函数却是必需的。由于 OSTaskStkInit_FPE_x86()函数为浮点仿真变量保留了一部分空间，所以实际的堆栈容量会被这个函数修改，这就是要传递这个变量指针的原因。必须保证传递给 OSTaskStkInit_FPE_x86()函数的堆栈足够大，能够容纳浮点仿真变量和任务预期的空间。

L14.4(1)　OSTaskStkInit_FPE_x86() 函数从 TOS 分解出段地址和偏移量开始。

L14.4(2)　将栈顶地址转换为线性地址。

注意：80x86 实模式下，段地址乘以 16，再加上偏移地址，形成实际内存地址。

L14.4(3)　确定栈的容量，以字节为单位。

注意：在 μC/OS-II 中，必须将堆栈声明为 OS_STK 数据类型，它可以是 8 位、16 位或者 32 位宽的堆栈。

对于 Borland 编译器，堆栈宽度是 16 位，但最好使用 C 语言中的sizeof()操作。

L14.4(4)　通过从栈顶地址减去分配给堆栈的字节数，就得到栈底的线性地址。

注意：将栈底地址增加了 15 B，并且和 0xFFFFFFF0L 做“与”运算，用以将 BOS 与段的边界，或者说按 16 B 边界对齐。

L14.4(5) 从 BOS 的线性地址，确定栈底的新的段地址。

L14.4(6) 建立一个带偏移量 0x0000 的远程指针，并赋值给 BOS 指针。

L14.4(7) 为了初始化新任务堆栈的浮点仿真变量区，只需简单地将调用本函数任务的堆栈底部复制给新任务的堆栈。

注意：调用本函数的任务栈必须是已经初始化了浮点仿真变量区的栈，不这样做会引起不可预期的后果。

Borland 浮点仿真(FPE)库认为，从 SS:0x0000 开始的大约 300 B 是保留给浮点仿真变量的。这只适用于大模式下的编译。笔者决定复制 384 B (0x0180)。实际上用户不必复制这么多字节，但笔者认为有一些余量会安全些。这样，用户任务的堆栈至少须有 384 B 加上用户任务原先的堆栈需求(当然包括中断嵌套的需求)。_SS 是 Borland 的虚拟寄存器，使代码可以得到 CPU 堆栈段寄存器的当前值。而且，笔者决定用 ANSI 的标准函数 memcpy()完成复制，因为 Borland 优化了该函数。

L14.4(8) 确定规格化的栈顶地址。因为要将堆栈对齐到段地址的边界上，需要先减去 16 B。如果能保证堆栈段地址的边界总是对齐的，就不必这么做。

L14.4(9) 利用新的段地址[见 L14.4(6)]和新的堆栈容量(对齐到段的边界)，通过一个远程指针确定新的栈顶地址 TOS。

L14.4(10) 将 BOS 上移 384 B，使之适用于堆栈检查(如果程序调用 OSTaskStkChk())。

L14.4(11)

L14.4(12) 如果使用堆栈检查，则 μC/OS-II 需要知道新堆栈的容量。当然不希望从原先堆栈的底部，而是从新堆栈底部开始堆栈检查。

图 F14.6 示意 OSTaskStkInit_FPE_x86()函数所做的堆栈规格化。段对齐没有在图中表示出来。

应如同程序清单 L14.5 所列那样调用 OSTaskStkInit_FPE_x86()函数。程序清单中有调用 OSTaskCreate() 和 OSTaskCreateExt()这 2 个函数的范例。这段代码表明，如果任务要做浮点数学运算，那么函数 OSTaskStkInit_FPE_x86()的调用必须在调用 OSTaskCreate()或 OSTaskCreateExt()函数之前，用以初始化任务堆栈，如同前面描述过的那样。返回的指针(ptos 和 pbos)在建立任务的函数中要用到。

注意：pbos 被传递给 OSTaskCreateExt()，作为新堆栈的栈底。若调用 OSTaskStkChk()(任务必须是由 OSTaskCreateExt()函数建立的)在运行时检查堆栈的使用情况，那么 OSTaskStkChk()报告的值比堆栈的实际容量少 384 B(见图 F14.6，堆栈调整后的状况)。

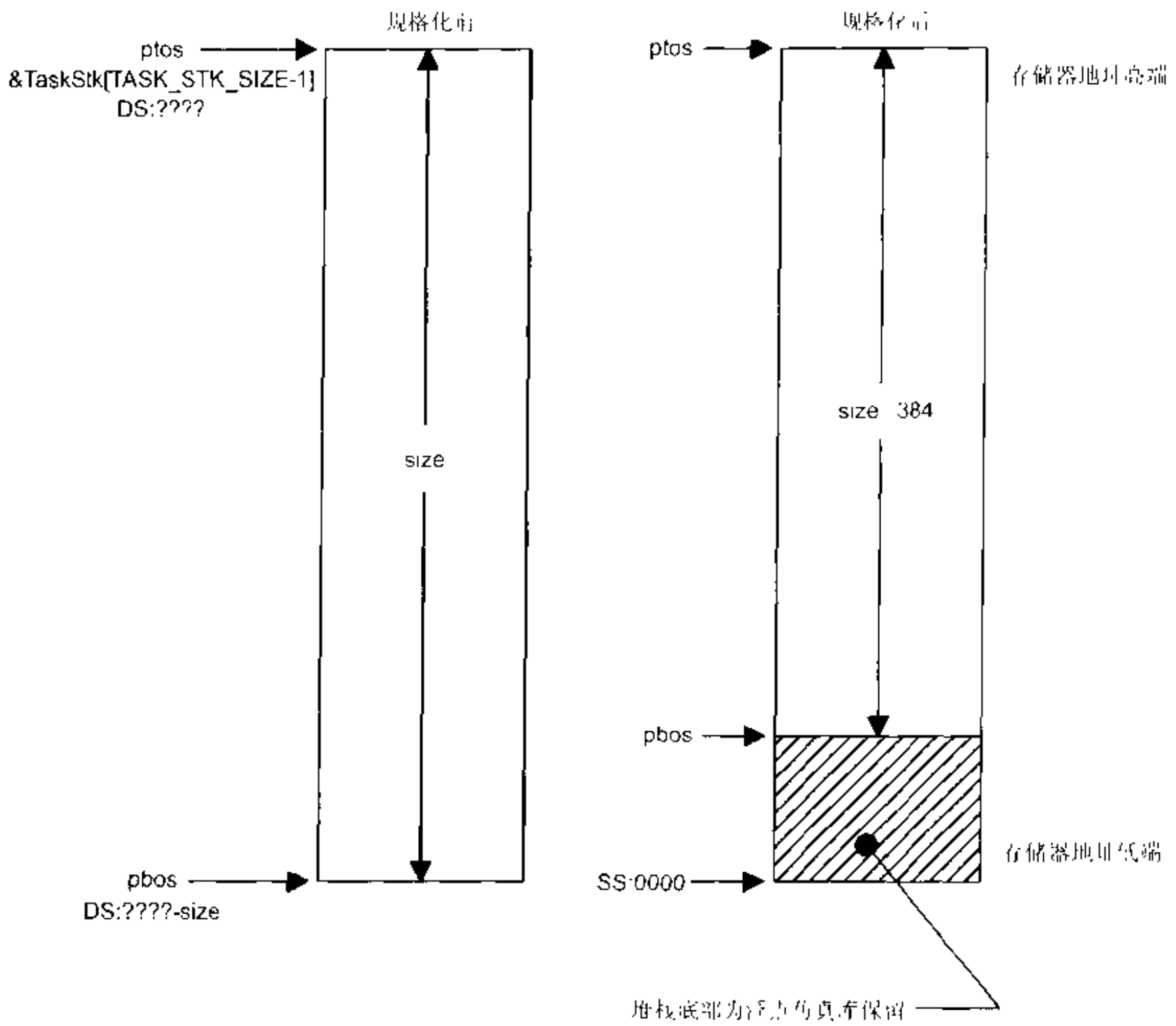


图 F14.6 OSTaskStkInit_FPE_x86()所做的堆栈规格化

程序清单 L14.5 OS_CPU_C.C,OSTaskStkInit_FPE_x86()函数的用法

```
OS_STK Task1Stk[1000];
OS_STK Task2Stk[1000];


void main (void)
{
    OS_STK *ptos;
    OS_STK *pbos;
    INT32U  size;


    OSInit();
```

```
    ptos   = &Task1Stk[999];
    pbos   = &Task1Stk[0];
    size   = 1000;
    OSTaskStkInit_FPE_x86(&ptos, &pbos, &size);
    OSTaskCreate(Task1,
                 (void *)0,
                 ptos,
                 10);
    .
    .
    ptos   = &Task2Stk[999];
    pbos   = &Task2Stk[0];
    size   = 1000;
    OSTaskStkInit_FPE_x86(&ptos, &pbos, &size);
    OSTaskCreateExt(Task2,
                    (void *)0,
                    ptos,
                    11,
                    11,
                    pbos,
                    size,
                    (void *)0,
                    OS_TASK_OPT_SAVE_FP);
    .
    .
    OSStart();
}
```

必须当心，应用代码不会做任何浮点异常处理（例如被0除）。这是因为浮点库在这样的环境中工作不完全正常，可以通过增加数据范围检查，以避免运行中出现异常。

14.04.03 任务建立接口函数 OSTaskCreateHook()

像前面提到的那样，在OS_CPU_C.C中没有定义该函数。也就是说，本移植中建立一个任务时，没有做额外的工作。将ptcb赋值为ptcb，只是为了编译时不会报"错"。该参数在OSTaskCreateHook()函数中没有使用。该函数的源代码见程序清单L14.6。

程序清单L14.6 OS_CPU_C.C，OSTaskCreateHook()

```
void  OSTaskCreateHook (OS_TCB *ptcb)
{
    ptcb = ptcb;
}
```

14.04.04 任务删除接口函数 OSTaskDelHook()

如同前面提到的，在 OS_CPU_C.C 中没有定义该函数。也就是说，本移植中建立一个任务时，没有做额外工作。将 ptcb 赋值为 ptcb，只是为了编译时不会报"错"——OSTaskDelHook() 函数没有使用该参数。该函数的源代码见程序清单 L14.7。

程序清单 L14.7 OS_CPU_C.C, OSTaskDelHook()

```
void  OSTaskDelHook (OS_TCB *ptcb)
{
    ptcb = ptcb;
}
```

14.04.05 任务切换接口函数 OSTaskSwHook()

OS_CPU_C.C 中的该函数什么也不做。考虑到用户可能会利用该函数做点什么，仅在这里给出函数代码的框架，如程序清单 L14.8 所列。

程序清单 L14.8 OS_CPU_C.C, OSTaskSwHook()

```
void  OSTaskSwHook (void)
{
#if 0
    if (OSRunning == TRUE) {
        /* 保存拟被挂起任务的寄存器 */
    }
        /* 恢复拟运行任务的寄存器 */
#endif
}
```

14.04.06 空闲任务接口函数 OSTaskIdleHook()

OS_CPU_C.C 中的该函数什么也不做，见程序清单 L14.9。

程序清单 L14.9 OS_CPU_C.C, OSTaskIdleHook()

```
void  OSTaskIdleHook (void)
{
}
```

14.04.07 统计任务接口函数 OSTaskStatHook()

OS_CPU_C.C 中的该函数什么也不做。至于用户可以利用该函数做什么事情，可参见

第1章："初识 μC/OS-II"。

该函数的源代码如程序清单 L14.10 所列。

程序清单 L14.10 OS_CPU_C.C, OSTaskStatHook()

```
void  OSTaskStatHook (void)
{
}
```

14.04.08 时钟节拍接口函数 OSTimeTickHook()

OS_CPU_C.C 中的该函数什么也不做。

该函数的源代码如程序清单 L14.11 所列。

程序清单 L14.11 OS_CPU_C.C, OSTimeTickHook()

```
void  OSTimeTickHook (void)
{
}
```

14.04.09 系统初始化开始接口函数 OSInitHookBegin()

OS_CPU_C.C 中的该函数什么也不做。

该函数的源代码如程序清单 L14.12 所列。

程序清单 L14.12 OS_CPU_C.C, OSInitHookBegin()

```
void  OSInitHookBegin (void)
{
}
```

14.04.10 系统初始化结束接口函数 OSInitHookEnd()

OS_CPU_C.C 中的该函数什么也不做。

该函数的源代码如程序清单 L14.13 所列。

程序清单 L14.13 OS_CPU_C.C, OSInitHookEnd()

```
void  OSInitHookEnd (void)
{
}
```

14.04.11 控制块初始化接口函数 OSTCBInitHook()

OS_CPU_C.C 中的该函数什么也不做。

该函数的源代码如程序清单 L14.14 所列。

程序清单 L14.14 OS_CPU_C.C, OSTCBInitHook()

```
void  OSTCBInitHook (void)
{
}
```

14.05 OS_CPU_A.ASM 文件

μC/OS-II 的移植需要改写 OS_CPU_A.ASM 中的 4 个函数：

OSStartHighRdy()；

OSCtxSw()；

OSIntCtxSw()；

OSTickISR()。

14.05.01 OSStartHighRdy()

该函数由 OSStart() 函数调用，功能是让进入就绪态的优先级最高的任务运行。在调用 OSStart()之前，必须先调用 OSInit()函数，且已经建立了至少 1 个任务(见 OSTaskCrea-te()和 OSTaskCreateExt()函数)。OSStartHighRdy()默认指针 OSTCBHighRdy 指向优先级最高的就绪态任务的任务控制块(OS_TCB)。图 F14.7 给出了由函数 OSTaskCreate()或 OSTaskCreateExt()建立的任务的堆栈结构。在 OSStart() 调用 OSStartHighRdy()函数之前，任务堆栈结构就是这个样子。很明显，OSTCBHighRdy->OSTCBStkPtr 指向任务堆栈的顶端。函数 OSStartHighRdy()的代码见程序清单 L14.15。

程序清单 L14.15 OSStartHighRdy()

```
_OSStartHighRdy  PROC FAR

            MOV   AX, SEG _OSTCBHighRdy
            MOV   DS, AX
;
            CALL  FAR PTR _OSTaskSwHook                                  (1)
;
            MOV   AL, 1                                                  (2)
            MOV   BYTE PTR DS:_OSRunning, AL
;
            LES   BX, DWORD PTR DS:_OSTCBHighRdy                         (3)
```

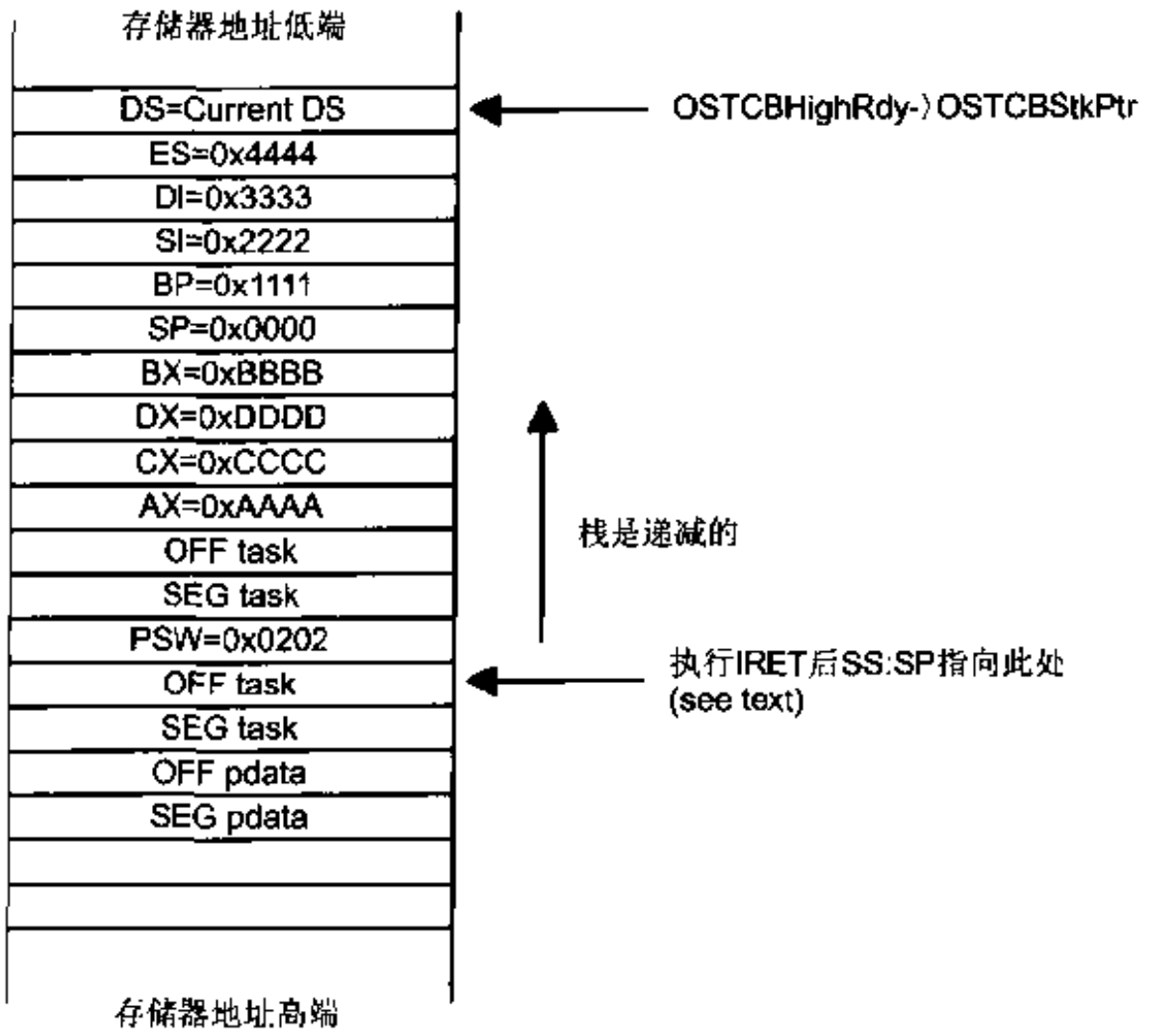


图 F14.7 任务创立时的 80x86 堆栈结构

```
        MOV   SS, ES:[BX + 2]
        MOV   SP, ES:[BX + 0]
;
        POP   DS                                                    (4)
        POP   ES
        POPA
;
        IRET                                                        (5)

_OSStartHighRdy  ENDP
```

L14.15(1) 在第 13 章中提到，在 OSStartHighRdy()启动之前，必须先调用函数 OSTaskSwHook()，而 OSTaskSwHook()函数必须查询 OSRunning 的状态(此时 OSRunning 的状态是 FALSE)。OSTaskSwHook()只做恢复 CPU 寄存器的操作，并不需要做先保存、再恢复的操作。

L14.15(2) OSStartHighRdy()函数把 OSRunning 设成 TRUE，这样以后在调用 OSTaskSwHook()时，OSTaskSwHook()函数便可以进行先保存、再恢复环境的操作了。由于代码是用汇编语言写的，不能从 C 编译器中得到 TRUE 的准确值。这里假设 TRUE 是 1。

L14.15(3) OSStartHighRdy()从任务控制块 OS_TCB 中获得并恢复堆栈指针。如同前面提到的，堆栈指针的值保存在任务控制块 TCB 的开始处(也就是任务

控制块 OS_TCB)，这样较容易用汇编语言访问该指针。

L14.15(4) OSStartHighRdy()把 CPU 所有整数寄存器的值保存在任务堆栈中。

L14.15(5) 执行 IRET 指令，从中断返回。这里建立了任务的堆栈结构，看起来好像刚刚发生了中断，所有的 CPU 寄存器都被推入堆栈中。IRET 指令将任务的入口地址从堆栈中弹出，并把任务入口地址放在 CS:IP 寄存器中，地址后面跟着的值(叫做状态字或者状态位)放在 SW 寄存器中。

如图 F14.7 所示，执行 IRET 指令时，堆栈指针(SS:SP)指向任务返回地址，这样看起来像是任务被一个普通函数调用了。SS:SP+4 指向自变量 pdata，pdata 会给任务传递参数。换句话说，任务无须了解它是被 OSStartHighRdy()调用的，还是被其他函数调用的。

14.05.02 OSCtxSw()

OSCtxSw()是一个任务级的任务切换函数，在任务中调用，区别于在中断程序中调用的 OSIntCtxSw()。在 80x86 系统上，它通过执行一条软中断指令实现任务切换，软中断向量指向 OSCtxSw()。在 μC/OS-II 中，如果任务调用了某函数，而该函数的执行结果可能造成系统任务重新调度，例如唤醒了一个优先级更高的任务，使当前运行着的任务已经不是最重要的任务了，则 μC/OS-II 会在函数的末尾调用 OSSched()。如果 OSSched()判断需要进行任务调度，则会找到该任务控制块 OS_TCB 的地址，并将该地址复制给 OSTCBHighRdy，然后通过宏 OS_TASK_SW()执行软中断，进行任务切换。

注意：在此过程中变量 OSTCBCur 始终包含指向当前运行任务的控制块 OS_TCB 的指针。

程序清单 L14.16 为 OSCtxSw()的代码。图 F14.8 是任务挂起或恢复时的堆栈结构。

程序清单 L14.16 OSCtxSw()

```
_OSCtxSw    PROC    FAR                                          (1)
;
            PUSHA                                                (2)
            PUSH    ES
            PUSH    DS
;
            MOV     AX, SEG _OSTCBCur
            MOV     DS, AX
;
            LES     BX, DWORD PTR DS:_OSTCBCur                   (3)
            MOV     ES:[BX + 2], SS
            MOV     ES:[BX + 0], SP
;
            CALL    FAR PTR _OSTaskSwHook                        (4)
;
            MOV     AX, WORD PTR DS:_OSTCBHighRdy + 2            (5)
            MOV     DX, WORD PTR DS:_OSTCBHighRdy
            MOV     WORD PTR DS:_OSTCBCur + 2, AX
```

```
            MOV     WORD PTR DS:_OSTCBCur, DX
;
            MOV     AL, BYTE PTR DS:_OSPrioHighRdy                          (6)
            MOV     BYTE PTR DS:_OSPrioCur, AL
;
            LES     BX, DWORD PTR DS:_OSTCBHighRdy                          (7)
            MOV     SS, ES:[BX+2]
            MOV     SP, ES:[BX]
;
            POP     DS                                                      (8)
            POP     ES
            POPA
;
            IRET                                                            (9)
;
_OSCtxSw    ENDP
```

F14.8(1)

L14.16(1) 在 80x86 处理器上，软中断指令把状态寄存器 SW 的值推入当前的任务栈中，接着入栈的是执行软中断指令的任务（调用 OS_TASK_SW()的任务）的中断返回地址（段地址和偏移量地址）。

F14.8(2)

L14.16(2) 将挂起任务的其他 CPU 寄存器保存到当前任务堆栈中。

F14.8(3)

L14.16(3) 指向新的堆栈结构的指针保存在任务控制块 OS_TCB 中。这个指针包括堆栈段寄存器（SS）和堆栈指针寄存器（SP）。μC/OS-II 将堆栈指针放在任务控制块 OS_TCB 结构的开头，这样用汇编语言可以很容易地保存和恢复堆栈指针。

L14.16(4) 调用可由用户定义的应用程序接口函数 OSTaskSwHook()。

注意：当调用 OSTaskSwHook()时，OSTCBCur 指向当前任务的任务控制块 OS_TCB，而 OSTCBHighRdy 指向新任务的任务控制块 OS_TCB。这样可在接口函数 OSTaskSwHook()中与上述 2 个任务控制块 OS_TCB 的任何一个打交道。

如果不想使用 OSTaskSwHook()，则可以注释掉这个调用。这样在任务切换时，可节省几个时钟周期。换句话说，若不打算使用OSTaskSwHook()，也就没有必要为调用这个函数和从这个函数返回支付额外的 CPU 负荷。一般情况下，在移植时都会调用这个函数。

L14.16(5) 从 OSTaskSwHook()返回后，OSTCBHighRdy 被复制给 OSTCBCur。这是因为新任务就是当前任务。

L14.16(6) OSPrioHighRdy 被复制给 OSPrioCur。这同样是因为新任务就是当前任务。

F14.8(4)

L14.16(7) 至此，OSCtxSw()把新任务栈中保存的栈指针内容传递给处理器的相应寄存器。这是通过从新任务的 OS_TCB 获得寄存器 SS 和寄存器 SP 的值完成的。

F14.8(5)

L14.16(8) 从新任务的堆栈中弹出相应值到 CPU 的其他寄存器。

F14.8(6)

L14.16(9) 执行中断返回指令 IRET，装入新任务的程序计数器值和状态字。执行了这一指令后，处理器使新任务恢复运行。

F14.8 为任务级做任务切换时的 80x86 堆栈结构。

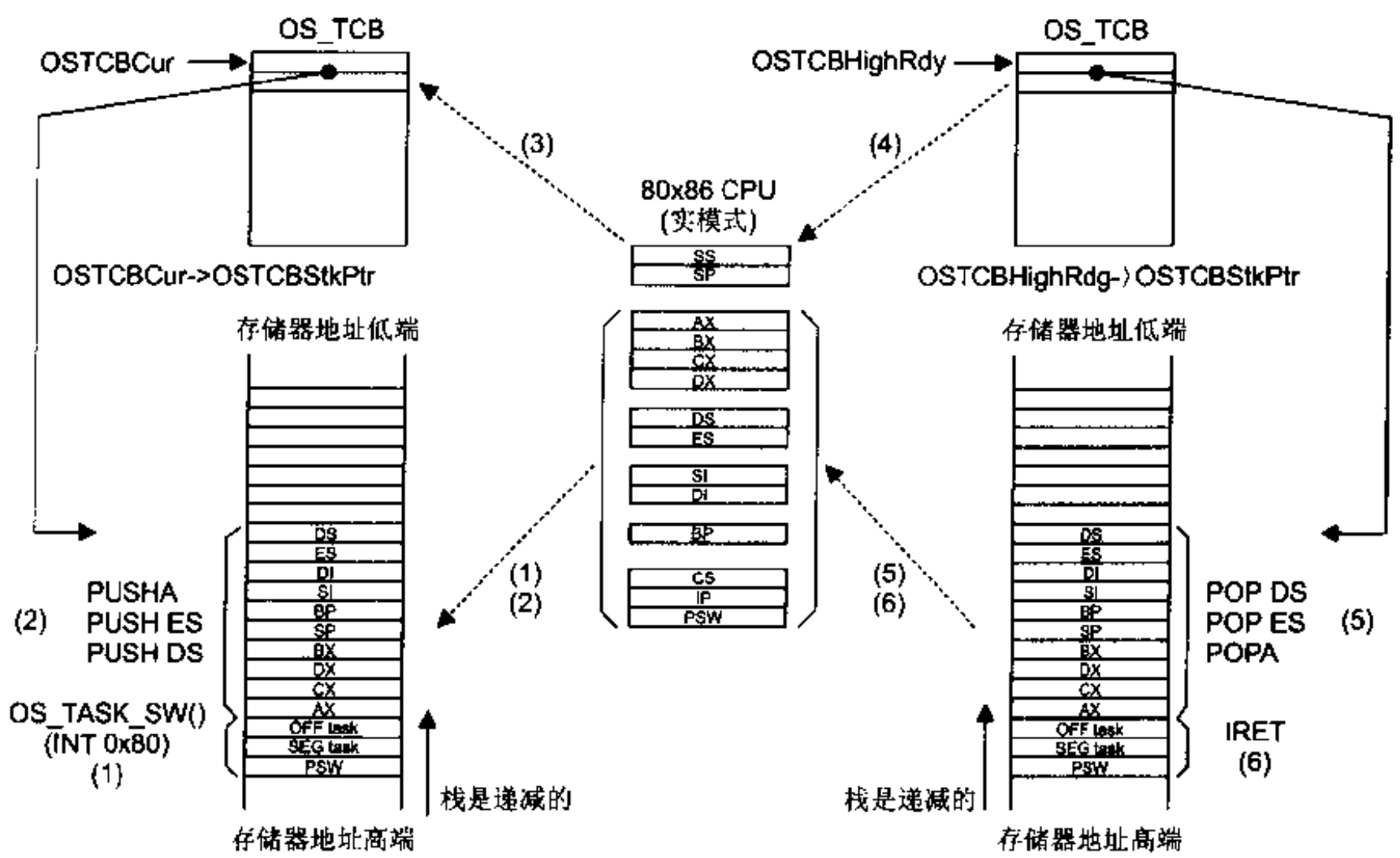


图 F14.8 在任务级做任务切换时的 80x86 堆栈结构

注意：在执行 OSCtxSw()时，中断是关掉的；执行用户定义的 OSTaskSwHook()接口函数时，中断也是关掉的。

可以访问 www.uCOS-II.com 网站，参考 Intel 80x86 处理器的任务切换函数实例。

14.05.03 OSIntCtxSw()

由于中断可能会使更高优先级的任务进入就绪态，为了让更高优先级的任务能立即运行，需要进行任务切换。在中断服务子程序的最后，OSIntExit()函数会调用 OSIntCtxSw()做任务切换，所以 OSIntCtxSw()又称为中断级的任务切换函数。由于在调用 OSIntCtxSw() 之前已经发生了中断，OSIntCtxSw()已经将默认 CPU 寄存器保存在被中断了的任务的堆栈中。

程序清单 L14.17 给出的代码大部分与 OSCtxSw() 的代码相同；不同之处是，由于中断已经发生，此处无需再保存 CPU 寄存器(没有 PUSHA，PUSH ES 或 PUSH DS 等指令)。同样，堆栈

指针也被中断服务子程序保存在任务控制块 OS_TCB 中。图 F14.9 描述了任务切换的过程。

为了理解它们之间的区别，假设处理器接收到一个中断，并且中断是开着的。处理器完成当前的指令并开始响应中断，进入中断处理程序。

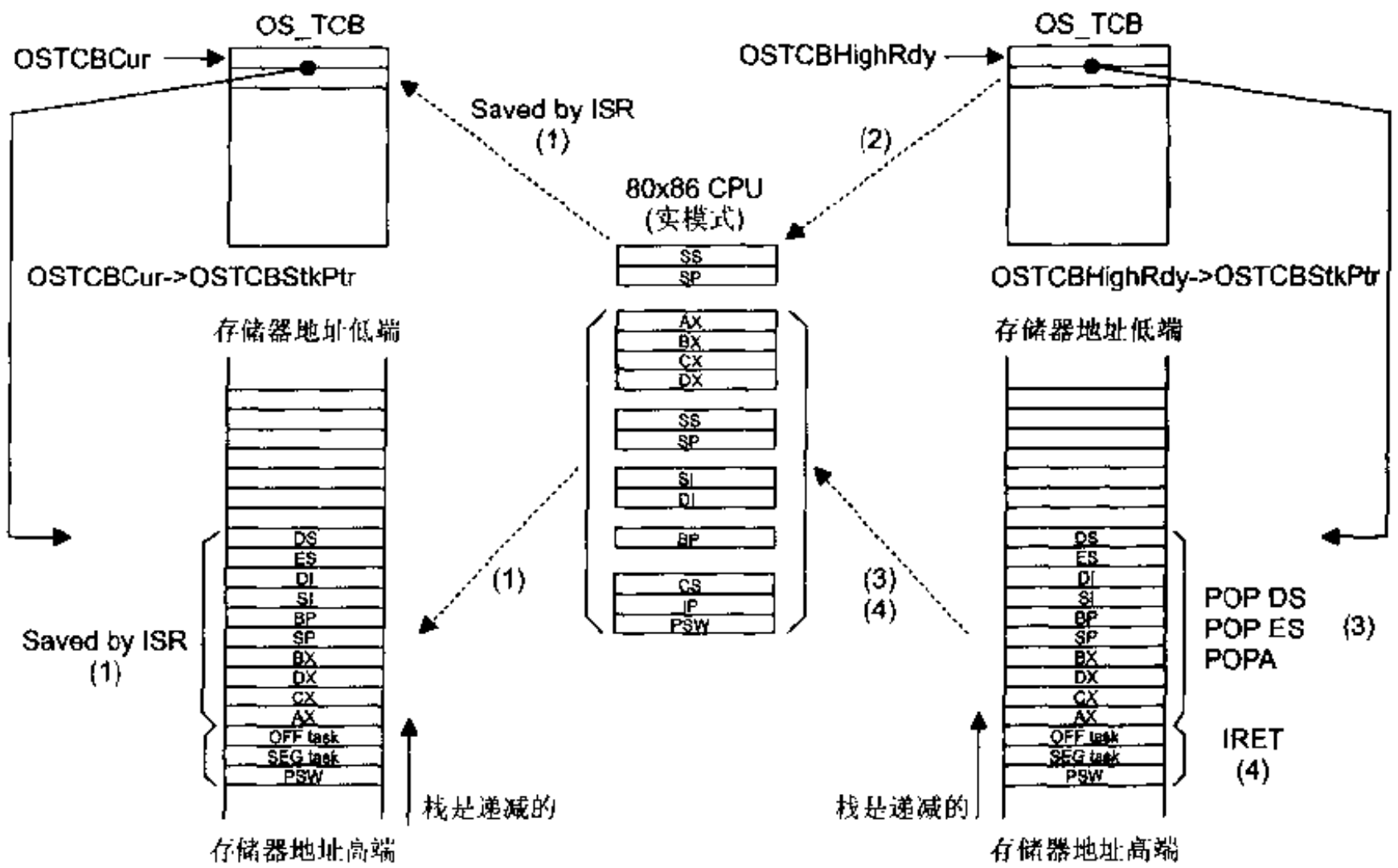


图 F14.9 中断级任务切换时的 80x86 堆栈结构

F14.9(1) 80x86 自动把处理器的 SW 寄存器推入堆栈，然后把中断任务的返回地址也推入堆栈。接下来 CPU 从中断向量处找到中断服务子程序的入口地址，运行中断服务子程序。在 μC/OS-II 中，要求在中断服务子程序的开头保存 CPU 其他寄存器的内容。保存了寄存器的内容后，μC/OS-II 还要求把堆栈指针保存在任务控制块 OS_TCB 中。

此后，必须调用 OSIntEnter()，或者直接给全局变量 OSIntNesting 加 1。至此可以认为任务已被挂起，如果需要，则可以切换到另一个任务。

现在，中断服务子程序可服务于产生中断的设备了。该服务有可能调用了 OSMboxPost()，OSQPostFront()，OSQPost()或 OSQPostOpt()等函数给其他任务发了消息，从而使一个优先级更高的任务进入了就绪态，需要做任务切换。也可能唤醒了一个更高优先级的任务，例如调用了 OSTaskResume()，OSTimeTick()及 OSTimeDlyResume()等等，需要做任务切换。

如果中断服务子程序使一个更高优先级的任务进入了就绪态，那么 μC/OS-II 要求在中断服务子程序完成对中断设备的服务以后，调用 OSIntExit()。OSIntExit()主要是告诉 μC/OS-II，所有嵌套的中断都已经退出，可以返回到任务级代码了。换句话说，当中断嵌套层数 OSIntNesting 减为 0 时，OSIntExit()就可以返回到任务级代码了。

当 OSIntExit()开始执行时，该函数得知被中断的任务不再是应该继续运行的任务，有更

高优先级的任务要运行。这种情况下，指针 OSTCBHighRdy 应该指向新任务的控制块 OS_TCB，OSIntExit()会调用 OSIntCtxSw()进行任务切换。

程序清单 L14.17　OSIntCtxSw()

```
_OSIntCtxSw  PROC    FAR
;
             CALL    FAR PTR _OSTaskSwHook                          (1)
;
             MOV     AX, SEG _OSTCBCur
             MOV     DS, AX
;
             MOV     AX, WORD PTR DS:_OSTCBHighRdy + 2              (2)
             MOV     DX, WORD PTR DS:_OSTCBHighRdy
             MOV     WORD PTR DS:_OSTCBCur + 2, AX
             MOV     WORD PTR DS:_OSTCBCur, DX
;
             MOV     AL, BYTE PTR DS:_OSPrioHighRdy                 (3)
             MOV     BYTE PTR DS:_OSPrioCur, AL
;
             LES     BX, DWORD PTR DS:_OSTCBHighRdy                 (4)
             MOV     SS, ES:[BX + 2]
             MOV     SP, ES:[BX]
;
             POP     DS                                             (5)
             POP     ES
             POPA
;
             IRET                                                   (6)
;
_OSIntCtxSw ENDP
```

L14.17(1)　OSIntCtxSw()做的第 1 件事是，调用可由用户定义的任务切换接口函数 OSTaskSwHook()。

注意：当调用 OSTaskSwHook()时，OSTCBCur 指向当前任务的任务控制块 OS_TCB，而 OSTCBHighRdy 指向新任务的任务控制块 OS_TCB。这样可以从 OSTaskSwHook()中访问这 2 个任务的任务控制块 OS_TCB 中的任何一个。

同样，如果不想使用 OSTaskSwHook()，则可以注释掉这条语句。这样在任务切换时，可以节省几个时钟周期。

L14.17(2)　当从 OSTaskSwHook()返回时，OSTCBHighRdy 被复制给 OSTCBCur，这是因为此时新任务应该是当前任务。

L14.17(3)　OSPrioHighRdy 也被复制给 OSPrioCur，这也是因为此时新任务应该是当

前任务。

F14.9(2)

L14.17(4) 至此，OSCtxSw()应该把新任务的运行环境装入到处理器的寄存器中。这是通过从新任务的任务控制块 OS_TCB 中获取寄存器 SS 和寄存器 SP 的值来实现的。

F14.9(3)

L14.17(5) 从堆栈中弹出和恢复 CPU 的其余寄存器。

F14.9(4)

L14.17(6) 执行 IRET 指令，设置新任务的程序计数器和状态字。执行了这一指令后，处理器恢复执行新任务。

注意：在执行 OSIntCtxSw()时，中断是关掉的；在执行用户定义的 OSTaskSwHook()接口函数时，中断也是关掉的。

14.05.04 OSTickISR()

在 14.03.05 小节："OS_CPU.H，时钟节拍频率"中已经提到，实时操作系统的时钟节拍频率应该为 10～100 Hz。在 PC 中，时钟节拍由硬件定时器产生，硬件定时器会中断 CPU，间隔是 54.93 ms (18.206 48 Hz)。笔者将时钟节拍频率设为 200 Hz。PC 机时钟节拍的中断向量为 0x08，μC/OS-II 将此向量截取，使之指向 μC/OS-II 的时钟节拍中断服务子程序 OSTickISR()，而原先的中断向量保存在中断向量 129(0x81)中(参见 PC.C 文件中的 PC_DOSSaveReturn()函数)。为满足 DOS 的需要，原先的中断服务还是每隔 54.93 ms(实际上还要短些)调用 1 次。图 F14.10 为安装 μC/OS-II 之前与之后的中断向量表。

在 μC/OS-II 中，当调用 OSStart() 启动多任务环境后，时钟中断的作用是非常重要的；但在 PC 环境下，启动 μC/OS-II 之前就已经有时钟中断发生了。实际上希望在 μC/OS-II 初始化完成之后，再发生时钟中断，调用 OSTickISR()。

为了防止中断服务子程序 ISR 在 μC/OS-II 准备好之前调用 OSTickISR()，需要做以下的工作：

Main()

调用 OSInit()初始化 μC/OS-II；

调用 PC_DOSSaveReturn()(见 PC.C)；

调用 PC_VectSet，把 OSCtxSw 放在向量 0x80 处；

建立至少 1 个应用任务；

调用 OSStart()，以运行多任务。

第 1 个任务要做的工作有：

把 OSTickISR()的起始地址放在向量 0x08 处；

时钟节拍从 18.206 48 Hz 改变为 200 Hz。

PC 上时钟处理很巧妙，以程序清单 L14.18 中的示意性代码解释其过程。这些代码一般情况下是用汇编语言编写的。

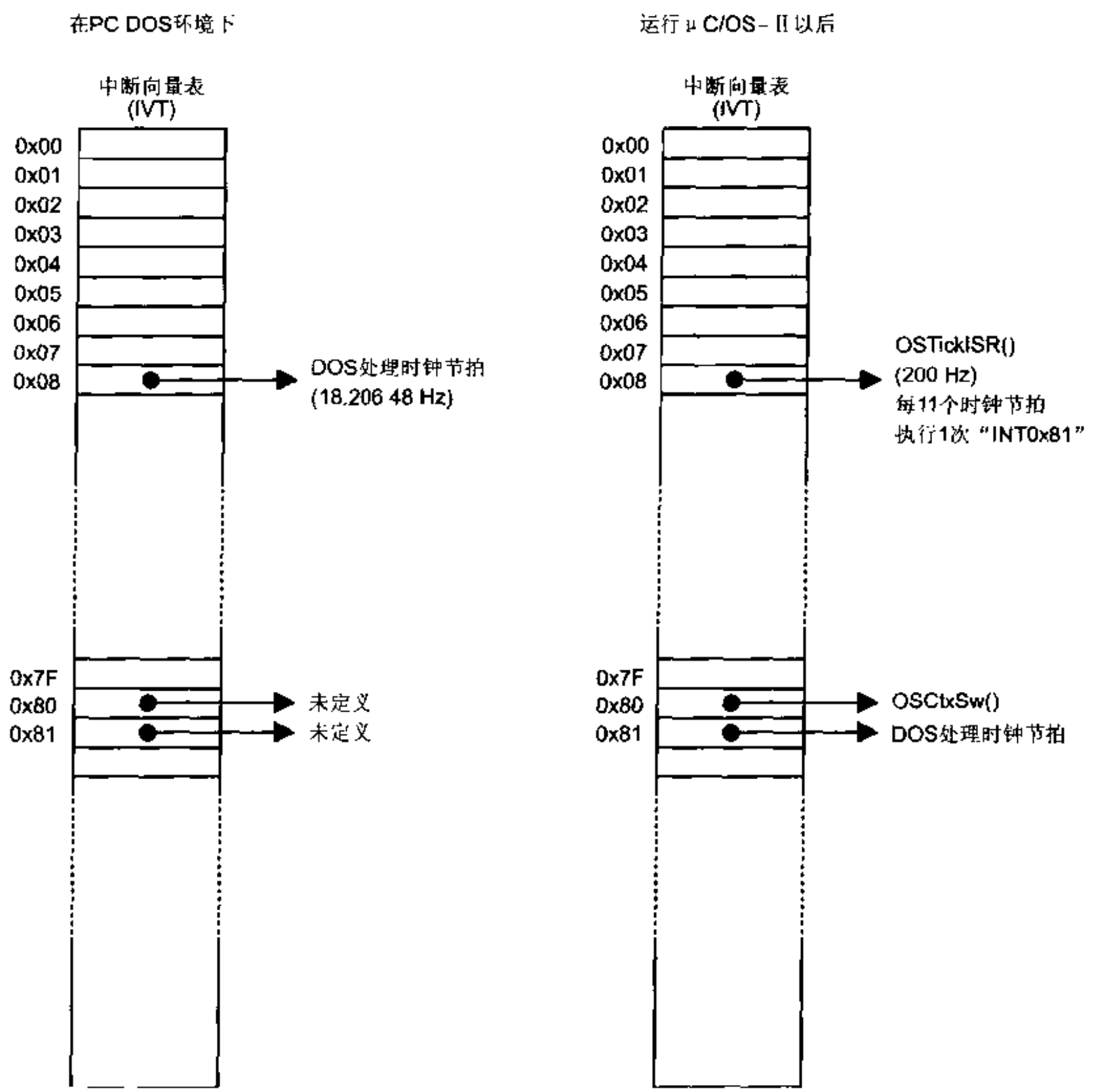


图 F14.10　PC 中断向量表(IVT)

程序清单 L14.18　OSTickISR()的示意性代码

```
void  OSTickISR (void)
{
    将全部寄存器保存到当前任务的堆栈中；                                    (1)
    OSIntNesting ++ ;                                                       (2)
    if (OSIntNesting == 1) {                                                (3)
        OSTCBCur->OSTCBStkPtr = SS:SP                                       (4)
    }
    OSTickDOSCtr -- ;                                                       (5)
    if (OSTickDOSCtr == 0) {                                                (6)
        OSTickDOSCtr = 11;                                                  (7)
```

```
        INT 81H;      /* DOS 会清中断 */
    } else {
        将 EOI 送往 PIC;                                          (8)
    }
    OSTimeTick();                                                 (9)
    OSIntExit();                                                  (10)
    从当前任务的堆栈中恢复全部寄存器;                              (11)
    中断返回;                                                     (12)
}
```

L14.18(1) 如同所有 μC/OS-II 的中断服务子程序一样，所有的寄存器须保存在当前任务的堆栈中。

L14.18(2) 当进入中断服务子程序 ISR 时，须告诉 μC/OS-II，进入中断服务子程序了。可通过调用 OSIntEnter()实现，或以直接给中断嵌套层数 OSIntNesting 加 1 的方式实现。直接给 OSIntNesting 加 1 会更快一些。OSIntEnter()会检查 OSIntNesting 是否超过了 255，这样做中断嵌套更安全。

L14.18(3)

L14.18(4) 如果该中断服务子程序是中断的第 1 层，即没有中断嵌套，则需把堆栈指针保存在当前任务的任务控制块 OS_TCB 中。

L14.18(5)

L14.18(6)

L14.18(7) 计数器 OSTickDOSCtr 减 1，当 OSTickDOSCtr 为 0 时，调用 DOS 的时钟节拍处理函数。这种情况每隔 54.93 ms 发生 1 次。

L14.18(8) 11 次中有 10 次，中断优先级控制器 PIC 会收到一个清除中断的命令。

注意：当调用 DOS 的时钟节拍处理函数时，这一操作是不必要的，因为 DOS 的时钟节拍处理函数直接清除中断源。

L14.18(9) OSTickISR()调用 OSTimeTick()，这样 μC/OS-II 就给所有延迟任务的等待时间以及有时限地等待某事件发生的任务的等待时间的节拍参数减 1。

L14.18(10) 当所有中断服务子程序完成时，调用 OSIntExit()。如果中断服务子程序(或者其他嵌套的中断服务子程序)使一个更高优先级的任务进入了就绪态，并且当前中断服务子程序已经脱离了中断嵌套，那么 OSIntExit() 不再返回 OSTickISR()。OSIntCtxSw()恢复新任务的所有寄存器，并执行一个 IRET 指令。如果中断服务子程序没有脱离中断嵌套，或者中断服务子程序没有使更高优先级的任务进入就绪态，OSIntExit() 就返回OSTickISR()。

L14.18(11)

L14.18(12) 如果 OSIntExit()返回，则是因为 OSIntExit()没有发现更高优先级的任务。这样被中断任务的寄存器会被恢复。当执行 IRET 指令时，中断服务子程序返回，让被中断了的任务继续运行。

程序清单L14.19是OSTickISR()函数的汇编代码，程序清单L14.19中的数字标号与程序清单L14.18中的标号是相互对应的。文件中的真实代码实际上还包括注释。

程序清单L14.19 OSTickISR()

```
_OSTickISR  PROC   FAR
;
            PUSHA                                                   (1)
            PUSH   ES
            PUSH   DS
;
            MOV    AX, SEG(_OSIntNesting)                           (2)
            MOV    DS, AX
            INC    BYTE PTR DS:_OSIntNesting
;
            CMP    BYTE PTR DS:_OSIntNesting, 1                     (3)
            JNE    SHORT _OSTickISR1
            MOV    AX, SEG(_OSTCBCur)
            MOV    DS, AX
            LES    BX, DWORD PTR DS:_OSTCBCur                       (4)
            MOV    ES:[BX + 2], SS
            MOV    ES:[BX + 0], SP
;
_OSTickISR1:
            MOV    AX, SEG(_OSTickDOSCtr)                           (5)
            MOV    DS, AX
            DEC    BYTE PTR DS:_OSTickDOSCtr
            CMP    BYTE PTR DS:_OSTickDOSCtr, 0                     (6)
            JNE    SHORT _OSTickISR2
;
            MOV    BYTE PTR DS:_OSTickDOSCtr, 11                    (7)
            INT    081H
            JMP    SHORT _OSTickISR3

_OSTickISR2:
            MOV    AL, 20H                                          (8)
            MOV    DX, 20H
            OUT    DX, AL
;
_OSTickISR3:
            CALL   FAR PTR _OSTimeTick                              (9)
;
            CALL   FAR PTR _OSIntExit                               (10)
```

```
;
            POP     DS                                          (11)
            POP     ES
            POPA
;
            IRET                                                (12)
;
_OSTickISR  ENDP
```

如果时钟节拍的频率不从 18.206 48 Hz 增加到 200 Hz，便可以简化 OSTickISR()，如示意性代码程序清单 L14.20 所列。程序清单 L14.21 中的数字标号与程序清单 L14.20 中的标号相互对应。可以模仿它们在中断服务子程序中编写自己的代码。

程序清单 L14.20　18.2 Hz OSTickISR()的示意性代码

```
void OSTickISR (void)
{
    将全部寄存器保存到当前任务的堆栈中；                          (1)
    OSIntNesting ++ ;                                           (2)
    if (OSIntNesting == 1) {                                    (3)
        OSTCBCur->OSTCBStkPtr = SS:SP                           (4)
    }
    INT 81H;                                                    (5)
    OSTimeTick();                                               (6)
    OSIntExit();                                                (7)
    从当前任务的堆栈中恢复全部寄存器；                            (8)
    中断返回；                                                    (9)
}
```

L14.20(1)　同 μC/OS-II 其他的中断服务子程序一样，所有的寄存器都要保存到当前任务的堆栈中。

L14.20(2)　当进入中断服务子程序时，需要告诉 μC/OS-II，进入中断服务子程序了。可以通过调用 OSIntEnter()来实现，也可以直接给 OSIntNesting 加 1 实现。采用直接给 OSIntNesting 加 1 会快一些。

L14.20(3)

L14.20(4)　如果中断服务子程序是第 1 层，即没有中断嵌套，则需要把堆栈指针保存在当前任务的任务控制块 OS_TCB 中。

L14.20(5)　通过执行 INT 81H 指令，调用 DOS 时钟节拍处理程序（见图 F14.10："PC 中断向量表(IVT)"）。

注意：没有必要再次清中断，因为 DOS 时钟节拍处理程序已经清除了中断。

L14.20(6)　调用 OSTimeTick()，这样 μC/OS－II 便给所有延迟任务的等待时间以及有时限地等待某事件发生的任务的等待时限节拍参数减掉 1。如果中断服务子程序不是 DOS 时钟节拍，那么应该在这里添加用户的中断服务子程序。

L14.20(7)　中断服务子程序 ISR 完成后，调用 OSIntExit()。如果中断服务子程序 ISR 使优先级更高的任务进入了就绪态，则 OSIntExit() 不返回，而是做任务切换，让优先级更高的任务运行。

L14.20(8)　恢复寄存器。

L14.20(9)　执行中断返回指令 IRET，中断服务子程序返回中断源。

注意：若使用程序清单 L14.20 给出的代码，则无须调用 PC_SetTickRate() 来改变时钟频率。也就是说，用户打算使用 DOS 的时钟节拍频率，应将 OS_CFG.H 文件中的 OS_TICKS_PRE_SEC 参数由 200 改为 18。DOS 时钟频率实际上是 18.206 48，而不是 18。例如，需要延时 10 s，设定为 10 * OS_TICKS_PRE_SEC，实际延时的时间是 9.886 6 s，而不是 10 s。

程序清单 L14.21　18.2 Hz OSTickISR()

```
_OSTickISR   PROC   FAR
;
             PUSHA                                                    (1)
             PUSH   ES
             PUSH   DS
;
             MOV    AX, SEG(_OSIntNesting)                            (2)
             MOV    DS, AX
             INC    BYTE PTR DS:_OSIntNesting
;
             CMP    BYTE PTR DS:_OSIntNesting, 1                      (3)
             JNE    SHORT _OSTickISR1
             MOV    AX, SEG(_OSTCBCur)
             MOV    DS, AX
             LES    BX, DWORD PTR DS:_OSTCBCur                        (4)
             MOV    ES:[BX + 2], SS
             MOV    ES:[BX + 0], SP
;
_OSTickISR1:
             INT    081H                                              (5)
;
             CALL   FAR PTR _OSTimeTick                               (6)
;
             CALL   FAR PTR _OSIntExit                                (7)
```

```
;
            POP     DS                                                  (8)
            POP     ES
            POPA
;
            IRET                                                        (9)
;
_OSTickISR  ENDP
```

14.06 内存占用

表 T14.3 是 μC/OS-II 要用到的存储空间，包括数据和代码，取决于相应的配置常数值。如果将 μC/OS-II 用于嵌入式系统中，Data 表示 RAM，code 表示 ROM。

详细数据表在本书所附的光盘中，则具体位置如下：

\SOFTWARE\uCOS-II\Ix86L\BC45\DOC\80x86L-ROM-RAM.XLS

上述文件用 Office 2000(或者更高版本)中的 Excel 软件可以打开。可以改变配置常数的值(以红色表示)，看看它们怎样影响 80x86 占用的 ROM 和 RAM。??? _EN 的值必须取 0 或 1。

笔者是用 Borland 编译器编译代码的，选择速度优先。表中给出的字节数并不是精确的数字，只是告知 μC/OS-II 的功能函数大概占用多少内存。例如，假设不需要消息队列(OS_Q_EN 设为 0)，可以节省 1 900～2 200 B 的代码空间。

在 OS_CFG.H 文件中改变 OS_ARG_CHK_EN 的值，代码长短会有变化，表中已经列出。无须改变 OS_ARG_CHK_EN 的值去观察这一变化。

Data(数据)一栏不易懂。注意，空闲任务(idle)和统计任务(statistics)的堆栈都设为 1 024 B(1 KB)。根据用户的要求，它们堆栈的容量可以增减。如果配置成允许的最多任务数(62 个用户任务)，那么 μC/OS-II 内部的数据结构最少需要占用 3 500 B 的 RAM。

表 T14.4 说明了如何禁止掉多数 μC/OS-II 的功能函数，裁剪 μC/OS-II，使之占用尽量少的内存。此处的小系统最多有 16 个任务，20 个优先级。代码空间为 2 400～2 700 B，数据空间大约 500 B。虽然仍可以运行多任务，但任务中可以使用的功能函数只剩下 OSTimeDly()了，但仍可以运行多任务。

如果使用的是 80x86 CPU，那么一般情况下不太会受存储器容量的限制。在多数情况下，μC/OS-II 也不会是占用内存最多的模块。

表 T14.3　μC/OS－II 的最大配置

配置参数	在 OS_CFG.H 中的值	数据/B	代码/B OS_ARG_CHK_EN==0	代码/B OS_ARG_CHK_EN==1	代码变化/B	代码变化率/%
总　数		**5 523**	**13 048**	**14 919**	**1 871**	**14**
OS_MAX_EVENTS	10	164				
OS_MAX_FLAGS	**2**	**14**				
OS_MAX_MEM_PART	2	44				
OS_MAX_QS	**2**	**52**				
OS_MAX_TASKS	62	2 880				
OS_LOWEST_PRIO	63	264				
OS_TASK_IDLE_STK_SIZE	512	1 024				
OS_TASK_STAT_EN	1	10	351	351		
OS_TASK_STAT_STK_SIZE	**512**	**1 024**				
OS_ARG_CHK_EN	**1**					
OS_CPU_HOOKS_EN	1					
最小占用			2 177	2 493	316	
OS_FLAG_EN	1		2 174	2 539	82	
OS_FLAG_WAIT_CLR_EN	**1**				**108**	
OS_FLAG_ACCEPT_EN	1				41	
OS_FLAG_DEL_EN	**1**				**95**	
OS_FLAG_QUERY_EN	1				39	
OS_MBOX_EN	1		958	1 185	55	
OS_MBOX_ACCEPT_EN	**1**				**23**	
OS_MBOX_DEL_EN	1				49	
OS_MBOX_POST_EN	**1**				**36**	
OS_MBOX_POST_OPT_EN	1				39	
OS_MBOX_QUERY_EN	**1**				**25**	
OS_MEM_EN	**1**		**689**	**838**	**123**	

续表 T14.3

配置参数	在 OS_CFG.H 中的值	代码/B OS_ARG_CHK_EN==0	代码/B OS_ARG_CHK_EN==1	代码变化/B 代码变化率/%
OS_MEM_QUERY_EN	1			26
OS_MUTEX_EN	1	1 596	1 792	83
OS_MUTEX_ACCEPT_EN	1			39
OS_MUTEX_DEL_EN	1			47
OS_MUTEX_QUERY_EN	1			27
OS_Q_EN	1	1 917	2 206	45
OS_Q_ACCEPT_EN	1			23
OS_Q_DEL_EN	1			49
OS_Q_FLUSH_EN	1			25
OS_Q_POST_EN	1			40
OS_Q_POST_FRONT_EN	1			40
OS_Q_POST_OPT_EN	1			40
OS_Q_QUERY_EN	1			27
OS_SEM_EN	1	707	864	62
OS_SEM_ACCEPT_EN	1			21
OS_SEM_DEL_EN	1			49
OS_SEM_QUERY_EN	1			25
OS_TASK_CHANGE_PRIO_EN	1	444	466	22
OS_TASK_CREATE_EN	1	185	196	11
OS_TASK_CREATE_EXT_EN	1	441	467	26
OS_TASK_DEL_EN	1	527	578	51
OS_TASK_SUSPEND_EN	1	264	300	36
OS_TASK_QUERY_EN	1	87	103	16
OS_TIME_DLY_HMSM_EN	1	248	248	
OS_TIME_DLY_RESUME_EN	1	122	132	10
OS_TIME_GET_SET_EN	1	59	59	
OS_SCHED_LOCK_EN	1	102	102	

续表 T14.3

配置参数	在 OS_CFG.H 中的值	数据/B	代码/B OS_ARG_CHK_EN==0	代码/B OS_ARG_CHK_EN==1	代码变化/B	代码变化率/%
μC/OS－II 内部占用		47				
应用程序堆栈	0					
应用程序占用 RAM	0					

表 T14.4　μC/OS－II 的最小配置

配置参数	在 OS_CFG.H 中的值	数据/B	代码/B OS_ARG_CHK_EN==0	代码/B OS_ARG_CHK_EN==1	代码变化/B	代码变化率/%
总　数		1 508	2 362	2 689	327	14
OS_MAX_EVENTS	10					
OS_MAX_FLAGS	2					
OS_MAX_MEM_PART	2					
OS_MAX_QS	2					
OS_MAX_TASKS	16	360				
OS_LOWEST_PRIO	20	87				
OS_TASK_IDLE_STK_SIZE	512	1 024				
OS_TASK_STAT_EN	0					
OS_TASK_STAT_STK_SIZE	512					
OS_ARG_CHK_EN	1					
OS_CPU_HOOKS_EN	1					
最小占用			2 177	2 493	316	
OS_FLAG_EN	0					
OS_FLAG_WAIT_CLR_EN	1					
OS_FLAG_ACCEPT_EN	1					
OS_FLAG_DEL_EN	1					
OS_FLAG_QUERY_EN	1					
OS_MBOX_EN	0					
OS_MBOX_ACCEPT_EN	1					
OS_MBOX_DEL_EN	1					
OS_MBOX_POST_EN	1					

续表 T14.4

配置参数	在 OS_CFG.H 中的值	代码/B OS_ARG_CHK_EN==0 数据/B	代码/B OS_ARG_CHK_EN==1	代码变化/B 代码变化率/%
OS_MBOX_POST_OPT_EN	1			
OS_MBOX_QUERY_EN	1			
OS_MEM_EN	0			
OS_MEM_QUERY_EN	1			
OS_MUTEX_EN	0			
OS_MUTEX_ACCEPT_EN	1			
OS_MUTEX_DEL_EN	1			
OS_MUTEX_QUERY_EN	1			
OS_Q_EN	0			
OS_Q_ACCEPT_EN	1			
OS_Q_DEL_EN	1			
OS_Q_FLUSH_EN	1			
OS_Q_POST_EN	1			
OS_Q_POST_FRONT_EN	1			
OS_Q_POST_OPT_EN	1			
OS_Q_QUERY_EN	1			
OS_SEM_EN	0			
OS_SEM_ACCEPT_EN	1			
OS_SEM_DEL_EN	1			
OS_SEM_QUERY_EN	1			
OS_TASK_CHANGE_PRIO_EN	0			
OS_TASK_CREATE_EN	1	185	196	11
OS_TASK_CREATE_EXT_EN	0			
OS_TASK_DEL_EN	0			
OS_TASK_SUSPEND_EN	0			
OS_TASK_QUERY_EN	0			
OS_TIME_DLY_HMSM_EN	0			
OS_TIME_DLY_RESUME_EN	0			

续表 T14.4

配置参数	在 OS_CFG.H 中的值	代码/B OS_ARG_CHK_EN==0	代码变化/B	
		数据/B	代码/B OS_ARG_CHK_EN==1	代码变化率/%
OS_TIME_GET_SET_EN	0			
OS_SCHED_LOCK_EN	0			
μC/OS-Ⅱ内部占用		37		
应用程序堆栈	0			
应用程序占用 RAM	0			

表 T14.5 为 80x86 数据宽度。

表 T14.5　80x86 数据宽度

数据结构	字　节
编译对齐	2
布尔型	1
INT8S	1
INT8U	1
INT16U	2
INT32U	4
OS_FLAGS	2
OS_STK	2
指　针	4

第 15 章

μC/OS－II 在带有硬件浮点运算单元的 80x86 上的移植

实模式，大模式下编译

本章介绍如何将 μC/OS－II 移植到带浮点运算单元 FPU(Float Point Unit)的 Intel 80x86 系列 CPU 上。能够利用本移植代码的处理器有 Intel 80486™，Pentiums™(所有类型)，Xeon™，AMD Athlon™，K6™ 系列及 ElanSC520™ 等。本移植假设使用 Borland C/C++ V4.51 编译器生成大内存模式(large-memory model)代码，并假设处理器运行在实模式下。本移植代码与第 14 章的代码很相近，只介绍不同之处。

在本移植中假设允许使用 OSTaskCreateExt()函数(通过在头文件 OS_CFG.H 中将 OS_TASK_CREATE_EXT_EN置为 1)，且允许使用内存管理功能(通过在 OS_CFG.H 文件中将 OS_MEM_EN 置为 1)。当然，须将 OS_MAX_MEM_PART 设置为不小于 1。最后，带浮点运算操作的任务须通过调用 OSTaskCreateExt()和设置 OS_TASK_OPT_SAVE_FP 选项来建立。

图 F15.1 表示了 80x86 处理器实模式下的编程模式。整数寄存器与第 14 章介绍的一样。实际上，它们也是用同样的方法保存和恢复。本移植与第 14 章的移植之间惟一的不同点是，还需要保存和恢复浮点运算单元 FPU 的寄存器。这一步通过 context-switch-hook 函数完成。

15.00 开发工具

与第 14 章一样，采用 Borland C/C++ V4.51 编译器和 Borland Turbo Assembler 汇编器完成程序的移植和测试。编译器产生可重入型代码，支持 C 程序中嵌入汇编语句，可以让编译器生成使用 FPU 的代码。笔者将代码在一台 300 MHz 的 Pentium－II 计算机上完成测试，操作系统是 Microsoft Windows 2000。实际上编译器生成的是 DOS 可执行文件，在 Windows的 DOS 窗口中运行。

最后，还可以修改本章提供的移植代码，使之适合别的 80x86 编译器，只要这些编译器可以生成实模式代码。如果开发环境不同，则很可能只须更改一下编译器和汇编器的设置即可。

表 T15.1 是 Borland C/C++ V4.51 编译器提供的命令行选项，这些设置用于编译移植代码和第 1 章提供的范例。

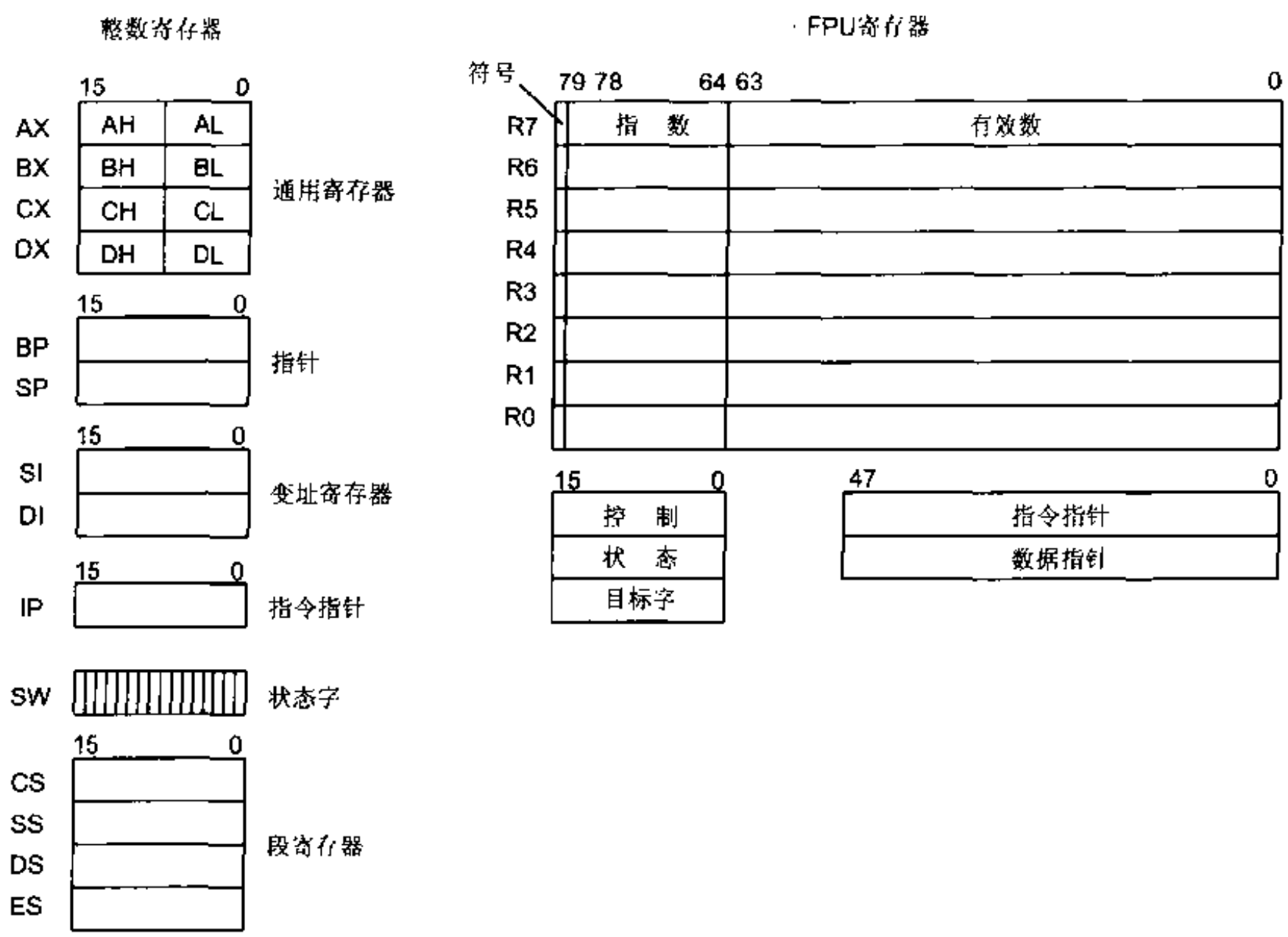


图 F15.1 80x86 实模式下的内部寄存器

表 T15.1 用于编译移植范例的编译器选项

选 项	描 述
-1	生成 80186 代码
-B	若有在线汇编语句,则编译并调用汇编程序处理
-c	编译生成 .OBJ 文件,不链接
-d	合并重复字符串
-f287	使用协处理器 80287 FPU 的硬件指令
-G	生成速度最优化代码
-I	编译器包含文件路径为 C:\BC45\INCLUDE
-k-	标准栈结构
-L	编译器库文件路径为 C:\BC45\LIB
-ml	大模式下编译
-N-	不检查堆栈溢出
-n..\obj	存放目标文件路径 ..\OBJ

续表 T15.1

选　项	描　述
-O	优化跳转语句
-Ob	删除无用代码
-Oe	分配寄存器给全局变量，并分析变量作用期
-Og	全程优化，整个函数内消除重复子表达式
-Oi	扩展公共内联函数
-Ol	循环优化
-Om	恒定代码转移，将不变代码转移到循环体外
-Op	复制传播，即复制相同的常量、变量及表达式
-Ov	归纳变量，削减计算强度
-v	打开源码调试
-vi	内联函数展开
-wpro	错误报告：无原型函数调用
-Z	允许寄存器优化

表 T15.2 是命令行提供的 Borland Turbo Assembler V4.0 选项。这些设置用于汇编移植代码的 OS_CPU_A.ASM 文件。

表 T15.2　汇编 .ASM 文件时使用的汇编选项

选　项	描　述
/MX	全局变量和标号区分大小写
/ZI	目标文件包含全部调试信息
/O	生成代码覆盖同名文件

15.01　目录和文件

在安装 μC/OS-II 时，附带光盘中的安装程序将把针对 Intel 80x86（实模式、大模式下编译，且带硬件浮点运算单元 FPU）的移植代码安装到 \SOFTWARE\uCOS-II\Ix86L-FP\BC45 目录下。

目录名称代表 Intel 80x86 实模式，大模式 L 下编译，且支持硬件浮点 FP 指令，放在 BC45 目录下。移植部分的代码可在下述文件中找到：OS_CPU.H，OS_CPU_C.C 以及 OS_CPU_A.ASM。

15.02 INCLUDES.H 文件

程序清单 L15.1 是本 80x86 移植的 INCLUDE.H 文件内容，与第 14 章的内容是一样的。INCLUDES.H 不涉及真正移植部分的内容，但由于在编译移植代码时需要它，所以仍在这里说明。

程序清单 L15.1 INCLUDES.H

```
#include    <stdio.h>
#include    <string.h>
#include    <ctype.h>
#include    <stdlib.h>
#include    <conio.h>
#include    <dos.h>
#include    <math.h>
#include    <setjmp.h>

#include    "os_cpu.h"
#include    "os_cfg.h"
#include    "ucos_ii.h"
#include    "pc.h"
```

15.03 OS_CPU.H 文件

OS_CPU.H 文件中包含与处理器相关的常量、宏及结构体的定义。程序清单 L15.2 是为 80x86 编写的 OS_CPU.H 文件的内容。该文件的大部分内容与第 14 章相同。

程序清单 L15.2 OS_CPU.H

```
#ifdef  OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT  extern
#endif

typedef unsigned char  BOOLEAN;                                    (1)
typedef unsigned char  INT8U;
typedef signed   char  INT8S;
typedef unsigned int   INT16U;
typedef signed   int   INT16S;
```

```
typedef unsigned long  INT32U;
typedef signed   long  INT32S;
typedef float          FP32;                                    (2)
typedef double         FP64;

typedef unsigned int   OS_STK;                                  (3)
typedef unsigned short OS_CPU_SR;                               (4)
```

15.03.01 OS_CPU.H,数据类型

L15.2(1) 如果参考 Borland 编译器的文档,则会发现 int 型是 16 位, long 型是 32 位。

L15.2(2) 因为假设任务中要进行浮点操作,所以浮点数据类型也被包含其中;然而 μC/OS-II本身不使用浮点数。

L15.2(3) 实模式运行的 80x86 处理器堆栈入口为 16 位宽,OS_STK 就是据此声明的。本移植中堆栈宽度不改变。所有的任务堆栈必须以 OS_STK 为数据类型。

L15.2(4) 实模式下运行的 80x86 处理器的状态寄存器(也称为处理器标志)也是 16 位宽。OS_CPU_SR 数据类型只在关中断方式 OS_CRITICAL_METHOD 为 3 时使用,在该移植中不使用。包含了 OS_CPU_SR 数据类型,是考虑到用户使用不同编译器时,可能会用到 OS_CRITICAL_METHOD #3。

15.03.02 OS_CPU.H,OS_ENTER_CRITICAL()及 OS_EXIT_CRITICAL()

程序清单 L15.2 OS_CPU.H(续)

```
#define  OS_CRITICAL_METHOD    2                                (5)

#define  OS_ENTER_CRITICAL()   asm {PUSHF; CLI}                 (6)
#define  OS_EXIT_CRITICAL()    asm  POPF
```

L15.2(5) 对于本移植,首选的临界段保护方法是第 2 种,因为是编译器直接支持的。

L15.2(6) 实现 OS_ENTER_CRITICAL()的方法是,将原中断状态保存到堆栈中,然后关中断。这一步在 80x86 上是通过执行 PUSHF 指令和 CLI 指令实现的。OS_EXIT_CRITICAL()只需简单地执行 POPF 指令,恢复处理器中 SW 寄存器原来的内容。

15.03.03 OS_CPU.H,堆栈方向

程序清单 L15.2 OS_CPU.H(续)

```
#define  OS_STK_GROWTH         1                                (7)
```

L15.2(7) 80x86 处理器的堆栈从内存高地址向低地址递减，故 OS_STK_GROWTH 必须设置为 1。

15.03.04 OS_CPU.H，OS_TASK_SW()

程序清单 L15.2 OS_CPU.H(续)

```
#define  uCOS              0x80                          (8)

#define  OS_TASK_SW()      asm  INT   uCOS               (9)
```

L15.2(9) 为了任务切换，OS_TASK_SW()须模仿一次中断过程。80x86 有 256 个软中断可供选用。中断服务子程序 ISR(也称为异常处理)的入口地址必须指向汇编函数 OSCtxSw()(见文件 OS_CPU_A.ASM)。这里须确认 0x80 中断向量指向 OSCtxSw()。

L15.2(8) 在 PC 机上测试了代码，并决定用向量号为 128(0x80)的中断。

15.03.05 OS_CPU.H，时钟节拍频率

将 PC 时钟节拍频率从标准的 18.206 48 Hz 改为 200 Hz(详述请见第 14 章)。

程序清单 L15.2 OS_CPU.H(续)

```
OS_CPU_EXT  INT8U  OSTickDOSCtr;                          (10)
```

L15.2(10) 在这里声明了一个 8 位变量 OSTickDOSCtr，用来保存时钟节拍发生的次数。每发生 11 次，就调用 DOS 的时钟节拍函数 1 次，从而实现与 DOS 时钟的同步。OSTickDOSCtr 在 OS_CPU_A.ASM 中使用，是专门为 PC 环境而定义的。

15.03.06 OS_CPU.H，浮点函数

本移植定义了 3 个专门针对 80x86 浮点运算功能的函数。换句话说，在本移植中增加了 3 个函数，用于同浮点处理单元硬件打交道。

程序清单 L15.2 OS_CPU.H(续)

```
void     OSFPInit(void);                                  (11)
void     OSFPRestore(void *pblk);                         (12)
void     OSFPSave(void *pblk);                            (13)
```

L15.2(11) 初始化浮点处理单元的函数。

L15.2(12) OSFPRestore()用于在任务切换时恢复浮点寄存器。实际上该函数是用汇编语言编写的，所以放在 OS_CPU_A.ASM 文件中。

L15.2(13) OSFPSave()用于在任务挂起时保存浮点寄存器中的内容。该函数也是用汇编语言编写的,放在 OS_CPU_A.ASM 文件中。

15.04 OS_CPU_C.C 文件

如第 13,14 章描述的,μC/OS-II 的移植须写 10 个非常简单的 C 函数:

OSTaskStkInit();
OSTaskCreateHook();
OSTaskDelHook();
OSTaskSwHook();
OSTaskIdleHook();
OSTaskStatHook();
OSTimeTickHook();
OSInitHookBegin();
OSInitHookEnd();
OSTCBInitHook()。

只有 OSTaskStkInit()函数在 μC/OS-II 中是必要的。另外 9 个函数必须声明,但可以不包含任何代码。然而,本移植还需要函数 OSTaskCreateHook(),OSTaskDelHook(),OSTaskSwHook()及 OSInitHookEnd()。

#define constant OS_CPU_HOOKS_EN(见 OS_CFG.H 文件)必须设置为 1。

15.04.01 任务堆栈初始化函数 OSTaskStkInit()

该函数由 OSTaskCreate()或 OSTaskCreateExt()调用,与第 14 章描述的函数一样,用于初始化任务的堆栈。堆栈的结构被初始化成看起来好像刚刚发生过中断一样,处理器的所有寄存器被推入堆栈。图 F15.2(同图 F14.3)给出了 OSTaskStkInit()初始化后的堆栈内容。

注意:图中示意的不是调用 OSTaskStkInit()函数的任务的堆栈结构,而是新建立任务的堆栈结构。同时,堆栈中只有整数寄存器,而不包括浮点寄存器。

下面将介绍如何处理浮点寄存器。

程序清单 L15.3 是 OSTaskStkInit()函数的源代码。它与第 14 章中的程序清单 L14.3 相同。

程序清单 L15.3 OS_CPU_C.C, OSTaskStkInit()

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

```
    opt      = opt;
    stk      = (INT16U *)ptos;
    *stk--   = (INT16U)FP_SEG(pdata);
    *stk--   = (INT16U)FP_OFF(pdata);
    *stk--   = (INT16U)FP_SEG(task);
    *stk--   = (INT16U)FP_OFF(task);
    *stk--   = (INT16U)0x0202;
    *stk--   = (INT16U)FP_SEG(task);
    *stk--   = (INT16U)FP_OFF(task);
    *stk--   = (INT16U)0xAAAA;
    *stk--   = (INT16U)0xCCCC;
    *stk--   = (INT16U)0xDDDD;
    *stk--   = (INT16U)0xBBBB;
    *stk--   = (INT16U)0x0000;
    *stk--   = (INT16U)0x1111;
    *stk--   = (INT16U)0x2222;
    *stk--   = (INT16U)0x3333;
    *stk--   = (INT16U)0x4444;
    *stk     = _DS;
    return   ((OS_STK *)stk);
}
```

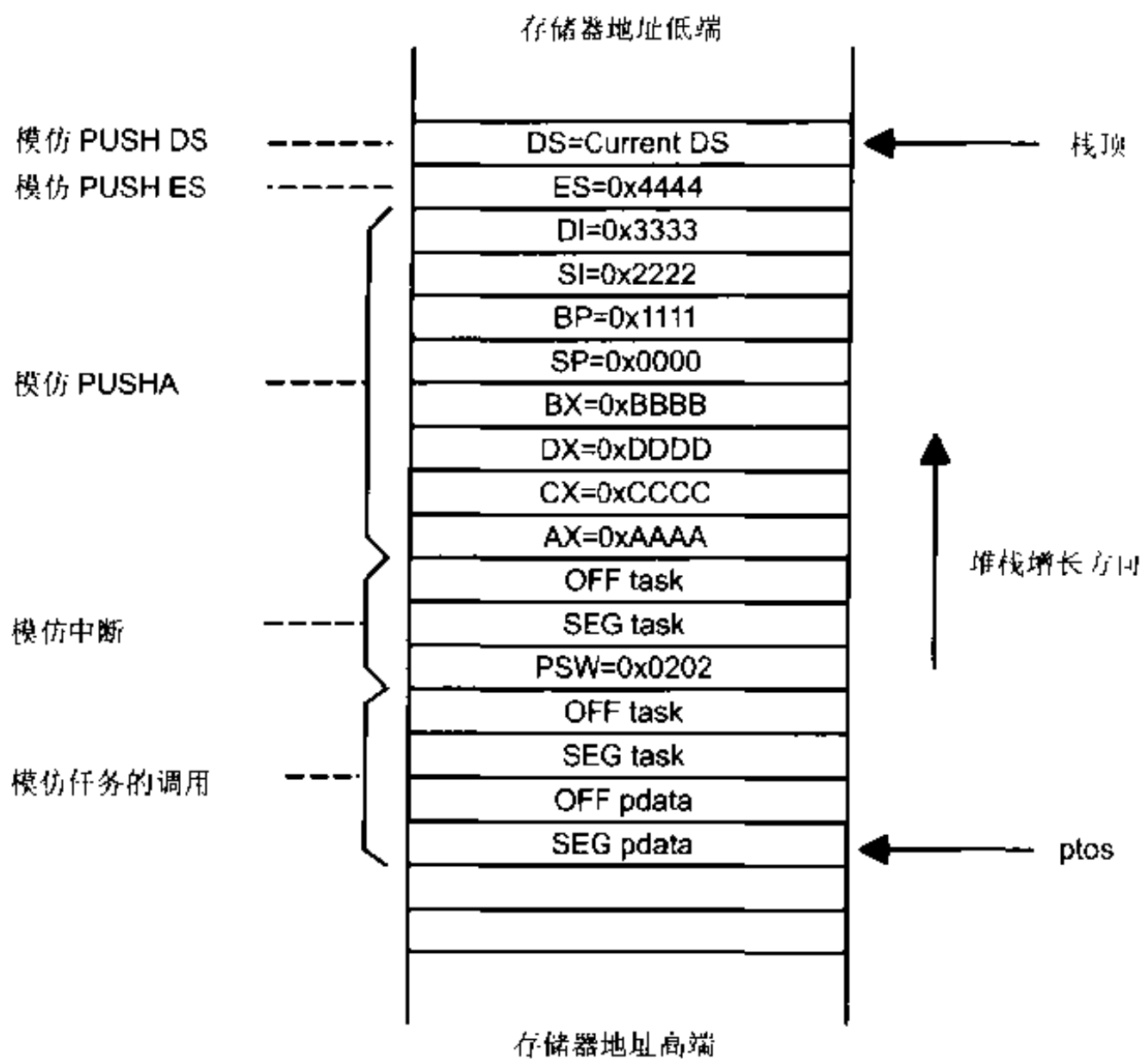


图 F15.2　传递了 pdata 参数的堆栈初始化结构

15.04.02 浮点处理单元初始化函数 OSFPInit()

在 OSInit()函数的末尾,OSInitHookEnd()函数将调用 OSFPInit(),初始化 μC/OS-II 的内部结构。OSFPInit()函数主要用于初始化本章提及的浮点寄存器切换机制。OSFPInit()认为用户打算使用 μC/OS-II 的内存管理功能(必须在 OS_CFG.H 中将 OS_MEM_EN设置为1)。OSFPInit()函数的源代码在程序清单 L15.4 中给出。

程序清单 L15.4 OS_CPU_C.C, OSFPInit()

```
#define OS_NTASKS_FP        (OS_MAX_TASKS + OS_N_SYS_TASKS - 1)                    (1)
#define OS_FP_STORAGE_SIZE 128                                                     (2)

static  OS_MEM  *OSFPPartPtr;                                                      (3)
static  INT32U  OSFPPart[OS_NTASKS_FP][OS_FP_STORAGE_SIZE / sizeof(INT32U)];       (4)

void  OSFPInit (void)
{
    INT8U     err;
#if OS_TASK_STAT_EN
    OS_TCB   *ptcb;
    void     *pblk;
#endif

    OSFPPartPtr = OSMemCreate(&OSFPPart[0][0],                                     (5)
                  OS_NTASKS_FP,
                  OS_FP_STORAGE_SIZE,
                  &err);

#if OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_EN
    ptcb              = OSTCBPrioTbl[OS_STAT_PRIO];                                (6)
    ptcb->OSTCBOpt |= OS_TASK_OPT_SAVE_FP;                                         (7)
    pblk              = OSMemGet(OSFPPartPtr, &err);                               (8)
    if (pblk != (void *)0) {                                                       (9)
        ptcb->OSTCBExtPtr = pblk;                                                  (10)
        OSFPSave(pblk);                                                            (11)
    }
#endif
}
```

L15.4(1) 尽管不是 OSFPInit()函数的一部分,还是定义了该常量,用于确定需要多少存储空间保存 FPU 寄存器。这里,分配的存储空间块数为用户任务数加1,

加的 1 是给统计任务用的，如下所述。

L15.4(2) 80x86 FPU 需要保存 108 B。为便于将来扩充，决定分配 128 B。如果内存紧张，则可以设为 108 B，为每个任务可节省 20 B。

L15.4(3) 用一个 μC/OS-II 内存分区保存 FPU 寄存器内容。OSFPPartPtr 是一个指针，指向为此建立的分区。由于 OSFPPartPtr 声明为静态变量，用户应用程序不能访问 OSFPPartPtr。

L15.4(4) OSFPPart[][]是实际的分区，保存了所有任务的 FPU 寄存器内容。至少要为它提供 (OS_MAX_TASKS + 1) ×128 B 的 RAM。因为 OSFPPart[][]声明为静态，用户应用程序不能访问该数组。

L15.4(5) OSFPInit()将此分区信息告知 μC/OS-II。OSMemCreate()函数把一个内存分区分成多个内存块(每块 128 B)，并且将这些块链接成单向链表。当需要一个内存块保存 FPU 寄存器时，只须调用 OSMemGet()函数(在 OSTaskCreateHook()一节中讨论)。

L15.4(6) 改变 OS_TaskStat()属性，使之能进行浮点数学运算。读者可能会对此感到疑惑，因为 OS_TaskStat()函数不进行任何浮点数操作。这么做是为了使用户可以通过 OSTaskStatHook()来扩展 OS_TaskStat()函数的功能，使之可以做浮点运算。在此，OSFPInit()得到统计任务的任务控制块 OS_TCB 的指针。

L15.4(7) .OSTCBOpt 标志设置为 1，用于指明统计任务 OS_TaskStat()是需要保存和恢复浮点寄存器的，因为 μC/OS-II 默认该选项为 0。

L15.4(8) 得到一个缓冲区，用于在统计任务 OS_TaskStat()挂起时，保存该任务的浮点寄存器内容。

L15.4(9) 检查一下指针的合法性总是有益的。

L15.4(10) 让保存在 OS_TCB 的扩展变量 .OSTCBExtPtr 中的指针指向保存 FPU 内容的存储区。

这个过程使得任务切换代码能够知道浮点寄存器保存在哪里。

L15.4(11) 调用 OSFPSave()函数(见 OS_CPU_A.ASM 文件)，将 FPU 寄存器当前内容保存在 pblk 指向的地方。

这一过程中，FPU 包含的内容是什么无关紧要，重要的是 FPU 寄存器包含了有效的数值。OSFPSave()函数在第 15.05.05 小节："OSFPSave()"中讨论。

注意：应用代码不会做任何浮点异常处理(例如被 0 除)，这是因为 μC/OS-II 中没有与浮点数相关的任何操作。可以通过增加数据范围检查代码，来避免运行中出现异常。应该养成一种习惯，在应用程序中相关操作前，检查数据范围(例如被 0 除等)出现异常的可能性。

15.04.03 任务建立接口函数 OSTaskCreateHook()

程序清单 L15.5 是 OSTaskCreateHook()函数的源代码。

OSTaskCreateHook()是被 OS_TCBInit()函数调用的，后者在 OSTaskCreate() 或者

OSTaskCreateExt()中被调用。

程序清单 L15.5 OS_CPU_C.C, OSTaskCreateHook()

```
void  OSTaskCreateHook (OS_TCB *ptcb)
{
    INT8U   err;
    void   *pblk;


    if (ptcb->OSTCBOpt & OS_TASK_OPT_SAVE_FP) {                          (1)
        pblk = OSMemGet(OSFPPartPtr, &err);                              (2)
        if (pblk != (void *)0) {                                         (3)
            ptcb->OSTCBExtPtr = pblk;                                    (4)
            OSFPSave(pblk);                                              (5)
        }
    }
}
```

L15.5(1) 如果建立一个进行浮点运算的任务，则必须将 OSTaskCreateExt()函数的 opt 参数中 OS_TASK_OPT_SAVE_FP 位设置成 1。该选项告知函数 OSTaskCreateHook()，函数在任务中使用 FPU，于是需要在任务切换过程中保存或恢复这些寄存器的值。

L15.5(2) 因为要建立一个使用 FPU 的任务，需要为 FPU 寄存器分配存储空间。

L15.5(3) 再次进行指针合法性检查。

L15.5(4) 存储空间的地址被保存进新建立任务的任务控制块 OS_TCB 中。

L15.5(5) 再次调用 OSFPSave()函数(见 OS_CPU_A.ASM 文件)，将 FPU 寄存器当前内容保存在 pblk 指向的地方。在这一过程中，FPU 包含的内容是什么无关紧要，重要的是，FPU 寄存器包含了有效的数值。OSFPSave()函数在第 15.05.05 小节："OSFPSave()"中讨论。

图 F15.3 是 OSTaskCreateHook()函数执行之后一些数据结构之间的关系。

15.04.04 任务删除接口函数 OSTaskDelHook()

OSTaskDel()通过调用 OSTaskDelHook()函数扩展其功能。因为在建立任务时，分配了一个内存块，用于保存浮点寄存器的内容，所以在任务删除时，需要释放该内存块。程序清单 L15.6 描述了 OSTaskDelHook()是如何实现这个功能的。

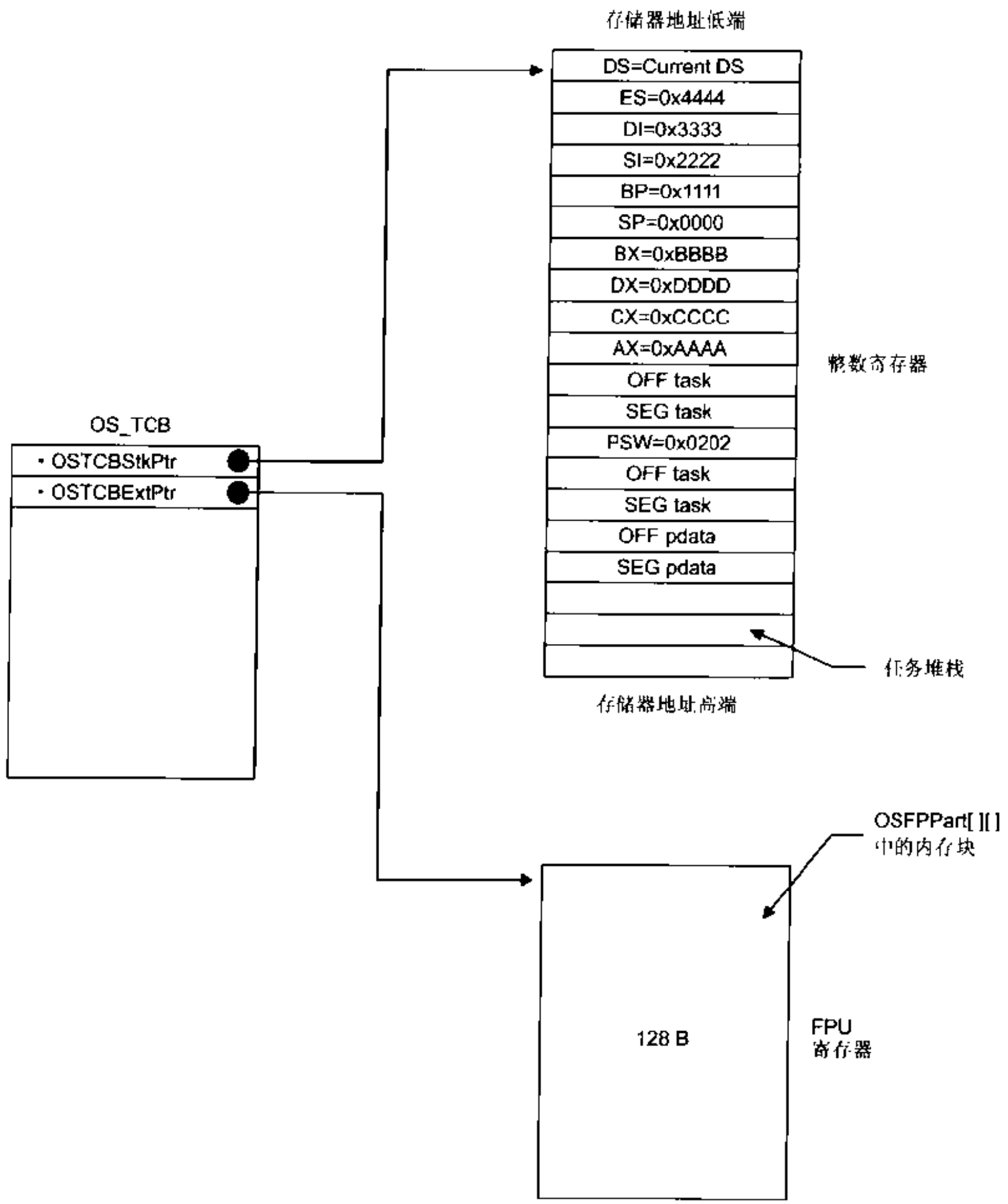


图 F15.3 堆栈的初始化和 FPU 寄存器内容的保存

程序清单 L15.6 OS_CPU_C.C, OSTaskDelHook()

```
void OSTaskDelHook (OS_TCB *ptcb)
{
    if (ptcb->OSTCBOpt & OS_TASK_OPT_SAVE_FP) {                    (1)
        if (ptcb->OSTCBExtPtr != (void *)0) {                      (2)
```

```
            OSMemPut(OSFPPartPtr, ptcb->OSTCBExtPtr);                    (3)
        }
    }
}
```

L15.6(1)

L15.6(2) 首先须确认已经分配了一个内存块,用于保存浮点寄存器内容。

L15.6(3) 把内存块释放到原来的内存分区。

15.04.05 任务切换接口函数 OSTaskSwHook()

OSTaskSwHook()函数用于扩展任务切换代码的功能。OSTaskSwHook()函数被OSStartHighRdy()、任务级任务切换函数OSCtxSw()及中断级任务切换函数OSIntCtxSw()调用。程序清单L15.7给出了OSTaskSwHook()函数的实现过程。

程序清单 L15.7 OS_CPU_C.C, OSTaskSwHook()

```
void  OSTaskSwHook (void)
{
    INT8U   err;
    void    *pblk;

    if (OSRunning == TRUE) {                                             (1)
        if (OSTCBCur->OSTCBOpt & OS_TASK_OPT_SAVE_FP) {                  (2)
            pblk = OSTCBCur->OSTCBExtPtr;
            if (pblk != (void *)0) {                                     (3)
                OSFPSave(pblk);                                          (4)
            }
        }
    }

    if (OSTCBHighRdy->OSTCBOpt & OS_TASK_OPT_SAVE_FP) {                  (5)
        pblk = OSTCBHighRdy->OSTCBExtPtr;
        if (pblk != (void *)0) {                                         (6)
            OSFPRestore(pblk);                                           (7)
        }
    }
}
```

L15.7(1) 当OSStartHighRdy()调用OSTaskSwHook()函数时,试图恢复最高优先级任务的浮点寄存器内容。若调用OSStartHighRdy()函数时,OSRunning为FALSE,则表明此时还没有启动多任务,OSTaskSwHook()不必保存浮点寄存器内容。

L15.7(2) 如果 OSTaskSwHook()是被 OSCtxSw()或者 OSIntCtxSw()函数调用的，则表明正在挂起一个低优先级任务，应该检查任务在建立时是否带浮点选项。

L15.7(3) 为了保险起见，须检查 .OSTCBExtPtr 的内容，确认不是一个空指针；它也不应该是。

L15.7(4) 调用 OSFPSave()函数，将浮点寄存器当前内容保存到为之分配的相应内存块中。

L15.7(5) 确认即将恢复运行的任务(或者说高优先级任务)是带浮点选项建立的。换句话说，该函数检查是否已告知 OSTaskCreateExt()，新建立的任务需要进行浮点运算。

L15.7(6) 为了保险起见，须检查 .OSTCBExtPtr 的内容，确认不是一个空指针。

L15.7(7) 调用 OSFPRestore()函数(见 OS_CPU_A.ASM 文件)，将 FPU 寄存器当前内容从 pblk 指向的区域中恢复。OSFPRestore()函数的代码见 15.05.06 小节："OSFPRestore()"。

15.04.06 空闲任务接口函数 OSTaskIdleHook()

在 OS_CPU_C.C 中该函数什么也不做。该函数的源代码如程序清单 L15.8 所列。

程序清单 L15.8 OS_CPU_C.C, OSTaskIdleHook()

```
void  OSTaskIdleHook (void)
{
}
```

15.04.07 统计任务接口函数 OSTaskStatHook()

在 OS_CPU_C.C 中该函数什么也不做。至于用户可以利用这个函数做些什么，请见第 1 章的范例 3。该函数的源代码如程序清单 L15.9 所列。

程序清单 L15.9 OS_CPU_C.C, OSTaskStatHook()

```
void  OSTaskStatHook (void)
{
}
```

15.04.08 时钟节拍接口函数 OSTimeTickHook()

在 OS_CPU_C.C 中该函数什么也不做。其源代码见程序清单 L15.10。

程序清单 L15.10 OS_CPU_C.C, OSTimeTickHook()

```
void  OSTimeTickHook (void)
{
}
```

15.04.09 系统初始化开始接口函数 OSInitHookBegin()

在 OS_CPU_C.C 中该函数什么也不做。其源代码见程序清单 L15.11。

程序清单 L15.11 OS_CPU_C.C, OSInitHookBegin()

```
void  OSInitHookBegin (void)
{
}
```

15.04.10 系统初始化结束接口函数 OSInitHookEnd()

OSInit()在返回之前调用 OSInitHookEnd()函数。这意味着 OSInit()初始化了 μC/OS-II的内存管理功能(使用本移植,必须在 OS_CFG.H 文件中将 OS_MEM_EN 设置为 1)。OSInitHook()简单地调用 OSFPInit()函数(见 15.04.02 小节:"浮点处理单元初始化函数 OSFPInit()"),后者负责设置保留的内存分区,该内存分区用来保存各个任务的浮点寄存器。OSInitHookEnd()函数的源代码如程序清单 L15.12 所列。

程序清单 L15.12 OS_CPU_C.C, OSInitHookEnd()

```
void  OSInitHookEnd (void)
{
    OSFPInit();
}
```

15.04.11 控制块初始化接口函数 OSTCBInitHook()

在 OS_CPU_C.C 中该函数什么也不做。

其源代码见程序清单 L15.13。

程序清单 L15.13 OS_CPU_C.C, OSTCBInitHook()

```
void  OSTCBInitHook (void)
{
}
```

15.05 OS_CPU_A.ASM 文件

μC/OS-II 的移植需要改写 OS_CPU_A.ASM 中的 4 个函数:

OSStartHighRdy();

OSCtxSw();

OSIntCtxSw();

OSTickISR()。

在 OS_CPU_A.ASM 文件中，还须增加 2 个函数：OSFPSave()和 OSFPRestore()。这 2 个函数的作用是，在任务切换时单独保存浮点寄存器。

15.05.01 OSStartHighRdy()

该函数由 OSStart()函数调用，功能是运行优先级最高的就绪态任务。它与第 14 章描述的 OSStartHighRdy()函数类似(见 14.05.01 小节："OSStartHighRdy()")，程序清单 L15.14 是该函数的源代码。

程序清单 L15.14 OSStartHighRdy()

```
_OSStartHighRdy  PROC FAR

            MOV     AX, SEG _OSTCBHighRdy
            MOV     DS, AX
;
            CALL    FAR PTR _OSTaskSwHook
;
            MOV     AL, 1
            MOV     BYTE PTR DS:_OSRunning, AL
;
            LES     BX, DWORD PTR DS:_OSTCBHighRdy
            MOV     SS, ES:[BX+2]
            MOV     SP, ES:[BX+0]
;
            POP     DS
            POP     ES
            POPA
;
            IRET

_OSStartHighRdy  ENDP
```

15.05.02 OSCtxSw()

OSCtxSw()用于实现任务级的任务切换(在任务中调用，区别于在中断程序中调用的 OSIntCtxSw())。在 80x86 系统上，它通过执行一条软中断指令来实现任务切换。软中断向量指向 OSCtxSw()。在 μC/OS-II 中，如果任务调用了某个函数，而该函数的执行结果可能造成系统任务重新调度(例如唤醒了一个优先级更高的任务)，则在函数的末尾会调用 OSSched()。若 OSSched()判断需要做任务调度，则会找到该任务的任务控制块 OS_TCB 的地址，并将该地址复制给 OSTCBHighRdy，然后通过宏 OS_TASK_SW()执行软中断，进行任

务切换。

注意：在此过程中，变量 OSTCBCur 始终保存的是指向当前运行任务的控制块 OS_TCB 的指针。

程序清单 L15.15 是 OSCtxSw()的源代码，与第 14 章中的代码完全相同。这里再次讨论 OSCtxSw()，是因为增加了浮点寄存器的切换，增加了复杂性。

程序清单 L15.15　OSCtxSw()

```
_OSCtxSw    PROC  FAR                                          (1)
;
            PUSHA                                              (2)
            PUSH  ES
            PUSH  DS
;
            MOV   AX, SEG _OSTCBCur
            MOV   DS, AX
;
            LES   BX, DWORD PTR DS:_OSTCBCur                   (3)
            MOV   ES:[BX + 2], SS
            MOV   ES:[BX + 0], SP
;
            CALL  FAR PTR _OSTaskSwHook                        (4)
;
            MOV   AX, WORD PTR DS:_OSTCBHighRdy + 2            (5)
            MOV   DX, WORD PTR DS:_OSTCBHighRdy
            MOV   WORD PTR DS:_OSTCBCur + 2, AX
            MOV   WORD PTR DS:_OSTCBCur, DX
;
            MOV   AL, BYTE PTR DS:_OSPrioHighRdy               (6)
            MOV   BYTE PTR DS:_OSPrioCur, AL
;
            LES   BX, DWORD PTR DS:_OSTCBHighRdy               (7)
            MOV   SS, ES:[BX + 2]
            MOV   SP, ES:[BX]
;
            POP   DS                                           (8)
            POP   ES
            POPA
;
            IRET                                               (9)
;
_OSCtxSw    ENDP
```

图 F15.4 是在任务级做任务切换时的堆栈结构和任务挂起、恢复时浮点单元(FPU)存储区的数据结构示意图。

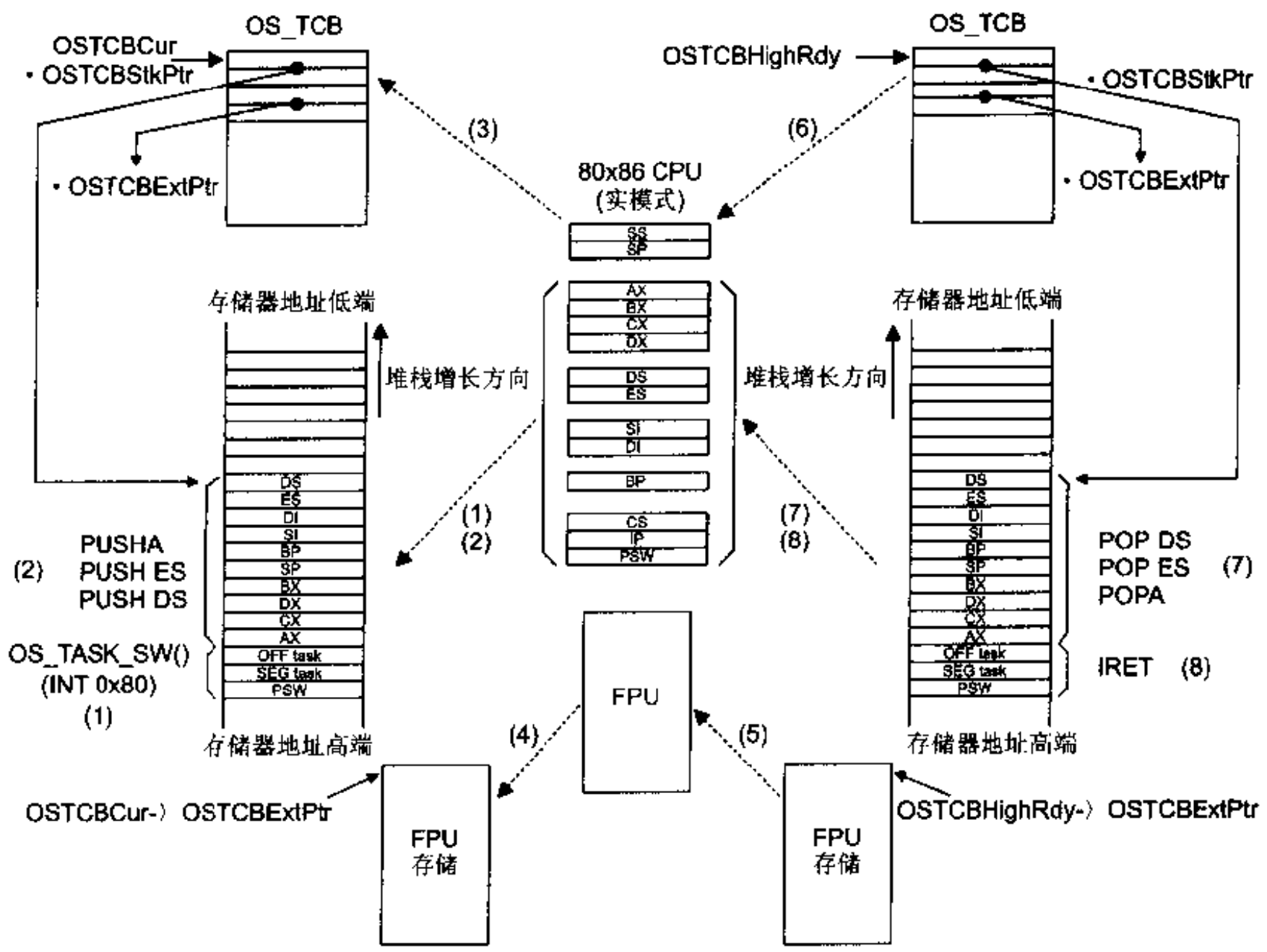


图 F15.4 在任务级做任务切换时 80x86 的堆栈结构和浮点单元(FPU)的存储

F15.4(1)

L15.15(1) 在 80x86 处理器上,软中断指令把 SW 寄存器的值推入到当前的任务堆栈中,接着入栈的是执行软中断指令的任务(也就是说,调用 OS_TASK_SW()的任务)的中断返回地址(段地址和偏移量地址)。

F15.4(2)

L15.15(2) 挂起任务的其他 CPU 寄存器保存在当前任务堆栈中。

F15.4(3)

L15.15(3) 指向新堆栈的指针保存在任务控制块 OS_TCB 中。这个指针包括堆栈段(SS 寄存器)和堆栈指针(SP 寄存器)。μC/OS-II 将堆栈指针放在任务控制块 OS_TCB 结构的开头,这样用汇编语言可以很容易地保存和恢复堆栈指针。

F15.4(4)

F15.4(5)

L15.15(4) 调用应用程序接口函数 OSTaskSwHook()。

注意：当调用 OSTaskSwHook()函数时，OSTCBCur 指向当前任务的任务控制块 OS_TCB，而 OSTCBHighRdy 则指向新任务的任务控制块 OS_TCB。这样可以在接口函数 OSTaskSwHook()中与上述 2 个任务控制块 OS_TCB 的任何一个打交道。

如果不想使用 OSTaskSwHook()，则可以注释掉这个调用，这样在环境切换时可节省几个时钟周期。换句话说，若不打算使用OSTaskSwHook()，也就没有必要为调用这个函数和从这个函数返回支付额外的 CPU 负荷。

一般情况下，在移植时都会调用这个函数。

L15.15(5) 从 OSTaskSwHook()函数返回后，OSTCBHighRdy 被复制给 OSTCBCur。因为新任务就是该运行的当前任务。

L15.15(6) OSPrioHighRdy 被复制给 OSPrioCur。这同样是因为新任务就是当前任务。

F15.4(6)

L15.15(7) 至此，OSCtxSw()把新任务栈中保存栈指针的内容传给处理器的相应寄存器。这是通过从新任务的 OS_TCB 获得寄存器 SS 和寄存器 SP 的值来完成的。

F15.4(7)

L15.15(8) 从新任务的堆栈中弹出相应值到 CPU 的其他寄存器。

F15.4(8)

L15.15(9) 执行中断返回指令 IRET，装入新任务的程序计数器值和状态字。执行了这一指令后，处理器使新任务开始运行。

注意：在执行 OSCtxSw()时，中断是关闭的；在执行 OSTaskSwHook()接口函数时，中断也是关闭的。

15.05.03 OSIntCtxSw()

由于中断可能会使更高优先级的任务进入就绪态，为了让更高优先级的任务能立即运行，需要进行任务切换。在中断服务子程序的最后，OSIntExit()函数会调用 OSIntCtxSw()函数做任务切换，故 OSIntCtxSw()又称为中断级的任务切换函数。由于在调用 OSIntCtxSw()之前已经发生了中断，OSIntCtxSw()已经将默认 CPU 寄存器保存在被中断了的任务堆栈中。

程序清单 L15.16 给出的代码与第 14 章给出的 OSIntCtxSw()的相同。浮点寄存器是由 OSTaskSwHook()函数处理的。

图 F15.5 描述了环境切换的过程。

与第 14 章相同，假设处理器接收到一个中断，并且此时中断是开着的。处理器完成当前的指令并开始响应中断，进入中断处理程序。

图 F15.5(1) 80x86 自动把处理器的 SW 寄存器推入堆栈，然后把中断的返回地址也推入堆栈。接下来 CPU 从中断向量处找到中断服务子程序的入口地址，执行中断服务子程序。在 μC/OS-II 中，要求用户的中断服务子程序在开

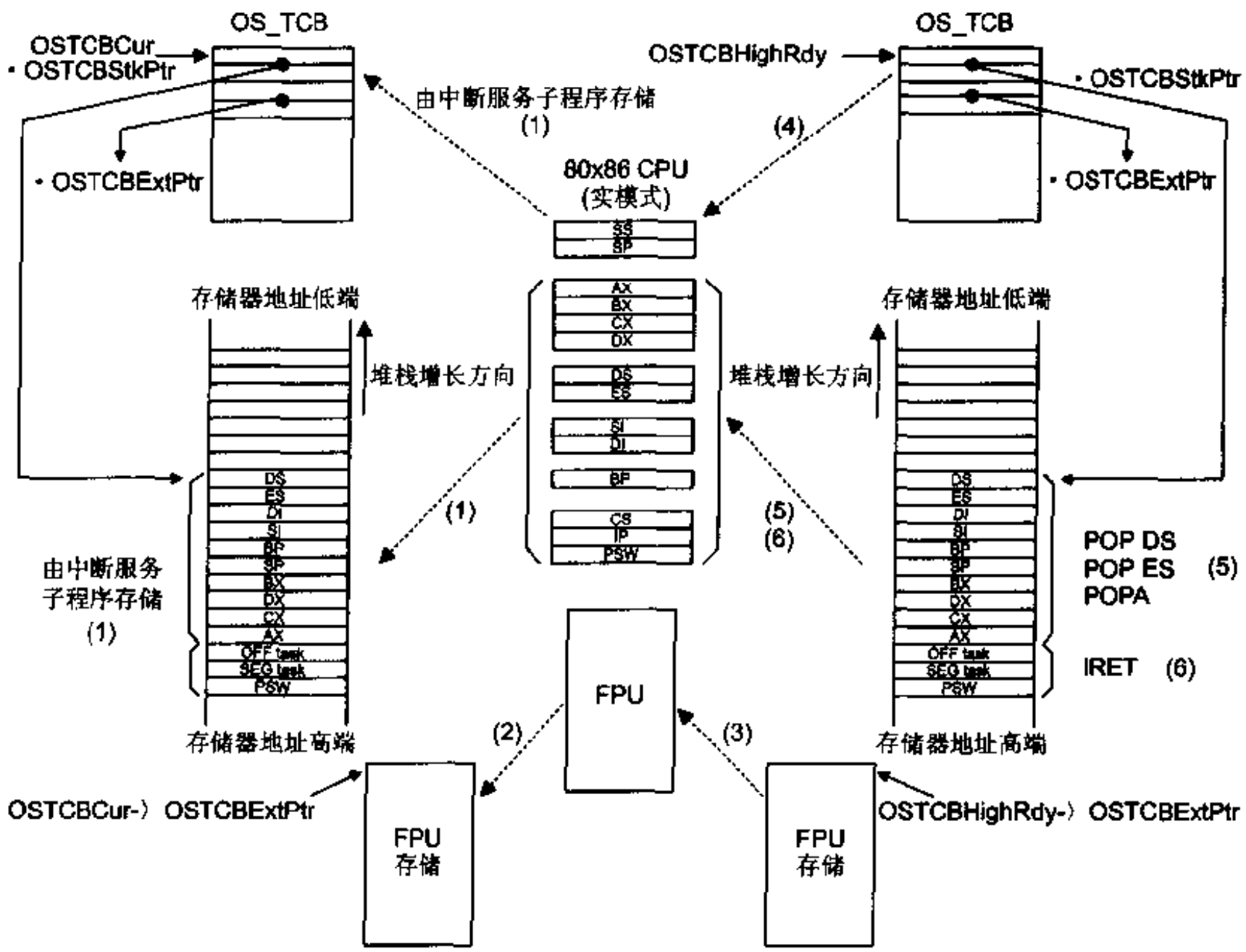


图 F15.5 中断级任务切换时的 80x86 堆栈结构和浮点单元(FPU)存储

头保存 CPU 其他寄存器的内容。保存了寄存器的内容后，μC/OS－II 还要求把堆栈指针保存在任务控制块 OS_TCB 中。

此后，必须调用 OSIntEnter()，或者直接给全局变量 OSIntNesting 加 1。至此可以认为任务已被挂起，如果需要，则可以切换到另一个任务。

现在，中断服务子程序可服务于产生中断的设备了。该服务有可能调用了 OSMboxPost()，OSQPostFront()，OSQPost()或 OSQPostOpt()等函数给其他任务发了消息，从而使一个优先级更高的任务进入了就绪态，须做任务切换。也可能唤醒了一个更高优先级的任务(例如调用了 OSTaskResume()，OSTimeTick()及 OSTimeDlyResume()等)，须做任务切换。

如果中断服务子程序使一个更高优先级的任务进入就绪态，那么 μC/OS－II 要求在中断服务程序完成对中断设备的服务以后，调用 OSIntExit()。OSIntExit()主要是告诉 μC/OS－II，所有嵌套的中断都已经完成，可以返回到任务级代码了。换句话说，当中断嵌套层数 OSIntNesting 减为 0 时，OSIntExit()就可以返回到任务级代码了。

当 OSIntExit()开始执行时，该函数得知被中断的任务不再是应该继续运行的任务，有更高优先级的任务需要运行。这种情况下，指针 OSTCBHighRdy 应该指向新任务的控制块 OS_TCB，OSIntExit()会调用 OSIntCtxSw()进行任务切换。

程序清单 L15.16 OSIntCtxSw()

```
_OSIntCtxSw PROC    FAR
;
            CALL    FAR PTR _OSTaskSwHook                          (1)
;
            MOV     AX, SEG _OSTCBCur
            MOV     DS, AX
;
            MOV     AX, WORD PTR DS:_OSTCBHighRdy + 2              (2)
            MOV     DX, WORD PTR DS:_OSTCBHighRdy
            MOV     WORD PTR DS:_OSTCBCur + 2, AX
            MOV     WORD PTR DS:_OSTCBCur, DX
;
            MOV     AL, BYTE PTR DS:_OSPrioHighRdy                 (3)
            MOV     BYTE PTR DS:_OSPrioCur, AL
;
            LES     BX, DWORD PTR DS:_OSTCBHighRdy                 (4)
            MOV     SS, ES:[BX + 2]
            MOV     SP, ES:[BX]
;
            POP     DS                                             (5)
            POP     ES
            POPA
;
            IRET                                                   (6)
;
_OSIntCtxSw ENDP
```

F15.5(2)

F15.5(3)

L15.16(1) OSIntCtxSw()做的第 1 件事情是，调用可由用户定义的任务切换接口函数 OSTaskSwHook()。

注意：当调用 OSTaskSwHook()时，OSTCBCur 指向当前任务的任务控制块 OS_TCB，而 OSTCBHighRdy 指向新任务的任务控制块 OS_TCB。这样用户可以在 OSTaskSwHook()函数中访问这 2 个任务的任务控制块 OS_TCB。

同样，如果不想使用 OSTaskSwHook()，则可以注释掉这条语句，这样在任务切换时，可以节省几个时钟周期。

L15.16(2) 当从 OSTaskSwHook()返回时，OSTCBHighRdy 被复制给 OSTCBCur，这是因为此时当前任务应该是新任务。

L15.16(3) OSPrioHighRdy 被复制给 OSPrioCur。这也是因为此时新任务应该是当

前任务。

F15.5(4)

L15.16(4) 至此，OSCtxSw()应该把新任务的运行环境装入到处理器的寄存器中，这要通过从新任务的任务控制块 OS_TCB 中获取寄存器 SS 和寄存器 SP 的值来实现。

F15.5(5)

L15.16(5) 从堆栈中弹出和恢复 CPU 的其余寄存器。

F15.5(6)

L15.16(6) 执行 IRET 指令，设置新任务的程序计数器和状态字。执行了这一指令后，处理器开始执行新任务。

注意：在执行 OSIntCtxSw()时，中断是关闭的；在执行 OSTaskSwHook()接口函数时，中断也是关闭的。

15.05.04 OSTickISR()

在 14.03.05 小节："OS_CPU.H，时钟节拍频率"中已经提到过，实时操作系统的时钟节拍频率应该为 10～100 Hz。在 PC 中，时钟节拍由硬件定时器产生，硬件定时器会中断 CPU，间隔是 54.93 ms (18.206 48 Hz)。笔者将时钟节拍频率设为 200 Hz。PC 时钟节拍的中断向量为 0x08，μC/OS-II 将此向量截取，使之指向 μC/OS-II 的时钟节拍中断服务子程序 OSTickISR()，而原先的中断向量保存在中断 129(0x81)中（参见 PC.C 文件中的函数 PC_DOSSaveReturn()）。为满足 DOS 的需要，原先的中断服务还是每隔 54.93 ms(实际上还要短些)调用 1 次。这里的 OSTickISR()与 14.05.04 小节中的 OSTickISR()相同，所以没有必要重复描述了；但这里还是给出了程序清单 L15.17。

程序清单 L15.17 OSTickISR()

```
_OSTickISR  PROC    FAR
;
            PUSHA
            PUSH    ES
            PUSH    DS
;
            MOV     AX, SEG(_OSIntNesting)
            MOV     DS, AX
            INC     BYTE PTR DS:_OSIntNesting
;
            CMP     BYTE PTR DS:_OSIntNesting, 1
            JNE     SHORT _OSTickISR1
            MOV     AX, SEG(_OSTCBCur)
            MOV     DS, AX
            LES     BX, DWORD PTR DS:_OSTCBCur
```

```
            MOV     ES:[BX+2], SS
            MOV     ES:[BX+0], SP
;
_OSTickISR1:
            MOV     AX, SEG(_OSTickDOSCtr)
            MOV     DS, AX
            DEC     BYTE PTR DS:_OSTickDOSCtr
            CMP     BYTE PTR DS:_OSTickDOSCtr, 0
            JNE     SHORT _OSTickISR2
;
            MOV     BYTE PTR DS:_OSTickDOSCtr, 11
            INT     081H
            JMP     SHORT _OSTickISR3

_OSTickISR2:
            MOV     AL, 20H
            MOV     DX, 20H
            OUT     DX, AL
;
_OSTickISR3:
            CALL    FAR PTR _OSTimeTick
;
            CALL    FAR PTR _OSIntExit
;
            POP     DS
            POP     ES
            POPA
;
            IRET
;
_OSTickISR  ENDP
```

15.05.05 OSFPSave()

OSFPSave()不是 μC/OS-II 移植所必需的。OSFPSave()主要是将浮点寄存器的内容保存到指针变量 pblk 指向的区域中。pblk 是传递给 OSFPSave()函数的参数。OSFPSave()是在 C 语言程序中调用的，但是用汇编语言编写的。这是因为此函数要用到一条浮点单元 FPU 指令，而这条指令不能用 C 语言编写。OSFPSave()被 OSFPInit()、OSTaskCreateHook()及 OSTaskSwHook()等 C 语言函数调用，调用形式如下：

```
OSFPSave((void *pblk);
```

这里 pblk 是存储浮点单元环境的地址，至少要有 108 B。程序清单 L15.18 是 OSFPSave()的代码。

程序清单 L15.18　OSFPSave()

```
_OSFPSave     PROC    FAR
;
              PUSH    BP                                          (1)
              MOV     BP,SP
              PUSH    ES
              PUSH    BX
;
              LES     BX, DWORD PTR [BP+6]                        (2)
;
              FSAVE   ES:[BX]                                     (3)
;
              POP     BX                                          (4)
              POP     ES
              POP     BP
;
              RET                                                 (5)
;
_OSFPSave     ENDP
```

L15.18(1)　OSFPSave()把整型寄存器的值保存到当前的任务堆栈中，这是因为 OSFPSave()要用到这些整型寄存器的值。

L15.18(2)　放在 ES:BX 中的指针作为自变量传给 OSFPSave()。

L15.18(3)　执行浮点单元指令 FSAVE。这个指令把浮点单元 FPU 的内容(108 B)放在地址 ES:BX 中。

L15.18(4)　恢复临时寄存器的值。

L15.18(5)　OSFPSave()返回调用函数。

15.05.06　OSFPRestore()

OSFPRestore()也不是移植所必需的。OSFPRestore()主要恢复浮点单元寄存器。要恢复的浮点单元寄存器的数据存放在传给 OSFPRestore()的指针参数指向的存储区里。OSFPRestore()是从 C 函数中调用的，但是用汇编语言编写的。这是因为此函数要用到一条浮点单元 FPU 指令，而这个指令不能用 C 语言编写。OSFPRestore()只被 OSTaskSwHook()接口函数调用，调用形式如下：

```
OSFPRestore(void *pblk);
```

这里 pblk 是存储区的首地址，存储区用来存放浮点单元的数据。存储区至少要有 108 B。程序清单 L15.19 是 OSFPRestore()的源代码。

程序清单 L15.19 OSFPRestore()

```
_OSFPRestore PROC   FAR
;
             PUSH   BP                                    (1)
             MOV    BP,SP
             PUSH   ES
             PUSH   BX
;
             LES    BX, DWORD PTR [BP+6]                  (2)
;
             FRSTOR ES:[BX]                               (3)
;
             POP    BX                                    (4)
             POP    ES
             POP    BP
;
             RET                                          (5)
;
_OSFPRestore ENDP
```

L15.19(1) OSFPRestore()把整型寄存器的值保存到当前的任务堆栈中，这是因为OSFPRestore()要用到这些整型寄存器的值。

L15.19(2) 放在 ES:BX 中的指针作为自变量传给 OSFPRestore()。

L15.19(3) 执行浮点单元指令 FRSTOR。这条指令从以 ES:BX 寄存器内容为首地址的区域中恢复浮点单元 FPU。

L15.19(4) 恢复临时寄存器的值。

L15.19(5) OSFPRestore()返回调用函数。

15.06 内存占用

与第14章相比，OS_CPU_A.ASM，OS_CPU_C.C 及 OS_CPU.H 3个文件有变化，其他的文件无改变。这3个文件只增加了164 B的代码空间(ROM)。必须用 OSTaskCreateExt()的代码(把 OS_CFG.H 文件中的 OS_TASK_CREATE_EXT 设置为1)，还必须使用内存管理功能函数(把 OS_CFG.H 文件中的 OS_MEM_EN 设置为1)的代码，因为这里的移植要用到这2个功能函数。

在数据空间方面，对于每个涉及浮点操作的任务，移植都需要128 B的缓存区(实际上只

需要 108 B)。

注意：详细数据表在本书所附的光盘中。

具体位置如下：

\SOFTWARE\uCOS-II\Ix86L\BC45\DOC\80x86L-ROM-RAM. XLS

用 Office 2000(或者更高版本)的 Excel 软件可以打开。可以改变配置的值(红色)，看 80x86 的 ROM 和 RAM 受到了什么影响。??? _EN 的值必须取 0 或 1。

与第 14 章一样，是用 Borland 编译器编译的，选择速度优先。表中给出的字节数并不是精确的数字，只是表明 μC/OS－II 的功能函数大概占用了多少内存。

在 OS_CFG. H 文件中改变 OS_ARG_CHK_EN 的值，代码大小会有变化，表中已经列出。不需要改变 OS_ARG_CHK_EN 的值观察这一变化。

Data 一栏不易懂。

注意：空闲任务(idle)和统计任务(statistics)的堆栈都置为 1 024 B(1 KB)。

根据用户的要求，它们堆栈的容量可以增减。如果配置成允许的最多任务数(62 个用户任务)，则 μC/OS－II 内部的数据结构最少需要占用 3 500 B 的 RAM。增加了一个条目用来记录可以做浮点操作的任务。请记住，这样的任务需要 128 B 的缓冲区。至少需要一个缓冲区，因为改变了统计任务，统计任务要使用浮点操作。

如果使用的是 80x86 CPU，那么一般情况下不太会受存储器容量的限制。在多数情况下，μC/OS－II 也不会是占用内存最多的模块。

第 16 章

μC/OS－II 参考手册

本章是 μC/OS－II 的用户参考手册。每一个可以调用的内核函数都按字母顺序加以说明，包括：

- 函数的功能描述；
- 函数原型；
- 函数名称及源代码；
- 如何定义允许或禁止功能函数的常量；
- 函数参数；
- 函数返回值；
- 特殊说明和注意点；
- 用一两个例子说明函数如何使用。

OS_ENTER_CRITICAL( )
OS_EXIT_CRITICAL( )

所在章	所属文件	调用者	开关量
3	OS_CPU.H	任务或中断	无

OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()为定义的宏，用来关/开处理器的中断。

参 数

无。

返回值

无。

注意/警告

1. OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()必须成对使用。
2. 如果 OS_CRITICAL_METHOD 设置为 3，则用户程序中应分配类型为 OS_CPU_SR 的局部变量，名为 CPU_SR，如下所示：

```
#if OS_CRITICAL_METHOD == 3        /* 为 CPU 状态寄存器分配存储空间 */
    OS_CPU_SR  cpu_sr;
#endif
```

范 例

```
void TaskX (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
      OS_CPU_SR   cpu_sr;
#endif

      for (;;) {
         .
         .
         OS_ENTER_CRITICAL();          /* 关中断          */
         .                             /* 处理临界段代码 */
         OS_EXIT_CRITICAL();           /* 开中断          */
         .
         .
      }
}
```

OSFlagAccept( )

```
OS_FLAGS OSFlagAccept(OS_FLAG_GRP *pgrp,
                      OS_FLAGS flags,
                      INT8U wait_type,
                      INT8U *err);
```

所在章	所属文件	调用者	开关量
9	OS_FLAG.C	任务或中断	OS_FLAG_EN && OS_FLAG_ACCEPT_EN

OSFlagAccept()函数检查事件标志组中的事件标志位是置位还是清 0。应用程序可以检查任意一位是置位还是清 0,也可以检查所有位都置位还是清 0。这个函数与 OSFlagPend()不同之处在于,如果需要的事件标志没有产生,那么调用该函数的任务并不挂起。

参 数

pgrp 指向事件标志组的指针。建立事件标志组时(见 OSFlagCreate())得到该指针。

flags 指定需要检查的事件标志位。置为 1,则检查对应位;设为 0,则忽略对应位。

wait_type 定义等待事件标志位的方式。可以定义为以下 4 种方式之一:

OS_FLAG_WAIT_CLR_ALL	所有指定事件标志位清 0。
OS_FLAG_WAIT_CLR_ANY	任意指定事件标志位清 0。
OS_FLAG_WAIT_SET_ALL	所有指定事件标志位置位。
OS_FLAG_WAIT_SET_ANY	任意指定事件标志位置位。

如果需要在得到期望的事件标志后,恢复该事件标志,则可以在调用函数时,将该参数加上一个常量 OS_FLAG_CONSUME 。例如,如果等待事件标志组中任意指定事件标志位置位,并且在任意事件标志位置位后清除该位,则可把参数 wait_type 设置为:

OS_FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME

err 指向出错代码的指针。出错代码为以下值之一:

OS_NO_ERR	成功调用。
OS_ERR_EVENT_TYPE	pgrp 指针不是指向事件标志组的指针。
OS_FLAG_ERR_WAIT_TYPE	wait_type 参数不是指定的 4 种方式之一。
OS_FLAG_INVALID_PGRP	pgrp 是一个 NULL 指针。
OS_FLAG_ERR_NOT_RDY	指定的事件标志没有发生。

返回值

返回事件标志组的事件标志状态。

注意/警告

1. 必须先建立事件标志组，然后使用。
2. 如果指定的事件标志没有发生，则调用任务并不挂起。

范 例

```
#define  ENGINE_OIL_PRES_OK    0x01
#define  ENGINE_OIL_TEMP_OK    0x02
#define  ENGINE_START          0x04


OS_FLAG_GRP *EngineStatus;


void Task (void *pdata)
{
    INT8U      err;
    OS_FLAGS   value;


    pdata = pdata;
    for (;;) {
        value = OSFlagAccept(EngineStatus,
                             ENGINE_OIL_PRES_OK + ENGINE_OIL_TEMP_OK,
                             OS_FLAG_WAIT_SET_ALL,
                             &err);
        switch (err) {
            case OS_NO_ERR:
                /* 得到等待的事件标志 */
                break;

            case OS_FLAG_ERR_NOT_RDY:
                /* 期待的事件标志没有发生 */
                break;
        }
        .
        .
    }
}
```

OSFlagCreate()

OS_FLAG_GRP *OSFlagCreate(OS_FLAGS flags, INT8U *err);

所在章	所属文件	调用者	开关量
9	OS_FLAG.C	任务或初始化代码	OS_FLAG_EN

OSFlagCreate()用于创建并初始化一个事件标志组。

参 数

flags 事件标志组的事件标志初值。

err 指向出错代码的指针。出错代码为以下值之一：

OS_NO_ERR	成功创建事件标志组。
OS_ERR_CREATE_ISR	从中断中调用 OSFlagCreate()函数。
OS_FLAG_GRP_DEPLETED	系统没有剩余的空闲事件标志组，需要更改 OS_CFG.H 中的系统事件标志组数目的配置。

返回值

如果成功创建事件标志组，则返回该事件标志组的指针。如果系统没有剩余的空闲事件标志组，则返回空指针。

注意/警告

在使用任何事件标志组功能前，必须使用该函数创建事件标志组。

范 例

```
OS_FLAG_GRP *EngineStatus;

void main (void)
{
    INT8U  err;

    .
    OSInit();          /* 初始化 μC/OS-II */
    .
    .
                       /* 创建事件标志组 */
    EngineStatus = OSFlagCreate(0x00, &err);
    .
    .
    OSStart();         /* 开始多任务内核 */
}
```

OSFlagDel()

OS_FLAG_GRP *OSFlagDel(OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err);

所在章	所属文件	调用者	开关量
9	OS_FLAG.C	任 务	OS_FLAG_EN 和 OS_FLAG_DEL_EN

OSFlagDel() 函数用于删除一个事件标志组。因为多任务可能会试图继续使用已经删除了的事件标志组，故调用本函数有风险。使用本函数需特别小心。一般地说，在删除事件标志组之前，应该首先删除与本事件标志组有关的任务。

参 数

pgrp 指向事件标志组的指针。建立事件标志组时(见 OSFlagCreate())得到该指针。

opt 指明是仅在没有任务等待事件标志组时删除该事件标志组(OS_DEL_NO_PEND)，还是不管是否有任务等待事件标志组都删除该事件标志组(OS_DEL_ALWAYS)。如果是后者，所有等待该事件标志组的任务都被置为就绪。

err 指向出错代码的指针。出错代码为以下值之一：

OS_NO_ERR	成功删除事件标志组。
OS_ERR_DEL_ISR	从中断中调用 OSFlagDel()函数。
OS_FLAG_INVALID_PGRP	pgrp 是空指针。
OS_ERR_EVENT_TYPE	pgrp 不是指向事件标志组的指针。
OS_ERR_INVALID_OPT	opt 参数不是指定的值。
OS_ERR_TASK_WAITING	如果 opt 参数为 OS_DEL_NO_PEND，那么此时有任务等待事件标志组。

返回值

如果事件标志组被删除，则返回空指针；如果没有删除，则仍然返回指向该事件标志组的指针。在后一种情况下，需要检查出错代码，找出事件标志组没有成功删除的原因。

注意/警告

1. 需要小心，可能有其他任务正在等待该事件标志组的事件标志。
2. 该函数有可能长时间关闭中断，其时间长短决定于等待该事件标志组的任务个数。

范 例

```
OS_FLAG_GRP *EngineStatusFlags;


void Task (void  *pdata)
{
    INT8U        err;
    OS_FLAG_GRP *pgrp;


    pdata = pdata;
    while (1) {
        .
        .
        pgrp = OSFlagDel(EngineStatusFlags, OS_DEL_ALWAYS, &err);
        if (pgrp == (OS_FLAG_GRP *)0) {
            /* 删除事件标志组 */
        }
        .
        .
    }
}
```

OSFlagPend()

```
OS_FLAGS OSFlagPend(OS_FLAG_GRP *pgrp,
                    OS_FLAGS flags,
                    INT8U wait_type,
                    INT16U timeout,
                    INT8U *err);
```

所在章	所属文件	调用者	开关量
9	OS_FLAG.C	任 务	OS_FLAG_EN

OSFlagPend()用于任务等待事件标志组中的事件标志，可以是多个事件标志的不同组合方式。可以等待任意指定事件标志位置位或清0，也可以是全部指定事件标志位置位或清0。如果任务等待的事件标志位条件尚不满足，则任务会被挂起，直到指定的事件标志组合发生或指定的等待时间超时。

参 数

pgrp　指向事件标志组的指针。建立事件标志组时(见 OSFlagCreate())得到该指针。

flags　指定需要检查的事件标志位。置为1，则检查对应位；置为0，则忽略对应位。

wait_type　定义等待事件标志位的方式。可以定义为以下4种方式之一：

OS_FLAG_WAIT_CLR_ALL	所有指定事件标志位清0。
OS_FLAG_WAIT_CLR_ANY	任意指定事件标志位清0。
OS_FLAG_WAIT_SET_ALL	所有指定事件标志位置位。
OS_FLAG_WAIT_SET_ANY	任意指定事件标志位置位。

如果需要在得到期望的事件标志后恢复该事件标志，则可以在调用该函数时，将该参数加上一个常量 OS_FLAG_CONSUME 。例如，如果等待事件标志组中任意指定事件标志位置位，并且在任意事件标志位置位后清除该位，则可以把参数 wait_type 设置为：

```
OS_FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME
```

err　指向出错代码的指针。出错代码为以下值之一：

OS_NO_ERR	成功调用。
OS_ERR_PEND_ISR	从中断中调用该函数，这是规则不允许的。
OS_FLAG_INVALID_PGRP	pgrp 为空指针。
OS_ERR_EVENT_TYPE	pgrp 不是指向事件标志组的指针。
OS_TIMEOUT	等待事件标志组的事件标志超时。
OS_FLAG_ERR_WAIT_TYPE	wait_type 不是指定的参数值之一。

返回值

如果使用了 OS_FLAG_CONSUME 选项，则返回清理后的事件标志组事件标志状态；否则返回 OSFlagPend()函数运行结束后的事件标志组事件标志状态；如果发生了超时，则返回 0。

注意/警告

必须首先创建事件标志组，再使用。

范 例

```
#define  ENGINE_OIL_PRES_OK   0x01
#define  ENGINE_OIL_TEMP_OK   0x02
#define  ENGINE_START         0x04


OS_FLAG_GRP *EngineStatus;


void Task (void *pdata)
{
    INT8U     err;
    OS_FLAGS  value;


    pdata = pdata;
    for (;;) {
        value = OSFlagPend(EngineStatus,
                           ENGINE_OIL_PRES_OK   + ENGINE_OIL_TEMP_OK,
                           OS_FLAG_WAIT_SET_ALL + OS_FLAG_CONSUME,
                           10,
                           &err);
        switch (err) {
            case OS_NO_ERR:
              /* 成功返回，得到期望的事件标志位 */
                break;

            case OS_TIMEOUT:
              /* 在指定的 10 个时钟滴答内没有得到期望的事件标志位 */
                break;
        }
        .
        .
    }
}
```

OSFlagPost()

```
OS_FLAGS OSFlagPost(OS_FLAG_GRP *pgrp,
                    OS_FLAGS flags,
                    INT8U opt,
                    INT8U *err);
```

所在章	所属文件	调用者	开关量
9	OS_FLAG.C	任务或中断	OS_FLAG_EN

OSFlagPost()用于给出设定的事件标志位。指定的事件标志位可以设定为置位或清除。若OSFlagPost()设置的事件标志位正好满足某个等待该事件标志组的任务，则OSFlagPost()将该任务设为就绪。

参 数

pgrp 指向事件标志组的指针。建立事件标志组时(见OSFlagCreate())得到该指针。

flags 指定需要检查的事件标志位。如果opt参数为OS_FLAG_SET，那么事件标志组中对应的事件标志位被置位。例如，如果要置位事件标志组的事件标志0，4和5，则需要把flags参数设置为0x31(bit 0是最低位)。若opt参数为OS_FLAG_CLR，那么事件标志组中对应的事件标志位被清0。

opt 表明是置位指定事件标志位(OS_FLAG_SET)，还是清0指定事件标志位(OS_FLAG_CLR)。

err 指向出错代码的指针。出错代码为以下值之一：

OS_NO_ERR	成功调用。
OS_FLAG_INVALID_PGRP	pgrp指针为空指针。
OS_ERR_EVENT_TYPE	pgrp指针没有指向事件标志组结构。
OS_FLAG_INVALID_OPT	opt参数不是指定的参数值之一。

返回值

事件标志组的新的事件标志状态。

注意/警告

1. 必须先创建事件标志组，然后使用。
2. 这个函数的运行时间决定于等待事件标志组的任务的数目。
3. 关闭中断的时间也决定于等待事件标志组的任务的数目。

范 例

```
#define  ENGINE_OIL_PRES_OK    0x01
#define  ENGINE_OIL_TEMP_OK    0x02
#define  ENGINE_START          0x04

OS_FLAG_GRP   *EngineStatusFlags;

void  TaskX (void *pdata)
{
    INT8U  err;

    pdata = pdata;
    for (;;) {
        .
        .
        err = OSFlagPost(EngineStatusFlags, ENGINE_START. OS_FLAG_SET, &err);
        .
        .
    }
}
```

OSFlagQuery()

OS_FLAGS OSFlagQuery(OS_FLAG_GRP *pgrp, INT8U *err);

所在章	所属文件	调用者	开关量
9	OS_FLAG.C	任务或中断	OS_FLAG_EN 和 OS_FLAG_QUERY_EN

OSFlagQuery()用于查询事件标志组的当前事件标志状态。在现在的版本中，该函数还不能返回等待该事件标志组的任务列表。

参 数

pgrp 指向事件标志组的指针。建立事件标志组时(见 OSFlagCreate())得到该指针。

err 指向出错代码的指针。出错代码为以下值之一：

OS_NO_ERR	成功调用。
OS_FLAG_INVALID_PGRP	pgrp 指针为空指针。
OS_ERR_EVENT_TYPE	pgrp 指针没有指向事件标志组结构。

返回值

当前事件标志组的事件标志状态。

注意/警告

1. 必须首先创建事件标志组，然后使用和查询。
2. 可以从中断中调用该函数。

范 例

```
OS_FLAG_GRP *EngineStatusFlags;

void Task (void *pdata)
{
    OS_FLAGS flags;
    INT8U    err;

    pdata = pdata;
    for (;;) {
        .
        .
        flags = OSFlagQuery(EngineStatusFlags, &err);
        .
        .
    }
}
```

OSInit()

void OSInit(void);

所在章	所属文件	调用者	开关量
3	OS_CORE.C	启动代码	无

OSInit()初始化 μC/OS-II。对这个函数的调用必须在调用 OSStart()函数之前、OSStart()函数真正开始运行多任务时。

参 数

无。

返回值

无。

注意/警告

必须先于 OSStart()函数的调用。

范 例

```
void main (void)
{
    .
    .
    OSInit();       /* 初始化 μC/OS-II */
    .
    .
    OSStart();      /* 启动多任务内核 */
}
```

OSIntEnter()

void OSIntEnter(void);

所在章	所属文件	调用者	开关量
3	OS_CORE.C	只能在中断服务子程序中	无

OSIntEnter()通知 μC/OS-II,一个中断处理函数正在执行,这有助于 μC/OS-II 掌握中断嵌套的情况。OSIntEnter()函数通常和 OSIntExit()函数联合使用。

参 数

无。

返回值

无。

注意/警告

1. 在任务级不能调用该函数。
2. 如果系统使用的处理器能够执行自动的独立执行读取—修改—写入的操作,那么就可以直接递增中断嵌套层数(OSIntNesting),这样可以避免调用函数所带来的额外开销。在中断服务子程序中给 OSIntNesting 加 1 是不会有问题的,因为给 OSIntNesting 加 1 时,中断是关闭的。
3. 中断嵌套深度可达 255。

范例 1

Intel 80x86 的实模式,在大模式下编译。

使用 OSIntEnter(),为的是与 μC/OS 兼容。

```
ISRx PROC   FAR
     PUSHA                          ;保存中断现场
     PUSH   ES
     PUSH   DS
;
     CALL   FAR PTR _OSIntEnter     ;通知 μC/OS-II,进入中断服务子程序了
     .
     .
     POP    DS                      ;恢复处理器寄存器
```

```
        POP     ES
        POPA
        IRET                                    ; 中断返回
ISRx    ENDP
```

范例 2

Intel 80x86 的实模式，在大模式下编译。

```
ISRx    PROC    FAR
        PUSHA                                   ; 保存中断现场
        PUSH    ES
        PUSH    DS
;
        MOV     AX, SEG(_OSIntNesting)          ; 恢复 DS
        MOV     DS, AX
;
        INC     BYTE PTR _OSIntNesting          ; 通知内核进入中断
        .
        .
        .
        POP     DS                              ; 恢复 CPU 寄存器
        POP     ES
        POPA
        IRET                                    ; 中断返回
ISRx    ENDP
```

OSIntExit()

void OSIntExit(void);

所在章	所属文件	调用者	开关量
3	OS_CORE.C	只能在中断服务子程序中	无

OSIntExit()通知 μC/OS-II,一个中断服务已执行完毕,这有助于 μC/OS-II 掌握中断嵌套的情况。通常 OSIntExit()和 OSIntEnter()联合使用。当最后一层嵌套的中断执行完毕时,如果有更高优先级的任务准备就绪,则 μC/OS-II 会调用任务调度函数。在这种情况下,中断返回到更高优先级的任务,而不是被中断了的任务。

参 数

无。

返回值

无。

注意/警告

在任务级不能调用该函数,并且即使没有调用 OSIntEnter()函数,而是使用直接递增 OSIntNesting的方法,也必须调用 OSIntExit()函数。

范 例

Intel 80x86 的实模式,在大模式下编译。

```
ISRx    PROC    FAR
        PUSHA                           ; 保存 CPU 寄存器
        PUSH    ES
        PUSH    DS
        .
        .
        CALL    FAR PTR _OSIntExit      ; 通知内核中断执行完毕
        POP     DS                      ; 恢复 CPU 寄存器
        POP     ES
        POPA
        IRET                            ; 返回到中断的任务
ISRx    ENDP
```

OSMboxAccept()

void *OSMboxAccept(OS_EVENT *pevent);

所在章	所属文件	调用者	开关量
10	OS_MBOX.C	任务或中断	OS_MBOX_EN 和 OS_MBOX_ACCEPT_EN

OSMboxAccept()函数查看指定的消息邮箱是否有需要的消息。不同于 OSMboxPend()函数，如果没有需要的消息，那么 OSMboxAccept()函数并不挂起任务。如果消息已经到达，则该消息被传递到用户任务并从消息邮箱中清除。通常中断调用该函数，因为中断不允许挂起等待消息。

参 数

pevent 指向需要查看的消息邮箱的指针。当建立消息邮箱时，该指针返回到用户程序(见 OSMboxCreate()函数)。

返回值

如果消息已经到达，则返回指向该消息的指针；如果消息邮箱没有消息，则返回空指针。

注意/警告

必须先建立消息邮箱，然后使用。

范 例

```
OS_EVENT *CommMbox;

void Task (void *pdata)
{
    void *msg;

    pdata = pdata;
    for (;;) {
        msg = OSMboxAccept(CommMbox);      /* 检查消息邮箱是否有消息 */
        if (msg != (void *)0) {
                                           /* 收到了消息,处理消息 */
            .
            .
        } else {
                                           /* 没有消息，就做… */
```

```
        /* …其他的事 */
    }
    }
  }
```

OSMboxCreate( )

OS_EVENT *OSMboxCreate(void *msg);

所在章	所属文件	调用者	开关量
10	OS_MBOX.C	任务或启动代码	OS_MBOX_EN

OSMboxCreate()建立并初始化一个消息邮箱。消息邮箱允许任务或中断向其他一个或几个任务发送消息。

参 数

msg 用来初始化建立的消息邮箱。如果该指针不为空，则建立的消息邮箱将含有消息。

返回值

指向分配给所建立的消息邮箱的事件控制块的指针。如果没有可用的事件控制块，则返回空指针。

注意/警告

必须先建立消息邮箱，然后再使用。

范 例

```
OS_EVENT *CommMbox;


void main (void)
{
    .
    .
    OSInit();                                   /* 初始化 μC/OS-II */
    .
    .
    CommMbox = OSMboxCreate((void *)0);         /* 建立消息邮箱 */
    OSStart();                                  /* 启动多任务内核 */
}
```

OSMboxDel()

OS_EVENT *OSMboxDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

所在章	所属文件	调用者	开关量
10	OS_MBOX.C	任 务	OS_MBOX_EN 和 OS_MBOX_DEL_EN

OSMboxDel()用于删除消息邮箱。因为多任务可能会试图继续使用已经删除了的邮箱，故调用本函数有风险。使用本函数须特别小心。一般地说，删除邮箱之前，应该首先删除与本邮箱有关的任务

参 数

pevent 指向邮箱的指针。该指针是在邮箱建立时，返回给用户应用程序的指针（参见OSMboxCreate()）。

opt 该选项定义邮箱的删除条件，可以选择只能在已经没有任何任务在等待该邮箱的消息时，才能删除邮箱（OS_DEL_NO_PEND）；或者，不管有没有任务在等待邮箱的消息，立即删除邮箱（OS_DEL_ALWAYS），在这种情况下，所有等待邮箱消息的任务都立即进入就绪态。

err 指向出错代码的指针。返回的出错代码可以是以下几种之一：

OS_NO_ERR	调用成功，邮箱已被删除。
OS_ERR_DEL_ISR	试图在中断服务子程序中删除邮箱。
OS_ERR_INVALID_OPT	无效的 opt 参数，用户没有将 opt 定义为上述 2 种情况之一。
OS_ERR_EVENT_TYPE	pevent 不是指向邮箱的指针。
OS_ERR_PEVENT_NULL	已经没有 OS_EVENT 数据结构可以使用。

返回值

返回空指针 NULL，表示邮箱已被删除；返回 pevent，表示邮箱没有删除，在这种情况下，应该进一步查看出错代码，找到出错原因。

注意/警告

1. 使用这个函数调用时，须特别小心，因为其他任务可能还要用这个邮箱。
2. 当挂起的任务进入就绪态时，中断是关闭的，这就意味着中断延迟时间与在等待邮箱的消息的任务数有关。
3. 调用 OSMboxAccept() 函数也不可能知道邮箱是否已经被删除了。

* 原文如此。应为"传递的 pevent 为一空指针。"详见相应源程序。——译者注

范 例

```
OS_EVENT *DispMbox;

void Task (void *pdata)
{
    INT8U  err;

    pdata = pdata;
    while (1) {
        .
        .
        DispMbox = OSMboxDel(DispMbox, OS_DEL_ALWAYS, &err);
        if (DispMbox == (OS_EVENT *)0) {
            /* 邮箱已被删除 */
        }
        .
        .
    }
}
```

OSMboxPend()

void *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

所在章	所属文件	调用者	开关量
10	OS_MBOX.C	任 务	OS_MBOX_EN

OSMboxPend()用于任务等待消息。消息通过中断或另外的任务发送给需要的任务。消息是一个以指针定义的变量，在不同的程序中消息的使用也可能不同。如果在调用函数OSMboxPend()时，消息邮箱已经存在需要的消息，那么该消息被返回给OSMboxPend()的调用者，消息邮箱中清除该消息；如果调用OSMboxPend()函数时，消息邮箱中没有需要的消息，那么OSMboxPend()函数挂起当前任务，直到得到需要的消息或超出定义等待超时的时间。如果同时有多个任务等待同一则消息，那么μC/OS-II默认最高优先级的任务取得消息并且任务恢复执行。一个由OSTaskSuspend()函数挂起的任务也可以接收消息，但这个任务将一直保持挂起状态，直到通过调用OSTaskResume()函数恢复任务的运行。

参 数

pevent 指向即将接收消息的消息邮箱的指针。在建立该消息邮箱时可以得到该指针的值(见OSMboxCreate()函数)。

timeout 允许一个任务在经过了指定数目的时钟节拍后还没有得到需要的消息时恢复运行。如果该值为0，则表示任务将持续等待消息。最长的等待时间为65 535个时钟节拍。这个时间长度并不是非常严格的，可能存在一个时钟节拍的误差，因为只有在一个时钟节拍结束后，才会减少定义的等待超时时钟节拍。

err 指向包含错误码的变量的指针。OSMboxPend()函数返回的错误码可能为下述几种之一：

OS_NO_ERR	消息被正确地接收。
OS_TIMEOUT	消息没有在指定的等待时间内送到。
OS_ERR_EVENT_TYPE	pevent不是指向消息邮箱的指针。
OS_ERR_PEND_ISR	从中断调用该函数。虽然规定了不允许从中断调用该函数，但μC/OS-II仍然包含了检测这种情况的功能。
OS_ERR_PEVENT_NULL	pevent是空指针。

返回值

OSMboxPend()函数返回接收的消息并将*err置为OS_NO_ERR。如果没有在指定数目的时钟节拍内接收到需要的消息，那么OSMboxPend()函数返回空指针，并且将*err设置为OS_TIMEOUT。

注意/警告

1. 必须先建立消息邮箱，然后使用。
2. 不允许从中断调用该函数。

范　例

```
OS_EVENT *CommMbox;


void CommTask (void *pdata)
{
    INT8U   err;
    void    *msg;


    pdata = pdata;
    for (;;) {
        .
        .
        msg = OSMboxPend(CommMbox, 10, &err);
        if (err == OS_NO_ERR) {
            .
            .   /* 接收消息的代码 */
            .
        } else {
            .
            .   /* 在指定时间内没有接收到消息代码 */
            .
        }
        .
        .
    }
}
```

OSMboxPost()

INT8U OSMboxPost(OS_EVENT *pevent, void *msg);

所在章	所属文件	调用者	开关量
10	OS_MBOX.C	任务或中断	OS_MBOX_EN 和 OS_MBOX_POST_EN

OSMboxPost()函数通过消息邮箱向任务发送消息。消息是一个以指针表示的某种数据类型的变量,在不同的程序中消息的使用也可能不同。如果消息邮箱中已经存在消息,则返回错误码,说明消息邮箱已满。OSMboxPost()函数立即返回调用者,消息也没有能够发到消息邮箱。如果有任何任务在等待消息邮箱的消息,那么最高优先级的任务将得到这则消息。如果等待消息的任务优先级比发送消息的任务优先级高,那么高优先级的任务将得到消息而恢复执行。也就是说,发生了一次任务切换。

参 数

pevent 指向即将接收消息的消息邮箱的指针。在建立该消息邮箱时可以得到该指针的值(见 OSMboxCreate()函数)。

msg 即将实际发送给任务的消息。消息是一个以指针表示的某种数据类型的变量,在不同的程序中消息的使用也可能不同。不允许传递一个空指针,因为这意味着消息邮箱为空。

返回值

OSMboxPost()函数的返回值为下述之一:

OS_NO_ERR	消息成功地放到消息邮箱中。
OS_MBOX_FULL	消息邮箱已经包含了其他消息,已满。
OS_ERR_EVENT_TYPE	pevent 不是指向消息邮箱的指针。
OS_ERR_PEVENT_NULL	pevent 是空指针。
OS_ERR_POST_NULL_PTR	用户试图发出空指针。根据规则,在这里不支持空指针。

注意/警告

1. 必须先建立消息邮箱,然后再使用。
2. 不允许传递一个空指针,因为这意味着消息邮箱为空。

范 例

```
OS_EVENT *CommMbox;
INT8U    CommRxBuf[100];


void CommTaskRx (void *pdata)
{
    INT8U  err;

    pdata = pdata;
    for (;;) {
        .
        .
        err = OSMboxPost(CommMbox, (void *)&CommRxBuf[0]);
        .
        .
    }
}
```

OSMboxPostOpt()

INT8U OSMboxPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);

所在章	所属文件	调用者	开关量
10	OS_MBOX.C	任务或中断	OS_MBOX_EN 和 OS_MBOX_POST_OPT_EN

OSMboxPostOpt() 的工作方式与 OSMboxPost() 相同，只是允许用户程序发消息给多个任务。换句话说，OSMboxPostOpt() 允许将消息广播给所有的等待邮箱消息的任务。OSMboxPostOpt() 实际上可以取代 OSMboxPost()，因为它可以仿真 OSMboxPost()。

OSMboxPostOpt() 通过邮箱给任务发消息。消息是一个以指针表示的某种数据类型的变量，在不同的程序中消息的使用也可能不同。如果消息邮箱中已经存在消息，则返回错误码，说明消息邮箱已满。OSMboxPostOpt() 函数立即返回调用者，消息也没有能够发到消息邮箱。如果有任何任务在等待消息邮箱的消息，那么 OSMboxPostOpt() 允许用户选择以下2种情况之一：或者让最高优先级的任务得到这则消息（opt 置为 OS_POST_OPT_NONE），或者让所有等待邮箱消息的任务都得到消息（opt 置为 OS_POST_OPT_BROADCAST）。无论在哪种情况下，如果得到消息的任务优先级比发送消息的任务优先级高，那么得到消息的最高优先级的任务将恢复执行，发消息的任务将被挂起。也就是说，发生了一次任务切换。

参 数

pevent　指向即将接收消息的消息邮箱的指针。在建立该消息邮箱时可以得到该指针的值（见 OSMboxCreate() 函数）。

msg　即将实际发送给任务的消息。消息是一个以指针表示的某种数据类型的变量，在不同的程序中消息的使用也可能不同。不允许传递一个空指针，因为这意味着消息邮箱为空。

opt　定义消息只发给等待邮箱消息的任务中优先级最高的任务（将 opt 置为 OS_POST_OPT_NONE），或者让所有等待邮箱消息的任务都得到消息（将 opt 置为 OS_POST_OPT_BROADCAST）。

返回值

err　指向包含错误码的变量的指针。返回的错误码可能为下述几种之一：

OS_NO_ERR	调用成功，消息已经发出。
OS_MBOX_FULL	邮箱中已经有消息。每次只能发送一则消息到消息邮箱，在允许发送下一则消息之前，前面一则消息必须被取走。
OS_ERR_EVENT_TYPE	pevent 不是指向邮箱的指针。
OS_ERR_PEVENT_NULL	pevent 是空指针。

OS_ERR_POST_NULL_PTR 试图发送空指针。根据规则，空指针无效。

注意/警告

1. 必须先建立消息邮箱，然后再使用。
2. 不允许向邮箱发送空指针，因为这意味着消息邮箱为空。
3. 若想使用本函数，又希望压缩代码长度，则可以将 OSMboxPost()函数的开关量关掉。因为 OSMboxPostOpt() 可以仿真 OSMboxPost()。
4. OSMboxPostOpt() 在广播方式下，即已将 opt 置为 OS_POST_OPT_BROADCAST，函数的执行时间取决与等待邮箱消息的任务的数目。

范 例

```
OS_EVENT *CommMbox;
INT8U     CommRxBuf[100];


void CommRxTask (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    for (;;) {
        .
        .
        err = OSMboxPostOpt(CommMbox, (void *)&CommRxBuf[0], OS_POST_OPT_BROADCAST);
        .
        .
    }
}
```

OSMboxQuery()

INT8U OSMboxQuery(OS_EVENT *pevent, OS_MBOX_DATA *pdata);

所在章	所属文件	调用者	开关量
10	OS_MBOX.C	任务或中断	OS_MBOX_EN 和 OS_MBOX_QUERY_EN

OSMboxQuery()用于取得消息邮箱的消息。用户程序必须分配一个 OS_MBOX_DATA 的数据结构,该结构用于从消息邮箱的事件控制块接收数据。通过调用 OSMboxQuery()函数,可以知道任务是否在等待消息,以及有多少个任务在等待消息,还可以检查消息邮箱现在的消息。

参 数

pevent 指向即将接收消息的消息邮箱的指针。在建立该消息邮箱时可以得到该指针的值(见 OSMboxCreate()函数)。

pdata 指向 OS_MBOX_DATA 数据结构的指针,该数据结构包含下述成员:

```
void     *OSMsg;                          /* 消息邮箱中消息的复制    */
INT8U     OSEventTbl[OS_EVENT_TBL_SIZE];  /* 消息邮箱等待队列的复制 */
INT8U     OSEventGrp;
```

返回值

OSMboxQuery()函数返回值为下述之一:

OS_NO_ERR	调用成功。
OS_ERR_EVENT_NULL	pevent 是空指针。
OS_ERR_EVENT_TYPE	pevent 不是指向消息邮箱的指针。

注意/警告

必须先建立消息邮箱,然后才能使用。

范 例

```
OS_EVENT *CommMbox;


void Task (void *pdata)
{
        OS_MBOXDATA mbox_data;
        INT8U       err;


        pdata = pdata;
        for (;;) {
            .
            .
            err = OSMboxQuery(CommMbox, &mbox_data);
            if (err == OS_NO_ERR) {
                /* 如果 mbox_data.OSMsg 为非空指针,说明消息邮箱非空 */
            }
            .
            .
        }
}
```

OSMemCreate()

OS_MEM *OSMemCreate(void *addr, INT32U nblks, INT32U blksize, INT8U *err);

所在章	所属文件	调用者	开关量
12	OS_MEM.C	任务或启动代码	OS_MEM_EN

OSMemCreate()函数建立并初始化一块内存区。一块内存区包含指定数目的、大小确定的内存块。程序可以包含这些内存块，并在用完后释放回内存区。

参 数

addr 建立的内存区的起始地址。内存区可使用静态数组或在初始化时使用malloc()函数建立。

nblks 需要的内存块的数目。每一个内存区需要定义至少2个内存块。

blksize 每个内存块的大小。每个内存块应该能够容纳至少一个指针。

err 指向包含错误码的变量的指针。OSMemCreate()函数返回的错误码可能为下述几种之一：

OS_NO_ERR	成功建立内存区。
OS_MEM_INVALID_ADDR	非法地址，即地址为空指针。
OS_MEM_INVALID_PART	没有空闲的内存区。
OS_MEM_INVALID_BLKS	没有为每一个内存区建立至少2个内存块。
OS_MEM_INVALID_SIZE	内存块大小不足以容纳一个指针变量。

返回值

OSMemCreate()函数返回指向内存区控制块的指针。如果没有剩余内存区，那么函数OSMemCreate()返回空指针。

注意/警告

必须首先建立内存区，然后才能使用。

范 例

```
OS_MEM *CommMem;
INT8U   CommBuf[16][128];


void main (void)
{
      INT8U err;


      OSInit();                          /* 初始化 μC/OS-II */
      .
      .
      CommMem = OSMemCreate(&CommBuf[0][0], 16, 128, &err);
      .
      .
      OSStart();                         /* 启动多任务内核 */
}
```

OSMemGet()

```
void    *OSMemGet(OS_MEM *pmem, INT8U *err);
```

所在章	所属文件	调用者	开关量
12	OS_MEM.C	任务或中断	OS_MEM_EN

OSMemGet()函数用于从内存区分配一个内存块。用户程序必须知道所建立的内存块的大小,同时用户程序必须在使用完内存块后,使用OSMemPut()函数释放内存块。可以多次调用OSMemGet()函数。

参 数

pmem 指向内存区控制块的指针,可以从OSMemCreate()函数返回得到。

err 指向包含错误码的变量的指针。OSMemGet()函数返回的错误码可能为下述几种之一:

OS_NO_ERR	成功得到一个内存块。
OS_MEM_NO_FREE_BLKS	内存区已经没有空间分配给内存块。
OS_MEM_INVALID_PMEM	pmem是空指针。

返回值

OSMemGet()函数返回指向内存块的指针。如果没有空间分配给内存块,OSMemGet()函数则返回空指针。

注意/警告

必须首先建立内存区,然后才能使用。

范 例

```
OS_MEM *CommMem;


void Task (void *pdata)
{
      INT8U *msg;


      pdata = pdata;
      for (;;) {
         msg = OSMemGet(CommMem, &err);
         if (msg != (INT8U *)0) {
            .                          /* 内存块已经分配 */
            .
         }
         .
         .
      }
}
```

OSMemPut()

INT8U OSMemPut(OS_MEM *pmem, void *pblk);

所在章	所属文件	调用者	开关量
12	OS_MEM.C	任务或中断	OS_MEM_EN

OSMemPut()函数释放一个内存块,内存块必须释放回原先申请的内存区。

参 数

pmem 指向内存区控制块的指针,可以从OSMemCreate()函数返回得到。

pblk 指向将被释放的内存块的指针。

返回值

OSMemPut()函数的返回值为下述之一:

OS_NO_ERR	成功释放内存块。
OS_MEM_FULL	内存区已不能再接收更多释放的内存块。这种情况说明程序出现了错误,释放了多于用OSMemGet()得到的内存块。
OS_MEM_INVALID_PMEM	pmem是空指针。
OS_MEM_INVALID_PBLK	pblk是空指针。

注意/警告

1. 必须首先建立内存区,然后再使用。
2. 内存块必须释放回原先申请的内存区。

范 例

```
OS_MEM  *CommMem;
INT8U   *CommMsg;


void Task (void *pdata)
{
      INT8U err;


      pdata = pdata;
      for (;;) {
         err = OSMemPut(CommMem, (void *)CommMsg);
         if (err == OS_NO_ERR) {
            .                                  /* 释放内存块 */
            .
         }
         .
         .
      }
}
```

OSMemQuery()

INT8U OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata);

所在章	所属文件	调用者	开关量
12	OS_MEM.C	任务或中断	OS_MEM_EN 和 OS_MEM_QUERY_EN

OSMemQuery()函数得到内存区的信息。该函数返回 OS_MEM 结构包含的信息,但使用了一个新的 OS_MEM_DATA 的数据结构。OS_MEM_DATA 数据结构还包含了正被使用的内存块数目的域。

参 数

pmem 指向内存区控制块的指针,可以从 OSMemCreate()函数返回得到。

pdata 指向 OS_MEM_DATA 数据结构的指针,该数据结构包含了以下的域:

```
void      *OSAddr;        /* 指向内存区起始地址的指针          */
void      *OSFreeList;    /* 指向空闲内存块列表起始地址的指针  */
INT32U     OSBlkSize;     /* 每个内存块的大小                  */
INT32U     OSNBlks;       /* 该内存区的内存块总数              */
INT32U     OSNFree;       /* 空闲的内存块数目                  */
INT32U     OSNUsed;       /* 使用的内存块数目                  */
```

返回值

OSMemQuery()函数返回以下错误代码之一:

OS_NO_ERR	存储块有效,返回用户应用程序。
OS_MEM_INVALID_PMEM	pmem 是空指针。
OS_MEM_INVALID_PDATA	pdata 是空指针。

注意/警告

必须首先建立内存区,然后再使用。

范 例

```
OS_MEM        *CommMem;


void Task (void *pdata)
{
      INT8U        err;
      OS_MEM_DATA  mem_data;


      pdata = pdata;
      for (;;) {
          .
          .
          err = OSMemQuery(CommMem, &mem_data);
          .
          .
      }
}
```

OSMutexAccept()

INT8U OSMutexAccept(OS_EVENT *pevent, INT8U *err);

所在章	所属文件	调用者	开关量
8	OS_MUTEX.C	任 务	OS_MUTEX_EN

OSMutexAccept() 函数用于检查互斥型信号量，以判断某资源是否可以使用。与OSMutexPend()不同的是，若资源不能使用，则调用 OSMutexAccept() 函数的任务并不被挂起，即 OSMutexAccept() 仅查询状态。

参 数

pevent 指向管理某资源的互斥型信号量。程序在建立 mutex 时，得到该指针(参见OSMutexCreate())。

err 指向出错代码的指针。OSMutexAccept() 函数置 *err 为指向以下值之一：

OS_NO_ERR	调用成功。
OS_ERR_EVENT_TYPE	pevent 不是指向 mutex 类型的指针。
OS_ERR_PEVENT_NULL	pevent 是空指针。
OS_ERR_PEND_ISR	在中断服务子程序中调用 OSMutexAccept()。

返回值

如果 mutex 有效，OSMutexAccept() 函数则返回 1；如果 mutex 被其他任务占用，OSMutexAccept()则返回 0。

注意/警告

1. 必须先建立 mutex，然后才能使用。
2. 在中断服务子程序中不能调用 OSMutexAccept()函数。
3. 若使用 OSMutexAccept()获取 mutex 的状态，那么使用完共享资源后，必须调用 OSMutexPost() 函数释放 mutex。

范 例

```
OS_EVENT *DispMutex;


void Task (void *pdata)
{
    INT8U  err;
    INT8U  value;


    pdata = pdata;
    for (;;) {
        value = OSMutexAccept(DispMutex, &err);
        if (value == 1) {
            .                          /* 资源可用 */
            .
        } else {
            .                          /* 资源不可用 */
            .
        }
        .
        .
    }
}
```

OSMutexCreate()

OS_EVENT *OSMutexCreate(INT8U prio, INT8U *err);

所在章	所属文件	调用者	开关量
8	OS_MUTEX.C	任务或启动代码	OS_MUTEX_EN

OSMutexCreate() 用于互斥型信号量 mutex 的建立和初始化。在与共享资源打交道时，使用 mutex 可以保证满足互斥条件。

参 数

prio 优先级继承优先级(PIP)。当一个高优先级的任务想要得到某 mutex，而此时这个 mutex 却被一个低优先级的任务占用时，低优先级任务的优先级可以提升到 PIP，直到其释放共享资源。

err 指向出错代码的指针。出错代码为以下值之一：

OS_NO_ERR	调用成功，mutex 已被成功地建立。
OS_ERR_CREATE_ISR	试图在中断服务子程序中建立 mutex。
OS_PRIO_EXIST	优先级为 PIP 的任务已经存在。
OS_ERR_PEVENT_NULL	已经没有 OS_EVENT 结构可以使用的了。
OS_PRIO_INVALID	定义的优先级非法，其值大于 OS_LOWEST_PRIO。

返回值

返回一个指针，该指针指向分配给 mutx 的事件控制块。如果得不到事件控制块，则返回一个空指针。

注意/警告

1. 必须先建立 mutex，然后才能使用 mutex。
2. 必须确保优先级继承优先级，即 prio 高于可能与相应共享资源打交道的任务中优先级最高的任务的优先级。例如，若有 3 个优先级分别为 20，25，30 的任务会使用 mutex，那么 prio 的值必须小于 20；并且，已经建立了的任务没有占用这个优先级。

范 例

```
OS_EVENT *DispMutex;

void main (void)
{
    INT8U  err;

    .
    .
    OSInit();                          /* 初始化 μC/OS-II */
    .
    .
    DispMutex = OSMutexCreate(20, &err);      /* 建立用于显示的 Mutex */
    .
    .
    OSStart();                         /* 开始的任务 */
}
```

OSMutexDel()

OS_EVENT *OSMutexDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

所在章	所属文件	调用者	开关量
8	OS_MUTEX.C	任 务	OS_MUTEX_EN 和 OS_MUTEX_DEL_EN

OSMutexDel() 函数用于删除一个 mutex。使用这个函数有风险，因为多任务中其他任务可能还想用这个实际上已经被删除了的 mutex。使用这个函数时必须十分小心，一般地说，要删除一个 mutex，首先应删除可能会用到这个 mutex 的所有任务。

参 数

pevent 指向 mutex 的指针。应用程序建立 mutex 时得到该指针（见 OSMutexCreate()）。

opt 该选项定义 mutex 的删除条件。可以选择只能在已经没有任何任务在等待该 mutex时，才能删除（OS_DEL_NO_PEND）；或者，不管有没有任务在等待这个 mutex，立即删除（OS_DEL_ALWAYS），在这种情况下，所有等待 mutex 的任务都立即进入就绪态。

err 指向出错代码的指针。出错代码可以是以下值之一：

OS_NO_ERR	调用成功，mutex 已经删除。
OS_ERR_DEL_ISR	试图在中断服务子程序中删除 mutex。
OS_ERR_INVALID_OPT	定义的 opt 参数无效，不是上面提到的 2 个参数之一。
OS_ERR_TASK_WAITING	定义了 OS_DEL_NO_PEND，而有一个或一个以上的任务在等这个 mutex。
OS_ERR_EVENT_TYPE	pevent 不是指向 mutex 的指针。
OS_ERR_PEVENT_NULL	已经没有可以使用的 OS_EVENT 数据结构了。*

返回值

如果 mutex 已经删除，则返回空指针；如果 mutex 没能删除，则返回 pevent。在后一种情况下，程序应检查出错代码，以查明原因。

注意/警告

使用本函数要特别小心，因为其他任务可能还要用这个 mutex。

* 原文如此。应为"传递的 pevent 为一空指针。"详见源程序。——译者注

范 例

```
OS_EVENT *DispMutex;


void Task (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    while (1) {

        .
        DispMutex = OSMutexDel(DispMutex, OS_DEL_ALWAYS, &err);
        if (DispMutex == (OS_EVENT *)0) {
            /* mutex已经被删除了 */
        }
        .
        .
    }
}
```

OSMutexPend()

void OSMutexPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

所在章	所属文件	调用者	开关量
8	OS_MUTEX.C	只能是任务	OS_MUTEX_EN

当任务需要独占共享资源时，应使用 OSMutexPend() 函数。如果任务在调用本函数时共享资源可以使用，则 OSMutexPend() 函数返回，调用 OSMutexPend() 函数的任务得到了 mutex。

注意：OSMutexPend()实际上并没有"给"调用本函数的任务什么值，只不过参数 err 的值被置为 OS_NO_ERR，调用本函数的任务好像得到了 mutex 并继续运行。

然而，如果 mutex 已经被别的任务占用了，那么 OSMutexPend()函数就将调用该函数的任务放入等待 mutex 的任务列表中，这个任务于是进入了等待状态，直到占有 mutex 的任务释放了 mutex 以及共享资源，或者直到定义的等待时限超时。如果在等待时限内 mutex 得以释放，那么 μC/OS-II 恢复运行等待 mutex 的任务中优先级最高的任务。

注意：如果 mutex 被优先级较低的任务占用了，那么 OSMutexPend()会将占用 mutex 的任务的优先级提升到优先级继承优先级 PIP。PIP 是在 mutex 建立时定义的(参见函数 OSMutexCreate())。

参 数

pevent　指向 mutex 的指针。应用程序是在建立 mutex 时得到该指针的(参见 OSMutexCreate())。

timeout　以时钟节拍数目定义的等待超时时限。如果在这一时限中得不到 mutex，任务将恢复执行。timeout 的值为 0，表示将无限期地等待 mutex。timeout 的最大值是 65 535 个时钟节拍。timeout 的值并不与时钟节拍同步，timeout 计数器在下一个时钟节拍到来时开始递减。在这里，所谓下一个时钟节拍，也可能立刻就到来了。

err　指向出错代码的指针。OSMutexPend()置出错代码 *err 为以下值之一：

OS_NO_ERR	调用成功，mutex 可以使用。
OS_TIMEOUT	在定义的等待时限内得不到 mutex。
OS_ERR_EVENT_TYPE	用户没能向 OSMutexPend() 传递指向 mutex的指针。
OS_ERR_PEVENT_NULL	pevent 是空指针。
OS_ERR_PEND_ISR	试图在中断服务子程序中获得 mutex。

返回值

无。

注意/警告

1. 必须先建立 mutex，然后才能使用 mutex。
2. 不要将占有 mutex 的任务挂起，也不要让占有 mutex 的任务等待 μC/OS-II 提供的信号量、邮箱及消息队列等，不要将占用 mutex 的任务延迟。换言之，用户代码应该抓紧时间，尽量快地释放共享资源。

范 例

```
OS_EVENT *DispMutex;


void  DispTask (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    for (;;) {
        .
        .
        OSMutexPend(DispMutex, 0, &err);
        .                                   /* 使这个任务得以继续运行下去的惟一途径是… */
        .                                   /* mutex 可以使用…                          */
    }
}
```

OSMutexPost()

INT8U OSMutexPost(OS_EVENT *pevent);

所在章	所属文件	调用者	开关量
8	OS_MUTEX.C	任 务	OS_MUTEX_EN

调用 OSMutexPost() 可以发出 mutex。只有当用户程序已调用 OSMutex-Accept() 或 OSMutexPend() 请求得到 mutex 时，OSMutexPost() 函数才起作用。当优先级较高的任务试图得到 mutex 时，如果占用 mutex 的任务的优先级已经被升高，那么 OSMutexPost() 函数使优先级升高了的任务恢复原来的优先级。如果有一个以上的任务在等待这个 mutex，那么等待 mutex 的任务中优先级最高的任务将得到 mutex。然后本函数会调用调度函数，看被唤醒的任务是不是进入就绪态任务中优先级最高的任务。如果是，则做任务切换，让这个任务运行。如果没有等待 mutex 的任务，那么本函数只不过是将 mutex 的值设为 0xFF，表示 mutex 可以使用。

参 数

pevent 指向 mutex 的指针。应用程序是在建立 mutex 时得到该指针的（参见函数 OSMutexCreate()）。

返回值

OSMutexPost() 返回以下出错代码之一：

OS_NO_ERR	调用成功，mutex 被释放。
OS_ERR_EVENT_TYPE	向 OSMutexPost() 传递的不是指向 mutex的指针。
OS_ERR_PEVENT_NULL	pevent 是空指针。
OS_ERR_POST_ISR	试图在中断服务子程序中调用 OSMutexPost() 函数。
OS_ERR_NOT_MUTEX_OWNER	发出 mutex 的任务实际上并不占有 mutex。

注意/警告

1. 必须先建立 mutex，然后才能使用 mutex。
2. 在中断服务子程序中不能调用 OSMutexPost() 函数。

范 例

```
OS_EVENT   *DispMutex;


void  TaskX (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    for (;;)
        .
        .
        err = OSMutexPost(DispMutex);
        switch (err) {
           case OS_NO_ERR:          /* mutex 已发出 */
                .
                .
                break;

           case OS_ERR_EVENT_TYPE:
                .
                .
                break;

           case OS_ERR_PEVENT_NULL:
                .
                .
                break;

           case OS_ERR_POST_ISR:
                .
                .
                break;

        }
        .
        .
    }
}
```

OSMutexQuery()

INT8U OSMutexQuery(OS_EVENT *pevent, OS_MUTEX_DATA *pdata);

所在章	所属文件	调用者	开关量
8	OS_MUTEX.C	任 务	OS_MUTEX_EN 和 OS_MUTEX_QUERY_EN

调用 OSMutexQuery() 函数可以得到 mutex 当前状态的信息。应用程序必须给 OS_MUTEX_DATA 数据结构分配存储空间，这个数据结构用于接收来自 mutex 的事件控制块的数据。通过调用 OSMutexQuery() 函数，得知是否有其他任务在等待 mutex。计算在 .OSEventTbl[]中有几个任务在等待 mutex(计算有几个 1)，得到 PIP 的值，以及确认 mutex 是否可以使用(是 1 还是 0)。

注意：.OSEventTbl[] 的容量大小是由 uCOS_II.H 文件中的以下语句定义的：

#define constant OS_EVENT_TBL_SIZE。

参 数

pevent 指向 mutex 的指针。应用程序是在建立 mutex 时得到该指针的(参见函数 OSMutexCreate())。

pdata 指向类型为 OS_MUTEX_DATA 的数据结构的指针。这个数据结构包括以下域：

```
INT8U  OSMutexPIP;        /* mutex 的优先级继承优先级 PIP                    */
INT8U  OSOwnerPrio;       /* 占用 mutex 任务的优先级                         */
INT8U  OSValue;           /* 当前 mutex 的值。1 表示可以使用;0 表示不能使用* */
INT8U  OSEventGrp;        /* 复制等待 mutex 的任务列表                       */
INT8U  OSEventTbl[OS_EVENT_TBL_SIZE];
```

返回值

OSMutexQuery() 返回以下出错代码之一：

OS_NO_ERR	调用成功。
OS_ERR_EVENT_TYPE	传给 OSMutexQuery() 的不是指向 mutex 的指针。
OS_ERR_PEVENT_NULL	pevent 是空指针。
OS_ERR_QUERY_ISR	试图在中断服务子程序中调用 OSMutexQuery()。

* 实际上，非 0 值即表示可以使用。——译者注

注意/警告

1. 必须先建立 mutex，然后才能使用 mutex。
2. 在中断服务子程序中不能调用 OSMutexPost()函数。

范 例

本例中，检查 mutex 的当前状况，找出等待 mutex 的任务中优先级最高的任务。

```
OS_EVENT *DispMutex;


void Task (void *pdata)
{
    OS_MUTEX_DATA mutex_data;
    INT8U         err;
    INT8U         highest;        /* 等待 mutex 的任务中优先级最高的任务 */
    INT8U         x;
    INT8U         y;


    pdata = pdata;
    for (;;) {
        .
        .
        err = OSMutexQuery(DispMutex, &mutex_data);
        if (err == OS_NO_ERR) {
            if (mutex_data.OSEventGrp != 0x00) {
                y       = OSUnMapTbl[mutex_data.OSEventGrp];
                x       = OSUnMapTbl[mutex_data.OSEventTbl[y]];
                highest = (y << 3) + x;
                .
                .
            }
        }
        .
        .
    }
}
```

OSQAccept()

```
void    *OSQAccept(OS_EVENT    *pevent);
```

所在章	所属文件	调用者	开关量
11	OS_Q.C	任务或中断	OS_Q_EN

OSQAccept()函数检查消息队列中是否已经有需要的消息。不同于OSQPend()函数，如果没有需要的消息，那么OSQAccept()函数并不挂起任务。如果消息已经到达，该消息被传递到用户任务。通常中断调用该函数，因为中断不允许挂起等待消息。

参 数

pevent 指向需要查看的消息队列的指针。当建立消息队列时，该指针返回到用户程序(见OSMboxCreate()函数)。

返回值

如果消息已经到达，则返回指向该消息的指针；如果消息队列没有消息，则返回空指针。

注意/警告

必须先建立消息队列，然后才能使用。

范 例

```
OS_EVENT *CommQ;

void Task (void *pdata)
{
    void *msg;


    pdata = pdata;
    for (;;) {
        msg = OSQAccept(CommQ);        /* 检查消息队列 */
        if (msg != (void *)0) {
                                       /* 处理接收的消息 */
            .
            .
```

```
        } else {
            .                       /* 没有消息… */
            .                       /* …做其他事 */
        }
        .
        .
    }
}
```

OSQCreate()

OS_EVENT *OSQCreate(void **start, INT8U size);

所在章	所属文件	调用者	开关量
11	OS_Q.C	任务或启动代码	OS_Q_EN

OSQCreate()函数建立一个消息队列。任务或中断可以通过消息队列向其他一个或多个任务发送消息。消息的含义是与具体的应用密切相关的。

参 数

start 消息内存区的基地址。消息内存区是一个指针数组。

size 消息内存区的大小。

返回值

OSQCreate()函数返回一个指向消息队列事件控制块的指针。如果没有空余的事件控制块,OSQCreate()函数则返回空指针。

注意/警告

必须先建立消息队列,然后才能使用。

范 例

```
OS_EVENT *CommQ;
void     *CommMsg[10];

void main (void)
{
    OSInit();                                   /* 初始化 μC/OS-II    */
    .
    .
    CommQ = OSQCreate(&CommMsg[0], 10);         /* 建立消息队列       */
    .
    .
    OSStart();                                  /* 启动多任务内核     */
}
```

OSQDel()

OS_EVENT *OSQDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

所在章	所属文件	调用者	开关量
11	OS_Q.C	任 务	OS_Q_EN 和 OS_Q_DEL_EN

OSQDel() 用于删除消息队列。使用这个函数有风险，因为多任务中的其他任务可能还想用这个消息队列。使用这个函数要特别小心。一般地说，应先删除可能会用到这个消息队列的所有任务，再调用本函数。

参 数

pevent 指向消息队列的指针。该指针的值在建立消息队列时可以得到(见函数 OSQCreate()函数)。

opt 该选项定义消息队列的删除条件。可以选择只能在已经没有任何任务在等待该消息队列的消息时，才能删除该消息队列(OS_DEL_NO_PEND)；或者，不管有没有任务在等待消息队列的消息，立即删除消息队列(OS_DEL_ALWAYS)，在这种情况下，所有等待消息的任务都立即进入就绪态。

err 指向出错代码的指针。出错代码可以是以下值之一：

OS_NO_ERR	调用成功，消息队列已被删除。
OS_ERR_DEL_ISR	试图在中断服务子程序中删除消息队列。
OS_ERR_INVALID_OPT	opt 参数没有定义成上述 2 个值之一。
OS_ERR_TASK_WAITING	有一个或一个以上的任务在等待消息队列中的消息。
OS_ERR_EVENT_TYPE	pevent 不是指向消息队列的指针。
OS_ERR_PEVENT_NULL	已经没有 OS_EVENT 数据结构可以使用了。*

返回值

如果消息队列删除成功，则返回空指针；若消息队列没能被删除，则返回 pevent。在后一种情况下，须查看出错代码，找出原因。

注意/警告

1. 调用本函数要特别小心，因为其他任务可能还想使用这个消息队列。
2. 当挂起的任务进入就绪态时，中断是关闭的，这就意味着中断延迟时间取决于等待消息队列的任务数目。

* 原文如此。应为"传递的 pevent 为一空指针。"详见相应源程序。——译者注

范 例

```
OS_EVENT *DispQ;


void Task (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    while (1) {
        .
        .
        DispQ = OSQDel(DispQ, OS_DEL_ALWAYS, &err);
        if (DispQ == (OS_EVENT *)0) {
            /* 消息队列已被删除 */
        }
        .
        .
    }
}
```

OSQFlush()

INT8U *OSQFlush(OS_EVENT *pevent);

所在章	所属文件	调用者	开关量
11	OS_Q.C	任务或中断	OS_Q_EN 和 OS_Q_FLUSH_EN

OSQFlush()函数清空消息队列，并且忽略发往队列的所有消息。不管队列中是否有消息，这个函数的执行时间都是相同的。

参 数

pevent 指向消息队列的指针。该指针的值是在建立消息队列时得到的(参见函数 OSQCreate())。

返回值

OSQFlush()函数的返回值为下述之一：

OS_NO_ERR	消息队列被成功清空。
OS_ERR_EVENT_TYPE	试图清除不是消息队列的对象。
OS_ERR_PEVENT_NULL	pevent 是空指针。

注意/警告

必须先建立消息队列，然后才能使用。

范 例

```
OS_EVENT *CommQ;

void main (void)
{
        INT8U err;

        OSInit();                          /* 初始化 μC/OS-II */
        .
        .
        err = OSQFlush(CommQ);
        .
        .
        OSStart();                         /* 启动多任务内核 */
}
```

OSQPend()

```
void *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
```

所在章	所属文件	调用者	开关量
11	OS_Q.C	任 务	OS_Q_EN

OSQPend()函数用于任务等待消息。消息通过中断或另外的任务发送给需要的任务。消息是一个以指针定义的变量，在不同的程序中消息的使用也可能不同。如果调用OSQPend()函数时，队列中已经存在需要的消息，那么该消息被返回给OSQPend()函数的调用者，队列中清除该消息；若调用OSQPend()函数时，队列中没有需要的消息，那么OSQPend()函数挂起当前任务，直到得到需要的消息或超出定义的超时时限。如果同时有多个任务等待同一则消息，那么μC/OS-II默认最高优先级的任务取得消息，并且任务恢复执行。一个由OSTaskSuspend()函数挂起的任务也可以接收消息，但这个任务将一直保持挂起状态，直到通过调用OSTaskResume()函数恢复任务的运行。

参 数

pevent　指向即将接收消息的队列的指针。该指针的值在建立该队列时可以得到(见OSMboxCreate()函数)。

timeout　允许一个任务在经过了指定数目的时钟节拍后还没有得到需要的消息时，恢复运行状态。如果该值为0，则表示任务将持续地等待消息。最长的等待时间为65 535个时钟节拍。这个时间长度并不是非常严格的，可能存在一个时钟节拍的误差，因为只有在一个时钟节拍结束后，才会减少定义的等待超时时钟节拍。

err　指向包含错误码的变量的指针。OSQPend()函数返回的错误码可能为下述几种之一：

OS_NO_ERR	消息被正确地接收。
OS_TIMEOUT	消息没有在指定的等待时限内送到。
OS_ERR_EVENT_TYPE	pevent不是指向消息队列的指针。
OS_ERR_PEVENT_NULL	pevent是空指针。
OS_ERR_PEND_ISR	在中断中调用该函数。虽然规定了不允许从中断调用该函数，但μC/OS-II仍然包含了检测这种情况的功能。

返回值

OSQPend()返回接收的消息，并将*err置为OS_NO_ERR。若没有在指定数目的时钟节拍内接收到需要的消息，OSQPend()则返回空指针，且将*err设置为OS_TIMEOUT。

注意/警告

1. 必须先建立消息邮箱，然后才能使用。
2. 不允许在中断中调用该函数。

范 例

```
OS_EVENT *CommQ;


void CommTask (void *data)
{
    INT8U   err;
    void   *msg;


    pdata = pdata;
    for (;;) {
        .
        .
        msg = OSQPend(CommQ, 100, &err);
        if (err == OS_NO_ERR) {
            .
            .                    /* 在指定时间内接收到消息 */
            .
        } else {
            .
            .                    /* 在指定的时间内没有接收到消息 */
            .
        }
        .
        .
    }
}
```

OSQPost()

INT8U OSQPost(OS_EVENT *pevent, void *msg);

所在章	所属文件	调用者	开关量
11	OS_Q.C	任务或中断	OS_Q_EN 和 OS_Q_POST_EN

OSQPost()函数通过消息队列向任务发送消息。消息是一个以指针定义的变量，在不同的程序中消息的使用也可能不同。如果队列中已经存满消息，则返回错误码。OSQPost()函数立即返回调用者，消息也没有能够发到队列。如果有任何任务在等待队列中的消息，那么最高优先级的任务将得到这则消息。如果等待消息的任务优先级比发送消息的任务优先级高，那么高优先级的任务将得到消息而恢复执行。也就是说，发生了一次任务切换。消息队列是先入先出(FIFO)机制的，先进入队列的消息先被传递给任务。

参 数

pevent　指向即将接收消息的消息队列的指针。该指针的值在建立消息队列时可以得到(见 OSQCreate()函数)。

msg　即将发送给任务的消息。消息是一个指向某种变量类型的指针，在不同的应用程序中，消息的类型也可能不同。不允许传递一个空指针。

返回值

OSQPost()函数的返回值为下述之一：

OS_NO_ERR	消息成功地放到消息队列中。
OS_Q_FULL	消息队列已满。
OS_ERR_EVENT_TYPE	pevent 不是指向消息队列的指针。
OS_ERR_PEVENT_NULL	pevent 是空指针。
OS_ERR_POST_NULL_PTR	用户发出空指针。根据规则，这里不支持空指针。

注意/警告

1. 必须先建立消息队列，然后使用。
2. 不允许传递一个空指针。

范 例

```
OS_EVENT  *CommQ;
INT8U     CommRxBuf[100];


void CommTaskRx (void *pdata)
{
      INT8U  err;


      pdata = pdata;
      for (;;) {
          .
          .
          err = OSQPost(CommQ, (void *)&CommRxBuf[0]);
          switch (err) {
              case OS_NO_ERR:
                                  /* 消息已放入消息队列 */
                  break;

              case OS_Q_FULL:
                                  /* 消息队列已满 */
                  break;
              .
          }
          .
          .
      }
}
```

OSQPostFront()

INT8U OSQPostFront(OS_EVENT *pevent, void *msg);

所在章	所属文件	调用者	开关量
11	OS_Q.C	任务或中断	OS_Q_EN 和 OS_Q_POST_FRONT_EN

OSQPostFront()函数通过消息队列向任务发送消息。OSQPostFront()与OSQPost()函数非常相似,不同之处在于,OSQPostFront()函数将发送的消息插到消息队列的最前端。也就是说,OSQPostFront()函数使得消息队列按照后进先出(LIFO)的方式工作,而不是先进先出(FIFO)。消息是一个以指针定义的变量,在不同的程序中消息的使用也可能不同。如果队列中已经存满消息,则返回错误码。OSQPostFront()函数立即返回调用者,消息也没能发到队列。如果有任何任务在等待队列中的消息,则最高优先级的任务将得到这则消息。如果等待消息的任务优先级比发送消息的任务优先级高,那么高优先级的任务将得到消息而恢复执行。也就是说,发生了一次任务切换。

参 数

pevent 指向即将接收消息的消息队列的指针。该指针的值在建立该消息队列时可以得到(见 OSQCreate()函数)。

msg 即将实际发送给任务的消息。消息是一个以指针定义的变量,在不同的程序中消息的使用也可能不同。不允许传递一个空指针。

返回值

OSQPostFront()函数的返回值为下述之一:

OS_NO_ERR	消息已成功地放到消息队列中。
OS_Q_FULL	消息队列已满。
OS_ERR_EVENT_TYPE	pevent 不是指向消息队列的指针。
OS_ERR_PEVENT_NULL	pevent 是空指针。
OS_ERR_POST_NULL_PTR	用户发出空指针,根据规则,这里不支持空指针。

注意/警告

1. 必须先建立消息队列,然后才能使用。
2. 不允许传递一个空指针。

范 例

```
OS_EVENT *CommQ;
INT8U    CommRxBuf[100];


void CommTaskRx (void *pdata)
{
      INT8U  err;


      pdata = pdata;
      for (;;) {
          .
          .
          err = OSQPostFront(CommQ, (void *)&CommRxBuf[0]);
          switch (err) {
              case OS_NO_ERR:
                  /* 消息已放入消息队列 */
                  break;


              case OS_Q_FULL:
                  /* 消息队列已满 */
                  break;
              .
          }
          .
          .
      }
}
```

OSQPostOpt()

INT8U OSQPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);

所在章	所属文件	调用者	开关量
11	OS_Q.C	任务或中断	OS_Q_EN 和 OS_Q_POST_OPT_EN

OSQPostOpt() 函数通过消息队列向任务发消息。消息是一个以指针表示的某种数据类型的变量，在不同的程序中消息的使用也可能不同。如果消息队列中的消息已满，则返回出错代码，说明消息队列已满。OSQPostOpt()函数立即返回调用者，消息也没有能够发到消息队列。如果有任何任务在等待消息队列的消息，那么 OSQPostOpt()允许选择以下 2 种情况之一：让最高优先级的任务得到这则消息(opt 置为 OS_POST_OPT_NONE)；或者让所有等待队列消息的任务都得到消息(opt 置为 OS_POST_OPT_BROADCAST)。无论在哪种情况下，如果得到消息的任务优先级比发送消息的任务优先级高，那么得到消息的最高优先级的任务将恢复执行，发消息的任务将被挂起。也就是说，发生了一次任务切换。

OSQPostOpt() 仿真 OSQPost()和 OSQPostFront()这 2 个函数，并允许程序发消息给多个任务。换句话说，OSQPostOpt() 允许将消息广播给所有的等待队列消息的任务。OSQPostOpt() 实际上可以取代 OSQPost()，因为可以通过设置 opt 参数定义队列消息的接收方式，这样做可以减小 μC/OS-II 占用的代码空间。

参 数

pevent　指向即将接收消息的消息队列的指针。该指针的值在建立该消息队列时可以得到(见 OSQCreate()函数)。

msg　即将发送给任务的消息。消息是一个指向某种变量类型的指针，在不同的应用程序中，消息的类型是用户定义的。不允许传递一个空指针。

opt　决定消息发送方式的选项，它可以是以下值之一：

OS_POST_OPT_NONE	发消息给一个等待消息的任务(等同于 OSQPost())。
OS_POST_OPT_BROADCAST	发消息给所有等待队列消息的任务。
OS_POST_OPT_FRONT	以后进先出方式发消息(仿真 OSQPostFront())。

以下是所有标志可能的组合：

OS_POST_OPT_NONE	等同于 OSQPost()。
OS_POST_OPT_FRONT	等同于 OSQPostFront()。
OS_POST_OPT_BROADCAST	等同于 OSQPost()，但广播给所有等待队列消息的任务。

OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST

等同于 OSQPostFront()，不同的是，它将消息广播给所有等待队列消息的任务。

返回值

err 指向出错代码的指针。出错代码可以为以下值之一：

OS_NO_ERR	调用成功，消息已经发出。
OS_Q_FULL	消息队列已满，不能再接收新消息。
OS_ERR_EVENT_TYPE	pevent 指向的数据类型错。
OS_ERR_PEVENT_NULL	pevent 是空指针。
OS_ERR_POST_NULL_PTR	用户程序试图发出空指针。

注意/警告

1. 必须先建立消息队列，然后才能使用。
2. 不允许传递一个空指针。
3. 如果想使用本函数，又希望压缩代码长度，则可以将 OSQPost()函数的开关量关闭（置 OS_CFG.H 文件中的 OS_Q_POST_EN 为 0），并将 OSQPostFront()的开关量关闭（置 OS_CFG.H 文件中的 OS_Q_POST_FRONT_EN 为 0），因为 OSQPostOpt()可以仿真这 2 个函数。
4. OSQPostOpt() 在广播方式下，即已将 opt 置为 OS_POST_OPT_BROADCAST，函数的执行时间取决于等待队列消息的任务的数目。

范 例

```
OS_EVENT *CommQ;
INT8U    CommRxBuf[100];


void CommRxTask (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    for (;;) {
        .
        .
        err = OSQPostOpt(CommQ, (void *)&CommRxBuf[0], OS_POST_OPT_BROADCAST);
        .
        .
    }
}
```

OSQQuery()

INT8U OSQQuery(OS_EVENT *pevent, OS_Q_DATA *pdata);

所在章	所属文件	调用者	开关量
11	OS_Q.C	任务或中断	OS_Q_EN 和 OS_QUERY_EN

OSQQuery()函数用于取得消息队列本身的消息。程序必须建立一个 OS_Q_DATA 的数据结构。该结构用于保存从消息队列的事件控制块得到的数据。通过调用 OSQQuery()函数,可以知道任务是否在等待消息,有多少个任务在等待消息,队列中有多少消息以及消息队列可以容纳的消息数。OSQQuery()函数还可以得到即将被传递给任务的消息的信息。

参 数

pevent 指向即将接收消息的消息队列的指针。该指针的值在建立该消息队列时可以得到(见 OSQCreate()函数)。

pdata 指向 OS_Q_DATA 数据结构的指针,该数据结构包含下述成员:

```
void      *OSMsg;                          /* 下一个可用的消息    */
INT16U     OSNMsgs;                        /* 队列中的消息数目    */
INT16U     OSQSize;                        /* 消息队列的大小      */
INT8U      OSEventTbl[OS_EVENT_TBL_SIZE];  /* 消息队列的等待列表 */
INT8U      OSEventGrp;
```

返回值

OSQQuery()函数的返回值为下述之一:

OS_NO_ERR	调用成功
OS_ERR_EVENT_TYPE	pevent 不是指向消息队列的指针。
OS_ERR_PEVENT_NULL	pevent 是空指针。

注意/警告

必须先建立消息队列,然后才能使用。

范 例

```
OS_EVENT *CommQ;


void Task (void *pdata)
{
      OS_Q_DATA qdata;
      INT8U     err;


      pdata = pdata;
      for (;;) {
          .
          .
          err = OSQQuery(CommQ, &qdata);
          if (err == OS_NO_ERR) {
             .    /* 消息队列中的消息 qdata 有效 */
          }
          .
          .
      }
}
```

OSSchedLock()

void OSSchedLock(void);

所在章	所属文件	调用者	开关量
3	OS_CORE.C	任务或中断	OS_SCHED_LOCK_EN

OSSchedLock()函数停止任务调度，只有使用配对的函数OSSchedUnlock()，才能重新开始内核的任务调度。调用OSSchedLock()函数的任务独占CPU，不管有没有其他高优先级的就绪任务。在这种情况下，中断仍然可以被接收和执行(中断必须允许)。OSSchedLock()和OSSchedUnlock()必须配对使用。μC/OS-II可以支持多达254层的OSSched-Lock()函数嵌套，必须调用同样次数的OSSchedUnlock()函数，才能恢复任务调度。

参 数

无。

返回值

无。

注意/警告

任务调用了OSSchedLock()函数后，绝不能再调用可能导致当前任务挂起的系统函数：OSTimeDly()，OSTimeDlyHMSM()，OSSemPend()，OSMboxPend()及OSQPend()。因为任务调度已经被禁止，其他任务不能运行，否则会导致系统死锁。

范 例

```
void TaskX (void *pdata)
{
    pdata = pdata;
    for (;;) {
        .
        OSSchedLock();          /* 停止任务调度 */
        .
        .                       /* 须使用禁止任务切换来保护的代码 */
        .
        OSSchedUnlock();        /* 恢复任务调度 */
        .
    }
}
```

OSSchedUnlock()

void OSSchedUnlock(void);

所在章	所属文件	调用者	开关量
3	OS_CORE.C	任务或中断	OS_SCHED_LOCK_EN

在调用了 OSSchedLock()函数后，OSSchedUnlock()函数恢复任务调度。

参 数

无。

返回值

无。

注意/警告

任务调用了 OSSchedLock()函数后，绝不能再调用可能导致当前任务挂起的系统函数：OSTimeDly()，OSTimeDlyHMSM()，OSSemPend()，OSMboxPend()及 OSQPend()。因为任务调度已经被禁止，其他任务不能运行，否则会导致系统死锁。

范 例

```
void TaskX (void *pdata)
{
    pdata = pdata;
    for (;;) {
        .
        OSSchedLock();          /* 停止任务调度 */
        .
        .                       /* 须使用禁止任务切换来保护的代码 */
        .
        OSSchedUnlock();        /* 恢复任务调度 */
        .
    }
}
```

OSSemAccept()

INT16U OSSemAccept(OS_EVENT *pevent);

所在章	所属文件	调用者	开关量
7	OS_SEM.C	任务或中断	OS_SEM_EN 和 OS_SEM_ACCEPT_EN

OSSemAccept()函数查看资源是否可以使用或事件是否发生。不同于OSSemPend()函数,如果资源不可使用,OSSemAccept()函数并不挂起任务。中断调用该函数查询信号量。

参 数

pevent 指向需要保护的共享资源的信号量。当建立信号量时,用户程序得到该指针(参见OSSemCreate()函数)。

返回值

当调用OSSemAccept()函数时,共享资源信号量的值大于0,则说明共享资源可以使用,这个值被返回调用者,信号量的值减1;如果调用OSSemAccept()函数时,信号量的值等于0,则说明资源不能使用,返回0。

注意/警告

必须先建立信号量,然后才能使用。

范 例

```
OS_EVENT *DispSem;

void Task (void *pdata)
{
        INT16U value;


    pdata = pdata;
    for (;;) {
        value = OSSemAccept(DispSem);      /* 查看共享资源是否可以使用 */
        if (value > 0) {
                                           /* 共享资源可以使用,执行处理代码 */
            .
            .
        }
        .
        .
    }
}
```

OSSemCreate()

OS_EVENT *OSSemCreate(INT16U value);

所在章	所属文件	调用者	开关量
7	OS_SEM.C	任务或启动代码	OS_SEM_EN

OSSemCreate()函数建立并初始化一个信号量。信号量的作用如下：

- 允许一个任务与其他任务或中断同步；
- 取得共享资源的使用权；
- 标志事件的发生。

参 数

value 建立的信号量的初始值，可以为 0～65 535 的任何值。

返回值

OSSemCreate()函数返回指向分配给所建立的消息邮箱的事件控制块的指针。如果没有可用的事件控制块，OSSemCreate()函数则返回空指针。

注意/警告

必须先建立信号量，然后才能使用。

范 例

```
OS_EVENT *DispSem;

void main (void)
{
    .
    .
    OSInit();                          /* 初始化 μC/OS-II */
    .
    .
    DispSem = OSSemCreate(1);          /* 建立显示设备的信号量 */
    .
    .
    OSStart();                         /* 启动多任务内核 */
}
```

OSSemDel()

OS_EVENT *OSSemDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

所在章	所属文件	调用者	开关量
7	OS_SEM.C	任 务	OS_SEM_EN 和 OS_SEM_DEL_EN

OSSemDel() 函数用于删除信号量。使用本函数有风险，因为多任务中的其他任务可能还想使用这个信号量。使用本函数须特别小心。一般地说，在删除信号量之前，应该先删除所有可能会用到这个信号量的任务。

参 数

pevent 指向信号量的指针。该指针的值在建立该信号量时得到（见 OSSemCreate() 函数）。

opt 该选项定义信号量的删除条件。可以选择只能在已经没有任何任务在等待该信号量时，才能删除该信号量（OS_DEL_NO_PEND）；或者，不管有没有任务在等待该信号量，立即删除这个信号量（OS_DEL_ALWAYS），在这种情况下，所有等待该信号量的任务都立即进入就绪态。

err 指向包含错误码的变量的指针。返回的错误码可能为下述几种之一：

OS_NO_ERR	调用成功，信号量已被删除。
OS_ERR_DEL_ISR	试图在中断服务子程序中删除信号量。
OS_ERR_INVALID_OPT	没有将 opt 参数定义为 2 种合法参数之一。
OS_ERR_TASK_WAITING	有一个或一个以上的任务在等待信号量。
OS_ERR_EVENT_TYPE	pevent 不是指向信号量的指针。
OS_ERR_PEVENT_NULL	已经没有可用的 OS_EVENT 数据结构了。*

返回值

如果信号量已被删除了，则返回空指针；若信号量没能被删除，则返回 pevent。后一种情况下，应该查看出错代码，以查明原因。

注意/警告

1. 使用本函数调用须特别小心，因为其他任务可能还会用到这个信号量。
2. 当挂起的任务进入就绪态时，中断是关闭的，这就意味着中断延迟时间与等待信号量的任务的数目有关。

* 原文如此。应为"传递的 pevent 为一空指针。"详见相应源程序。——译者注

范 例

```
OS_EVENT *DispSem;


void Task (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    while (1) {
        .
        DispSem = OSSemDel(DispSem, OS_DEL_ALWAYS, &err);
        if (DispSem == (OS_EVENT *)0) {
            /* 信号量已被删除 */
        }
        .

    }
}
```

OSSemPend()

void OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

所在章	所属文件	调用者	开关量
7	OS_SEM.C	只能是任务	OS_SEM_EN

OSSemPend()函数用于任务试图取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的发生的场合。若任务调用OSSemPend()函数,且信号量的值大于0,那么OSSemPend()递减该值并返回该值;若调用时信号量的值等于0,那么OSSemPend()函数将任务加入该信号量的等待列表中。OSSemPend()函数挂起当前任务,直到其他的任务或中断置位信号量或超出等待的时限。如果在预期的时钟节拍内信号量被置位,那么μC/OS-II默认最高优先级的任务取得信号量,并恢复执行。一个被OSTaskSuspend()函数挂起的任务,也可以接收信号量,但这个任务将一直保持挂起状态,直到通过调用OSTaskResume()函数恢复任务的运行。

参 数

pevent 指向信号量的指针。该指针的值在建立该信号量时得到(见OSSemCreate()函数)。

timeout 允许一个任务在经过指定数目的时钟节拍后还没有得到需要的信号量时,恢复运行状态。如果该值为0,则表示任务将持续地等待信号量。最长的等待时间为65 535个时钟节拍。这个时间长度并不是非常严格的,可能存在一个时钟节拍的误差,因为只有在一个时钟节拍结束后,才会给定义的等待超时时钟节拍减1。

err 指向包含错误码的变量的指针。OSSemPend()函数返回的错误码可能为下述几种之一:

OS_NO_ERR	信号量是可用的。
OS_TIMEOUT	信号量没有在指定的周期数内置位。
OS_ERR_EVENT_TYPE	pevent不是指向信号量的指针。
OS_ERR_PEND_ISR	在中断中调用该函数。虽然规定了不允许在中断中调用该函数,但μC/OS-II仍然包含了检测这种情况的功能。
OS_ERR_PEVENT_NULL	pevent是空指针。

返回值

无。

注意/警告

必须先建立信号量，然后才能使用。

范 例

```
OS_EVENT *DispSem;


void DispTask (void *pdata)
{
    INT8U  err;

    pdata = pdata;
    for (;;) {
        .
        .
        OSSemPend(DispSem, 0, &err);
        .                          /* 只有信号量有效，该任务才能执行 */
        .
    }
}
```

OSSemPost()

INT8U OSSemPost(OS_EVENT *pevent);

所在章	所属文件	调用者	开关量
7	OS_SEM.C	任务或中断	OS_SEM_EN

OSSemPost()函数置位指定的信号量。如果指定的信号量是0或大于0，OSSemPost()函数则递增该信号量并返回。如果有任何任务在等待信号量，那么最高优先级的任务将得到信号量并进入就绪态。如果被唤醒的任务是最高优先级的就绪态的任务，任务调度函数将进行任务调度。

参 数

pevent 指向信号量的指针。该指针的值在建立该信号量时可以得到（参见函数OSSemCreate()函数）。

返回值

OSSemPost()函数的返回值为下述之一：

OS_NO_ERR	信号量成功地置位。
OS_SEM_OVF	信号量的值溢出。
OS_ERR_EVENT_TYPE	pevent不指向信号量。
OS_ERR_PEVENT_NULL	pevent是空指针。

注意/警告

必须先建立信号量，然后才能使用。

范 例

```
OS_EVENT *DispSem;


void TaskX (void *pdata)
{
      INT8U  err;


      pdata = pdata;
      for (;;) {
          .
          .
          err = OSSemPost(DispSem);
          switch (err) {
              case OS_NO_ERR:
                    /* 信号量置起 */
                    break;

              case OS_SEM_OVF:
                    /* 信号量溢出 */
                    break;

          .
          }
          .
          .
      }
}
```

OSSemQuery()

INT8U OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata);

所在章	所属文件	调用者	开关量
7	OS_SEM.C	任务或中断	OS_SEM_EN 和 OS_SEM_QUERY_EN

OSSemQuery()函数用于获取某个信号量的信息。在使用 OSSemQuery()之前，应用程序需要先建立类型为 OS_SEM_DATA 的数据结构，用来保存从信号量的事件控制块中取得的数据。使用 OSSemQuery()可以得知，是否有以及有多少任务位于信号量的任务等待队列中（通过查询.OSEventTbl[ ]域中1的数目），还可以获取信号量的值。OSEventTbl[ ]域的大小由语句：#define constant OS_EVENT_TBL_SIZE 来定义（参阅文件 uCOS_II.H）。

参 数

pevent 一个指向信号量的指针。该指针在信号量建立后返回调用程序（参见 OSSemCreate()函数）。

pdata 一个指向数据结构 OS_SEM_DATA 的指针。该数据结构包含下述域：

```
INT16U  OSCnt;                          /* 当前信号量的值 */
INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];  /* 信号量等待列表 */
INT8U   OSEventGrp;
```

返回值

OSSemQuery()函数返回值为下述之一：

OS_NO_ERR	调用成功。
OS_ERR_EVENT_TYPE	pevent 不指向信号量。
OS_ERR_PEVENT_NULL	pevent 是空指针。

注意/警告

被操作的信号量必须是已经建立的。

范 例

在本例中，应用程序检查信号量，查找等待列表中优先级最高的任务。

```
OS_EVENT *DispSem;


void Task (void *pdata)
{
    OS_SEM_DATA sem_data;
    INT8U       err;
    INT8U       highest;  /* 等待信号量的优先级最高的任务 */
    INT8U       x;
    INT8U       y;


    pdata = pdata;
    for (;;) {
        .
        .
        err = OSSemQuery(DispSem, &sem_data);
        if (err == OS_NO_ERR) {
            if (sem_data.OSEventGrp != 0x00) {
                y       = OSUnMapTbl[sem_data.OSEventGrp];
                x       = OSUnMapTbl[sem_data.OSEventTbl[y]];
                highest = (y << 3) + x;
                .
                .
            }
        }
    }
}
```

OSStart()

void OSStart(void);

所在章	所属文件	调用者	开关量
3	OS_CORE.C	只能是启动代码	无

OSStart() 启动 μC/OS-II 的多任务环境。

参 数

无。

返回值

无。

注意/警告

在调用 OSStart()之前，必须先调用 OSInit()。在用户程序中只能调用 OSStart() 1 次。第 2 次调用 OSStart()将不进行任何操作。

范 例

```
void main (void)
{
    .                                   /* 用户代码          */
    .
    OSInit();                           /* 初始化 μC/OS-II   */
    .                                   /* 用户代码          */
    .
    OSStart();                          /* 启动多任务环境    */
    /* 写在这里的代码永远不会执行! */
}
```

OSStatInit()

void OSStatInit(void);

所在章	所属文件	调用者	开关量
3	OS_CORE.C	只能是启动代码	OS_TASK_STAT_EN 和 OS_TASK_CREATE_EXT_EN

OSStatInit()函数获取系统中没有其他任务运行时 32 位计数器所能达到的最大值。OSStatInit()的调用时机是，多任务环境已经启动，且系统中只有一个任务在运行。也就是说，该函数只能在第 1 个建立并运行的任务中调用。

参 数

无。

返回值

无。

注意/警告

无。

范 例

```
void FirstAndOnlyTask (void *pdata)
{
    .
    .
    OSStatInit();            /* 计算没有任务运行时 CPU 的容量 */
    .
    OSTaskCreate(...);       /* 建立其他任务 */
    OSTaskCreate(...);
    .
    for (;;) {
        .
        .
    }
}
```

OSTaskChangePrio()

INT8U OSTaskChangePrio(INT8U oldprio, INT8U newprio);

所在章	所属文件	调用者	开关量
4	OS_TASK.C	任 务	OS_TASK_CHANGE_PRIO_EN

OSTaskChangePrio()改变一个任务的优先级。

参 数

oldprio 任务原先的优先级。

newprio 任务的新优先级。

返回值

OSTaskChangePrio()的返回值为下述之一：

OS_NO_ERR	任务优先级被成功改变。
OS_PRIO_INVALID	参数中任务原先的优先级或新优先级的值大于或等于OS_LOWEST_PRIO。
OS_PRIO_EXIST	参数中任务的新优先级已经存在。
OS_PRIO_ERR	参数中任务原先的优先级不存在。

注意/警告

参数中的新优先级必须是没有使用过的，否则会返回错误码。在 OSTaskChangePrio()中，还会先判断要改变优先级的任务是否存在。

范 例

```
void TaskX (void *data)
{
    INT8U  err;

    for (;;) {
        .
        .
        err = OSTaskChangePrio(10, 15);
        .
        .
    }
}
```

OSTaskCreate()

```
INT8U OSTaskCreate(void (*task)(void *pd),
                   void *pdata,
                   OS_STK *ptos,
                   INT8U prio);
```

所在章	所属文件	调用者	开关量
4	OS_TASK.C	任务或启动代码	OS_TASK_CREATE_EN

OSTaskCreate()建立一个新任务。可以在多任务环境启动之前，或在正在运行的任务中建立任务。在中断处理程序中不能建立任务。一个任务必须为无限循环结构(如下所示)，且不能有返回点。

OSTaskCreate()是为了与先前的 μC/OS 版本保持兼容，在 OSTaskCreateExt()函数中新增加了许多特性。

无论用户程序中是否产生中断，在初始化任务堆栈时，堆栈的结构必须与 CPU 中断后寄存器入栈的顺序结构相同。详细说明请参考所用处理器的手册。

```
void Task (void *pdata)
{
    .                                   /* 对参数"pdata"进行操作          */
    for (;;) {                          /* 任务函数体                     */
        .
        .
        /* 在任务体中必须调用如下函数之一:
        /*     OSMboxPend()                                               */
        /*     OSFlagPend()                                               */
        /*     OSMutexPend()                                              */
        /*     OSQPend()                                                  */
        /*     OSSemPend()                                                */
        /*     OSTimeDly()                                                */
        /*     OSTimeDlyHMSM()                                            */
        /*     OSTaskSuspend()      (挂起任务本身)                        */
        /*     OSTaskDel()          (删除任务本身)                        */
        .
        .
    }
}
```

参 数

task 指向任务代码的指针。

pdata 一个指针，指向一个类型可以选择的数据区。该数据区中的变量用于在建立任务时向任务传递参数。建立任务的任务或建立任务的初始化代码应该把 pdata 传给要建立的任务。pdata 用于向建立的任务传递参数。例如，可以建立一个通用的任务，以管理串行通信口，pdata 可以把诸如串行口的地址、波特率、每次传送几位、校验方式以及其他一些参数传递给这个任务。

ptos 指向任务堆栈栈顶的指针。任务堆栈用于保存局部变量、函数参数、返回地址以及任务被中断时的 CPU 寄存器内容。任务堆栈的大小由任务的需要及预计的中断嵌套层数决定。计算堆栈的大小，须知道任务的局部变量所占的空间、可能产生嵌套调用的函数，以及中断嵌套所需要的空间。如果将初始化常量 OS_STK_GROWTH设为 1，则堆栈被设为从内存高地址向低地址递减，此时 ptos 应该指向任务堆栈空间的最高地址；反之，如果 OS_STK_GROWTH 设为 0，堆栈将从内存的低地址向高地址增长。

prio 任务的优先级。每个任务必须有一个惟一的优先级作为标志。数字越小，优先级越高（任务越重要）。

返回值

OSTaskCreate()的返回值为下述之一：

OS_NO_ERR	函数调用成功。
OS_PRIO_EXIST	具有该优先级的任务已经存在。
OS_PRIO_INVALID	参数指定的优先级大于 OS_LOWEST_PRIO。
OS_NO_MORE_TCB	系统中没有 OS_TCB 可以分配给任务了。

注意/警告

1. 任务堆栈必须声明为 OS_STK 类型。
2. 在任务中必须调用 μC/OS-II 提供的下述过程之一：延时等待、任务挂起及等待事件发生（信号量、消息邮箱及消息队列），以使其他任务得到 CPU。
3. 在用户程序中，不能够使用的优先级有：0，1，2，3，OS_LOWEST_PRIO-3，OS_LOWEST_PRIO-2，OS_LOWEST_PRIO-1 及 OS_LOWEST_PRIO。这些优先级被 μC/OS-II系统保留，其余的 56 个优先级提供给应用程序。

范例 1

本例中，传递给任务 Task1()的参数 pdata 不使用，所以指针 pdata 被设为 NULL。注意到程序中设定堆栈向低地址递减，传递的栈顶指针为高地址 &Task1Stk [1023]。如果在程序中设定堆栈向高地址增长，则传递的栈顶指针应该为 &Task1Stk [0]。

给 pdata 赋值，让它等于自身，是为了避免编译时出现警告"错误"——pdata 这个参数没有用到。换句话说，如果不加这一行代码，有些编译器就会报"错"，显示"WARNING — variable pdata not used."。

```
OS_STK  Task1Stk[1024];

void main (void)
{
      INT8U err;

      .
      OSInit();                    /* 初始化 μC/OS-II          */
      .
      OSTaskCreate(Task1,
                   (void *)0,
                   &Task1Stk[1023],
                   25);
      .
      OSStart();                   /* 启动多任务环境           */
}


void Task1 (void *pdata)           /* 防止编译时出现警告信息 */
{
      pdata = pdata;
      for (;;) {
          .                        /* 任务代码                 */
          .
      }
}
```

范例 2

可以创立一个通用的函数，多个任务可以共享一个通用的函数体，例如一个处理串行通信口的函数。传递不同的初始化数据（端口地址、波特率）和指定不同的通信口，就可以作为不同的任务运行。每个任务都有自己单独的栈空间和互不相同的优先级。本例中，COM1 在两个口中更为重要。

```
OS_STK      *Comm1Stk[1024];
COMM_DATA    Comm1Data;          /* 包含 COMM 口初始化数据的数据结构        */
                                 /* 通道 1 的数据 */
OS_STK      *Comm2Stk[1024];
COMM_DATA    Comm2Data;          /* 包含 COMM 口初始化数据的数据结构        */
                                 /* 通道 2 的数据                           */

void main (void)
{
   INT8U err;


   .
   OSInit();                     /* 初始化 μC/OS-II                         */
   .
   OSTaskCreate(CommTask,
                (void *)&Comm1Data,
                &Comm1Stk[1023],
                25);             /* 建立管理 COM1 的任务                    */
   OSTaskCreate(CommTask,
                (void *)&Comm2Data,
                &Comm2Stk[1023],
                26);             /* 建立管理 COM2 的任务                    */
   .
   OSStart();                    /* 启动多任务环境                          */
}


void CommTask (void *pdata)      /* 通信任务                                */
{
      for (;;) {
         .                       /* 任务代码                                */
         .
      }
}
```

OSTaskCreateExt()

```
INT8U OSTaskCreateExt(void     (*task)(void *pd),
                      void     *pdata,
                      OS_STK   *ptos,
                      INT8U    prio,
                      INT16U   id,
                      OS_STK   *pbos,
                      INT32U   stk_size,
                      void     *pext,
                      INT16U   opt);
```

所在章	所属文件	调用者	开关量
4	OS_TASK.C	任务或启动代码	无

OSTaskCreateExt()函数建立一个新的任务。与 OSTaskCreate()函数不同的是，OSTaskCreateExt()允许设置更多的细节内容。可以在多任务环境启动之前，或在正在运行的任务中建立任务，但中断处理程序中不能建立新任务。一个任务必须为无限循环结构，如以下代码所示，且不能返回。在初始化任务堆栈时，堆栈的结构必须与 CPU 中断后寄存器入栈的顺序结构相同，无论程序里中断是开着的还是关闭的。详细说明请参考所用处理器的手册。

注意：函数前 4 个参数的意义与 OSTaskCreate()的完全相同，为的是容易过渡到功能更强的函数 OSTaskCreateExt()。特别推荐使用 OSTaskCreateExt()，而不使用 OSTaskCreate()，因为 OSTaskCreateExt()要灵活得多。

```
void Task (void *pdata)
{
                          /* 对参数 pdata 进行操作，例如 pdata = pdata          */
    for (;;) {            /* 任务函数体，总是为无限循环结构                     */
        .
        .
        /* 任务中必须调用如下函数之一：
        /*     OSMboxPend()                                                     */
        /*     OSFlagPend()                                                     */
        /*     OSMutexPend()                                                    */
        /*     OSQPend()                                                        */
        /*     OSSemPend()                                                      */
        /*     OSTimeDly()                                                      */
        /*     OSTimeDlyHMSM()                                                  */
```

```
        /*    OSTaskSuspend()      (挂起任务自身)                    */
        /*    OSTaskDel()          (删除任务自身)                    */
        .
        .
    }
}
```

参 数

task　指向任务代码的指针。

pdata　指向一个类型可以选择的数据区，该数据区中的变量用于在建立任务时向任务传递参数。建立任务的任务或建立任务的初始化代码应该把 pdata 传给要建立的任务。pdata 用于向建立的任务传递参数。例如，可以建立一个通用的任务，以管理串行通信口，pdata 可以把诸如串行口的地址、波特率、每次传送几位、校验方式以及其他一些参数传递给这个任务。

ptos　指向任务堆栈栈顶的指针。任务堆栈用于保存局部变量、函数参数、返回地址以及中断时 CPU 寄存器的内容。

任务堆栈的大小决定于任务的需要及预计的中断嵌套层数。计算堆栈的大小，需要知道任务的局部变量所占的空间、可能产生嵌套调用的函数及中断嵌套所需的空间。

如果将初始化常量 OS_STK_GROWTH 设为 1，将堆栈设为向低端递减(从内存高地址向低地址递减)，那么此时 ptos 应指向任务堆栈空间的最高地址；反之，如果 OS_STK_GROWTH 设为 0，那么堆栈将从低地址向高地址递增。

prio　任务的优先级。每个任务必须有一个惟一的优先级作为标识。数字越小，优先级越高。

id　任务的标识。目前这个参数没有实际的用途，但保留在 OSTaskCreateExt()中，供今后扩展。应用程序中可设置 id 与优先级相同。

pbos　指向堆栈底端的指针。如果初始化常量 OS_STK_GROWTH 设为 1，那么堆栈被设为从内存高地址向低地址递减，此时 pbos 应指向任务堆栈空间的最低地址；反之，如果 OS_STK_GROWTH 设为 0，堆栈将从低地址向高地址递增，pbos 应指向堆栈空间的最高地址。参数 pbos 用于堆栈检测函数 OSTaskStkChk()。

stk_size　指定任务堆栈的大小。其单位由 OS_STK 定义：当 OS_STK 的类型定义为 INT8U，INT16U 及 INT32U 时，stk_size 的单位分别为字节(8 位)、字(16 位)及双字(32 位)。

pext　定义数据结构的指针，可作为 TCB 的扩展。例如，当任务切换时，定义的数据结构中可存放浮点寄存器的数值、任务运行时间及任务切入次数等信息。

opt　存放与任务相关的操作信息。opt 的低 8 位由 μC/OS-II 保留，用户不能使用。用户可以使用 opt 的高 8 位。每一种操作由 opt 中的 1 位或几位指定，当

相应的位被置位时，表示选择某种操作。当前的 μC/OS-II 版本支持下列操作：

OS_TASK_OPT_STK_CHK	决定是否进行任务堆栈检查。
OS_TASK_OPT_STK_CLR	决定是否清空堆栈。
OS_TASK_OPT_SAVE_FP	决定是否保存浮点寄存器的数值，此项操作仅当处理器有浮点硬件时有效。保存操作由硬件的相关代码完成。

其他选项请参考文件 uCOS_II.H。

返回值

OSTaskCreateExt()的返回值为下述之一：

OS_NO_ERR	函数调用成功。
OS_PRIO_EXIST	具有该优先级的任务已经存在。
OS_PRIO_INVALID	参数指定的优先级大于 OS_LOWEST_PRIO。
OS_NO_MORE_TCB	系统中没有 OS_TCB 可以分配给任务了。

注意/警告

1. 任务堆栈必须声明为 OS_STK 类型。
2. 在任务中必须进行 μC/OS-II 提供的下述过程之一：延时等待、任务挂起及等待事件发生(信号量、消息邮箱及消息队列)，以使其他任务得到 CPU。
3. 在用户程序中，不能够使用的优先级有：0，1，2，3，OS_LOWEST_PRIO-3，OS_LOWEST_PRIO-2，OS_LOWEST_PRIO-1 及 OS_LOWEST_PRIO。这些优先级被 μC/OS-II系统保留，其余 56 个优先级提供给应用程序。

范例 1

E1(1) 任务控制块使用了一个用户自定义的数据结构 TASK_USER_DATA，在其中保存了任务名称和其他一些数据。

E1(2) 任务名称可以用标准库函数 strcpy()初始化。

E1(3) 本例中设定堆栈是向低地址递减的，OS_STK_GROWTH 设为 1。程序注释中的 TOS(Top-Of-Stack)意为堆栈顶端，BOS(Bottom-Of Stack)意为堆栈底端。

E1(4) 注意：本例中堆栈检查操作是允许的，程序可以调用 OSTaskStkChk()函数。

```
typedef struct {                                    /* 用户定义的数据结构        */   (1)
    char      OS_TaskName[20];
    INT16U    OS_TaskCtr;
    INT16U    OS_TaskExecTime;
    INT32U    OS_TaskTotExecTime;
} OS_TASK_USER_DATA;
```

```
OS_STK              TaskStk[1024];
TASK_USER_DATA      TaskUserData;


void main (void)
{
        INT8U err;

        .
        OSInit();                                   /* 初始化 μC/OS-II          */
        .
        strcpy(TaskUserData.TaskName, "MyTaskName");  /* 任务名                  */   (2)
        err = OSTaskCreateExt(Task,
                (void *)0,
                &TaskStk[1023],                     /* 堆栈向低地址增长(TOS)    */   (3)

                10,
                &TaskStk[0],                        /* 堆栈向低地址增长(BOS)    */   (3)
                1024,
                (void *)&TaskUserData,              /* TCB 的扩展               */
                OS_TASK_OPT_STK_CHK);               /* 允许堆栈检查             */   (4)
        .
        OSStart();                                  /* 启动多任务环境           */
}


void Task (void *pdata)
{
        pdata = pdata;                              /* 此句可避免编译中的警告信息*/
        for (;;) {
                .                                   /* 任务代码                 */
                .
        }
}
```

范例 2

E2(1) 本例中创立的任务将运行在堆栈向高地址递增的处理器上，如 Intel 的 MCS-51。此时 OS_STK_GROWTH 设为 0。

E2(2) 在本例中，允许堆栈检查操作，程序可以调用 OSTaskStkChk()函数。程序注释中 TOS(Top-Of Stack)意为堆栈顶端，BOS(Bottom Of Stack)意为堆栈底端。

```
OS_STK *TaskStk[1024];


void main (void)
{
    INT8U err;

    .
    OSInit();                           /* 初始化 μC/OS-II                  */
    .
    err = OSTaskCreateExt(Task,
            (void *)0,
            &TaskStk[0],                /* 堆栈向高地址增长(TOS)            */      (1)
            10,
            10,
            &TaskStk[1023],             /* 堆栈向高地址增长(BOS)            */      (1)
            1024,
            (void *)0,
            OS_TASK_OPT_STK_CHK);       /* 允许堆栈检查                     */      (2)
    .
    OSStart();                          /* 启动多任务环境                   */
}


void Task (void *pdata)
{
    pdata = pdata;                      /* 此句可避免编译中出现警告信息 */
    for (;;) {
        .                               /* 任务代码                         */
        .
    }
}
```

OSTaskDel()

INT8U OSTaskDel(INT8U prio);

所在章	所属文件	调用者	开关量
4	OS_TASK.C	只能是任务	OS_TASK_DEL_EN

OSTaskDel()删除一个指定优先级的任务。任务可以传递自己的优先级给OSTaskDel()，从而删除自身。如果任务不知道自己的优先级，还可以传递参数OS_PRIO_SELF，被删除的任务将回到睡眠状态。任务被删除后，可以用函数OSTaskCreate()或OSTaskCreateExt()重新建立。

参 数

prio 指定要删除任务的优先级，也可以用参数OS_PRIO_SELF代替，此时，下一个优先级最高的就绪任务将开始运行。

返回值

OSTaskDel()的返回值为下述之一：

OS_NO_ERR	函数调用成功。
OS_TASK_DEL_IDLE	试图删除空闲任务(idle task)。
OS_TASK_DEL_ ERR	指定要删除的任务不存在。
OS_PRIO_INVALID	参数指定的优先级大于OS_LOWEST_PRIO。
OS_TASK_DEL_ISR	试图在中断处理程序中删除任务。

注意/警告

1. OSTaskDel()将判断是否试图删除μC/OS-II中的空闲任务(idle task)。
2. 在删除占用系统资源的任务时要小心，此时，为了安全起见，可以使用另一个函数OSTaskDelReq()。

范 例

```
void TaskX (void *pdata)
{
    INT8U err;


    for (;;) {
        .
        .
        err = OSTaskDel(10);            /* 删除优先级为 10 的任务 */
        if (err == OS_NO_ERR) {
                                        /* 任务被删除 */
            .
            .
        }
        .
        .
    }
}
```

OSTaskDelReq()

INT8U OSTaskDelReq(INT8U prio);

所在章	所属文件	调用者	开关量
4	OS_TASK.C	只能是任务	OS_TASK_DEL_EN

OSTaskDelReq()函数请求任务删除其自身。通常OSTaskDelReq()用于删除一个占有系统资源的任务(例如任务建立了信号量)。对于此类任务，在删除任务之前，应先释放任务占用的系统资源。具体做法是：在需要被删除的任务中调用OSTaskDelReq()，检测是否有其他任务的删除请求。如果有，则释放自身占用的资源，然后调用OSTaskDel()，删除自身。例如，假设任务5要删除任务10，而任务10占有系统资源，此时任务5不能直接调用OSTaskDel(10)删除任务10，而应该调用OSTaskDelReq(10)向任务10发送删除请求。在任务10中调用OSTaskDelReq(OS_PRIO_SELF)，并检测返回值。如果返回OS_TASK_DEL_REQ，则表明有来自其他任务的删除请求，此时任务10应该先释放资源，然后调用OSTaskDel(OS_PRIO_SELF)删除自己。任务5可以循环调用OSTaskDelReq(10)并检测返回值。如果返回OS_TASK_NOT_EXIST，则表明任务10已经成功删除。

参 数

prio 要求删除任务的优先级。如果参数为OS_PRIO_SELF，则表示调用函数的任务正在查询是否有来自其他任务的删除请求。

返回值

OSTaskDelReq()的返回值为下述之一：

OS_NO_ERR	删除请求已经被任务记录。
OS_TASK_NOT_EXIST	指定的任务不存在。发送删除请求的任务可以等待此返回值，看删除是否成功。
OS_TASK_DEL_IDLE	试图删除空闲任务。
OS_PRIO_INVALID	参数指定的优先级大于OS_LOWEST_PRIO，或没有设定OS_PRIO_SELF的值。
OS_TASK_DEL_REQ	当前任务收到来自其他任务的删除请求。

注意/警告

OSTaskDelReq()将判断用户是否试图删除μC/OS-II中的空闲任务。

范 例

```
void TaskThatDeletes (void *pdata)                          /* 任务优先级 5                    */
{
    INT8U err;


    for (;;) {
        .
        .
        err = OSTaskDelReq(10);                             /* 请求任务 10 删除自身            */
        if (err == OS_NO_ERR) {
            while (err != OS_TASK_NOT_EXIST) {
                err = OSTaskDelReq(10);
                OSTimeDly(1);                               /* 等待任务删除                    */
            }
            .                                               /* 任务 10 已被删除                */
        }
        .


    }
}


void TaskToBeDeleted (void *pdata)                          /* 任务优先级 10                   */
{
    .
    .
    pdata = pdata;
    for (;;) {
        OSTimeDly(1);
        if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ) {
            /* 释放任务占用的系统资源                                                          */
            /* 释放动态分配的内存                                                              */
            OSTaskDel(OS_PRIO_SELF);
        }
    }
}
```

OSTaskQuery()

INT8U OSTaskQuery(INT8U prio, OS_TCB *pdata);

所在章	所属文件	调用者	开关量
4	OS_TASK.C	任务或中断	无

OSTaskQuery()用于获取任务信息，函数返回任务TCB的一个完整的拷贝。应用程序必须建立一个OS_TCB类型的数据结构，容纳返回的数据。须提醒的是，在对任务OS_TCB对象中的数据进行操作时需小心，尤其是数据项OSTCBNext和OSTCBPrev。它们分别指向任务链表中OS_TCB的后一项和前一项。这个函数可提供有关内核的调试功能。

参 数

prio　指定要获取TCB内容的任务优先级，也可以指定参数OS_PRIO_SELF，获取调用任务的信息。

pdata　指向一个OS_TCB类型的数据结构，容纳返回的任务TCB的一个拷贝。

返回值

OSTaskQuery()的返回值为下述之一：

OS_NO_ERR　函数调用成功。

OS_PRIO_ERR　试图得到非法任务的信息。

OS_PRIO_INVALID　参数指定的优先级大于OS_LOWEST_PRIO。

注意/警告

任务控制块(TCB)中所包含的数据成员取决于下述开关量在初始化时的设定(参见OS_CFG.H)。

- OS_TASK_CREATE_EN；
- OS_Q_EN；
- OS_FLAG_EN；
- OS_MBOX_EN；
- OS_SEM_EN；
- OS_TASK_DEL_EN。

范 例

```
void Task (void *pdata)
{
      OS_TCB  task_data;
      INT8U   err;
      void   *pext;
      INT8U   status;


      pdata = pdata;
      for (;;) {
         .
         .
         err = OSTaskQuery(OS_PRIO_SELF, &task_data);
         if (err == OS_NO_ERR) {
            pext   = task_data.OSTCBExtPtr;     /* 获取 TCB 扩展数据结构的指针 */
            status = task_data.OSTCBStat;       /* 获取任务状态                */
            .
            .
         }
         .
         .
      }
}
```

OSTaskResume()

INT8U OSTaskResume(INT8U prio);

所在章	所属文件	调用者	开关量
1	OS_TASK.C	只能是任务	OS_TASK_SUSPEND_EN

OSTaskResume()唤醒一个用OSTaskSuspend()函数挂起的任务。OSTaskResume()也是惟一能"解挂"挂起任务的函数。

参 数

prio 指定要唤醒任务的优先级。

返回值

OSTaskResume()的返回值为下述之一：

OS_NO_ERR	函数调用成功。
OS_TASK_RESUME_PRIO	要唤醒的任务不存在。
OS_TASK_NOT_SUSPENDED	要唤醒的任务不在挂起状态。
OS_PRIO_INVALID	参数指定优先级大于或等于OS_LOWEST_PRIO。

注意/警告

无。

范 例

```
void TaskX (void *pdata)
{
    INT8U err;

    for (;;) {
        .
        .
        err = OSTaskResume(10);          /* 唤醒优先级为10的任务 */
        if (err == OS_NO_ERR) {
            .                            /* 任务被唤醒           */
            .
        }
        .
        .
    }
}
```

OSTaskStkChk()

INT8U OSTaskStkChk(INT8U prio, OS_STK_DATA *pdata);

所在章	所属文件	调用者	开关量
4	OS_TASK.C	只能是任务	OS_TASK_CREATE_EXT

OSTaskStkChk()检查任务堆栈状态，计算指定任务堆栈中的未用空间和已用空间。使用 OSTaskStkChk()函数要求所检查的任务是被 OSTaskCreateExt()函数建立的，且 opt 参数中 OS_TASK_OPT_STK_CHK 操作项打开。

计算堆栈未用空间的方法是，从堆栈底端向顶端逐个字节比较，检查堆栈中 0 的个数，直到一个非 0 的数值出现。这种方法的前提是，堆栈建立时已经全部清 0。要实现清 0 操作，需在任务建立初始化堆栈时，设置 OS_TASK_OPT_STK_CLR 为 1。如果应用程序在初始化时已将全部 RAM 清 0，且不进行任务删除操作，那么也可以设置 OS_TASK_OPT_STK_CLR 为 0，这将加快 OSTaskCreateExt()函数的执行速度。

参 数

prio 指定要获取堆栈信息的任务优先级，也可以指定参数 OS_PRIO_SELF，获取调用任务本身的信息。

pdata 指向一个类型为 OS_STK_DATA 的数据结构，其中包含如下信息：

```
INT32U OSFree;          /* 堆栈中未使用的字节数          */
INT32U OSUsed;          /* 堆栈中已使用的字节数          */
```

返回值

OSTaskStkChk()的返回值为下述之一：

OS_NO_ERR	函数调用成功。
OS_PRIO_INVALID	参数指定的优先级大于 OS_LOWEST_PRIO，或未指定 OS_PRIO_SELF。
OS_TASK_NOT_EXIST	指定的任务不存在。
OS_TASK_OPT_ERR	任务用 OSTaskCreateExt()函数建立时，没有指定 OS_TASK_OPT_STK_CHK 操作，或者任务是用函数 OSTaskCreate()建立的。

注意/警告

1. 函数的执行时间是由任务堆栈的大小决定的，事先不可预料。
2. 在应用程序中可以把 OS_STK_DATA 结构中的数据项 OSFree 和 OSUsed 相加，可得到堆栈的大小。
3. 原则上该函数可以在中断程序中调用，但由于该函数可能执行很长时间，所以实际中不提倡这种做法。

范 例

```
void Task (void *pdata)
{
    OS_STK_DATA stk_data;
    INT32U      stk_size;


    for (;;) {
        .
        .
        err = OSTaskStkChk(10, &stk_data);
        if (err == OS_NO_ERR) {
            stk_size = stk_data.OSFree + stk_data.OSUsed;
        }
        .
        .
    }
}
```

OSTaskSuspend()

INT8U OSTaskSuspend(INT8U prio);

所在章	所属文件	调用者	开关量
4	OS_TASK.C	只能是任务	OS_TASK_SUSPEND_EN

OSTaskSuspend()无条件地挂起一个任务。调用此函数的任务也可以传递参数OS_PRIO_SELF,挂起调用任务本身。当前任务挂起后,只有其他任务才能唤醒。在任务挂起后,系统会重新进行任务调度,运行下一个优先级最高的就绪任务。唤醒挂起任务必须调用函数OSTaskResume()。

任务的挂起是可以叠加到其他操作上的。例如,任务被挂起时正在进行延时操作,那么任务的唤醒就需要 2 个条件:延时的结束以及其他任务的唤醒操作。又如,任务被挂起时正在等待信号量,当任务从信号量的等待队列中被清除后,也不能立即运行,而必须等到唤醒操作后。

参 数

prio 指定要挂起的任务优先级,也可以指定参数 OS_PRIO_SELF,挂起任务本身。此时,下一个优先级最高的就绪任务将运行。

返回值

OSTaskSuspend()的返回值为下述之一:

OS_NO_ERR	函数调用成功。
OS_TASK_SUSPEND_IDLE	试图挂起 μC/OS-II 中的空闲任务。此为非法操作。
OS_PRIO_INVALID	参数指定的优先级大于 OS_LOWEST_PRIO,或没有设定 OS_PRIO_SELF 的值。
OS_TASK_SUSPEND_PRIO	要挂起的任务不存在。

注意/警告

1. 在程序中 OSTaskSuspend()和 OSTaskResume()应该成对使用。
2. 用 OSTaskSuspend()挂起的任务只能用 OSTaskResume()唤醒。

范 例

```
void TaskX (void *pdata)
{
    INT8U err;


    for (;;) {
        .
        .
        err = OSTaskSuspend(OS_PRIO_SELF); /* 挂起当前任务 */
        .                                  /* 当其他任务唤醒被挂起任务时,任务可继续运行 */
        .
        .
    }
}
```

OSTimeDly()

void OSTimeDly(INT16U ticks);

所在章	所属文件	调用者	开关量
5	OS_TIME.C	只能是任务	无

OSTimeDly()将一个任务延时若干个时钟节拍。如果延时时间大于 0,系统将立即进行任务调度。延时时间的长度可为 0～65 535 个时钟节拍。延时时间为 0,表示不进行延时,函数将立即返回调用者。延时的具体时间依赖于系统每秒有多少时钟节拍(由文件 OS_CFG.H 中的常量 OS_TICKS_PER_SEC 设定)。

参 数

ticks 要延时的时钟节拍数。

返回值

无。

注意/警告

1. 延时时间为 0,表示不进行延时操作,而立即返回调用者。
2. 为了确保设定的延时时间,建议设定的时钟节拍数加 1。例如,希望延时 10 个时钟节拍,可设定参数为 11。

范 例

```
void TaskX (void *pdata)
{
    for (;;) {
        .
        .
        OSTimeDly(10);                  /* 任务延时 10 个时钟节拍 */
        .
        .
    }
}
```

OSTimeDlyHMSM()

void OSTimeDlyHMSM(INT8U hours,INT8U minutes,INT8U seconds,INT8U milli);

所在章	所属文件	调用者	开关量
5	OS_TIME.C	只能是任务	无

OSTimeDlyHMSM()将一个任务延时若干时间。延时的单位是[小]时、分、秒及毫秒,所以使用OSTimeDlyHMSM()比使用OSTimeDly()更方便。调用OSTimeDlyHMSM()后,如果延时时间不为0,系统将立即进行任务调度。

参 数

hours 延时[小]时数,范围0~255。

minutes 延时分[钟]数,范围0~59。

seconds 延时秒数,范围0~59。

milli 延时毫秒数,范围0~999。

注意:延时操作函数都是以时钟节拍为单位的。实际的延时时间是时钟节拍的整数倍。例如系统每个时钟节拍间隔是10 ms,如果设定延时为5 ms,将不产生任何延时操作;而设定延时15 ms,实际的延时是2个时钟节拍,也就是20 ms。

返回值

OSTimeDlyHMSM()的返回值为下述之一:

OS_NO_ERR	函数调用成功。
OS_TIME_INVALID_MINUTES	参数错误,分[钟]数大于59。
OS_TIME_INVALID_SECONDS	参数错误,秒数大于59。
OS_TIME_INVALID_MILLI	参数错误,毫秒数大于999。
OS_TIME_ZERO_DLY	4个参数全为0。

注意/警告

OSTimeDlyHMSM(0,0,0,0)表示不进行延时操作,而立即返回调用者。另外,如果延时总时间超过65 535个时钟节拍,则不能用OSTimeDlyResume()函数终止延时并唤醒任务。

范 例

```
void TaskX (void *pdata)
{
    for (;;) {
        .
        .
        OSTimeDlyHMSM(0, 0, 1, 0);  /* 任务延时 1 s */
        .
        .
    }
}
```

OSTimeDlyResume()

INT8U OSTimeDlyResume(INT8U prio);

所在章	所属文件	调用者	开关量
5	OS_TIME.C	只能是任务	无

OSTimeDlyResume()唤醒一个用OSTimeDly()或OSTimeDlyHMSM()函数延时的任务。

参 数

prio 指定要唤醒任务的优先级。

返回值

OSTimeDlyResume()的返回值为下述之一：

OS_NO_ERR	函数调用成功。
OS_PRIO_INVALID	参数指定的优先级大于OS_LOWEST_PRIO。
OS_TIME_NOT_DLY	要唤醒的任务不在延时状态。
OS_TASK_NOT_EXIST	指定的任务不存在。

注意/警告

1. 不应该用OSTimeDlyResume()唤醒一个设置了等待超时操作并正在等待事件发生的任务，操作的结果将使该任务结束等待；除非的确希望这么做。
2. OSTimeDlyResume()不能唤醒一个用OSTimeDlyHMSM()延时且延时时间总计超过65 535个时钟节拍的任务。例如，若系统时钟为100 Hz，OSTimeDly-Resume()不能唤醒延时OSTimeDlyHMSM(0,10,55,350)或更长时间的任务。*

* OSTimeDlyHMSM(0,10,55,350)共延时[10 minutes × 60 +(55+0.35)seconds] × 100 =65 535个时钟节拍。——译者注

范 例

```
void TaskX (void *pdata)
{
    INT8U err;


    pdata = pdata;
    for (;;) {
        .
        err = OSTimeDlyResume(10);          /* 唤醒优先级为 10 的任务      */
        if (err == OS_NO_ERR) {
            .                               /* 任务被唤醒                  */
            .
        }
        .
    }
}
```

OSTimeGet()

INT32U OSTimeGet(void);

所在章	所属文件	调用者	开关量
5	OS_TIME.C	任务或中断	无

OSTimeGet()获取当前系统时钟数值。系统时钟是一个32位的计数器，记录系统上电后或时钟重新设置后的时钟数值。

参 数

无。

返回值

当前时钟数值(时钟节拍数)。

注意/警告

无。

范 例

```
void TaskX (void *pdata)
{
    INT32U clk;


    for (;;) {
        .
        .
        clk = OSTimeGet();  /* 获取当前系统时钟的值 */
        .
        .
    }
}
```

OSTimeSet()

void OSTimeSet(INT32U ticks);

所在章	所属文件	调用者	开关量
5	OS_TIME.C	任务或中断	无

OSTimeSet()设置当前系统时钟数值。系统时钟是一个 32 位的计数器，记录系统上电后或时钟重新设置后的时钟数值。

参 数

ticks 要设置的时钟数，单位是时钟节拍数。

返回值

无。

注意/警告

无。

范 例

```
void TaskX (void *pdata)
{
    for (;;) {
        .
        .
        .
        OSTimeSet(0L);    /* 复位系统时钟 */
        .
        .
    }
}
```

OSTimeTick()

void OSTimeTick(void);

所在章	所属文件	调用者	开关量
5	OS_TIME.C	任务或中断	无

每个时钟节拍，μC/OS-II 都将执行 OSTimeTick()函数。OSTimeTick()检查处于延时状态的任务是否达到延时时间(用 OSTimeDly()或 OSTimeDlyHMSM()函数延时)，或正在等待事件的任务是否超时。

参 数

无。

返回值

无。

注意/警告

OSTimeTick()的运行时间与系统中的任务数直接相关，在任务或中断中都可以调用。如果在任务中调用，则任务的优先级应该很高(优先级数值很小)，这是因为 OSTimeTick()负责所有任务的延时操作。

范 例

Intel 80x86，实模式，大模块下编译。

```
_OSTickISR PROC FAR
                PUSHA                                   ; 保存 CPU 寄存器内容
                PUSH  ES
                PUSH  DS
;
                MOV   AX, SEG(_OSIntNesting)            ; 恢复 DS
                MOV   DS, AX
                INC   BYTE PTR DS:_OSIntNesting         ; 通知 μC/OS-II,已进入中断处理程序
;
                CMP   BYTE PTR DS:_OSIntNesting, 1      ; 如果 (OSIntNesting == 1)
                JNE   SHORT _OSTickISR1
                MOV   AX, SEG(_OSTCBCur)                ; 恢复 DS
                MOV   DS, AX
                LES   BX, DWORD PTR DS:_OSTCBCur        ; OSTCBCur->OSTCBStkPtr = SS:SP
```

```
                MOV   ES:[BX+2], SS                ;
                MOV   ES:[BX+0], SP                ;
                CALL  FAR  PTR  _OSTimeTick        ; 调用时钟节拍处理函数
                .                                  ; 用户代码清除中断
                .
                CALL  FAR  PTR  _OSIntExit         ; 通知 μC/OS-II,即将退出中断处理程序

                POP   DS                           ; 恢复 CPU 寄存器内容
                POP   ES
                POPA
;
                IRET                               ; 返回到被中断了的任务
_OSTickISR      ENDP
```

OSVersion()

INT16U OSVersion(void);

所在章	所属文件	调用者	开关量
3	OS_CORE.C	任务或中断	无

OSVersion()获取当前 μC/OS-II 的版本号。

参 数

无。

返回值

当前版本,格式为 x.yy,返回值为乘以 100 后的数值。例如当前版本 2.52,则返回 252。

注意/警告

无。

范 例

```
void TaskX (void *pdata)
{
    INT16U os_version;


    for (;;) {
        .
        .
        os_version = OSVersion();  /* 获取 μC/OS-II 的版本号 */
        .
        .
    }
}
```

第 17 章

μC/OS－II 配置手册

本章介绍 μC/OS－II 中的初始化配置选项。由于 μC/OS－II 向用户提供源代码，初始化配置项由一系列 #define constants 语句构成，这些语句都放在文件 OS_CFG.H 中。用户的工程文件组中都应该包含这个文件。换言之，初始化配置是通过条件编译来实现的。

17.00 杂 项

本节介绍 OS_CFG.H 文件中每个用 #define constants 定义的常量。

OS_ARG_CHK_EN

OS_ARG_CHK_EN 用于设定是否希望大多数 μC/OS－II 中的函数执行参数检查的功能。如果该项设置为 1，μC/OS－II 将会保证传递给函数的指针变量不能为空，参数必须在允许范围之内等等。OS_ARG_CHK_EN 会尽力减少 μC/OS－II 所需要的代码量和处理时间。若必须将代码量降至最低，可设 OS_ARG_CHK_EN 为 0。但一般地说，应该允许参数检查，也就是说该项应当设为 1。

OS_CPU_HOOKS_EN

OS_CPU_HOOKS_EN 用于设定是否在 OS_CPU_C.C 中声明对外接口函数(hook function)。该项设置为 1，表明需要这些接口函数。μC/OS－II 中提供了 9 个对外接口函数，它们可在移植文件(OS_CPU_C.C)中声明，也可以在用户代码中声明，这些函数是：

OSInitHookBegin()	OSTaskStatHook()
OSInitHookEnd()	OSTaskSwHook()
OSTaskCreateHook()	OSTCBInitHook()
OSTaskDelHook()	OSTimeTickHook()
OSTaskIdleHook()	

OS_LOWEST_PRIO

OS_LOWEST_PRIO 用于设定系统中要使用的最低任务优先级(最大优先级数)。设定该项可以节省 μC/OS－II 所需要的 RAM 空间。μC/OS－II 中的优先级可以从 0(最高优先级)到 63(最低优先级)。设定 OS_LOWEST_PRIO 为一个小于 63 的数值，意味着程序中不能建立优先级数值大于 OS_LOWEST_PRIO 的任务。实际上，μC/OS－II 系统将保留 2 个优

先级自用：OS_LOWEST_PRIO 和 OS_LOWEST_PRIO - 1。其中 OS_LOWEST_PRIO 留给空闲任务(idle task)——OSTaskIdle()，而 OS_LOWEST_PRIO - 1 则留给统计任务——OSTaskStat()。任务的优先级可以为 0～OS_LOWEST_PRIO - 2。最低任务优先级 OS_LOWEST_PRIO和 OS_MAX_TASKS 之间没有什么关系。例如，可以设 OS_MAX_TASKS 为 10，而 OS_LOWEST_PRIO 为 32，此时系统最多可有 10 个任务，任务的优先级可以是 0～30 的数值。

但是需注意的是，每个任务必须具有不同于其他任务的优先级数值。所以，OS_LOWEST_PRIO设定的数值必须大于系统中的实际任务数。例如，设 OS_MAX_TASKS 为 20，而 OS_LOWEST_PRIO 为 10，这时用户最多可以创建 8 个任务(0，…，7)，因此必然造成 RAM 空间的浪费。

OS_MAX_EVENTS

OS_MAX_EVENTS 用于定义系统中可分配的事件控制块的最大数目。系统中的每一个消息邮箱、消息队列、互斥型信号量或者信号量都需要一个事件控制块。例如，系统中有 10 个消息邮箱、5 个消息队列、4 个互斥型信号量及 3 个信号量，则 OS_MAX_EVENTS 最小应该为 22。OS_MAX_EVENTS 的值必须大于 0，该项设置可参看 OS_MBOX_EN，OS_Q_EN，OS_MUTEX_EN 以及 OS_SEM_EN 等项。

OS_MAX_FLAGS

OS_MAX_FLAGS 用于定义用户程序中所需要的事件标志的最大数目，该项的数值必须大于 0。且必须设定 OS_FLAG_EN 的值为 1，才能使用事件标志组函数。

OS_MAX_MEM_PART

OS_MAX_MEM_PART 用于定义系统中内存块的最大数目，内存块将由内存管理函数操作(定义在文件 OS_MEM.C 中)。开关量 OS_MEM_EN 必须设定为 1，才能使用内存块。并且，OS_MAX_MEM_PART 必须设为大于 0 的数值，也就是说，可以只使用一个内存块。

OS_MAX_QS

OS_MAX_QS 用于定义系统中可以创建的消息队列的最大数目。开关量 OS_Q_EN 必须设定为 1，才能使用消息队列函数。并且，OS_MAX_QS 必须设为大于 0 的数值，也就是说，可以只使用一个消息队列。

OS_MAX_TASKS

OS_MAX_TASKS 用于定义用户程序中可以使用的最多任务数。需要注意的是，OS_MAX_TASKS的值不能大于 62。这是由于 μC/OS-II 保留了 2 个系统任务(见文件 uCOS_II.H 中的 OS_N_SYS_TASKS)。如果设定 OS_MAX_TASKS 刚好等于系统中的任务数，则增加新任务时要注意修改该项的值；但若将 OS_MAX_TASKS 设定为远远大于目前任务总数的值(便于日后扩展)，则会大大浪费宝贵的内存空间。如果对于产品来说，RAM 空间不存在问题，则应将 OS_MAX_TASKS 的值设为 62。

OS_TASK_IDLE_STK_SIZE

OS_TASK_IDLE_STK_SIZE 用于设定 μC/OS-II 中空闲任务堆栈的容量。注意堆栈容量的单位不是字节，而是 OS_STK*。空闲任务堆栈的容量取决于所使用的处理器，以及预期的最大中断嵌套数。虽然空闲任务几乎不做什么工作，但还是要预留足够的堆栈空间，保存 CPU 寄存器的内容以及处理可能出现的中断嵌套情况。

OS_TASK_STAT_EN

OS_TASK_STAT_EN 设定系统是否使用 μC/OS-II 中的统计任务(statistic task)及其初始化函数。如果设为 1，则使用统计任务 OSTaskStat()及其初始化函数。统计任务运行 1 次/秒，计算以百分数表示的当前系统的 CPU 使用率，结果保存在 8 位变量 OSCPUUsage 中。每次运行，OSTaskStat()都将调用 OSTaskStatHook()函数，自定义的统计功能可以放在这个函数中。详细情况请参考 OS_CORE.C 文件。统计任务 OSTaskStat()的优先级总是设为 OS_LOWEST_PRIO-1。

若不使用统计任务，则可将 OS_TASK_STAT_EN 设为 0，这时全局变量 OSCPUUsage，OSIdleCtrMax，OSIdleCtrRun 以及 OSStatRdy 都不声明，以节省内存空间。在每次清空 OSIdleCtr 之前，OSIdleCtrRun 都会保存一个 OSIdleCtr 的镜像。μC/OS-II 不会把 OSIdleCtrRun用于其他目的，但是，用户可以根据需要读取和显示 OSIdleCtrRun 的内容。

OS_TASK_STAT_STK_SIZE

OS_TASK_STAT_STK_SIZE 用于设置 μC/OS-II 中统计任务堆栈的容量。注意单位不是字节，而是 OS_STK。统计任务堆栈的容量取决于所使用的处理器类型，以及如下的操作：

- 进行 32 位算术运算(减法和除法)所需的堆栈空间；
- 调用 OSTimeDly()所需的堆栈空间；
- 调用 OSTaskStatHook()所需的堆栈空间；
- 预计最大的中断嵌套数。

如果想在统计任务中进行堆栈检查，并判断实际的堆栈使用量，那么，需要设定 OS_TASK_CREATE_EXT_EN为 1，并使用 OSTaskCreateExt()函数建立任务。

OS_SHED_LOCK_EN

该开关量用于控制是否使用 OSSchedLock() 和 OSSchedUnlock()函数。当设为 1 时，表示使用这 2 个函数。

OS_TICKS_PER_SEC

常量 OS_TICKS_PER_SEC 用于设定调用 OSTimeTick()函数的频率。用户需要在自己

---

* μC/OS-II 中堆栈统一用 OS_STK 声明，根据不同的硬件环境，OS_STK 可为不同的长度。——译者注

的初始化程序中保证 OSTimeTick()按所设定的频率* 调用。在 OSStatInit()，OSTaskStat()及 OSTimeDlyHMSM()函数中都会用到 OS_TICKS_PER_SEC。

17.01 事件标志

OS_FLAG_EN

OS_FLAG_EN 用于控制是否使用所有事件标志函数及其相关的数据结构。当设定为 1 时，表示使用；如果不需要使用事件标志，则可以设定该开关量为 0，以节省代码和数据空间。当 OS_FLAG_EN 设为 0 时，不需要再另外设定本节中的其他常量。

OS_FLAG_WAIT_CLR_EN

OS_FLAG_WAIT_CLR_EN 用于控制是否允许生成用于等待事件标志清 0 的代码。当设为 1 时表示允许。一般地说，用户希望等待至事件标志变为 1，但是，有时用户也可能希望等待至事件标志清 0，这时可通过此项进行设置。

OS_FLAG_ACCEPT_EN

OS_FLAG_ACCEPT_EN 控制是否使用 OSFlagAccept()函数。当设为 1 时，表示使用。

OS_FLAG_DEL_EN

OS_FLAG_DEL_EN 控制是否使用 OSFlagDel()函数。设为 1 时，表示使用。

OS_FLAG_QUERY_EN

OS_FLAG_QUERY_EN 控制是否使用 OSFlagQuery()函数。设为 1 时，表示使用。

17.02 消息邮箱

OS_MBOX_EN

OS_MBOX_EN 控制是否使用 μC/OS-II 中的消息邮箱函数及其相关的数据结构。设为 1，表示使用；如果不使用，则关闭此开关量，以节省内存。当 OS_MBOX_EN 设为 0 时，用户不需要再设定本节中其他用 #define constants 定义的常量。

OS_MBOX_ACCEPT_EN

OS_MBOX_ACCEPT_EN 控制是否使用 OSMboxAccept()函数。设为 1 时，表示使用。

---

* 系统硬件定时器中断发生的频率。——译者注

OS_MBOX_DEL_EN

OS_MBOX_DEL_EN 控制是否使用 OSMboxDel()函数。设为 1 时,表示使用。

OS_MBOX_POST_EN

OS_MBOX_POST_EN 控制是否使用 OSMboxPost()函数。设为 1 时,表示使用;若希望用功能更强的 OSMboxPostOpt()函数替代 OSMboxPost()函数,应将本开关量设为 0。

OS_MBOX_POST_OPT_EN

OS_MBOX_POST_OPT_EN 控制是否使用 OSMboxPostOpt()函数。设为 1 时,表示使用;若不需要函数 OSMboxPostOpt()所提供的附加功能,应将此开关量设为 0,从而使用 OSMboxPost()函数。使用后者将会生成较少的代码。

OS_MBOX_QUERY_EN

OS_MBOX_QUERY_EN 控制是否使用 OSMboxQuery()函数。设为 1 时,表示使用。

17.03 内存管理

OS_MEM_EN

OS_MEM_EN 控制是否使用 μC/OS II 中的内存块管理函数及其相关的数据结构。设为 1 时,表示使用;如果不使用,则将此开关量设为 0,以节省内存。

OS_MEM_QUERY_EN

OS_MEM_QUERY_EN 控制是否使用 OSMemQuery()函数。设为 1 时,表示使用。

17.04 互斥型信号量

OS_MUTEX_EN

OS_MUTEX_EN 控制是否使用互斥型信号量函数及其相关的数据结构。设为 1 时,表示使用;若不使用,则应将该开关量设为 0,以减少所需代码和数据空间。当 OS_MUTEX_EN 设为 0 时,不需要另外设置本节中其他用 #define constants 定义的常量。

OS_MUTEX_ACCEPT_EN

OS_MUTEX_ACCEPT_EN 控制是否使用 OSMutexAccept()函数。设为 1 时,表示使用。

OS_MUTEX_DEL_EN

OS_MUTEX_DEL_EN 控制是否使用 OSMutexDel()函数。设为 1 时，表示使用。

OS_MUTEX_QUERY_EN

OS_MUTEX_QUERY_EN 控制是否使用 OSMutexQuery()函数。设为 1 时，表示使用。

17.05 消息队列

OS_Q_EN

OS_Q_EN 控制是否使用 μC/OS-II 中的消息队列函数及其相关的数据结构。设为 1 时，表示使用；如果不使用，则将此开关量设为 0，以减少所需的代码空间。如果 OS_Q_EN 设为 0，则不需要另外设置本节中其他用 #define constants 定义的常量。另外值得注意的是，此时常量 OS_MAX_QS 无效。

OS_Q_ACCEPT_EN

OS_Q_ACCEPT_EN 控制是否使用 OSQAccept()函数。设为 1 时，表示使用。

OS_Q_DEL_EN

OS_Q_DEL_EN 控制是否使用 OSQDel()函数。设为 1 时，表示使用。

OS_Q_FLUSH_EN

OS_Q_FLUSH_EN 控制是否使用 OSQFlush()函数。设为 1 时，表示使用。

OS_Q_POST_EN

OS_Q_POST_EN 控制是否使用 OSQPost()函数。设为 1 时，表示使用；若希望使用功能更强的 OSQPostOpt()函数代替 OSQPost()函数，应将此开关量设为 0。

OS_Q_POST_FRONT_EN

OS_Q_POST_FRONT_EN 控制是否使用 OSQPostFront()函数。设为 1 时，表示使用；若希望使用功能更强的 OSQPostOpt()函数替代 OSQPostFront()函数，则应将此开关量设为 0。

OS_Q_POST_OPT_EN

OS_Q_POST_OPT_EN 控制是否使用 OSQPostOpt()函数。设为 1 时，表示使用；若不需要该函数所提供的附加功能，可将此开关量设为 0，从而使用 OSQPost()函数，以减少代码量。

OS_Q_QUERY_EN

OS_Q_QUERY_EN 控制是否使用 OSQQuery()函数。设为 1 时，表示使用。

17.06 信号量

OS_SEM_EN

OS_SEM_EN 控制是否使用 μC/OS-II 中的信号量管理函数及其相关的数据结构。设为 1，表示使用；如果程序不需要使用信号量，应将此开关量设为 0，以减少所需的代码和数据空间。当 OS_SEM_EN 设为 0 时，不需要另外设置本节中其他用 #define constants 定义的常量。

OS_SEM_ACCEPT_EN

OS_SEM_ACCEPT_EN 控制是否使用 OSSemAccept()函数。设为 1 时，表示使用。

OS_SEM_DEL_EN

OS_SEM_DEL_EN 控制是否使用 OSSemDel()函数。设为 1 时，表示使用。

OS_SEM_QUERY_EN

OS_SEM_QUERY_EN 控制是否使用 OSSemQuery()函数。设为 1 时，表示使用。

17.07 任务管理

OS_TASK_CHANGE_PRIO_EN

此开关量控制是否使用 μC/OS-II 中的 OSTaskChangePrio()函数。设为 1 时，表示使用；如果在应用程序中不需要改变运行任务的优先级，则应将此开关量设为 0，以减少 μC/OS-II所需的代码空间。

OS_TASK_CREATE_EN

此开关量控制是否使用 μC/OS-II 中的 OSTaskCreate()函数。设为 1 时，表示使用。使用该函数可以使 μC/OS-II 与 μC/OS 的建立任务函数向上兼容。在 μC/OS-II 中推荐使用 OSTaskCreateExt()函数建立任务。如果不使用 OSTaskCreate()函数，应将本开关量设为 0，以减少 μC/OS-II 所需的代码空间。

注意：OS_TASK_CREATE_EN 和 OS_TASK_CREATE_EXT_EN 至少有 1 个须为 1。当然如果都使用也可以。

OS_TASK_CREATE_EXT_EN

此开关量控制是否使用 μC/OS-II 中的 OSTaskCreateExt()函数。设为1时，表示使用。与 OSTaskCreate()函数相比，该函数为扩展的、功能更强的新建任务函数。如果程序不使用该函数，则应将 OS_TASK_CREATE_EXT_EN 设为0，以减少 μC/OS-II 所需的代码空间。

注意：如果要使用堆栈检查函数 OSTaskStkChk()，则必须用 OSTaskCreateExt()建立任务。

OS_TASK_DEL_EN

此开关量控制是否使用 μC/OS-II 中的 OSTaskDel()函数。设为1时，可以使用此函数删除任务。如果在应用程序中不使用此函数，则应将 OS_TASK_DEL_EN 设为0，以减少 μC/OS-II所需的代码空间。

OS_TASK_SUSPEND_EN

此开关量控制是否使用 μC/OS-II 中的 OSTaskSuspend()和 OSTaskResume()函数。设为1时，表示使用这2个函数挂起和唤醒任务。如果在应用程序中不使用这2个函数，应将 OS_TASK_SUSPEND_EN 设为0，以减少 μC/OS-II 所需的代码空间。

OS_TASK_QUERY_EN

OS_TASK_QUERY_EN 控制是否使用 OSTaskQuery()函数。设为1时，表示使用；若应用程序不使用本函数，则应将本开关量设为0，以减少 μC/OS-II 所需的代码空间。

17.08 时钟管理

OS_TIME_DLY_HMSM_EN

OS_TIME_DLY_HMSM_EN 控制是否使用 OSTimeDlyHMSM()函数。设为1时，表示可以使用该函数对一个任务进行一定时间的延时。这个时间是以[小]时、分、秒及毫秒为单位定义的。

OS_TIME_DLY_RESUME_EN

OS_TIME_DLY_RESUME_EN 控制是否使用 OSTimeDlyResume()函数。设为1时，表示使用。

OS_TIME_GET_SET_EN

OS_TIME_GET_SET_EN 控制是否使用 OSTimeGet()和 OSTimeSet()函数。设为1时，表示使用。若不需要使用32位的时钟计数器 OSTime，则可设定本开关量为0，以节省4B的数据和代码空间。

17.09 函数概述

表 T17.1 分类列出了 μC/OS-II 中的每个函数。"置 1"表示对某个开关量置 1，才可以使用该函数。另外还列出了其他影响函数配置的常量。

当然，为了使所需的配置生效，在编译 μC/OS-II 时，必须包含 OS_CFG.H 文件。

表 T17.1 μC/OS-II 函数和相关常量的配置用 #define constants 定义

表 T17.1 μC/OS-II 函数和相关常量的配置

类 型	置 1	其他常量
杂 项		
OSInit()	无	OS_MAX_EVENTS OS_Q_EN 和 OS_MAX_QS OS_MEM_EN OS_TASK_IDLE_STK_SIZE OS_TASK_STAT_EN OS_TASK_STAT_STK_SIZE
OSSchedLock()	OS_SCHED_LOCK_EN	无
OSSchedUnlock()	OS_SCHED_LOCK_EN	无
OSStart()	无	无
OSStatInit()	OS_TASK_STAT_EN 和 OS_TASK_CREATE_EXT_EN	OS_TICKS_PER_SEC
OSVersion()	无	无
中断处理		
OSIntEnter()	无	无
OSIntExit()	无	无
事件标志		
OSFlagAccept()	OS_FLAG_EN	OS_FLAG_ACCEPT_EN
OSFlagCreate()	OS_FLAG_EN	OS_MAX_FLAGS
OSFlagDel()	OS_FLAG_EN	OS_FLAG_DEL_EN
OSFlagPend()	OS_FLAG_EN	OS_FLAG_WAIT_CLR_EN
OSFlagPost()	OS_FLAG_EN	无
OSFlagQuery()	OS_FLAG_EN	OS_FLAG_QUERY_EN
消息邮箱		
OSMboxAccept()	OS_MBOX_EN	OS_MBOX_ACCEPT_EN
OSMboxCreate()	OS_MBOX_EN	OS_MAX_EVENTS
OSMboxDel()	OS_MBOX_EN	OS_MBOX_DEL_EN
OSMboxPend()	OS_MBOX_EN	无

续表 F17.1

类　型	置1	其他常量
OSMboxPost()	OS_MBOX_EN	OS_MBOX_POST_EN
OSMboxPostOpt()	OS_MBOX_EN	OS_MBOX_POST_OPT_EN
OSMboxQuery()	OS_MBOX_EN	OS_MBOX_QUERY_EN
内存块管理		
OSMemCreate()	OS_MEM_EN	OS_MAX_MEM_PART
OSMemGet()	OS_MEM_EN	无
OSMemPut()	OS_MEM_EN	无
OSMemQuery()	OS_MEM_EN	OS_MEM_QUERY_EN
互斥型信号量管理		
OSMutexAccept()	OS_MUTEX_EN	OS_MUTEX_ACCEPT_EN
OSMutexCreate()	OS_MUTEX_EN	OS_MAX_EVENTS
OSMutexDel()	OS_MUTEX_EN	OS_MUTEX_DEL_EN
OSMutexPend()	OS_MUTEX_EN	无
OSMutexPost()	OS_MUTEX_EN	无
OSMutexQuery()	OS_MUTEX_EN	OS_MUTEX_QUERY_EN
消息队列		
OSQAccept()	OS_Q_EN	OS_Q_ACCEPT_EN
OSQCreate()	OS_Q_EN	OS_MAX_EVENTS OS_MAX_QS
OSQDel()	OS_Q_EN	OS_Q_DEL_EN
OSQFlush()	OS_Q_EN	OS_Q_FLUSH_EN
OSQPend()	OS_Q_EN	无
OSQPost()	OS_Q_EN	OS_Q_POST_EN
OSQPostFront()	OS_Q_EN	OS_Q_POST_FRONT_EN
OSQPostOpt()	OS_Q_EN	OS_Q_POST_OPT_EN
OSQQuery()	OS_Q_EN	OS_Q_QUERY_EN
信号量管理		
OSSemAccept()	OS_SEM_EN	OS_SEM_ACCEPT_EN
OSSemCreate()	OS_SEM_EN	OS_MAX_EVENTS
OSSemDel()	OS_SEM_EN	OS_SEM_DEL_EN
OSSemPend()	OS_SEM_EN	无
OSSemPost()	OS_SEM_EN	无
OSSemQuery()	OS_SEM_EN	OS_SEM_QUERY_EN
任务管理		
OSTaskChangePrio()	OS_TASK_CHANGE_PRIO_EN	OS_LOWEST_PRIO
OSTaskCreate()	OS_TASK_CREATE_EN	OS_MAX_TASKS

续表 T17.1

类 型	置 1	其他常量
OSTaskCreateExt()	OS_TASK_CREATE_EXT_EN	OS_MAX_TASKS OS_TASK_STK_CLR
OSTaskDel()	OS_TASK_DEL_EN	OS_MAX_TASKS
OSTaskDelReq()	OS_TASK_DEL_EN	OS_MAX_TASKS
OSTaskResume()	OS_TASK_SUSPEND_EN	OS_MAX_TASKS
OSTaskStkChk()	OS_TASK_CREATE_EXT_EN	OS_MAX_TASKS
OSTaskSuspend()	OS_TASK_SUSPEND_EN	OS_MAX_TASKS
OSTaskQuery()	OS_TASK_QUERY_EN	OS_MAX_TASKS
时钟管理		
OSTimeDly()	无	无
OSTimeDlyHMSM()	OS_TIME_DLY_HMSM_EN	OS_TICKS_PER_SEC
OSTimeDlyResume()	OS_TIME_DLY_RESUME_EN	OS_MAX_TASKS
OSTimeGet()	OS_TIME_GET_SET_EN	无
OSTimeSet()	OS_TIME_GET_SET_EN	无
OSTimeTick()	无	无
用户定义函数		
OSTaskCreateHook()	OS_CPU_HOOKS_EN	无
OSTaskDelHook()	OS_CPU_HOOKS_EN	无
OSTaskStatHook()	OS_CPU_HOOKS_EN	无
OSTaskSwHook()	OS_CPU_HOOKS_EN	无
OSTimeTickHook()	OS_CPU_HOOKS_EN	无

第 18 章

PC 中的功能函数

本书中的代码是在 PC 机上测试的。

建立一系列的系统服务(功能函数)来发挥 PC 机的强大性能,会大大地方便开发者。这些服务都放在一个名为"PC.C"的文件中,并被测试代码所调用。

因为工业 PC 已被广泛地用做嵌入式系统的平台,故本章所提供的函数是很有用的。假定使用 DOS 操作系统,或是 Microsoft Windows 95,98,NT 及 2000 系统的 DOS 窗口,必须注意的是,在上述的 DOS 窗口环境中,是一个仿真的 DOS(虚拟的 x86 环境),而不是真正的 DOS 系统。由于这个原因,一些函数的结果可能会有变化。

可以在 \SOFTWARE\BLOCKS\PC\BC45 目录中找到 PC.C 和 PC.H,这些函数集成了 PC 的许多功能函数。这种集成使得用户可以很容易地把代码移植到其他编译器上。PC.C 文件主要包含了 3 类服务:字符显示、运行时间测量及一些其他服务。所有的这些函数都以 PC_ 为前缀。

18.00 字符显示

PC.C 提供了一些功能函数,可以在 PC 机的 VGA 显示器上显示 ASCII 字符和一些特殊字符。

在正常模式(字符模式)下,PC 机的显示器可以显示多至 2 000 个字符,分为 25 行(y 坐标)和 80 列(x 坐标),可参看图 F18.1。

图中屏幕高、宽比并非完全精确,显示器的实际高、宽比一般是 3×4。PC 机的显示缓冲区是映射到内存地址的,对于 VGA 显示器,显示缓冲区从绝对地址 0x000B8000 开始(或用段:偏移量表示为 B800:0000)。

每个可显示的字符需要 2 B 描述。第 1 个字节(内存地址的低位)是需要显示的字符代码;而第 2 个字节是该字符的属性,用来确定前景色和背景色。前景色用低 4 位表示,而背景色用第 4~6 位表示。最高位表示这个字符是否闪烁,1 表示闪烁,0 表示不闪烁。这 2 B 的内容可参看图 F18.2。

表 T18.1 是在 PC 机 VGA 字符模式下可以使用的色彩列表。其中 B7 表示第 7 位,其他同理。

需要注意的是,背景色只有 8 种选择,而前景色却有 16 种选择。在 PC.H 中包含了一些 #define 语句,可以定义常量,允许方便地选择前景色和背景色。这些 #define 语句定义的常量如表 T18.1 所列。例如,需要在黑色的背景上显示一个不闪烁的白色字符,只需简单地增

加 DISP_FGND_WHITE 和 DISP_BGND_BLACK（FGND 表示前景色，BGND 表示背景色）。这些设置对应于一个 16 进制的数值 0x0F*，正是 PC 机上显示字符的默认属性。应当注意到，DISP_BGND_BLACK 对应一个值 0x00，不需要专门指定。同样，对于字符为白色这一属性，也可以通过 DISP_ FGND_WHITE 指定。推荐使用 #define 定义的常数，而不使用 16 进制数值，以增强代码的可读性。

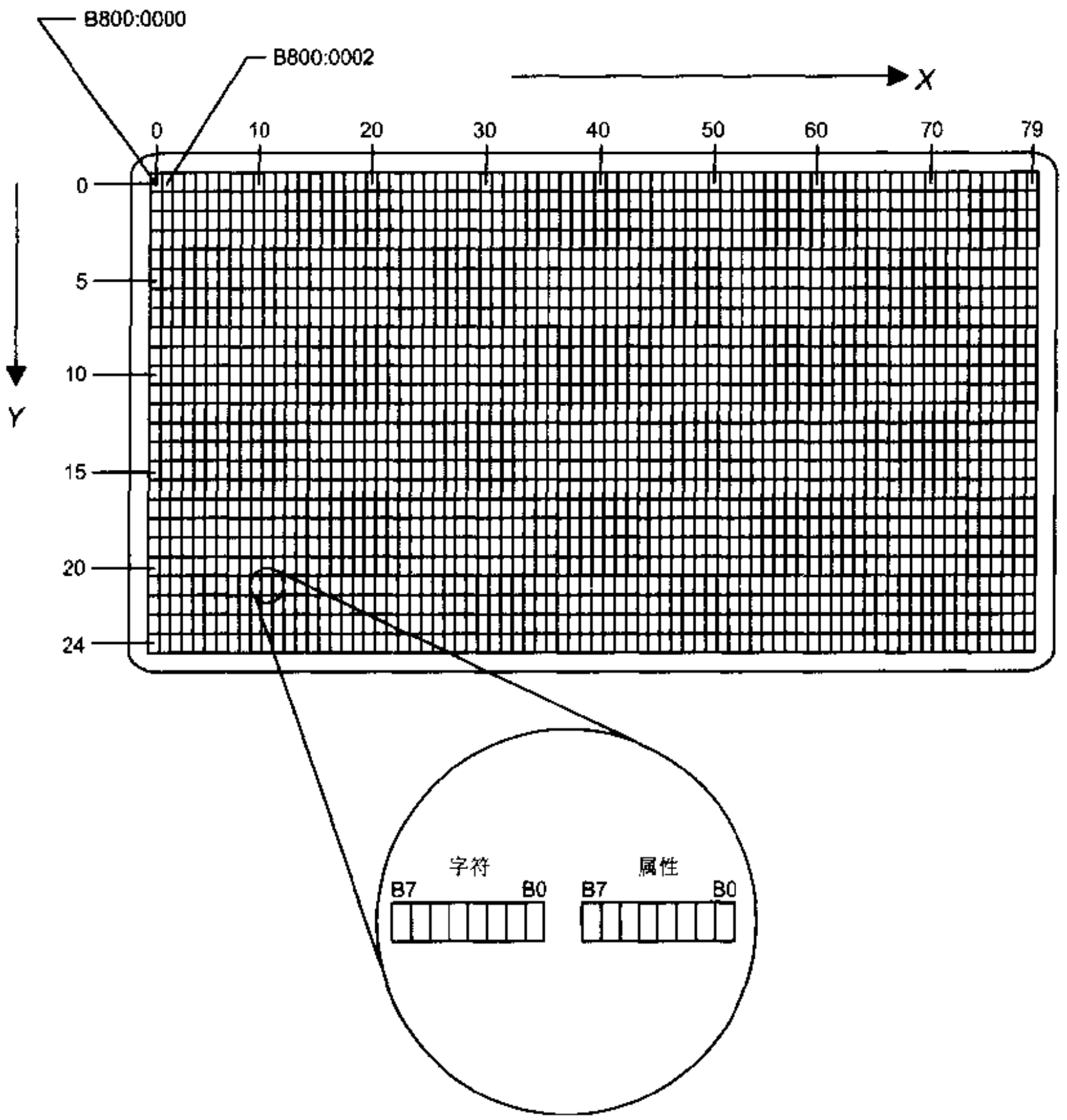


图 F18.1　在 VGA 显示器上显示 80×25 个字符

* 原文误为 0x07。——译者注

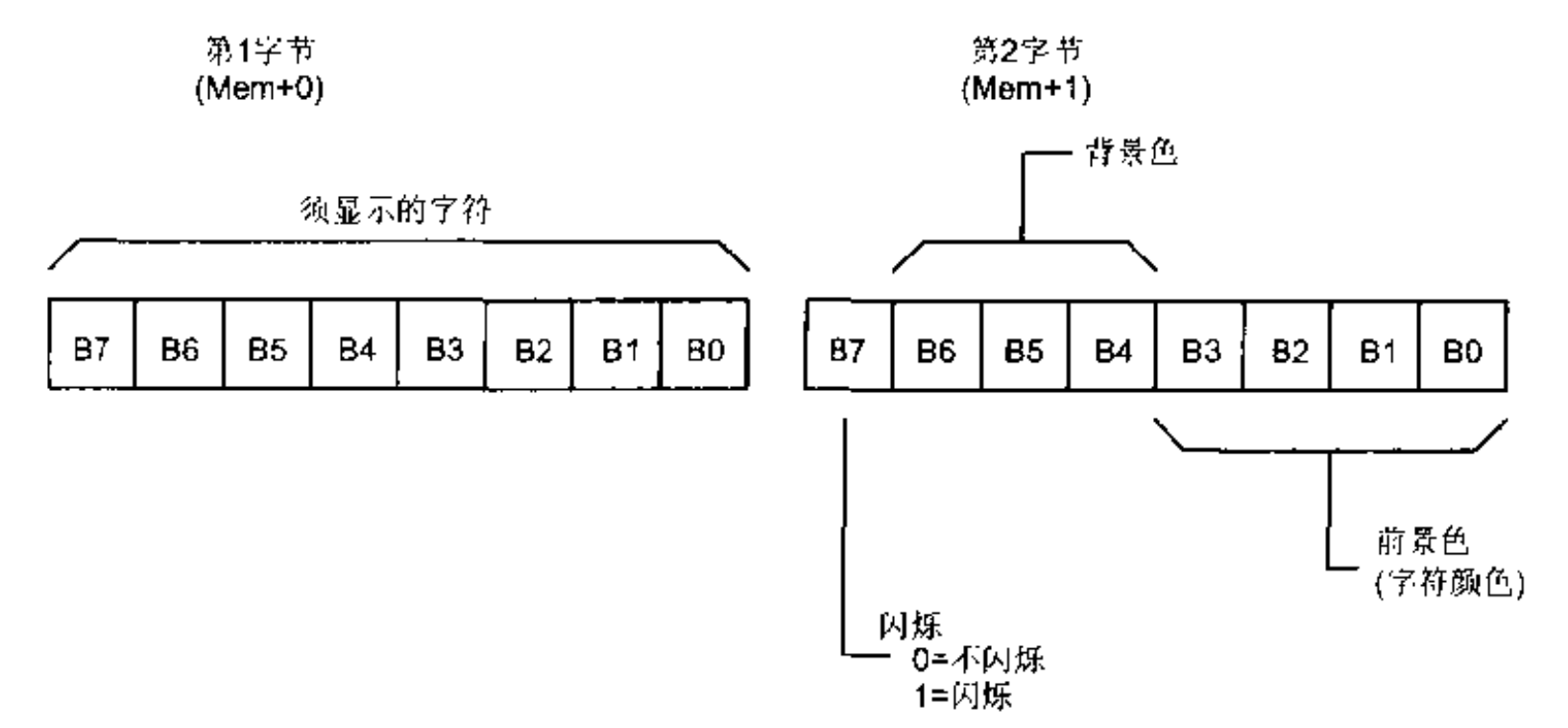


图 F18.2 VGA 显示器中字符和属性 2 B 的内容

表 T18.1 属性字节的值

闪烁(B7)		
是否闪烁?	#define 定义的常量	16 进制
否		0x00
是	DISP_BLINK	0x80
背景色(B6 B5 B4)		
色 彩	#define 定义的常量	16 进制
Black	DISP_BGND_BLACK	0x00
Blue	DISP_BGND_BLUE	0x10
Green	DISP_BGND_GREEN	0x20
Cyan	DISP_BGND_CYAN	0x30
Red	DISP_BGND_RED	0x40
Purple	DISP_BGND_PURPLE	0x50
Brown	DISP_BGND_BROWN	0x60
Light Gray	DISP_BGND_LIGHT_GRAY	0x70
前景色(B3 B2 B1 B0)		
色 彩	#define 定义的常量	16 进制
Black	DISP_FGND_BLACK	0x00
Blue	DISP_FGND_BLUE	0x01
Green	DISP_FGND_GREEN	0x02
Cyan	DISP_FGND_CYAN	0x03
Red	DISP_FGND_RED	0x04
Purple	DISP_FGND_PURPLE	0x05
Brown	DISP_FGND_BROWN	0x06
Light Gray	DISP_FGND_LIGHT_GRAY	0x07

续表 T18.1

前景色(B3 B2 B1 B0)		
色　彩	#define 定义的常量	16 进制
Dark Gray	DISP_FGND_DARK_GRAY	0x08
Light Blue	DISP_FGND_LIGHT_BLUE	0x09
Light Green	DISP_FGND_LIGHT_GREEN	0x0A
Light Cyan	DISP_FGND_LIGHT_CYAN	0x0B
Light Red	DISP_FGND_LIGHT_RED	0x0C
Light Purple	DISP_FGND_LIGHT_PURPLE	0x0D
Yellow	DISP_FGND_YELLOW	0x0E
White	DISP_FGND_WHITE	0x0F

使用 PC.C 文件中的显示函数，可以通过设定 x 和 y 坐标，在屏幕的任何地方写入 ASCII 码字符和其他特殊字符，所用的坐标系统的显示可见图 F18.1。需要注意的是，坐标原点(0,0)位于屏幕的左上角，这与我们的习惯(坐标在左下角)不同，但这可以简化所显示字符位置的计算过程。屏幕上每个字符在显示缓冲区中的位置通过下式给出：

$$字符位置 = 0x000B8000 + Y * 160 + X * 2$$

该字符的属性位于内存中的下一个字节，即：

$$属性字节位置 = 0x000B8000 + Y * 160 + X * 2 + 1$$

PC.C 中的显示函数直接对显示缓冲区执行写操作，虽然大部分 PC 的 BIOS 也可以完成同样的功能，但其可移植性较差。选择直接写显示缓冲区，是出于提高性能的考虑。

PC.C 文件包括以下 5 个函数，在本章接口函数一节中将对其进行更详细的叙述。

PC_DispChar()　　在屏幕任何位置上显示一个 ASCII 字符。

PC_DispClrCol()　　清一列。

PC_DispClrRow()　　清一行。

PC_DispClrScr()　　清屏。

PC_DispStr()　　在屏幕任何位置上显示一个 ASCII 字符串。

18.01　保存和恢复 DOS 的环境设置

函数 PC_DOSSaveReturn()(见程序清单 L18.1)用于保存当前 DOS 环境，主程序 main()通过调用该函数完成以下功能：

- 设置 μC/OS-II 的环境软中断向量，以实现任务切换。
- 设置时钟中断服务子程序 ISR 的中断向量。
- 保存 DOS 环境设置，使得在需要终止一个基于 μC/OS-II 的应用程序时，可以返回 DOS。

PC_DOSSaveReturn()函数的源代码如程序清单 L18.1 所列。

程序清单 L18.1 保存 DOS 环境设置

```
void PC_DOSSaveReturn (void)
{
    PC_ExitFlag  = FALSE;                                          (1)
    OSTickDOSCtr =     1;                                          (2)
    PC_TickISR   = PC_VectGet(VECT_TICK);                          (3)

    OS_ENTER_CRITICAL();
    PC_VectSet(VECT_DOS_CHAIN, PC_TickISR);                        (4)
    OS_EXIT_CRITICAL();

    setjmp(PC_JumpBuf);                                            (5)
    if (PC_ExitFlag == TRUE) {
        OS_ENTER_CRITICAL();
        PC_SetTickRate(18);                                        (6)
        PC_VectSet(VECT_TICK, PC_TickISR);                         (7)
        OS_EXIT_CRITICAL();
        PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);          (8)
        exit(0);                                                   (9)
    }
}
```

L18.1(1) PC_DOSSaveReturn()从设定 PC_ExitFlag 标志位 FALSE 开始，标志着现在不希望返回 DOS。

L18.1(2) 函数将 OSTickDOSCtr 初始化为 1，这个变量将在函数 OSTickISR()中递减。当其值为 0 时，下次递减将会翻转为 255。

L18.1(3)

L18.1(4) PC_DOSSaveReturn()把 DOS 中的时钟处理程序保存在一个空的中断向量入口处，这样 μC/OS-II 的时钟处理程序就可以调用它了（称为向量的转移）。

L18.1(5) PC_DOSSaveReturn()调用函数 setjmp()。这个函数将得到处理器的状态（各个主要寄存器的内容），并把它们保存在一个结构体 PC_JumpBuf 中。获取处理器的状态后，setjmp()将返回 PC_DOSSaveReturn()函数，然后从下一条语句起继续执行。因为 PC_ExitFlag 被初始化为 FALSE（参见 L18.1(1)），PC_DOSSaveReturn()将会跳过 if 语句，返回到主函数 main()。

L18.2(1)

L18.2(2) 若希望返回 DOS，只须调用 PC_DOSReturn()函数（见程序清单 L18.2）即可。在这个函数中，PC_ExitFlag 被设定为 TRUE，并且执行了跳转函数

longjmp()。

L18.1(5)　跳转函数 longjmp()使处理器返回到函数 PC_DOSSaveReturn()(调用 setjmp()函数的下一条语句)。

L18.1(6)　因为 PC_ExitFlag 为 TRUE,将执行 if 语句后的代码

L18.1(7)

L18.1(8)

L18.1(9)　PC_DOSSaveReturn()把时钟节拍改回 18.2 Hz,恢复 PC 原来的时钟中断服务子程序并清屏,通过 exit(0)函数返回 DOS 命令行状态。

程序清单 L18.2　返回 DOS

```
void PC_DOSReturn (void)
{
    PC_ExitFlag = TRUE;                                    (1)
    longjmp(PC_JumpBuf, 1);                                (2)
}
```

18.02　程序运行时间的测量

程序运行时间测量函数用于确定函数的运行时间,这是通过 PC 机的 82C54 定时器#2 完成的。在实际测量时,只需把被测量的代码放在 PC_ElapsedStart()和 PC_ElapsedStop()2 个函数之间即可;但在使用这 2 个函数之前,必须先调用初始化函数 PC_ElapsedInit(),以测量上面 2 个函数本身执行所需的时间。这样,PC_ElapsedStop()函数返回的时间(单位为 μs)就是执行测量代码所需的实际时间。

注意:上面 3 个函数都不具备可重入性,所以,必须小心:不得有多个任务同时调用这些函数。

18.03　其他函数

调用 PC_GetDateTime()函数,可得到 PC 中的当前日期和时间,并且以 ASCII 码的形式返回,格式是"YYYY-MM-DD　HH:MM:SS"。至少需要 21 个字符(包括 NULL 字符)存放这些数据。

注意:日期数据和时间数据有 2 个空格字符,因此才需要 21 个字符,而不是 20 个。

该函数使用了 Borland C/C++的库函数 gettime() 和 getdate() ,其他 DOS 环境下的 C 编译器也应该有类似的函数。

PC_GetKey()函数检查键盘是否有键被按下。如果有,就得到按键值并返回。该函数使用了 Borland C/C++的库函数 kbhit() 和 getch(),其他 DOS 环境下的 C 编译器也同样有类似函数。

PC_SetTickRate() 允许通过指定所需要的频率,改变时钟节拍的频率值。在 DOS 下,时

钟节拍产生 18.206 48 次/秒，或每隔 54.925 ms 产生 1 次。这是因为 82C54 芯片的计数器没有被初始化，而是使用默认值 65 535。如果芯片被初始化为 59 659，那么时钟节拍就会是精确的 20.000 Hz。笔者决定将时钟节拍设得更快一些，用的是大约 200 Hz(实际为199.9966 Hz)。

OS_CPU_A.ASM 文件中的代码将会每 11 个时钟节拍调用 1 次 DOS 的时钟节拍处理程序，这是为了保证 DOS 下必需的一些处理函数能够正确执行。如果用户将时钟节拍设为20 Hz，则不必这么做。在返回 DOS 以前，须调用 PC_SetTickRate()函数，并将频率设为18 Hz。

PC.C 中最后 2 个函数用于获取和设置中断向量，分别是 PC_VectGet() 和 PC_VectSet()。

这 2 个函数应该是与编译器无关的，只要编译器支持以下 3 个宏即可：

MK_FP()　　定义长指针。

FP_OFF()　　得到一个长指针的偏移量。

FP_SEG()　　得到一个长指针的段地址。

18.04 接口函数

本节可作为 PC 功能函数的参考手册。

PC_DispChar( )

void PC_DispChar(INT8U *x*, INT8U *y*, INT8U *c*, INT8U color)

PC_DispChar()允许在显示器的任何位置显示单个 ASCII 码字符或其他特殊字符。

参　数

x 和 *y*　指定所显示字符的坐标(列和行)。行的值为 0～DISP_MAX_Y - 1,而列的值为 0～DISP_MAX_X -1,可参看文件 PC.C。

c　需要显示的字符。可以指定任何 ASCII 码字符。若 *c* 的值大于 128,则可以指定任何其他特殊字符。

color　指定所显示字符的属性,即字符的色彩组合。可添加一个 DISP_FGND_???常数和一个 DISP_BGND_??? 常数,以得到所需要的色彩组合。这 2 个常数的定义见 PC.H 文件。

返回值

无。

注意/警告

无。

范　例

```
void Task (void *pdata)
  {
      .
    for (;;) {
        .
      PC_DispChar(0, 0,'$', DISP_FGND_WHITE);
        .
        .
    }
  }
```

PC_DispClrCol()

void PC_DispClrCol(INT8U x, INT8U color)

PC_DispClrCol() 允许清除一列的内容(一共25个字符)。

参 数

x　　指定清除哪一列，列值为0～DISP_MAX_X－1(见PC.C文件)的数。

color　指定字符属性字节的内容。因为用来清除一列的字符是空格字符"' '"，它只显示背景色，也可以通过定义DISP_BGND_???来指定任意一种颜色。

返回值

无。

注意/警告

无。

范 例

```
void Task (void *pdata)
{
    .
    .
    for (;;) {
        .
        PC_DispClrCol(0, DISP_BGND_BLACK);
        .
    }
}
```

PC_DispClrRow()

void PC_DispClrRow(INT8U y, INT8U color)

PC_DispClrRow() 允许清除一行的内容(共 80 个字符)。

参　数

y　　指定清除哪一行，行值为 0～DISP_MAX_Y－1(见 PC.C)的数。

color　指定字符属性字节的内容。因为用来清除一行的字符是空格字符“ ”，它只显示背景色，也可以通过定义 DISP_BGND_??? 来指定任意一种颜色。

返回值

无。

注意/警告

无。

范　例

```
void Task (void *pdata)
{
    .
    .
    for (;;) {
        .
        PC_DispClrRow(10, DISP_BGND_BLACK);
        .
        .
    }
}
```

PC_DispClrScr( )

void PC_DispClrScr(INT8U color)

PC_DispClrScr() 允许清空所有显示内容。

参 数

color 指定字符属性字节的内容。因为用来清屏的字符是空格字符“’ ’”，它只显示背景色，也可以通过定义 DISP_BGND_??? 来指定任意一种颜色。

返回值

无。

注意/警告

应当使用 DISP_FGND_WHITE 常数，而不要使用 DISP_BGND_BLACK 常数，因为这样前景色和背景色都为黑色，不是我们想要的。

范 例

```
void Task (void *pdata)
{
    .
    .
    PC_DispClrScr(DISP_FGND_WHITE);
    for (;;) {
        .
        .
    }
}
```

PC_DispStr()

void PC_DispStr(INT8U x, INT8U y, INT8U *s, INT8U color)

PC_DispStr() 用于显示一个 ASCII 码字符串。实际上，可以显示由 255 个 ASCII 字符所组成的任意字符串数组，只要该数组以 NULL 结束即可。

参 数

x 和 y　指定所显示字符串中第 1 个字符的坐标（列和行）。行的值为 0～DISP_MAX_Y－1，而列的值为 0～DISP_MAX_X－1，可参看文件 PC.C。

s　指向需要显示字符串的指针，字符串必须以 NULL 结束。可显示 0x00～0xFF 的任何字符。

color　指定所显示字符的属性，即字符的色彩组合。可添加一个 DISP_FGND_??? 常数和一个 DISP_BGND_??? 常数，以得到所需要的色彩组合。这 2 个常数的定义见 PC.H 文件。

返回值

无。

注意/警告

字符串数组中的所有字符都以相同的颜色属性显示。

范例 1

下面的代码用于显示全局变量 Temperature 的当前值。所用的颜色由温度决定：温度低于 100 为白色；温度高于 100 而低于 200 为黄色；温度高于 200 时，在红色背景下显示闪烁的白色字体。

```
FP32 Temperature;

void Task (void *pdata)
{
    char s[20];

    .
    .
    .
    PC_DispStr(0, 0,"Temperature:", DISP_FGND_YELLOW + DISP_BGND_BLUE);
    for (;;) {
        sprintf(s," %6.1f", Temperature);
```

```
        if (Temperature < 100.0) {
            color = DISP_FGND_WHITE;
        } else if (Temperature < 200.0) {
            color = DISP_FGND_YELLOW;
        } else {
            color = DISP_FGND_WHITE + DISP_BGND_RED + DISP_BLINK;
        }
        PC_DispStr(13, 0, s, color);
        .
        .
    }
}
```

范例 2

下面这段代码在屏幕中央显示一个10个字符宽度、7个字符高度的方框

```
INT8U  Box[7][11] = {
    {0xDA, 0xC4, 0xC4, 0xC4, 0xC4, 0xC4, 0xC4, 0xC4, 0xC4, 0xBF, 0x00},
    {0xB3, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0xB3, 0x00},
    {0xB3, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0xB3, 0x00},
    {0xB3, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0xB3, 0x00},
    {0xB3, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0xB3, 0x00},
    {0xB3, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0xB3, 0x00},
    {0xC0, 0xC4, 0xC4, 0xC4, 0xC4, 0xC4, 0xC4, 0xC4, 0xC4, 0xD9, 0x00}
};

void Task (void *pdata)
{
    INT8U i;
    .
    .
    for (i = 0; i < 7; i++) {
        PC_DispStr(35, i + 9, Box[i], DISP_FGND_WHITE);
    }
    for (;;) {
        .
        .
    }
}
```

PC_DOSReturn()

void PC_DOSReturn(void)

PC_DOSReturn() 让程序返回 DOS。前提是已经调用过 PC_DOSSaveReturn()保存重要的寄存器内容，以保证正常地返回 DOS。本函数的使用方法可参看第 1 章："初识 μC/OS－II"。

参　数

无。

返回值

无。

注意/警告

用户必须保证：在调用 PC_DOSReturn()函数前，已经调用过 PC_DOSSaveReturn()函数。

范　例

```
void Task (void *pdata)
{
    INT16U key;

    .
    .
    for (;;) {
        .
        .
        if (PC_GetKey(&key) == TRUE) {
            if (key == 0x1B) {
                PC_DOSReturn();                    /* 返回 DOS */
            }
        }
        .
        .
    }
}
```

PC_DOSSaveReturn()

void PC_DOSSaveReturn(void)

PC_DOSSaveReturn()允许程序在正式开始多任务前，保存重要的寄存器的值，以保证μC/OS-II能够正常地返回DOS。一般说来，应当在main()函数中调用本函数，正如下面范例代码中所列。

参 数

无。

返回值

无。

注意/警告

必须在设定μC/OS-II的环境切换中断向量之前调用本函数，如范例所列。

范 例

```
void  main (void)
{
    OSInit();                          /* 初始化 μC/OS-II                    */
    .
    PC_DOSSaveReturn();                /* 保存 DOS 的环境                    */
    .
    PC_VectSet(uCOS, OSCtxSw);         /* 用于 μC/OS-II 任务调试的软中断向量 */
    OSTaskCreate(...);
    .
    .
    OSStart();                         /* 开始多任务                         */
}
```

PC_ElapsedInit()

void PC_ElapsedInit(void)

PC_ElapsedInit()用于计算 PC_ElapsedStart()和 PC_ElapsedStop() 2 个函数本身运行所需的时间。只有这样，PC_ElapsedStop()函数才能返回所测代码的准确执行时间(单位为 μs)。

参 数

无。

返回值

无。

注意/警告

在调用 PC_ElapsedStart()或 PC_ElapsedStop()函数之前，必须调用 PC_ElapsedInit()函数。

范 例

```
void  main (void)
{
    OSInit();                          /* 初始化 μC/OS-II          */
    .
    .
    PC_ElapsedInit();                  /* 计算和测量额外的 CPU 负荷  */
    .
    .
    OSStart();                         /* 开始多任务               */
}
```

PC_ElapsedStart()

void PC_ElapsedStart(void)

PC_ElapsedStart() 与 PC_ElapsedStop()函数联合使用，可以测量程序代码的执行时间。

参 数

无。

返回值

无。

注意/警告

1. 在调用 PC_ElapsedStart()或 PC_ElapsedStop()之前，一定要调用 PC_ElapsedInit()函数。
2. 本函数不具备可重入性，不能在没有适当保护机制(信号量和调度锁定等)的情况下，由多个任务同时调用。
3. 用户代码的执行时间必须短于 54.93 ms，以保证时间测量函数能够正常工作。

范 例

```
void  main (void)
{
    OSInit();                          /* 初始化 μC/OS-II          */
    .
    .
    PC_ElapsedStart();                 /* 计算和测量额外的 CPU 负荷  */
    .
    .
    OSStart();                         /* 开始多任务               */
}

void Task (void *pdata)
{
    INT16U time_us;

    .
    .
```

```
    for (;;) {
        .
        .
        PC_ElapsedStart();
        /* 测量程序代码的执行时间 */
        time_us = PC_ElaspedStop();
        .
        .
    }
}
```

PC_ElapsedStop()

INT16U PC_ElapsedStop(void)

PC_ElapsedStop()与 PC_ElapsedStart()函数联合使用，用于测量程序代码的执行时间。

参 数

无。

返回值

以 μs 为单位返回 PC_ElapsedStart() 和 PC_ElapsedStop() 2 个函数之间的用户代码运行所需的时间。

注意/警告

1. 在调用 PC_ElapsedStart()或 PC_ElapsedStop()之前，一定要调用 PC_ElapsedInit()函数。
2. 本函数不具备可重入性，不能在没有适当保护机制(信号量和调度锁定等)的情况下，由多个任务同时调用。
3. 用户代码的执行时间必须短于 54.93 ms，以保证时间测量函数能够正常工作。

范 例

参见前面 PC_ElapsedStart()函数。

PC_GetDateTime( )

void PC_GetDateTime(char *s)

PC_GetDateTime() 用于从 PC 的实时时钟芯片中获取当前的日期和时间，并将这些信息以长度不少于 21 个字符的 ASCII 码字符串形式返回。

参　数

s　指向 ASCII 字符串保存位置的指针。ASCII 字符串的格式是：

YYYY-MM-DD　HH:MM:SS

需要 21 个字节保存。

注意：日期和时间之间有 2 个空格。

返回值

无。

注意/警告

无。

范例

```
void Task (void *pdata)
{
    char s[80];
    .
    .
    for (;;) {
        .
        .
        PC_GetDateTime(&s[0]);
        PC_DispStr(0, 24, s, DISP_FGND_WHITE);
        .
        .
    }
}
```

PC_GetKey()

BOOLEAN PC_GetKey(INT16S *key)

PC_GetKey() 用于检查键盘是否有键被按下。如果有，就得到按键值并返回。通常用户需要频繁地调用本函数，以检测是否有按键被按下，也就是对键盘进行轮询。

注意：PC 实际上通过中断服务子程序得到按键值，并放到缓冲区中，最多可缓存 10 个按键值。

参 数

key 指向按键值存放地址的指针。如果没有按键被按下，则该指针值为 0x0000。

返回值

如果按键被按下，则返回 TRUE；否则返回 FALSE。

注意/警告

无。

范 例

```
void Task (void *pdata)
{
    INT16S   key;
    BOOLEAN  avail;

    .
    .
    for (;;) {
        .
        .
        avail = PC_GetKey(&key);
        if (avail == TRUE) {
            /* 有键被按下 */
        }
        .
        .
    }
}
```

PC_SetTickRate( )

void PC_SetTickRate(INT16U freq)

PC_SetTickRate( ) 用于将 PC 的时钟节拍从标准值 18.206 48 Hz 改变成更快。一个 200 Hz的时钟节拍就是 18.206 48 Hz 的 11 倍。

参　数

freq　所需要的时钟频率。

返回值

无。

注意/警告

1. 只可将时钟节拍修改为大于 18.206 48 Hz 的值。
2. 频率越快,CPU 的额外开销就越大。

范　例

```
void  Task (void *pdata)
{
    .
    .
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(400);        /* 改变时钟节拍到 400 Hz */
    OS_EXIT_CRITICAL();
    .
    .
    for (;;) {
        .
        .
    }
}
```

PC_VectGet()

void *PC_VectGet(INT8U vect)

PC_VectGet() 用于获取由中断向量值 vect 指定的中断服务子程序的地址。80x86 处理器可支持多达 256 个中断/异常处理程序。

参 数

vect 中断向量值,是一个介于 0～255 的数值。

返回值

由中断向量值指定的中断/异常处理程序的地址。

注意/警告

1. 中断向量值 0 对应于复位处理程序。
2. 假定 80x86 的代码在编译时选择大模式,即所有的返回指针都是长指针。
3. 假定 80x86 运行在实时模式下。

范 例

```
void  Task (void *pdata)
{
    void ( *p_tick_isr)(void);



    .
    .
    p_tick_isr = PC_VectGet(0x08);  /* 时钟节拍中断处理子程序的入口地址 */
    .
    .
    for (;;) {
        .
        .
    }
}
```

PC_VectSet()

void PC_VectSet(INT8U vect, void *(pisr)(void))

PC_VectSet() 用于设定中断向量表的内容。80x86 处理器可支持多达 256 个中断/异常处理程序。

参 数

vect 中断向量值,是一个介于 0～255 的数值。

pisr 中断/异常处理程序的地址。

返回值

无。

注意/警告

1. 设定中断向量时必须小心,因为有些中断向量是被操作系统(DOS 和/或 μC/OS-II)使用的。
2. 假定 80x86 的代码在编译时选择大模式,即所有的返回指针都是长指针。

范 例

```
void  InterruptHandler (void)
{
}


void  Task (void *pdata)
{
    .
    .
    PC_VectSet(64, InterruptHandler);
    .
    .
    for (;;) {
        .
        .
    }
}
```

18.05 参考文献

1 Chappell Geoff. DOS Internals. Reading, Massachusetts: Addison—Wesley, 1994

2 Tischer Michael. PC Internals, System Programming 5th Edition. Grand Rapids, Michigan: Abacus, 1995

3 Villani Pat. FreeDOS Kernel. Lawrence, Kansas: CMP Books, 1996

附录 A

C 语言编程规约

编程规约应建立于一项工程之初，且该规约应贯穿整个工程之始终。采用一些规约有助于提高劳动生产率，并使工程软件便于维护。

用 C 语言或其他语言编写程序的方式有很多种，惯用的、好的书写风格以及他人用的一些好方法，无非是想力图达到以下目的：

- 可移植性好；
- 兼容性好；
- 干净整齐；
- 易维护性好；
- 易看懂；
- 简明扼要。

无论采用什么样的编程风格，必须在整个工程中采用始终如一的格式。对于一项大工程的整个团队，建议使用统一的编程风格。在此，推荐以下 C 语言编程规约，可用于规范程序的书写方法。采用共同的编写风格，可以少发生一些程序维护中令人头痛的事情，降低维护成本；采用统一的书写方法，有助于避免代码的重复编写。这里展示的 C 语言程序的编程风格，主要强调如何使源码便于阅读与维护。

笔者不喜欢把 C 源代码的宽度限制在 80 个字符。限制只是基于 215.9 mm×279.4 mm (8.5 in×11 in)的纸上，使用字符宽度为 8 点阵的定宽度字符集时，一行可以打印出多少个字符。实际上，使用 8 点阵宽度的字符集，一行可以容纳 132 个字符，左边还有足够的地方供打孔用，以便夹到 3 眼文件夹中。使用每行 132 字符的格式，可避免源代码与注解分成 2 行。

A.01 文件头

以下是 C 语言程序的文件头(header)。公司的名称和地址可写在前几行；然后是标题，标出文件的主题内容；接着是著作权声明，以标明软件的专有权。

然后是文件名和程序员的姓名。第 1 个人的名字是建立这个文件的编程员的名字。文件头中的最后一项是对文件内容的描述。

当需要分页时，笔者喜欢使用分页符。实际上，笔者喜欢按逻辑关系分页，譬如，若在一页中一个注解块后的版面放不下一整段代码，就宁可分页。由于历史的原因，笔者在特别注解"/* $PAGE */"后面插入了换页符 (0x0C)。因为"/* PAGE */"告知，这里换页了，故笔者喜欢使用它。

文件头

```
/*
*********************************************************************************
*                                    Company Name
*                                      Address
*
*                        (c) Copyright 20xx, Company Name, City, State
*                                  All Rights Reserved
*
*
* Filename      :
* Programmer(s) :
* Description   :
*********************************************************************************
*/

/* $PAGE */
```

A.02 文件包含

应用工程需要的头文件紧跟在版本变更记录的后面。可以只列出本模块需要的头文件，也可以用一个名为INCLUDE.H的头文件将所有的头文件包含进去。使用INCLUDE.H文件，就不必记住哪个头文件是给哪个源文件用的，特别是当须增加新的模块时。这样做惟一的缺点是，编译每个模块时所需的时间要长一些。

文件包含

```
/*
*********************************************************************************
*                                   INCLUDE FILES
*********************************************************************************
*/

#include "INCLUDES.H"

/* $PAGE */
```

A.03 标志符命名法

符合 ANSI C 标准的 C 编译器(现在大多数 C 编译器都如此),允许使用 32 个字符命名标志符。标志符可以是变量、结构/联合的成员、函数、宏及用 #difine 语句定义的对象等。利用可以使用 32 个字符命名标志符这一特点,可以使用缩写、略语及助记符来描述标志符(见 A.04,"缩写、略语及助记符")。标志符的名字应该反映该标志符的用途。笔者喜欢使用划分级别的方法命名标志符。例如,函数 OSSemPend() 表示:它是操作系统的一部分(OS),它是一个信号量(Semaphore),它的功能是等待(Pend)信号量。这种方法能够将所有关于信号量的函数归纳为一组。读者可能已经注意到,μC/OS-II 中的有些函数是以 OS_开头的,而不是 OS,这表示以 OS_开头的函数是 μC/OS-II 的内部函数,尽管这些函数是全局函数。

变量名的声明应该是每行一个变量,而不要在一行中声明多个变量,这样易于给每一个变量加上注释。

用文件名作为变量名的前缀,不论是局部变量还是全局变量。这样,该变量是全局地、还是局部地用在哪个文件中就很清楚。例如,名为 KEY.C 的文件中的局部和全局变量可用以下方法命名。

变量的命名法

```
static  INT16U  KeyCharCnt;          /* 按下的键的数目                  */
static  char    KeyInBuf[100];       /* 保存字符的缓冲区(Buffer)        */
        char    KeyInChar;           /* 字符类型                        */

/* $PAGE */
```

大写字母用于将标志符中的不同词分开。使用这种方法而不使用下划线(_),是因为下划线不会给名字增加任何含义,还占用了一个字符的位置。

可以任意给全局变量(对文件来说是外部变量)起名字,只需大写和小写字母混用,并冠以模块或文件名的前缀(所有全局的、与键盘(keyboard)相关的变量名都冠以 Key 这个词)。

函数的形式参数和函数内部的局部变量用小写字母声明。小写字母使该变量一看就知道是某函数的一个局部变量,因为全局变量是大写字母和小写字母混用的。为了使变量名便于阅读,可以使用下划线(_)分开各词。

在函数内部,可以使用一些总是具有同样含义的变量名来命名局部变量。以下是一些例子,当然也可以用其他一些名字,只须保持前后一致。

i, j, k	用做循环次数计数器。
p1, p2, …, pn	用做指针。
c, c1, …, cn	用做字符。
s, s1, …, sn	用做字符串。
ix, iy, iz	用做中间整数变量。

fx，fy，fz　　用做中间浮点数变量。

现总结如下：

- 形式参数：在函数中应只用小写字母。
- 自动变量名：应只用小写字母。
- 静态变量和函数：应使用文件名或模块名（或部分文件名、模块名）为前缀，大小写字母混用。
- 外部变量和函数：应使用文件名或模块名（或部分文件名、模块名）为前缀，大小写字母混用。

A.04 缩写、略语及助记符

在给变量和函数等标志符命名时，可以使用缩写（如OS、ISR及TCB等）、略语（如buf、doc等）及助记符（如clr、cmp等）。使用缩写、略语及助记符可以用较少的字符描述标志符；但是，如果缩写、略语及助记符的使用前后不一致，反而会更多地造成混乱。为了保证前后一致，笔者为自己所有工程项目中用到的缩写、略语及助记符造了一张表。称之为缩写、略语及助记符词典。只要列入了表中，就始终如一地使用同样的缩写、略语及助记符。为μC/OS-II列出的表如表TA.01所列。当需要其他缩写、略语及助记符时，就把它们加到表中。

表TA.1　本书中用到的缩写、略语及助记符

缩写、略语及助记符	含　义
Addr	Address
Blk	Block
Chk	Check
Clr	Clear
Cnt	Count
CPU	Central Processing Unit
Ctr	Counter
Ctx	Context
Cur	Current
Del	Delete
Dly	Delay
Err	Error
Ext	Extension
FP	Floating Point
Grp	Group
HMSM	Hours Minutes Seconds Milliseconds

续表 1A.1

缩写、略语及助记符	含 义
ID	Identifier
Init	Initialize
Int	Interrupt
ISR	Interrupt Service Routine
Max	Maximum
Mbox	Mailbox
Mem	Memory
Msg	Message
N	Number of
Opt	Option
OS	Operating System
Ovf	Overflow
Prio	Priority
Ptr	Pointer
Q	Queue
Rdy	Ready
Req	Request
Sched	Scheduler
Sem	Semaphore
Stat	Status or Statistic
Stk	Stack
Sw	Switch
Sys	System
Tbl	Table
TCB	Task Control Block
TO	Timeout

也许有时所有的产品都用一张表并不合适。若一家工程公司，为很多不同的客户开发类型完全不同的产品，每个项目都有各自不同的表会更好。农业用的词汇与军工用的词汇当然是不一样的。原则是，类似的产品用相同的词典。

使用同一本词典对于一个项目团队来说可以提高劳动生产率。不仅是在单独的程序中，在整个工程项目的始终都应保持前后一致，这一点特别重要。既然 buf 的含义已经被定义为 buffer，那么所有工程项目团队的人就都应这样用，而不能一部分人用 buffer，而另一部分人用 bfr。深化这一概念，即使标志符允许使用全名，也只能用缩写 buf，尽管可以完整地写出buffer 这个词

A.05 注 解

如果程序代码与注解混在一起，就很难阅读；因此，不混和编写代码和注解，注解写在C代码的右边。当需要将注解写得较长时，以函数描述注解块的形式来写。

在以下的例子中，注解块占用了一些整行。注解结束符"*/"本来不需要占用一行，但是为了整齐，也让它占用一行。程序中无需每行都加注解。

注解范例

```
/*
*********************************************************************************
*                                    atoi()
*
* 功能描述：本函数将字符串"s"转换成整数
* 参   数  ：要转换成整数的 ASCII 码字符串
*           （所有字符串中的字符都必须是 10 进制数(0～9)）
* 返回值   ：由字符串转换成的"int"
*********************************************************************************
*/

int atoi (char *s)
{
    int n;                                  /* 转换的结果                    */


    n = 0;                                  /* 初始化该结果                  */
    while ( *s >= '0' && *s <= '9' && *s) {  /* 所有有效字符，且非字符串结束  */
        n = 10 * n + *s - '0';              /* 将 char 转换成 int 并加到结果上 */
        s++;                                /* 下一个要转换的 char           */
    }
    return (n);                             /* 返回字符串转换结果            */
}


/* $PAGE */
```

A.06 定义语句

头文件（.H）和C的源文件（.C）常需要定义一些常数和宏定义。常数和宏定义永远要用大写，相应的词用下划线分开。

注意：16进制数总是以小写字母x开头，16进制中的A～F都要用大写，常数名和其数值要占一行。

常数和宏的定义

```
/*
*****************************************************************************
*                                CONSTANTS & MACROS
*****************************************************************************
*/

#define  KEY_FF              0x0F
#define  KEY_CR              0x0D
#define  KEY_BUF_FULL()      (KeyNRd > 0)

/* $PAGE */
```

A.07 数据类型

C语言允许用typedef关键字定义新的数据类型。笔者用大写字母定义了所有的数据类型，与定义常数及宏定义用的是同一规则。因为常数、宏及数据类型可以在上下文中区分，没有发生过标志符之间的混淆。因为不同的处理器有不同的字长，笔者声明了以下一些数据类型(使用的是Boland C++ V4.51)。

数据类型

```
/*
*****************************************************************************
*                                DATA TYPES
*****************************************************************************
*/

typedef  unsigned char    BOOLEAN;          /* Boolean              */
typedef  unsigned char    INT8U;            /* 8-bit unsigned       */
typedef  char             INT8S;            /* 8-bit signed         */
typedef  unsigned int     INT16U;           /* 16-bit 无符号的      */
typedef  int              INT16S;           /* 16-bit 带符号的      */
typedef  unsigned long    INT32U;           /* 32-bit 无符号的      */
typedef  long             INT32S;           /* 32-bit 带符号的      */
typedef  float            FP;               /* 浮点数               */

/* $PAGE */
```

用这些定义，读者很容易知道每种数据类型的字长。

A.08 局部变量

一些源程序模块须用到局部变量，这些变量仅用于这个源文件（文件作用域）。应将这些局部变量隐藏起来，使其他文件看不到。在C语言中隐藏这些变量是用static关键字实现的。变量可以按字母顺序排列，也可以按功能排列。

局部变量

```
/*
*********************************************************************************
*                                    LOCAL VARIABLES
*********************************************************************************
*/

static  char      KeyBuf[100];
static  INT16S    KeyNRd;

/* $PAGE */
```

A.09 函数原型

本节介绍在文件中声明函数原型（或调用规则）。函数原型声明的顺序应与函数定义的顺序一致。这一顺序可以使用户能从打印出的文件中很快地找到相应的函数。

注意：关键字static、返回值的类型及函数名都应上下对齐。

函数原型

```
/*
*********************************************************************************
*                                  FUNCTION PROTOTYPES
*********************************************************************************
*/

        void        KeyClrBuf(void);
static  BOOLEAN     KeyChkStat(void);
static  INT16S      KeyGetCnt(int ch);

/* $PAGE */
```

A.10 函数声明

从打印机中打印出的程序代码清单中，应尽可能地让每页纸只有一个函数。注解块应该放在函数的前面。所有的注解块的格式应相同，如以下代码所列。在注解块中，函数功能的必要信息应尽量多地给出。如果注解块和源代码加起来超过了一页纸，应加上强制分页符（可以加在注解块的末尾和下一个函数的开始之前）。这个分页符可使函数完整地打印在一页纸上，而不会在函数中间分页。如果函数本身比较长，超过了一页纸，应在逻辑上合理的地方插入分页符注释（/* $PAGE */）。例如，在 if 语句结束时分页，而不是在 if 语句的中间分页。很短的函数在同一页纸上可以多于 1 个，2 个函数之间至少应该空 2 行。

使用强制分页符

```
/*
*****************************************************************************
*                                CLEAR KEYBOARD BUFFER
*
* Description : Flush keyboard buffer
* Arguments   : none
* Returns     : none
* Notes       : none
*****************************************************************************
*/

void KeyClrBuf (void)
{

}
/* $PAGE */
```

只用于本文件的函数应用 static 声明为静态函数，以便隐藏起来不至于与其他文件中的函数混淆。

作为一条规则，在写到一个函数的调用时，在函数名与参数表之间不加空格；而在定义这个函数时，在函数名与形式参数表之间加一个空格。这样，可以利用 grep 命令很快地查找到这个函数是如何定义的。

函数名应该用相应文件名作为前缀。对于中到大型的程序，这样的前缀便于知道这个函数是在哪个文件中定义的。例如，所有文件 KEY.C 中的函数和所有文件 VIDEO.C 中的函数可以定义成以下形式：

```
KEY.C
        KeyGetChar()
        KeyGetLine()
        KeyGetFnctKey()
VIDEO.C
        VideoGetAttr()
        VideoPutChar()
        VideoPutStr()
        VideoSetAttr()
```

不必用整个文件名或模块名作为前缀。例如，文件名为 KEYBOARD.C，函数的前缀可以用 Key，而不必用 Keyboard。宁可使用大写字母分开不同的词，而不使用下划线。再次强调一下，下划线不会增加任何含义，还占用了一个字符的位置。如同前面提到的，形式参数和局部变量应该用小写，很清楚，这样的变量受作用域的限制。

声明局部变量时，每个局部变量名须占一行，以便需要时程序员可以给这个局部变量加上注释。声明局部变量时，前面缩进4个空格。局部变量和函数语句应该空3行。局部变量的声明应与函数语句明显地分开，因为它们是不同的。

A.11 缩进格式

缩进格式是表达函数流程的重要手段。问题是应该缩进几个空格？一个空格显然不够，而8个空格又太多了，折衷一下，笔者采用了4个空格。且笔者不用制表符(tab)，因为不同的打印机对于制表符的解释是不一样的，打印出来的文本式样也未必是所需要的。不用制表符并不意味着不可以使用键盘上的 Tab 键。好的文本编辑器在此情况下可以用4个空格换掉制表符 Tbl。

关键字 if、for，while 及 do 后面须加一个空格。如果使用括号，则关键字 else 的前面和后面都要加一个空格。笔者写条件语句时，if(条件)占一行，下一行写要执行的语句，如下所示：

```
if (x < 0)
    z = 25;

if (y > 2) {
    z = 10;
    x = 100;
    p++;
}
```

不要使用以下的写法：

```
if (x < 0) z = 25;
if (y > 2) {z = 10; x = 100; p++;}
```

建议这样写有2个原因：一是将判定部分和须执行的语句分开；二是要与while语句、for语句及do语句的格式保持一致。

可以像对待其他条件语句一样对待switch语句，case语句和case的标号要占用一行。这里的重点是switch语句，为便于阅读，case则要互相分得开。

```
if (x > 0) {
    y = 10;
    z =  5;
}

if (z < LIM) {
    x = y + z;
    z = 10;
} else {
    x = y - z;
    z = -25;
}

for (i = 0; i < MAX_ITER; i++) {
    *p2++ = *p1++;
    xx[i] = 0;
}

while (*p1) {
    *p2++ = *p1++;
    cnt++;
}

do {
    cnt--;
    *p2++ = *p1++;
} while (cnt > 0);
```

```
switch (key) {
    case KEY_BS :
        if (cnt > 0) {
            p-- ;
            cnt-- ;
        }
        break;

    case KEY_CR :
        *p = NUL;
        break;

    case KEY_LINE_FEED :
        p++ ;
        break;

    default:
        *p++ = key;
        cnt++ ;
        break;
}
```

A.12 语句和表达式

语句和表达式应在一行中写完，每行不多于一个赋值语句。不要用下面的写法：

```
x = y = z = 1;
```

虽然在C语言中这种写法是允许的，但是，当变量名变得更复杂时，程序的意图会变得不明显。

以下算符的前后都不要加空格：

->	结构指针算符	p->m
.	结构成员算符	s.m
[]	数组下标	a[i]

函数和函数后面的括号之间不要加空格；逗号后面要加一个空格，用于将函数参数表中的变元分开；括号中的表达式前后不要加空格；前括号后面以及后括号的前面也不要加空格；逗号和分号的后面应该加一个空格。

```
strncat(t, s, n);
for (i = 0; i < n; i++)
```

一元运算符与其操作数之间不要加空格。

```
!p    ~b    ++i    --j    (long)m    *p    &x    sizeof(k)
```

二元算符的前面和后面可以加一个或几个空格，三元运算符也是如此

```
c1 = c2      x + y      i += 2      n > 0 ? n : -n;
```

关键字 if、while、for、switch 以及 return 后面要加一个空格。

对于赋值语句，数值要按列对齐。再添加语句时也要这样，等号也要上下对齐。

```
x        = 100.567;
temp     =  12.700;
var5     =   0.768;
variable =  12;
storage  = &array[0];
```

A.13 结构和联合

结构是用 typedef 定义的，可以用一个标志符名代表整个结构。结构类型的声明一律用大写字母，用下划线将不同的词分开。

```
typedef struct line {          /* 定义 LINE 的 Structure                    */
    int  LineStartX;           /* "X","Y" 起点                              */
    int  LineStartY;
    int  LineEndX;             /* "X" ,"Y" 终点                             */
    int  LineEndY;
    int  LineColor;            /* line 的颜色                               */
} LINE;

typedef struct point {         /* 定义 POINT 的 structure                   */
    int  PointPosX;            /* 用 "X" ,"Y" 共同规范 point                */
    int  PointPosY;
    int  PointColor;           /* point 的颜色                              */
} POINT;
```

可以用同样的前缀来命名结构中的成员，如以上例子所示。成员名以结构的类型名开头，或者是相应类型名字的一部分开头，这样，当用指针指向该结构的成员时就非常清楚，例如：

```
p->LineColor;    /* 我们知道"p"是指向 LINE 的 */
```

A.14 参考文献

1 Babich Wayne A. Software Configuration Management. Reading, Massachusetts: Addison-Wesley Publishing Company, 1986

2 Long David W, Christopher P Duff. A Survey of Processes Used in the Development of Firmware for a Multiprocessor Embedded System. Hewlett-Packard Journal, 1993, p. 59～65

3 McConnell Steve. Code Complete. Redmond, Washington: Microsoft Press, 1993

4 Merant Inc. PVCS Version Manager. 735 SW 158th Avenue, Beaverton, OR 97006 (503) 645－1150

5 Merant Inc. PVCS Configuration Builder. 735 SW 158th Avenue, Beaverton, OR 97006 (503) 645－1150

附录 B

μC/OS-II 的使用许可证

虽然 μC/OS-II 以源码的方式提供给用户，但 μC/OS-II 不是自由软件或源码开放的软件。

B.01 用于学院和大学教学

μC/OS-II 的源码和目标码可以在有资质的学院和大学里免费提供给学生，用于非商业性目的。换句话说，如果 μC/OS-II 用于教学目的，则无须使用许可证。学院和大学应将相关课程的教学提纲发往 Micrium 网站登记，并提供相应的网页。该课程将被加到 Micrium 的网页上。请把相关信息发往：Universities@Micrium.com。

B.02 用于商业目的

将 μC/OS-II 的目标代码嵌入商业产品中，必须得到"目标代码销售许可证"。

这是一个以盈利为目的、将 μC/OS-II 的目标代码嵌入到商业产品中去销售的许可证。用于商业目的须付费，具体价格可与 Micrium 公司联系（见以下地址）。

如果销售 μC/OS-II 的源代码，则也应得到"源代码销售许可证"。再次强调，这样的许可证是要付费的，具体价格可与 Micrium 公司联系：

Licensing@Micrium.com

或者

Micrium, Inc.
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.
1-954-217-2036（电话）
1-954-217-2037（传真）
http://www.Micrium.com

附录 C

μC/OS－II 速查手册

本附录给出 μC/OS－II 提供的各种系统服务的汇总，假设用户允许了所有的功能（不打算把条件编译语句也编入这一手册中）。当然，有些功能不会在用户的应用程序中用到，这取决于如何设置文件 OS_CFG.H 的相关内容。

这些系统服务的排列是按照出现在本书相应章节的顺序编排的：

- 杂项（内核结构）；
- 任务管理；
- 时间管理；
- 信号量管理；
- 互斥型信号量管理；
- 事件标志组管理；
- 消息邮箱管理；
- 消息队列管理；
- 内存管理。

笔者还做了一张任务设计计划单，用户可以把计划中应用程序的各个任务列在这张设计计划单上。

杂 项

(第3章)

函数原型

```
void        OSInit(void);
void        OSIntEnter(void);
void        OSIntExit(void);
void        OSSchedLock(void);
void        OSSchedUnlock(void);
void        OSStart(void);
void        OSStatInit(void);
INT16U      OSVersion(void);
```

宏定义

```
OS_ENTER_CRITICAL()
OS_EXIT_CRITICAL()
```

全局变量

```
INT8S       OSCPUUsage          // CPU 利用率(%)
INT8U       OSIntNesting        // 中断嵌套层数(0~255)
INT8U       OSLockNesting       // OSSchedLock()的嵌套层数
BOOLEAN     OSRunning           // 正在运行多任务的标志
INT8U       OSTaskCtr           // 已经建立了的任务数
OS_TCB      *OSTCBCur           // 指向当前任务控制块 TCB 的指针
OS_TCB      *OSTCBHighRdy       // 指向最高优先级任务的控制块 TCB 的指针
```

任务管理

(第4章)

函数原型

```
INT8U        OSTaskChangePrio(INT8U oldprio, INT8U newprio);
INT8U        OSTaskCreate(void      (*task)(void *pd),
                          void      *pdata,
                          OS_STK    *ptos,
                          INT8U      prio);
INT8U        OSTaskCreateExt(void   (*task)(void *pd),
                             void    *pdata,
                             OS_STK  *ptos,
                             INT8U    prio,
                             INT16U   id,
                             OS_STK  *pbos,
                             INT32U   stk_size,
                             void    *pext,
                             INT16U   opt);
INT8U        OSTaskDel(INT8U prio);
INT8U        OSTaskDelReq(INT8U prio);
INT8U        OSTaskResume(INT8U prio);
INT8U        OSTaskSuspend(INT8U prio);
INT8U        OSTaskStkChk(INT8U prio, OS_STK_DATA *pdata);
INT8U        OSTaskQuery(INT8U prio, OS_TCB *pdata);
```

OSTaskCreateExt()的opt参数

```
OS_TASK_OPT_STK_CHK        // 允许该任务的堆栈检验
OS_TASK_OPT_STK_CLR        // 建立任务时将堆栈清0
OS_TASK_OPT_SAVE_FP        // 保存浮点数寄存器
```

OSTaskDelReq()的返回值

```
OS_NO_ERR                  // 删除任务请求已经注册
OS_TASK_NOT_EXIST          // 任务已经被删除
OS_TASK_DEL_IDLE           // 不能删除空闲任务!
OS_PRIO_INVALID            // 无效优先级
```

OSTaskStkChk( )数据结构

```
typedef struct {
    INT32U  OSFree;                 // 堆栈中从未使用过的字节数
    INT32U  OSUsed;                 // 堆栈中已用到过的字节数
} OS_STK_DATA;
```

OSTaskQuery( )数据结构

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;           // 堆栈指针

    void            *OSTCBExtPtr;           // TCB 扩展指针

    OS_STK          *OSTCBStkBottom;        // 指向栈底的指针
    INT32U           OSTCBStkSize;          // 堆栈容量(可保存的地址数)

    INT16U           OSTCBOpt;              // 任务选项

    INT16U           OSTCBId;               // 任务标志 Task ID (0~65 535)

    struct os_tcb   *OSTCBNext;             // 指向下一个任务控制块 TCB
    struct os_tcb   *OSTCBPrev;             // 指向上一个任务控制块 TCB

    OS_EVENT        *OSTCBEventPtr;         // 指向任务控制块 ECB 的指针

    void            *OSTCBMsg;              // 消息已经收到

    OS_FLAG_NODE    *OSTCBFlagNode;         // 指向事件标志节点的指针
    OS_FLAGS         OSTCBFlagsRdy;         // 使任务进入就绪态的事件标志

    INT16U           OSTCBDly;              // 任务延迟的时钟节拍数,或超时
    INT8U            OSTCBStat;             // 任务状态
    INT8U            OSTCBPrio;             // 任务优先级,0 为最高

    INT8U            OSTCBX;
    INT8U            OSTCBY;
    INT8U            OSTCBBitX;
    INT8U            OSTCBBitY;

    BOOLEAN          OSTCBDelReq;           // 告知任务删除自己的标志
} OS_TCB;
```

时间管理

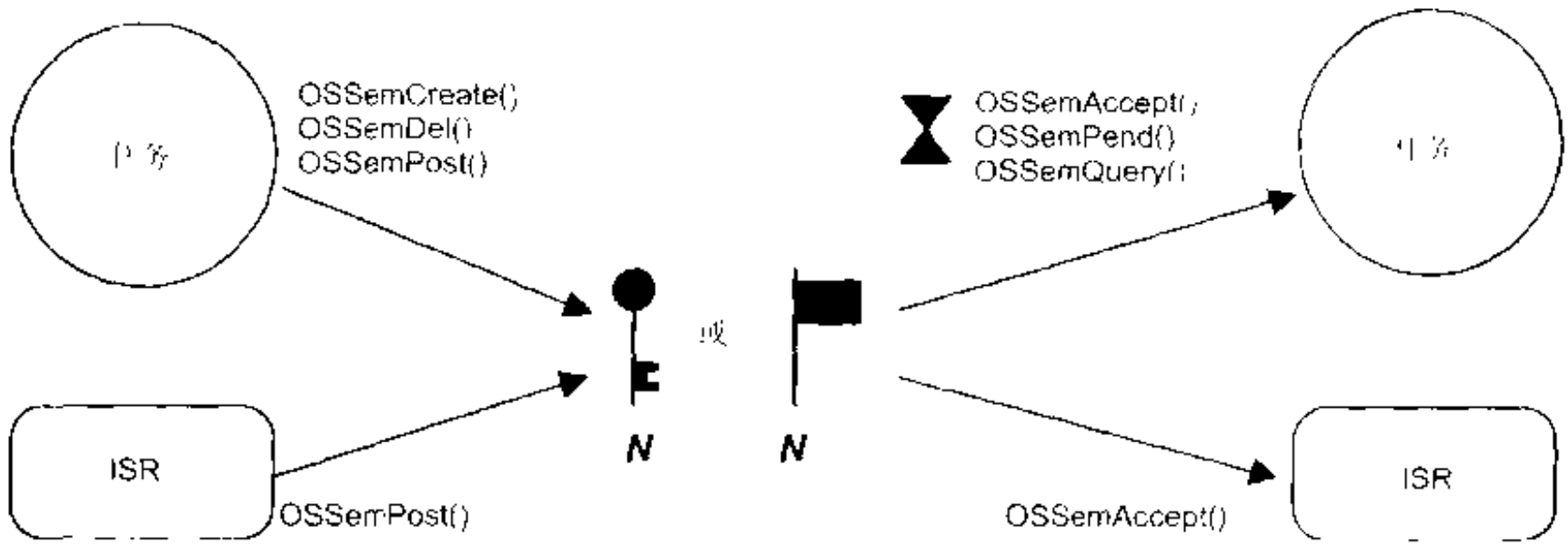
(第 5 章)

函数原型

```
void        OSTimeDly(INT16U ticks);

INT8U       OSTimeDlyHMSM(INT8U  hours,
                          INT8U  minutes,
                          INT8U  seconds,
                          INT16U milli);

INT8U                     OSTimeDlyResume(INT8U prio);

INT32U                    OSTimeGet(void);

void                      OSTimeSet(INT32U ticks);

void                      OSTimeTick(void);
```

信号量管理

(第 7 章)



函数原型

```
INT16U          OSSemAccept(OS_EVENT *pevent);

OS_EVENT       *OSSemCreate(INT16U cnt);

OS_EVENT       *OSSemDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

void            OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

INT8U           OSSemPost(OS_EVENT *pevent);

INT8U           OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata);
```

OSSemDel() 的 opt 参数

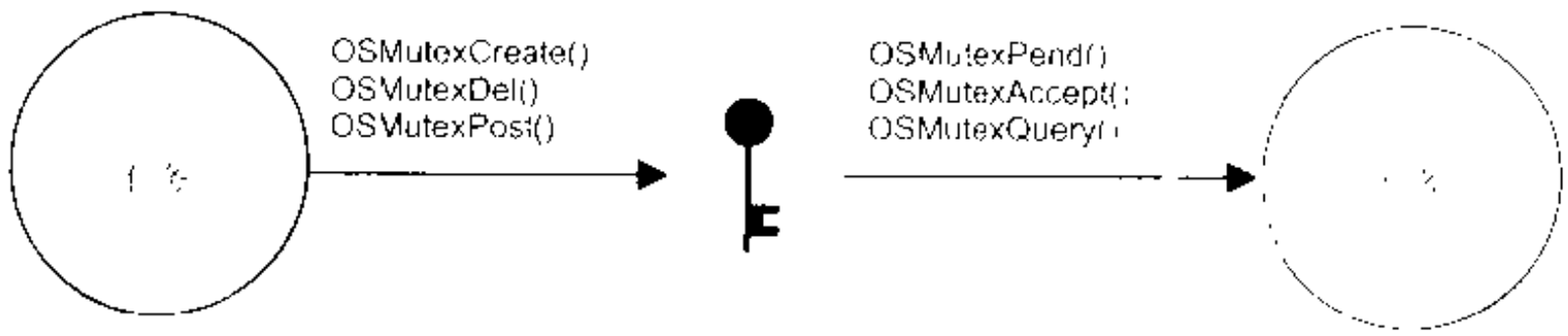
```
OS_DEL_NO_PEND                          // 没有任务等待时才删除
OS_DEL_ALWAYS                           // 无论有无等待的任务都删除
```

OSSemQuery()数据结构

```
typedef struct {
    INT16U  OSCnt;                      // 信号量计数器
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];  // 等待事件的任务列表
    INT8U   OSEventGrp;
} OS_SEM_DATA;
```

互斥型信号量管理

(第8章)



函数原型

```
INT8U          OSMutexAccept(OS_EVENT *pevent, INT8U *err);

OS_EVENT       *OSMutexCreate(INT8U prio, INT8U *err);

OS_EVENT       *OSMutexDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

void           OSMutexPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

INT8U          OSMutexPost(OS_EVENT *pevent);

INT8U          OSMutexQuery(OS_EVENT *pevent, OS_MUTEX_DATA *pdata);
```

OSMutexDel()的opt参数

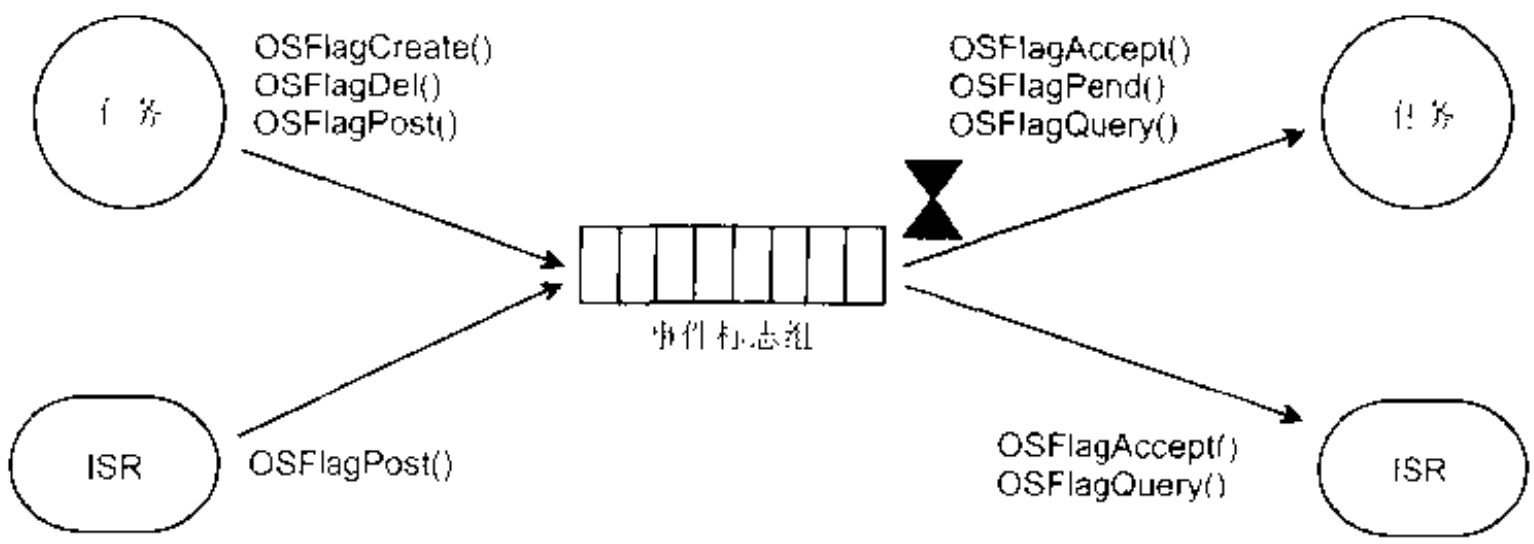
```
OS_DEL_NO_PEND                    // 没有任务等待时才删除
OS_DEL_ALWAYS                     // 无论有无等待的任务都删除
```

OSMutexQuery()数据结构

```
typedef struct {
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];   // 等待 mutex 的任务列表
    INT8U   OSEventGrp;
    INT8U   OSValue;                         // mutex 的值
                                             // (0=被占用，1=可以使用)
    INT8U   OSOwnerPrio;                     // 占用 mutex 的任务的优先级
    INT8U   OSMutexPIP;                      // 优先级继承优先级，或
                                             // 0xFF 没有被占用
} OS_MUTEX_DATA;
```

事件标志组管理

(第9章)



函数原型

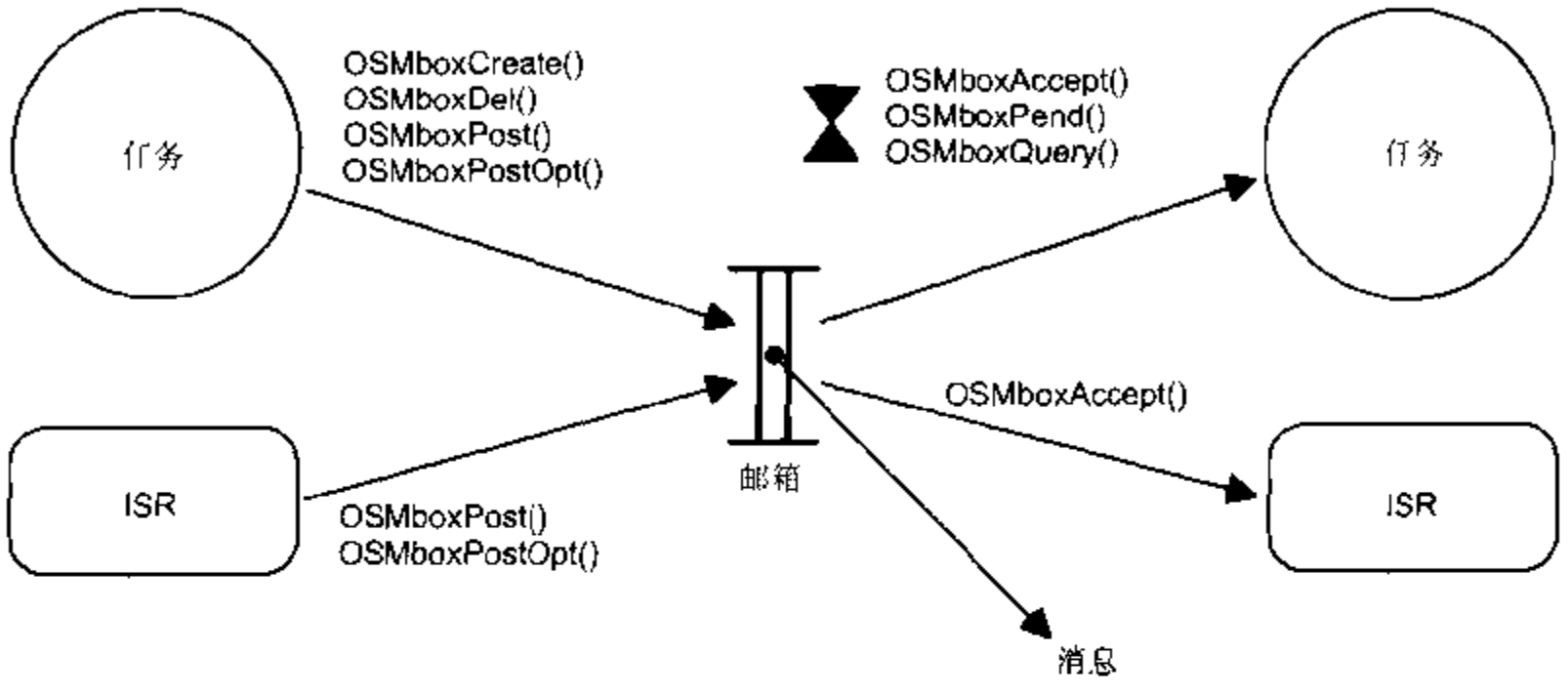
```
OS_FLAGS          OSFlagAccept(OS_FLAG_GRP *pgrp,
                               OS_FLAGS     flags,
                               INT8U        wait_type,
                               INT8U       *err);

OS_FLAG_GRP      *OSFlagCreate(OS_FLAGS      flags,
                               INT8U       *err);

OS_FLAG_GRP      *OSFlagDel(OS_FLAG_GRP     *pgrp,
                            INT8U            opt,
                            INT8U           *err);

OS_FLAGS          OSFlagPend(OS_FLAG_GRP    *pgrp,
                             OS_FLAGS        flags,
                             INT8U           wait_type,
                             INT16U          timeout,
                             INT8U          *err);

OS_FLAGS          OSFlagPost(OS_FLAG_GRP    *pgrp,
                             OS_FLAGS        flags,
                             INT8U           operation,
                             INT8U          *err);

OS_FLAGS          OSFlagQuery(OS_FLAG_GRP   *pgrp,
                              INT8U         *err);
```

OSFlagDel()的 opt 参数

```
OS_DEL_NO_PEND                    // 没有任务等待时才删除
OS_DEL_ALWAYS                     // 无论有无等待的任务都删除
```

消息邮箱管理

(第10章)



函数原型

```
void          *OSMboxAccept(OS_EVENT *pevent);

OS_EVENT      *OSMboxCreate(void *msg);

OS_EVENT      *OSMboxDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

void          *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

INT8U          OSMboxPost(OS_EVENT *pevent, void *msg);

INT8U          OSMboxPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);

INT8U          OSMboxQuery(OS_EVENT *pevent, OS_MBOX_DATA *pdata);
```

OSMboxDel() 的 opt 参数

```
OS_DEL_NO_PEND                          // 没有任务等待时才删除
OS_DEL_ALWAYS                           // 无论有无等待的任务都删除
```

OSMboxPostOpt() 的 opt 参数

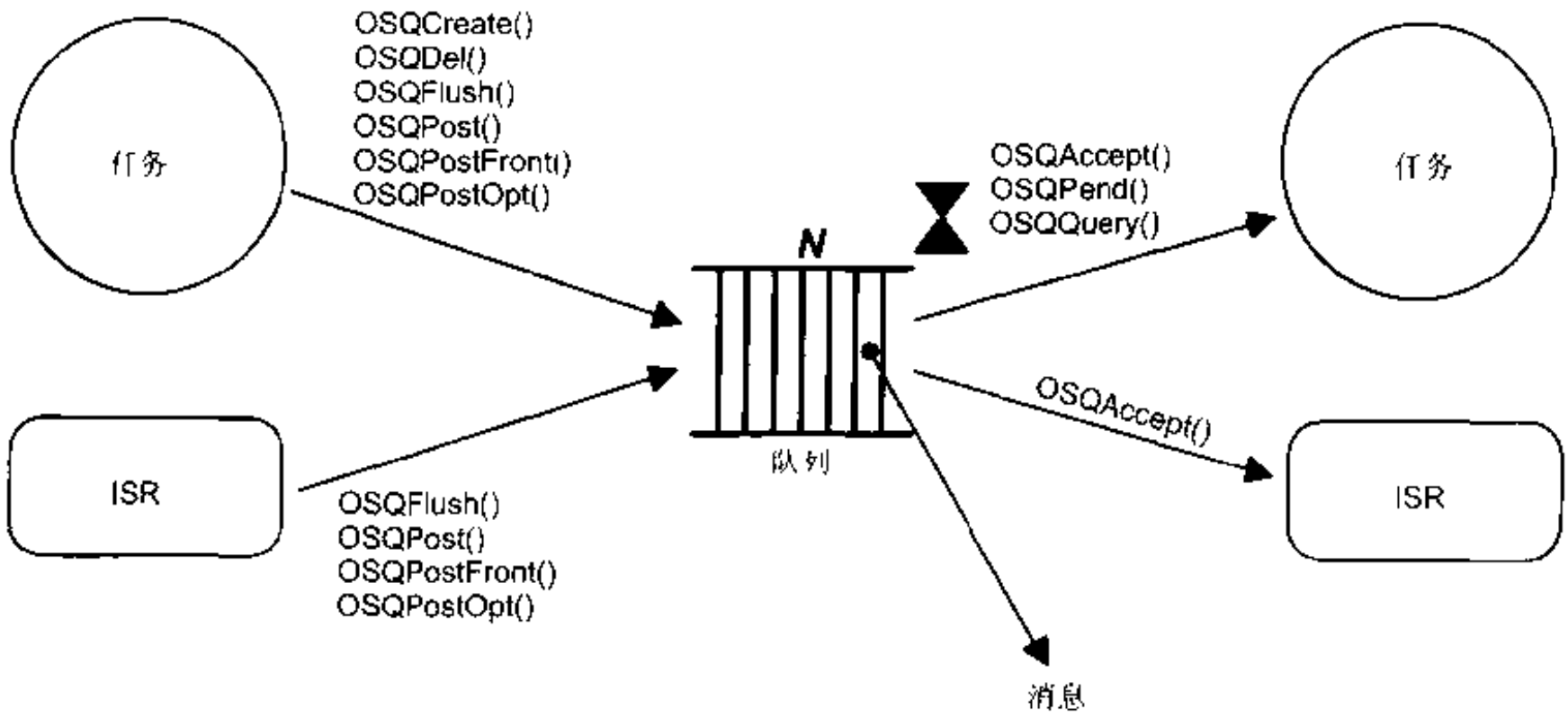
```
OS_POST_OPT_NONE                    // 消息发给一个等待邮箱消息的任务
                                    // (等同于 OSMboxPost())
OS_POST_OPT_BROADCAST               // 广播给所有等待邮箱消息的任务
```

OSMboxQuery()数据结构

```
typedef struct {
    void    *OSMsg;                     // 邮箱中指向消息的指针
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];  // 等待列表
    INT8U   OSEventGrp;
} OS_MBOX_DATA;
```

消息队列管理

(第11章)



函数原型

```
void          *OSQAccept(OS_EVENT *pevent);

OS_EVENT      *OSQCreate(void **start, INT16U size);

OS_EVENT      *OSQDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

INT8U          OSQFlush(OS_EVENT *pevent);

void          *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

INT8U          OSQPost(OS_EVENT *pevent, void *msg);

INT8U          OSQPostFront(OS_EVENT *pevent, void *msg);

INT8U          OSQPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);

INT8U          OSQQuery(OS_EVENT *pevent, OS_Q_DATA *pdata);
```

OSQDel( ) 的 opt 参数

```
OS_DEL_NO_PEND                    // 没有任务等待时才删除
OS_DEL_ALWAYS                     // 无论有无等待的任务都删除
OS_POST_OPT_FRONT                 // 模仿 OSQPostFront()(是 LIFO 还是 FIFO)
```

OSQPostOpt( ) 的 opt 参数

```
OS_POST_OPT_NONE                  // 消息发给一个等待消息的任务
                                  // (等同于 OSMboxPost())
OS_POST_OPT_BROADCAST             // 广播给所有等待消息的任务
```

OSQQuery( )数据结构

```
typedef struct {
    void        *OSMsg;                          // 指向下一则消息的指针
    INT16U       OSNMsgs;                        // 队列中的消息数目
    INT16U       OSQSize;                        // 消息队列的容量
    INT8U        OSEventTbl[OS_EVENT_TBL_SIZE];  // 等待列表
    INT8U        OSEventGrp;
} OS_Q_DATA;
```

内存管理
(第12章)

函数原型

```
OS_MEM          *OSMemCreate(void       *addr,
                             INT32U      nblks,
                             INT32U      blksize,
                             INT8U      *err);

void            *OSMemGet(OS_MEM *pmem, INT8U *err);

INT8U            OSMemPut(OS_MEM *pmem, void *pblk);

INT8U            OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata);
```

OSMemQuery()数据结构

```
typedef struct {
    void        *OSAddr;                // 指向存储块起始地址的指针
    void        *OSFreeList;            // 指向空余存储块列表中第1个存储块的指针
    INT32U       OSBlkSize;             // 每个存储块的容量(Bytes)
    INT32U       OSNBlks;               // 存储块的总数
    INT32U       OSNFree;               // 空余存储块
    INT32U       OSNUsed;               // 已经使用了的存储块
} OS_MEM_DATA;
```

μC/OS－II, The Real-Time Kernel

Task Assignment Worksheet

Priority	Task Name	Stack Size (Bytes)	Description	Mutex PIP?
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23				
24				
25				
26				
27				
28				
29				
30				
31				
32				
33				
34				
35				
36				
37				
38				
39				
40				
41				
42				
43				
44				
45				
46				
47				
48				
49				
50				
51				
52				
53				
54				
55				
56				
57				
58				
59				
60				
61				
62				
63	μC/OS－II Idel Task			N/A

附录 D

工具程序 TO

TO 是 DOS 下的工具程序，可以使用户变更路径时不必使用：

CD path

或

CD ..\path

TO 命令可能是笔者用得最多的 DOS 命令，因为这条命令使得目录之间的转换变得特别快。在 DOS 提示下，只要键入：

TO name

就能迅速进入想去的目录。这里用路径的代名词 name 表示想去的目录。路径和该路径的代名词是放在 ASCII 码文件 TO.TBL 中的，放在当前驱动器的根目录下。TO 扫描文件 TO.TBL，以找到命令行中的代名词。如果代名词存在于文件 TO.TBL 之中，则变换到相应路径；如果代名词在 TO.TBL 中找不到，则显示 Invalid Name。

DOS 下的可执行文件在\SOFTWARE\TO\EXE\TO.EXE 中，路径以及路径的代名词的范例文件是\SOFTWARE\TO\EXE\TO.TBL，TO 命令的源码见\SOFTWARE\TO\SOURCE\TO.C。

TO.TBL 文件的范例和格式如程序清单 LD.1 所列。

程序清单 D.1　TO.TBL 文件的范例

```
A,          ..\SOURCE
C,          ..\SOURCE
D,          ..\DOC
L,          ..\LST
O,          ..\OBJ
P,          ..\PROD
W,          ..\WORK
EX1L,       \SOFTWARE\uCOS-II\EX1_x86L\BC45\TEST                    (1)
EX2L,       \SOFTWARE\uCOS-II\EX2_x86L\BC45\TEST
EX3L,       \SOFTWARE\uCOS-II\EX3_x86L\BC45\TEST
Ix86L,      \SOFTWARE\uCOS-II\Ix86L\BC45
TO,         \SOFTWARE\TO\SOURCE
uCOS-II,    \SOFTWARE\uCOS-II\SOURCE
```

给TO.TBL增加一个路径代名词及路径，可以键入：

TO name path

这里name是拟去路径path的代名词。这样就免去了重新编辑TO.TBL文件。如果键入：

TO EXIL

就改变路径到\SOFTWARE\uCOS-II\EX1_x86L\BC45\TEST [LD.1(1)] 子目录。

TO.TBL文件可以很长，但不得重名。不同名的2个代名词可以指向同一个子目录。如果用文本编辑写TO.TBL文件，则所有字母须大写。因为在DOS下，键入的命令被转换成了大写。TO在扫描TO.TBL文件时，是从头扫描的。为了扫描得快，最常用的子目录和代名词应尽量往前放，虽然这对于当今速度很快的计算机来说也许并不需要。

附录 E

参考文献

1 Allw orth Steve T. Introduction To Real-Time Software Design. New York: Springer-Verlag, 1981

2 Bal Sathe Dhananjay. Fast Algorithm Determines Priority. EDN(India), 1988, September: p. 237

3 Comer Douglas. Operating System Design, The XINU Approach. Englewood Cliffs, New Jersey: Prentice-Hall, 1984

4 Deitel Harvey M, Michael S Kogan. The Design Of OS/2. Reading, Massachusetts: Addison-Wesley, 1992

5 Ganssle Jack G. The Art of Programming Embedded Systems. San Diego: Academic Press, 1992

6 Gareau Jean L. Embedded Systems Programming. Embedded x86 Programming: Protected Mode. 1998, April: p. 80～93

7 Halang Wolfgang A, Alexander D Stoyenko. Constructing Predictable Real Time Systems. Norwell, Massachusetts: Kluwer Academic Publishers Group, 1991

8 Hunter & Ready. VRTX Technical Tips. Palo Alto, California: Hunter & Ready, 1986

9 Hunter & Ready. Dijkstra Semaphores, Application Note. Palo Alto, California: Hunter & Ready, 1983

10 Hunter & Ready. VRTX and Event Flags. Palo Alto, California: Hunter & Ready, 1986

11 Intel Corporation. iAPX 86/88, 186/188 User's Manual: Programmer's Reference. Santa Clara, California: Intel Corporation, 1986

12 Kernighan Brian W, Dennis M Ritchie. The C Programming Language. 2nd edition. Englewood Cliffs, New Jersey: Prentice Hall, 1988

13 Klein Mark H, Thomas Ralya, Bill Pollak, Ray Harbour Obenza, Michael Gonzlez. A Practioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Norwell, Massachusetts: Kluwer Academic Publishers Group, 1993

14 Laplante Phillip A. Real-Time Systems Design and Analysis, An Engineer's Handbook. Piscataway, New Jersey: IEEE Computer Society Press, 1992

15 Lehoczky John, Lui Sha, Ye Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. Proceedings of the IEEE Real-Time

Systems Symposium, 1989, Los Alamitos, California. Piscataway, New Jersey: IEEE Computer Society, p. 166～171

16 Madnick E Stuart, John J Donovan. Operating Systems. New York: McGraw-Hill, 1974

17 Ripps David L. An Implementation Guide To Real-Time Programming. Englewood Cliffs, New Jersey: Yourdon Press, 1989

18 Savitzky Stephen R. Real-Time Microprocessor Systems. New York: Van Nostrand Reinhold, 1985

19 Wood Mike, Tom Barrett. Embedded Systems Programming. A Real-Time Primer, 1990, February: p. 20～28

附录 F

本书所附光盘

本书所附光盘中有可以自动解压缩并执行的文件 uCOSV252.EXE 。因为光盘尚有足够的空间，决定把全部文件按原样做上去，以便读者在光盘上就可以浏览，而不必安装在自己的计算机上。

假定读者使用的计算机装有 Microsoft Windows 95，98，NT，2000 或 XP 操作系统，用的是 80x86 的或奔腾级的处理器，或者是 AMD 的处理器，硬盘上至少有 10 MB 空间可以安装 μC/OS－II 及其源代码。

把光盘插入光盘驱动器中，执行光盘根目录下的 uCOSV252.EXE 文件，就会在屏幕中间弹出如图 FF.1 所示的图形。

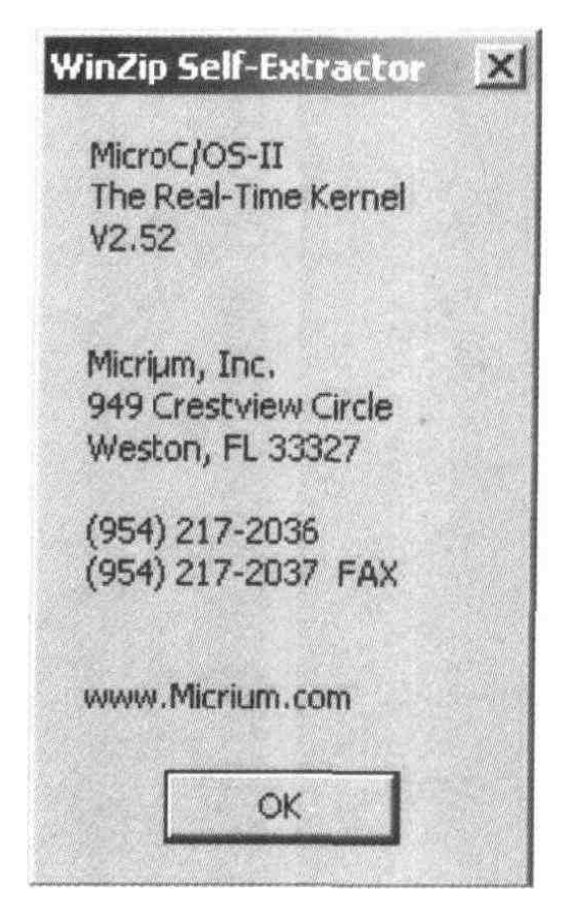


图 FF.1　执行 uCOSV252.EXE 弹出的图形

单击 OK，uCOSV252.EXE 在屏幕中间弹出如图 FF.2 所示的图形。要求确认打算安装 μC/OS－II的目录，树形结构的源文件默认的安装目录是驱动器 C:上的根目录。也可以通过指定路径安装到 PC 机的其他驱动器上。

选择安装部位以后（或接受默认路径），单击 Unzip 按钮解压缩。图 FF.3 是解压缩后屏幕上显示的信息。

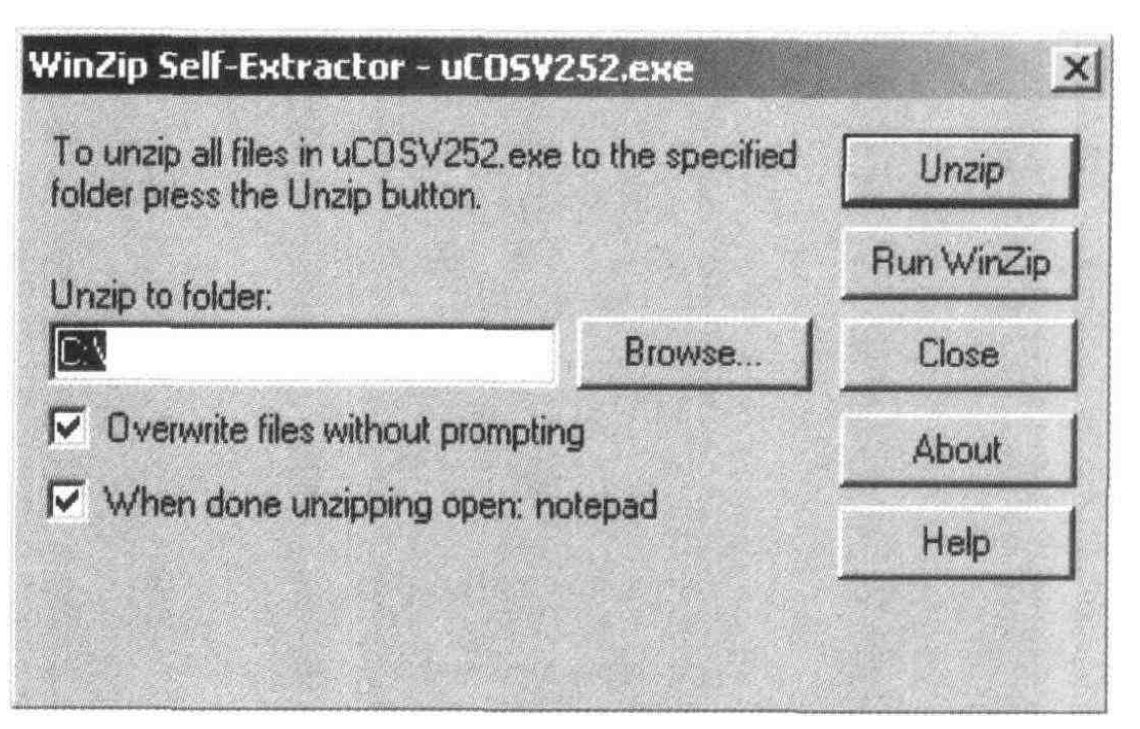


图 FF.2　选择 μC/OS－II 的安装目录

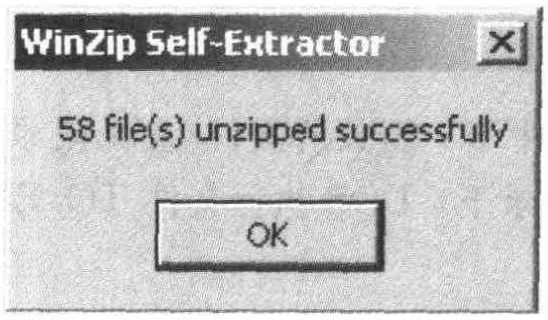


图 FF.3 文件的解压缩信息

单击 OK，Microsoft 的 Notepad 工具打开并显示文件 README. TXT，如图 FF. 4 所示。阅读完以后，在文件菜单中选择 close 关闭文件。

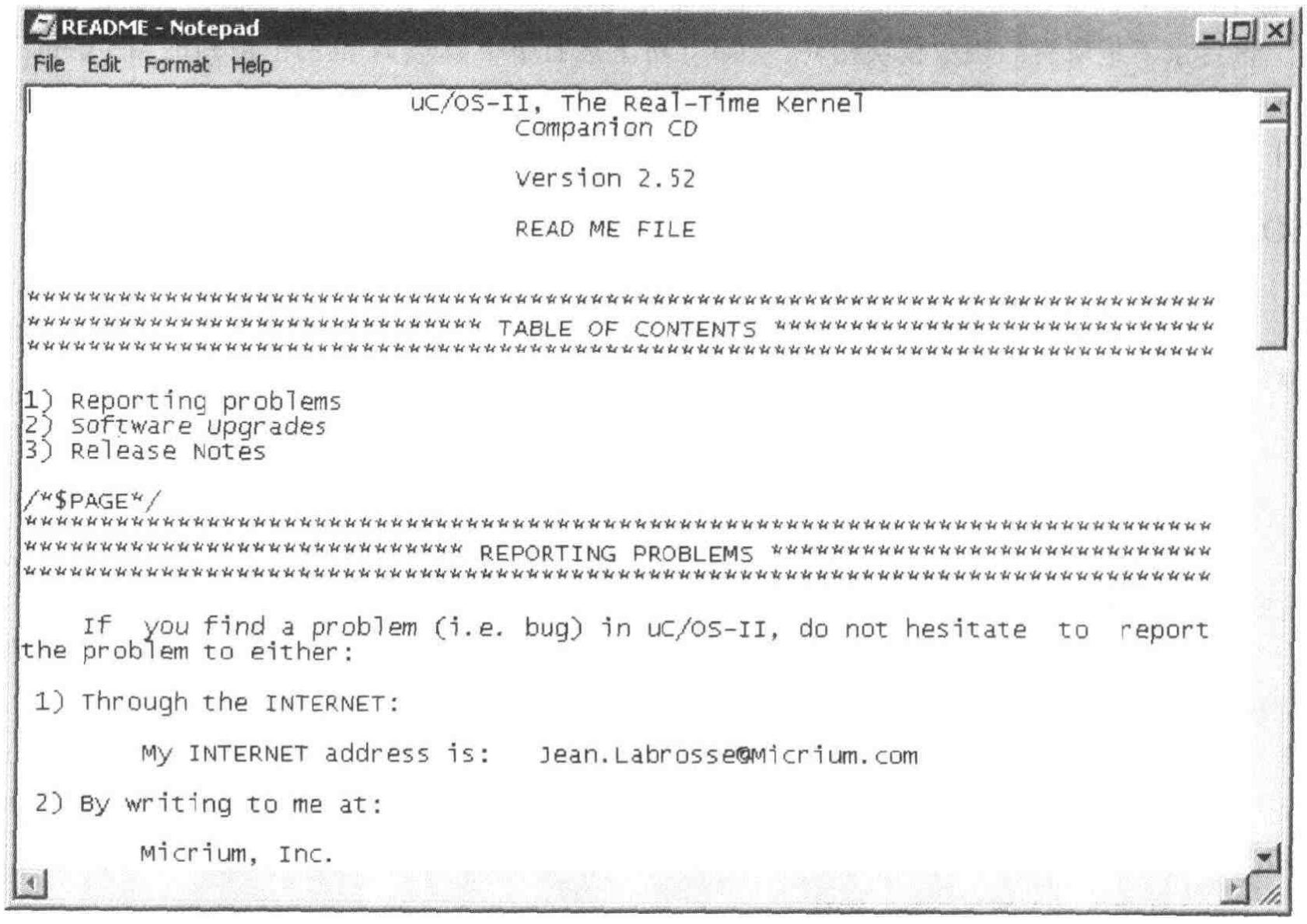


图 FF.4 文件 README. TXT

F.01 文件和目录

安装完成以后，目标硬盘上应该包含以下目录。实际上，光盘上也有这些文件和目录。

\SOFTWARE

相应驱动器根目录下的主目录，与 μC/OS－II 有关的所有文件都在这个目录下。

\SOFTWARE\TO

在这个目录下,有与工具软件 TO 相关的文件(参见附录 D)。源文件是 TO.C,存放在 \SOFTWARE\TO\SOURCE目录下。DOS 执行文件 TO.EXE 在 \SOFTWARE\TO\EXE 目录下。

注意:TO 需要一个名为 TO.TBL 的文件支持,该文件必须放在根目录下。在目录 \SOFTWARE\TO\EXE下有一个 TO.TBL 文件的范例。使用 TO.EXE 时,要把 TO.TBL 文件移到根目录下。

\SOFTWARE\BLOCKS

该目录下是所有的函数模块构件。在 PC 机上运行 μC/OS-II 范例中用到的与 PC 机有关的函数也在这个目录下。源文件 PC.C 和 PC.H 在 \SOFTWARE\BLOCKS\PC\BC45 目录下。

\SOFTWARE\uCOS-II

所有有关 μC/OS-II 的文件都在这个目录下。

\SOFTWARE\uCOS-II\DOC

该目录中包含所有的文档文件,主要包括:

README.TXT

为当前版本编写的 README 文件,在安装 μC/OS-II 的过程中会见到这个文件。

RevV252.PDF

该文件是对当前 μC/OS-II 的 V2.52 版本的说明,需要用 Adobe Acrobat Reader 软件工具阅读这个文件。

NewV252.PDF

该文件中描述 μC/OS-II 从 V2.00 到 V2.52 版本都做了哪些改进和变更。需要用 Adobe Acrobat Reader软件工具阅读该文件。

QuickRefChartV252-Color.PDF

该文件中有 μC/OS-II 可提供功能的快速参考流程图,也需要用 Adobe Acrobat Reader 软件工具来阅读。文件打印出来后,可以展开成一张大图,也可以叠成半张纸大小,做成袖珍参考手册。

TaskAssignmentWorksheet.PDF

TaskAssignmentWorksheet.XLS

用户计划和安排工程项目中的任务时,可以使用这些文件中的表格,也需要用 Adobe Acrobat Reader 软件工具打开这些文件。.XLS 文件是 Microsoft 的 Excel 列表文件,可用于给用户应用程序建立文档。

\SOFTWARE\uCOS-II\EX1_x86L\BC45

该目录下包含第 1 章中范例 1 的源代码,是为在 DOS 环境下运行而编写的(或在

Microsoft Windows 环境下的 DOS 窗口下运行)。BC45 的含义是,假定用户使用的是 Borland C/C++ 编译器 V4.5x 版本。当然,如果需要,用户可以修改这些文件,使之适用于其他版本的 C 编译器。BC45 目录下还有一些子目录,每个子目录下都包含以下 4 个文件:

\SOURCE

INCLUDES.H 文件包含的主文件,用于 μC/OS-II 和代码测试。

OS_CFG.H

μC/OS-II 的配置文件。该文件定义打算使用 μC/OS-II 中的哪些功能,应用程序中有多少个任务等等。

TEST.C

范例 1 的测试代码。

TEST.LNK

Turbo 汇编工具的链接命令文件。它定义在生成最终的可执行文件 TEST.EXE 时,使用什么样的目标文件以及使用什么样的函数库。

\TEST

该目录下的 MAKETEST.BAT 是 DOS 环境下的一个批处理文件,运行后可生成范例 1 的可执行代码。MAKETEST.BAT 假设驱动器 C:上有 Borland C 的 MAKE 工具。如果在别的目录下编译,则应该重新编辑 MAKETEST.BAT 文件。

TEST.MAK

TEST 的 makefile 文件。供用户生成 DOS 环境下的可执行文件 TEST.EXE。TEST.MAK文件包含所有的编译、汇编以及链接命令,用以生成 TEST.EXE。

TEST.EXE

笔者使用自己的编译工具生成的范例 1 的可执行文件,可以在 DOS 环境下执行。用户可以在 Microsoft Windows 95、98、ME、NT、2000 或 XP 的 DOS 窗口下直接运行这个文件。

TEST.MAP

链接生成的 MAP 文件。

\SOFTWARE\uCOS-II\EX2_x86L\BC45

该目录下包含第 1 章中范例 2 的源代码,是为在 DOS 环境下运行而编写的(或在 Microsoft Windows 环境下的 DOS 窗口下运行)。这个目录与前面描述的范例 1:EX1_x86L 是一样的。换句话说,包含的文件名都一样,只是内容不同。

\SOFTWARE\uCOS-II\EX3_x86L\BC45

该目录下包含第 1 章中范例 3 的源代码,是为在 DOS 环境下运行而编写的(或在 Microsoft Windows 环境下的 DOS 窗口下运行)。这个目录与前面描述的范例 1:EX1_x86L 是一样的。换句话说,包含的文件名都一样,只是内容不同。

\SOFTWARE\uCOS-II\EX4_x86L.FP\BC45

该目录下包含第 1 章中范例 4 的源代码,是为在 DOS 环境下运行而编写的(或在

Microsoft Windows 环境下的 DOS 窗口下运行)。这个目录与前面描述的范例 1:EX1_x86L 是一样的。换句话说,包含的文件名都一样,只是内容不同。

\SOFTWARE\uCOS-II\Ix86L\BC45

该目录下是 μC/OS-II 中所有与处理器类型相关部分的源代码(也称为移植代码),面向 80x86 处理器,运行于实模式下,用大模块编译,用的是 Borland C/C++ V4.5x 编译器。这些移植代码允许用户使用 Borland C 提供的浮点运算仿真库,并保证函数的可重入性。

OS_CPU_A.ASM

有关移植的用汇编语言编写的函数,这些函数是 OSStartHighRdy(),OSCtxSw(),OSIntCtxSw()及 OSTickISR()。

OS_CPU_C.C

这一部分的 C 语言函数文件。

OS_CPU.H

这一部分 C 程序的头文件。

\SOFTWARE\uCOS-II\Ix86L-FP\BC45

该目录下是 μC/OS-II 中所有与处理器类型相关部分的源代码(也称为移植代码),面向 80x86 处理器,运行于实模式下,用大模块编译,用的是 Borland C/C++ V4.5x 编译器。这些移植代码也适用于带浮点协处理单元的 80x86 处理器。换句话说,应用程序中的任务允许使用浮点处理单元(FPU),μC/OS-II 在做任务切换时保存 FPU 寄存器。

OS_CPU_A.ASM

有关移植的用汇编语言编写的函数,这些函数是 OSStartHighRdy(),OSCtxSw(),OSIntCtxSw()及 OSTickISR()。

OS_CPU_C.C

这一部分的 C 语言函数文件。

OS_CPU.H

这一部分 C 程序的头文件。

\SOFTWARE\uCOS-II\SOURCE

该目录下是 μC/OS-II 中所有与处理器类型无关部分的源代码。这些代码完全可以移植到其他架构的处理器上。这个目录下有以下一些文件:

OS_CORE.C	OS_FLAG.C
OS_MBOX.C	OS_MEM.C
OS_MUTEX.C	OS_Q.C
OS_SEM.C	OS_TASK.C
OS_TIME.C	uCOS_II.C
	uCOS_II.H

如果您从未用过RTOS，那么这本书将让您掌握RTOS——开发应用程序的得力工具；如果您打算选购商用RTOS，那么这本书为您揭示RTOS的内在奥秘。

——Jack Ganssle，顾问，教师，作家

μC/OS-II是一种可移植、可固化、可裁剪及可剥夺型的多任务实时内核（RTOS），适用于各种微处理器和微控制器。其性能足可媲美于商用内核，在某些方面甚至更佳。所有代码用ANSI的C语言编写，故具有良好的可移植性。实际上，μC/OS-II已经被移植到40多种处理器架构中，从8位到64位的各种CPU。自1992年起，μC/OS已在全世界从照相机到航空设备等各个领域得到广泛应用。这个RTOS的健壮性、可靠性和安全性已经得到了广泛的证明。对于各类系统，包括一些对安全性要求极为苛刻的系统，它都能够胜任。

本书介绍μC/OS-II的设计与实现，讲解多任务实时系统的基本原理。读者可以学到内核结构、调度算法、任务切换、时间管理、任务间通信与同步及存储空间的动态分配等知识。与本书的前一版本相比，书中程序的解释全部重新写过，给出了更详尽的说明，以便于理解；系统功能函数的描述从1章扩充到了6章，其中2章用以描述新增加的功能函数：互斥型信号量（mutex）和事件标志组函数；重写了程序移植指南，新增加了移植过程中的调试技巧等内容；编程规约函数以及变量等的速查手册也编排在附录中。

附带的光盘包含了μC/OS-II V2.52的全部源代码，以及移植到Intel 80x86处理器上的范例。实时模式，在大模块方式下编译，只需几分钟，就可以在用户PC机的Windows环境下测试μC/OS-II的一些特性。

ISBN 7-81077-290-2

9 787810 772907 >

ISBN 7-81077-290-2

定价：79.00元（含光盘）

μC/OS-II