任哲 编著

嵌入式实时操作系统 μC/OS-II 原理及应用

北京航空航天大学出版社

高等院校通用教材

嵌入式实时操作系统 μC/OS－II 原理及应用

任 哲 编著

北京航空航天大学出版社

前　言

有人说，嵌入式系统的应用和发展是计算机技术自桌面系统和网络技术之后，IT 产业和 IT 技术的又一个新的重大进展。

事实的确如此。就现在的情况来看，在当今人们的生产和生活中，嵌入式系统的应用和影响无处不在。暂且不论火箭、地球卫星、汽车生产线，也不论先进的武器装备系统，就是在日常生活中人们几乎天天离不开的移动电话、空调器、数码相机、洗衣机、电饭锅、电热水器等装置和设备中，都有嵌入式系统暗暗地在发挥着控制作用，从而使这些装置和设备具有了较高的自动化水平，甚至体现出了某种智能性，为人们提供了更为体贴、更为周到的服务。可以期望，就在不远的将来，普通百姓还会用到地球卫星定位系统、高级音像系统、智能可视电话、智能采暖设备、个人医疗保健设备、家用机器人等更为人性化的装置和设备，从而使人们的生活更舒适，工作效率更高。然而，这一切无不依赖于嵌入式系统技术的发展和应用。

众所周知，我国经济建设正处在一个高速发展时期，我国正在成为世界电子制造业的重要基地之一，从而形成了对嵌入式系统技术及相关技术人才的巨大需求。面对这种日益广泛而迫切的需求，各高等院校和培训机构都纷纷开设了与嵌入式系统技术相关的专业和课程，以期学生在校学习期间就能受到嵌入式系统技术方面的教育和训练。

但是，如何进行这方面的工作却遇到了挑战。从目前的情况来看，至少有两个主要的困难：一是不论哪个专业都不可能投入较大量的教学时数来专门用于嵌入式系统的教学；二是掌握嵌入式系统的技术和设计方法对学生的基础要求较高，任何一个专业都不可能以嵌入式系统为培养目标设置该专业的基础课程（至少目前是这样）。因此作者认为，在目前的情况下，嵌入式系统的教学应该依托于各专业的现有课程体系，选准切入点，以学生初步掌握嵌入式系统的基本概念、基本理论、基本应用、基本设计方法和基本特点为教学目标，设置 1～2 门小型课较为合适。

作为嵌入式系统这种比较特殊的计算机系统，其基本构成仍然是硬件和软件两部分。对于硬件部分的教学，各专业可在原教学内容基础上根据嵌入式系统的硬件特点加以适当的修改和补充（这样不会增加过多的课时）。而软件部分（主要是嵌入式操作系统），作为计算机专业可以把原有的操作系统课程在内容上进行延伸，比如把一个实际的嵌入式操作系统（例如 μC/OS－II）当作一个实例来介绍，同时也避免了原来操作系统课程过于空洞的弊端。当然，也可以把嵌入式操作系统单独作为一门小课来讲授。因为计算机专业学生已学过操作系统课程，所以单独设置的这门小课的内容就可以适当丰富一些（比如，μCLinux 和 μC/OS－II）。

由于非计算机专业原来就没有操作系统课程，于是就应该单独设置一门嵌入式操作系统的课程。课程的内容应以系统简单、概念清晰且有较强实用性的嵌入式操作系统为宜。

本书就是为使学生了解和初步掌握嵌入式操作系统特点、体系结构及实际应用而编写的教学用书。

本书的特点及内容如下：

- 以源码公开的嵌入式实时操作系统 μC/OS－II 为核心，介绍了嵌入式操作系统在任务、任务的调度和管理、任务之间的通信和同步、内存管理等方面的实现和应用特点。
- 语言文字通俗易懂，尽量避免了大量程序源代码的剖析讲解，而代之以插图和例题，力求重点突出。
- 在 μC/OS－II 系统的移植的讲解方面，尽量避免读者可能不太熟悉的计算机硬件系统，从而冲击读者学习的重点，而以大多数读者都比较了解和熟悉的 51 系列单片机为硬件系统（尽管 μC/OS－II 在 51 单片机上应用有些嫌大），以使读者可以集中精力学习和掌握系统移植方面的知识和方法。另外，鉴于基于 ARM 核的嵌入式处理器在国内嵌入式系统应用占有相当大的份额，并且发展态势也越来越猛，因此本书也单独用了一章的篇幅介绍了 μC/OS－II 在 ARM 上的移植。
- 由于实验条件的限制，也为了学习操作系统这类软件的方便，大多数读者通常还是要以 PC 机为学习工具的，因此本书的例题几乎都可在 PC 机中调试和运行。另外，为了使读者更好地使用例题代码和了解 μC/OS－II 在 PC 机中的移植，作者把相关内容作为附录收到本书中。
- 本书的教学时数以不超过 30 学时为宜，其中应有 6 学时左右的实验和 8 学时左右的大型作业。
- 本书是为教学目的编写的教材，适用于高等院校计算机技术及应用、自动化、生产过程控制、仪器仪表、电子技术等专业高年级的教学，同时也可用于工程技术人员的自学和培训。
- 阅读和学习本书的内容，须在 C 语言程序设计、汇编语言程序设计和微型计算机原理（最好学习过单片机原理）等方面具有一定的基础。

全书共 9 章。参加本书编写的有房红征、李益民、刘德华、胡晓红和任哲。由任哲担任主编并负责全书的统稿工作。

在本书的策划和编写过程中，得到了很多同仁的关心、指导和帮助。特别是本书有关 μC/OS－II 移植的第 7 章和第 8 章分别采用了杨屹先生和周立功先生的方法和程序代码，另

外本书还用到了 Jean J. Labrosse 先生书中的部分例题代码。为此，向为本书做过贡献的人们表示衷心的感谢！

由于作者在嵌入式系统知识方面的欠缺，尤其是在嵌入式系统教学经验方面的不足，因此本书无论是在教学内容的选取编排上、例题的选用上，还是对于难点、重点的讲解上，一定会有很多不足甚至错误，在此诚挚希望读者批评指正。作者的电子邮箱为 renzhe71@sina.com。

另外，对 μC/OS-II 在 51 单片机上的移植感兴趣的读者可与杨屹先生沟通探讨，其网址为 www.armecos.com，电子邮箱为 asdjf@163.com。对 μC/OS-II 在 ARM 核上的移植感兴趣的读者可与周立功先生联系，其电子邮箱为 zlg3@zlgmcu.com。

作　者

2005 年 5 月 8 日

目　录

第 1 章　嵌入式实时操作系统的基本概念

1.1　计算机操作系统 …… 1
1.1.1　什么是计算机操作系统 …… 1
1.1.2　操作系统的作用和功能 …… 2
1.2　嵌入式系统和嵌入式操作系统 …… 3
1.2.1　嵌入式系统的基本概念 …… 4
1.2.2　嵌入式操作系统 …… 7
1.2.3　实时操作系统需要满足的条件 …… 9
1.2.4　嵌入式系统的任务及嵌入式实时操作系统 …… 11
1.3　嵌入式实时操作系统 μC/OS－II 简介 …… 12
1.4　小　结 …… 13
1.5　练习题 …… 14

第 2 章　μC/OS－II 中的任务

2.1　任务的基本概念 …… 15
2.1.1　任务的状态 …… 17
2.1.2　用户任务代码的一般结构 …… 18
2.1.3　系统任务 …… 20
2.1.4　任务的优先权及优先级别 …… 21
2.2　任务堆栈 …… 21
2.2.1　任务堆栈的创建 …… 22
2.2.2　任务堆栈的初始化 …… 24
2.3　任务控制块及任务控制块链表 …… 25
2.3.1　任务控制块的结构 …… 25
2.3.2　任务控制块链表 …… 26
2.3.3　任务控制块的初始化 …… 28
2.4　任务就绪表及任务调度 …… 29
2.4.1　任务就绪表的结构 …… 29

2.4.2 对任务就绪表的操作…… 31
2.4.3 任务的调度…… 32
2.5 任务的创建…… 37
2.5.1 用函数 OSTaskCreate()创建任务 …… 37
2.5.2 用函数 OSTaskCreateExt()创建任务…… 39
2.5.3 创建任务的一般方法…… 39
2.6 任务的挂起和恢复…… 46
2.6.1 挂起任务…… 46
2.6.2 恢复任务…… 47
2.7 其他任务管理函数…… 53
2.7.1 任务优先级别的修改…… 53
2.7.2 任务的删除…… 53
2.7.3 查询任务的信息…… 60
2.8 μC/OS－II 的初始化和任务的启动…… 60
2.8.1 μC/OS－II 的初始化…… 60
2.8.2 μC/OS－II 的启动…… 62
2.9 一个示例…… 64
2.10 小 结 …… 72
2.11 练习题 …… 73

第 3 章 μC/OS－II 的中断和时钟

3.1 μC/OS－II 的中断…… 74
3.1.1 μC/OS－II 的中断过程…… 74
3.1.2 中断级任务切换函数…… 78
3.1.3 应用程序中的临界段…… 78
3.2 μC/OS－II 的时钟…… 80
3.3 时间管理…… 88
3.3.1 任务的延时…… 88
3.3.2 取消任务的延时…… 89
3.3.3 获取和设置系统时间…… 93
3.4 小 结…… 96
3.5 练习题…… 97

第4章 任务的同步与通信

4.1 任务间的同步和事件控制块 …… 98
4.1.1 任务间的同步 …… 98
4.1.2 事 件 …… 99
4.1.3 事件控制块 …… 112
4.1.4 操作事件控制块的函数 …… 113
4.1.5 空事件控制块链表 …… 115
4.2 信号量及其操作 …… 115
4.2.1 信号量 …… 115
4.2.2 信号量的操作 …… 116
4.3 互斥型信号量和任务优先级反转 …… 124
4.3.1 任务优先级的反转现象 …… 124
4.3.2 互斥型信号量 …… 130
4.4 消息邮箱及其操作 …… 133
4.4.1 消息邮箱 …… 133
4.4.2 消息邮箱的操作 …… 133
4.5 消息队列及其操作 …… 139
4.5.1 消息队列 …… 139
4.5.2 消息队列的操作 …… 142
4.6 小 结 …… 148
4.7 练习题 …… 149

第5章 信号量集

5.1 信号量集的结构 …… 150
5.1.1 信号量集的标志组 …… 150
5.1.2 等待任务链表 …… 152
5.1.3 对等待任务链表的操作 …… 154
5.2 信号量集的操作 …… 155
5.2.1 创建信号量集 …… 155
5.2.2 请求信号量集 …… 156
5.2.3 向信号量集发信号 …… 157
5.2.4 查询信号量集的状态 …… 163
5.2.5 删除信号量集 …… 166

5.3 小 结 …… 167
5.4 练习题 …… 167

第 6 章 内存的动态分配

6.1 内存控制块 …… 168
6.1.1 可动态分配内存的划分 …… 168
6.1.2 内存控制块 OS_MEM 的结构 …… 169
6.1.3 空内存控制块链表 …… 170
6.2 动态内存的管理 …… 170
6.2.1 创建动态内存分区 …… 171
6.2.2 请求获得一个内存块 …… 173
6.2.3 释放一个内存块 …… 175
6.2.4 查询一个内存分区的状态 …… 179
6.3 小 结 …… 184
6.4 练习题 …… 184

第 7 章 在 51 单片机上移植 μC/OS－II

7.1 μC/OS－II 移植的一般性问题 …… 185
7.1.1 可重入函数 …… 185
7.1.2 时钟节拍的产生 …… 186
7.1.3 任务堆栈的设计 …… 186
7.2 在 51 系列单片机上移植 μC/OS－II …… 187
7.2.1 改写文件 OS_CPU.H …… 187
7.2.2 任务堆栈的设计 …… 188
7.2.3 文件 OS_CPU_C.C 的改写 …… 194
7.2.4 几点必须注意的事项 …… 196
7.3 应用举例 …… 196
7.3.1 LED 数码显示器的驱动程序 …… 196
7.3.2 51 单片机串行接口的应用 …… 198
7.4 小 结 …… 206
7.5 练习题 …… 206

第 8 章 在 ARM7 上移植 μC/OS－II

8.1 移植规划 …… 207

8.1.1 编译器的选择 …… 207
8.1.2 ARM7 工作模式的选择 …… 207
8.2 移 植 …… 208
8.2.1 文件 OS_CPU.H 的编写 …… 208
8.2.2 文件 OS_CPU_C.C 的编写 …… 210
8.2.3 文件 OS_CPU_A.S 的编写 …… 216
8.2.4 关于中断及时钟节拍 …… 219
8.3 移植 μC/OS-II 到 LPC2000 …… 221
8.3.1 挂接 SWI 软件中断 …… 222
8.3.2 中断及时钟节拍中断 …… 222
8.3.3 一个基于 μC/OS-II 和 ARM 的应用程序实例 …… 223
8.4 小 结 …… 225
8.5 练习题 …… 225

第 9 章 μC/OS-II 可剪裁性的实现

9.1 文件 OS_CFG.H 中用于系统裁剪的常量 …… 226
9.2 配置常量的解释 …… 228
9.2.1 OS_MAX_EVENTS …… 228
9.2.2 OS_MAX_MEM_PARTS …… 229
9.2.3 OS_MAX_QS …… 229
9.2.4 OS_MAX_MEM_TASKS …… 229
9.2.5 OS_LOWEST_PRIO …… 229
9.2.6 OS_TASK_IDLE_STK_SIZE …… 230
9.2.7 OS_TASK_STAT_EN …… 230
9.2.8 OS_TASK_STAT_STK_SIZE …… 230
9.2.9 OS_CPU_HOOKS_EN …… 230
9.2.10 OS_MBOX_EN …… 231
9.2.11 OS_MEM_EN …… 231
9.2.12 OS_Q_EN …… 231
9.2.13 OS_SEM_EN …… 231
9.2.14 OS_TASK_CHANGE_PRIO_EN …… 231
9.2.15 OS_TASK_CREATE_EN …… 231
9.2.16 OS_TASK_CREATE_EXT_EN …… 232
9.2.17 OS_TASK_DEL_EN …… 232

9.2.18 OS_TASK_SUSPEND_EN …… 232
9.2.19 OS_TICKS_PER_SEC …… 232

附录 A 文件 PC.C 中的函数

A.1 字符显示函数 …… 233
A.1.1 显示一个字符的函数 PC_DispChar() …… 233
A.1.2 清屏幕一列显示的函数 PC_DispClrCol() …… 233
A.1.3 清屏幕一行显示的函数 PC_DispClrRow() …… 233
A.1.4 清屏函数 PC_DispClrScr() …… 234
A.1.5 显示字符串函数 PC_DispStr() …… 234
A.1.6 颜色常量的定义 …… 234
A.2 保存和恢复 DOS 环境的函数 …… 235
A.2.1 保存 DOS 环境的函数 PC_DOSSaveReturn() …… 235
A.2.2 恢复 DOS 环境的函数 PC_DOSReturn() …… 235
A.3 设置和获取中断向量的函数 …… 235
A.3.1 设置中断向量的函数 PC_VectSet() …… 235
A.3.2 获取中断向量的函数 PC_VectGet() …… 235

附录 B μC/OS－II 中使用的数据类型

附录 C μC/OS－II 在 80x86 上的移植

C.1 任务切换函数 OSCtxSw() …… 238
C.1.1 任务切换函数 OSCtxSw() …… 238
C.1.2 任务切换宏 OS_TASK_SW() …… 239
C.1.3 中断级任务切换函数 OSIntCtxSw() …… 240
C.2 μC/OS－II 的系统时钟 …… 242
C.2.1 PC 中 DOS 的系统时钟 …… 242
C.2.2 PC 中 μC/OS－II 的系统时钟 …… 242
C.2.3 μC/OS－II 系统时钟中断服务程序 …… 244
C.2.4 在任务中安装 μC/OS－II 系统时钟的中断向量 …… 244
C.2.5 在任务中由 μC/OS－II 返回 DOS 环境 …… 245

附录 D C51 开发工具 μVision2 简介

D.1 C51 语言的扩展 …… 246

D.1.1　数据类型 …… 246
D.1.2　存储器类型 …… 247
D.1.3　存储模式 …… 248
D.1.4　指　针 …… 248
D.1.5　可重入函数 …… 250
D.1.6　与汇编语言的接口 …… 251
D.1.7　库函数 …… 251
D.2　创建项目 …… 252
D.2.1　启动 μVision2 并创建一个项目 …… 252
D.2.2　新建一个源文件 …… 252
D.2.3　增加和配置初始化代码 …… 253
D.2.4　为目标设置工具选项 …… 254
D.2.5　创建项目并生成 HEX 文件 …… 254
D.3　常用的菜单选项 …… 254
D.3.1　视图菜单 View …… 254
D.3.2　项目菜单 Project …… 255
D.3.3　调试菜单 Debug …… 255
D.3.4　外围器件菜单 Peripherals …… 256

附录 E　MagicARM2200 教学实验开发平台简介

参考文献 …… 260

第1章 嵌入式实时操作系统的基本概念

操作系统(Operating System,OS)是一种系统软件。它在计算机硬件与计算机应用程序之间,通过提供应用程序接口(Application Programming Interface, API),屏蔽了计算机硬件工作的一些细节,从而使应用程序的设计人员得以在一个友好的平台上进行应用程序的设计和开发,大大提高了应用程序的开发效率。

嵌入式系统作为一种计算机系统,当然也需要一个合适的操作系统的支持,这种应用于嵌入式系统中的操作系统就叫做嵌入式操作系统。

本章的主要内容有:

- 计算机操作系统的基本概念;
- 嵌入式系统的基本概念;
- 实时操作系统的概念。

1.1 计算机操作系统

1.1.1 什么是计算机操作系统

众所周知,计算机是一种功能强大的数字运算装置。作为一种装置,它需要由诸如中央微处理器(CPU)、存储器、接口及外部设备等一些实际物理装置来构成。这些构成计算机的实际物理装置,就是计算机的硬件系统。

只由硬件构成的计算机叫做"裸机"。这种"裸机"是不能工作的,计算机必须在硬件的基础上配以相应的软件才能构成真正的计算机系统,才能完成人们所交付的各种计算任务。如果用人来做比喻的话,计算机的硬件就相当于人的骨骼、肌肉等看得见摸得着的实体,而计算机的软件就相当于人头脑中存储的思想、方法等看不见摸不着的东西。显然,没有思想的人不能叫做一个"活人",或者不能叫做一个"真正的人",这样的人是什么工作也做不了的。

现在人们对计算机软件是不会很陌生了,因为经常要使用 Microsoft Word 来书写和编辑文档,也经常使用 Microsoft Excel 做一些数据的统计工作。但是,当在计算机上安装这些软件时会被告知,必须查看计算机上是否装有 Windows,因为计算机上如果没有安装 Windows,

那么无论是 Microsoft Word 还是 Microsoft Excel 都是不能运行的。这里的 Windows 就是一种计算机操作系统，要使用的 Microsoft Word、Microsoft Excel 等软件必须以 Windows 操作系统为平台来运行；或者说 Microsoft Word、Microsoft Excel 等软件是通过 Windows 来使计算机硬件工作的。因此，一般计算机上都要安装某种操作系统。

在计算机系统中，操作系统负责对计算机系统的几类主要资源，如处理器、存储器、输入/输出设备、数据与文档资源、用户作业等进行管理，并向计算机用户提供若干服务。通过这些服务，将计算机硬件的复杂操作隐藏起来，从而为应用程序提供一个透明的操作环境。

计算机的操作系统、硬件与应用程序之间关系如图 1－1 所示。

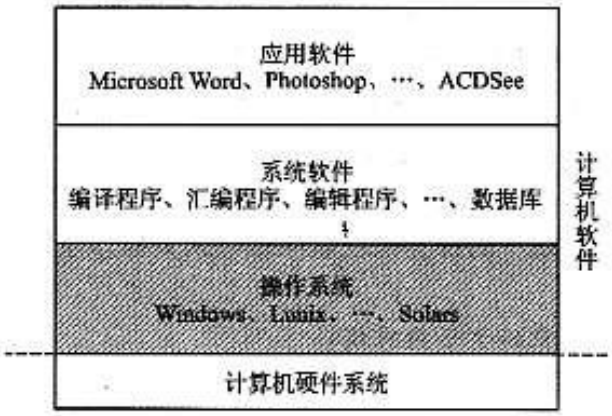


图 1－1　操作系统在计算机中的地位

从图 1－1 中可看到，一个完整的计算机系统是由硬件和软件两部分组成的。硬件是所有软件运行的物质基础，软件能充分发挥硬件的潜能和扩充硬件的功能，两者互相依存，缺一不可。

从图 1－1 中还可看到，在计算机系统中的硬件和软件是分层次的。下层是上层运行的基础，上层是下层功能的扩充；下层对上层隐藏了下层功能实现的细节，只对上层提供了使用这些功能的接口。操作系统层通常是最靠近硬件层的软件层，主要完成计算机硬件资源的调度和分配、信息的存储和保护以及并发活动的协调和控制等许多工作。

1.1.2　操作系统的作用和功能

1. 操作系统的作用

从应用程序设计人员的角度来看，操作系统是计算机硬件系统与应用程序之间的接口。应用程序设计人员只是以操作系统层为基础来使用计算机系统，所以程序设计人员看到和使用的只是一些由计算机操作系统所提供的函数。至于操作系统的这些函数是如何去控制硬件的，作为一个应用程序的设计人员，就可以完全不去管它了。因此，应用程序的设计人员可以专心地解决自己的应用问题而不必关心计算机的硬件如何工作。也可以说，计算机操作系统是计算机硬件的一个软件包装，它为应用程序设计人员提供了一个更便于使用的虚拟计算机(Virtual Machine)。又因为计算机的硬件系统及其他供应用程序使用的资源是靠计算机操作系统来管理的，所以它也可看作是计算机系统资源的管理者。

总之，计算机的操作系统为应用程序提供了一个界面友好，性能稳定、安全，效率高，操作方便的虚拟计算机。

2. 操作系统的功能

操作系统之所以可看成是应用程序与硬件之间的接口或者是虚拟机，其原因就在于其功能主要是对计算机资源进行管理。从这个角度来看，操作系统的主要功能如下：

- 处理器的管理。操作系统对处理器的管理主要有两项工作：一是对中断的管理；二是对处理器的工作进行调度。

  因为处理器硬件只能发现外部事件的中断申请，而不能对中断进行管理和处理，所以对中断的管理和处理只能由操作系统来承担。

  现代计算机应用程序大多是多道程序结构，那么一个处理器如何来运行这多道程序，就是一个较为复杂的问题。它要求操作系统应该能按照某种规则对处理器的使用进行合理的分配，这样才能使多道应用程序协调有秩地运行，这就是操作系统应该具有的处理器调度功能。
- 存储的管理。存储器是计算机的重要资源，如何合理地分配和使用该资源，是计算机操作系统责无旁贷的。
- 设备的管理。计算机系统一般都配有外部设备，因此计算机操作系统还必须有对这些外部设备管理的功能，以便完成用户提出的I/O请求，加快输入/输出的速度，提高I/O设备的利用率。当然，作为一个完善的计算机操作系统，还要提供外部设备的驱动程序。
- 文件的管理。在计算机中，程序和数据通常都以文件的形式存储在外存（例如硬盘、光盘等）上。由于在这些外存中文件量极其巨大，如果对它们没有良好的管理方式，就会导致严重的后果。为此，计算机操作系统必须具备文件管理功能。
- 网络和通信的管理。使用网络的计算机除了需要配备连网硬件之外，其操作系统还必须具有网上资源管理、网络通信、故障管理、安全管理及性能管理等网络功能。
- 提供用户接口。计算机操作系统除了提供以上提及的各项功能外，还要为用户提供良好使用上述功能的接口，以便用户能方便地使用操作系统的功能，从而能有效地组织作业及其工作，并使系统高效地运行。

1.2 嵌入式系统和嵌入式操作系统

随着计算机技术的进步，在工业生产及人们的日常生活中有很多设备和装置中都内置了计算机系统，使这些设备或装置具有很高的自动化性能和某种程度的智能性，从而极大地满足了人们生产和生活的需要。目前，随着这种应用的迅速普及，一项被称为“嵌入式系统”的技术就应运而生了。

1.2.1 嵌入式系统的基本概念

1. 什么是嵌入式系统

继桌面计算机之后，最重要的 IT 技术产业莫过于嵌入式系统了。

自 1946 年电子计算机诞生以来，在相当长的一个时期里，计算机始终是一种供养在技术要求极高的、价格极其昂贵的特殊机房中实现数值计算的大型昂贵设备。直到 20 世纪 70 年代，由于微处理器的出现，才使计算机的应用出现了历史性的变化。以微处理器作为核心的微型计算机，以其小型、价廉、高可靠性及具有高速数值解算能力的特点，迅速引起了自动控制领域专业人员的极大兴趣和关注。因为把微型机嵌入到一个对象体系（如汽车、火箭等）中，可以很方便地实现这个对象体系的智能化控制。例如，将微型计算机经电气加固、机械加固，并配置各种外围接口电路之后，安装到飞机中，就可以构成智能化程度很高的自动驾驶仪或发动机状态监测系统；安装到洗衣机中，就可以使洗衣机根据所要洗涤衣物的具体状况而自动采取不同的洗涤模式，从而提高洗衣的效率和效果；安装到照相机中，可以自动对摄像的各技术参数进行设置，从而可以获得质量更高的照片；安装到音像设备中，可以获得高保真的音响和影像等。

显而易见，这种用途的计算机系统在一定程度上改变了通用计算机系统的形态与通用计算机系统的功能。为了区别于原有的通用计算机系统，人们把嵌入到对象体系中，为实现对象体系智能化控制的计算机系统，称作嵌入式计算机系统，简称嵌入式系统。

目前，嵌入式系统的应用技术已成为通信和消费类产品的共同发展方向。在现代生活中，几乎所有电器设备上都有嵌入式系统的应用，例如掌上 PDA、电视机的机顶盒、移动电话、数字电视、汽车、微波炉、数码相机、家庭自动化系统、电梯、空调、安全系统、自动售货机、蜂窝式电话、工业自动化仪表及医疗仪器等。显然，嵌入式系统的这些应用对于提高人们的生活质量做出了极大的贡献，同时也深刻地影响了人们的生活方式。

对于什么是嵌入式系统，从不同的角度有不同的定义。一个较为通用的定义为：嵌入式系统是对对象进行自动控制而使其具有智能化并可嵌入对象体系中的专用计算机系统。“嵌入性”、“专用性”与“计算机系统”是嵌入式系统的三个基本要素。这里所说的对象系统就是指上面所提到的移动电话、数字电视、汽车、舰船、火箭、PDA、洗衣机、医疗设备、工业自动生产线等。对象系统也叫做嵌入式系统的宿主对象系统。图 1－2 所示为嵌入式系统的部分应用。

2. 嵌入式系统的发展历程

由于嵌入式计算机系统要嵌入到对象体系中，实现的是对象的智能化控制，因此，它有着与通用计算机系统完全不同的技术要求与技术发展方向。

通用计算机系统的技术要求是高速、海量的数值计算，技术发展方向是总线速度的无限提升、存储容量的无限扩大等；而嵌入式系统的技术要求则是对象的智能化控制能力，技术发展方向是与对象系统密切相关的嵌入性能、控制能力与控制的可靠性等。

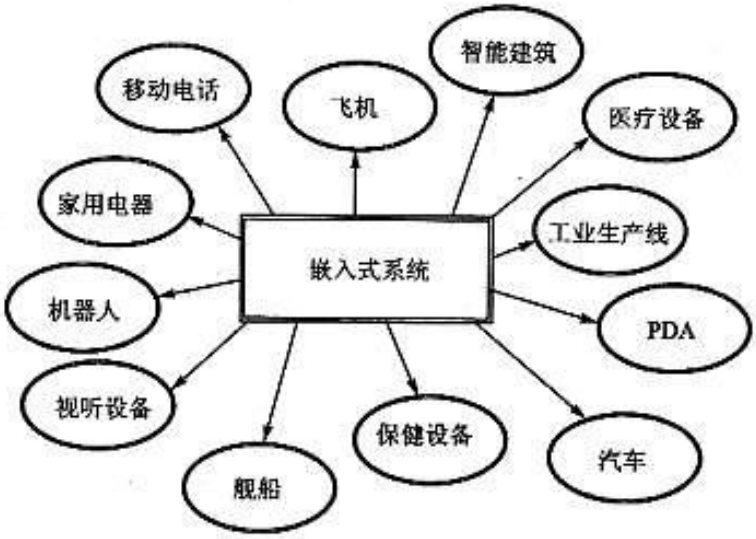


图 1-2 嵌入式系统的部分应用

嵌入式系统起源于微型计算机时代。微型计算机小巧的体积、低廉的价格、良好的可靠性、强大的计算能力，诱使人们把它装置在机器设备内来完成对机器设备的控制工作，从而出现了专门为工业控制定制的工业控制机(简称为工控机)。虽然工控机的出现为计算机控制技术的发展做出了不可磨灭的贡献，但随着时代的进步，基于通用计算机体系结构的微型计算机在体积、价位、可靠性方面都日渐无法满足广大对象系统日益增长的要求。因此，嵌入式系统就自然而然地走上了一个独立的发展道路——系统芯片化道路，力求将 CPU 与包括存储器、接口在内的计算机系统集成在一个芯片上，从而开创了嵌入式系统独立发展的新时代。

嵌入式系统独立发展的初期是单片机时代。在探索单片机的发展道路时，有过两种模式："Σ 模式"与"创新模式"。"Σ 模式"将通用计算机系统中的基本单元根据应用的需要进行裁剪后，集成在一个芯片上，构成单片微型计算机；"创新模式"则要在体系结构、微处理器、指令系统、总线方式、管理模式等方面完全按嵌入式应用要求设计全新的、满足嵌入式应用要求的芯片。可以说，Intel 公司的 MCS-48、MCS-51 就是按照创新模式发展起来的单片形态的嵌入式系统。历史证明，"创新模式"是嵌入式系统独立发展的正确道路，51 单片机的体系结构也因此成为嵌入式系统的一种主要的典型结构体系。

单片形态的嵌入式系统硬件自 20 世纪 70 年代末以来，大体上可分为 MCU、SoC 两个阶段。

微控制器(Micro Controller Unit，MCU)阶段。主要的技术发展方向是：不断地在一个芯片上扩展满足宿主对象系统所要求的各种外围电路与接口电路(例如并行接口、串行接口、定时器等)，以增强其对宿主对象的智能化控制能力。典型的产品就是 Intel 公司的 51 系列单片机等。

单片系统(System on Chip,SoC)阶段。发展 SoC 的重要动因,就是人们不断地寻求应用系统在芯片上的最大化解决方案。因此,在通用串行接口(USB)、数字信号处理器(DSP)、TCP/IP 通信模块、GPRS 通信模块、蓝牙模块接口等功能模块出现之后,人们又根据应用的需要把这些功能模块与 MCU 进行有机的结合,制造出集成度更高的系统级的芯片,这种芯片就叫做 SoC 系统。目前,随着现代微电子技术、IC 设计、EDA 工具的飞速发展,基于 SoC 的单片应用系统正在成为嵌入式系统的主流器件。

除了上面所说的单片型系统,在这里不能不提的是,在使用上述单片型系统作为嵌入式系统硬件的同时,也经常(特别是在嵌入式系统应用的初期)根据实际具体应用使用通用型的微处理器来实现嵌入式系统。不过,目前说到嵌入式系统的硬件时,通常指的是单片型系统芯片。

表 1－1 按在单一指令内所能处理数据的位数列出了嵌入式系统常用的芯片。

表 1－1 嵌入式系统的常用芯片

位 数	4	8	16	32	64
单片型系统	TMS1000 COPS	8048/49/50 8051/52 6801/04/05	8096/97 68200	ARM RISC Core MIPS 32Bits RISC Core	MIPS 64Bits RISC Core
通用微处理器	4004 4040	Z8 8085 6809 Z80 6502 6802	8086 80186 80286 Z8000	80386 80486 68000/10/20/30/40 32032	Pentium II/III

各种不同位数的嵌入式系统微处理器的应用层面也有明显的区分,位数越高的嵌入式微处理器,其数据处理能力及其附加值也越高。各种不同位数的微处理器应用分类情况见表 1－2。

表 1－2 不同位数的嵌入式微处理器的应用范围

微处理器的位数	应用的典型产品
4	计算机、遥控器、照相机、电话、CD 随身听、防盗器、玩具、充电器、各种简易测量仪表、计量仪表…
8	电动机控制器、电视游戏机、空气调节器、计算机显示器、传真机、BP 机、电话录音机…
16	手机、摄像机、录放机、各种多媒体设备…
32	调制解调器、掌上计算机、网络路由器、工作站、激光打印机、彩色传真机、数码相机、地球卫星定位系统…
64	高端工作站、新型电视游乐器、高级图像处理设备、高级音响处理设备、气象数据处理设备…

3. 嵌入式系统的特点

从嵌入式系统的构成上看，嵌入式系统是集软硬件于一体的、可独立工作的计算机系统；从外观上看，嵌入式系统像是一个“可编程”的电子“器件”；从功能上看，它是对宿主对象进行控制，使其具有“智能”的控制器。

嵌入式系统的硬件部分包括处理器/微处理器、存储器及外设器件和 I/O 端口、图形控制器等。这种系统有别于一般的计算机处理系统，比如它通常不使用像硬盘那样大容量的存储介质，而大多使用 EPROM、EEPROM 或闪存(Flash Memory)作为存储介质。

嵌入式系统的软件包括操作系统软件和应用软件。操作系统一般应该具有较强的实时性，并可以对多任务进行管理，而应用软件都是一些专门性很强的应用程序。

嵌入式计算机系统与通用型计算机系统相比，具有以下特点：

- 专用性强。嵌入式系统通常是面向某个特定应用的，所以嵌入式系统的硬件是为特定用户群来设计的，它通常都具有某种专用性的特点。
- 可裁剪性好。嵌入式系统的硬件和操作系统都必须设计成可裁剪的，以便用户可根据实际应用需要量体裁衣，去除冗余，从而使系统在满足应用要求的前提下达到最精简的配置。
- 实时性与可靠性好。嵌入式系统中的软件一般不是存储于磁盘等载体中，而都固化在存储器芯片或单片系统的存储器里，再加上精心设计的嵌入式操作系统，从而可以快速地响应外部事件，同时也大大提高了系统的可靠性。
- 功耗低。由于嵌入式系统中的软件一般不是存储于磁盘等载体中，而都固化在存储器芯片或单片系统的存储器之中，所以它具有功耗低的特点，从而便于它应用在飞机、舰船、数码相机等移动设备中。

1.2.2 嵌入式操作系统

1. 什么是嵌入式操作系统

由于硬件的限制，在使用 MCU 设计嵌入式系统的初期，程序设计人员得到的是只有硬件系统的“裸机”，没有任何类似于操作系统的软件作为开发平台，对 CPU、RAM 等这些硬件资源的管理工作都必须由程序员自己编写程序来解决，从而使程序员工作得十分辛苦，并且使应用程序的开发效率极低，所以那时从事嵌入式系统开发的人员就期望能有一个支持嵌入式系统开发的系统软件。

现在，由于技术的进步，单片系统硬件的规模越来越大，功能越来越强，从而给运行嵌入式操作系统提供了物质保证，于是就出现了很多具有不同应用特点的操作系统。

运行在嵌入式硬件平台上，对整个系统及其所操作的部件、装置等资源进行统一协调、指挥和控制的系统软件就叫做嵌入式操作系统。由于嵌入式操作系统的硬件特点、应用环境的多样性和开发手段的特殊性，使它与普通的操作系统有着很大的不同，其主要特点如下：

- 微型化。嵌入式系统芯片内部存储器的容量通常不会很大(1 MB 以内)，一般也不配置外存，加上电源的容量较小(常常用电池甚至微型电池供电)以及外部设备的多样化，因而不允许嵌入式操作系统占用较多的资源，所以在保证应用功能的前提下，嵌入式操作系统的规模越小越好。
- 可裁剪性。嵌入式操作系统运行的硬件平台多种多样，其宿主对象更是五花八门，所以要求嵌入式操作系统中提供的各个功能模块可以让用户根据需要选择使用，即要求它具有良好的可裁剪性。
- 实时性。目前，嵌入式系统广泛应用于生产过程控制、数据采集、传输通信等场合，这些应用的共同特点就是要求系统能快速响应事件，因此要求嵌入式操作系统要有较强的实时性。
- 高可靠性。嵌入式系统广泛应用于军事武器、航空航天、交通运输、重要的生产设备领域，所以要求嵌入式操作系统必须有极高的可靠性，对关键、要害的应用还要提供必要的容错和防错措施，以进一步提高系统的可靠性。
- 易移植性。为了适应多种多样的硬件平台，嵌入式操作系统应可在不做大量修改的情况下稳定地运行于不同的平台。

嵌入式操作系统与嵌入式系统的宿主对象的要求密切相关。按嵌入式操作系统的应用范围划分，可分为通用型嵌入式操作系统和专用型嵌入式操作系统。通用型嵌入式操作系统可用于多种应用环境，例如常见的 Windows CE、VxWorks、μCLinux 及本书将要介绍的μC/OS 等；专用型嵌入式操作系统则用于一些特定的领域，例如移动电话的Symbian、手持数字设备(PDA)的 Plam OS 等。

由于嵌入式系统存储器的容量较小，因此嵌入式系统的软件一般只有操作系统和应用软件两个层次。嵌入式操作系统在系统中的地位如图 1－3 所示。

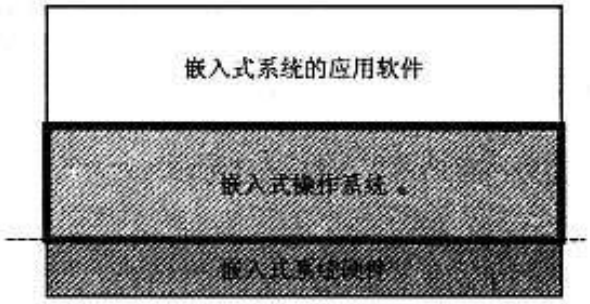


图 1－3　嵌入式操作系统在嵌入式系统中的地位

按对外部事件的响应能力来分类，嵌入式操作系统有实时操作系统和分时操作系统两类。

2. 实时操作系统

什么是实时？实时含有立即、及时之意。如果操作系统能使计算机系统及时响应外部事件的请求，并能及时控制所有实时设备与实时任务协调运行，且能在一个规定的时间内完成对事件的处理，那么这种操作系统就是一个实时操作系统(Real Time Operation System，RTOS)。

对实时系统有两个基本要求：第一，实时系统的计算必须产生正确的结果，称为逻辑或功

能正确(Logical or Functional Correctness);第二,实时系统的计算必须在预定的周期内完成,称为时间正确(Timing Correctness)。

按时间正确的程度来分,实时操作系统又分为硬实时操作系统和软实时操作系统两种。如果要求系统必须在极严格的时间内完成实时任务,那么这样的系统就叫做硬实时操作系统。例如,图 1-4 所示为硬实时系统的一个实例。为了防止航行中的舰船发生触礁事故,在舰船的头部装有用来发现礁石的声纳,声纳信号输入到嵌入式系统中,由嵌入式系统来控制舵机的动作。现在假设舰船前面出现礁石而不采取任何规避动作时,舰船会在 10 min 后与礁石相撞,而舵机完成合理的规避动作则需要 8 min。这就是说,从声纳发现礁石起到舵机开始动作,留给嵌入式系统用来计算控制舵机做出合理动作所需的数据的时间不能超过 2 min,并且计算结果应该正确无误;否则,后果将是灾难性的。

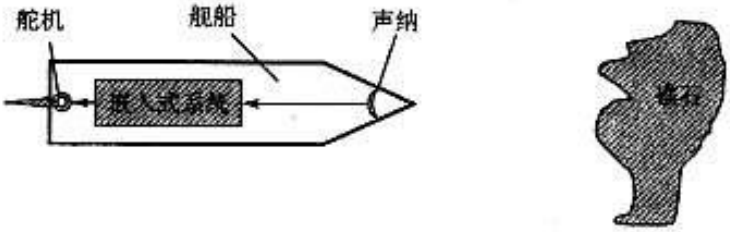


图 1-4 硬实时系统的实例

对于硬实时系统来说,超过截止时间计算出来的正确结果和错误的计算结果都是不能容忍的,因为事故已经发生了,结果再正确也没有什么用途了。

相对来说,如果系统完成实时任务的截止时间要求不是十分严格,那么这种系统就叫做软实时系统。也就是说,软实时系统对于计算超时具有一定的容忍度,超过允许计算时间得到的运算结果不会完全没有用途,只是这个结果的可信度要有某种程度的降低。

通过上面的叙述可知,一个系统的实时性除了需要硬件的保证之外,还需要操作系统的保证,即无论在什么情况下,操作系统完成任务所用的时间应该是在应用程序设计时就可预知的。

3. 分时操作系统

如果操作系统可按管理的任务数把 CPU 分成若干个时间片,将每个时间片分配给一个任务,CPU 按时间片轮流执行这些任务,那么这种操作系统就叫做分时操作系统。

1.2.3 实时操作系统需要满足的条件

嵌入式系统主要是对设备和装置进行控制的,在这些应用场合中,系统是否能及时快速地响应外部事件,常常是对系统的第一要求,因此嵌入式系统使用的操作系统大多是实时操作系统。对于一个实时操作系统来说,就是要求它在现有的硬件条件下,在接收输入后要尽可能快地计算出输出结果,并应使应用程序设计者在应用程序设计时,就能预先准确地确定完成任务

所需要的最长时间。

为达到上述要求，实时操作系统应满足以下三个条件：

- 实时操作系统必须是多任务系统；
- 任务的切换时间应与系统中的任务数无关；
- 中断延时的时间可预知并尽可能短。

1. 多任务

计算机在执行应用程序时，经常要用 I/O 设备进行数据的输入和输出，而 I/O 设备在工作时总是需要一段时间的。于是在 I/O 设备工作期间，如果 CPU 没有其他任务，那么就只能等待，因此就会使计算机运行一个应用程序所花的时间比较长，也就是说，这种系统的实时性较差。

如果把一个大的任务分解成多个可并行运行的小的任务，那么在一个任务需要等待 I/O 时，就可以交出对 CPU 的使用权，而让 CPU 去运行其他任务，这样就可以大大提高 CPU 的利用率。当然，系统完成任务所花的时间就会大大减少，从而给提高系统的实时性能创造了条件。

除此之外，多任务系统还带来了另外一个优点，即它可以让程序员把一个大的应用程序分成相对独立的多个任务来完成，从而给应用程序的设计和维护也提供了极大的方便。

由于多任务的诸多优点，因此现在的嵌入式实时操作系统都是多任务系统。

2. 内核的类型

由于嵌入式系统中只有一个 CPU，因此在一个具体时刻只能允许多个任务中的一个任务使用 CPU。根据系统中的任务获得使用 CPU 的权力的方式，多任务实时操作系统的内核分为可剥夺型和不可剥夺型两种类型。但无论在哪种类型的内核中，每个任务都必须具有一个惟一的优先级别来表示它获得 CPU 的权力。

不可剥夺型内核也叫做合作型多任务内核。在这种内核中，总是优先级别高的任务最先获得 CPU 的使用权。为防止某个任务始终霸占 CPU 的使用权，这种内核要求每个任务必须能主动放弃 CPU 的使用权。

在可剥夺型内核中，CPU 总是运行多个任务中优先级别最高的那个任务，即使 CPU 正在运行某个低优先级别的任务，当有高优先级别的任务准备就绪时，该高级别的任务就会剥夺正在运行任务的 CPU 使用权，而使自己获得 CPU 的使用权。

由于可剥夺型内核实时性较好，所以目前大多数嵌入式实时操作系统是可剥夺型内核。

3. 任务的切换时间

既然是多任务系统，那么就有任务之间的切换，操作系统的调度器就是做这项工作的。调度器在进行任务切换时当然需要一段时间，因此这段时间的长短也是影响系统实时性的一个重要因素。为了使应用程序的设计者可以计算出系统完成某一个任务的准确执行时间，所以

要求作为进行任务切换的调度器的运行时间应该是固定的，即调度器进行任务切换所用的时间不能受应用程序中其他因素(例如任务数目)的影响。

4. 中断延时

外部事件的发生常常以一个中断申请信号的形式来通知 CPU，然后才运行中断服务程序来处理该事件。自 CPU 响应中断到 CPU 转向中断服务程序之间所用的时间叫做中断延时。显然，中断延时要影响系统的实时性。因此，缩短中断延时也是实时操作系统需要解决的一项课题。

1.2.4 嵌入式系统的任务及嵌入式实时操作系统

1. 嵌入式系统的任务

由于嵌入式系统完成的是对一个装置或设备的控制任务，任务的功能相对固定，因此在一般情况下嵌入式实时操作系统支持的典型任务应该是一个无限循环结构。一个用 C 语言编写的任务的结构如例 1-1 所示。

例 1-1 一个用 C 语言编写的任务代码。

```
void mytask(void * pdata)
{
    for (;;)
    {
        用户编写的代码;
    }
}
```

由例 1-1 可以看到，从任务的代码来看，任务实质上就是一个返回类型为 void 的函数，并在函数的无限循环中完成用户的工作。那么，用户应用程序如何来响应用户的一些外部异步事件呢？当然要使用中断技术，并在中断服务程序中处理这些异步事件。嵌入式系统任务的典型结构如图 1-5 所示。

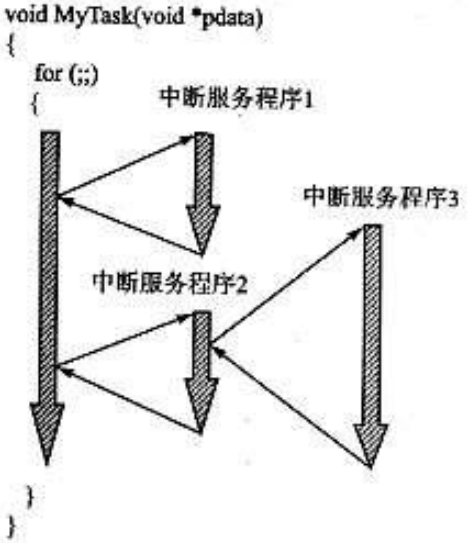


图 1-5 嵌入式系统任务的典型结构

现代的实时操作系统都支持多个任务，例如本书介绍的 μC/OS-II 就可支持多达 64 个任务。

综上所述，用于嵌入式系统，对系统资源和多个任务进行管理，且具有高可靠性、良好可裁剪性等优良性能的，为应用程序提供运行平台和实时服

务的微型系统软件叫做嵌入式实时操作系统。

2. 嵌入式实时操作系统

目前，嵌入式操作系统主要都以提供“微内核”为主，其他诸如窗口系统界面、文件管理模块、通信协议等还要由开发人员自己设计或者外购。大多数嵌入式操作系统主要提供三项服务来辅助应用程序设计人员。它们分别是：

- 内存管理。内存管理主要是动态内存的管理。当应用程序的某一部分需要使用内存时，可利用操作系统所提供的内存分配函数来获得足够的内存空间；一旦使用完毕，可调用系统提供的释放内存的函数，把曾经使用的内存空间还给系统，这样就使内存可以重复利用。
- 多任务管理。嵌入式实时操作系统应该提供丰富的多任务管理函数，以使程序设计人员设计多线程的应用程序。通常，嵌入式实时操作系统都会提供良好的任务调度机制，控制任务的启动、运行、暂停和结束等状态。通常这些调度算法是满足实时性要求的，也就是能使任务运行时的每个动作都会在一个严格要求的时间内执行完毕。
- 外围资源管理。一个完整的嵌入式应用系统，除了系统本身的微处理器、内存之外，还必须有多种外围设备的支持，例如键盘、显示装置、通信端口及外接的控制器等。这些外围设备都是系统中的各个任务可能用到的资源。由于资源有限，因此操作系统必须对这些资源进行合理的调度和管理，才能保证每个要使用资源的任务在运行时可获得足够的资源。

图 1－6　嵌入式操作系统的三项功能

3. 嵌入式操作系统的现状

IT 行业从来不是一个平静的世界，面对嵌入式系统的巨大应用领域和诱人的前景，世界上的各大软件开发公司、厂商甚至个人都纷纷开发出各具特色的嵌入式操作系统。

目前比较常见的嵌入式操作系统有 WindRiver 公司的 VxWorks、pSOS，微软公司的 Windows CE，QNX 公司的 QNX OS，在手持设备嵌入式操作系统中三分天下的 Plam、WinCE、EPOC 等，但是使用这些商业操作系统是需要高昂的费用的。面对这种情况，一些组织和个人也开发了一些免费的、源码开放的操作系统，在互联网发布，其中比较有名的是 μCLinux 和 μC/OS－II。

1.3　嵌入式实时操作系统 μC/OS－II 简介

μC/OS－II 是由 Jean J. Labrosse 于 1992 年编写的一个嵌入式多任务实时操作系统。最早这个系统叫做 μC/OS，后来经过近 10 年的应用和修改，在 1999 年 Jean J. Labrosse 推出了

μC/OS－II，并在 2000 年得到了美国联邦航空管理局对用于商用飞机的、符合 RTCA DO－178B 标准的认证，从而证明 μC/OS－II 具有足够的稳定性和安全性。

μC/OS－II 是用 C 语言和汇编语言编写的。其中绝大部分代码都是用 C 语言编写的，只有极少部分与处理器密切相关的代码是用汇编语言编写的，所以用户只要做很少的工作就可把它移植到各类 8 位、16 位和 32 位嵌入式处理器上。

由于 μC/OS－II 的构思巧妙，结构简洁精练，可读性很强，同时又具备了实时操作系统的全部功能；所以虽然它只是一个内核，但非常适合初次接触嵌入式实时操作系统的学生、嵌入式系统开发人员和爱好者学习，并且通过适当地扩展之后，还可应用到实际系统中去。

μC/OS－II 的体系结构如图 1－7 所示。

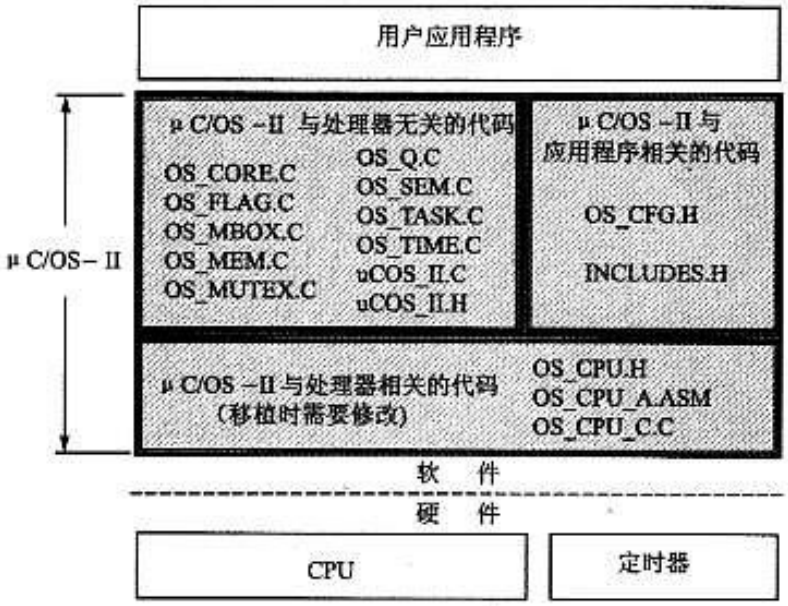


图 1－7　μC/OS－II 体系结构

如果读者希望了解关于 μC/OS－II 的更多信息，可以访问 Jean J. Labrosse 先生的网站 http://www.uCOS－II.com。

1.4　小　结

- 计算机操作系统是计算机硬件的一个软件包装，它为应用程序设计人员提供了一个更便于使用的虚拟计算机（Virtual Machine）。又由于计算机的硬件系统及其他供应用程序使用的资源是靠计算机操作系统来管理的，所以它也可看作是计算机系统资源的管理者。
- 嵌入到对象体系中，为实现对象体系智能化控制的计算机系统，称作嵌入式计算机

系统。

- 实时操作系统必须是多任务系统，任务的切换时间应与系统中的任务数无关，并且中断延时的时间应该可预知并尽可能短。

1.5 练习题

1. 什么是计算机的操作系统？它应该具备什么功能？
2. 简述嵌入式系统与普通操作系统的区别。
3. 观察人们日常生活中嵌入式系统的应用。
4. 什么是实时系统？试列举几个日常生活中的实时系统。

任务数无关，并且中

第2章 μC/OS－II 中的任务

μC/OS－II 操作系统内核的主要工作就是对任务进行管理和调度。弄清楚什么是任务、任务的结构和 μC/OS－II 对任务的管理方法，对于理解 μC/OS－II 的体系结构无疑是极其重要的。

本章的主要内容有：

- 任务的基本概念，用户任务和系统任务；
- 任务代码、任务控制块和任务堆栈；
- 任务的优先权及表示任务优先权的参数——优先级别；
- 任务就绪表结构及其操作；
- 任务切换及任务调度；
- 任务的创建、删除、挂起、恢复和查询；
- μC/OS－II 的初始化和启动。

2.1 任务的基本概念

人们在实际生活中处理一个大而复杂问题时，惯用的方法就是"分而治之"，即把一个大问题分解成多个相对简单、比较容易解决的小问题，小问题逐个被解决了，大问题也就随之解决了。同样，在设计一个较为复杂的应用程序时，也通常把一个大型任务分解成多个小任务，然后在计算机中通过运行这些小任务，最终达到完成大任务的目的，这种做法也使应用程序的维护变得容易起来。因此，现代操作系统几乎都是多任务操作系统。

在 μC/OS－II 中，与上述小任务对应的程序实体就叫做"任务"（实质上是一个线程）。μC/OS－II 就是一个能对这些小任务的运行进行管理和调度的多任务操作系统。

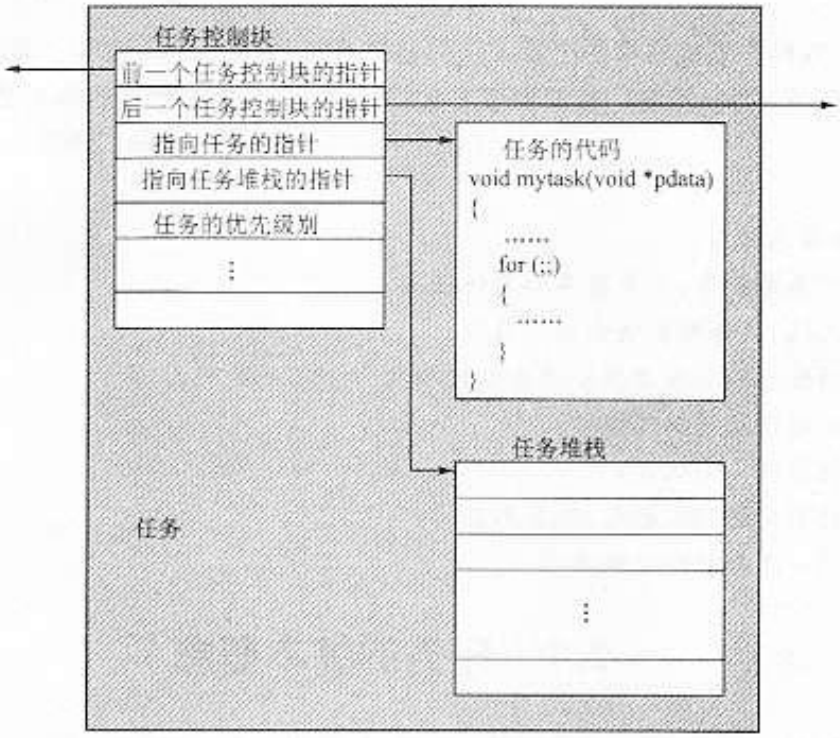
从应用程序设计的角度来看，μC/OS－II 的任务就是一个线程，就是一个用来解决用户问题的 C 语言函数和与之相关联的一些数据结构而构成的一个实体。

从任务的存储结构来看，μC/OS－II 的任务由如图 2－1(a)所示的三个组成部分构成：任务程序代码、任务堆栈和任务控制块。其中，任务控制块用来保存任务属性；任务堆栈用来保存任务工作环境；任务程序代码是任务的执行部分。

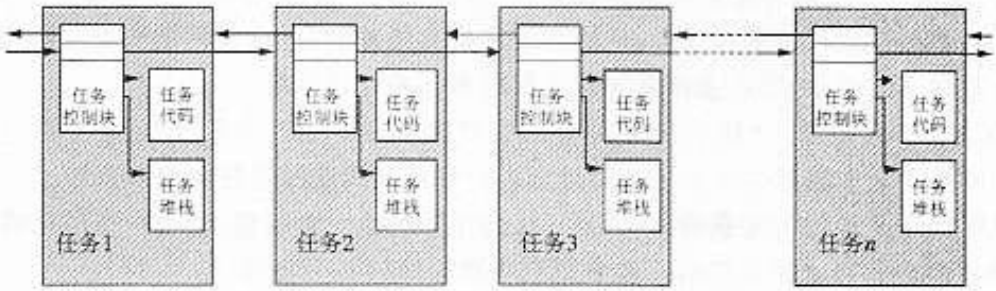
μC/OS－II 的任务有两种：用户任务和系统任务。由应用程序设计者编写的任务，叫做用户任务；由系统提供的任务叫做系统任务。用户任务是为解决应用问题而编写的；系统任务是为应用程序来提供某种服务的。

为了管理上的方便，μC/OS－II 把每一个任务都作为一个节点，然后把它们链接成如图 2－1(b)所示的一个任务链表。

目前，μC/OS－II 最多可以对 64 个任务(包括用户任务和系统任务)进行管理。



(a) 任务的组成



(b) 任务链表

图 2－1　μC/OS－II 的任务在内存中的结构

2.1.1 任务的状态

因为在嵌入式系统中只有一个 CPU，所以在一个具体时刻只能允许一个任务占用 CPU。根据任务是否占用 CPU，以及是否处于被中断、等待等情况，任务在 μC/OS－II 中可能处于表 2－1 所列的 5 种状态之一。

表 2－1 任务的 5 种状态

任务的状态	说 明
睡眠状态	任务只是以代码的形式驻留在程序空间（ROM 或 RAM），还没有交给操作系统管理时的情况叫做睡眠状态。简单地说，任务在没有被配备任务控制块或被剥夺了任务控制块时的状态叫做任务的睡眠状态
就绪状态	如果系统为任务配备了任务控制块且在任务就绪表中进行了就绪登记，则任务就具备了运行的充分条件，这时任务的状态叫做就绪状态
运行状态	处于就绪状态的任务如果经调度器判断获得了 CPU 的使用权，则任务就进入运行状态 任何时刻只能有一个任务处于运行状态，就绪的任务只有当所有优先级高于本任务的任务都转为等待状态时，才能进入运行状态
等待状态	正在运行的任务，需要等待一段时间或需要等待一个事件发生再运行时，该任务就会把 CPU 的使用权让给其他任务而使任务进入等待状态
中断服务状态	一个正在运行的任务一旦响应中断申请就会中止运行而去执行中断服务程序，这时任务的状态叫做中断服务状态

任务在不同状态之间的转换见图 2－2。

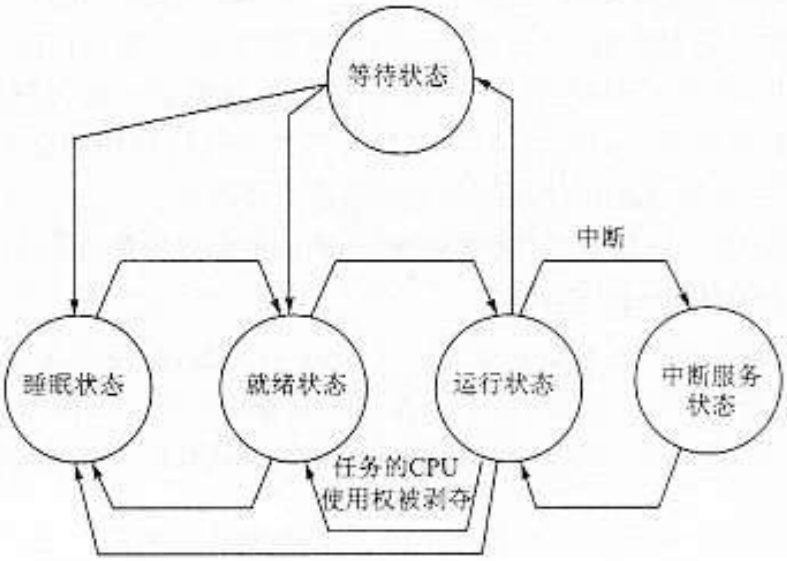


图 2－2 任务状态的转换

2.1.2 用户任务代码的一般结构

1. 用户任务代码的一般结构

根据嵌入式系统任务的工作特点，任务的执行代码通常是一个无限循环结构，并且在这个循环中可以响应中断，这种结构也叫做超循环结构。例 2－1 就是一个具有超循环结构的任务的示意性代码。

例 2－1 一个用 C 语言编写的任务。

```
void MyTask(void *pdata)
{
    for (;;)
    {
        可以被中断的用户代码;
        OS_ENTER_CRITICAL();          //进入临界段(关中断)
        不可以被中断的用户代码;
        OS_EXIT_CRITICAL();           //退出临界段(开中断)
        可以被中断的用户代码;
    }
}
```

从程序设计的角度来看，一个 μC/OS－II 任务的代码就是一个 C 语言函数。为了可以传递各种不同类型的数据甚至是函数，任务的参数是一个 void 类型的指针。

为了有效地对中断进行控制，在任务的代码里可使用 μC/OS－II 定义的宏 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()来控制任务何时响应中断，何时屏蔽中断。在运行这两个宏之间的代码时是不会响应中断的，这种受保护的代码段叫做临界段。在具体应用中可以，根据实际需要在一个任务中使用这对宏设置多个临界段。

因此可以说，μC/OS－II 任务的代码结构是一个带有临界段的无限循环。

2. 用户应用程序的一般结构

从程序代码上来看，用户任务似乎就是一个 C 语言函数，但是这个函数不是一般的 C 语言函数，它是一个任务(线程)。因此，它不是被主函数或其他函数调用的，主函数 main()只负责创建和启动它们，而由操作系统负责来调度运行它们。从代码上来看，用户应用程序的代码大体上如例 2－2 所示。

例 2－2　用户应用程序的结构。

```
void MyTask1(void * pdata)              //定义用户任务 1
{
    for (;;)
    {
        ……
    }
}
void MyTask2(void * pdata)              //定义用户任务 2
{
    for (;;)
    {
        ……
    }
}
void MyTask3(void * pdata)              //定义用户任务 3
{
    for (;;)
    {
        ……
    }
}
void main()
{
    ……
    OSInit();                           //初始化 μC/OS－II
    ……
    OSTaskCreate(MyTask1,……);           //创建用户任务 1
    OSTaskCreate(MyTask2,……);           //创建用户任务 2
    OSTaskCreate(MyTask3,……);           //创建用户任务 3
    ……
    OSStart();                          //启动任务
    ……
}
```

其中 OSTaskCreate()为操作系统 μC/OS－II 提供的用来创建任务的函数；OSStart()为操作系统 μC/OS－II 提供的启动各任务的函数。使用函数 OSStart()启动了各任务之后，任务就交由操作系统来管理和调度了。

2.1.3 系统任务

μC/OS－II 预定义了两个为应用程序服务的系统任务：空闲任务和统计任务。其中空闲任务是每个应用程序必须使用的，而统计任务则是应用程序可以根据实际需要来选择使用的。

1. 空闲任务

在多任务系统运行时，系统经常会在某个时间内无用户任务可运行而处于所谓的空闲状态。为了使 CPU 在没有用户任务可执行时有事可做，μC/OS－II 提供了一个叫做空闲任务 OSTaskIdle()的系统任务。空闲任务的代码如下：

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

    pdata = pdata;                      //防止某些编译器报错
    for(;;)
    {
        OS_ENTER_CRITICAL();            //关闭中断
            OSdleCtr ++ ;               //计数
        OS_EXIT_CRITICAL();             //开放中断
    }
}
```

这里有一行代码为 pdata = pdata，之所以安排了这样一行代码，是因为在这个任务中没有用到参数 pdata。对于某些 C 编译器，在对代码进行编译时会对这种情况报错（说定义了参数却没有使用），有了这行代码，编译器就不会报错了。

从上面的代码中可以看到，这个空闲任务几乎不做什么事情，只是对系统定义的一个空闲任务运行次数计数器 OSdleCtr 进行加 1 操作。当然，如果用户认为有必要，那么也可在空闲任务中编写一些做用户工作的代码。

μC/OS－II 规定，一个用户应用程序必须使用这个空闲任务，而且这个任务是不能用软件来删除的。

2. 统计任务

μC/OS－II 提供的另一个系统任务是统计任务 OSTaskStat()。这个统计任务每秒计算一次 CPU 在单位时间内被使用的时间，并把计算结果以百分比的形式存放在变量 OSCPUUsage 中，以便应用程序通过访问它来了解 CPU 的利用率，所以该系统任务 OSTaskStat()

叫做统计任务。

用户应用程序是否使用统计任务，用户可以根据应用程序的实际需要来进行选择。如果用户应用程序要使用这个统计任务，则必须把定义在系统头文件 OS_CFG.H 中的系统配置常数 OS_TASK_STAT_EN 设置为 1，并且必须在创建统计任务之前调用函数 OSStatInit() 对统计任务进行初始化。

2.1.4 任务的优先权及优先级别

前面已提及，μC/OS－II 的每个任务都必须具有一个惟一的优先级别。μC/OS－II 把任务的优先权分为 64 个优先级别，每一个级别都用一个数字来表示。数字 0 表示任务的优先级别最高；数字越大则表示任务的优先级别越低。

通常，一个应用程序的任务数小于 64。为了节省内存，用户可以根据应用程序的需要，在文件 OS_CFG.H 中通过给表示最低优先级别的常数 OS_LOWEST_PRIO 赋值的方法，来说明应用程序中任务优先级别的数目。该常数一旦被定义，就意味着系统中可供使用的优先级别为 0、1、2、…、OS_LOWEST_PRIO，共 OS_LOWEST_PRIO＋1 个。同时也限制了应用程序中任务的总数最多不能超过 OS_LOWEST_PRIO＋1 个。固定地，系统总是把最低优先级别 OS_LOWEST_PRIO 自动赋给空闲任务。如果应用程序中还使用了统计任务，则系统会把优先级别 OS_LOWEST_PRIO－1 自动赋给统计任务，因此用户任务可以使用的优先级别是 0、1、2、…、OS_LOWEST_PRIO－2，共 OS_LOWEST_PRIO－1 个。

例 2－3 如果希望应用程序中任务的优先级别为 28 个，则表示最低优先级别的常数 OS_LOWEST_PRIO 值应该是多少？如果应用程序中使用了系统提供的空闲任务和统计任务，则该应用程序最多可以安排多少个任务？

答 表示最低优先级别的常数 OS_LOWEST_PRIO 值应该为 27，优先级别分别为 0、1、2、3、…、27；由于系统空闲任务占用了优先级别 27，统计任务占用了优先级别 26，则应用程序中最多可以安排优先级别分别为 0、1、2、…、25 的 26 个任务。

给某一个用户任务的定义优先级别，需要在调用系统函数 OSTaskCreate() 来创建任务时，用该函数的第 4 个参数 prio 来指定。

由于每个任务都具有惟一的优先级别，因此 μC/OS－II 通常也用任务的优先级别来作为这个任务的标识。

2.2 任务堆栈

所谓堆栈，就是在存储器中按数据"后进先出(LIFO)"的原则组织的连续存储空间。为了满足任务切换和响应中断时保存 CPU 寄存器中的内容及存储任务私有数据的需要，每个任

务都应该配有自己的堆栈。任务堆栈是任务的重要组成部分。

2.2.1 任务堆栈的创建

为方便定义任务堆栈，在文件 OS_CPU.H 中专门定义了一个数据类型 OS_STK：

```
typedef  unsigned  int   OS_STK;          //该类型长度为 16 位
```

这样，在应用程序中定义任务堆栈的栈区就非常简单，即定义一个 OS_STK 类型的一个数组即可。例如：

```
#define  TASK_STK_SIZE     512            //定义堆栈的长度(1 024 字节)
OS_STK TaskStk[TASK_STK_SIZE];            //定义一个数组来作为任务堆栈
```

当调用函数 OSTaskCreate()来创建一个任务时，把数组的指针传递给函数 OSTaskCreate()中的堆栈栈顶参数 ptos，就可以把该数组与任务关联起来而成为该任务的任务堆栈。

例 2－4 创建任务函数 OSTaskCreate()的原型如下：

```
INT8U  OSTaskCreate (
                    void ( * task)(void  * pd),     //指向任务的指针
                    void  * pdata,                  //传递给任务的参数
                    OS_STK  * ptos,                 //任务堆栈栈顶的指针
                    INT8U prio                      //指定任务优先级别的参数
                    );
```

创建一个任务，该任务的代码如例 2－1 所示。堆栈的长度为 128 字节，优先级别为 20，任务参数 pdata 的实参为 MyTaskAgu。试写出需要的代码。

答

```
#define MyTaskStkN      64
OS_STK MyTaskStk[MyTaskStkN];

void main(void)
{
    ......
    OSTaskCreate(
                MyTask,                         //任务的指针
                &MyTaskAgu,                     //传递给任务的参数
                & MyTaskStk[MyTaskStkN－1],     //任务堆栈栈顶地址
                20                              //任务的优先级别
                );
    ......
}
```

需要注意的是，堆栈的增长方向是随系统所使用的处理器不同而不同的。有的处理器要求堆栈的增长方向是向上的，而另一些处理器要求堆栈的增长方向是向下的，如图 2-3 所示。因此在使用函数 OSTaskCreate()创建任务时，一定要注意所使用的处理器对堆栈增长方向的支持是向上的还是向下的。

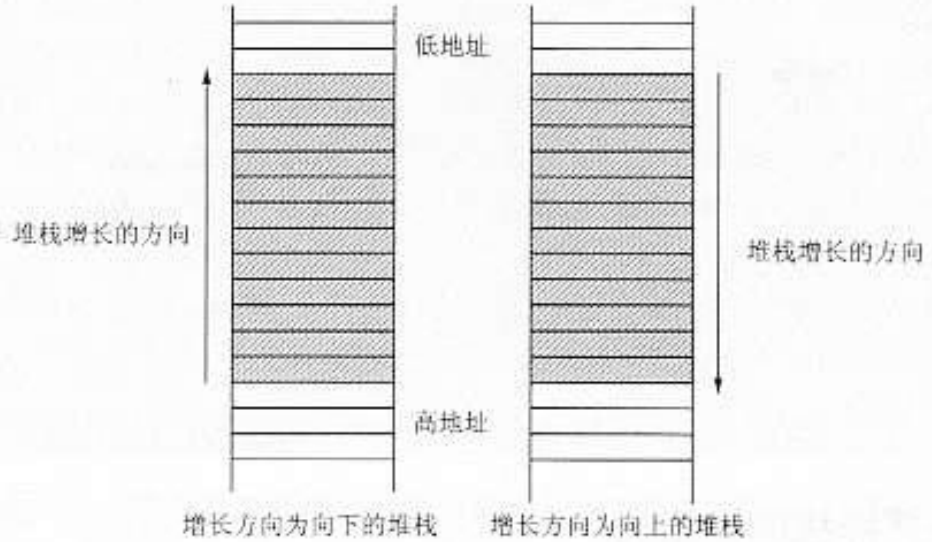


图 2-3 堆栈的不同增长方向

例 2-4 是假设使用了支持堆栈向下增长方式的处理器的条件下设置的函数参数 ptos。如果使用的处理器支持堆栈的增长方向是向上的，则对于例 2-4 来说，调用函数 OSTaskCreate()创建任务时应写成如下形式：

```
OSTaskCreate(MyTask, &MyTaskAgu, & MyTaskStk[0], 20);
```

为了提高应用程序的可移植性，在编写程序时也可把两种代码都编写出来，利用 OS_CFG.H 文件中的常数 OS_STK_GROWTH 作为选择开关，使用户可通过定义该常数的值来选择相应的代码段，以适应不同的堆栈增长方式的需要。这种情况下的一个可能的代码段如下：

```
#define MyTaskStkN      64
OS_STK MyTaskStk[MyTaskStkN];

void main(void)
{
    ......
#if OS_STK_GROWTH == 1
    OSTaskCreate(
```

```
                    MyTask,                          //任务的指针
                    &MyTaskAgu,                      //传递给任务的参数
                    &MyTaskStk[MyTaskStkN-1],        //任务堆栈栈顶地址
                    20                               //任务的优先级别
                    );
#else
    OSTaskCreate(
                    MyTask,                          //任务的指针
                    &MyTaskAgu,                      //传递给任务的参数
                    &MyTaskStk[0],                   //任务堆栈栈顶地址
                    20                               //任务的优先级别
                    );
#endif
    ……
}
```

2.2.2 任务堆栈的初始化

当 CPU 启动运行一个任务时，CPU 的各寄存器总是需要预置一些初始数据，例如指向任务的指针、程序状态字 PSW 等。

那么，在系统启动任务时，CPU 从何处可以获得这些数据呢？一个最方便的方法就是让 CPU 从这个任务的任务堆栈里获得这些数据。为此，应用程序在创建一个新任务时，就必须把在系统启动这个任务时 CPU 各寄存器所需要的初始数据（任务指针、任务堆栈指针、程序状态字等）事先存放在任务的堆栈中。这样，当任务获得 CPU 使用权时，就能把堆栈中的初始数据复制到 CPU 的各寄存器里，从而可使任务顺利地启动并运行。

任务堆栈的初始化工作应该是由操作系统负责的。μC/OS－II 在创建任务函数 OSTaskCreate()中通过调用任务堆栈初始化函数 OSTaskStkInit()来完成任务堆栈初始化工作，其原型如下：

```
OS_STK *OSTaskStkInit(
                    void (*task)(void *pd),
                    void *pdato,
                    OS_STK *ptos,
                    INT16U opt
                    );
```

目前，由于各种处理器的寄存器及对堆栈的操作方式不尽相同，因此该函数需要用户在进行 μC/OS－II 的移植时，按所使用的处理器由用户来编写。实现这个函数的具体细节，将在

本书有关μC/OS-II移植的章节中进一步介绍。

2.3 任务控制块及任务控制块链表

μC/OS-II用来记录任务的堆栈指针、任务的当前状态、任务的优先级别等一些与任务管理有关的属性的表就叫做任务控制块(OS_TCB)。

从图2-1中可以看到,任务控制块负责把任务代码和任务堆栈进行关联,而使任务控制块、任务代码和任务堆栈成为一个整体,并且系统要通过这个任务控制块来感知和管理一个任务。因此,任务控制块就相当于一个任务的身份证,没有任务控制块的任务是不能被系统承认和管理的。

为了管理系统中多个任务,μC/OS-II把系统所有任务的控制块链接为两条链表,并通过这两条链表管理各任务控制块,进而再通过任务控制块来对任务进行相关的操作。

2.3.1 任务控制块的结构

任务控制块是一个结构类型数据。当用户应用程序调用OSTaskCreate()函数创建一个用户任务时,这个函数就会对任务控制块中的所有成员赋予与该任务相关的数据,并驻留在RAM中。

任务控制块结构的定义如下:

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;          //指向任务堆栈栈顶的指针

#if OS_TASK_CREATE_EXT_EN
    void            *OSTCBExtPtr;          //指向任务控制块扩展的指针
    OS_STK          *OSTCBStkBottom;       //指向任务堆栈栈底的指针
    INT32U          OSTCBStkSize;          //任务堆栈的长度
    INT16U          OSTCBOpt;              //创建任务时的选择项
    INT16U          OSTCBId;               //目前该域未被使用
#endif

    struct os_tcb   *OSTCBNext;            //指向后一个任务控制块的指针
    struct os_tcb   *OSTCBPrev;            //指向前一个任务控制块的指针

#if (OS_Q_EN && (OS_MAX_OS >= 2)) || OS_MBOX_EN || OS_Sem_EN
    OS_EVENT        *OSTCBEventPtr;        //指向事件控制块的指针
#endif
```

```
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void            *OSTCBMsg;              //指向传递给任务消息的指针
#endif

    INT16U          OSTCBDly;               //任务等待的时报(节拍数)
    INT8U           OSTCBStat;              //任务的当前状态标志
    INT8U           OSTCBPrio;              //任务的优先级别

    INT8U           OSTCBX;                 //用于快速访问就绪表的数据
    INT8U           OSTCBY;                 //用于快速访问就绪表的数据
    INT8U           OSTCBBitX;              //用于快速访问就绪表的数据
    INT8U           OSTCBBitY;              //用于快速访问就绪表的数据

#if OS_TASK_DEL_EN
    BOOLEAN         OSTCBDelReq;            //请求删除任务时用到的标志
#endif
} OS_TCB;
```

其中成员 OSTCBStat 用来存放任务的当前状态，该成员变量可能的值见表 2－2。

表 2－2　OSTCBStat 可能的值

值	说　明
OS_STAT_RDY	表示任务处于就绪状态
OS_STAT_SEM	表示任务处于等待信号量状态
OS_STAT_MBOX	表示任务处于等待消息邮箱状态
OS_STAT_Q	表示任务处于等待消息队列状态
OS_STAT_SUSPEND	表示任务处于被挂起状态
OS_STAT_MUTEX	表示任务处于等待互斥型信号量状态

2.3.2　任务控制块链表

μC/OS－II 用两条链表来管理任务控制块。一条是空任务块链表(其中所有任务控制块还没有分配给任务)；另一条是任务块链表(其中所有任务控制块已经分配给任务)。空任务块链表是在应用程序调用函数 OSInit()对 μC/OS－II 系统进行初始化时建立的；而任务块链表则是在调用函数 OSTaskCreate()创建任务时建立的。建立任务控制块链表的具体做法是，从空任务控制块链表摘取一个空任务控制块，然后填充上任务属性后，再形成新的链表。

系统在调用函数 OSInit()对 μC/OS－II 系统进行初始化时，先在 RAM 中建立一个 OS_TCB 结构类型的数组 OSTCBTbl[ ]，这样每个数组元素就是一个任务控制块，然后把这些控制块链接成了一个如图 2－4 所示的链表。由于链表中的这些控制块还没有与具体任务

图 2－4　μC/OS－II 初始化时创建一个空的任务控制块链表

相关联，因此这个链表叫做空任务块链表。

从图 2－4 中可以看到，μC/OS－II 初始化时建立的空任务链表的元素一共是 OS_MAX_TASKS＋OS_N_SYS_TASKS 个。其中定义在文件 OS_CFG.H 中的常数 OS_MAX_TASKS 指明了用户任务的最大数目；而定义在文件 UCOS_II.H 中的常数 OS_N_SYS_TASKS 指明了系统任务的数目（在图 2－4 中，其值为 2：一个空闲任务，一个统计任务）。

每当应用程序调用系统函数 OSTaskCreate()或 OSTaskCreateExt()创建一个任务时，系统就会将空任务控制块链表头指针 OSTCBFreeList 指向的任务控制块分配给该任务。在给任务控制块中的各成员赋值后，就按任务控制块链表的头指针 OSTCBList 将其加入到任务控制块链表中。

图 2－5 是在图 2－4 所示空任务控制块链表基础上，应用程序创建了两个用户任务并使用了两个系统任务（空闲任务和统计任务）的情况时，空任务块链表和任务块链表的结构示意图（图中阴影区域为任务块链表）。

为了加快对任务控制块的访问速度，除了任务控制块链表被创建为双向链表之外，μC/OS－II 在 uCOS_II.H 文件中还定义了一个数据类型为 OS_TCB＊的数组 OSTCBTbl[ ]，专门用来以任务的优先级别为顺序在各个数组元素里存放指向各个任务控制块的指针。这样系统在访问一个任务的任务控制块时，就可以不必遍历任务控制块链表了。数组 OSTCBTbl[ ]与任务控制块链表中任务控制块之间的关系如图 2－5 所示。

另外，为了 μC/OS－II 能随时访问正在运行任务的任务控制块，μC/OS－II 还定义了一个 OS_TCB＊类型的变量 OSTCBCur，专门存放当前正在运行的任务的任务控制块指针。图 2－5 是在假设任务优先级别为 3 的任务正在运行时指针变量 OSTCBCur 的指向。

μC/OS－II 允许用函数 OSTaskDel()删除一个任务。删除一个任务，实质上就是把该任务的任务从任务控制块链表中删掉，并把它归还给空任务控制块链表。这样，μC/OS－II 对这个没有任务控制块的任务就不再理会了，因为与这个任务对应的任务控制块已经被“吊销”了。由此可见，任务的任务控制块就如同人的身份证一样重要。

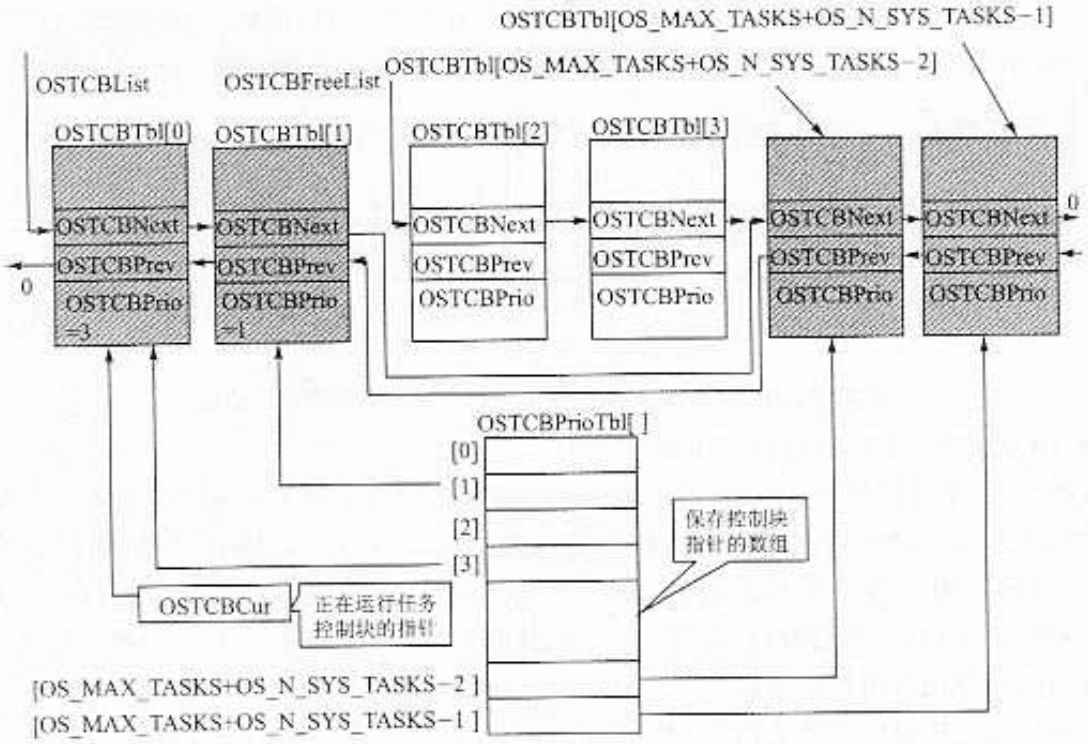


图 2－5　μC/OS－II 任务控制块链表和 OSTCBPrioTbl[]数组及变量 OSTCBCur

2.3.3　任务控制块的初始化

给用户任务分配任务控制块及对其进行初始化也是操作系统的职责。当应用程序调用函数 OSTaskCreate()创建一个任务时，这个函数会调用系统函数 OSTCBInit()来为任务控制块进行初始化。这个函数首先为被创建任务从空任务控制块链表获取一个任务控制块；然后用任务的属性对任务控制块各个成员进行赋值；最后再把这个任务控制块链入到任务控制块链表的头部。

初始化任务控制块函数 OSTCBInit()的原型如下：

```
INT8U OSTCBInit (
            INT8U prio,          //任务的优先级别，保存在 OSTCBPrio 中
            OS_STK *ptos,        //任务堆栈栈顶指针，保存在 OSTCBStkPtr 中
            OS_STK *pbos,        //任务堆栈栈底指针，保存在 OSTCBStkBottom 中
            INT16U id,           //任务的标识符，保存在 OSTCBId 中
            INT16U stk_size,     //任务堆栈的长度，保存在 OSTCBStkSize 中
            void *pext,          //任务控制块的扩展指针，保存在 OSTCBExtPtr 中
            INT16U opt           //任务控制块的选择项，保存在 OSTCBOpt 中
            );
```

2.4 任务就绪表及任务调度

多任务操作系统的核心工作就是任务调度。所谓调度，就是通过一个算法在多个任务中确定哪个任务来运行。做这项工作的函数就叫做调度器。

μC/OS－II进行任务调度的思想是，每时每刻总是让优先级最高的就绪任务处于运行状态。为了保证这一点，它在系统或用户任务调用系统函数及执行中断服务程序结束时，总是调用调度器来确定应该运行的任务并运行它。

2.4.1 任务就绪表的结构

μC/OS－II进行任务调度的依据就是任务就绪表。

从图2－2任务的状态转换图中可以看到，系统总是从处于就绪状态的任务中来选择一个任务运行。

为了能使系统清楚地知道系统中哪些任务已经就绪，哪些还没有就绪，μC/OS－II在RAM中设立了一个记录表，系统中的每个任务都在这个表中占据一个位置，并用这个位置的状态(1或者0)来表示任务是否处于就绪状态。这个表就叫做任务就绪状态表，简称为任务就绪表。

μC/OS－II用一个类型为INT8U的数组OSRdyTbl[ ]来充当这个任务就绪表。图2－6表示的是一个最多可以记录32个任务就绪状态的任务就绪表。在这个任务就绪表中，以任务的优先级别(也是任务的标识)高低为顺序，为每个任务安排了一个二进制位，并规定该位的值为1表示对应的任务处于就绪状态，而该位的值为0则表示对应的任务处于非就绪状态。

从图2－6中可以看到，由于每个任务的就绪状态只占据一位，因此OSRdyTbl[]数组的一个元素可表达8个任务的就绪状态。在本例的情况下，数组的4个元素一共可表达32个任务的就绪状态。也就是说，每一个数组元素描述了8个任务的就绪状态，于是这8个任务就可看成一个任务组。为了便于对就绪表的查找，μC/OS－II又定义了一个数据类型为INT8U的变量OSRdyGrp，并使该变量的每一个位都对应OSRdyTbl[]的一个任务组(即数组的一个元素)。如果某任务组中有任务就绪，则在变量OSRdyGrp里把该任务组所对应的位置为1；否则置为0。例如，如果OSRdyGrp＝11100101，那么就意味着OSRdyTbl[0]、OSRdyTbl[2]、OSRdyTbl[5]、OSRdyTbl[6]、OSRdyTbl[7]任务组中有任务就绪。

变量OSRdyGrp的格式如图2－7所示。

由于变量OSRdyGrp有8个二进制位，每位对应OSRdyTbl[]数组的一个元素，每个元素又可以记录8个任务的就绪状态，因此μC/OS－II最多可以管理8×8＝64个任务。

如何根据任务的优先级别来查找任务在就绪表的位置呢？由于优先级别是一个单字节的数字，而且其最大值不会超过63，即二进制形式的00111111，因此，可以把优先级别看成是一

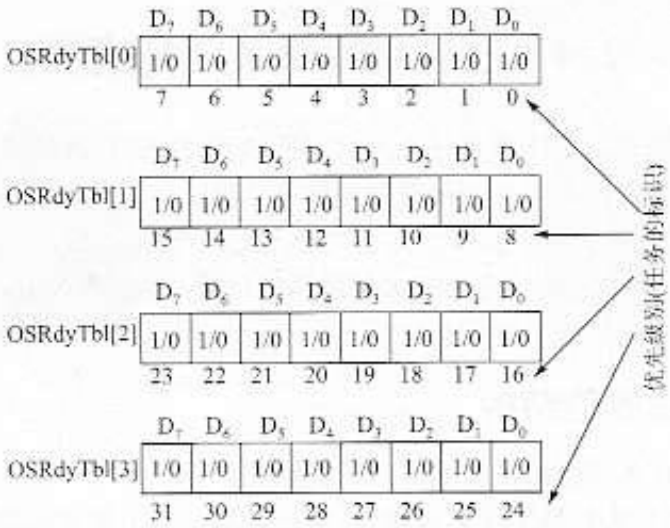


图 2－6　就绪表

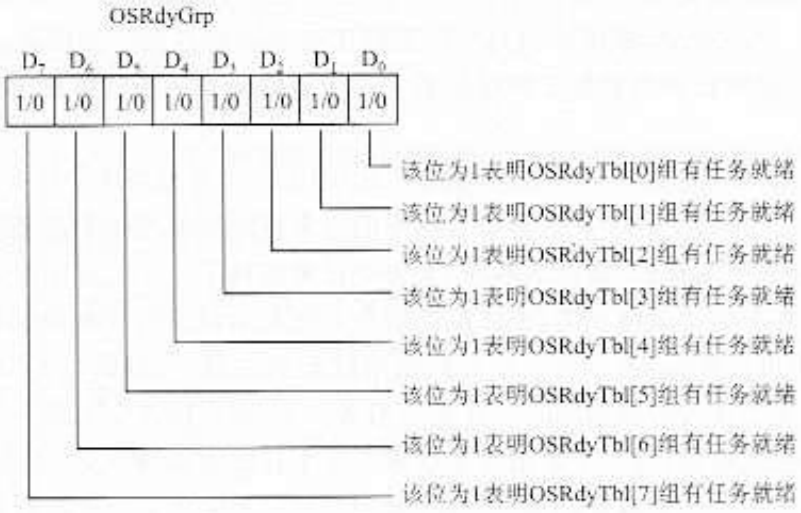


图 2－7　变量 OSRdyGrp 的格式及含义

个 6 位的二进制数，这样就可以用高 3 位（$D_5D_4D_3$）来指明变量 OSRdyGrp 的具体数据位，并用来确定就绪表数组元素的下标；用低 3 位（$D_2D_1D_0$）来指明该数组元素的具体数据位，见图 2－8。

例 2－5　已知某一个已经就绪的任务的优先级别 prio＝30，试判断应该在就绪表的哪一位上置 1。

答　30 的二进制形式为 00011110，其低 6 位为 011110，于是可知应该在 OSRdyTbl[3]的 D_6 位上置 1，同时要把变量 OSRdyGrp 的 D_3 位置 1。

图 2－8　任务的优先级别与 OSRdyGrp 各数据位及数组元素中各数据位之间的关系

2.4.2　对任务就绪表的操作

在程序中，可用类似于下面的代码把优先级别为 prio 的任务置为就绪状态。

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

其中 OSMapTbl[ ]是 μC/OS－II 为加快运算速度定义的一个数组，它的各元素值为：

```
OSMapTbl[0] = 00000001B
OSMapTbl[1] = 00000010B
OSMapTbl[2] = 00000100B
OSMapTbl[3] = 00001000B
OSMapTbl[4] = 00010000B
OSMapTbl[5] = 00100000B
OSMapTbl[6] = 01000000B
OSMapTbl[7] = 10000000B
```

如果要使一个优先级别为 prio 的任务脱离就绪状态，则可使用如下代码：

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

从任务就绪表中获取优先级别最高的就绪任务可用如下代码：

```
y = OSUnMapTal[OSRdyGrp];          //获得优先级别的 D5、D4、D3 位
x = OSUnMapTal[OSRdyTbl[y]];       //获得优先级别的 D2、D1、D0 位
prio = (y<<3) + x;                 //获得就绪任务的优先级别
```

或

```
y = OSUnMapTbl[OSRdyGrp];
prio  = (INT8U)((y<<3) + OSUnMapTbl[OSRdyTbl[y]]);
```

该代码执行后，得到的是最高优先级就绪任务的优先级别（即任务的标识）。其中 OSUnMapTbl[ ]同样是 μC/OS－II 为提高查找速度定义的一个数组，它共有 256 个元素，定义如下：

```
INT8U const OSUnMapTbl[ ] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

μC/OS－II 经常使用类似于就绪表的形式来记录任务的某种状态，因此读者一定要熟悉这种表的结构以及对这种表的基本操作。

2.4.3 任务的调度

在多任务系统中，令 CPU 中止当前正在运行的任务转而去运行另一个任务的工作叫做任务切换，而按某种规则进行任务切换的工作叫做任务的调度。

在 μC/OS－II 中，任务调度是由任务调度器来完成的。任务调度器的主要工作有两项：一是在任务就绪表中查找具有最高优先级别的就绪任务；二是实现任务的切换。μC/OS－II 有两种调度器：一种是任务级的调度器；另一种是中断级的调度器。任务级的调度器由函数 OSSched()来实现，而中断级的调度器由函数 OSIntExt()来实现。这里主要介绍任务级的调度器 OSSched()。

关于调度器在任务就绪表中查找具有最高优先级别就绪任务的代码，在 2.4.2 小节中已经有过叙述，本节主要介绍任务调度器是如何来进行任务切换的。调度器把任务切换的工作分为两个步骤：第一步是获得待运行任务的 TCB 指针；第二步是进行断点数据的切换。

1. 获得待运行就绪任务控制块的指针

由于操作系统是通过任务的任务控制块 TCB 来管理任务的，因此调度器真正实施任务切

换之前的主要工作就是要获得待运行任务的任务控制块指针和当前任务的任务控制块指针。

因为被中止任务的任务控制块指针存放在全局变量 OSTCBCur 中，所以调度器这部分的工作主要是获得待运行任务的任务控制块指针。

任务级调度器 OSSched()的源代码如下：

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

    INT8U y;

    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0)
    {
        y = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy                          //得到最高级优先任务
                = (INT8U)((y<<3) + UnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur)
        {
            OSTCBHighRdy                       //得到任务控制块指针
                        = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;                      //统计任务切换次数的计数器加 1
            OS_TASK_SW();
        }
    }
    OS_EXIT_CRITICAL();
}
```

μC/OS－II 允许应用程序通过调用函数 OSSchedLock()和 OSSchedUnlock()给调度器上锁和解锁。为了记录调度器被锁和解锁的情况，μC/OS－II 定义了一个变量 OSLockNesting：调度器每被上锁一次，变量 OSLockNesting 就加 1；反之，调度器每被解锁一次，变量 OSLockNesting 就减 1。因此，可以通过访问变量 OSLockNesting 了解调度器上锁的嵌套次数。

调度器 OSSched()在确认未被上锁并且不是中断服务程序调用调度器的情况下，首先从任务就绪表中查得的最高优先级别就绪任务的优先级别 OSPrioHighRdy；然后在确认这个就绪任务不是当前正在运行的任务（OSPrioCur 是存放正在运行任务的优先级别的变量）的条件下，用 OSPrioHighRdy 作为下标去访问数组 OSTCBPrioTbl[]，把数组元素 OSTCBPrioTbl[OSPrio-

HighRdy]的值(即待运行就绪任务的任务控制块指针)赋给指针变量 OSTCBHighRdy。于是下面就可以依据 OSTCBHighRdy 和 OSTCBCur 这两个分别指向待运行任务控制块和当前任务控制块的指针在宏 OS_TASK_SW()中实施任务切换了。

图 2－9　调度器在任务切换前获得的两个指针 OSTCBCur 和 OSTCBHighRdy

2. 任务切换宏 OS_TASK_SW()

其实任务切换的工作是靠 OSCtxSw()来完成的。

为了了解 OSCtxSw()的作用，再来分析一下任务切换。

简单地说，任务切换就是中止正在运行的任务(当前任务)，转而去运行另外一个任务的操作。当然，这个任务应该是就绪任务中优先级别最高的那个任务。

为了了解调度器是如何进行任务切换的，先来探讨一下一个被中止运行(可能因为中断或者调用)任务，将来又要"无缝"地恢复运行应该满足的条件。

为了讨论上的方便，如果把任务被中止运行时的位置叫做断点，把当时存放在 CPU 的 PC、PSW 和通用寄存器等各寄存器中的数据叫做断点数据，那么当任务恢复运行时，必须在断点处以断点数据作为初始数据接着运行，才能实现"无缝"的接续运行。因此，要实现这种"无缝"的接续运行，则必须在任务被中止时就把该任务的断点数据保存到堆栈中；而在被重新运行时，则要把堆栈中的这些断点数据再恢复到 CPU 的各寄存器中，只有这样才能使被中止运行的任务在恢复运行时实现"无缝"的接续运行。一个任务在被中止运行时保护断点数据的动作如图 2－10 所示。

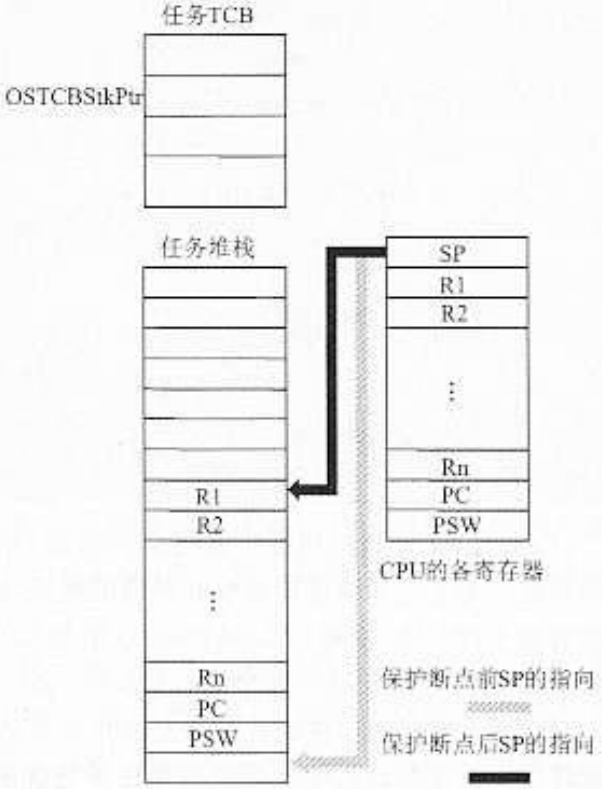


图 2－10　一个任务保护断点时的压栈动作

由上述可知，一个被中止的任务能否正确地在断点处恢复运行，其关键在于是否能正确地在CPU 各寄存器中恢复断点数据；而能正确地恢复断点数据的关键是 CPU 的堆栈指针 SP 是否有正确的指向。因此也可知，在系统中存在多个任务时，如果在恢复断点数据时用另一个任务的任务堆栈指针（存放在控制块成员 OSTCBStkPtr 中）来改变 CPU 的堆栈指针 SP，那么 CPU 运行的就不是刚才被中止运行的任务，而是另一个任务了，也就是实现任务切换了。当然，为防止被中止任务堆栈指针的丢失，被中止任务在保存断点时，要把当时 CPU 的 SP 值保存到该任务控制块的成员 OSTCBStkPtr 中。

综上所述，任务的切换就是断点数据的切换，断点数据的切换也就是 CPU 堆栈指针的切换，被中止运行任务的任务堆栈指针要保存到该任务的任务控制块中，待运行任务的任务堆栈指针要由该任务控制块转存到 CPU 的 SP 中。调度器在任务切换时的工作过程如图 2－11 所示。

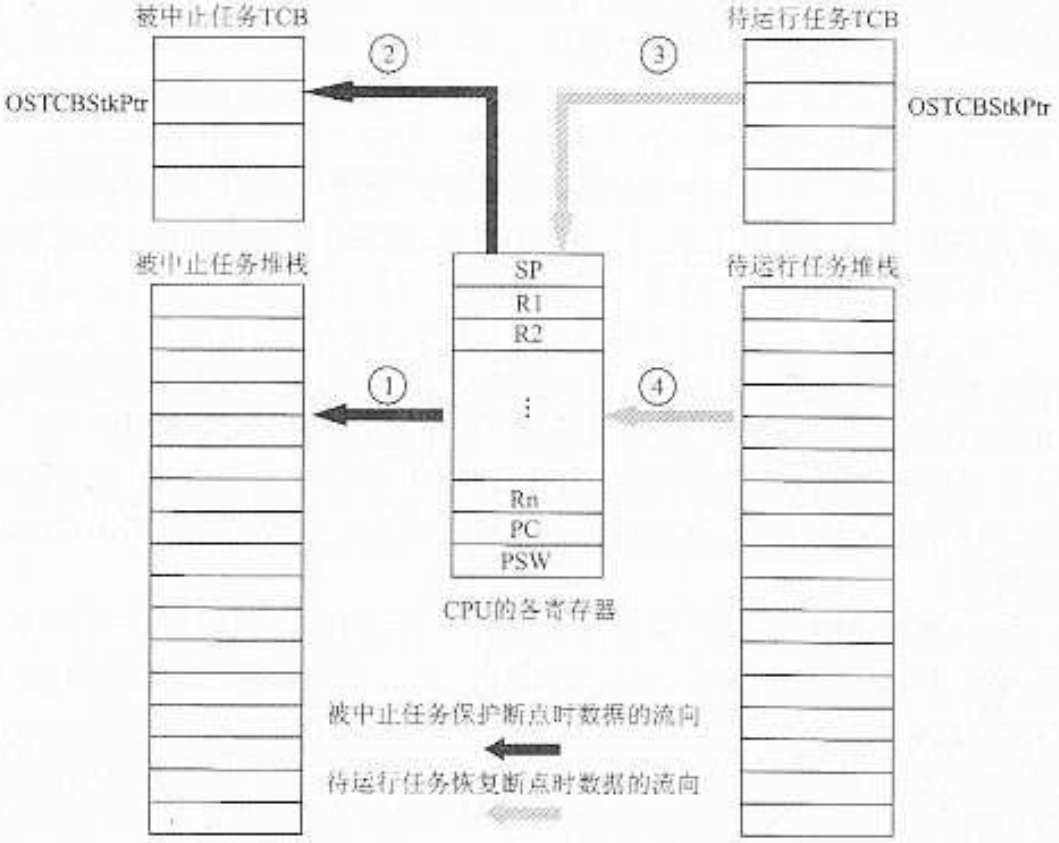


图 2－11　调度器进行任务切换的动作

为了完成上述操作，OSCtxSw()要依次做如下 7 项工作：

① 把被中止任务的断点指针保存到任务堆栈中；

② 把 CPU 通用寄存器的内容保存到任务堆栈中；

③ 把被中止任务的任务堆栈指针当前值保存到该任务的任务控制块的 OSTCBStkPtr 中；

④ 获得待运行任务的任务控制块；

⑤ 使 CPU 通过任务控制块获得待运行任务的任务堆栈指针；

⑥ 把待运行任务堆栈中通用寄存器的内容恢复到 CPU 的通用寄存器中；

⑦ 使 CPU 获得待运行任务的断点指针（该指针是待运行任务在上一次被调度器中止运行时保留在任务堆栈中的）。

由于 μC/OS－II 总是把当前正在运行任务的任务控制块的指针存放在一个指针变量 OSTCBCur 中，并且在调度器的前面代码中已经得到了待运行任务的任务控制块指针 OSTCBHighRdy，所以完成第②～⑥项工作非常容易。其示意性代码段如下：

```
用压栈指令把 CPU 通用寄存器 R1、R2…压入堆栈；
OSTCBCur -> OSTCBStkPtr = SP;          //把 SP 保存在中止任务控制块中
OSTCBCur = OSTCBHighRdy;               //使系统获得待运行任务控制块
SP = OSTCBHighRdy -> OSTCBStkPtr;      //把待运行任务堆栈指针赋给 SP
用出栈指令把 R1、R2…弹入 CPU 的通用寄存器；
```

完成第①项和第⑦项工作就有一些麻烦。众所周知，CPU 是按 CPU 中的一个特殊功能寄存器——程序指针 PC（也叫做程序计数器）的指向来运行程序的。或者说，只有使 PC 寄存器获得新任务的地址，才会使 CPU 运行新的任务。既然如此，对于被中止任务，应该把任务的断点指针（在 PC 寄存器中）压入任务堆栈；而对于待运行任务，应该把任务堆栈里上次任务被中止时存放在堆栈中的中断指针推入 PC 寄存器。但遗憾的是，目前的处理器一般没有对程序指针寄存器 PC 的出栈和入栈指令。这就不得不想办法用其他可以改变 PC 值的指令（如 CALL、INT 或 IRET 指令等）来变通一下了。也就是说，想办法引发一次中断（或者一次调用），并让中断向量指向 OSCtxSw()（其实这个函数就是中断服务程序），利用系统在跳转到中断服务程序时会自动把断点指针压入堆栈的功能，把断点指针存入堆栈，而利用中断返回指令 IRET（或有相同功能的指令）能把断点指针推入 CPU 的 PC 寄存器的功能，恢复待运行任务的断点，这样就可以实现断点的保存和恢复了。

由于任务切换时需要对 CPU 的寄存器进行操作，因此在一般情况下，中断服务程序 OSCtxSw() 都要用汇编语言来编写。在这里只给出 OSCtxSw()的示意性代码供读者参考。

```
void  OSCtxSw( void )
{
    用压栈指令把 CPU 通用寄存器 R1、R2…压入堆栈；
    OSTCBCur -> OSTCBStkPtr = SP;          //在中止任务控制块中保存 SP
    OSTCBCur = OSTCBHighRdy;               //任务控制块的切换
    OSPrioCur = OSPrioHighRdy;
    SP = OSTCBHighRdy -> OSTCBStkPtr;      //使 SP 指向待运行任务堆栈
    用出栈指令把 R1、R2…弹入 CPU 的通用寄存器；
    IRET;                                  //中断返回，使 PC 指向待运行任务
}
```

那么由什么来引发中断呢？这就是宏 OS_TASK_SW()的作用了。如果使用的微处理器具有软中断指令，那么在这个宏中封装一个软中断指令即可；如果使用的微处理器没有提供软中断指令，那么就可以试一试在宏 OS_TASK_SW()中封装其他可使 PC 等相关寄存器压栈的指令（例如调用指令）。

2.5 任务的创建

从前面的内容中可知，μC/OS－II 是通过任务控制块来管理任务的。因此，创建任务的工作实质上是创建一个任务控制块，并通过任务控制块把任务代码和任务堆栈关联起来形成一个完整的任务。当然，还要使刚创建的任务进入就绪状态，并接着引发一次任务调度。

μC/OS－II 有两个用来创建任务的函数：OSTaskCreate()和 OSTaskCreateExt()。其中函数 OSTaskCreateExt()是函数 OSTaskCreate()的扩展，并提供了一些附加功能。用户可根据需要使用这两个函数之一来完成任务的创建工作。

2.5.1 用函数 OSTaskCreate()创建任务

应用程序通过调用函数 OSTaskCreate()来创建一个任务。函数 OSTaskCreate()的源代码如下：

```
INT8U OSTaskCreate (
                    void ( * task)(void * pd),      //指向任务的指针
                    void * pdata,                   //传递给任务的参数
                    OS_STK * ptos,                  //指向任务堆栈栈顶的指针
                    INT8U prio                      //任务的优先级
                  )
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr;
# endif

    void     * psp;
    INT8U err;

    if (prio > OS_LOWEST_PRIO)                      //检测任务的优先级是否合法
    {
        return (OS_PRIO_INVALID);
    }
```

```
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB * )0)          //确认优先级未被使用
    {
        OSTCBPrioTbl[prio] = (OS_TCB * )1;           //保留优先级

        OS_EXIT_CRITICAL();
        psp = (void * )OSTaskStkInit(
                                     task,
                                     pdata,
                                     ptos,
                                     0
                                    );               //初始化任务堆栈
        err = OSTCBInit(
                        prio,
                        psp,
                        (void * )0, 0, 0,
                        (void * )0, 0
                       );                            //获得并初始化任务控制块
        if (err == OS_NO_ERR)
        {
            OS_ENTER_CRITICAL();
            OSTaskCtr ++ ;                           //任务计数器加 1
            OS_EXIT_CRITICAL();
            if (OSRunning)
            {
                OSSched();                           //任务调度
            }
        }
        else
        {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB )0;         //放弃任务
            OS_EXIT_CRITICAL();
        }
        return (err);
    }
    else
```

```
        {
            OS_EXIT_CRITICAL();
            return (OS_PRIO_EXIST);
        }
    }
```

从函数 OSTaskCreate()的源代码中可以看到，函数对待创建任务的优先级别进行一系列判断，确认该优先级别合法且未被使用之后，随即调用函数 OSTaskStkInit()和 OSTCBInit()对任务堆栈和任务控制块进行初始化。初始化成功后，除了把任务计数器加 1 外，还要进一步判断 μC/OS－II 的核是否在运行状态(即 OSRunning 的值是否为 1)。如果 OSRunning 的值是为 1，则调用 OSSched()进行任务调度。

调用函数 OSTaskCreate()成功后，将返回 OS_NO_ERR；否则，根据具体情况返回 OS_PRIO_INVALID、OS_PRIO_EXIST 及在函数内调用任务控制块初始化函数失败时返回的信息。

2.5.2 用函数 OSTaskCreateExt()创建任务

在任务及应用程序中也可以通过调用函数 OSTaskCreateExt()来创建一个任务。用 OSTaskCreateExt()来创建任务将更为灵活，但也会增加一些额外的开销。函数 OSTaskCreateExt()的原型如下：

```
INT8U OSTaskCreateExt (
                void        (*task)(void *pd),      //指向任务的指针
                void        *pdata,                 //传递给任务的参数
                OS_STK      *ptos,                  //指向任务堆栈栈顶的指针
                INT8U       prio,                   //任务的优先级
                INT16U      id,                     //任务的标识
                OS_STK      *pbos,                  //任务堆栈栈底的指针
                INT32U      stk_size,               //任务堆栈的容量
                void        *pext,                  //指向附加数据域的指针
                INT16U      opt                     //用于设定操作选项
            );
```

2.5.3 创建任务的一般方法

一般来说，任务可在调用函数 OSStart()启动任务调度之前来创建，也可在任务中来创建。但是，μC/OS－II 有一个规定：在调用启动任务函数 OSStart()之前，必须已经创建了至少一个任务。因此，习惯上在调用函数 OSStart()之前先创建一个任务，并赋予它最高的优先级别，从而使它成为起始任务；然后在这个起始任务中，再创建其他各任务。

如果要使用系统提供的统计任务，则统计任务的初始化函数也必须在这个起始任务中来调用。

下面是创建任务的示意性代码：

```
/*****************************主函数*****************************/
void main(void)
{
    ……
    OSInit();                          //对 μC/OS－II 进行初始化
    ……
    OSTaskCreate (TaskStart,……);       //创建起始任务 TaskStart
    OSStart();                         //开始多任务调度
}

/*****************************起始任务*****************************/
void TaskStart(void * pdata)
{
    ……//在这个位置安装并启动 μC/OS－II 的时钟
    OSStatInit();                      //初始化统计任务
    ……//在这个位置创建其他任务
    for(;;)
    {
        起始任务 TaskStart 的代码
    }
}
/*****************************************************************/
```

需要注意的是，μC/OS－II 不允许在中断服务程序中创建任务。

例 2－6 设计一个只有一个任务 MyTask 的应用程序，当程序运行后，任务 MyTask 的工作就是每秒在显示器上显示一个字符“M”。

答 该应用程序的代码如下：

```
/*****************************Test*****************************/
#include "includes.h"
#define  TASK_STK_SIZE    512                  //任务堆栈长度
OS_STK   MyTaskStk[TASK_STK_SIZE];             //定义任务堆栈区
INT16S   key;                                  //用于退出 μC/OS－II 的键
INT8U    x = 0,y = 0;                          //字符显示位置
```

```
void  MyTask(void * data);                          //声明一个任务
/*****************************主函数*********************************/
void  main (void)
{
    char * s_M = "M";                               //定义要显示的字符
    OSInit();                                       //初始化 μC/OS－II
    PC_DOSSaveReturn();                             //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                      //安装 μC/OS－II 任务切换中断向量
    OSTaskCreate(
        MyTask,                                     //创建任务 MyTask
            s_M,                                    //给任务传递参数
            &MyTaskStk[TASK_STK_SIZE - 1],          //设置任务堆栈栈顶指针
            0                                       // MyTask 优先级别为 0
        );
    OSStart();                                      //启动多任务管理
}

/*************************任务 MyTask*******************************/
void  MyTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                    //安装 μC/OS－II 时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);               //设置 μC/OS－II 时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                                   //初始化 μC/OS－II 的统计任务
    for (;;)
        {
            if (x>10)
            {
                x = 0;
                y += 2;
            }
        PC_DispChar(
```

```
            x, y,                          //字符的显示位置
                *(char *)pdata,            //被显示的字符
                DISP_BGND_BLACK + DISP_FGND_WHITE
            );
        x += 1;
        //如果按下 ESC 键,则退出 μCOS-II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();            //返回 DOS
            }
        }
    OSTimeDlyHMSM(0, 0, 1, 0);             //等待 1 s
    }
}
```

程序运行结果如图 2－12 所示。

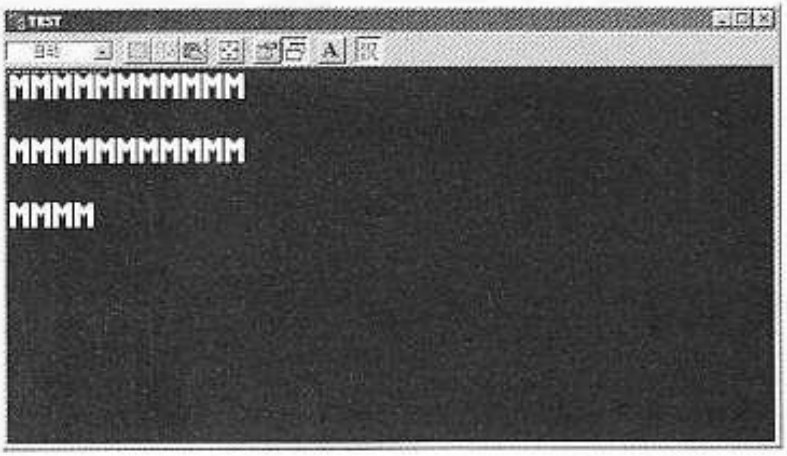


图 2－12　例 2－6 应用程序运行结果

说明：本书前 6 章中的例题代码，都是在把 μC/OS－II 操作系统移植到 PC 机上之后再来运行的(这样做更符合大多数读者的习惯和学习条件)，所以要用到 PC 机上的一些功能，本书中凡是名称以 PC 开头的函数，都调用了 PC 的一些功能。

PC 功能函数提供了 3 类服务：字符显示、运行时间测量及一些其他服务。在例 2－6 中用到的 PC 功能函数见表 2－3。关于其他 PC 功能函数的详细介绍参见本书的附录 A。

表 2－3　例 2－6 中用到的 PC 功能函数

PC 功能函数	说　明
PC_DOSSaveReturn()	保存 DOS 环境
PC_VectSet()	设置中断向量，即把中断向量添入中断向量码对应的中断向量表项中
PC_SetTickRate()	设置 μC/OS－II 时钟频率
PC_DispChar()	在 PC 的显示器上的指定位置显示一个字符
PC_GetKey()	获得键盘按键的键值
PC_DOSReturn()	恢复 DOS 环境

例 2－7　在例 2－6 应用程序的任务 MyTask 中再创建一个任务 YouTask。当程序运行后，任务 MyTask 的工作在显示器上显示一个字符"M"；而任务 YouTask 则是在显示器上显示字符"Y"。

答　应用程序的源代码如下：

```
/*********************************Test*********************************/
#include "includes.h"
#define  TASK_STK_SIZE    512                    //任务堆栈长度
OS_STK   MyTaskStk[TASK_STK_SIZE];               //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];              //定义任务堆栈区
INT16S   key;                                    //用于退出 μC/OS－II 的键
INT8U    x = 0,y = 0;                            //字符显示位置
void  MyTask(void * data);                       //声明任务
void  YouTask(void * data);                      //声明任务
/************************主函数*********************************/
void  main (void)
{
    char * s_M = "M";                            //定义要显示的字符
    OSInit();                                    //初始化 μC/OS－II
    PC_DOSSaveReturn();                          //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                   //安装 μC/OS－II 中断
    OSTaskCreate(
        MyTask,                                  //创建任务 MyTask
        s_M,                                     //给任务传递参数
        &MyTaskStk[TASK_STK_SIZE - 1],           //设置任务堆栈栈顶指针
        0                                        //任务的优先级别为 0
```

```
        );
    OSStart();                                  //启动多任务管理
}

/*******************任务 MyTask ************************************/

void  MyTask (void * pdata)
{
    char * s_Y = "Y";                           //定义要显示的字符
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);           //设置时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                               //初始化统计任务
    OSTaskCreate(
        YouTask,                                //创建任务 YouTask
            s_Y,                                //给任务传递参数
            &YouTaskStk[TASK_STK_SIZE - 1],     //设置任务堆栈栈顶指针
            2                                   //YouTask 的优先级别为 2
        );
    for (;;)
    {
        if (x>50)
        {
            x = 0;
            y += 2;
        }
        PC_DispChar(x, y,                       //字符的显示位置
         *(char *)pdata,
        DISP_BGND_BLACK + DISP_FGND_WHITE );
            x += 1;
        //如果按下 ESC 键,则退出 μC/OS－II
        if (PC_GetKey(&key) == TRUE)
```

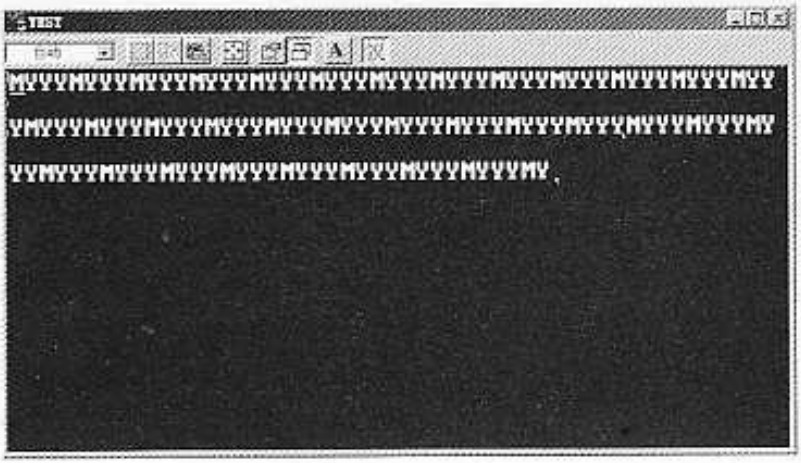
```
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();                 //恢复 DOS 环境
            }
        }
        OSTimeDlyHMSM(0, 0, 3, 0);              //等待 3 s
    }
}

/*************************任务 YouTask ******************************/

void  YouTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        if (x>50)
            {
                x = 0;
                y += 2;
        }
        PC_DispChar(
                x, y,                           //字符的显示位置
                *(char *)pdata,
                DISP_BGND_BLACK + DISP_FGND_WHITE
                );
            x += 1;
        OSTimeDlyHMSM(0, 0, 1, 0);              //等待 1 s
    }
}

/*************************End ************************************/
```

例 2－7 应用程序运行后的结果如图 2－13 所示。


MYYYMYYYMYYYM

当函数调用成功时，返回信息 OS_NO_ERR；否则根据出错的具体情况返回 OS_TASK_SUSPEND_IDLE、OS_PRIO_INVALID 和 OS_TASK_SUSPEND_PRIO 等。

从图 2-15 所示函数 OSTaskSuspend()流程图中可知，该函数在一系列的判断中主要是判断待挂起的任务是否调用这个函数的任务本身。如果是任务本身，则必须删除任务在任务就绪表中的就绪标志，并在任务控制块成员 OSTCBStat 中做了挂起记录之后，引发一次任务调度，以使 CPU 去运行就绪的其他任务。如果待挂起的任务不是调用函数的任务本身，而是其他任务，那么只要删除任务就绪表中被挂起任务的就绪标志，并在任务控制块成员 OSTCBStat 中做了挂起记录即可。

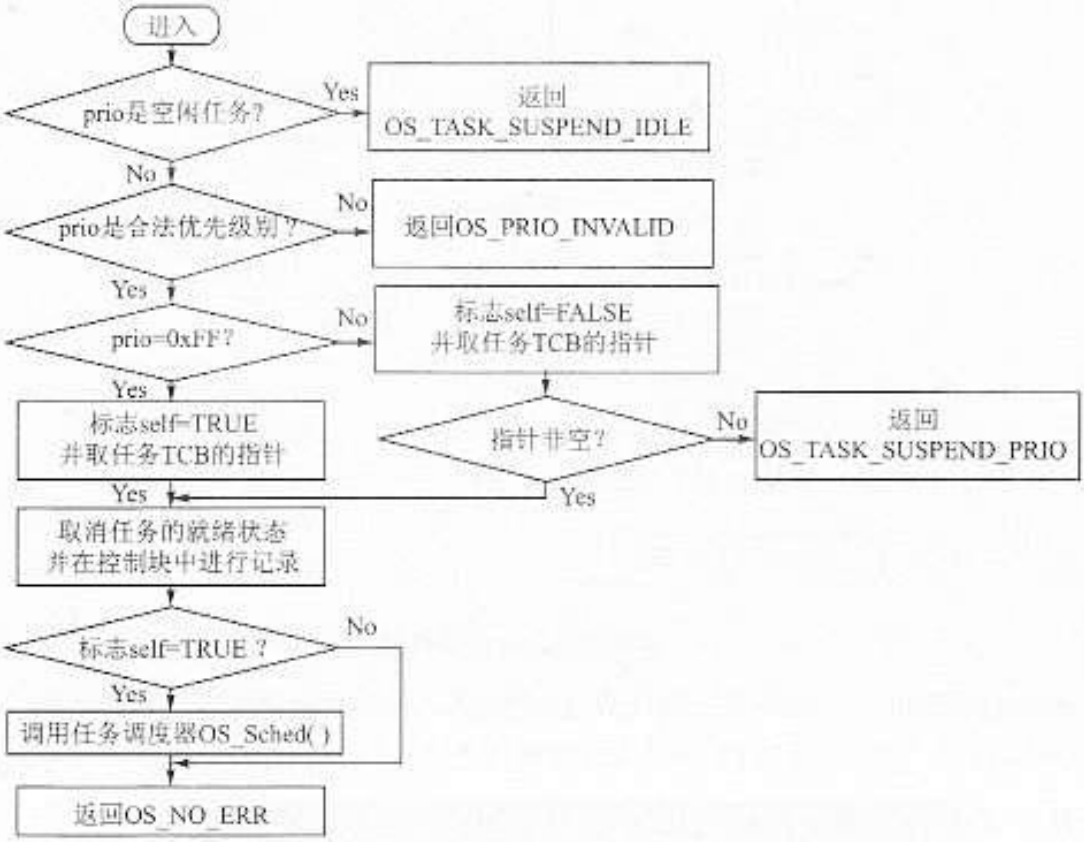


图 2-15 函数 OSTaskSuspend()的流程图

2.6.2 恢复任务

恢复任务函数 OSTaskResume()的原型如下：

```
INT8U  OSTaskResume (INT8U prio);
```

函数的参数为待恢复任务的优先级别。若函数调用成功，则返回信息 OS_NO_ERR；否则，根据出错的具体情况返回 OS_PRIO_INVALID、OS_TASK_RESUME_PRIO 和 OS_TASK_NOT_SUSPEND 等。

从图 2－16 所示函数 OSTaskResume()的流程图可知，函数在判断任务确实是一个已存在的挂起任务，同时它又不是一个等待任务（任务控制块成员 OSTCBDly＝0）时，就清除任务控制块成员 OSTCBStat 中的挂起记录并使任务就绪，最后调用调度器 OSSched()进行任务调度，并返回函数调用成功的信息 OS_NO_ERR。

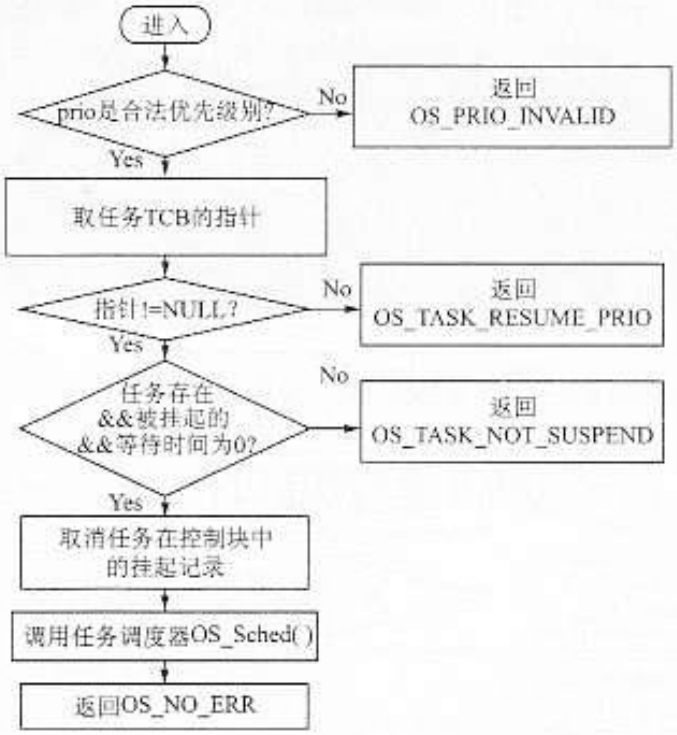


图 2－16　函数 OSTaskResume()的流程图

例 2－8　修改例 2－7 应用程序的任务 YouTask。要求任务 YouTask 运行 20 次后，挂起任务 MyTask；任务 YouTask 运行 40 次后，恢复任务 MyTask。

答　任务 YouTask 修改后的代码如下（其中**加粗字体**的代码为添加部分）：

```
/*************************任务 YouTask ******************************/
UINT8U time = 0;                          //任务 YouTask 的运行次数
void  YouTask (void * pdata)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif
    pdata = pdata;
    for (;;)
```

```
    {
        if(time == 20)
        {
            OSTaskSuspend(0);                          //挂起任务 MyTask
        }
        if(time == 40)
        {
            OSTaskResume(0);                           //恢复任务 MyTask
        }
        time += 1 ;

        if (x>50)
        {
            x = 0;
            y += 2;
        }

        PC_DispChar(x, y,                              //字符的显示位置
                    *(char *)pdata,
                    DISP_BGND_BLACK + DISP_FGND_WHITE );
                      x += 1;
                    OSTimeDlyHMSM(0, 0, 1, 0);         //等待 1 s
    }
}
/*************************End ****************************************/
```

例 2－8 应用程序的运行结果见图 2－17。

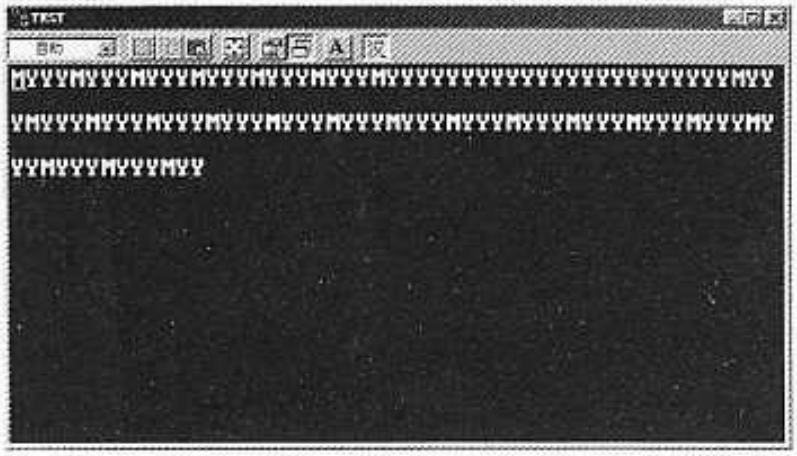


图 2－17　例 2－8 应用程序的运行结果

例 2－9 改造例 2－7 的任务 MyTask，当任务 MyTask 运行 10 次时，用函数 OSSchedLock()对调度器进行加锁；而当任务 MyTask 运行到第 80 次时，再用函数 OSSchedUnlock()对调度器进行解锁，并运行该程序。

答 应用程序的代码如下：

```
/*************************Test ************************************/
#include "includes.h"
#define  TASK_STK_SIZE    512                    //任务堆栈长度
OS_STK   MyTaskStk[TASK_STK_SIZE];               //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];              //定义任务堆栈区
INT16S   key;                                    //用于退出 μC/OS－II 的键
INT8U    x = 0,y = 0;                            //字符显示位置
INT8U    times = 0;                              //运行次数
void  MyTask(void *data);                        //声明任务
void  YouTask(void *data);                       //声明任务
/*************************主函数 ************************************/
void  main (void)
{
    char* s_M = "M";                             //定义要显示的字符
    OSInit();                                    //初始化 μC/OS－II
    PC_DOSSaveReturn();                          //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                   //安装 μC/OS－II 中断
    OSTaskCreate(MyTask,                         //创建任务 MyTask
        s_M,                                     //给任务传递参数
        &MyTaskStk[TASK_STK_SIZE - 1],           //设置任务堆栈栈顶指针
        0);                                      //使任务优先级别为 0
    OSStart();                                   //启动 μC/OS－II 的多任务管理
}

/*************************任务 MyTask ************************************/

void  MyTask (void *pdata)
{
    char* s_Y = "Y";                             //定义要显示的字符
#if OS_CRITICAL_METHOD == 3
```

```
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                    //安装 μC/OS-II 时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);               //设置 μC/OS-II 时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                                   //初始化 μC/OS-II 的统计任务
    OSTaskCreate(YouTask,                           //创建任务 YouTask
                 s_Y,                               //给任务传递参数
                 &YouTaskStk[TASK_STK_SIZE - 1],    //设置堆栈栈顶指针
                 2);                                //使任务的优先级别为 2
    for (;;)
    {
        if (x>50)
        {
            x = 0;
            y += 2;
        }
        times += 1;
        if (times == 10)
        {
            OSSchedLock();                          //锁调度器
        }
        if (times == 80)
        {
            OSSchedUnlock();                        //解锁调度器
        }
        PC_DispChar(x, y,                           //显示字符的位置
                    *(char *)pdata,
                    DISP_BGND_BLACK + DISP_FGND_WHITE );
            x += 1;
        //如果按下 ESC 键,则退出 μC/OS-II
        if (PC_GetKey(&key) == TRUE)
        {
```

```
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDlyHMSM(0, 0, 3, 0);                    //等待
    }
}
/*************************任务 YouTask ******************************/
void  YouTask (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
        {
        if (x>50)
        {
                x = 0;
                y += 2;
        }
        PC_DispChar(x, y,                             //显示字符的位置
        *(char *)pdata,
        DISP_BGND_BLACK + DISP_FGND_WHITE );
                x += 1;
        OSTimeDlyHMSM(0, 0, 1, 0);                    //等待
    }
}
/*************************End ************************************/
```

例 2－9 应用程序的运行结果如图 2－18 所示。

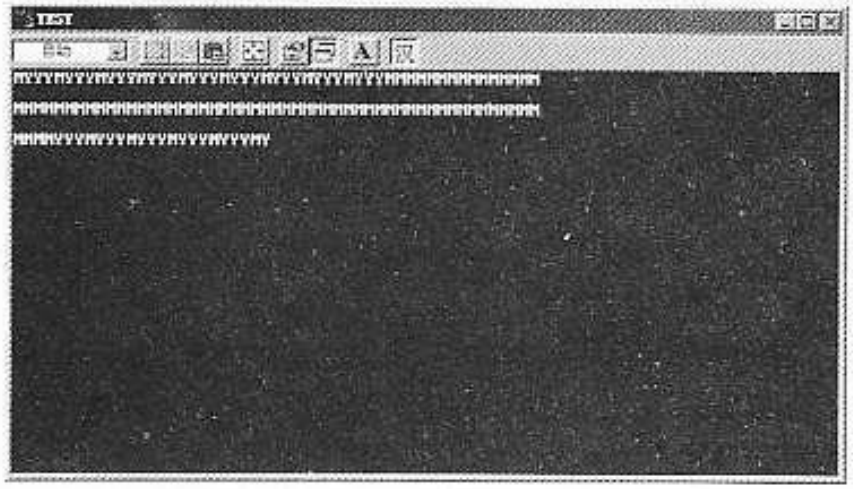


图 2－18　例 2－9 应用程序的运行结果

2.7　其他任务管理函数

2.7.1　任务优先级别的修改

每一个任务都必须有一个优先级别，但这个优先级别并不是一成不变的。在程序的运行过程中，任务可根据需要通过调用函数 OSTaskChangePrio()来改变任务的优先级别。

函数 OSTaskChangePrio()的原型如下：

```
INT8U OSTaskChangePrio (
                        INT8U oldprio,      //任务现在的优先级别
                        INT8U newprio       //要修改的优先级别
                       );
```

若调用函数 OSTaskChangePrio()成功，则函数返回 OS_NO_ERR。

2.7.2　任务的删除

所谓删除一个任务，就是把该任务置于睡眠状态。具体做法是，把被删除任务的任务控制块从任务控制块链表中删除，并归还给空任务控制块链表，然后在任务就绪表中把该任务的就绪状态位置 0，于是该任务就不能再被调度器调用了。简单地说，就是把它的身份证给吊销了。

在任务中，可以通过调用函数 OSTaskDel()来删除任务自身或者除了空闲任务之外的其他任务。

函数 OSTaskDel()的原型如下：

```
#if OS_TASK_DEL_EN
INT8U OSTaskDel (
                INT8U prio          //要删除任务的优先级别
                );
```

如果一个任务调用这个函数是为了删除任务自己，则应在调用函数时令函数的参数 prio 为 OS_PRIO_SELF。

有时，任务会占用一些动态分配的内存或信号量之类的资源。这时，如果有其他任务把这个任务删除了，那么被删除任务所占用的一些资源就会因为没有被释放而丢失，这是任何系统都无法接受的。因此，在删除一个占用资源的任务时，一定要谨慎。具体的办法是，提出删除任务请求的任务只负责提出删除任务请求，而删除工作则由被删除任务自己来完成。这样，被删除任务就可以根据自身的具体情况来决定何时删除自身，同时也有机会在删除自身之前把占用的资源释放掉。

显然，如果想使提出删除任务请求的任务和被删除任务之间，能够像上述方式来执行删除工作，则它们双方必须有某种通信方法。μC/OS－II 利用被删除任务的任务控制块成员 OSTCBDelReq 作为请求删除方的被删除方的联络信号，同时提供了一个双方都能调用的函数——请求删除任务函数 OSTaskDelReq()。这样，提出删除任务请求的任务和被删除任务的双方就都使用该函数来访问 OSTCBDelReq 这个信号，从而可以根据这个信号的状态来决定各自的行为。

函数 OSTaskDelReq()的原型如下：

```
INT8U  OSTaskDelReq(
                    INY8U prio      //待删除任务的优先级别
                    );
```

提出删除任务请求的任务在调用这个函数时，函数的参数应该为被删除任务的优先级别 prio；被删除任务在调用这个函数时，函数的参数应该为 OS_PRIO_SELF。

函数 OSTaskDelReq()的流程图如图 2－19 所示。

删除任务请求方要用被删除任务的优先级别 prio 作为参数来调用这个函数。从图 2－19 中可以看出，删除任务请求方调用这个函数的目的就是要查看被删除的任务控制块是否还在。如果在，则令被删除任务的任务控制块成员 OSTCBDelReq 的值为 OS_TASK_DEL_REQ，且通知该任务："已经有任务要求在合适的时候删除自己"；如果不在，则认为被删除任务已经被删除。

例如，任务请求删除优先级别为 44 的任务，那么完成这个任务的代码段如下：

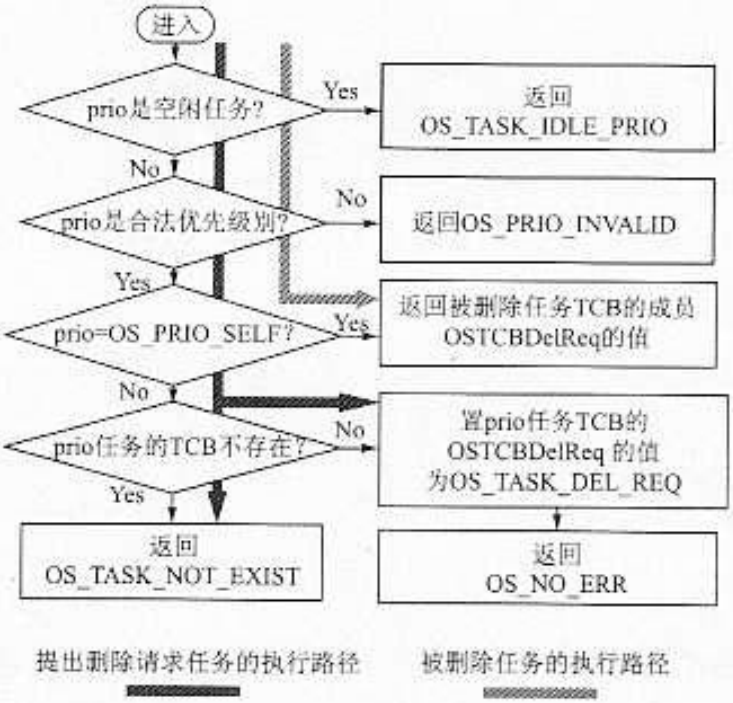


图 2－19　函数 OSTaskDelReq()的流程图

```
while( OSTaskDelReq( 44 ) != OS_TASK_NOT_EXIST )
{
    OSTimeDly(1);                    //延时 1 个时钟节拍
}
```

即通过不断地调用函数 OSTaskDelReq()来查询优先级别为 44 的任务是否还存在。只要还存在，就调用延时函数 OSTimeDly()等待，直到发现被删除任务不存在了(即被删除了)才继续运行。

被删除任务方一定要用 OS_PRIO_SELF 作为参数来调用函数 OSTaskDelReq()。从图 2－19 中可以看到，函数判断出参数是 OS_PRIO_SELF 时，将会返回任务 TCB 的域 OSTCBDelReq 的值。如果该值为 OS_TASK_DEL_REQ，意味着有其他任务发出了删除任务请求，那么被删除任务就应该在适当的时候调用函数 OSTaskDel(OS_PRIO_SELF)来删除自己。

被删除任务方调用函数 OSTaskDelReq()的典型代码段如下：

```
if( OSTaskDelReq(OS_PRIO_SELF ) == OS_TASK_DEL_REQ )
{
    //释放资源和动态内存的代码
    OSTaskDel( OS_PRIO_SELF );
```

```
}
else
{
    //其他应用代码
}
```

例 2－10 修改例 2－7 应用程序，使任务 MyTask 能删除任务 YouTask。

答 修改后的应用程序代码如下：

```
/*********************************************************************/
#include "includes.h"

#define  TASK_STK_SIZE    512                          //任务堆栈长度

OS_STK   MyTaskStk[TASK_STK_SIZE];                     //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];                    //定义任务堆栈区
INT16S   key;                                          //用于退出 μC/OS－II 的键
INT8U    x = 0,y = 0;                                  //字符显示位置

void  MyTask(void *data);                              //声明任务
void  YouTask(void *data);                             //声明任务
/************************主函数*************************************/
void  main (void)
{
    char* s_M = "M";                                   //定义要显示的字符
    OSInit();                                          //初始化 μC/OS－II

    PC_DOSSaveReturn();                                //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                         //安装 μC/OS－II 中断

    OSTaskCreate(MyTask,                               //创建任务 MyTask
        s_M,                                           //给任务传递参数
        &MyTaskStk[TASK_STK_SIZE - 1],                 //设置任务堆栈栈顶指针
        0);                                            //使任务优先级别为 0

    OSStart();                                         //启动 μC/OS－II 的多任务管理
```

```
}

/************************ 任务 MyTask ********************************/

void  MyTask (void * pdata)
{
    char * s_Y = "Y";                                   //定义要显示的字符
    char * s = "MyTask:我要求 YouTask 自己删除自己!";    //定义字符串
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif

    pdata = pdata;

    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                        //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);                   //设置 μC/OS－II 时钟频率
    OS_EXIT_CRITICAL();

    OSStatInit();                                       //初始化统计任务
    OSTaskCreate(YouTask,                               //创建任务 YouTask
        s_Y,                                            //给任务传递参数
        &YouTaskStk[TASK_STK_SIZE - 1],                 //设置任务堆栈栈顶指针
        2);                                             //使任务的优先级别为 2
    for (;;)
    {
        if (x>30)
        {
            while( OSTaskDelReq( 2 ) != OS_TASK_NOT_EXIST )
            {
                PC_DispStr(
                    10,8,
                    s,                                  //显示字符串
                    DISP_FGND_YELLOW + DISP_BGND_BLUE
```

```
                );
                OSTimeDly(1);                          //延时一个时钟节拍
            }
        }

        if (x>50)
        {
            x=0;
            y+=2;
        }

        PC_DispChar(x, y,                              //字符的显示位置
        *(char*)pdata,
        DISP_BGND_BLACK+DISP_FGND_WHITE);
            x += 1;

        //如果按下 ESC 键,则退出 μC/OS-II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }

        OSTimeDlyHMSM(0, 0, 3, 0);                     //等待
    }
}
/*************************任务 YouTask ******************************/
void  YouTask (void *pdata)
{
    char *s1="YouTask:我必须要删除我自己了!";          //定义字符串
    char *s2="YouTask:我已经删除我自己了!";            //定义字符串
```

```
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif

    pdata = pdata;

    for (;;)
    {
        if( OSTaskDelReq(OS_PRIO_SELF ) == OS_TASK_DEL_REQ )
            {
                PC_DispStr(10,10,s1,                    //显示字符串
                    DISP_FGND_WHITE + DISP_BGND_BLACK);
                OSTimeDlyHMSM(0, 0, 15, 0);             //延时 15 s
            PC_DispStr(10,12,s2,                        //显示字符串
                    DISP_FGND_WHITE + DISP_BGND_BLACK);
            OSTaskDel( OS_PRIO_SELF );                  //删除任务自身
        }
        if (x>50)
        {
            x = 0;
            y += 2;
        }

        PC_DispChar(
            x, y,                                       //字符的显示位置
            *(char *)pdata,
            DISP_BGND_BLACK + DISP_FGND_WHITE
                );
            x += 1;
        OSTimeDlyHMSM(0, 0, 1, 0);                      //等待 1 s
    }
}
/*************************End*************************************/
```

例 2－10 应用程序的运行结果见图 2－20。

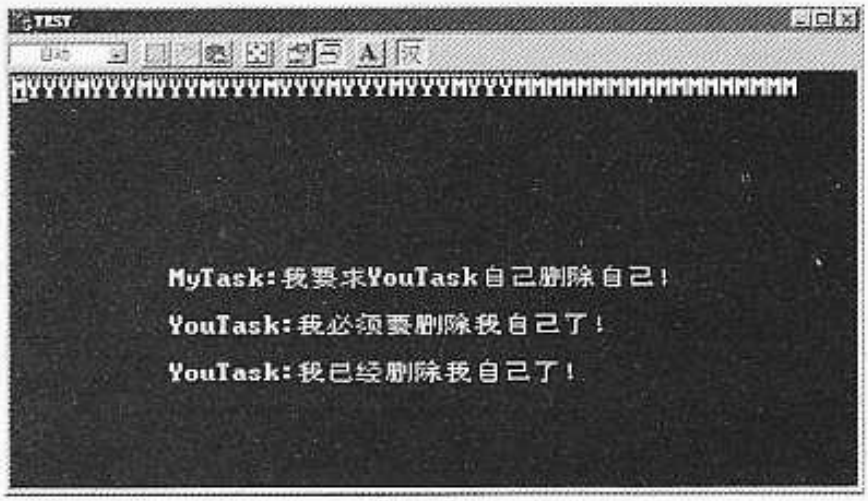


图 2－20　例 2－10 应用程序的运行结果

2.7.3　查询任务的信息

有时，在应用程序运行中需要了解一个任务的指针、堆栈等信息，这时就可以通过调用函数 OSTaskQuery()来获取选定的任务的信息。函数 OSTaskQuery()的原型如下：

```
INT8U  OSTaskQuery(
                    INT8U prio,          //待查询任务的优先级别
                    OS_TCB *pdata        //存储任务信息的结构
                   );
```

若调用函数 OSTaskQuery()查询成功，则函数将返回 OS_NO_ERR，并把查询得到的任务信息存放在结构 OS_TCB 类型的变量中。

2.8　μC/OS－II 的初始化和任务的启动

2.8.1　μC/OS－II 的初始化

在使用 μC/OS－II 的所有服务之前，必须调用 μC/OS－II 的初始化函数 OSInit()，对 μC/OS－II 自身的运行环境进行初始化。

函数 OSInit()将对 μC/OS－II 的所有全局变量和数据结构进行初始化，同时创建空闲任务 OSTaskIdle，并赋之以最低的优先级别和永远的就绪状态。如果用户应用程序还要使用统计任务(常数 OS_TASK_STAT_EN＝1)，则 OSInit()还要以优先级别为 OS_LOWEST_

PRIO－1 来创建统计任务。

初始化函数 OSInit()对数据结构进行初始化时，主要是创建包括空任务控制块链表在内的 5 个空数据缓冲区。同时，为了可以快速地查询任务控制块链表中的各个元素，初始化函数 OSInit()还要创建一个数组 OSTCBPrioTbl[OS_LOWEST_PRIO + 1]。在这个数组中，按任务优先级别的顺序把任务控制块的指针存放在对应的元素中。经过初始化之后，系统中的数据结构如图 2－21 所示。

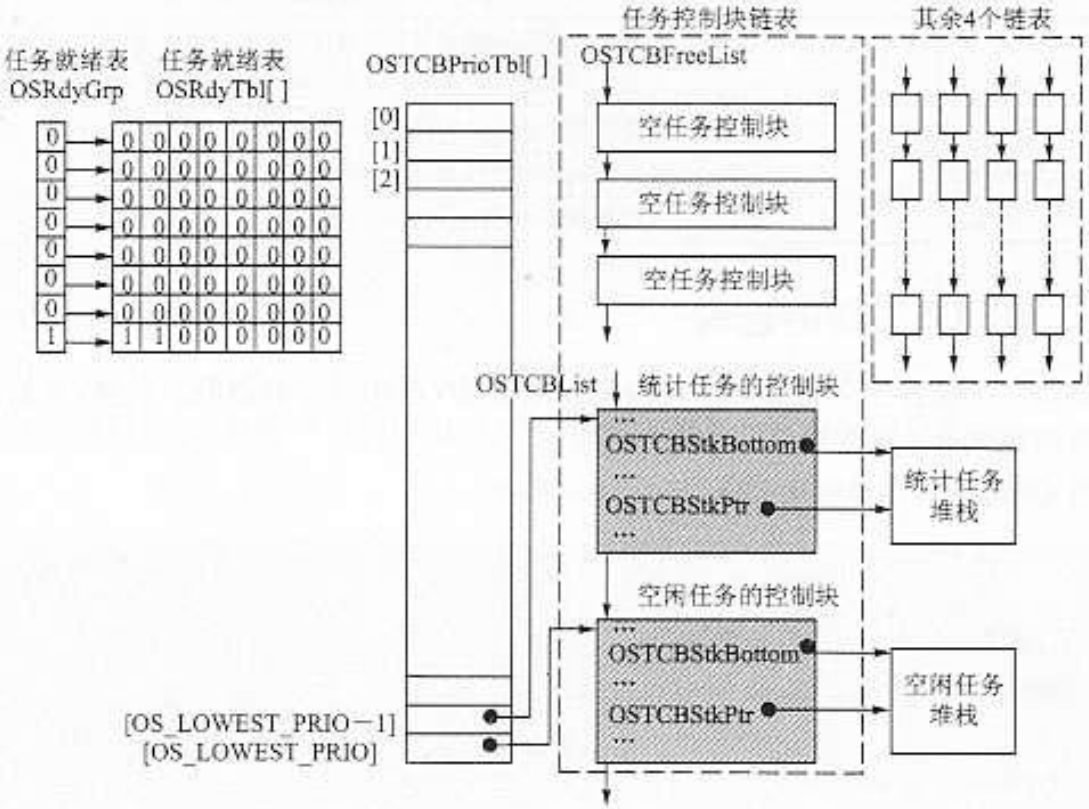


图 2－21　μC/OS－II 初始化后的数据结构

初始化之后各全局变量的情况见表 2－4。

表 2－4　初始化之后各全局变量的情况

变　量	值	变量的说明
OSPrioCur	0	类型为 INT8U，正在运行的任务的优先级
OSPrioHighRdy	0	类型为 INT8U，具有最高优先级别的就绪任务的优先级
OSTCBCur	NULL	类型为 OS_TCB *，指向正在运行任务控制块的指针
OSTCBHighRdy	NULL	类型为 OS_TCB *，指向最高级优先级就绪任务控制块的指针
OSTime	0L	类型为 INT32U，表示系统当前时间(节拍数)
OSIntNesting	0	类型为 INT8U，存放中断嵌套的层数(0～255)

续表 2－4

变　量	值	变量的说明
OSLockNesting	0	类型为 INT8U，调用了 OSSchedLock 的嵌套层数
OSCtxSwCtr	0	类型为 INT32U，上下文切换的次数
OSTaskCtr	2	类型为 INT8U，已经建立了的任务数
OSRunning	FALSE	类型为 BOOLEAN，μC/OS－II 核是否正在运行的标志
OSCPUUsage	0	类型为 INT8S，存放 CPU 的利用率(%)的变量
OSIdleCtrMax	0L	类型为 INT32U，表示每秒空闲任务计数的最大值
OSIdleCtrRun	0L	类型为 INT32U，表示空闲任务计数器每秒的计数值
OSIdleCtr	0L	类型为 INT32U，空闲任务的计数器
OSStatRdy	FALSE	类型为 BOOLEAN，统计任务是否就绪的标志
OSInitExity	0	类型为 INT8U，用于函数 OSInitExt()

2.8.2　μC/OS－II 的启动

μC/OS－II 进行任务的管理是从调用启动函数 OSStart()开始的。当然，其前提条件是在调用该函数之前至少创建了一个用户任务。

函数 OSStart()的源代码如下：

```
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();
    }
}
```

假如有代码如下：

```
#include "includes.h"
#define MY_TASK_STK_SIZE      512          //定义堆栈容量

void main( void )
{
    OSInit();                              //初始化 μC/OS－II
    ......
    OSTaskCreate( MyTask,                  //创建任务 MyTask
                  (void * )0,
                  &MyTaskStk[MY_TASK_STK_SIZE－1],
                  6                        //任务的优先级别为 6
                  );
    OSStart();                             //启动 μC/OS－II
}
```

那么对于上面的代码来说，主函数 main()调用了函数 OSStart()之后，μC/OS－II 立即进入了多任务管理阶段，这时的数据结构如图 2－22 所示，而各个变量的变化见表 2－5。

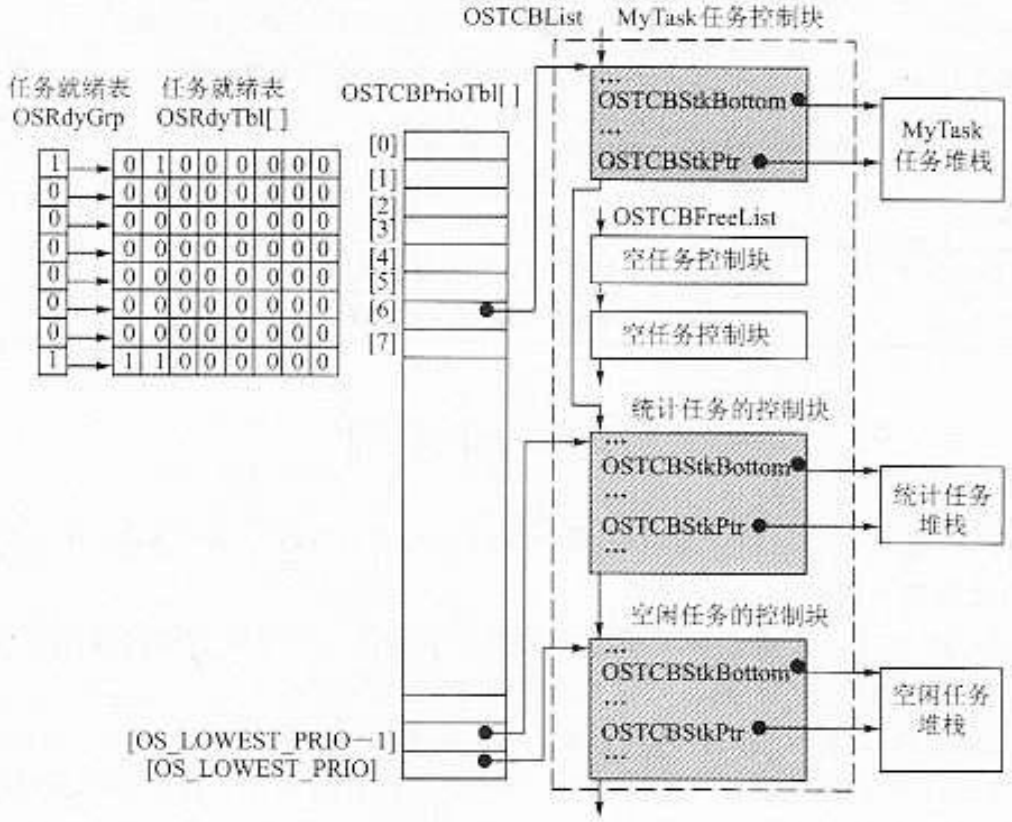


图 2－22　调用函数 OSStart()后的数据结构

OSStartHighRdy()在多任务系统启动函数 OSStart()中调用。实现的功能是，设置系统运行标志位 OSRunning＝TRUE，将就绪表中最高优先级任务的栈指针加载到 SP 中，并强制中断返回。这样就绪的最高优先级任务就如同从中断里返回到运行态一样，使得整个系统得以运转。

表 2－5　初始化之后各个全局变量的值

变　量	值	说　明
OSPrioCur	6	类型为 INT8U，正在运行的任务的优先级
OSPrioHighRdy	6	类型为 INT8U，具有最高优先级别的任务的优先级
OSTCBCur		类型为 OS_TCB *，指向当前任务控制块的指针
OSTCBHighRdy		类型为 OS_TCB *，指向最高级优先级任务控制块的指针
OSTime	0L	类型为 INT32U，表示系统当前时间(节拍数)
OSIntNesting	0	类型为 INT8U，表示存放中断嵌套的层数(0～255)
OSLockNesting	0	类型为 INT8U，表示调用了 OSSchedLock 的嵌套层数
OSCtxSwCtr	0	类型为 INT32U，表示上下文切换的次数
OSTaskCtr	3	类型为 INT8U，表示已经建立了的任务数
OSRunning	TRUE	类型为 BOOLEAN，表示正在运行多任务的标志
OSCPUUsage	0	类型为 INT8S，表示存放 CPU 的利用率(%)
OSIdleCtrMax	0L	类型为 INT32U，表示每秒空闲任务计数的最大值
OSIdleCtrRun	0L	类型为 INT32U，表示空闲任务计数器每秒的计数值
OSIdleCtr	0L	类型为 INT32U，空闲任务的计数器
OSStatRdy	FALSE	类型为 BOOLEAN，统计任务就绪的标志
OSIntExitY	0	类型为 INT8U，用于系统函数 OSIntExit()

2.9　一个示例

例 2－11　建立一个起始任务 TaskStart，由它再建立 4 个任务 Task1、Task2、Task3 和 TaskClk 来完成如下任务：

- 由 Task2 和 Task3 各自来完成一个动画显示任务：在屏幕上依次输出字符"|"、"\"、"－"、"/"，而产生一个向左转动和一个向右的动画标志；
- 由 TaskClk 来获取系统时间并显示；
- 由 Task1 来初始化屏幕，创建其他 4 个任务和检查所有 5 个任务的任务堆栈使用情况并显示出来。

应用程序代码如下：

```
//定义常量-----------------------------------------------------------------
#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_START_PRIO        10
#define          TASK_CLK_PRIO          11
#define          TASK_1_PRIO            12
#define          TASK_2_PRIO            13
#define          TASK_3_PRIO            14
//声明变量和函数-----------------------------------------------------------
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];

        void  TaskStart(void *data);                    //声明起始任务
static  void  TaskStartCreateTasks(void);               //声明创建任务函数
static  void  TaskStartDispInit(void);                  //声明屏幕初始化函数
static  void  TaskStartDisp(void);
        void  TaskClk(void *data);                      //声明计时任务
        void  Task1(void *data);                        //声明任务1~3
        void  Task2(void *data);
        void  Task3(void *data);
//主函数-------------------------------------------------------------------
void main (void)
{
    OS_STK *ptos;
    OS_STK *pbos;
    INT32U  size;

    PC_DispClrScr(DISP_FGND_WHITE);                     //清屏
    OSInit();
```

```
    PC_DOSSaveReturn();                                  //保存 DOS
    PC_VectSet(uCOS, OSCtxSw);                           //设置 μC/OS 中断向量

    PC_ElapsedInit();

    ptos      = &TaskStartStk[TASK_STK_SIZE - 1];        //起始任务堆栈栈顶
    pbos      = &TaskStartStk[0];                        //起始任务堆栈栈底
    size      = TASK_STK_SIZE;                           //起始任务堆栈长度
    OSTaskStkInit_FPE_x86(&ptos, &pbos, &size);
    OSTaskCreateExt(TaskStart,                           //创建起始任务
                    (void *)0,
                    ptos,
                    TASK_START_PRIO,
                    TASK_START_ID,
                    pbos,
                    size,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK
                    | OS_TASK_OPT_STK_CLR);
    OSStart();                                           //启动 μC/OS
}
//起始任务-----------------------------------------------------------------
void  TaskStart (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif

    INT16S     key;

    pdata = pdata;

    TaskStartDispInit();                                 //初始化屏幕
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                         //安装系统时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);                    //设置节拍
    OS_EXIT_CRITICAL();
```

```
    OSStatInit();                                   //初始化统计任务
    TaskStartCreateTasks();                         //创建其他任务
    for (;;) {
        TaskStartDisp();                            //调用刷新屏幕函数
        if (PC_GetKey(&key)) {
            if (key == 0x1B) {
                PC_DOSReturn();
            }
        }
        OSCtxSwCtr = 0;
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
//屏幕初始化函数-----------------------------------------------------------------
static  void  TaskStartDispInit (void)
{
    ......
}
//屏幕刷新函数-------------------------------------------------------------------
static  void  TaskStartDisp (void)
{
    char   s[80];

    sprintf(s, "%5d", OSTaskCtr);
    PC_DispStr(18, 22, s, DISP_FGND_YELLOW + DISP_BGND_BLUE);
    sprintf(s, "%3d", OSCPUUsage);
    PC_DispStr(36, 22, s, DISP_FGND_YELLOW + DISP_BGND_BLUE);
    sprintf(s, "%5d", OSCtxSwCtr);
    PC_DispStr(18, 23, s, DISP_FGND_YELLOW + DISP_BGND_BLUE);
    sprintf(s, "V%4.2f", (float)OSVersion() * 0.01);
    PC_DispStr(75, 24, s, DISP_FGND_YELLOW + DISP_BGND_BLUE);
    switch (_8087) {
        case 0:
            PC_DispStr(71, 22, " NO  FPU ",
                       DISP_FGND_YELLOW + DISP_BGND_BLUE);
            break;
        case 1:
```

```
            PC_DispStr(71, 22, " 8087 FPU",
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
            break;
        case 2:
            PC_DispStr(71, 22, "80287 FPU",
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
            break;
        case 3:
            PC_DispStr(71, 22, "80387 FPU",
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
            break;
    }
}
//创建任务函数-----------------------------------------------------------------
static  void  TaskStartCreateTasks (void)
{
    OSTaskCreateExt(TaskClk,
                    (void * )0,
                    &TaskClkStk[TASK_STK_SIZE - 1],
                    TASK_CLK_PRIO,
                    TASK_CLK_ID,
                    &TaskClkStk[0],
                    TASK_STK_SIZE,
                    (void * )0,
                    OS_TASK_OPT_STK_CHK
                                | OS_TASK_OPT_STK_CLR);

    OSTaskCreateExt(Task1,                          //创建任务 1
                    (void * )0,
                    &Task1Stk[TASK_STK_SIZE - 1],
                    TASK_1_PRIO,
                    TASK_1_ID,
                    &Task1Stk[0],
                    TASK_STK_SIZE,
                    (void * )0,
                    OS_TASK_OPT_STK_CHK
                                | OS_TASK_OPT_STK_CLR);
```

```
    OSTaskCreateExt(Task2,                                  //创建任务 2
                    (void *)0,
                    &Task2Stk[TASK_STK_SIZE - 1],
                    TASK_2_PRIO,
                    TASK_2_ID,
                    &Task2Stk[0],
                    TASK_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK
                                | OS_TASK_OPT_STK_CLR);

    OSTaskCreateExt(Task3,                                  //创建任务 3
                    (void *)0,
                    &Task3Stk[TASK_STK_SIZE - 1],
                    TASK_3_PRIO,
                    TASK_3_ID,
                    &Task3Stk[0],
                    TASK_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK
                                | OS_TASK_OPT_STK_CLR);
}
//任务 1-----------------------------------------------------------------------
void  Task1 (void *pdata)
{
    INT8U       err;
    OS_STK_DATA data;
    INT16U      time;
    INT8U       i;
    char        s[80];

    pdata = pdata;

    for (;;) {
        for (i = 0; i < 5; i++) {
            PC_ElapsedStart();                              //开始测试
            //调用 μC/OS 的测试函数
```

```
            err   = OSTaskStkChk(TASK_START_PRIO + i, &data);
            time = PC_ElapsedStop();                    //结束测试
            if (err == OS_NO_ERR) {
                sprintf(s, "%4ld          %4ld          %4ld          %6d",
                        data.OSFree + data.OSUsed,
                        data.OSFree,
                        data.OSUsed,
                        time);                          //输出结果
                PC_DispStr(19, 12 + i, s,
                        DISP_FGND_BLACK + DISP_BGND_LIGHT_GRAY);
            }
        }
        OSTimeDlyHMSM(0, 0, 0, 100);
    }
}
//任务2动画显示----------------------------------------------------------------
void  Task2 (void *data)
{
    data = data;

    for (;;) {
        PC_DispChar(70, 15, '|',
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
        PC_DispChar(70, 15, '/',
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
        PC_DispChar(70, 15, '-',
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
        PC_DispChar(70, 15, '\\',
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
    }
}
//任务3动画显示----------------------------------------------------------------
void  Task3 (void *data)
```

```
{
    char    dummy[200];
    INT16U  i;

    data = data;
    for (i = 0; i < 199; i++) {
        dummy[i] = '?';
    }
    for (;;) {
        PC_DispChar(70, 16, '|',
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
        PC_DispChar(70, 16, '\\',
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
        PC_DispChar(70, 16, '-',
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
        PC_DispChar(70, 16, '/',
                    DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
    }
}
//计时任务-----------------------------------------------------------------
void  TaskClk (void *data)
{
    char s[40];

    data = data;
    for (;;) {
        PC_GetDateTime(s);
        PC_DispStr(60, 23, s, DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
//-------------------------------------------------------------------------
```

应用程序说明：

- 由于要对任务堆栈进行检查，因此在本程序创建任务时用到了函数 OSTaskCreateExt()。
- 在应用程序的起始任务 TaskStart()中，为了初始化显示屏幕，调用了自编的函数 TaskStartDispInit()，由于这个函数的代码太长，故在此省略，请读者阅读光盘中的代码。
- 在应用程序的起始任务 TaskStart()中，为了创建其余 4 个任务，调用了自编的函数 TaskStartCreateTasks()。
- 应用程序的起始任务 TaskStart()，通过调用系统提供的任务堆栈检查函数 OSTaskStkChk()，对应用程序中的 5 个任务堆栈的使用情况进行了检查。
- 在应用程序的起始任务 TaskStart()中，当数据发生变化时，为了更新屏幕，调用了自编的屏幕刷新函数 TaskStartDisp()。
- 在任务 Task3()中定义了一个数组，用以模仿任务堆栈被占用后的情况。

本应用程序的运行结果见图 2－23。

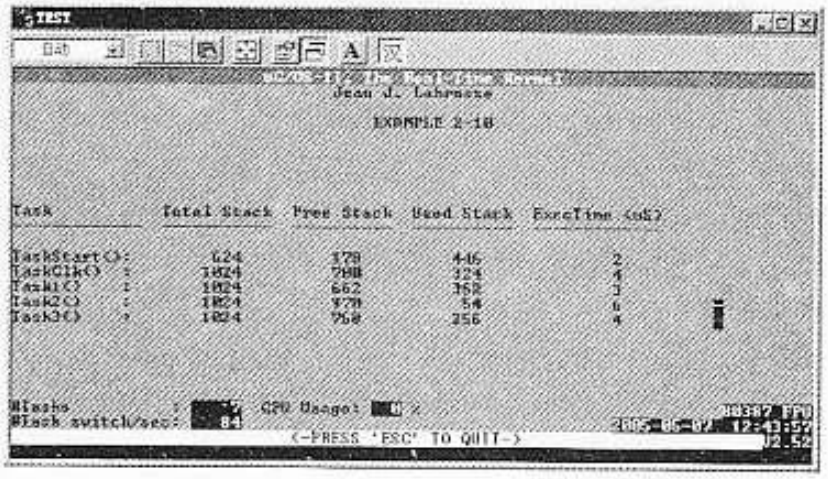


图 2－23　例 2－11 应用程序的运行结果

2.10　小　结

- 任务由任务控制块、任务堆栈和任务代码三部分组成。系统通过任务控制块来感知和控制任务；任务堆栈主要用来保护断点和恢复断点；任务代码是一个超循环结构，它描述了任务的执行过程。在创建一个任务时，函数 OSTaskCreate()或 OSTaskCreateExt()负责给任务分配任务控制块和任务堆栈，并对它们进行初始化，然后把任务控制块、任务堆栈、任务代码三者关联起来形成一个完整任务。
- 系统是按任务就绪表和任务的优先级别来调度任务的。执行任务调度工作的是调度器，它负责查找具有最高优先级别的就绪任务并运行它，同时把这个任务 TCB 的指针

存放在OSTABCur中。通常，系统在调用API函数和运行中断服务程序之后都要调用函数OSSched()来进行一次任务调度。

- 任务切换的核心工作是任务堆栈指针的切换。
- 任务调度器代码的设计，使得它的运行时间与系统中的任务数无关，从而使它满足了实时系统的要求。
- 任务的优先级别也是任务的标识。
- 应用程序首先应该调用函数OSInit()对全局变量和数据结构进行初始化，以建立μC/OS-II的运行环境。
- 应用程序是通过调用函数OSStart()开始进入多任务管理的，但在调用函数OSStart()之前，必须至少创建了一个任务。

2.11 练习题

1. 什么是可剥夺型内核？
2. 一个应用程序为什么一定要使用空闲任务？
3. 统计任务是必须使用的吗？
4. 什么叫做任务的优先权？μC/OS-II是用什么来描述任务的优先权的？
5. 在μC/OS-II中任务有哪5种状态？
6. 任务控制块记录了任务的哪些信息？
7. 什么是空任务控制块链表？什么是任务控制块链表？
8. 数组OSTCBTbl[ ]有什么用途？
9. 正在运行任务的任务控制块指针存放在哪个指针变量中？
10. 变量OSRdyGrp有什么用？
11. 简述在任务就绪表中查找具有最高优先级别的就绪任务的过程。
12. 试对例2-8与例2-9两个应用程序的结果进行比较和分析。
13. 编写一个有3个任务的应用程序，每个任务均会在显示器上显示一个字符，并让3个任务具有不同的等待时间，观察应用程序运行中任务被调度的情况。
14. 编写一个有3个任务的应用程序，每个任务均会在显示器上显示一个字符，并让1个任务查询另外2个任务的信息，并在显示器上显示出来。
15. 编写一个有3个任务的应用程序，并让其中2个任务在你认为合适的时候删除自己。
16. 编写一个有3个任务的应用程序，在调度器每进行一次任务切换后，在显示器上显示正在运行任务的任务控制块指针。
17. 编写一个有2个任务的应用程序，每一个任务每次运行显示一个字符。当调度器进行5次调度之后，这些显示的字符会在显示器上构成一个字符串"Hello，μC/OS-II!"。

第3章 μC/OS－II的中断和时钟

中断是计算机系统处理异步事件的重要机制。当异步事件发生时，事件通常是通过硬件向CPU发出中断请求的。在一般情况下，CPU响应这个请求后会立即运行中断服务程序来处理该事件。

为了处理任务延时、任务调度等一些与时间有关的事件，任何一个计算机系统都应该有一个系统时钟。与其他计算机系统一样，μC/OS－II的时钟是通过硬件定时器产生定时中断来实现的。

本章的主要内容有：

- μC/OS－II的中断管理和中断服务程序的结构；
- μC/OS－II的系统时钟及实现方法；
- μC/OS－II的时间管理服务：延时、取消延时。

3.1 μC/OS－II的中断

任务在运行过程中，应内部或外部异步事件的请求中止当前任务，而去处理异步事件所要求的任务的过程叫做中断。应中断请求而运行的程序叫做中断服务子程序(Interrupt Service Routines，ISR)，中断服务子程序的入口地址叫做中断向量。

3.1.1 μC/OS－II的中断过程

μC/OS－II系统响应中断的过程是：系统接收到中断请求后，如果这时CPU处于中断允许状态(即中断是开放的)，系统就会中止正在运行的当前任务，而按照中断向量的指向转而去运行中断服务子程序；当中断服务子程序的运行结束后，系统将会根据情况返回到被中止的任务继续运行，或者转向运行另一个具有更高优先级别的就绪任务。

在上面叙述的中断过程中，有一点是需要特别注意的，这就是对于可剥夺型的μC/OS－II内核来说，中断服务子程序运行结束之后，系统将会根据情况进行一次任务调度去运行优先级别最高的就绪任务，而不是一定要接续运行被中断的任务。

μC/OS－II系统允许中断嵌套，即高优先级别的中断源的中断请求可以中断低优先级别

的中断服务程序的运行。为记录中断嵌套的层数，μC/OS－II定义了一个全局变量OSIntNesting。

μC/OS－II响应中断的过程示意图如图3－1所示。

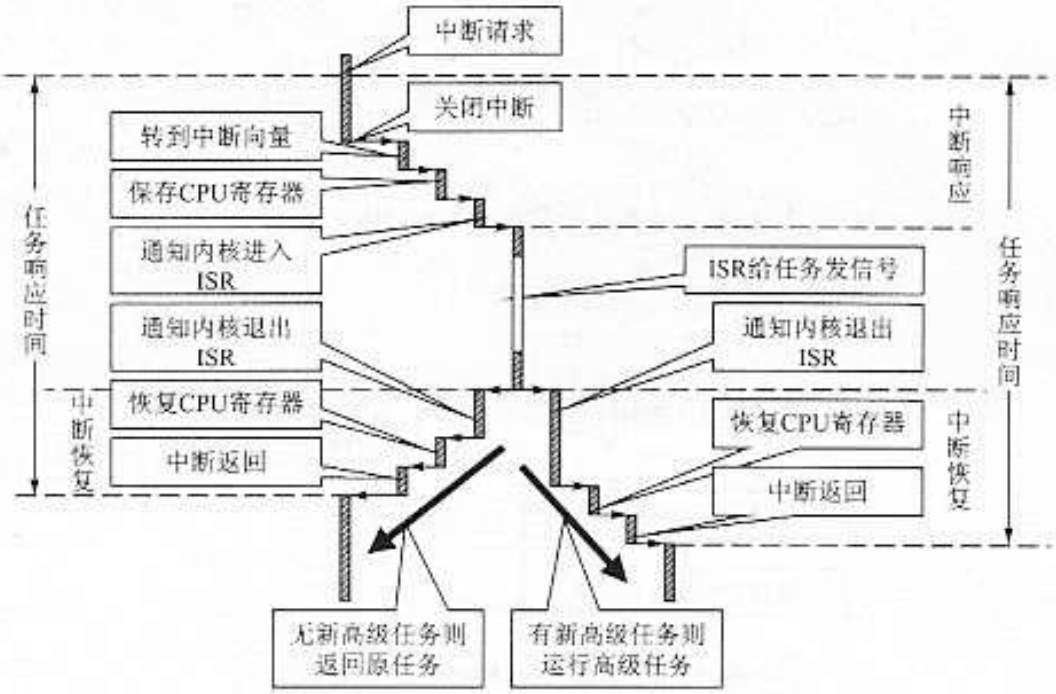


图3－1 中断的响应过程

在编写μC/OS－II的中断服务程序时，要用到两个重要的函数OSIntEnter()和OSIntExit()。

函数OSIntEnter()比较简单，它的作用就是把全局变量OSIntNesting加1，从而用它来记录中断嵌套的层数。函数OSIntEnter()的代码如下：

```
void  OSIntEnter(void)
{
    if (OSRunning == TRUE)
    {
     if (OSIntNesting < 255)
     {
        OSIntNesting++;                    //中断嵌套层数计数器加1
     }
    }
}
```

函数OSIntEnter()经常在中断服务程序保护被中断任务的断点数据之后，运行用户中断

服务代码之前来调用，所以通常把它叫做进入中断服务函数。

另一个在中断服务程序中要调用的函数叫做退出中断服务函数 OSIntExit()。函数 OSIntExit()的流程图如图 3－2 所示。

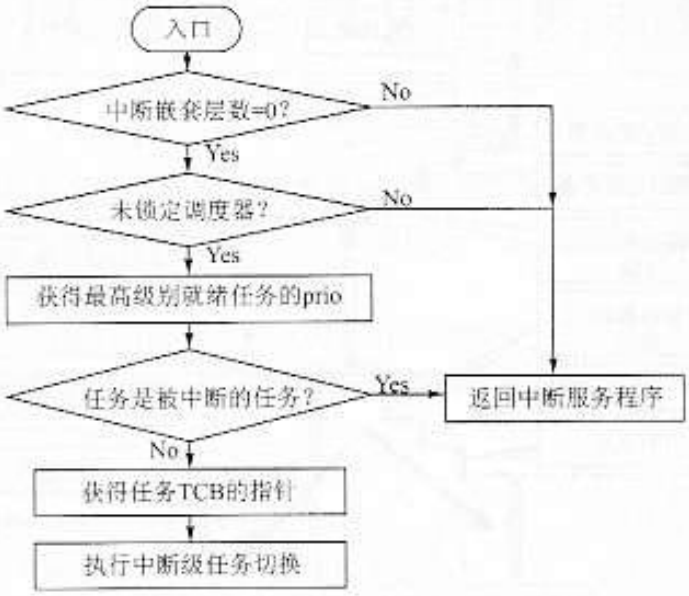


图 3－2　函数 OSIntExit()的流程图

从图 3－2 中可以看到，这个函数在中断嵌套层数计数器为 0、调度器未被锁定且从任务就绪表中查找到的最高级就绪任务又不是被中断的任务的条件下将要进行任务切换，否则就返回被中断的服务子程序。

退出中断服务函数 OSIntExit()的源代码如下：

```
void  OSIntExit (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    if (OSRunning == TRUE)
    {
        OS_ENTER_CRITICAL();
        if (OSIntNesting > 0)
        {
            OSIntNesting--;                    //中断嵌套层数计数器减 1
        }
        if ((OSIntNesting == 0) && (OSLockNesting == 0))
```

```
        {
            OSIntExitY = OSUnMapTbl[OSRdyGrp];
            OSPrioHighRdy = (INT8U)((OSIntExitY <<3)
                        + OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
            if (OSPrioHighRdy != OSPrioCur)
            {
                OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
                OSCtxSwCtr ++ ;
                OSIntCtxSw();
            }
        }
        OS_EXIT_CRITICAL();
    }
}
```

一个中断服务子程序的流程如图 3－3 所示。

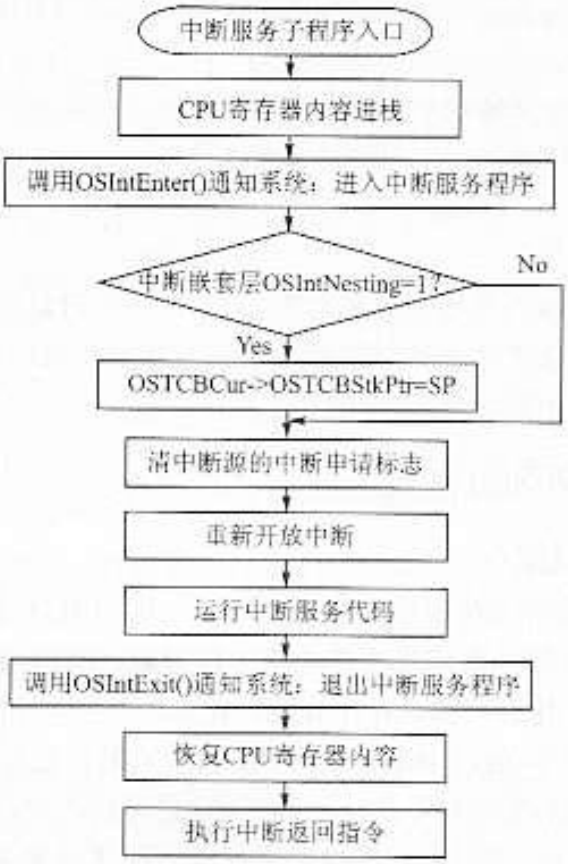


图 3－3　中断服务子程序的流程

在 μC/OS－II 中，通常用一个任务来完成异步事件的处理工作，而在中断服务程序中只是通过向任务发送消息的方法去激活这个任务。

3.1.2 中断级任务切换函数

前面谈到，μC/OS－II 在运行完中断服务程序之后，并不一定返回到被中断的任务上去，而是要进行一次任务调度来决定是返回被中断的任务还是运行一个具有更高优先级别的就绪任务，因此系统需要一个中断级任务调度器。

从图 3－2 的函数 OSIntExit()流程图及其源代码中已经知道，函数在中断嵌套层数计数器为 0、调度器未被锁定且从任务就绪表中查找到的最高级就绪任务又不是被中断的任务的条件下将要进行任务切换，这个在中断服务程序中调用的负责任务切换工作的函数 OSIntCtxSw()叫做中断级任务切换函数。

与任务级切换函数 OSCtxSw()的原因一样，中断级任务切换函数 OSIntCtxSw()通常是用汇编语言来编写的，其示意性代码如下：

```
OSIntCtxSw()
{
    OSTCBCur = OSTCBHighRdy;                    //任务控制块的切换
    OSPrioCur = OSPrioHighRdy;
    SP = OSTCBHighRdy->OSTCBStkPtr;             //使 SP 指向待运行任务堆栈
    用出栈指令把 R1、R2…弹入 CPU 的通用寄存器；
    RETI;                                       //中断返回，使 PC 指向待运行任务
}
```

把上面的代码与任务级任务切换函数 OSCtxSw()的代码对照一下就会发现，中断级任务切换函数的代码与任务级任务切换函数的后半段完全相同。其中的道理也很简单：被中断任务的断点保护工作已经在中断服务程序中完成了。

3.1.3 应用程序中的临界段

1. 临界段的基本概念

前面提到，当有异步事件发生时会引发中断请求，但 CPU 不是在任何情况下都会响应这个请求的。CPU 只有在中断开放期间才能响应中断请求，而在其他时间是不能响应中断请求的。因此，所有 CPU 都有对于中断是否开放进行控制的开中断和关中断指令，以使应用程序可通过这些指令来控制 CPU 何时中断开放，何时中断关闭。之所以如此，是因为在应用程序中经常有一些代码段必须不受任何干扰地连续运行，这样的代码段叫做临界段。因此，为了使临界段在运行时不受中断所打断，在临界段代码前必须用关中断指令使 CPU 屏蔽中断请求，而在临界段代码后必须用开中断指令解除屏蔽，使得 CPU 可以响应中断请求。

由于各厂商生产的 CPU 和 C 编译器的关中断和开中断的方法和指令不尽相同，为增强 μC/OS－II 的可移植性(即在 μC/OS－II 的各个 C 语言函数中尽可能地不出现汇编语言代

码)，μC/OS－II用OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()这两个宏来实现中断的开放和关闭，而把与系统硬件相关的关中断和开中断的指令分别封装在这两个宏中。

另外，μC/OS－II提示，不要在临界段中调用μC/OS－II提供的功能函数，以免系统崩溃。

2. 宏OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()的实现方法

OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()可以有3种不同的实现方法。至于在实际应用时使用哪种方法，取决于用户使用的处理器及C编译器。用户可通过定义移植文件OS_CPU.H中的常数OS_CRITICAL_METHOD来选择实现方法。

第1种方法最简单，即直接使用处理器的开中断和关中断指令来实现宏，这时需要令常数OS_CRITICAL_METHOD=1。其示意性代码如下：

```
#define  OS_ENTER_CRITICAL()     \
         asm("DI")                    //关中断

#define  OS_EXIT_CRITICAL()      \
         asm("EI")                    //开中断
```

第2种方法稍微复杂一些，但可使CPU中断允许标志的状态在临界段前和临界段后不发生改变。在宏OS_ENTER_CRITICAL()中，把CPU的允许中断标志保存到堆栈中，然后再关闭中断，这样在临界段结束时，即在调用宏OS_EXIT_CRITICAL()时，只要把堆栈中保存的CPU允许中断状态恢复即可。这两个宏的示意性代码如下：

```
#define  OS_ENTER_CRITICAL() \
         asm("PUSH    PSW") \        //通过保存程序状态字来保存中断允许标志
         asm("DI")                   //关中断

#define  OS_EXIT_CRITICAL() \
         asm("POP        PSW")       //恢复中断允许标志
```

如果要使用这种方法实现宏，则需要令常数OS_CRITICAL_METHOD=2。

第3种方法的前提条件是，用户使用的C编译器具有扩展功能。用户可获得程序状态字的值，这样就可把该值保存在C语言函数的局部变量中，而不必压到堆栈里。这时，两个宏的实现如下：

```
#define  OS_ENTER_CRITICAL()     \
         cpu_sr = get_processor_psw();\      //获得程序状态字并保存在全局变量sr中
         disable_interrupts();               //关中断

#define  OS_EXIT_CRITICAL()      \
         set_processor_psw(cpu_sr );         //用sr恢复程序状态字
```

由于不知道用户使用的 C 语言编译器所提供的函数名，因此上面宏中使用的函数名称只是示意而已。如果用户要使用这种方法实现宏，则需要令常数 OS_CRITICAL_METHOD=3。

3.2 μC/OS－II 的时钟

任何操作系统都要提供一个周期性的信号源，以供系统处理诸如延时、超时等与时间有关的事件，这个周期性的信号源叫做时钟。

μC/OS－II 与大多数计算机系统一样，用硬件定时器产生一个周期为毫秒(ms)级的周期性中断来实现系统时钟。最小的时钟单位就是两次中断之间间隔的时间，这个最小时钟单位叫做时钟节拍(Time Tick)。

硬件定时器以时钟节拍为周期定时地产生中断，该中断的中断服务程序叫做 OSTickISR()。中断服务程序通过调用函数 OSTimeTick()来完成系统在每个时钟节拍时需要做的工作。

因为使用 C 语言不便于对 CPU 的寄存器进行处理，所以时钟节拍的中断服务程序 OSTickISR()是用汇编语言来编写的。OSTickISR()的示意性代码如下：

```
void OSTickISR(void)
{
    保存 CPU 寄存器；
    调用 OSIntEnter();                    //记录中断嵌套层数
    if (OSIntNesting == 1;
    {
        OSTCBCur->OSTCBStkPtr = SP;       //在任务 TCB 中保存堆栈指针
    }
    调用 OSTimeTick();                    //节拍处理
    清除中断；
    开中断；
    调用 OSIntExit();                     //中断嵌套层数减 1
    恢复 CPU 寄存器；
    中断返回；
}
```

在时钟中断服务程序中调用的 OSTimeTick()叫做时钟节拍服务函数。该函数的源代码如下：

```
void  OSTimeTick (void)
{
#if OS_CRITICAL_METHOD == 3
```

```
    OS_CPU_SR  cpu_sr;
#endif
    OS_TCB    *ptcb;
    OSTimeTickHook();
#if OS_TIME_GET_SET_EN > 0
    OS_ENTER_CRITICAL();
    OSTime++;                                   //记录节拍数
    OS_EXIT_CRITICAL();
#endif
    if (OSRunning == TRUE) {
        ptcb = OSTCBList;
        while (ptcb->OSTCBPrio != OS_IDLE_PRIO)
        {
            OS_ENTER_CRITICAL();
            if (ptcb->OSTCBDly != 0)
            {
                if (--ptcb->OSTCBDly == 0)      //任务的延时时间减1
                {
                    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND)
                                == OS_STAT_RDY)
                    {
                        OSRdyGrp |= ptcb->OSTCBBitY;
                        OSRdyTbl[ptcb->OSTCBY]
                                        |= ptcb->OSTCBBitX;
                    }
                    else
                    {
                        ptcb->OSTCBDly = 1;
                    }
                }
            }
            ptcb = ptcb->OSTCBNext;
            OS_EXIT_CRITICAL();
        }
    }
```

从加粗的代码段中可知，μC/OS-II 在每次响应定时中断时调用 OSTimeTick()做了两件事情：一是给计数器 OSTime 加 1；二是遍历任务控制块链表中的所有任务控制块，把各个

任务控制块中用来存放任务延时时限的 OSTCBDly 变量减 1，并使该项为 0，同时又不使被挂起的任务进入就绪状态。

简单地说，函数 OSTimeTick()的任务就是在每个时钟节拍了解每个任务的延时状态，使其中已经到了延时时限的非挂起任务进入就绪状态。

例 3－1 在例 2－6 应用程序的基础上，在 OS_CPU.C 文件中按如下代码定义函数 OSTimeTickHook()，然后运行并查看运行结果。

```
INT16U d = 0;
INT16U d1 = 0;
void  OSTimeTickHook (void)
{
    char * s0 = "500";
    char * s1 = "每";
    char * s2 = "次中断的调度次数:";
    char s[8];
    if(d == 500)
    {
        PC_DispStr(14,4,s1,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
        PC_DispStr(18,4,s0,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
        PC_DispStr(24,4,s2,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
        sprintf(s," %d",OSCtxSwCtr);
        PC_DispStr(20,d1 + 5,s,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
        d = 0;
        d1 += 1;
    }
    d += 1;
}
```

应用程序改写了时钟节拍服务函数 OSTimeTick()中调用的钩子函数 OSTimeTickHook()在显示器上显示系统在每发生 500 次时钟中断期间，调度器进行任务调度的次数。例 3－1 应用程序的运行结果见图 3－4。

OSTimeTick()是系统调用的函数，为了方便应用程序设计人员能在系统调用的函数中插入一些自己的工作，μC/OS－II 提供了时钟节拍服务函数的钩子函数 OSTimeTickHook()。

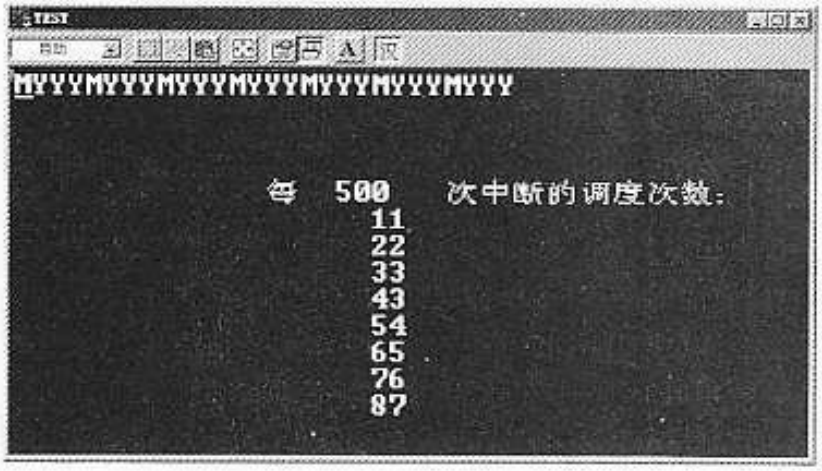


图 3－4 例 3－1 应用程序的运行结果

此外，μC/OS－II 还提供了 OSStkInitHook()、OSInitHookBegin()、OSInitHookEnd()、OSTaskCreateHook()、OSTaskDelHook()、OSTaskSwHook()、OSTaskStatHook()、OSTCBInitHook()、OSTaskIdleHook()等与 OSTimeTickHook()一起共 10 个钩子函数，以供用户在系统调用函数中书写自己的代码。

例 3－2 设计一个有 3 个任务的应用程序 Test。这 3 个任务分别是：MyTask、YouTask 和 InterTask。其中任务 InterTask 是在时钟节拍中断服务程序中用钩子函数 OSTimeTickHook()中断了 10 000 次时使用一个信号变量 InterKey 激活的。运行并分析由中断服务程序激活任务的工作特点。

答 应用程序的代码如下：

```
/*********************************Test *********************************/
#include "includes.h"
#define  TASK_STK_SIZE    512                    //任务堆栈长度
OS_STK   MyTaskStk[TASK_STK_SIZE];               //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];              //定义任务堆栈区
OS_STK   InterTaskStk[TASK_STK_SIZE];            //定义任务堆栈区
INT16S   key;                                    //用于退出 μC/OS－II 的键
INT8U    x = 0,y = 0;                            //字符显示位置
BOOLEAN  InterKey = FALSE ;                      //中断与任务联系的变量
char * s = "运行了中断所要求运行的任务 InterTask。";
void  MyTask(void *data);                        //声明任务
void  YouTask(void *data);                       //声明任务
void  InterTask(void *data);                     //声明任务
```

```
/*************************主函数*************************************/
void  main (void)
{
    char * s_M = "M";                          //定义要显示的字符
    OSInit();                                  //初始化 μC/OS－II
    PC_DOSSaveReturn();                        //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                 //安装 μC/OS－II 中断
    OSTaskCreate(
        MyTask,                                //创建任务 MyTask
        s_M,                                   //给任务传递参数
        &MyTaskStk[TASK_STK_SIZE - 1],         //设置任务堆栈栈顶指针
        0                                      //任务的优先级别为 0
        );
    OSStart();                                 //启动多任务管理
}

/*******************任务 MyTask*************************************/
void  MyTask (void * pdata)
{
    char * s_Y = "Y";                          //定义要显示的字符
    char * s_H = "H";
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);               //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);          //设置时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                              //初始化统计任务
    OSTaskCreate(
            YouTask,                           //创建任务 YouTask
            s_Y,                               //给任务传递参数
            &YouTaskStk[TASK_STK_SIZE - 1],    //设置任务堆栈栈顶指针
            1                                  // YouTask 的优先级别为 1
            );
    OSTaskCreate(
```

```
            InterTask,                          //创建任务 InterTask
            s_B,                                //给任务传递参数
            &InterTaskStk[TASK_STK_SIZE - 1],   //设置任务堆栈栈顶指针
            2                                   // InterTask 的优先级别为 2
            );
    for (;;)
    {
        if (x>=50)
        {
            x = 0;
            y += 2;
        }
        PC_DispChar(x, y,                       //字符的显示位置
        *(char *)pdata,
        DISP_BGND_BLACK + DISP_FGND_WHITE );
            x += 1;
        //如果按下 ESC 键,则退出 μC/OS－II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();                 //恢复 DOS 环境
            }
        }
        OSTimeDlyHMSM(0, 0, 3, 0);              //等待 3 s
    }
}

/*************************任务 YouTask ******************************/

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

```
    {
        if (x>50)
        {
            x = 0;
            y += 2;
        }
        PC_DispChar(
                x, y,                          //字符的显示位置
                *(char *)pdata,
                DISP_BGND_BLACK + DISP_FGND_WHITE
                );
            x += 1;
        OSTimeDlyHMSM(0, 0, 1, 0);             //等待 1 s
    }
}
/*********************** InterTask ********************************/
void  InterTask (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        if(InterKey)
        {
            if (x>50)
            {
                x = 0;
                y += 2;
            }
            PC_DispChar(
                x, y,
                *(char *)pdata,
                DISP_BGND_BLACK + DISP_FGND_WHITE
                );
        PC_DispStr(5,6,s,
```

```
                DISP_BGND_BLACK + DISP_FGND_WHITE
                );
                    x += 1;
            }
            InterKey = FALSE;
            OSIntNesting -- ;
        OSTimeDlyHMSM(0, 0, 1, 0);                    //等待 1 s
    }
}
/*********************OSTimeTickHook *************************/
extern BOOLEAN InterKey;
INT16U InterCtr = 0;
void  OSTimeTickHook (void)
{
    if(InterCtr == 10000)
        {
            InterKey = TRUE;
        }
    InterCtr ++ ;
}
/*************************End ************************************/
```

例 3－2 应用程序的运行结果如图 3－5 所示。

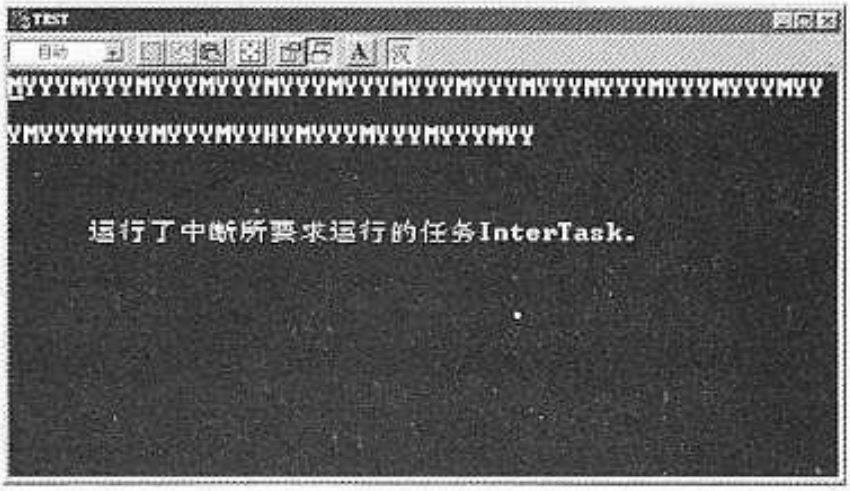


图 3－5　例 3－2 应用程序的运行结果

3.3 时间管理

3.3.1 任务的延时

由于嵌入式系统的任务是一个无限循环，并且 μC/OS－II 还是一个抢占式内核，所以为了使高优先级别的任务不至于独占 CPU，可以给其他优先级别较低的任务获得 CPU 使用权的机会，μC/OS－II 规定：除了空闲任务之外的所有任务必须在任务中合适的位置调用系统提供的函数 OSTimeDly()，使当前任务的运行延时（暂停）一段时间并进行一次任务调度，以让出 CPU 的使用权。

OSTimeDly()函数的代码如下：

```
void  OSTimeDly (INT16U ticks)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
     if (ticks > 0)
     {
          OS_ENTER_CRITICAL();
        if ((OSRdyTbl[OSTCBCur->OSTCBY]
                  &= ~OSTCBCur->OSTCBBitX) == 0)
        {
          OSRdyGrp
              &= ~OSTCBCur->OSTCBBitY;          //取消当前任务的就绪状态
        }
          OSTCBCur->OSTCBDly = ticks;           //延时节拍数存入任务控制块
          OS_EXIT_CRITICAL();
        OS_Sched();                             //调用调度函数
     }
}
```

函数的参数 ticks 是以时钟节拍数为单位的延时时间的。

为了能使用更为习惯的方法来使任务延时，μC/OS－II 还提供了一个可以用时、分、秒为参数的任务的延时函数 OSTimeDlyHMSM()。该函数的原型如下：

```
INT8U OSTimeDlyHMSM(
                        INT8U hours,      //时
                        INT8U minutes,    //分
                        INT8U seconds,    //秒
                        INY16U milli      //毫秒
                        );
```

这个函数与函数 OSTimeDly()一样也要引发一次调度。

调用了函数 OSTimeDly()或 OSTimeDlyHMSM()的任务，当规定的延时时间期满，或有其他任务通过调用函数 OSTimeDlyResume()取消了延时时，它会立即进入就绪状态。

3.3.2 取消任务的延时

延时的任务可通过在其他任务中调用函数 OSTimeDlyResume()取消延时而进入就绪状态。如果任务比正在运行的任务优先级别高，则立即引发一次任务调度。

OSTimeDlyResume()函数的原型如下：

```
INT8U OSTimeDlyResume( INT8U prio);
```

参数 prio 为被取消延时任务的优先级别。

OSTimeDlyResume()函数的源代码如下：

```
INT8U  OSTimeDlyResume (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_TCB      *ptcb;

    if (prio >= OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    ptcb = (OS_TCB *)OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {
        if (ptcb->OSTCBDly != 0) {
            ptcb->OSTCBDly = 0;
            if ((ptcb->OSTCBStat & OS_STAT_SUSPEND)
                                        == OS_STAT_RDY)
            {
```

```
                OSRdyGrp                    |= ptcb->OSTCBBitY;
                OSRdyTbl[ptcb->OSTCBY]      |= ptcb->OSTCBBitX;
                OS_EXIT_CRITICAL();
                OS_Sched();
            } 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);
}
```

例 3－3 本例应用程序的任务使用了延时函数 OSTimeDly()进行延时。在任务 MyTask 中还调用了函数 OSTimeDlyResume(prio)取消了任务 YouTask 的延时。为了观察任务 YouTask 的延时时间的变化，在钩子函数 OSTimeTickHook()输出了任务 YouTask 在延时时间到时的时钟节拍数。

答 应用程序代码如下：

```
/*************************Test ************************************/
#include "includes.h"
#define  TASK_STK_SIZE   512                    //任务堆栈长度
OS_STK   MyTaskStk[TASK_STK_SIZE];              //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];             //定义任务堆栈区
INT16S   key;                                   //用于退出 μC/OS－II 的键
INT8U    x=0,y=0;                               //字符显示位置
void  MyTask(void *data);                       //声明任务
void  YouTask(void *data);                      //声明任务
/*************************主函数 ************************************/
void  main (void)
{
    char* s_M="M";                              //定义要显示的字符
    OSInit();                                   //初始化 μC/OS－II
    PC_DOSSaveReturn();                         //保存 DOS 环境
```

```
    PC_VectSet(uCOS, OSCtxSw);                  //安装μC/OS－II中断
    OSTaskCreate(MyTask,                        //创建任务MyTask
            s_M,                                //给任务传递参数
            &MyTaskStk[TASK_STK_SIZE - 1],      //设置任务堆栈栈顶指针
            0);                                 //使任务的优先级别为0
    OSStart();                                  //启动μC/OS－II的多任务管理
}

/*************************任务MyTask*********************************/
void  MyTask (void * pdata)
{
    char * s_Y="Y";                             //定义要显示的字符
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                //安装μC/OS－II时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);           //设置μC/OS－II时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                               //初始化μC/OS－II的统计任务
    OSTaskCreate(YouTask,                       //创建任务YouTask
        s_Y,                                    //给任务传递参数
        &YouTaskStk[TASK_STK_SIZE - 1],         //设置任务堆栈栈顶指针
        2);                                     //使任务的优先级别为2
    for (;;)
    {
        if (x>50)
        {
            x=0;
            y+=2;
        }
        if(y>1) OSTimeDlyResume(2);             //取消YouTask任务的延时
        PC_DispChar(x, y,                       //显示字符的位置
                *(char *)pdata,
                DISP_BGND_BLACK+DISP_FGND_WHITE );
            x += 1;
```

```
    //如果按下 ESC 键,则退出 μC/OS－II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDly(100);                          //延时 100 个时钟节拍
    }
}
/*************************任务 YouTask ******************************/
void  YouTask (void * pdata)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif
    pdata = pdata;
    for (;;)
    {
        if (x>50)
        {
            x = 0;
            y += 2;
        }
        PC_DispChar(x, y,                        //显示字符的位置
                    * (char * )pdata,
                    DISP_BGND_BLACK + DISP_FGND_WHITE );
            x += 1;
        OSTimeDly(500);                          //延时 500 个时钟节拍
    }
}
/*************************OSTimeTickHook() *************************/
INT8U d = 0;
INT8U l = 0;
INT16U tt = 1;                                   //时钟计数器
char s[5];
```

```
void  OSTimeTickHook (void)
{
    if(OSTCBPrioTbl[2]->OSTCBDly == 1)
        {
            sprintf(s," %5d",tt);
            PC_DispStr(d,l + 4,s,DISP_BGND_BLACK + DISP_FGND_WHITE );
            d += 6;
        }
    tt += 1;
}
/*****************************End ********************************/
```

例 3－3 应用程序的运行结果如图 3－6 所示。

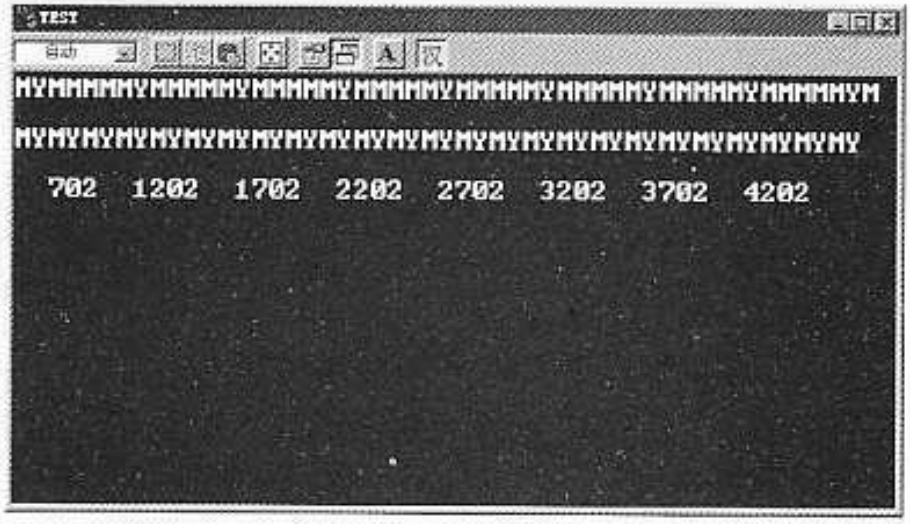


图 3－6　例 3－3 应用程序的运行结果

3.3.3　获取和设置系统时间

为了方便，系统定义了一个 INT32U 类型的全局变量 OSTime 来记录系统发生的时钟节拍数。OSTime 在应用程序调用 OSStart()时被初始化为 0，以后每发生 1 个时钟节拍，OSTime 的值就被加 1。

在应用程序调用函数 OSTimeGet()可获取 OSTime 的值。函数 OSTimeGet()的原型如下：

```
INT32U  OSTimeGet( void );
```

函数的返回值即为 OSTime 的值。

如果在应用程序调用函数 OSTimeSet()，则可设置 OSTime 的值。函数 OSTimeSet()的原型如下：

```
void  OSTimeSet( INT32U ticks );
```

函数的参数 ticks 为 OSTime 的设置值(节拍数)。

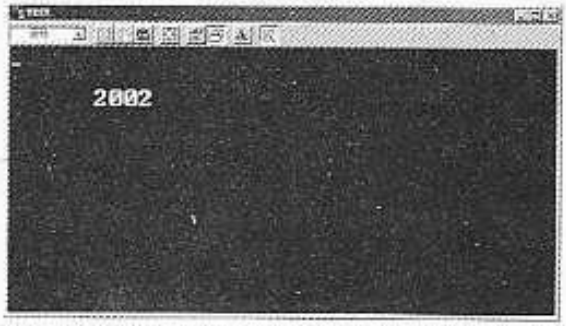
例 3－4 设计一个应用程序，在任务中调用函数 OSTimeGet()获得并显示系统的时钟节拍数 OSTime。当任务运行 10 s 时，调用函数 OSTimeSet()将 OSTime 设置为 10。

答 应用程序的代码如下：

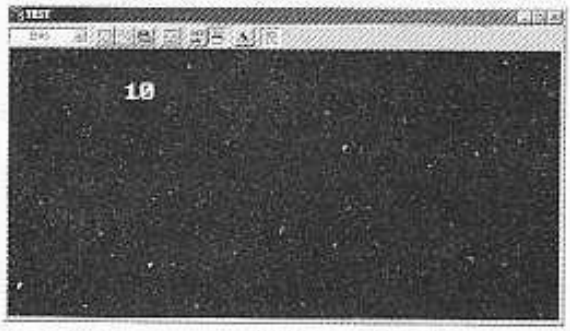
```
/*************************Test ****************************************/
#include "includes.h"
#define  TASK_STK_SIZE    512                          //任务堆栈长度
OS_STK   TaskStartStk[TASK_STK_SIZE];                  //定义任务堆栈区
INT16S   key;                                          //用于退出 μC/OS－II 的键
INT32U   stime;                                        //存放节拍的变量
INT8U    x = 0;
void  MyTask(void * data);                             //声明一个任务
/*************************主函数 ****************************************/
void  main (void)
{
    OSInit();                                          //初始化 μC/OS－II
    PC_DOSSaveReturn();                                //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                         //安装 μC/OS－II 中断
    OSTaskCreate(
                MyTask,                                //创建任务 MyTask
                (void * )0,                            //给任务传递参数
                &TaskStartStk[TASK_STK_SIZE - 1],      //设置堆栈栈顶指针
                0                                      //使任务的优先级别为 0
                );
    OSStart();                                         //启动 μC/OS－II 的多任务管理
}
/*************************MyTask ****************************************/
void  MyTask (void * pdata)
{
char s[5];
#if OS_CRITICAL_METHOD == 3
```

```
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                          //安装 μC/OS－II 时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);                     //设置 μC/OS－II 时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                                         //初始化 μC/OS－II 的统计任务
    for (;;)
    {
        if (x == 10)
        {
            OSTimeSet(10);
        }
        stime = OSTimeGet();
        sprintf(s," %5d",stime);
        PC_DispStr( 5, 2,                                 //在 x,y 位置显示 s 中的字符
                    s,
                    DISP_BGND_BLACK + DISP_FGND_WHITE );
        x += 1;
        //如果按下 ESC 键,则退出 μC/OS－II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDlyHMSM(0, 0, 1, 0);                        //等待
    }
}
/*************************End ************************************/
```

例 3－4 应用程序的运行结果见图 3－7。图 3－7(a)为调用函数 OSTimeSet()前的情况；图 3－7(b)为调用函数 OSTimeSet()后的情况。



(a) 调用函数OSTimeGet( )获得系统时钟节拍数OSTime的值



(b) 调用函数OSTimeSet( )设置系统时钟节拍数OSTime的值

图 3－7　例 3－4 应用程序的运行结果

3.4　小　结

- 在 μC/OS－II 中，中断服务子程序运行结束之后，系统将会根据情况进行一次中断级的任务调度去运行优先级别最高的就绪任务，而并不一定要接续运行被中断的任务。
- μC/OS－II 的中断允许嵌套，用全局变量 OSIntNesting 来记录中断的嵌套数。
- μC/OS－II 的中断服务程序的工作通常是由中断激活的一个任务来完成的。这样做的优点是可以使应用程序的设计更为灵活。
- 在任务中可以用设置临界区的方法来屏蔽中断。设置临界区的宏有 3 种实现方法。
- μC/OS－II 的时钟通常是一个由硬件计数器定时产生周期性中断信号来实现的，每一次中断叫做一个节拍，其中断服务程序叫做节拍服务程序。
- μC/OS－II 在每一个节拍服务里要遍历系统中的全部任务的任务控制块，把其中记录任务延时时间的成员 OSTCBDly 减 1，并使延时时间到的任务进入就绪状态。
- μC/OS－II 有 10 个函数提供了钩子函数，应用程序设计人员可在钩子函数中编写一些自己的代码。
- 在 μC/OS－II 进行时间管理的函数中，最重要的是延时函数 OSTimeDly( ) 和

OSTimeDlyHMSM()。它们的作用不仅仅是使任务的运行停止并等待一段时间，更重要的是，它们都要调用任务调度器进行一次任务调度，这样就使低优先级别的任务有了运行的机会。

3.5 练习题

1. 简述 μC/OS－II 的中断响应过程。
2. 全局变量 OSIntNesting 的作用是什么？
3. μC/OS－II 的中断服务程序何时返回被中断的任务？何时不返回被中断的任务？
4. μC/OS－II 的系统时钟是如何实现的？在时钟节拍服务中做了什么工作？
5. 如何在中断服务程序中激活一个任务？
6. 说明延时函数 OSTimeDly()与 OSTimeDlyHMSM()的区别。

第4章 任务的同步与通信

应用程序中的各个任务，必须通过彼此之间的有效合作，才能完成一项大规模的工作。因为这些任务在运行时，经常需要互相无冲突地访问同一个共享资源，或者需要互相支持和依赖，甚至有时还要互相加以必要的制约，才能保证任务的顺利运行。因此，操作系统必须具有对任务的运行进行协调的能力，从而使任务之间可以无冲突、流畅地同步运行，而不致导致灾难性的后果。

与人们依靠通信来互相沟通，从而使人际关系和谐，工作顺利的做法一样，计算机系统是依靠任务之间的良好通信来保证任务与任务的同步的。

本章的主要内容有：

- 事件及描述事件的数据结构——事件控制块；
- 信号量及其使用；
- 消息邮箱及其使用；
- 消息队列及其使用。

4.1 任务间的同步和事件控制块

嵌入式系统中的各个任务都是以并发的方式来运行的，并为同一个大的任务服务。它们不可避免地要共同使用一些共享资源，并且在处理一些需要多个任务共同协作来完成的工作时，还需要相互的支持和限制。因此，对于一个完善的多任务操作系统来说，系统必须具有完备的同步和通信机制。

4.1.1 任务间的同步

为了实现各任务之间的合作和无冲突的运行，在各任务之间必须建立一些制约关系。其中一种制约关系叫做直接制约关系，另一种制约关系则叫做间接制约关系。

直接制约关系源于任务之间的合作。例如，有两个任务：任务 A 和任务 B，它们需要通过访问同一个数据缓冲区合作完成一项工作，任务 A 负责向缓冲区写入数据，任务 B 负责从缓冲区读取该数据。显然，当任务 A 还未向缓冲区写入数据时（缓冲区为空时），任务 B 因不能

从缓冲区得到有效数据而应该处于等待状态；只有等任务 A 向缓冲区写入了数据之后，才应该通知任务 B 去取数据。相反，当缓冲区的数据还未被任务 B 读取时（缓冲区为满时），任务 A 就不能向缓冲区写入新的数据而应该处于等待状态；只有等任务 B 自缓冲区读取数据后，才应该通知任务 A 去写入数据。显然，如果这两个任务不能如此协调工作，将势必造成严重的后果。

间接制约关系源于对资源的共享。例如，任务 A 和任务 B 共享一台打印机，如果系统已经把打印机分配给了任务 A，则任务 B 因不能获得打印机的使用权而应该处于等待状态；只有当任务 A 把打印机释放后，系统才能唤醒任务 B 使其获得打印机的使用权。如果这两个任务不这样做，那么也会造成极大的混乱。

由上可知，在多任务合作工作的过程中，操作系统应该解决两个问题：一是各任务间应该具有一种互斥关系，即对于某个共享资源，如果一个任务正在使用，则其他任务只能等待，等到该任务释放该资源后，等待的任务之一才能使用它；二是相关的任务在执行上要有先后次序，一个任务要等其伙伴发来通知，或建立了某个条件后才能继续执行，否则只能等待。

任务之间这种制约性的合作运行机制叫做任务间的同步。系统中任务的同步是依靠任务与任务之间互相发送消息来保证同步的。

4.1.2 事 件

任务间的同步依赖于任务间的通信。在 μC/OS－II 中，使用信号量、邮箱（消息邮箱）和消息队列这些被称作事件的中间环节来实现任务之间的通信。

图 4－1 是两个任务通过事件进行通信的示意图。任务 1 是发信方，任务 2 是收信方。作为发信方，任务 1 的责任是把信息发送到事件上，这项操作叫做发送事件。作为收信方，任务 2 的责任是通过读事件操作对事件进行查询：如果有信息，则读取信息；如果没有，则等待。读事件操作叫做请求事件。

图 4－1 两个任务使用事件进行通信的示意图

μC/OS－II 把任务发送事件、请求事件以及其他对事件的操作都定义成为全局函数，以供应用程序的所有任务来调用。

1. 信号量

信号量是一类事件。使用信号量的最初目的，是为了共享资源设立一个表示该共享资源被占用情况的标志。这样，就可使任务在访问共享资源之前，先对这个标志进行查询，在了解资源被占用的情况之后，再来决定自己的行为。

观察一下人们日常生活中常用的一种共享资源——公用电话亭的使用规则，就会发现它的规则很适合在协调关系时应用。如果一个电话亭只允许一个人进去打电话，那么电话亭的

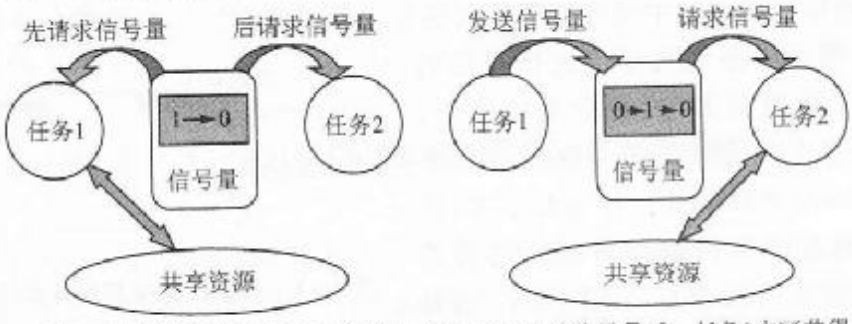
门上就应该有一个可以变换两种颜色的牌子(例如,用红色表示"有人",用绿色表示"无人")。当有人进去时,牌子会变成红色;出来时,牌子又会变成绿色。这样来打电话的人就可根据牌子的颜色来了解电话亭的被占用情况。例如,如果某一个人去电话亭打电话时见到牌子上的颜色是绿色,那么他就可以进去打电话;如果是红色,那么他只好等待;如果又陆续来了很多人,那么就要排队等待。

显然,电话亭门上的这个牌子就是一个表示电话亭是否已被占用的信号。

显然,这是一个二值信号,由于这种二值信号可以实现共享资源的独占式占用,所以叫做互斥型信号量。

如果电话亭可以允许多人打电话,那么电话亭门前就不应该是只有红色和绿色两种状态的牌子,而应该是一个计数器,并且这个计数器在每进去一个人时会自动减 1,而每出去一个人时会自动加 1。如果计数器的初值按电话亭的最大容量来设置,那么来人只要见到计数器的值大于 0,就可以进去打电话;否则只好等待。这种计数式的信号叫做信号量。

图 4－2 是两个任务使用互斥型信号量进行通信,从而可使这两个任务无冲突地访问一个共享资源的示意图。任务 1 在访问共享资源之前先进行请求信号量的操作,当任务 1 发现信号量的标志为"1"时,它一方面把信号量的标志由"1"改为"0",另一方面进行共享资源的访问。如果任务 2 在任务 1 已经获得信号之后来请求信号量,那么因为它获得的标志值是"0",所以任务 2 就只有等待而不能访问共享资源了(见图 4－2(a))。显然,这种做法可以有效地防止两个任务同时访问同一个共享资源所造成的冲突。



(a) 任务1先获得信号量并使用共享资源,而任务2只能等待信号量　(b) 任务1释放信号量后,任务2方可获得信号量并使用共享资源

图 4－2　两个任务使用信号量进行通信的示意图

那么任务 2 何时可以访问共享资源呢?当然是在任务 1 使用完共享资源之后,由任务 1 向信号量发信号使信号量标志的值由"0"再变为"1"时,任务 2 就有机会访问共享资源了。同任务 1 一样,任务 2 一旦获得了共享资源的访问权,那么在访问共享资源之前一定要把信号量标志的值由"1"变为"0"(见图 4－2(b))。

例 4－1　本例的应用程序中有两个用户任务:MyTask 和 YouTask。这两个任务都要访问同一个共享资源 s,但 YouTask 访问 s 需要的时间要长一些(本例中使用了一个循环来模拟

访问时间)，而 MyTask 访问 s 的时间要短一些，这样就不可避免地出现了在任务 YouTask 访问 s 期间，任务 MyTask 也来访问 s，从而出现了干扰。

答 应用程序代码如下：

```
/*********************************Test*************************************/
#include "includes.h"

#define  TASK_STK_SIZE    512                    //任务堆栈长度
char * ss;
OS_STK   MyTaskStk[TASK_STK_SIZE];               //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];              //定义任务堆栈区
INT16S   key;                                    //用于退出 μC/OS-II 的键
INT8U    y1 = 0,y2 = 0;                          //字符显示位置
char *  s = "原始数据";                          //定义要显示的字符
void  MyTask(void *data);                        //声明任务
void  YouTask(void *data);                       //声明任务
/*************************主函数*************************************/
void  main (void)
{
    OSInit();                                    //初始化 μC/OS-II
    PC_DOSSaveReturn();                          //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                   //安装 μC/OS-II 中断
    OSTaskCreate(MyTask,                         //创建任务 MyTask
        (void *)0,                               //给任务传递参数
        &MyTaskStk[TASK_STK_SIZE - 1],           //设置任务堆栈栈顶指针
        0);                                      //使任务的优先级别为 0
    OSStart();                                   //启动多任务管理
}

/************************任务 MyTask*********************************/

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

```
INT8U err;
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                    //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);               //设置 μC/OS－II 时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                                   //初始化统计任务
    OSTaskCreate(YouTask,                           //创建任务 YouTask
        (void *)0,                                  //给任务传递参数
        &YouTaskStk[TASK_STK_SIZE - 1],             //设置任务堆栈栈顶指针
        2);                                         //使任务的优先级别为 2
    for (;;)
    {
        s = "MyTask 访问共享数据 s";
        PC_DispStr(5, ++y1,                         //显示字符串
            s,
            DISP_BGND_BLACK + DISP_FGND_WHITE);
    //如果按下 ESC 键,则退出 μC/OS－II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDly(200);                             //等待 200 个时钟节拍
    }
}
/*************************** 任务 YouTask ********************************/
void  YouTask (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U err;
    pdata = pdata;
    for (;;)
```

```
    {
        s = "YouTask 访问共享数据 s";
        PC_DispStr(28, ++y2,                    //显示字符串
            s,
            DISP_BGND_BLACK + DISP_FGND_WHITE );
        OSTimeSet(0);                           //置 OSTime 为 0
        while(OSTime<500)
        {
            sprintf(ss," %d",OSTimeGet());
            PC_DispStr(55, y2,                  //显示字符串
                    s,
                    DISP_BGND_BLACK + DISP_FGND_WHITE );
        }
        OSTimeDly(10);                          //等待 10 个时钟节拍
    }
}
/*************************End ***************************************/
```

从例 4-1 应用程序的运行结果(见图 4-3)中可以看到,在 YouTask 访问共享资源 s 期间高优先级别的任务 MyTask 也访问了 s,从而干扰了任务 YouTask 对共享资源 s 的访问(图中的表现就是在任务 YouTask 的延时期间前,共享资源 s 的内容发生了变化)。

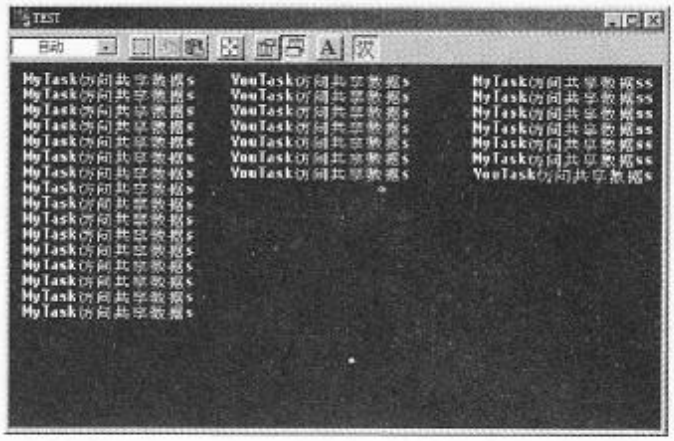


图 4-3 例 4-1 应用程序的运行结果

解决这个问题的一个办法就是设置一个信号量,从而使任务可根据信号量的状态来判断何时访问共享资源。

例 4-2 在例 4-1 的应用程序中定义一个全局变量 ac_key 来作为信号量,并根据该信号量的状态来访问共享资源 s,以解决冲突问题。

答　应用程序代码如下：

```
/*************************Test*************************************/
#include "includes.h"

#define  TASK_STK_SIZE    512                    //任务堆栈长度
OS_STK   MyTaskStk[TASK_STK_SIZE];               //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];              //定义任务堆栈区
INT16S   key;                                    //用于退出 μC/OS－II 的键
INT8U    y1 = 0,y2 = 0;                          //字符显示位置
BOOLEAN  ac_key;                                 //定义信号量
char * s = "原始数据";                            //定义要显示的字符
void  MyTask(void *data);                        //声明任务
void  YouTask(void *data);                       //声明任务
/*************************主函数*********************************/
void  main (void)
{
    OSInit();                                    //初始化 μC/OS－II
    ac_key = 1 ;                                 //设置信号量初值
    PC_DOSSaveReturn();                          //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                   //安装 μC/OS－II 中断
    OSTaskCreate(MyTask,                         //创建任务 MyTask
        (void *)0,                               //给任务传递参数
        &MyTaskStk[TASK_STK_SIZE - 1],           //设置任务堆栈栈顶指针
        0);                                      //使任务的优先级别为 0

    OSStart();                                   //启动 μC/OS－II 的多任务管理
}

/*************************任务 MyTask*********************************/

void  MyTask (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U err;
    pdata = pdata;
```

```
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                    //安装 μC/OS-II 时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);               //设置 μC/OS-II 时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                                   //初始化 μC/OS-II 的统计任务
    OSTaskCreate(YouTask,                           //创建任务 YouTask
        (void*)0,                                   //给任务传递参数
        &YouTaskStk[TASK_STK_SIZE - 1],             //设置任务堆栈栈顶指针
        2);                                         //使任务的优先级别为 2
    for (;;)
    {
        if(ac_key)
        {
            ac_key = FALSE ;                        //使信号量无效
            s = "MyTask 访问共享数据 s";
            PC_DispStr(5, ++y1,                     //显示字符的位置
                s,
                DISP_BGND_BLACK + DISP_FGND_WHITE);
                ac_key = TRUE;                      //发信号
        }
    //如果按下 ESC 键,则退出 μC/OS-II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDly(20);                              //等待 20 个时钟节拍
    }
}
/*************************任务 YouTask ****************************/
void  YouTask (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
```

```
    INT8U err;
    pdata = pdata;
    for (;;)
    {
        if(ac_key)
        {
            ac_key = FALSE ;                        //使信号量为无信号
            s = "YouTask 访问共享数据 s";
            PC_DispStr(28, ++y2,                    //显示字符串
                s,
                DISP_BGND_BLACK + DISP_FGND_WHITE );
            OSTimeSet(0);                           //置 OSTime 为 0
            while(OSTime<500)
            {
                PC_DispStr(55, y2,                  //显示字符串
                    s,
                    DISP_BGND_BLACK + DISP_FGND_WHITE );
            }
            ac_key = TRUE ;                         //发信号
        }
        OSTimeDly(10);                              //等待 10 个时钟节拍
    }
}
/*************************End ***************************************/
```

从例 4－2 应用程序的运行结果(见图 4－4)中可以看到，任务 MyTask 就不会在任务 YouTask 访问共享资源 s 期间来访问 s 了。这样就避免了多个任务使用同一个共享资源时出现的冲突现象。

如果把任务 YouTask 代码中的发信号语句 ac_key＝TRUE 删掉，那么再运行这个程序就会发现，由于任务 YouTask 不发信号，所以会使得高优先级别的任务 MyTask 虽然获得了 CPU 的使用权，但由于始终得不到信号量而不能运行，当然也就没有机会让出 CPU 的使用权，从而也就导致任务 YouTask 不能运行，于是应用程序就死掉了。

解决上述问题的一个合理的办法就是给等待信号量的任务设置一个等待时限。当等待信号量的任务因等待某信号量的时间超过这个时限时，可以使等待任务脱离等待状态而继续运行，这样就不会出现上述的死机现象了(这部分内容见本小节的“等待任务列表”部分)。

2. 消息邮箱

在多任务操作系统中，常常需要在任务与任务之间通过传递一个数据(这种数据叫做“消

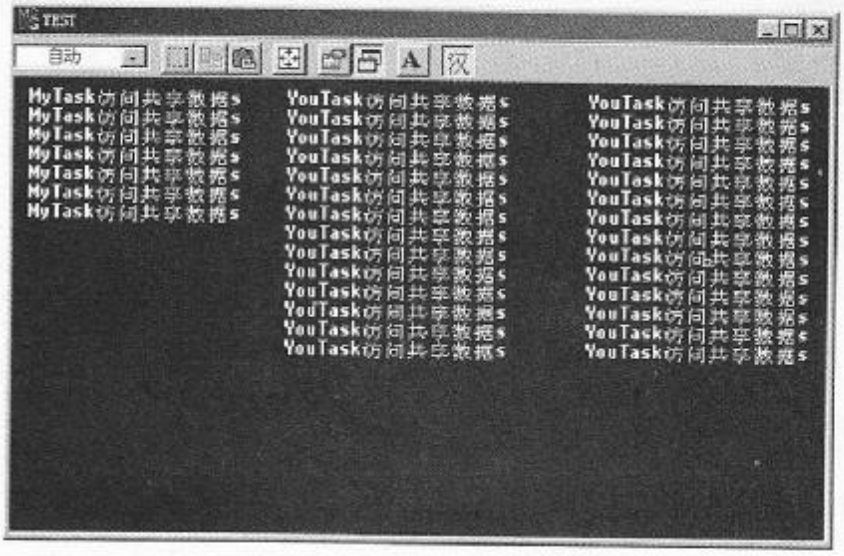


图 4-4　例 4-2 应用程序的运行结果

息")的方式来进行通信。为了达到这个目的，可以在内存中创建一个存储空间作为该数据的缓冲区。如果把这个缓冲区叫做消息缓冲区，那么在任务间传递数据(消息)的一个最简单的方法就是传递消息缓冲区的指针。因此，用来传递消息缓冲区指针的数据结构就叫做消息邮箱。

图 4-5 是两个任务使用消息邮箱进行通信的示意图。任务 1 在向消息邮箱发送消息，任务 2 在从消息邮箱读取消息。读取消息也叫做请求消息。

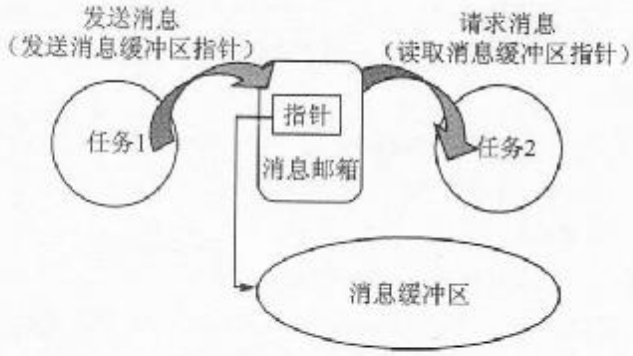


图 4-5　两个任务使用消息邮箱进行通信的示意图

例 4-3　下面是一个利用简单消息邮箱进行通信的例子。本例中有两个任务：MyTask 和 YouTask。由于任务 YouTask 要向任务 MyTask 发送消息，因此定义了一个全局的指针变量 msg_p 作为邮箱来传递消息的指针。

答　应用程序代码如下：

```
/*************************Test*************************************/
#include "includes.h"
#define  TASK_STK_SIZE    512                    //任务堆栈长度
OS_STK   StartTaskStk[TASK_STK_SIZE];            //定义任务堆栈区
OS_STK   MyTaskStk[TASK_STK_SIZE];               //定义任务堆栈区
```

```
OS_STK   YouTaskStk[TASK_STK_SIZE];                    //定义任务堆栈区
INT16S   key;                                          //用于退出的键
INT8U    y1 = 0,y2 = 0;                                //字符显示位置
void * msg_p ;                                         //消息邮箱
void  StartTask(void * data);                          //声明起始任务
void  MyTask(void * data);                             //声明任务
void  YouTask(void * data);                            //声明任务
/*************************主函数 **********************************/
void  main (void)
{
    OSInit();                                          //初始化 μC/OS－II
    PC_DOSSaveReturn();                                //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                         //安装 μC/OS－II 中断
    OSTaskCreate(StartTask,                            //创建任务 StartTask
                 (void * )0,                           //给任务传递参数
                 &StartTaskStk[TASK_STK_SIZE - 1],     //设置任务堆栈栈顶
                 0);                                   //任务的优先级别为 0
    OSStart();                                         //启动多任务管理
}
/*************************任务 StartTask ********************************/
void  StartTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                       //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);                  //设置时钟频率
    OS_EXIT_CRITICAL();

    OSStatInit();                                      //初始化统计任务
    OSTaskCreate(MyTask,                               //创建任务 MyTask
                 (void * )0,                           //给任务传递参数
                 &MyTaskStk[TASK_STK_SIZE - 1],        //设置任务堆栈栈顶
                 1);                                   //任务的优先级别为 1
    OSTaskCreate(YouTask,                              //创建任务 YouTask
```

```
                (void * )0,                                  //给任务传递参数
                &YouTaskStk[TASK_STK_SIZE - 1],              //设置任务堆栈栈顶
                2);                                          //任务的优先级别为 2
    for (;;)
    {
    //如果按下 ESC 键,则退出 μC/OS - II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDlyHMSM(0, 0, 3, 0);                           //等待 3 s
    }
}
/*************************任务 MyTask *********************************/
void  MyTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        PC_DispStr(5, ++yl,                                  //显示字符串
                "MyTask",
                DISP_BGND_BLACK + DISP_FGND_WHITE );
        if(msg_p!= (void * ) 0)                              //请求消息邮箱
        {
            PC_DispStr(15,yl,
                    msg_p,                                   //显示收到的消息
                    DISP_BGND_BLACK + DISP_FGND_WHITE );
            msg_p = (void * ) 0;                             //使消息邮箱为空
        }
        OSTimeDlyHMSM(0, 0, 1, 0);                           //等待 1 s
    }
```

```
}
/*************************任务 YouTask ******************************/
void  YouTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    char * s = "YouTask 发送的消息";                    //定义消息
    pdata = pdata;
    for (;;)
    {
        PC_DispStr(40, ++y2,                            //显示字符串
                "YouTask",
                DISP_BGND_BLACK + DISP_FGND_WHITE );

        if(OSTimeGet()>500 && msg_p == (void * ) 0)
        {
            msg_p = s ;                                 //发送消息
            PC_DispStr(50,y2,
                "YouTask 发了一个消息",
                DISP_BGND_BLACK + DISP_FGND_WHITE );
        }
        OSTimeDlyHMSM(0, 0, 2, 0);                      //等待 2 s
    }
}
/*************************End ****************************************/
```

例 4－3 应用程序的运行结果如图 4－6 所示。从图中可以看到，使用消息邮箱机制可以确保消息准确地传递。

3. 消息队列

上面谈到的消息邮箱不仅可用来传递一个消息，而且也可定义一个指针数组。让数组的每个元素都存放一个消息缓冲区指针，那么任务就可通过传递这个指针数组指针的方法来传递多个消息了。这种可以传递多个消息的数据结构叫做消息队列。

图 4－7 是两个任务使用消息队列进行通信的示意图。任务 1 向消息队列发送消息缓冲区指针数组的指针，这个操作叫做发送消息队列；任务 2 从消息队列读取消息缓冲区指针数组的指针，这个操作叫做请求消息队列。

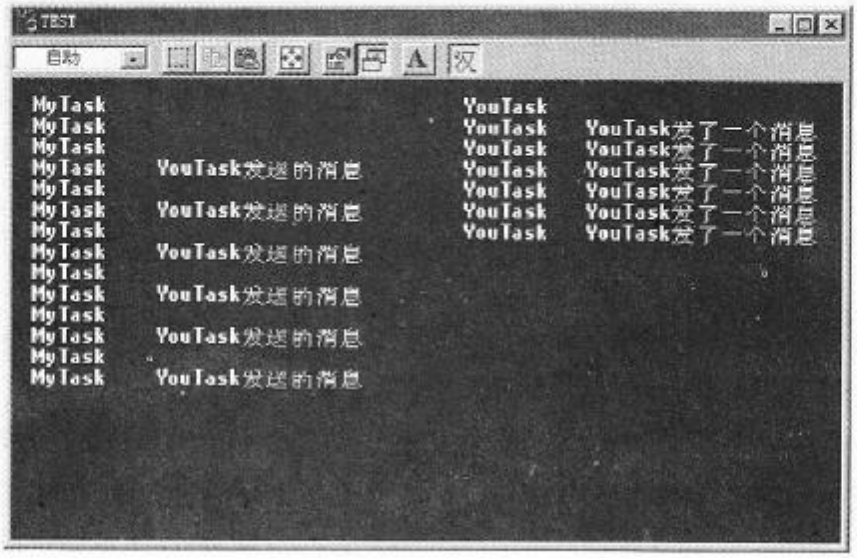


图 4-6　例 4-3 应用程序的运行结果

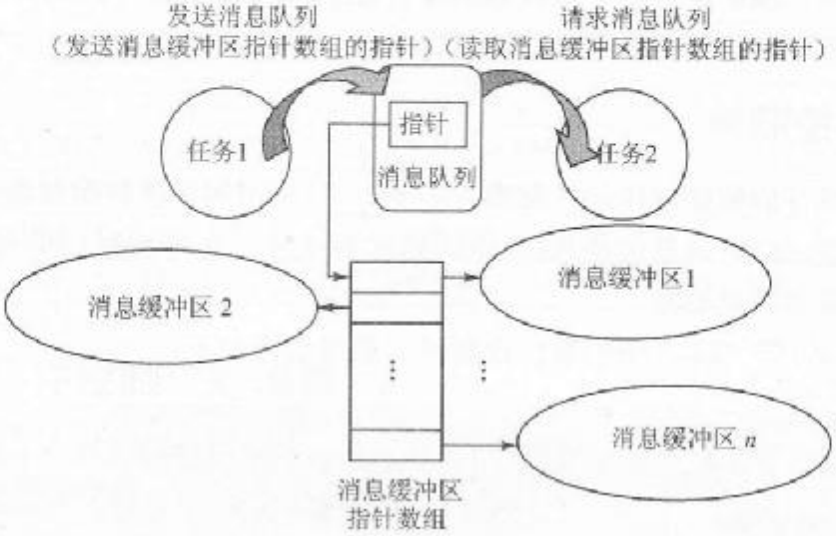


图 4-7　两个任务使用消息队列进行通信的示意图

4. 等待任务列表

前面已经讲过，μC/OS-II 把信号量、消息邮箱和消息队列这类用于任务同步和通信的数据结构叫做事件。为了使读者对事件有一个基本概念，在前面列举了一些事件的简单例子，正是由于它们的简单，所以功能极不完善。在多任务系统中，当一个事件被占用时，其他请求该事件的任务在暂时得不到事件的服务时应该处于等待状态。因此，作为功能完善的事件，应该有对这些等待任务具有一定的管理功能。这个管理功能应该包括两个方面：一是要对等待事件的所有任务进行记录并排序；二是应该允许任务有一定的等待时限。

对于等待事件任务的记录，μC/OS－II 采用了与任务就绪表类似的方法，使用一个 INT8U 类型的数组 OSEventTbl[ ]作为记录等待事件任务的记录表，这个表叫做等待任务表。在这个等待任务表中仍然是以任务的优先级别为顺序，令系统中的每个任务都在表中占据一位，并用该位为 1 来表示这一位对应的任务为事件的等待任务，否则不是等待任务。同样，为了加快对该表的访问速度，也定义了一个 INT8U 类型的变量 OSEventGrp 来表示等待任务表中的任务组。等待任务表 OSEventTbl[ ]与变量 OSEventGrp 的示意图如图 4－8 所示。

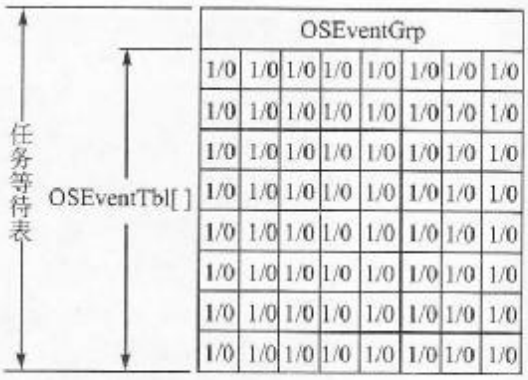


图 4－8　事件的等待任务表

至于等待任务的等待时限，则记录在等待任务的任务控制块 TCB 的成员 OSTCBDly 中，并在每个时钟节拍中断服务程序中对该数据进行维护。每当有任务的等待时限已到时，将该任务从等待任务表中删除，并使它进入就绪状态。

4.1.3　事件控制块

为了把描述事件的数据结构统一起来，μC/OS－II 使用叫做事件控制块 ECB 的数据结构来描述诸如信号量、邮箱（消息邮箱）和消息队列这些事件。事件控制块中包含包括等待任务表在内的所有有关事件的数据。

定义在文件 μC/OS－II.H 中的事件控制块的数据结构如下：

```
typedef struct
{
    INT8U   OSEventType;                        //事件的类型
    INT16U OSEventCnt;                          //信号量计数器
    void  * OSEventPtr;                         //消息或消息队列的指针
    INT8U   OSEventGrp;                         //等待事件的任务组
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE];        //任务等待表
} OS_EVENT;
```

事件控制块结构的示意图如图 4－9 所示。

应用程序中的任务通过指针 pevent 来访问事件控制块。

成员 OSEventCnt 为信号量的计数器。

成员 OSEventPtr 主要用来存放消息邮箱或消息队列的指针。

成员 OSEventTbl[OS_EVENT_TBL_SIZE]是一个数组。这个数组的格式与前面任务就绪表的格式一样。应用程序中的所有任务按照优先级别各自在表中占据一个二进制位，并用该位的值是"1"还是"0"来表示该位对应的任务是否为正在等待事件的任务，这个表叫做任务等待表。

与任务就绪表类似，结构成员 OSEventGrp 表示任务等待表中的各任务组是否存在等待任务。

事件控制块 ECB 结构中的成员 OSEventType 用来指明事件的类型。成员 OSEventType 可能的取值见表 4-1。

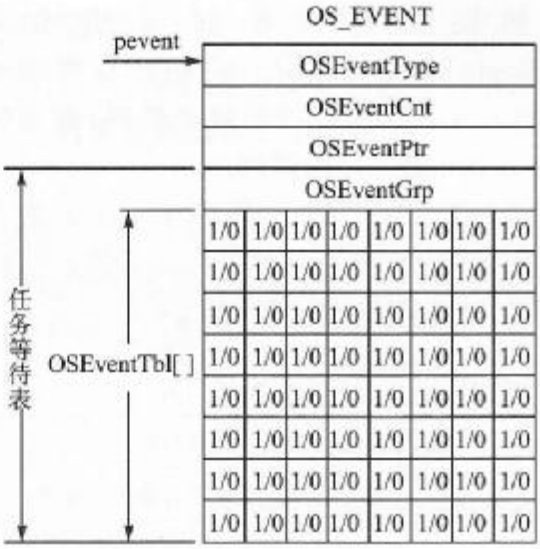


图 4-9 事件控制块 ECB 结构

表 4-1 OSEventType(定义在文件 uCOS_II.H 中)可取的值

OSEventType 的值	说 明
OS_EVENT_TYPE_SEM	表明事件是信号量
OS_EVENT_TYPE_MUTEX	表明事件是互斥型信号量
OS_EVENT_TYPE_MBOX	表明事件是消息邮箱
OS_EVENT_TYPE_Q	表明事件是消息队列
OS_EVENT_TYPE_UNUSED	空事件控制块(未被使用的事件控制块)

4.1.4 操作事件控制块的函数

μC/OS-II 有 4 个对事件控制块进行基本操作的函数(定义在文件 OS_CORE.C 中)，以供操作信号量、消息邮箱、消息队列等事件的函数来调用。

1. 事件控制块的初始化函数

调用 EventWaitListInit()函数可以对事件控制块进行初始化。函数 EventWaitListInit()的原型如下：

```
void  OS_EventWaitListInit (
                            OS_EVENT *pevent      //事件控制块的指针
                          );
```

这个函数的作用就是把变量 OSEventGrp 及任务等待表中的每一位都清 0，即令事件的任务等待表中不含有任何等待任务。

事件控制块被初始化后的情况如图 4-10 所示。

初始化事件控制块的函数 OS_EventWaitListInit()，将在任务调用函数 OS×××Create()创建事件时，被函数 OS×××Create()所调用。

函数 OS×××Create()中×××的含义见表 4－2。

表 4－2　×××的含义

×××	含　义
Sem	对信号量进行操作的函数
Mutex	对互斥型信号量进行操作的函数
Mbox	对消息邮箱进行操作的函数
Q	对消息队列进行操作的函数

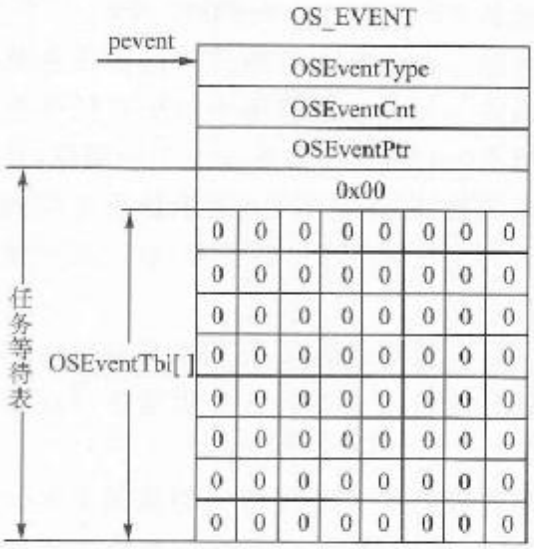


图 4－10　初始化后的事件控制块

2. 使一个任务进入等待状态的函数

把一个任务置于等待状态要调用函数 OS_EventTaskWait()。该函数的原型如下：

```
void  OS_EventTaskWait (
                        OS_EVENT *pevent      //事件控制块的指针
                        );
```

函数 OS_EventTaskWait()将在任务调用函数 OS×××Pend()请求一个事件时，被函数 OS×××Pend()所调用。

3. 使一个正在等待任务进入就绪状态的函数

如果一个正在等待的任务具备了可以运行的条件，那么就要使它进入就绪状态。这时要调用函数 OS_EventTaskRdy()。该函数的作用就是把调用这个函数的任务在任务等待表中的位置清 0(解除等待状态)后，再把任务在任务就绪表中对应的位置 1，然后引发一次任务调度。

函数 OS_EventTaskRdy()的原型如下：

```
INT8U  OS_EventTaskRdy (
                        OS_EVENT *pevent,     //事件控制块的指针
                        void *msg,            //未使用
                        INT8U msk             //清除 TCB 状态标志掩码
                       );
```

函数 OS_EventTaskRdy()将在任务调用函数 OS×××Post()发送一个事件时，被函数 OS×××Post()所调用。

4. 使一个等待超时的任务进入就绪状态的函数

如果一个正在等待事件的任务已经超过了等待的时间，却仍因为没有获取事件等原因而未具备可以运行的条件，却又要使它进入就绪状态，这时要调用函数 OS_EventTO()。

函数 OS_EventTO()的原型如下：

```
void  OS_EventTO (
                  OS_EVENT * pevent          //事件控制块的指针
                  );
```

函数 OS_EventTO()将在任务调用函数 OS×××Pend()请求一个事件时，被函数 OS×××Pend()所调用。

4.1.5 空事件控制块链表

在 μC/OS-II 初始化时，系统会在初始化函数 OSInit()中按应用程序使用事件的总数 OS_MAX_EVENTS(在文件 OS_CFG.H 中定义)，创建 OS_MAX_EVENTS 个空事件控制块并借用成员 OSEventPtr 作为链接指针，把这些空事件控制块链接成一个如图 4-11 所示的单向链表。由于链表中的所有控制块尚未与具体事件相关联，因此该链表叫做空事件控制块链表。以后，每当应用程序创建一个事件时，系统就会从链表中取出一个空事件控制块，并对它进行初始化以描述该事件。而当应用程序删除一个事件时，就会将该事件的控制块归还给空事件控制块链表。

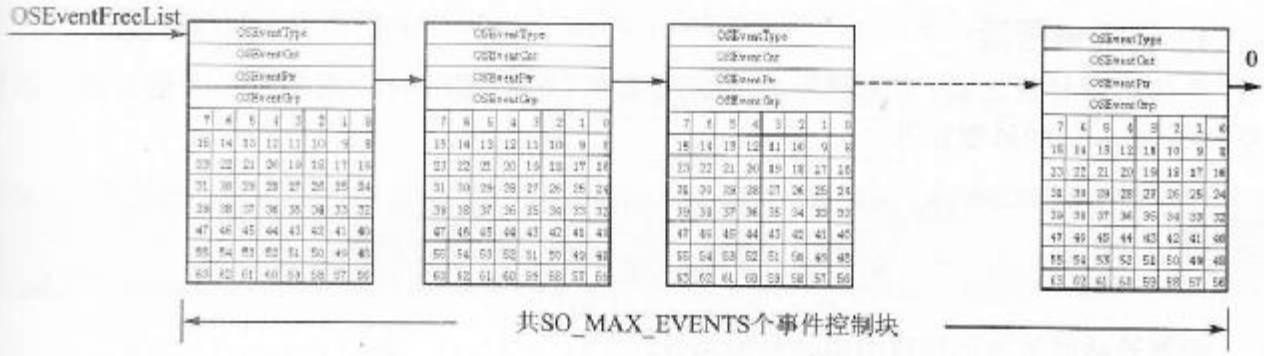


图 4-11　空事件控制块链表

4.2 信号量及其操作

4.2.1 信号量

当事件控制块成员 OSEventType 的值被设置为 OS_EVENT_TYPE_SEM 时，这个事件

控制块描述的就是一个信号量。信号量由信号量计数器和任务等待表两部分组成。

信号量使用事件控制块的成员 OSEventCnt 作为计数器，而用 OSEvevtTbl[ ]数组来充当等待任务表。

每当有任务申请信号量时，如果信号量计数器 OSEventCnt 的值大于 0，则把 OSEventCnt 减 1 并使任务继续运行；如果 OSEventCnt 的值为 0，则会将任务列入任务等待表 OSEvevtTbl[ ]，而使任务处于等待状态。如果有正在使用信号量的任务释放了该信号量，则会在任务等待表中找出优先级别最高的等待任务，并在使它就绪后调用调度器引发一次调度；如果任务等待表中已经没有等待任务，则信号量计数器就只简单地加 1。

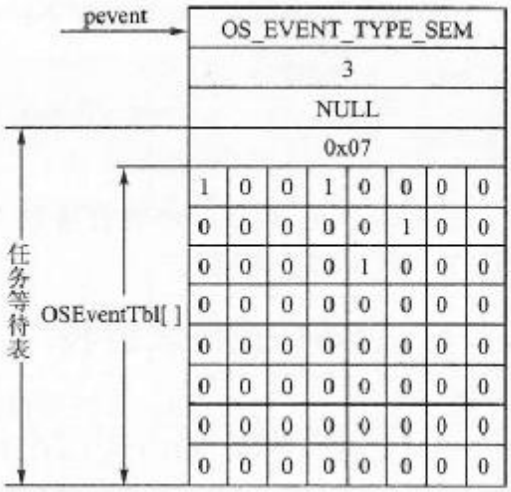


图 4－12　一个信号量的事件控制块

图 4－12 是一个计数器当前值为 3 且有 4 个等待任务的信号量的示意图。等待信号量任务的优先级别分别为 4、7、10、19。

信号量不使用事件控制块的成员 OSEventPtr。

4.2.2　信号量的操作

1. 创建信号量

在使用信号量之前，应用程序必须调用函数 OSSemCreate()来创建一个信号量。函数 OSSemCreate()的原型如下：

```
OS_EVENT    * OSSemCreate (
                          INT16U cnt          //信号量计数器初值
                        );
```

函数的返回值为已创建的信号量的指针。

OSSemCreate()的源代码如下：

```
OS_EVENT    * OSSemCreate (INT16U cnt)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif

    OS_EVENT    * pevent;
```

```
    if (OSIntNesting > 0)
    {
        return ((OS_EVENT *)0);
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;
    if (OSEventFreeList != (OS_EVENT *)0)
    {
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0)
    {
        pevent->OSEventType = OS_EVENT_TYPE_SEM;  //设置为信号量
        pevent->OSEventCnt  = cnt;                //置计数器初值
        pevent->OSEventPtr  = (void *)0;          //置空指针
        OS_EventWaitListInit(pevent);             //初始化控制块
    }
    return (pevent);
}
```

一个刚创建且计数器初值 OSEventCnt 为 10 的信号量的示意图如图 4-13 所示。

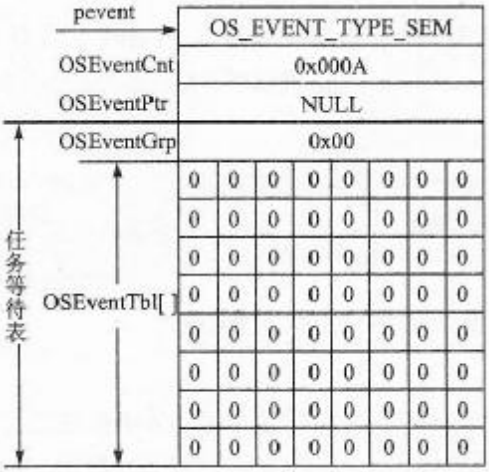


图 4-13　一个刚创建且计数器初值为 10 的信号量

2. 请求信号量

任务通过调用函数 OSSemPend()请求信号量。函数 OSSemPend()的原型如下：

```
void  OSSemPend ( OS_EVENT * pevent,      //信号量的指针
                  INT16U timeout,         //等待时限
                  INT8U * err);           //错误信息
```

参数 pevent 是被请求信号量的指针。

为防止任务因得不到信号量而处于长期的等待状态，函数 OSSemPend()允许用参数 timeout 设置一个等待时间的限制。当任务等待的时间超过 timeout 时，可以结束等待状态而进入就绪状态。如果参数 timeout 被设置为 0，则表明任务的等待时间为无限长。

函数调用成功后，err 的值为 OS_NO_ERR。如果函数调用失败，则函数会根据在函数中出现的具体错误，令 err 的值分别为 OS_ERR_PEND_ISR、OS_ERR_PEVENT_NULL、OS_ERR_EVENT_TYPE 和 OS_TIMEOUT。

当任务需要访问一个共享资源时，先要请求管理该资源的信号量，这样就可以根据信号量当前是否有效(即信号量的计数器 OSEventCnt 的值是否大于 0)来决定该任务是否可以继续运行。

如果信号量有效(即信号量的计数器 OSEventCnt 的值大于 0)，则把信号量计数器减 1，然后继续运行任务。

如果信号量无效(即信号量的计数器 OSEventCnt 的值等于 0)，则会在等待任务表中把该任务对应的位置 1 而让任务处于等待状态，并把等待时限 timeout 保存在任务控制块 TCB 的成员 OSTCBDly 中。

当一个任务请求信号量时，如果希望在信号量无效时准许任务不进入等待状态而继续运行，则不调用函数 OSSemPend()，而是调用函数 OSSemAccept()来请求信号量。该函数的原型如下：

```
INT16U  OSSemAccept (
                      OS_EVENT * pevent     //信号量的指针
                    );
```

调用函数成功后，函数返回值为 OS_ON_ERR。

3. 发送信号量

任务获得信号量，并在访问共享资源结束以后，必须释放信号量。释放信号量也叫做发送信号量。发送信号量须调用函数 OSSemPost()。函数 OSSemPost()在对信号量的计数器操作之前，首先要检查是否还有等待该信号量的任务：如果没有，就把信号量计数器 OSEventCnt 加 1；如果有，则调用调度器 OS_Sched()去运行等待任务中优先级别最高的任务。

函数 OSSemPost()的原型如下：

```
INT8U  OSSemPost (
                    OS_EVENT * pevent      //信号量的指针
                  );
```

若调用函数成功，则函数返回值为 OS_ON_ERR；否则会根据具体错误返回 OS_ERR_EVENT_TYPE、OS_SEM_OVF。

例 4-4 试编写一个应用程序，其中有一个函数 Fun()和两个任务(任务 MyTask()与任务 YouTask())。应用程序中的两个任务都可以调用函数 Fun()，但不能同时调用。

答 应用程序的代码如下：

```
/*************************Test ************************************/
#include "includes.h"
#define  TASK_STK_SIZE    512                          //任务堆栈长度
char * s1 = "MyTask";
char * s2 = "YouTask";
INT8U err;                                             //用于退出的键
INT8U y = 0;                                           //字符显示位置
INT16S   key;
OS_EVENT * Fun_Semp ;                                  //声明信号量
OS_STK   StartTaskStk[TASK_STK_SIZE];                  //定义任务堆栈区
OS_STK   MyTaskStk[TASK_STK_SIZE];                     //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];                    //定义任务堆栈区
void  Fun(INT8U x,INT8U y);                            //函数声明
void  StartTask(void * data);                          //声明起始任务
void  MyTask(void * data);                             //声明任务
void  YouTask(void * data);                            //声明任务
/*************************主函数 ***********************************/
void  main (void)
{
    OSInit();                                          //初始化 μC/OS-II
    Fun_Semp = OSSemCreate (1);                        //定义信号量
    PC_DOSSaveReturn();                                //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                         //安装 μC/OS-II 中断
    OSTaskCreate(StartTask,                            //创建任务 StartTask
                 (void * )0,                           //给任务传递参数
                 &StartTaskStk[TASK_STK_SIZE - 1],     //设置任务堆栈栈顶
                 0);                                   //使任务的优先级别为 0
```

```
    OSStart();                                          //启动多任务管理
}
/*************************任务 StartTask *********************************/
void  StartTask (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                        //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);                   //设置 μC/OS－II 时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                                       //初始化统计任务
    OSTaskCreate(MyTask,                                //创建任务 MyTask
                 (void*)0,                              //给任务传递参数
                 &MyTaskStk[TASK_STK_SIZE - 1],         //设置任务堆栈栈顶
                 1);                                    //使任务的优先级别为 1
    OSTaskCreate(YouTask,                               //创建任务 YouTask
                 (void*)0,                              //给任务传递参数
                 &YouTaskStk[TASK_STK_SIZE - 1],        //设置任务堆栈栈顶
                 2);                                    //使任务的优先级别为 2
    for (;;)
    {
    //如果按下 ESC 键,则退出 μC/OS－II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDlyHMSM(0, 0, 3, 0);                      //等待 3 s
    }
}
/*************************任务 MyTask *********************************/
void  MyTask (void *pdata)
```

```
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;

    for (;;)
    {
        OSSemPend(Fun_Semp,0,&err);                    //请求信号量
        PC_DispStr(0, ++y,
                   s1,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
        Fun(7,y);                                      //调用函数 Fun()
        OSSemPost(Fun_Semp);                           //发送信号量
        OSTimeDlyHMSM(0, 0, 1, 0);                     //等待 1 s
    }
}
/************************** 任务 YouTask ********************************/
void  YouTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        OSSemPend(Fun_Semp,0,&err);                    //请求信号量
        PC_DispStr(0, ++y,
                   s2,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
        Fun(7,y);                                      //调用函数 Fun()
        OSSemPost(Fun_Semp);                           //发送信号量
        OSTimeDlyHMSM(0, 0, 2, 0);                     //等待 2 s
    }
}
/************************** Fun() ************************************/
void  Fun(INT8U x,INT8U y)
```

```
{
    PC_DispStr(x,y,
        "调用了 Fun()函数",
        DISP_BGND_BLACK + DISP_FGND_WHITE );
}
/*************************End ****************************************/
```

例 4－4 应用程序的运行结果见图 4－14。

从例 4－4 的代码中可以看到，使用事件控制块描述的信号量，无论在使用方面，还是在信号量的完善性方面，它的确比简单形式的信号量要方便得多。

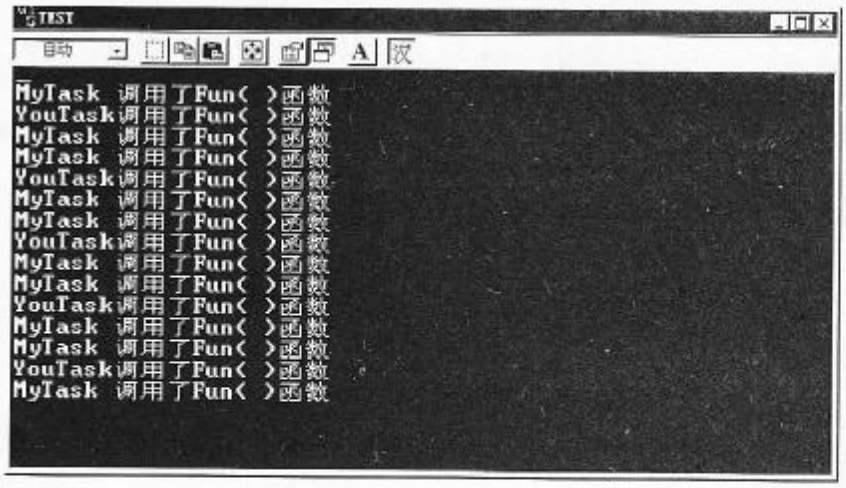


图 4－14　例 4－4 应用程序的运行结果

例 4－5　应用程序中有一个函数 Fun()，如果想使任务 MyTask() 必须经过任务 YouTask()允许才能调用这个函数一次，试写出这两个任务的示意性代码。

答　示意性代码如下：

```
OS_ECB *Fun_Semp;                          //定义一个信号量

//主函数 main()
void main(void)
{
    ……
    Fun_Semp = OSSemCreate (0);            //创建一个初值为 0 的信号量
    ……
```

```
}

//任务 MyTask
void MyTask(void * pdata)
{
    UINT8U err;
    ……
    OSSemPend(Fun_Semp,0,&err);          //请求信号量
    Fun();                               //调用函数 Fun()
    ……
}

//任务 YouTask
void YouTask(void * pdata)
{
    UINT8U err;
    ……
    OSSemPost(Fun_Semp);                 //发送信号量
    ……
}
```

4. 删除信号量

如果应用程序不需要某个信号量，那么可以调用函数 OSSemDel()来删除该信号量。这个函数的原型如下：

```
OS_EVENT   * OSSemDel (
                        OS_EVENT * pevent,    //信号量的指针
                        INT8U opt,            //删除条件选项
                        INT8U * err           //错误信息
                       );
```

函数中的参数 opt 用来指明信号量的删除条件。该参数有两个参数值可以选择：如果选择常数 OS_DEL_NO_PEND，则当等待任务表中已没有等待任务时才删除信号量；如果选择常数 OS_DEL_ALLWAYS，则表明在等待任务表中无论是否有等待任务都立即删除信号量。

函数调用成功后，err 的值为 OS_NO_ERR。

需要注意的是，只能在任务中删除信号量，而不能在中断服务程序中删除。

5. 查询信号量的状态

任务可以调用函数 OSSemQuery()随时查询信号量的当前状态。这个函数的原型如下：

```
INT8U  OSSemQuery (
                    OS_EVENT * pevent,     //信号量指针
                    OS_SEM_DATA * pdata    //存储信号量状态的结构
                   );
```

该函数的第二个参数 pdata 是一个 OS_SEM_DATA 结构的指针。OS_SEM_DATA 结构如下：

```
typedef struct {
    INT16U  OSCnt;
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];
    INT8U   OSEventGrp;
} OS_SEM_DATA;
```

任务调用函数 OSSemQuery()对信号量查询后，会把信号量中的相关信息存储到 OS_SEM_DAT 类型的变量中，因此在调用函数 OSSemQuery()之前，须定义一个 OS_SEM_DAT 结构类型的变量。

函数调用成功后，返回值为 OS_NO_ERR。

4.3 互斥型信号量和任务优先级反转

前面说过，互斥型信号量是一个二值信号，它可以使任务以独占方式使用共享资源。

由于使用互斥型信号量会出现任务优先级反转的问题，所以本节首先介绍产生任务优先级反转的原因，然后再介绍互斥型信号量以及它是如何解决任务优先级反转问题的。

4.3.1 任务优先级的反转现象

在可剥夺型内核中，当任务以独占方式使用共享资源时，会出现低优先级任务先于高优先级任务而被运行的现象，这种现象叫做任务优先级反转。在一般情况下是不允许出现这种任务优先级反转现象的。下面就对优先级的反转现象进行详细分析，以期找出原因及解决方法。

图 4-15 描述了 A、B、C 三个任务的运行情况。其中，任务 A 的优先级别高于任务 B，任务 B 的优先级别高于任务 C。任务 A 和任务 C 都要使用同一个共享资源 S，而用于保护该资源的信号量在同一时间只能允许一个任务以独占的方式对该资源进行访问，即这个信号量是一个互斥型信号量。

现在，假如任务 A 和任务 B 都在等待与各自任务相关的事件发生而处于等待状态，而任务 C 正在运行，且在 t_1 时刻取得了信号量并开始访问共享资源 S。

如果在任务 C 使用共享资源 S 过程中的 t_2 时刻，任务 A 等待的事件已经到来，那么由于任务 A 的优先级别高于任务 C 的优先级别，所以任务 A 就剥夺任务 C 的 CPU 使用权而进入

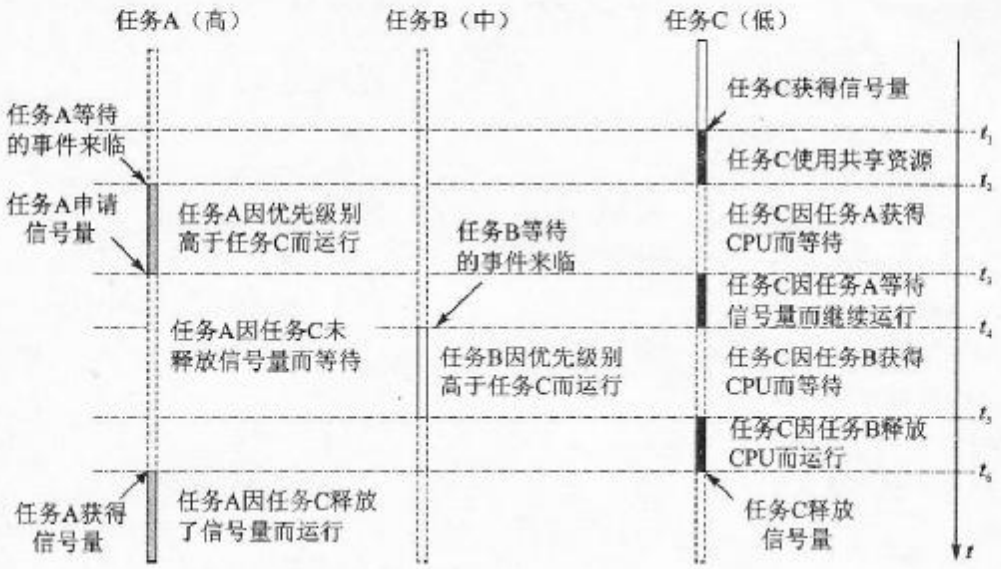


图 4-15　任务优先级反转示意图

运行状态，而使任务 C 中止运行，这样任务 C 就失去了释放信号量的机会。如果任务 A 在运行中的 t_3 时刻又要访问共享资源 S，但由于任务 C 还未释放信号量，因此任务 A 只好等待，以使任务 C 可以继续使用共享资源 S。

以上过程都是正常的，是应用程序设计者意料之中的事情。问题是，如果在任务 C 继续使用共享资源 S 过程中的 t_4 时刻，任务 B 所等待的事件也来临，由于任务 B 的优先级别高于任务 C 的优先级别，任务 B 当然要剥夺任务 C 的 CPU 使用权而进入运行状态，而任务 C 则只好等待。这样，任务 A 只有当任务 B 运行结束，并使任务 C 继续运行且释放了信号量的 t_6 时刻之后，才能获得信号量而得以重新运行。

综上所述，任务优先级低的任务 B 反而先于任务优先级高的任务 A 运行了。换句话说，从实际运行的结果来看，似乎任务 B 的优先级高于任务 A 了。系统中的这种现象叫做任务优先级的反转。

之所以出现上述的优先级反转现象，是因为一个优先级别较低的任务在获得了信号量使用共享资源期间，被具有较高优先级别的任务所打断而不能释放信号量，从而使正在等待这个信号量的更高级别的任务因得不到信号量而被迫处于等待状态，在这个等待期间，就让优先级别低于它而高于占据信号量的任务的任务先运行了。显然，如果这种优先级别介于使用信号量的两个任务优先级别中间的中等优先级别任务较多，则会极大地恶化高优先级别任务的运行环境，是实时系统所无法容忍的。

例 4-6　下面是一个使用信号量实现独占式访问共享资源而出现了任务优先级反转的应用程序示例。请运行该程序并分析它的运行结果。

```
/*************************Test ************************************/
#include "includes.h"
#define  TASK_STK_SIZE    512                          //任务堆栈长度
OS_STK   StartTaskStk[TASK_STK_SIZE];                  //定义任务堆栈区
OS_STK   MyTaskStk[TASK_STK_SIZE];                     //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];                    //定义任务堆栈区
OS_STK   HerTaskStk[TASK_STK_SIZE];                    //定义任务堆栈区
INT16S   key;                                          //用于退出的键
char * s1 = "MyTask 正在运行";
char * s2 = "YouTask 正在运行";
char * s3 = "HerTask 正在运行";
char * ss = "MyTask 请求信号量";
INT8U err;
INT8U y = 0;                                           //字符显示位置
INT32U Times = 0;
OS_EVENT * Semp;                                       //定义事件控制块
void  StartTask(void * data);                          //声明起始任务
void  MyTask(void * data);                             //声明任务
void  YouTask(void * data);                            //声明任务
void  HerTask(void * data);                            //声明任务
/************************* 主函数 ***********************************/
void  main (void)
{
    OSInit();                                          //初始化 μC/OS－II
    PC_DOSSaveReturn();                                //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                         //安装 μC/OS－II 中断
    Semp = OSSemCreate (1);                            //定义信号量
    OSTaskCreate(StartTask,                            //创建任务 StartTask
                 (void * )0,
                 &StartTaskStk[TASK_STK_SIZE - 1],
                 0);
    OSStart();                                         //启动多任务管理
}
/************************* 任务 StartTask ***************************/
void  StartTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
```

```
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                    //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);               //设置时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                                   //初始化统计任务
    OSTaskCreate(MyTask,                            //创建任务 MyTask
                 (void*)0,
                 &MyTaskStk[TASK_STK_SIZE - 1],
                 3);
    OSTaskCreate(YouTask,                           //创建任务 YouTask
                 (void*)0,
                 &YouTaskStk[TASK_STK_SIZE - 1],
                 4);
    OSTaskCreate(HerTask,                           //创建任务 HerTask
                 (void*)0,
                 &HerTaskStk[TASK_STK_SIZE - 1],
                 5);
    for (;;)
    {
    //如果按下 ESC 键,则退出 μC/OS - II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDlyHMSM(0, 0, 3, 0);                  //等待 3 s
    }
}
/*************************任务 MyTask *******************************/
void  MyTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
```

```
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        OSTimeDlyHMSM(0, 0, 1, 200);                    //等待 200 ms
        {
                    PC_DispStr(10, ++y,
                        ss,
                        DISP_BGND_BLACK + DISP_FGND_WHITE );
            OSSemPend(Semp,0,&err);                     //请求信号量
                    PC_DispStr(10, ++y,
                        s1,
                        DISP_BGND_BLACK + DISP_FGND_WHITE );
                    OSSemPost(Semp);                    //发送信号量
        }
        OSTimeDlyHMSM(0, 0, 0, 200);                    //等待 200 ms
    }
}
/*************************任务 YouTask ******************************/
void  YouTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        PC_DispStr(10, ++y,
                s2,
                DISP_BGND_BLACK + DISP_FGND_WHITE );
        OSTimeDlyHMSM(0, 0, 0, 300);                    //等待 300 ms
    }
}
/*************************任务 HerTask ******************************/
void  HerTask (void * pdata)
{
```

```
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        OSSemPend(Semp,0,&err);                    //请求信号量
            PC_DispStr(10, ++y,
                    s3,
                    DISP_BGND_BLACK + DISP_FGND_WHITE );
        for(Times;Times<20000000;Times++)          //延时
            {
                OS_Sched();
            }
        OSSemPost(Semp);                           //发送信号量
        OSTimeDlyHMSM(0, 0, 1, 0);                 //等待 1 s
    }
}
/*************************End ************************************/
```

例 4-6 应用程序的运行结果如图 4-16 所示。

图 4-16　优先级反转现象

通过例 4-6 可以发现，使用信号量的任务是否能够运行是受任务的优先级别以及是否占用信号量两个条件约束的，而信号量的约束高于优先级别的约束。于是当出现低优先级别的任务与高优先级别的任务使用同一个信号量，而系统中还存在其他中等优先级别的任务时，如果低优先级别的任务先获得了信号量，就会使高优先级别的任务处于等待状态；而那些不使用该信号量的中等优先级别的任务却可以剥夺低优先级别的任务的 CPU 使用权，而先于高优

先级别的任务而运行了。

解决问题的办法之一是，使获得信号量任务的优先级别在使用共享资源期间暂时提升到所有任务最高优先级的高一个级别上，以使该任务不被其他任务所打断，从而能尽快地使用完共享资源并释放信号量；然后在释放信号量之后，再恢复该任务原来的优先级别。

4.3.2 互斥型信号量

互斥型信号量是一个二值信号量，因此也叫做信号。任务可以用互斥型信号量来实现对共享资源的独占式处理。为了解决任务在使用独占式资源出现的优先级反转问题，互斥型信号量除了具有普通信号量的机制外，还有一些其他特性。

μC/OS－II 仍然用事件控制块来描述一个互斥型信号量，如图 4－17 所示。

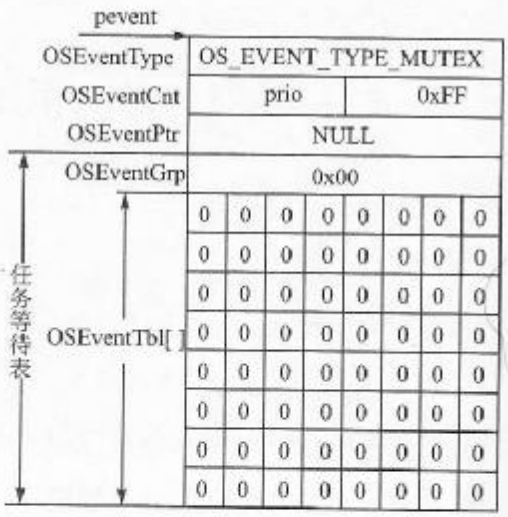


图 4－17 互斥型信号量的结构

在描述互斥型信号量的事件控制块中，除了成员 OSEventType 要赋以常数 OS_EVENT_TYPE_MUTEX 以表明这是一个互斥型信号量和仍然没有使用成员 OSEventPtr 之外，成员 OSEventCnt 被分成了低 8 位和高 8 位两部分：低 8 位用来存放信号值（该值为 0xFF 时，信号为有效，否则信号为无效）；高 8 位用来存放为了避免出现优先级反转现象而要提升的优先级别 prio。

1. 创建互斥型信号量

创建互斥型信号量需要调用函数 OSMutexCreate()。函数 OSMutexCreate()的原型如下：

```
OS_EVENT    * OSMutexCreate   (
                              INT8U prio,      //优先级别
                              INT8U * err      //错误信息
                              );
```

函数 OSMutexCreate()从空事件控制块链表获取一个事件控制块，把成员 OSEventType 赋以常数 OS_EVENT_TYPE_MUTEX，以表明这是一个互斥型信号量；然后再把成员 OSEventCnt 的高 8 位赋以 prio（要提升的优先级别），低 8 位赋以常数 OS_MUTEX_AVAILABLE（该常数值为 0xFFFF）的低 8 位（0xFF），以表明信号量尚未被任何任务所占用。

处于有效状态。

函数 OSMutexCreate()创建的互斥型信号量如图 4-17 所示。

2. 请求互斥型信号量

当任务需要访问一个独占式共享资源时，就要调用函数 OSMutexPend()来请求管理这个资源的互斥型信号量。如果信号量有信号(OSEventCnt 的低 8 位为 0xFF)，则意味着目前尚无任务占用资源，于是任务可以继续运行并对该资源进行访问；否则就进入等待状态，直至占用这个资源的其他任务释放了该信号量。

为防止任务因得不到信号量而处于长期的等待状态，函数 OSMutexPend()允许用参数 timeout 设置一个等待时间的限制。当任务等待的时间超出该时间限制值时，可以结束等待状态。

函数 OSMutexPend()的原型如下：

```
void  OSMutexPend  (
                    OS_EVENT * pevent,        //互斥型信号量指针
                    INT16U timeout,           //等待时限
                    INT8U * err               //错误信息
                    );
```

任务也可以通过调用函数 OSMutexAccept()无等待地请求一个互斥型信号量。该函数的原型如下：

```
INT8U  OSMutexAccept  (
                       OS_EVENT * pevent,     //互斥型信号量指针
                       INT8U * err            //错误信息
                       );
```

3. 发送互斥型信号量

任务可以通过调用函数 OSMutexPost()发送一个互斥型信号量。该函数的原型如下：

```
INT8U  OSMutexPost (
                    OS_EVENT * pevent         //互斥型信号量指针
                    );
```

4. 获取互斥型信号量的当前状态

任务可以通过调用函数 OSMutexQuery()获取互斥型信号量的当前状态。该函数的原型如下：

```
INT8U  OSMutexQuery (
                     OS_EVENT * pevent,       //互斥型信号量指针
                     OS_MUTEX_DATA * pdata    //存放互斥型信号量状态的结构
                     );
```

函数的参数 pdata 是 OS_MUTEX_DATA 结构类型的指针。函数被调用后，在 pdata 指向的结构中存放了互斥型信号量的相关信息。OS_MUTEX_DATA 结构定义如下：

```
typedef struct {
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];
    INT8U   OSEventGrp;
    INT8U   OSValue;
    INT8U   OSOwnerPrio;
    INT8U   OSMutexPIP;
} OS_MUTEX_DATA;
```

5. 删除互斥型信号量

任务调用函数 OSMutexDel()可以删除一个互斥型信号量。该函数的原型如下：

```
OS_EVENT    *OSMutexDel (
                        OS_EVENT *pevent,     //互斥型信号量指针
                        INT8U opt,            //删除方式选项
                        INT8U *err            //错误信息
                        );
```

例 4－7 在例 4－6 的应用程序中，把使用的信号量改互斥型信号量，然后运行该程序并观察其运行结果。

例 4－7 应用程序的运行结果如图 4－18 所示。

图 4－18 使用互斥型信号量消除了优先级反转现象

4.4 消息邮箱及其操作

4.4.1 消息邮箱

如果任务与任务之间要传递一个数据，那么为了适应不同数据的需要最好在存储器中建立一个数据缓冲区，把要传递的数据放在这个缓冲区中，就可以实现任务间的数据通信了。

如果把数据缓冲区的指针赋给一个事件控制块的成员 OSEventPrt，同时使事件控制块的成员 OSEventType 为常数 OS_EVENT_TYPE_MBOX，则该事件控制块就叫做消息邮箱。消息邮箱是在两个需要通信的任务之间通过传递数据缓冲区指针的方法来通信的。

邮箱的数据结构如图 4-19 所示。

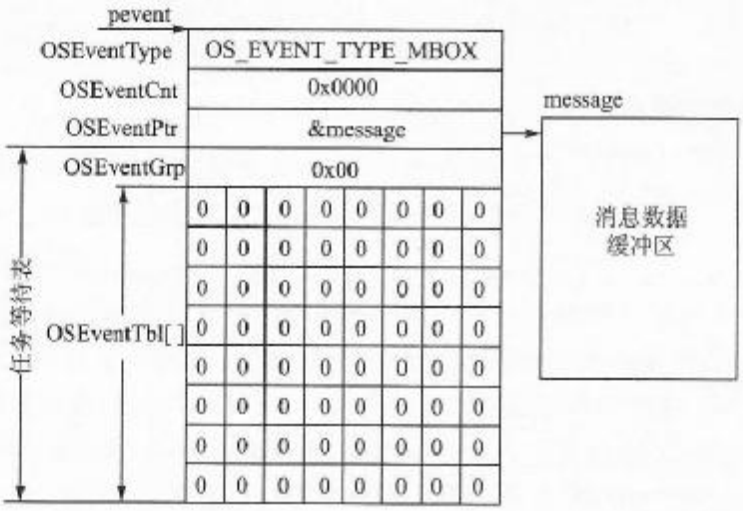


图 4-19 消息邮箱的结构

4.4.2 消息邮箱的操作

1. 创建消息邮箱

创建邮箱需要调用函数 OSMboxCreate()。该函数的原型如下：

```
OS_EVENT    * OSMboxCreate (
                            void * msg      //消息指针
                          );
```

函数中的参数 msg 为消息的指针，函数的返回值为消息邮箱的指针。

调用函数 OSMboxCreate()须先定义 msg 的初始值。在一般情况下，这个初始值为

NULL；但也可以事先定义一个邮箱，然后把这个邮箱的指针作为参数传递到函数 OSMboxCreate()中，使其一开始就指向一个邮箱。

函数 OSMboxCreate()的源代码如下：

```
OS_EVENT    * OSMboxCreate (void * msg)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif
    OS_EVENT    * pevent;

    if (OSIntNesting > 0)
    {
        return ((OS_EVENT * )0);
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;
    if (OSEventFreeList != (OS_EVENT * )0)
    {
        OSEventFreeList = (OS_EVENT * )OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT * )0)
    {
        pevent->OSEventType = OS_EVENT_TYPE_MBOX;
            pevent->OSEventCnt = 0;
            pevent->OSEventPtr = msg;
            OS_EventWaitListInit(pevent);
    }
    return (pevent);
}
```

2. 向消息邮箱发送消息

任务可以通过调用函数 OSMboxPost()向消息邮箱发送消息。该函数的原型如下：

```
INT8U  OSMboxPost (
                    OS_EVENT * pevent,     //消息邮箱指针
                    void * msg             //消息指针
                    );
```

函数中的第二个参数 msg 为消息缓冲区的指针，函数的返回值为错误号。

函数 OSMboxPost()的源代码如下：

```
INT8U  OSMboxPost (
                    OS_EVENT * pevent,          //消息邮箱指针
                    void * msg                  //消息指针
                  );
```

μC/OS-II 在 μC/OS 的基础上又增加了一个向邮箱发送消息的函数 OSMboxPostOpt()，这个函数可以广播的形式向事件等待任务表中的所有任务发送消息。该函数的原型如下：

```
INT8U  OSMboxPostOpt (
                        OS_EVENT * pevent,      //消息邮箱指针
                        void * msg,             //消息指针
                        INT8U opt               //广播选项
                      );
```

函数中的第三个参数 opt 用来说明是否把消息向所有等待任务广播。如果该值为 OS_POST_OPT_BROADCAST，则意味着把消息向所有等待任务广播；如果该值为 OS_POST_OPT_NONE，则把消息只向优先级别最高的等待任务发送。

3. 请求消息邮箱

当一个任务请求邮箱时需要调用函数 OSMboxPend()，这个函数的主要作用就是查看邮箱指针 OSEventPtr 是否为 NULL。如果邮箱指针 OSEventPtr 不是 NULL，则把邮箱中的消息指针返回给调用函数的任务，同时用 OS_NO_ERR 通过函数的参数 err 通知任务获取消息成功；如果邮箱指针 OSEventPtr 是 NULL，则使任务进入等待状态，并引发一次任务调度。

函数 OSMboxPend()的原型如下：

```
void   * OSMboxPend (
                     OS_EVENT * pevent,         //请求消息邮箱指针
                     INT16U timeout,            //等待时限
                     INT8U * err                //错误信息
                     );
```

任务在请求邮箱失败时也可以不进行等待而继续运行。如果要以这种方式来请求邮箱，则任务需要调用函数 OSMboxAccept()。该函数的原型如下：

```
void   * OSMboxAccept (
                       OS_EVENT * pevent        //消息邮箱指针
                       );
```

该函数的返回值为消息的指针。

4. 查询邮箱的状态

任务可以调用函数 OSMboxQuery()查询邮箱的当前状态，并把相关信息存放在一个结构 OS_MBOX_DATA 中。

```
INT8U  OSMboxQuery (
                    OS_EVENT * pevent,          //消息邮箱指针
                    OS_MBOX_DATA * pdata        //存放邮箱信息的结构
                    );
```

OS_MBOX_DATA 结构如下：

```
typedef struct
{
    void     * OSMsg;
    INT8U    OSEventTbl[OS_EVENT_TBL_SIZE];
    INT8U    OSEventGrp;
} OS_MBOX_DATA;
```

5. 删除邮箱

任务可以调用函数 OSMboxDel()来删除一个邮箱。该函数的原型如下：

```
OS_EVENT   * OSMboxDel (
                        OS_EVENT * pevent,      //消息邮箱指针
                        INT8U opt,              //删除选项
                        INT8U * err             //错误信息
                        );
```

例 4－8 设计一个应用程序，该程序有两个任务 MyTask 和 YouTask。在任务 MyTask 中用一个变量 Times 记录任务 MyTask 的运行次数，并将其作为消息用邮箱 Str_Box 发给任务 YouTask 且由任务 YouTask 显示出来。

答 应用程序主要代码如下：

```
/*************************Test ************************************/
#include "includes.h"
#define  TASK_STK_SIZE    512                    //任务堆栈长度
OS_STK   StartTaskStk[TASK_STK_SIZE];            //定义任务堆栈区
OS_STK   MyTaskStk[TASK_STK_SIZE];               //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];              //定义任务堆栈区
INT16S   key;                                    //用于退出的键
char * s;
```

```
char * ss;
INT8U err;
INT8U y = 0;                                    //字符显示位置
INT32U Times = 0;
OS_EVENT * Str_Box;                             //定义事件控制块指针
void  StartTask(void * data);                   //声明起始任务
void  MyTask(void * data);                      //声明任务
void  YouTask(void * data);                     //声明任务
/************************* 主函数 ***********************************/
void  main (void)
{
    OSInit();                                   //初始化 μC/OS-II
    ……
    Str_Bcx = OSMboxCreate ((void * )0);        //创建消息邮箱
    OSTaskCreate(StartTask,                     //创建任务 StartTask
                 (void * )0,
                 &StartTaskStk[TASK_STK_SIZE - 1],
                 0);                            //任务的优先级别为 0
    OSStart();                                  //启动多任务管理
}
/************************* 任务 StartTask ********************************/
void  StartTask (void * pdata)
{
……
    OSStatInit();                               //初始化统计任务
    OSTaskCreate(MyTask,                        //创建任务 MyTask
                 (void * )0,
                 &MyTaskStk[TASK_STK_SIZE - 1],
                 3);                            //任务的优先级别为 3
    OSTaskCreate(YouTask,                       //创建任务 YouTask
                 (void * )0,
                 &YouTaskStk[TASK_STK_SIZE - 1],
                 4);                            //任务的优先级别为 4
    for (;;)
    {
    //如果按下 ESC 键,则退出 μC/OS-II
        if (PC_GetKey(&key) == TRUE)
```

```
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDlyHMSM(0, 0, 3, 0);                    //等待 3 s
    }
}
/************************任务 MyTask ********************************/
void  MyTask (void * pdata)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif
    pdata = pdata;

    for (;;)
    {
        sprintf(s," % d" ,Times);                     //记录运行次数
        OSMboxPost(Str_Box,s);                        //发送消息
        Times ++ ;
        OSTimeDlyHMSM(0, 0, 1, 0);                    //等待 1 s
    }
}
/************************任务 YouTask ********************************/
void  YouTask (void * pdata)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif
    pdata = pdata;

    for (;;)
    {
        ss = OSMboxPend(Str_Box,10,&err);             //请求消息邮箱
        PC_DispStr(10, ++ y,
```

```
                    ss,
                    DISP_BGND_BLACK + DISP_FGND_WHITE);
        OSTimeDlyHMSM(0, 0, 1, 0);                    //等待 1 s
    }
}
/*************************** End ***************************************/
```

例 4-8 应用程序的运行结果见图 4-20。

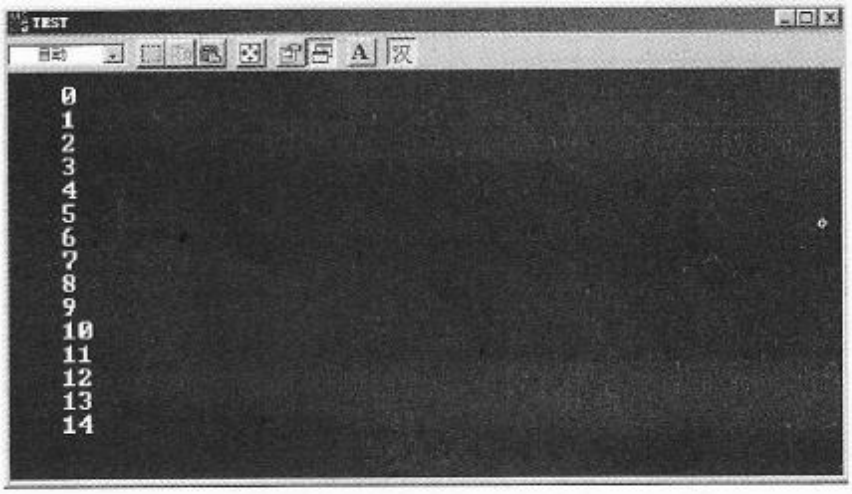


图 4-20 例 4-8 应用程序的运行结果

4.5 消息队列及其操作

4.5.1 消息队列

使用消息队列可在任务之间传递多条消息。消息队列由 3 部分组成：事件控制块、消息队列和消息。

当把事件控制块成员 OSEventType 的值置为 OS_EVENT_TYPE_Q 时，该事件控制块描述的就是一个消息队列。

消息队列的数据结构如图 4-21 所示。从图中可以看到，消息队列相当于一个共用一个任务等待列表的消息邮箱数组，事件控制块成员 OSEventPtr 指向一个叫做队列控制块(OS_Q)的结构，该结构管理一个数组 MsgTbl[ ]，该数组中的元素都是一些指向消息的指针。

1. 消息指针数组

消息队列的核心是消息指针数组。图 4-22 表示了消息指针数组的结构，其中各参数的

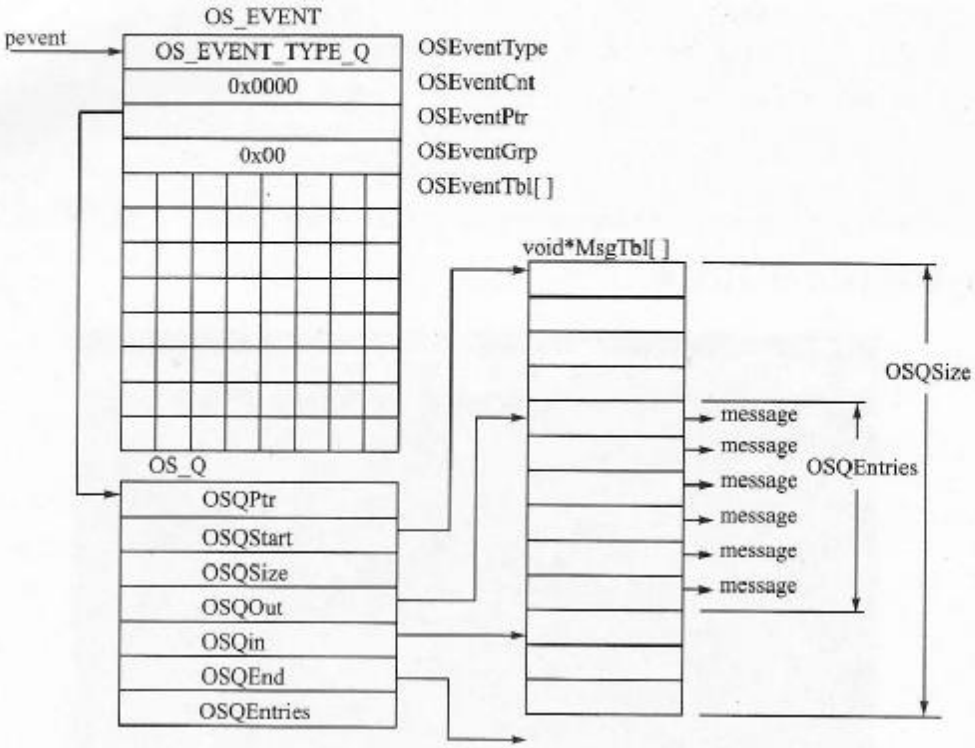


图 4－21　消息队列的数据结构

含义见表 4－3。

表 4－3　图 4－7 中各参数的含义

参　数	说　明
OSQSize	数组的长度
OSQEntres	已存放消息指针的元素数目
OSQStart	指针，指向消息指针数组的起始地址
OSQEnd	指针，指向消息指针数组结束单元的下一个单元。它使得数组构成了一个循环的缓冲区
OSQIn	指针，指向插入一条消息的位置。当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元
OSQOut	指针，指向被取出消息的位置，当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元

其中，可以移动的指针为 OSQIn 和 OSQOut，而指针 OSQStart 和 OSQEnd 只是一个标志（常指针）。当可移动的指针 OSQIn 或 OSQOut 移动到数组末尾，也就是与 OSQEnd 相等时，可移动的指针将会被调整到数组的起始位置 OSQStart。也就是说，从效果上看，指针 OSQEnd 与 OSQStart 等值。于是，这个由消息指针构成的数组就头尾衔接起来形成了一个如图 4－23 所示的循环队列。

向指针数组中插入消息指针的方式有 2 种：先进先出（FIFO）方式和后进先出（LIFO）方

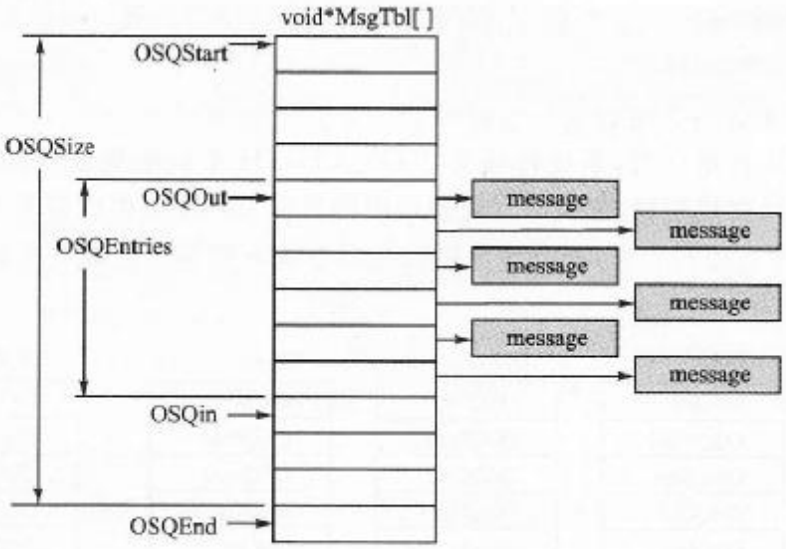


图 4-22 消息指针数组

式。当采用先进先出方式时，消息队列将在指针 OSQIn 指向的位置插入消息指针，而把指针 OSQOut 指向消息指针作为输出。当采用后进先出方式时，则只使用指针 OSQOut，当向队列插入消息指针时，指针 OSQOut 将先移动到图 4-23 所示的位置（虚线所示），再按指针 OSQOut 指向的位置插入消息指针；输出时则指针 OSQOut 无须进行移动，就把指针 OSQOut 指向的消息指针输出。

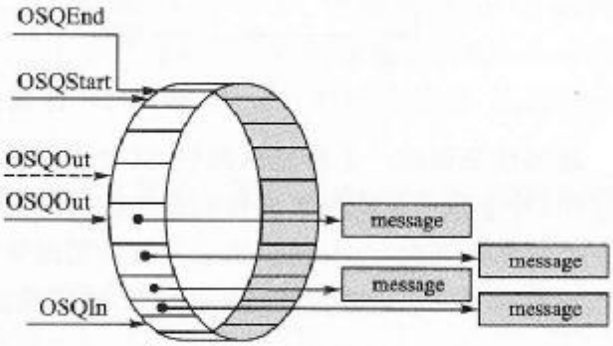


图 4-23 消息指针数组是一个环形的数据缓冲区

2. 队列控制块

为了对图 4-22 所示的消息指针数组进行有效的管理，μC/OS-II 把消息指针数组的基本参数都记录在一个叫做队列控制块的结构中。队列控制块的结构如下：

```
typedef struct os_q
{
    struct os_q * OSQPtr;
    void * * OSQStart;
    void * * OSQEnd;
    void * * OSQIn;
    void * * OSQOut;
```

```
    INT16U  OSQSize;
    INT16U  OSQEntries;
} OS_Q;
```

在 μC/OS－II 初始化时，系统将按文件 OS_CFG.H 中的配置常数 OS_MAX_QS 定义 OS_MAX_QS 个队列控制块，并用队列控制块中的指针 OSQPtr 将所有队列控制块链接为链表。由于这时还没有使用它们，因此这个链表叫做空队列控制块链表。空队列控制块链表如图 4－24 所示。

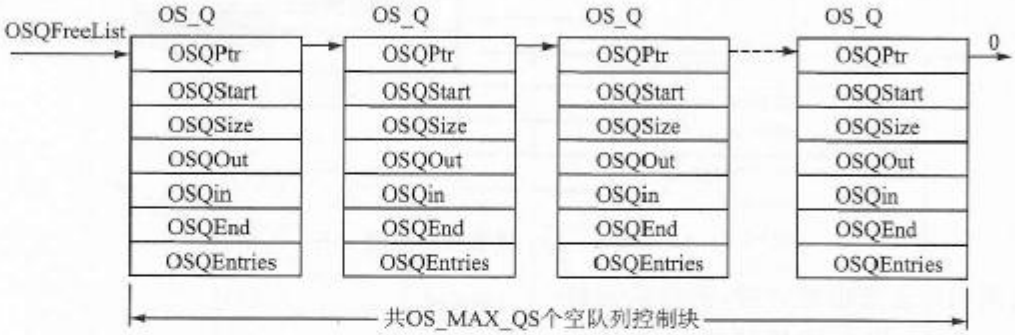


图 4－24　空队列控制块链表

每当任务创建一个消息队列时，就会在空队列控制块链表中摘取一个控制块供消息队列来使用，并令该消息队列事件控制块中的指针 OSEventPtr 指向这个队列控制块；而当任务释放一个消息队列时，就会将该消息队列使用的队列控制块归还空队列控制块链表。

消息队列的事件控制块、队列控制块、消息指针数组和消息之间的关系见图 4－21。

4.5.2　消息队列的操作

1. 创建消息队列

创建一个消息队列首先需要定义一个指针数组，然后把各个消息数据缓冲区的首地址存入这个数组中，最后再调用函数 OSQCreate()来创建消息队列。创建消息队列函数 OSQCreate()的原型如下：

```
OS_EVENT OSQCreate(
                    void * * start,     //指针数组的地址
                    INT16U size         //数组长度
                  );
```

函数中的参数 start 为存放消息缓冲区指针数组的地址；参数 size 为该数组的大小。函数的返回值为消息队列的指针。

函数 OSQCreate()首先从空闲队列控制块链表摘取一个控制块并按参数 start 和 size 填

写诸项，然后把消息队列初始化为空（即其中不包含任何消息）。

2. 请求消息队列

请求消息队列的目的是为了从消息队列中获取消息。任务请求消息队列需要调用函数OSQPend()。该函数的原型如下：

```
void * OSQPend(
                OS_EVENT * pevent,      //所请求的消息队列的指针
                INT16U timeout,         //等待时限
                INT8U * err             //错误信息
            );
```

函数的返回值为消息指针。

函数的参数 pevent 是要访问的消息队列事件控制块的指针；参数 timeout 是任务等待的时限。

函数要通过访问事件控制块的成员 OSEventPtr 指向的队列控制块 OS_Q 的成员 OSQEntries 来判断是否有消息可用。如果有消息可用，则返回 OS_Q 成员 OSQOut 指向的消息，同时调整指针 OSQOut，使之指向下一条消息并把有效消息数的变量 OSQEntries 减 1；如果无消息可用（即 OSQEntries=0），则使用调用函数 OSQPend()的任务挂起，使之处于等待状态并引发一次任务调度。

如果希望任务无等待地请求一个消息队列，则需要调用函数 OSQAccept()。该函数的原型如下：

```
void OSQAccept(
                OS_EVENT * pevent       //所请求的消息队列的指针
            );
```

3. 向消息队列发送消息

任务需要通过调用函数 OSQPost()或 OSQPostFront()来向消息队列发送消息。函数 OSQPost()以 FIFO（先进先出）的方式组织消息队列；函数 OSQPostFront()以 LIFO（后进先出）的方式组织消息队列。这两个函数的原型如下：

```
INT8U OSQPost(
                OS_EVENT * pevent,      //消息队列的指针
                void * msg              //消息指针
            );
```

和

```
/*************************主函数*********************************/
void  main (void)
{
    OSInit();                                   //初始化 μC/OS-II
    PC_DOSSaveReturn();                         //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                  //安装 μC/OS-II 中断
    Str_Q = OSQCreate (&MsgGrp[0],N_MESSAGES);  //创建消息队列
    OSTaskCreate(StartTask,                     //创建任务 StartTask
                 (void*)0,
                 &StartTaskStk[TASK_STK_SIZE - 1],
                 0);
    OSStart();                                  //启动多任务管理
}
/*************************任务 StartTask*********************************/
void  StartTask (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);           //设置 μC/OS-II 时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                               //初始化统计任务
    OSTaskCreate(MyTask,                        //创建任务 MyTask
                 (void*)0,
                 &MyTaskStk[TASK_STK_SIZE - 1],
                 3);
    OSTaskCreate(YouTask,                       //创建任务 YouTask
                 (void*)0,
                 &YouTaskStk[TASK_STK_SIZE - 1],
                 4);
    s="这个串能收到几次?";
    OSQPostFront(Str_Q,s);                      //发送消息

    for (;;)
    {
```

```
    if(OSTimeGet()>100 && OSTimeGet()<500)
    {
        s100 = "现在 OSTime 的值在 100 到 500 之间";
        OSQPostFront(Str_Q,s100);                //发送消息
        s = "这个串是哪个任务收到的?";
        OSQPostFront(Str_Q,s);                   //发送消息

    }
    if(OSTimeGet()>5000 && OSTimeGet()<5500)
    {
        s500 = "现在 OSTime 的值在 5000 到 5500 之间";
        OSQPostFront(Str_Q,s500);                //发送消息
    }
    //如果按下 ESC 键,则退出 μC/OS－II
    if (PC_GetKey(&key) == TRUE)
    {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
    }
        OSTimeDlyHMSM(0, 0, 1, 0);               //等待 1 s
    }
}
/************************* 任务 MyTask ********************************/
void  MyTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;

    for (;;)
    {
        ss = OSQPend(Str_Q,0,&err);              //请求消息队列
        PC_DispStr(10, ++y,
                   ss,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
```

```
        OSTimeDlyHMSM(0, 0, 1, 0);              //等待 1 s
    }
}
/*************************任务 YouTask ******************************/
void  YouTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        ss = OSQPend(Str_Q,0,&err);             //请求消息队列
        PC_DispStr(10, ++y,
                   ss,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
        OSTimeDlyHMSM(0, 0, 1, 0);              //等待 1 s
    }
}

/*************************End ****************************************/
```

例 4－9 应用程序的运行结果如图 4－25 所示。

图 4－25　例 4－9 应用程序的运行结果

4. 清空消息队列

任务可以通过调用函数 OSQFlush()来清空消息队列。该函数的原型如下：

```
INT8U OSQFlush(
                OS_EVENT * pevent          //消息队列指针
              );
```

5. 删除消息队列

任务可以通过调用函数 OSQDel()来删除一个已存在的消息队列。该函数的原型如下：

```
OS_EVENT * OSQDel(
                    OS_EVENT * pevent      //消息队列指针
                  );
```

6. 查询消息队列

任务可以通过调用函数 OSQQuery()来查询一个消息队列的状态。该函数的原型如下：

```
INT8U OSQQuery(
                OS_EVENT * pevent,         //消息队列指针
                OS_Q_DATA * pdata          //存放状态信息的结构
              );
```

函数中的参数 pdata 是 OS_Q_DATA * pdata 类型的指针。OS_Q_DATA * pdata 的结构如下：

```
typedef struct {
    void     * OSMsg;
    INT16U   OSNMsgs;
    INT16U   OSQSize;
    INT8U    OSEventTbl[OS_EVENT_TBL_SIZE];
    INT8U    OSEventGrp;
} OS_Q_DATA;
```

函数 OSQQuery()的查询结果就放在以 OS_Q_DATA 为类型的变量中。

4.6 小 结

- 在 μC/OS - II 中，信号量是一个表明共享资源被使用情况的标志，该标志实质上是一个计数器。如果计数器的初值大于 1，则叫做信号量；如果计数器的值只能为 1 和 0 两个数值，则叫做信号。

- 能防止出现优先级反转现象的信号叫做互斥型信号量。
- 消息邮箱是能在任务之间传递消息指针的数据结构。
- 消息队列是能在任务之间传递一组消息指针的数据结构。
- 信号量、消息邮箱和消息队列都叫做"事件"，每个事件都有一个用来记录等待事件的任务的表——等待任务表，而任务的等待时限则记录在 OSTCBDly 中。
- μC/OS－II 统一用事件控制块来描述各种事件。
- 通过事件发送事件，获取事件信息叫做请求事件。
- 操作系统中各任务之间的同步与通信是通过各种各样的事件来完成的。

4.7 练习题

1. 叙述信号量的工作过程。
2. 使用信号量可在应用程序中完成哪些工作？
3. 什么叫优先级反转？这种现象会在什么情况下发生？有什么危害？
4. 互斥型信号量是如何防止优先级反转现象出现的？
5. 什么叫消息邮箱？
6. 能否使用全局变量来实现任务间的通信？如果可以，它有什么缺点？
7. 使用消息邮箱实现任务之间的通信有什么好处？
8. 什么是消息队列？
9. 想一想，在应用程序中消息队列都可以做些什么工作？

第5章 信号量集

在实际应用中，任务常常需要与多个事件同步，即要根据多个信号量组合作用的结果来决定任务的运行方式。μC/OS-II 为了实现多个信号量组合的功能定义了一种特殊的数据结构——信号量集。

信号量集所能管理的信号量都是一些二值信号。所有信号量集实质上是一种可以对多个输入的逻辑信号进行基本逻辑运算的组合逻辑，其示意图如图 5-1 所示。

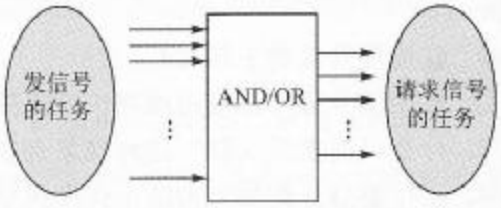


图 5-1 信号量集的示意图

在 μC/OS-II 中，请求信号量集的任务得以继续运行的条件与所请求的信号量之间有两种逻辑关系："与"(AND 或 ALL)关系和"或"(OR 或 ANY)关系。在"与"关系下，只有当任务所请求的信号量都有效时，任务才能继续运行；而在"或"关系下，只要在任务所请求信号量中有信号有效，任务就会继续运行。

本章的主要内容有：

- 信号量集的基本概念和信号量集的结构；
- 操作等待任务链表的函数；
- 任务可以调用的信号量集操作函数。

5.1 信号量集的结构

μC/OS-II 的信号量集由两部分组成：第一部分叫做标志组，其中存放了信号量集中的所有信号；第二部分叫做等待任务链表，链表中的每个节点都对应一个正在等待信号量集的等待任务，信号量集根据这个链表来管理等待任务。

5.1.1 信号量集的标志组

不同于信号量、消息邮箱和消息队列等事件，μC/OS-II 不使用事件控制块来描述信号量集，而使用了一个叫做标志组的结构 OS_FLAG_GRP。

OS_FLAG_GRP 结构如下：

```
typedef struct{
    INT8U       OSFlagType;         //识别是否为信号量集的标识
    void        *OSFlagWaitList;    //指向等待任务链表的指针
    OS_FLAGS    OSFlagFlags;        //所有信号列表
}OS_FLAG_GRP;
```

结构中，成员 OSFlagType 是信号量集的标识，该成员变量的值应该固定为 OS_EVENT_TYPE_FLAG。

成员 OSFlagFlags 是 OS_FLAGS 类型的变量，该变量用来存放信号量集所有信号的状态，每一个信号占据一个二进制位。信号量集中可以存放多少个信号，取决于 OSFlagFlags 的长度。这个长度可以根据应用程序需要信号的数目定义为 8 位、16 位或 32 位（在文件 OS_CFG.H 中来定义，目前系统定义的是 16 位）。图 5-2 是把 OS_FLAGS 定义为 INT8U 类型时（信号量集最多可描述 8 个事件的状态），信号量集标志组的结构示意图。

成员 OSFlagWaitList 是一个指针，当一个信号量集被创建后，这个指针指向了这个信号量集的等待任务链表。

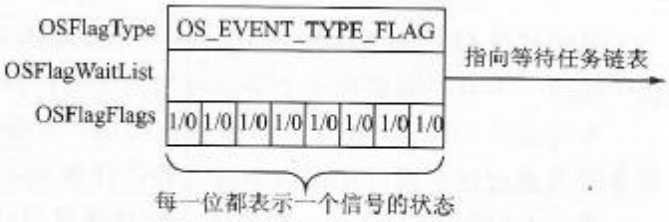


图 5-2 信号量集标志组的示意图

在 μC/OS-II 初始化时，系统会根据在文件 OS_CFG.H 中定义的常数 OS_MAX_FLAGS 来创建 OS_MAX_FLAGS 个标志组，并借用成员 OSFlagWaitList 作为指针把这些标志组链接成一个单向链表。由于这个链表中的各个标志组还未被真正创建，因此这个链表叫做空标志组链表。

空标志组链表的头指针存放在系统全局变量 OSFlagList 中，每当应用程序创建一个信号量集时，就从这个链表中取一个标志组，并移动头指针 OSFlagList，使之指向下一个空标志组。

空标志组链表的结构如图 5-3 所示。

图 5-3 空信号量集标志组表链表

5.1.2 等待任务链表

与其他前面介绍过的事件不同，信号量集用一个双向链表来组织等待任务，每一个等待任务都是该链表中的一个节点(Node)。标志组 OS_FLAG_GRP 的成员 OSFlagWaitList 就指向了信号量集的这个等待任务链表。

等待任务链表节点 OS_FLAG_NODE 的结构如下：

```
typedef struct {
    void          *OSFlagNodeNext;        //指向下一个节点的指针
    void          *OSFlagNodePrev;        //指向前一个节点的指针
    void          *OSFlagNodeTCB;         //指向对应任务的任务控制块的指针
    void          *OSFlagNodeFlagGrp;     //反向指向信号量集的指针
    OS_FLAGS      OSFlagNodeFlags;        //信号过滤器
    INT8U         OSFlagNodeWaitType;     //定义逻辑运算关系的数据
} OS_FLAG_NODE;
```

其中成员 OSFlagNodeFlagGrp 是一个反向指向信号量集标志组的指针，是在等待任务链表中删除一个节点或添加一个节点时用到的指针。

节点成员 OSFlagNodeTCB 是指向等待任务 TCB(任务控制块)的指针，信号量集的等待任务链表通过这个指针把链表节点与等待任务关联了起来。

成员 OSFlagNodeFlags 相当于一个过滤器，利用它可在标志组成员 OSFlagNodeFlags 的信号中只把请求任务需要的信号筛选出来，而把其余信号屏蔽掉。也就是说，一个请求信号量集的任务可以需要信号量集的所有信号，也可以只需要其中的部分信号，它究竟需要哪些信号，要通过在成员 OSFlagNodeFlags 中与所等待信号对应的二进制位进行置 1 来指定，而任务不需要的信号的位置 0。简单地说，OSFlagNodeFlags 就是一个屏蔽字，通过它可获得信号量集信号的子集。

等待的任务，只有在所需要的信号有效且满足指定的逻辑关系时，才能由等待状态进入就绪状态。这个逻辑关系可以通过给结构 OS_FLAG_NODE 中的成员 OSFlagNodeWaitType 赋值的方法来指定，这个值可以是表 5－1 所列举的 4 个常数之一。表 5－1 所列举的 4 个常数既指定了逻辑关系，也指定了信号的有效状态。

图 5－4 是信号量集等待任务链表中一个节点的示意图。图中 OSFlagNodeFlags 的值表明等待任务所等待的信号是标志组成员 OSFlagFlags 中的第 0、第 4 和第 7 个信号；OSFlagNodeWaiteType 的值表明：信号的有效状态被定义为 0，即当任务所等待的第 0、第 4 和第 7 个信号的状态都为 0 时，任务才可以结束等待状态而进入就绪状态。

图 5－5 是信号量集对信号的逻辑运算与控制的示意图。从图中可以清楚地看出，信号量集用 OSFlagFlags 来记录信号，用 OSFlagNodeFlags 来筛选信号，用 OSFlagNodeWaiteType

来控制信号的有效状态和信号量集有效之间的逻辑关系。

表 5-1 定义信号的有效状态及等待任务与信号之间的逻辑关系的常数

常 数	信号有效状态	等待任务的就绪条件
OS_FLAG_WAIT_CLR_ALL 或 OS_FLAG_WAIT_CLR_AND	0	信号全部有效(全 0)
OS_FLAG_WAIT_CLR_ANY 或 OS_FLAG_WAIT_CLR_OR	0	信号有 1 个或 1 个以上有效(有 0)
OS_FLAG_WAIT_ SET_ALL 或 OS_FLAG_WAIT_ SET _AND	1	信号全部有效(全 1)
OS_FLAG_WAIT_ SET_ANY 或 OS_FLAG_WAIT_ SET _OR	1	信号有 1 个或 1 个以上有效(有 1)

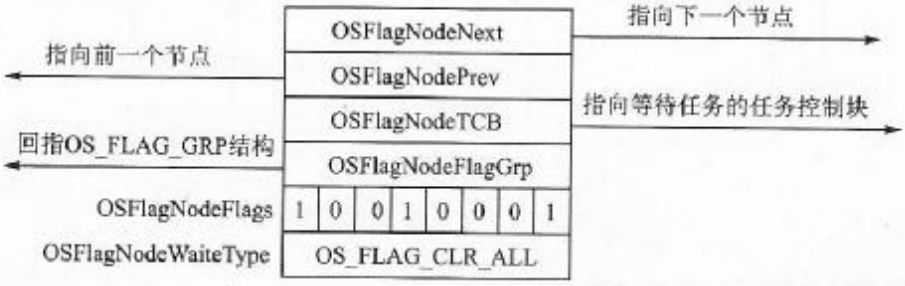


图 5-4 信号量集等待任务链表的一个节点

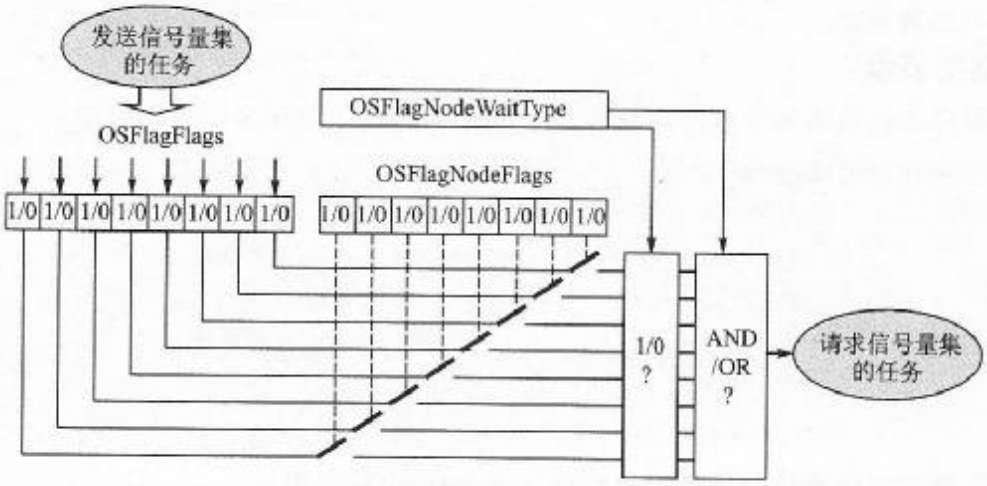


图 5-5 信号量集对信号的逻辑运算及控制

把等待任务链表的节点链接起来就形成了等待任务链表。在等待任务链表的基础上，再加上标志组和各个节点对应的任务控制块就形成了整个的信号量集。信号量集的示意图如图 5－6 所示。

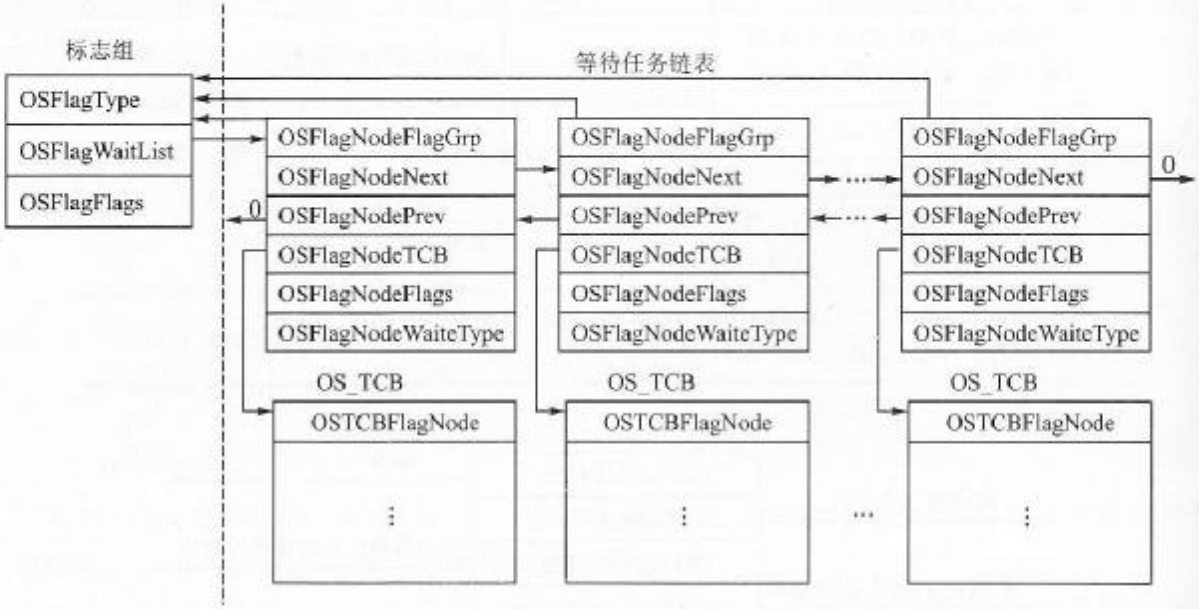


图 5－6　信号量集结构图

5.1.3　对等待任务链表的操作

μC/OS－II 定义了两个对等待任务链表的基本操作：添加节点和删除节点，以供对信号量集操作的函数调用。

1. 添加节点

给等待任务链表添加节点的函数为 OS_FlagBlock()。该函数的原型如下：

```
static  void  OS_FlagBlock (
                OS_FLAG_GRP  * pgrp,      //信号量集指针
                OS_FLAG_NODE  * pnode,    //待添加的等待任务节点指针
                OS_FLAGS flags,           //指定等待信号的数据
                INT8U wait_type,          //信号与等待任务之间的逻辑
                INT16U timeout            //等待时限
                );
```

这个函数将在请求信号量集函数 OSFlagPend()中被调用。

2. 删除节点

从等待任务链表中删除一个节点的函数为 OS_FlagUnlink()。该函数的原型如下：

```
void  OS_FlagUnlink (OS_FLAG_NODE * pnode);
```

这个函数将在发送信号量集函数 OSFlagPost()中被调用。

5.2 信号量集的操作

5.2.1 创建信号量集

任务可以通过调用函数 OSFlagCreate()来创建一个信号量集。该函数的原型如下：

```
OS_FLAG_GRP   * OSFlagCreate (
                              OS_FLAGS flags,     //信号的初始值
                              INT8U * err         //错误信息
                              );
```

函数 OSFlagCreate()的源代码如下：

```
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)
    {
        * err = OS_ERR_CREATE_ISR;
        return ((OS_FLAG_GRP * )0);
    }
    OS_ENTER_CRITICAL();
    pgrp = OSFlagFreeList;
    if (pgrp != (OS_FLAG_GRP * )0)
    {
        OSFlagFreeList
                = (OS_FLAG_GRP * )OSFlagFreeList->OSFlagWaitList;
        pgrp->OSFlagType = OS_EVENT_TYPE_FLAG;
        pgrp->OSFlagFlags = flags;
        pgrp->OSFlagWaitList = (void * )0;
        OS_EXIT_CRITICAL();
        * err = OS_NO_ERR;
```

```
    }
    else
    {
        OS_EXIT_CRITICAL();
        *err = OS_FLAG_GRP_DEPLETED;
    }
    return (pgrp);
}
```

从函数的源代码中可知，创建信号量集的函数主要做了两项工作：一是从空标志组链表中取下一个标志组，并同时给成员 OSFlagType 和 OSFlagFlags 赋初值；二是令指向等待任务链表的指针 OSFlagWaitList 为空指针。

在实际应用中，OSFlagFlags 可以根据需要赋值。

创建一个信号量集分为两个步骤：首先要定义一个全局的 OS_FLAG_GRP 类型的指针，然后在应用程序需要创建信号量集的位置调用函数 OSFlagCreate()。例如：

```
OS_FLAG_GRP  *FlagPtr;
INT8U err;

void main(void)
{
    ......
    OSInit();
    ......
    FlagPtr = OSFlagCreate (
                            (OS_FLAGS)0,        //所有信号的初始值为 0
                            &err
                            );
    ......
}
```

调用创建信号量集 OSFlagCreate()成功后，该函数返回的是这个信号量集的标志组的指针，应用程序可以用这个指针对信号量集进行相应的操作。

5.2.2 请求信号量集

任务可以通过调用函数 OSFlagPend()请求一个信号量集。该函数的原型如下：

```
OS_FLAGS  OSFlagPend (
                OS_FLAG_GRP *pgrp,      //所请求的信号量集指针
                OS_FLAGS flags,         //滤波器
                INT8U wait_type,        //逻辑运算类型
                INT16U timeout,         //等待时限
                INT8U *err              //错误信息
                    );
```

函数参数 flags 是用来给等待任务链表节点成员 OSFlagNodeFlags 赋值的；参数 wait_type 应该是表 5-2 所列举的 4 个常数之一。

函数调用成功后，将返回标志组成员 OSFlagFlags 的值。

任务也可通过调用函数 OSFlagAccept()无等待地请求一个信号量集。该函数的原型如下：

```
OS_FLAGS  OSFlagAccept (
                OS_FLAG_GRP *pgrp,      //所请求的信号量集指针
                OS_FLAGS flags,         //请求的信号
                INT8U wait_type,        //任务就绪与信号之间的逻辑关系
                INT8U *err              //错误信息
                    );
```

函数 OSFlagAccept()的参数除了少了一个等待时限 timeout 之外，其余与函数 OSFlagPend()的参数相同，其返回值也是标志组的成员 OSFlagFlags。

5.2.3 向信号量集发信号

任务可以通过调用函数 OSFlagPost()向信号量集发信号。该函数的原型如下：

```
OS_FLAGS  OSFlagPost (
                OS_FLAG_GRP *pgrp,      //信号量集指针
                OS_FLAGS flags,         //选择所要发送的信号
                INT8U opt,              //信号有效的选项
                INT8U *err              //错误信息
                    );
```

所谓任务向信号量集发信号，就是对信号量集标志组中的信号进行置 1(置位)或置 0(复位)的操作。至于对信号量集中的哪些信号进行操作，由函数中的参数 flags 来指定；对指定的信号是置 1 还是置 0，由函数中的参数 opt 来指定(opt = OS_FLAG_SET 为置 1 操作；opt = OS_FLAG_CLR 为置 0 操作)。

例如，要对信号量集 FlagPtr 发送信号，待发送的信号为 OSFlagFlags 中的第 0 位和第 3

位并且是要把它们置 1，则调用时的代码如下：

```
OS_FLAGS  OSFlagPost (
                    FlagPtr,                    //信号量集指针
                    (OS_FLAGS ) 9,              //选择所要发送的信号
                    OS_FLAG_SET,                //信号有效的选项
                    &err                        //错误信息
                    );
```

例 5－1 设计一个有三个任务的应用程序，这三个任务分别叫做 MyTask、YouTask 和 HerTask。要求用一个信号量集来控制 MyTask 的运行，即任务 YouTask 发送一个信号，任务 HerTask 发送一个信号，当这两个任务都发了信号之后，MyTask 才能运行。

答 应用程序的代码如下：

```
/*************************Test*************************************/
#include "includes.h"
#define  TASK_STK_SIZE   512                    //任务堆栈长度
OS_STK   StartTaskStk[TASK_STK_SIZE];           //定义任务堆栈区
OS_STK   MyTaskStk[TASK_STK_SIZE];              //定义任务堆栈区
OS_STK   YouTaskStk[TASK_STK_SIZE];             //定义任务堆栈区
OS_STK   HerTaskStk[TASK_STK_SIZE];             //定义任务堆栈区
INT16S   key;                                   //用于退出的键
char * s1 = "MyTask 正在运行";
char * s2 = "YouTask 正在运行";
char * s3 = "HerTask 正在运行";
INT8U err;
INT8U y = 0;                                    //字符显示位置
OS_FLAG_GRP * Sem_F ;
void  StartTask(void * data);                   //声明起始任务
void  MyTask(void * data);                      //声明任务
void  YouTask(void * data);                     //声明任务
void  HerTask(void * data);                     //声明任务
/*************************主函数*********************************/
void  main (void)
{
    OSInit();                                   //初始化 μC/OS－II
    PC_DOSSaveReturn();                         //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                  //安装中断
```

```
    Sem_F = OSFlagCreate (0,&err);                  //创建信号量集
    OSTaskCreate(StartTask,                         //创建任务 StartTask
                 (void * )0,
                 &StartTaskStk[TASK_STK_SIZE - 1],
                 0);
    OSStart();                                      //启动多任务管理
}
/*********************** 任务 StartTask ********************************/
void  StartTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                    //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);               //设置时钟频率
    OS_EXIT_CRITICAL();
    OSStatInit();                                   //初始化统计任务
    OSTaskCreate(MyTask,                            //创建任务 MyTask
                 (void * )0,
                 &MyTaskStk[TASK_STK_SIZE - 1],
                 3);
    OSTaskCreate(YouTask,                           //创建任务 YouTask
                 (void * )0,
                 &YouTaskStk[TASK_STK_SIZE - 1],
                 4);
    OSTaskCreate(HerTask,                           //创建任务 HerTask
                 (void * )0,
                 &HerTaskStk[TASK_STK_SIZE - 1],
                 5);
    for ( ; ; )
    {
    //如果按下 ESC 键,则退出 μC/OS - II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
```

```
            {
                PC_DOSReturn();
            }
        }
        OSTimeDlyHMSM(0, 0, 3, 0);                  //等待 3 s
    }
}
/*************************任务 MyTask ********************************/
void  MyTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for ( ; ; )
    {
        OSFlagPend(                                 //请求信号量集
            Sem_F,
            (OS_FLAGS)3,                            //请求第 0 位和第 1 位信号
            OS_FLAG_WAIT_SET_ALL,                   //信号都为 1 表示有效
            0,
            &err
                );
        PC_DispStr(10, ++ y,
                s1,    DISP_BGND_BLACK + DISP_FGND_WHITE );
        OSTimeDlyHMSM(0, 0, 2, 0);                  //等待 2 s
    }
}
/*************************任务 YouTask ********************************/
void  YouTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;

    for ( ; ; )
```

```
    {
        PC_DispStr(10, ++y,
                s2,     DISP_BGND_BLACK + DISP_FGND_WHITE );
        OSTimeDlyHMSM(0, 0, 8, 0);                  //延时 8 s
        OSFlagPost(                                 //发送信号量集
                Sem_F,
                (OS_FLAGS)2,                        //给第 1 位发信号
                OS_FLAG_SET,                        //信号置 1
                &err
                    );
        OSTimeDlyHMSM(0, 0, 2, 0);                  //等待 2 s
    }
}
/*************************任务 HerTask ******************************/
void  HerTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for ( ; ; )
    {
        PC_DispStr(10, ++y,
                s3,     DISP_BGND_BLACK + DISP_FGND_WHITE );
        OSFlagPost(                                 //发送信号量集
                Sem_F,
                (OS_FLAGS)1,                        //给第 0 位发信号
                OS_FLAG_SET,                        //信号置 1
                &err
        );
        OSTimeDlyHMSM(0, 0, 1, 0);                  //等待 1 s
    }
}
/*************************End ***************************************/
```

例 5-1 应用程序的运行结果如图 5-7 所示。

例 5-2 把例 5-1 中任务 MyTask 调用的请求信号量集函数 OSFlagPend()改为无等待请求函数 OSFlagAccept()，运行该程序后，观察运行结果并与例 5-2 应用程序的运行结果进

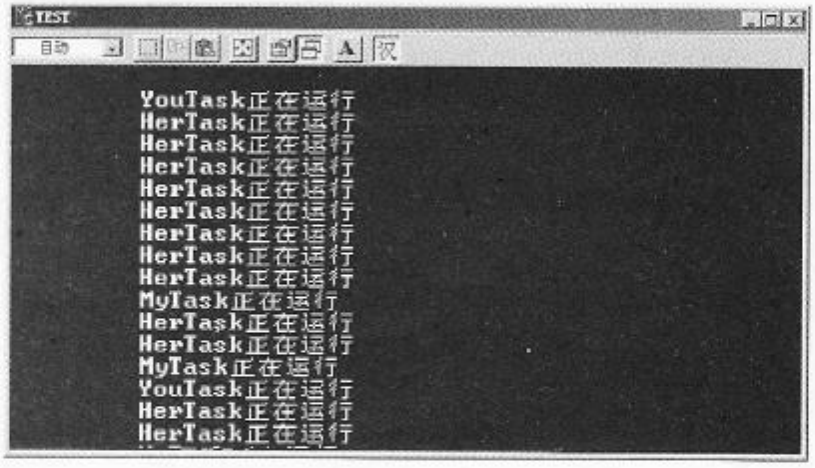


图 5－7　例 5－1 应用程序的运行结果

行比较。

答　任务 MyTask 修改后的代码如下：

```
/*************************任务 MyTask *********************************/
void  MyTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        OSFlagAccept(                       //无等待地请求信号量集
                Sem_F,
                (OS_FLAGS)3,                //请求第 0 位和第 1 位信号
                OS_FLAG_WAIT_SET_ALL,       //信号都为 1 表示有效
                &err
                    );
        PC_DispStr(10, ++y,
                s1,
                DISP_BGND_BLACK + DISP_FGND_WHITE );
        OSTimeDlyHMSM(0, 0, 2, 0);          //等待 2 s
    }
}
```

例 5-2 应用程序的运行结果如图 5-8 所示。

图 5-8 例 5-2 应用程序的运行结果

5.2.4 查询信号量集的状态

调用函数 OSFlagQuery()可以查询一个信号量集的状态。该函数的原型如下：

```
OS_FLAGS  OSFlagQuery (
                        OS_FLAG_GRP  * pgrp,     //待查询的信号量集的指针
                        INT8U  * err;            //错误信息
                      );
```

函数的返回值为被查询信号量集标志组的成员 OSFlagFlags，应用程序可以用它来完成一些更为复杂的控制。

例 5-3 修改例 5-2 应用程序，使任务 MyTask 可以根据信号的不同状态实现不同的功能。

答 修改后三个任务的代码如下：

```
OS_FLAGS Flags;                                  //定义变量

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

```
pdata = pdata;
for (;;)
{
    Flags = OSFlagQuery(                    //查询信号量集的状态
                    Sem_F,
                    &err
                      );
switch(Flags)
{
case 1:
    s1 = "第 0 位信号有效";
    PC_DispStr(10, ++y,
        s1,
        DISP_BGND_BLACK + DISP_FGND_WHITE );
    break;
case 2:
    s1 = "第 1 位信号有效";
    PC_DispStr(10, ++y,
        s1,
        DISP_BGND_BLACK + DISP_FGND_WHITE );
    break;
case 3:
    s1 = "第 0 和第 1 位信号都有效";
    PC_DispStr(10, ++y,
        s1,
        DISP_BGND_BLACK + DISP_FGND_WHITE );
    break;
}
OSFlagPost(                             //发送信号量集
            Sem_F,
            (OS_FLAGS)3,                //给第 0、1 位发信号
            OS_FLAG_CLR,                //信号置 0
            &err
          );

    OSTimeDlyHMSM(0, 0, 2, 0);          //等待 2 s
```

```
    }
}
/*************************任务 YouTask ******************************/
void  YouTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        PC_DispStr(10, ++y,
                   s2,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
        OSTimeDlyHMSM(0, 0, 4, 0);              //延时 4 s
        OSFlagPost(                             //发送信号量集
                   Sem_F,
                   (OS_FLAGS)2,                 //给第 1 位发信号
                   OS_FLAG_SET,                 //信号置 1
                   &err
                   );
        OSTimeDlyHMSM(0, 0, 2, 0);              //等待 2 s
    }
}
/*************************任务 HerTask ******************************/
void  HerTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for (;;)
    {
        PC_DispStr(10, ++y,
                   s3,
                   DISP_BGND_BLACK + DISP_FGND_WHITE );
```

```
        if(y<14)
        {
            OSFlagPost(                         //发送信号量集
                        Sem_F,
                        (OS_FLAGS)1,            //给第 0 位发信号
                        OS_FLAG_SET,            //信号置 1
                        &err
                      );
        }
        OSTimeDlyHMSM(0, 0, 1, 0);              //等待 1 s
    }
}
/*************************End*************************************/
```

例 5－3 应用程序的运行结果如图 5－9 所示。

图 5－9　例 5－3 应用程序的运行结果

5.2.5　删除信号量集

通过调用函数 OSFlagDel()可以删除一个信号量集。该函数的原型如下：

```
OS_FLAG_GRP    * OSFlagDel (
                    OS_FLAG_GRP * pgrp,         //待删除的信号量集指针
                    INT8U opt, INT8U * err;     //错误信息
                        );
```

5.3 小 结

- 实现多个信号量组合的功能定义了一种特殊的数据结构叫做信号量集，该信号量集中的信号都是二值信号量。
- μC/OS－II 是用链表来管理信号量集的。
- 信号量集的输出与输入可以是 AND 关系，也可以是 OR 关系。
- 等待信号量集的任务也是用链表来管理的。

5.4 练习题

1. 叙述信号量集的工作过程。
2. 使用信号量集可以在应用程序中完成哪些工作？

第6章 内存的动态分配

应用程序在运行中为了某种特殊需要，经常需要临时获得一些内存空间，因此作为一个比较完善的操作系统，必须具有动态分配内存的能力。

能否合理、有效地对内存储器进行分配和管理，是衡量一个操作系统品质的指标之一。特别是对于实时操作系统来说，还应该保证系统在动态分配内存时，它的执行时间必须是可确定的。μC/OS-II 改进了 ANSI C 用来动态分配和释放内存的函数 malloc()和 free()，使它们可以对大小固定的内存块进行操作，从而使函数 malloc()和 free()的执行时间成为可确定的，满足了实时操作系统的要求。

本章的主要内容有：

- μC/OS-II 对内存的分区及分块；
- 描述内存块的数据结构——内存控制块；
- 内存控制块与内存分区之间的关系；
- 对内存的操作。

6.1 内存控制块

μC/OS-II 对内存进行两级管理，即把一个大片连续的内存空间分成了若干个分区，每个分区又分成了若干个大小相等的内存块来进行管理。操作系统以分区为单位来管理动态内存，而任务以内存块为单位来获得和释放动态内存。内存分区及内存块的使用情况由表——内存控制块来记录。

本节首先介绍内存分区和分区中的内存块，然后介绍内存控制块。

6.1.1 可动态分配内存的划分

如果应用程序要使用动态内存，则首先要在内存中划分出可以进行动态分配的区域，这个划分出来的区域叫做内存分区，每个分区要包含若干个内存块。μC/OS-II 要求同一个分区中内存块的字节数必须相等，而且每个分区与该分区内存块的数据类型必须相同。

在内存中划分一个内存分区与内存块的方法非常简单，只要定义一个二维数组即可，其中每个一维数组就是一个内存块。例如，定义一个用来存储 INT16U 类型的数据，有 10 个内存块，每个内存块长度为 10 的内存分区的代码如下：

```
INT16U IntMemBuf[10][10];
```

需要注意的是，上面这个定义只是在内存中划分出了分区及内存块的区域，还不是一个真正的可以动态分配的内存区，如图 6-1(a)所示。只有当把内存控制块与分区关联起来之后，系统才能对其进行相应的管理和控制，它才能是一个真正的动态内存区，如图 6-1(b)所示。

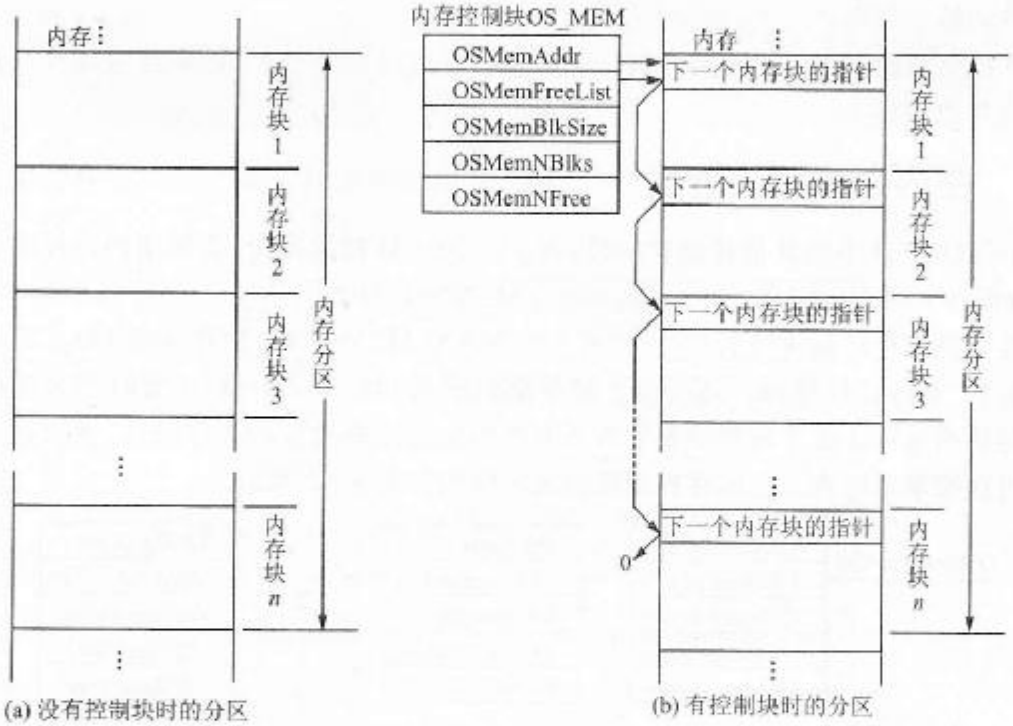


图 6-1 内存控制块与内存分区和内存块的关系

6.1.2 内存控制块 OS_MEM 的结构

为了使系统能够感知和有效地管理内存分区，μC/OS-II 给每个内存分区定义了一个叫做内存控制块(OS_MEM)的数据结构。系统就用这个内存控制块来记录和跟踪每一个内存分区的状态。内存控制块的结构如下：

```
typedef struct {
    void    *OSMemAddr;        //内存分区的指针
    void    *OSMemFreeList;    //内存控制块链表的指针
```

```
    INT32U  OSMemBlkSize;     //内存块的长度
    INT32U  OSMemNBlks;       //分区内内存块的数目
    INT32U  OSMemNFree;       //分区内当前可分配的内存块的数目
} OS_MEM;
```

当应用程序调用函数 OSMemCreate()建立一个内存分区之后，内存控制块与内存分区和内存块之间的关系如图 6－1 所示。

从图 6－1(b)中可知，内存控制块的内存分区指针 OSMemAddr 指向了内存分区，内存分区中的各个内存块又组成了一个单向链表，内存控制块的链表指针 OSMemFreeList 就指向了这个单向链表的头。

内存控制块的其他 3 个变量分别记录了分区中内存块的长度、总数目以及现在还没有被分配的内存块数目。

6.1.3 空内存控制块链表

与 μC/OS－II 中的其他控制块一样，在 μC/OS－II 初始化时，会调用内存控制块的初始化函数 OS_MemInit()定义并初始化一个空内存控制块链表。

在这个空内存控制块链表中，一共有 OS_MAX_MEM_PART(在文件 OS_CFG.H 中定义的常数)个空内存控制块。这时内存控制块的成员 OSMemFreeList 暂时作为指向下一个内存控制块的指针。由于这时链表中的所有控制块还没有对应的内存分区，所以这个链表就叫做空内存控制块链表。空内存控制块链表的结构如图 6－2 所示。

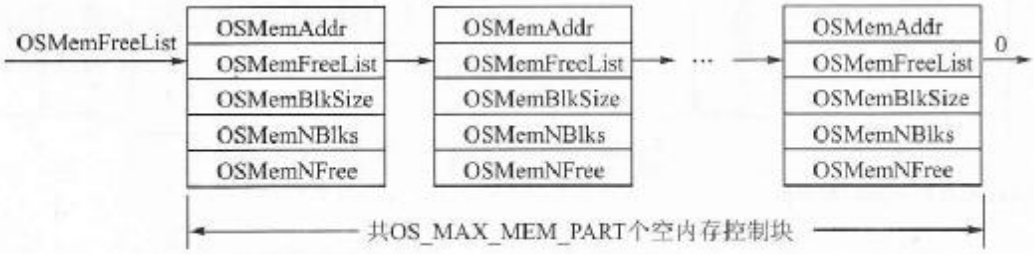


图 6－2 空内存控制块链表

每当应用程序需要创建一个内存分区时，系统就会从空内存控制块链表中摘取一个控制块，而把链表的头指针 OSMemFreeList 指向下一个空内存控制块；而每当应用程序释放一个内存分区时，则会把该分区对应的内存控制块归还给空内存控制块链表。

6.2 动态内存的管理

μC/OS－II 用于动态内存管理的函数有：创建动态内存分区函数 OSMemCreate()、请求获得内存块函数 OSMemGet()、释放内存块函数 OSMemPut()和查询动态内存分区状态函数

OSMemQuery()等函数。

6.2.1 创建动态内存分区

在用6.1.1小节介绍的方法划分要使用的分区和内存块之后，应用程序可通过调用函数OSMemCreate()来建立一个内存分区。该函数的原型如下：

```
OS_MEM * OSMemCreate(
                void * addr,        //内存分区的起始地址
                INT32U nblks,       //分区中内存块的数目
                INT32U blksize,     //每个内存块的字节数
                INT8U * err         //错误信息
              );
```

函数OSMemCreate()的流程图如图6-3所示。

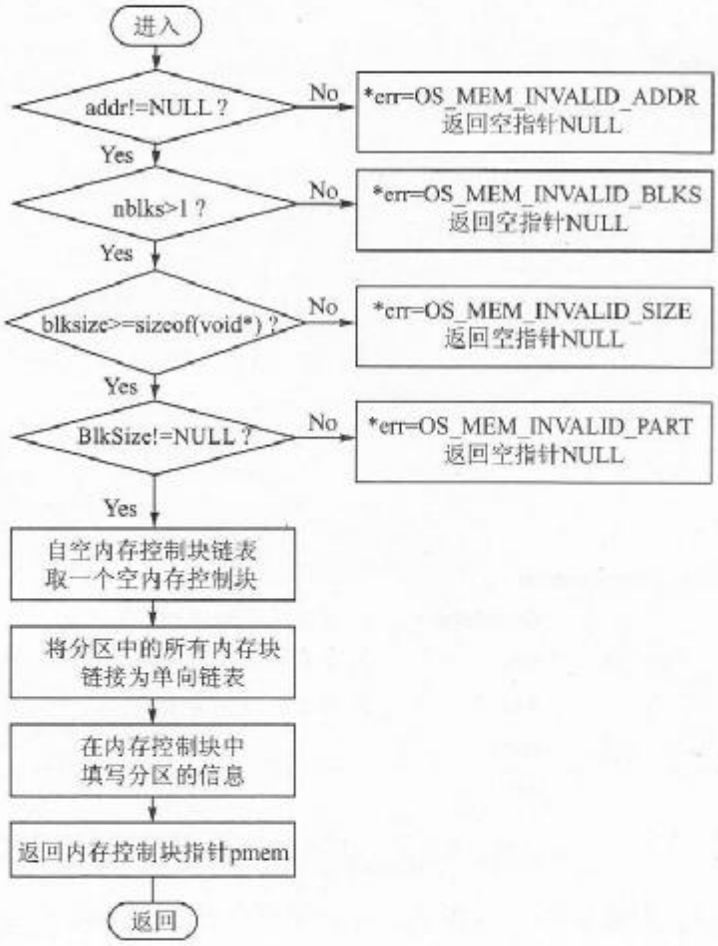


图6-3 函数OSMemCreate()的流程图

从图 6－3 中可以看到，函数首先对创建一个内存分区的基本条件做一系列的判断，然后才定义内存分区。如果其中有一个条件不满足，就意味着函数调用失败，于是函数就返回一个 NULL 指针，并把相应的错误信息传递到 err 中。

在这一系列的条件判断中有两个问题值得注意：一是分区的内存块至少有两块；二是每个内存块的空间得至少能存放一个指针，因为要在内存块中建立一个用于把分区内的内存块链接为一个链表的指针。

在接下来的工作中，函数 OSMemCreate()主要做了 3 项工作：首先自空内存控制块链表取一个控制块；然后把分区内的内存块链接成链表建立内存分区；最后再把刚建立的内存分区的相关信息填入内存控制块，并返回与这个刚建立的内存分区相关联的内存控制块的指针，以作为其他内存管理函数调用时的参数。

例 6－1 建立一个含有 50 个内存块并且每块的长度为 64 字节的内存分区。试写出主要代码。

```
//定义全局变量
OS_MEM *CommTxBuffer;                    //定义内存分区指针
INT8U    CommTxPart[50][64];             //定义分区和内存块
INT8U    err;

//在主函数的适当位置建立内存分区

void main(void)
{
    INT8U err;
        ...
    OSInit();
        ...
    CommTxBuffer = OSMemCreate(
                        CommTxPart,  //内存分区的首地址
                        50,          //分区内内存块的数目
                        64,          //每个内存块的长度
                        &err
                        );
        ...
    OSStart();
}
```

6.2.2 请求获得一个内存块

在应用程序需要一个内存块时，应用程序可以通过调用函数 OSMemGet()向某内存分区请求获得一个内存块。该函数的原型如下：

```
void   * OSMemGet (
                OS_MEM * pmem,     //内存分区的指针
                INT8U * err        //错误信息
                );
```

函数 OSMemGet()中的参数 pmem 是应用程序希望获得的内存块所在内存分区所对应的内存控制块的指针。函数调用成功后，其返回值为所请求的内存块的指针。

函数 OSMemGet()的流程图如图 6－4 所示。可以看到，函数 OSMemGet()在判断了应用程序传递来的内存控制块的指针为非 NULL 及内存分区尚存在未被分配的内存块后，就将内存块链表的第一个块的指针 OSMemFreeList 赋给了指针 pblk；然后就重新调整内存块链表，并使指针 OSMemFreeList 指向新的链表头；最后返回指针 pblk。

当函数调用失败时，函数返回空指针 NULL。

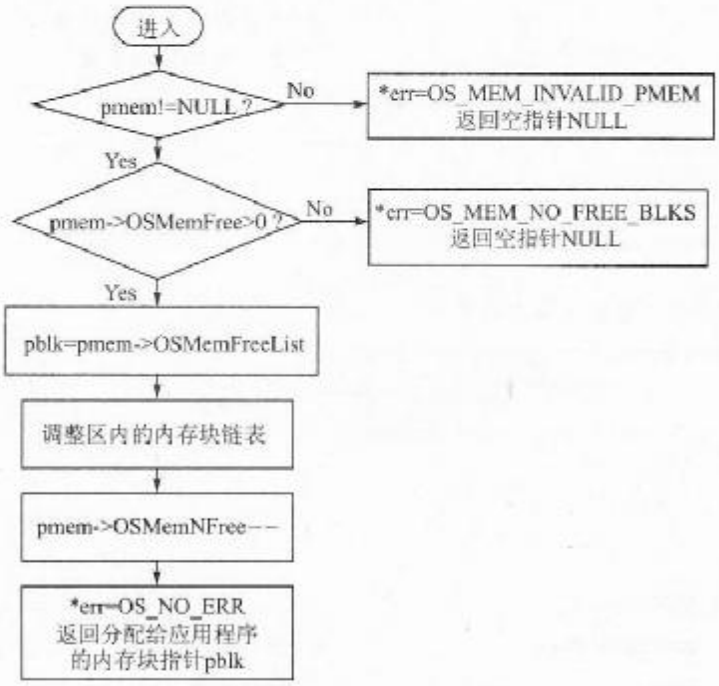


图 6－4　函数 OSMemGet()的流程图

例 6－2　在例 6－1 的基础上写出任务 MyTask 请求一个内存块的代码。

答　应用程序主要代码如下：

```
//定义全局变量

OS_MEM *CommTxBuffer;                          //定义内存分区指针
INT8U    CommTxPart[50][64];                   //定义分区和内存块
INT8U    err;
INT8U    *BlkPtr;                              //定义内存块指针

//在主函数的适当位置建立内存分区

void main(void)
{
    INT8U err;
        ...
    OSInit();
        ...
    CommTxBuffer = OSMemCreate(
                            CommTxPart,        //内存分区的首地址
                            50,                //分区内内存块的数目
                            64,                //每个内存块的长度
                            &err
                            );
        ...
    OSStart();
}
//在任务 Mytask 的合适位置请求内存块
void MyTask((void *)pdata)
{
    ......
    for( ; ; )
    {
        ......
        BlkPtr= OSMemGet (
                CommTxBuffer,                  //内存分区的指针
                &err                           //错误信息
                );
        ......
    }
}
```

需要注意的是，应用程序在调用函数 OSMemGet()时，应该事先知道该分区中内存块的大小，并且在使用该内存块时不能超其长度，否则会引起灾难性的后果。当应用程序不再需要该内存块时，必须及时将其释放。

6.2.3 释放一个内存块

当应用程序不再使用一个内存块时，必须及时将其释放。应用程序通过调用函数 OSMemPut() 来释放一个内存块。该函数的原型如下：

```
INT8U  OSMemPut (
            OS_MEM   * pmem,     //内存块所属内存分区的指针
            void * pblk          //待释放内存块的指针
            );
```

函数 OSMemPut()的返回值是错误信息。如果函数调用成功，将返回信息 OS_NO_ERR；否则将根据具体错误返回 OS_MEM_INVALID_PMEM(控制块指针为空指针)、OS_INVALID_PBLK(释放内存块指针为空指针)和 OS_MEM_FULL(分区已满)。

函数 OSMemPut()的流程图如图 6-5 所示。

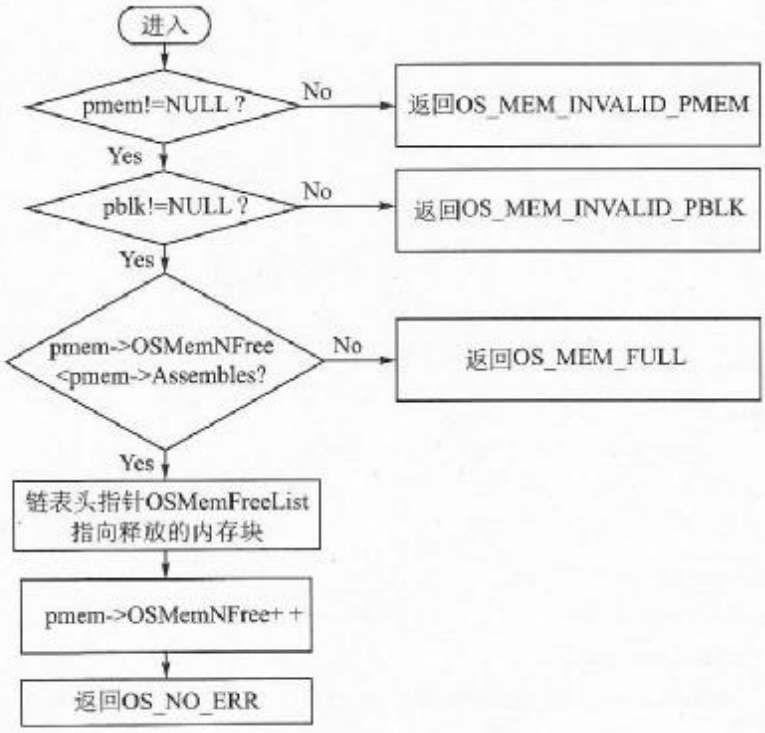


图 6-5 函数 OSMemPut()的流程图

需要注意的是，在调用函数 OSMemPut ()的一个内存块时，一定要确保把该内存块释放到它原来所属的内存分区中，否则会引起灾难性的后果。

例如，对例 6－2 中任务 MyTask 使用的内存块，在任务不再使用它时，应该用下面的代码来释放它：

```
OSMemPut (
            CommTxBuffer,                          //内存块所属内存分区的指针
            BlkPtr                                 //待释放内存块的指针
            );
```

例 6－3 设计一个含有一个任务的应用程序，该任务负责打印两个起始显示位置不同的相同字符串。要求在任务中申请一个内存块，并把存放字符串显示起始位置的数据变量定义在该内存块中。

答 应用程序代码如下：

```
/*************************Test*****************************************/
#include "includes.h"
#define  TASK_STK_SIZE    512
OS_STK   StartTaskStk[TASK_STK_SIZE];
OS_STK   MyTaskStk[TASK_STK_SIZE];
OS_MEM   *IntBuffer;                                //定义内存控制块指针
INT8U    IntPart[50][64];                           //划分分区及内存块
INT8U    *IntBlkPtr;                                //定义指针
INT16S   key;
char *s1 = "MyTask 正在运行";
INT8U err;
INT8U y = 0;
void  StartTask(void *data);
void  MyTask(void *data);                           //声明任务
/*************************主函数*********************************/
void  main (void)
{
    OSInit();
    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);
    IntBuffer = OSMemCreate(IntPart,50,64,&err);    //创建动态内存分区
    OSTaskCreate(StartTask,
                 (void*)0,
                 &StartTaskStk[TASK_STK_SIZE - 1],
```

```
                0);
    OSStart();
}
/************************ 任务 StartTask ********************************/
void  StartTask (void * pdata)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(OS_TICKS_PER_SEC);
    OS_EXIT_CRITICAL();
    OSStatInit();
    OSTaskCreate(MyTask,                              //创建任务 MyTask
                 (void * )0,
                 &MyTaskStk[TASK_STK_SIZE - 1],
                 3);
    for ( ; ; )
    {
    //如果按下 ESC 键,则退出 μC/OS - II
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B)
            {
                PC_DOSReturn();
            }
        }
        OSTimeDlyHMSM(0, 0, 3, 0);                    //等待 3 s
    }
}
/************************ 任务 MyTask ********************************/
void  MyTask (void * pdata)
{
# if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
# endif
```

```
    pdata = pdata;
    for (;;)
    {
        IntBlkPtr = OSMemGet (                          //请求内存块
                              IntBuffer,                //内存分区的指针
                              &err                      //错误信息
                              );
        * IntBlkPtr = 1 ;
        PC_DispStr( * IntBlkPtr * 10, ++ y,
            s1,
            DISP_BGND_BLACK + DISP_FGND_WHITE );
        * ++ IntBlkPtr = 2 ;
        PC_DispStr( * IntBlkPtr * 10, ++ y,
            s1,
            DISP_BGND_BLACK + DISP_FGND_WHITE );
        IntBlkPtr -- ;
        OSMemPut (                                      //释放内存块
                  IntBuffer,                            //内存分区的指针
                  IntBlkPtr                             //待释放内存块的指针
                  );
        OSTimeDlyHMSM(0, 0, 1, 0);                      //等待 1 s
    }
}
/*************************End*************************************/
```

例 6－3 应用程序的运行结果如图 6－6 所示。

图 6－6　例 6－3 应用程序的运行结果

6.2.4 查询一个内存分区的状态

应用程序可以通过调用函数 OSMemQuery()来查询一个分区目前的状态信息。该函数的原型如下：

```
INT8U  OSMemQuery (
            OS_MEM * pmem,          //待查询的内存控制块的指针
            OS_MEM_DATA * pdata     //存放分区状态信息的结构的指针
                )
```

函数 OSMemQuery()中的参数 pdata 是一个 OS_MEM_DATA 类型的结构。该结构的定义如下：

```
typedef struct {
    void    * OSAddr;           //内存分区的指针
    void    * OSFreeList;       //分区内内存块链表的头指针
    INT32U  OSBlkSize;          //内存块的长度
    INT32U  OSNBlks;            //分区内内存块的数目
    INT32U  OSNFree;            //分区内空闲内存块的数目
    INT32U  OSNUsed;            //已被分配的内存块数目
} OS_MEM_DATA;
```

调用函数 OSMemQuery()后，内存分区的状态信息就存放在这个结构中。

调用函数 OSMemQuery()成功后，返回常数 OS_NO_ERR。

例 6-4 设计一个有 3 个任务的应用程序，这 3 个任务分别是 MyTask、YouTask 和 HerTask。在应用程序中创建一个动态内存分区，该分区有 8 个内存块，每个内存块的长度为 6 字节。应用程序的任务 YouTask 和 HerTask 都在任务运行后请求一个内存块，随后就释放它；任务 MyTask 也在任务运行后请求一个内存块，但是要在任务 MyTask 运行 6 次后，才释放它所申请的内存块。

为了了解内存分区变化的情况，编写代码来观察分区头指针和已被使用内存块的个数。

答 应用程序代码如下：

```
/*************************Test ****************************************/
#include "includes.h"
#define  TASK_STK_SIZE    512
OS_STK   StartTaskStk[TASK_STK_SIZE];
OS_STK   MyTaskStk[TASK_STK_SIZE];
OS_STK   YouTaskStk[TASK_STK_SIZE];
```

```
OS_STK   HerTaskStk[TASK_STK_SIZE];
 INT16S   key;
char * s;
char * s1 = "MyTask";
char * s2 = "YouTask";
char * s3 = "HerTask";
INT8U err;
INT8U y = 0;
INT8U Times = 0;
OS_MEM * IntBuffer ;                    //定义内存控制块指针
INT8U    IntPart[8][6];                 //划分分区及内存块
INT8U    * IntBlkPtr;
OS_MEM_DATA MemInfo ;
void  StartTask(void * data);
void  MyTask(void * data);
void  YouTask(void * data);
void  HerTask(void * data);
/************************主函数 **********************************/
void  main (void)
{
    OSInit();
    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);
    IntBuffer = OSMemCreate(IntPart,8,6,&err);
    OSTaskCreate(StartTask,(void * )0,
                &StartTaskStk[TASK_STK_SIZE - 1],    0);
    OSStart();
}
/************************任务 StartTask ********************************/
void  StartTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(OS_TICKS_PER_SEC);
```

```
    OS_EXIT_CRITICAL();
    OSStatInit();
    OSTaskCreate(MyTask,    (void * )0,
                &MyTaskStk[TASK_STK_SIZE - 1],3);
    OSTaskCreate(YouTask,(void * )0,&YouTaskStk[TASK_STK_SIZE - 1],4);
    OSTaskCreate(HerTask,(void * )0,&HerTaskStk[TASK_STK_SIZE - 1],5);
    for ( ; ; )
    {
    //如果按下 ESC 键,则退出 μC/OS - II
    if (PC_GetKey(&key) == TRUE)
    {
        if (key == 0x1B)
        {
                PC_DOSReturn();
        }
    }
        OSTimeDlyHMSM(0, 0, 3, 0);
    }
}
/*************************任务 MyTask ********************************/
void  MyTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for ( ; ; )
    {
        PC_DispStr(10, ++y,s1,DISP_BGND_BLACK + DISP_FGND_WHITE );
        IntBlkPtr = OSMemGet (                      //请求内存块
                    IntBuffer,                      //内存分区的指针
                    &err                            //错误信息
                    );
        OSMemQuery (                                //查询内存控制块信息
                IntBuffer,                          //待查询内存控制块指针
                    &MemInfo
                    );
        sprintf(s," %0x",MemInfo.OSFreeList);       //显示头指针
```

```
        PC_DispStr(30,y,s,DISP_BGND_BLACK + DISP_FGND_WHITE );
        sprintf(s," %d",MemInfo.OSNUsed);              //显示已用内存块数目
        PC_DispStr(40,y,s,DISP_BGND_BLACK + DISP_FGND_WHITE );
        if(Times>4)
            {
                OSMemPut (                             //释放内存块
                        IntBuffer,                     //内存分区的指针
                        IntBlkPtr                      //待释放内存块的指针
                        );
            }
        Times ++ ;
        OSTimeDlyHMSM(0, 0, 1, 0);                     //等待 1 s
    }
}
/*************************任务 YouTask ******************************/
void  YouTask (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;

    for ( ; ; )
    {
        PC_DispStr(10, ++ y,s2,DISP_BGND_BLACK + DISP_FGND_WHITE );
        IntBlkPtr = OSMemGet (                         //请求内存块
                            IntBuffer,                 //内存分区的指针
                            &err                       //错误信息
                            );
        OSMemQuery (                                   //查询内存控制块信息
                            IntBuffer,                 //待查询的内存控制块指针
                            &MemInfo
                    );
        sprintf(s," %0x",MemInfo.OSFreeList);          //显示头指针

        PC_DispStr(30,y,s,DISP_BGND_BLACK + DISP_FGND_WHITE );
            sprintf(s," %d",MemInfo.OSNUsed);          //显示已用内存块数目
        PC_DispStr(40,y,s,DISP_BGND_BLACK + DISP_FGND_WHITE );
```

```
            OSMemPut (                              //释放内存块
                    IntBuffer,                      //内存分区的指针
                    IntBlkPtr                       //待释放内存块的指针
                    );
        OSTimeDlyHMSM(0, 0, 2, 0);                  //等待 2 s
    }
}
/*************************任务 HerTask ******************************/
void  HerTask (void * pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    pdata = pdata;
    for ( ; ; )
    {
        PC_DispStr(10, ++ y,s3,DISP_BGND_BLACK + DISP_FGND_WHITE );
        IntBlkPtr = OSMemGet (                      //请求内存块
                        IntBuffer,                  //内存分区的指针
                        &err                        //错误信息
                        );
        OSMemQuery (                                //查询内存控制块信息
                    IntBuffer,                      //待查询的内存控制块指针
                    &MemInfo
                );
            sprintf(s," % 0x",MemInfo.OSFreeList);  //显示头指针

    PC_DispStr(30,y,s,DISP_BGND_BLACK + DISP_FGND_WHITE );
            sprintf(s," % d",MemInfo.OSNUsed);      //显示已用内存块数目
        PC_DispStr(40,y,s,DISP_BGND_BLACK + DISP_FGND_WHITE );
            OSMemPut (                              //释放内存块
                    IntBuffer,                      //内存分区的指针
                    IntBlkPtr                       //待释放内存块的指针
                    );
        OSTimeDlyHMSM(0, 0, 1, 0);                  //等待 1 s
    }
}
/*************************End ************************************/
```

例 6－4 应用程序的运行结果如图 6－7 所示。

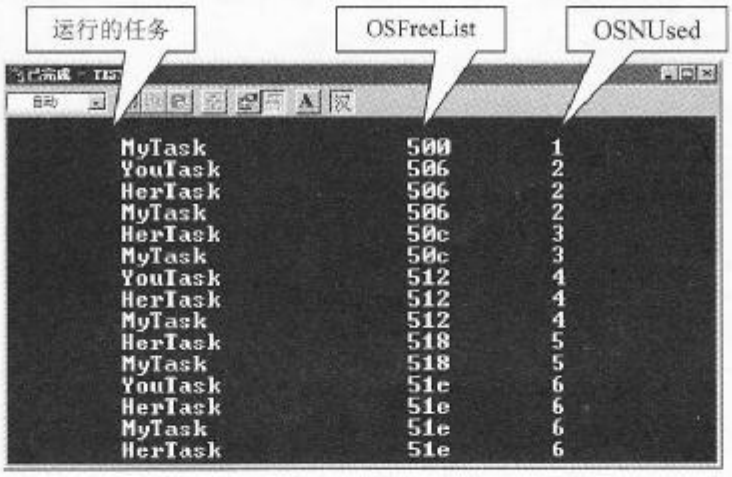


图 6－7　例 6－4 应用程序的运行结果

6.3　小　结

- 通过定义一个二维数组就可在内存中划分出一个内存分区，并保证分区中的所有内存块大小相等。
- 系统是通过与内存分区相关联的内存控制块来实现对内存分区的管理的。
- 划分及创建内存分区是应用程序根据需要由应用程序来负责的，而系统只提供了可供任务调用的相关函数。
- 在 μC/OS－II 中，应用程序在释放使用的动态内存时的安全性是由用户应用程序来维护的。

6.4　练习题

1. 什么叫内存分区？什么叫内存块？内存分区与内存块之间有什么关系？
2. 在同一个内存分区中内存块是按什么方式组织起来的？
3. 什么叫内存控制块？内存控制块中都记录了一些什么信息？
4. 设计一个有两个任务的应用程序，其中一个任务用来进行两个数的加法运算，另一个任务则用来显示结果，要求加法运算的和存放在动态内存区中。

第7章 在51单片机上移植μC/OS-II

操作系统是一种与硬件(处理器)相关的软件,根据某一种处理器来设计的操作系统一般是不能在其他种类的处理器上运行的。如果要在其他处理器上运行该操作系统,则必须对这个操作系统做相应的改造,即所谓操作系统的移植。

本章的主要内容有:

- 移植μC/OS-II时的一般性问题;
- 在51单片机上移植μC/OS-II时的堆栈设计;
- 在51单片机上基于μC/OS-II操作系统应用程序的设计举例。

7.1 μC/OS-II移植的一般性问题

一个软件的移植工作总是与用户所选择的处理器硬件结构相关的,因此把一个软件往不同的处理器上移植的方法也不尽相同,但一些一般性问题还是有其共同特点的。

7.1.1 可重入函数

在多任务操作系统环境中,应用程序的各个任务是并发运行的,这些并发运行的任务经常会出现调用同一个函数的情况,那么就可能出现任务之间通过这个函数而产生意外的耦合,从而互相干扰。例如,有一个任务A和任务B都要调用的函数Swap(),而该函数又使用了全局变量temp。

```
int temp;
void Swap(int * x,int * y)
{
    temp = *x;
    *x = *y;
    *y = temp;
}
```

于是,如果当任务A调用Swap()函数期间,系统发生了任务切换而使任务B也调用了函

数 Swap()，那么任务 B 将要改变全局变量 temp 的值，使任务 A 传递给全局变量 temp 的值丢失而出现错误。

显然，是不允许出现上述现象的，所以在多任务操作系统中，系统提供的函数应该能允许同时被多个任务所调用，而不会通过函数中变量的耦合引起任务之间的相互干扰。这样的函数叫做可重入函数。

一般来说，一个可重入函数应该在函数中只使用局部变量，因为函数的局部变量存储在任务的堆栈中，所以可保证不同的任务在调用同一个函数时不会发生冲突。如果函数一定要使用全局变量，则需要对使用的全局变量做必要的保护。

另外，为了能够产生可重入函数，用户使用的处理器的 C 编译器应该具有产生可重入代码的能力。

7.1.2 时钟节拍的产生

通过前面的学习已经知道，μC/OS－II 是通过硬件中断来实现系统时钟，并在时钟中断服务程序中来处理与时间相关的问题的。因此，用户所选用的处理器必须具有响应中断的能力，并同时应该具有开中断与关中断的指令。

一般情况下应该使用硬件定时器来作为时钟中断源，这个定时器可以是与微处理器集成在一个芯片上的，也可以是分立的。定时中断的频率以 10～100 Hz 为宜。

7.1.3 任务堆栈的设计

要使 μC/OS－II 能够正常运行，用户选用的处理器必须可支持一定数量的硬件堆栈，并且应该有对堆栈指针进行读/写操作的指令。

1. 堆栈的格式

众所周知，不同的处理器支持堆栈的增长方向也不同，因此在对 μC/OS－II 的移植时，定要正确地定义堆栈的格式。

2. 系统堆栈与任务堆栈的关系

有些处理器对于堆栈的设置有特殊的要求，即要求堆栈必须设置在一个特定的区域。例如，51 系列单片机要求堆栈必须设置在片内 RAM 中。由于这种处理器芯片内的 RAM 极其有限，不可能把应用程序中所有任务的任务堆栈都设置在片内 RAM 中，所以就只能把应用程序中各个任务堆栈的内容存放在片外 RAM 中，而只在片内 RAM 中设置一个公用的堆栈。

如果把在片外 RAM 中用来存放任务堆栈内容的区域叫做任务堆栈映像，而片内 RAM 中的公用堆栈叫做系统堆栈，那么，当系统运行某个任务时，就要把该任务的堆栈映像复制到系统堆栈中；而在中止这个任务时，再把系统堆栈中的内容复制回任务堆栈映像中。系统堆栈与任务堆栈映像之间的关系如图 7－1 所示。

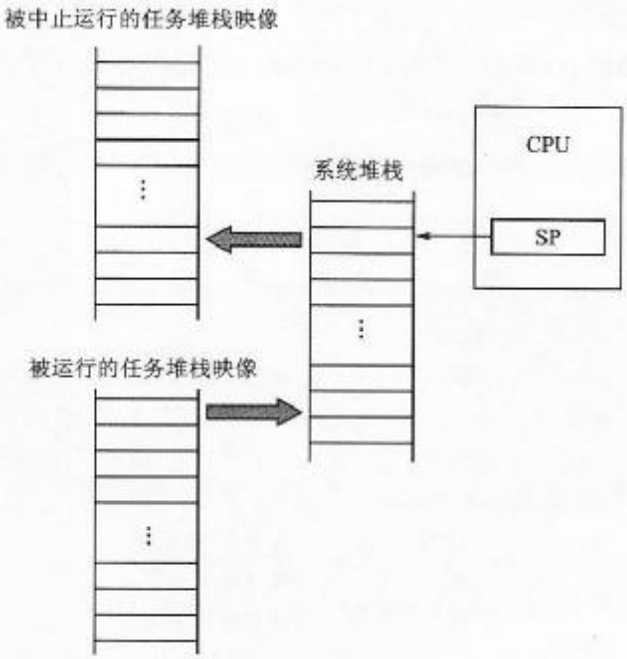


图 7-1 系统堆栈与任务堆栈映像之间的关系

7.2 在 51 系列单片机上移植 μC/OS - II

为了提高可移植性，μC/OS - II 的绝大部分代码都是用 C 语言编写的。在一般情况下，这部分代码不需要修改就可以使用，因此它的移植工作主要与 4 个文件相关：汇编文件(OS_CPU_A. ASM)、处理器相关 C 语言文件(OS_CPU. H、OS_CPU_C. C)和配置文件(OS_CFG. H)。

7.2.1 改写文件 OS_CPU. H

1. 堆栈的增长方向

51 单片机的堆栈是从低地址向高地址增长(由下往上增长)的，所以定义堆栈增长方向的常数 OS_STK_GROWTH 应该为 1，即

```
#define  OS_STK_GROWTH     1
```

2. 定义临界段的宏

设置临界区的两个宏分别直接使用 51 单片机的开中断和关中断指令来实现。

```
#define  OS_ENTER_CRITICAL()  EA = 0
#define  OS_EXIT_CRITICAL()   EA = 1
```

3. 定义任务切换宏

任务切换是由汇编语言编写的函数 OSCtxSw()实现的。任务切换宏 OS_TASK_SW()封装了对任务切换函数 OSCtxSw()的调用。

```
#define  OS_TASK_SW()      OSCtxSw()
```

4. 定义数据类型

```
typedef unsigned    char    BOOLEAN;      //不要 bit 定义
typedef unsigned    char    INT8U;        //无符号 8 位数
typedef signed      char    INT8S;        //有符号 8 位数
typedef unsigned    int     INT16U;       //无符号 16 位数
typedef signed      int     INT16S;       //有符号 16 位数
typedef unsigned    long    INT32U;       //无符号 32 位数
typedef signed      long    INT32S;       //有符号 32 位数
typedef float               FP32;         //单精度浮点数
typedef double              FP64;         //双精度浮点数

typedef unsigned    char    OS_STK;       //栈单元宽度为 8 位

#define BYTE                INT8S         //兼容以前版本的数据类型
#define UBYTE               INT8U         //μC/OS－II 可以不用这些数据类型
#define WORD                INT16S
#define UWORD               INT16U
#define LONG                INT32S
#define ULONG               INT32U
```

7.2.2 任务堆栈的设计

前面讲过，由于 51 单片机的堆栈只能设置在内部 RAM 中，而由于内部 RAM 空间的限制，就只能在外部 RAM 中设置任务堆栈映像了，于是在任务切换时要进行任务堆栈映像和系统堆栈之间的复制操作。为了方便复制工作，应该在堆栈中存有堆栈长度的数据。

51 单片机的外部 RAM 中设置任务堆栈映像，任务堆栈映像的最低地址空间用来存放用户堆栈的长度，并使该任务的 TCB 中的 OSTCBStkPtr 指针变量指向该地址。

51 单片机的堆栈指针 SP 总是先加 1 再存数据，因此，SP 初始时指向系统堆栈起始地址(OSStack)减 1 处(OSStkStart)。很明显，系统堆栈长度为 SP－OSStkStart。

在任务切换时，应先把当前任务在系统堆栈的内容复制保存到该任务堆栈映像中。也就是计算 SP－OSStkStart，得出堆栈长度，并将其写入任务堆栈映像最低地址空间中；然后以其

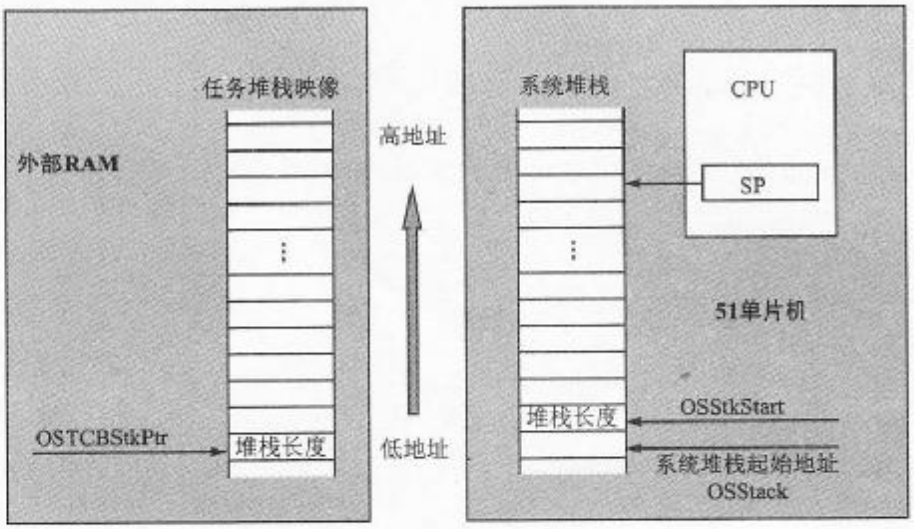


图 7－2　在 51 单片机上移植 μC/OS－II 时，
系统堆栈与任务堆栈映像之间的关系

为长度，以任务堆栈映像最低地址为目标起址，以 OSStkStart 为源起址，由系统堆栈向任务堆栈映像复制数据。

然后把就绪最高优先级任务堆栈映像中的内容复制到系统堆栈。方法是，自任务控制块中获得该任务堆栈映像的最低地址，从中取出堆栈长度，以最高优先级任务堆栈映像最低地址为源起址，以系统堆栈起址 OSStkStart 为目标起址，由任务堆栈映像向系统堆栈复制数据。

任务切换函数 OSCtxSw()及在任务切换时对系统堆栈和任务堆栈映像的操作都是用汇编语言编写的，其代码如下(注意，函数 OSCtxSw()的入口在标号 OSCtxSw 处)：

```
;-----------------------------------------------------------------------
        RSEG ? PR? OSStartHighRdy? OS_CPU_A
OSStartHighRdy:
        USING 0
        LCALL _? OSTaskSwHook

OSCtxSw_in:
        ;OSTCBCur => DPTR   获得当前 TCB 指针
        MOV  R0,#LOW (OSTCBCur)
        INC  R0
        MOV  DPH,@R0
        INC  R0
        MOV  DPL,@R0
```

```
        ;任务堆栈映像指针 => DPTR
        INC  DPTR
        MOVX A,@DPTR
        MOV  R0,A
        INC  DPTR
        MOVX A,@DPTR
        MOV  R1,A
        MOV  DPH,R0
        MOV  DPL,R1

        ;任务堆栈映像长度 * UserStkPtr => R5
        MOVX A,@DPTR
        MOV  R5,A                ;R5 = 任务堆栈映像长度

        ;把待运行任务堆栈映像的内容复制到系统堆栈
        MOV  R0,#OSStkStart
restore_stack:
        INC  DPTR
        INC  R0
        MOVX A,@DPTR
        MOV  @R0,A
        DJNZ R5,restore_stack

        ;恢复系统堆栈指针 SP
        MOV  SP,R0

        ;恢复任务堆栈映像指针?C_XBP
        INC  DPTR
        MOVX A,@DPTR
        MOV  ?C_XBP,A
        INC  DPTR
        MOVX A,@DPTR
        MOV  ?C_XBP+1,A

        ;OSRunning = TRUE
        MOV  R0,#LOW (OSRunning)
```

```
        MOV   @R0,#01

        POPALL        ;把系统堆栈内容弹入 CPU
        SETB EA       ;开中断
        RETI          ;利用中断返回指令,使系统切换至待运行任务
;-----------------------------------------------------------------------
        RSEG ?PR?OSCtxSw?OS_CPU_A
;任务切换函数入口
OSCtxSw:
        PUSHALL       ;把被中止运行任务的 CPU 内容压栈

OSIntCtxSw_in:
        ;获得堆栈长度和起址
        MOV  A,SP
        CLR  C
        SUBB A,#OSStkStart
        MOV  R5,A                ;获得堆栈长度

        ;OSTCBCur => DPTR      ;获得当前任务 TCB 指针
        MOV  R0,#LOW (OSTCBCur) ;
        INC  R0
        MOV  DPH,@R0
        INC  R0
        MOV  DPL,@R0

        ;获得任务堆栈映像指针
        INC  DPTR
        MOVX A,@DPTR
        MOV  R0,A
        INC  DPTR
        MOVX A,@DPTR
        MOV  R1,A
        MOV  DPH,R0
        MOV  DPL,R1

        ;保存堆栈长度
        MOV  A,R5
```

```
        MOVX @DPTR,A

        MOV  R0,#OSStkStart   ;获得堆栈起址
;把系统堆栈内容复制到任务堆栈映像
save_stack:
        INC  DPTR
        INC  R0
        MOV  A,@R0
        MOVX @DPTR,A
        DJNZ R5,save_stack

        ;保存任务堆栈映像指针?C_XBP
        INC  DPTR
        MOV  A,?C_XBP
        MOVX @DPTR,A
        INC  DPTR
        MOV  A,?C_XBP+1
        MOVX @DPTR,A

        ;调用用户程序
        LCALL _?OSTaskSwHook

        ;获得最高级就绪任务的 TCB
        MOV  R0,#OSTCBCur
        MOV  R1,#OSTCBHighRdy
        MOV  A,@R1
        MOV  @R0,A
        INC  R0
        INC  R1
        MOV  A,@R1
        MOV  @R0,A
        INC  R0
        INC  R1
        MOV  A,@R1
        MOV  @R0,A

        MOV  R0,#OSPrioCur
```

```
        MOV  R1,#OSPrioHighRdy
        MOV  A,@R1
        MOV  @R0,A

        LJMP OSCtxSw_in
;-----------------------------------------------------------------------
        RSEG ?PR?OSIntCtxSw?OS_CPU_A

OSIntCtxSw:

        ;调整 SP 指针去掉在调用 OSIntExit(),OSIntCtxSw()
        ;过程中压入堆栈的多余内容 SP = SP - 4
        MOV  A,SP
        CLR  C
        SUBB A,#4
        MOV  SP,A
        LJMP OSIntCtxSw_in
```

其中压栈和出栈代码段代码如下：

```
;定义压栈出栈宏-----------------------------------------------------------
PUSHALL    MACRO
        PUSH PSW
        PUSH ACC
        PUSH B
        PUSH DPL
        PUSH DPH
        MOV  A,R0     ;R0～R7 入栈
        PUSH ACC
        MOV  A,R1
        PUSH ACC
        MOV  A,R2
        PUSH ACC
        MOV  A,R3
        PUSH ACC
        MOV  A,R4
        PUSH ACC
        MOV  A,R5
```

```
        PUSH ACC
        MOV  A,R6
        PUSH ACC
        MOV  A,R7
        PUSH ACC
        ;PUSH SP      ;不必保存 SP,任务切换时由相应程序调整
        ENDM
;定义出栈宏----------------------------------------------------------------
POPALL    MACRO
        ;POP  ACC     ;不必保存 SP,任务切换时由相应程序调整
        POP  ACC      ;R0～R7 出栈
        MOV  R7,A
        POP  ACC
        MOV  R6,A
        POP  ACC
        MOV  R5,A
        POP  ACC
        MOV  R4,A
        POP  ACC
        MOV  R3,A
        POP  ACC
        MOV  R2,A
        POP  ACC
        MOV  R1,A
        POP  ACC
        MOV  R0,A
        POP  DPH
        POP  DPL
        POP  B
        POP  ACC
        POP  PSW
        ENDM
;----------------------------------------------------------------------
```

7.2.3 文件 OS_CPU_C.C 的改写

在文件 OS_CPU_C.C 中主要应改写任务堆栈初始化函数 OSTaskStkInit ()。由于要使用单片机上的定时器为系统设置时钟中断，因此还添加对 51 单片机定时器的初始化程序。

1. 改写任务堆栈初始化函数

```
void * OSTaskStkInit (     void ( * task)(void * pd),
                        void * ppdata, void * ptos,
                        INT16U opt
                        )reentrant
{
    OS_STK * stk;

    ppdata = ppdata;
    opt    = opt;                          //opt 没被用到,保留此语句防止告警产生
    stk    = (OS_STK * )ptos;              //用户堆栈最低有效地址
    * stk ++  = 15;                        //用户堆栈长度
    * stk ++  = (INT16U)task & 0xFF;       //任务地址低 8 位
    * stk ++  = (INT16U)task >> 8;         //任务地址高 8 位
    * stk ++  = 0x00;                      //PSW
    * stk ++  = 0x0A;                      //ACC
    * stk ++  = 0x0B;                      //B
    * stk ++  = 0x00;                      //DPL
    * stk ++  = 0x00;                      //DPH
    * stk ++  = 0x00;                      //R0

    /* R3、R2、R1 用于传递任务参数 ppdata,其中 R3 代表存储器类型,R2 为高字节偏移,R1 为低字节
       位移
       通过分析 KEIL 汇编,了解到任务的 void ppdata 参数恰好是用 R3、R2、R1 传递,不是通过虚拟堆
       栈 */
    * stk ++  = (INT16U)ppdata & 0xFF;     //R1
    * stk ++  = (INT16U)ppdata >> 8;       //R2
    * stk ++  = 0x01;                      //R3
    * stk ++  = 0x04;                      //R4
    * stk ++  = 0x05;                      //R5
    * stk ++  = 0x06;                      //R6
    * stk ++  = 0x07;                      //R7
      //不用保存 SP,任务切换时根据用户堆栈长度计算得出
    * stk ++  = (INT16U) (ptos + MaxStkSize) >> 8;   //堆栈映像指针高 8 位
    * stk ++  = (INT16U) (ptos + MaxStkSize) & 0xFF;  //堆栈映像指针低 8 位
    return ((void * )ptos);
}
```

2. 系统时钟初始化

如果使用 51 单片机的 T0 定时器作为系统时钟的中断源，那么它的初始化代码如下：

```
void InitTimer0(void) reentrant
{
    TMOD = TMOD&0xF0;
    TMOD = TMOD|0x01;          //模式 1(16 位定时器)，仅受 TR0 控制
    TH0 = 0x70;                //定义 Tick = 50 次/秒(即 0.02 秒/次)
    TL0 = 0x00;
    //ET0 = 1;                 //允许 T0 中断，此时 EA = 0(51 上电缺省值)，中断还不会发生，满足
                               //在 OSStart()前不产生中断的要求
    TR0 = 1;
}
```

7.2.4 几点必须注意的事项

本来原则上在对 μC/OS－II 进行移植时，不需要修改与处理器无关的代码，但是由于 KEIL 编译器的特殊性，这些代码仍要多处改动。因为在 KEIL 缺省情况下编译的代码不可重入，而要想使 KEIL 编译器编译的代码为可重入代码，则必须在需要可重入的函数声明的后面标注 reentrant 关键字。

另外，pdata、data 在 μC/OS－II 中用作一些函数的形参，但它同时又是 KEIL 的关键字，会导致编译错误。为防止出现此类问题，可以把 μC/OS－II 中函数的形参做一下修改，例如把 pdata 改成 ppdata，data 改成 ddata。

在汇编语言程序中用到了 OSTCBCur、OSTCBHighRdy、OSRunning、OSPrioCur 和 OSPrioHighRdy 这几个变量，为了使用 Ri 而不用 DPTR 对它们进行访问，应该用 KEIL 扩展关键字 IDATA 将它们定义在内部 RAM 中。

7.3 应用举例

7.3.1 LED 数码显示器的驱动程序

LED 显示以其简单、实用、价格低廉等优点，使它是单片机应用中经常使用的一种显示方式。其中一种方案的硬件电路如图 7－3 所示。

在图 7－3 中，8 位 8 段 LED 显示器为共阴极接法，采用并行接口 8155 通过驱动器 7407 来进行驱动。该方案中使用 8155 的 PA 端口进行显示器的位选择(低电平有效)；使用 PB 端口进行显示器的字段选择(高电平有效)。

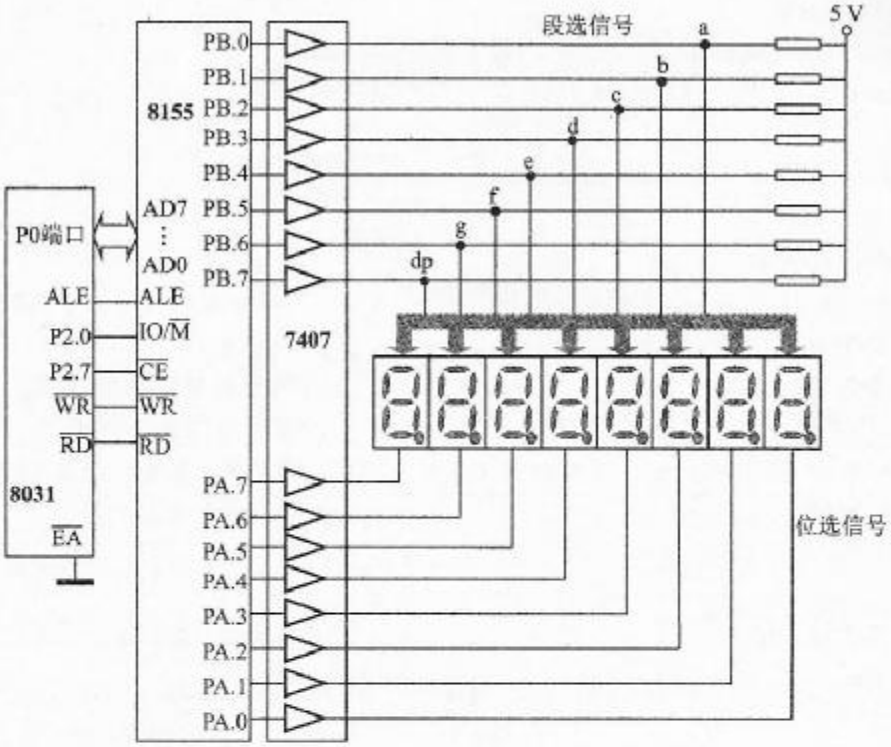


图 7-3 LED 显示硬件原理图

已知 8155 命令字端口地址为 7F00H，PA 端口地址为 7F01H，PB 端口地址为 7F02H，显示器的段选码存放在单片机内部 RAM 中从标号地址 TABLE 开始的连续 16 个单元中。

```
#include <includes.h>
INT8U xdata * DspBuffer;                    //定义显示缓冲区指针
INT8U code * Tbl;                           //定义段码表指针
INT8U xdata * Cmd;                          //定义 8155 命令字寄存器指针
INT8U xdata * Pa;                           //定义 8155 的 PA 端口指针
INT8U xdata * Pb;                           //定义 8155 的 PB 端口指针
INT8U bitCtl;                               //定义位控信号字
INT8U i;
void TaskLEDDsp (void * ppdata) reentrant;
OS_STK TaskLEDDspStk [64];                  //任务堆栈

void main(void)
{
```

```
    OSInit();
    InitTimer0();
    OSTaskCreate(TaskLEDDsp, (void *)0, &TaskLEDDspStk[0],2);
    OSStart();
}
void TaskLEDDsp(void *ppdata) reentrant
{
    ppdata = ppdata;
    Cmd = 0x7F00;                                   //Cmd 指向 8155 命令字寄存器
    Pa = 0x7F01;                                    //Pa 指向 8155 的 PA 端口
    Pb = 0x7F02;                                    //Pb 指向 8155 的 PB 端口
    *Cmd = 0x03;                                    //传送 8155 命令字
    DspBuffer = DSPB;                               //显示缓冲区首地址
    Tbl = TABLE;                                    //段码表首地址
    for(;;)
    {
        bitCtl = 0x7F;                              //位控信号赋初值
        for(i = 0;i<8;i++)
        {
            *Pa = bitCtl;                           //位控信号送 PA 端口
            *Pb = *(Tbl + *(DspBuffer + i));        //段码送 PB 端口
            OSTimeDly(OS_TICKS_PER_SEC);            //延时 1 ms
            bitCtl = bitCtl>>1;                     //位控字左移 1 位
            bitCtl = bitCtl|0x80;                   //位控字高位补 1
        }
    }
}
```

7.3.2 51 单片机串行接口的应用

1. 51 单片机的串行接口

51 系列单片机的内部有一个可编程的全双工异步串行通信接口，它主要由 2 个共用端口(SBUF)的串行数据缓冲器和 1 个波特率发生器构成，片外有一根串行数据接收线 RXD(P3.0)和一根串行数据发送线 TXD(P3.1)。51 系列单片机串行数据接口的结构如图 7－4 所示。

串行数据接口有如表 7－1 所列的 4 种工作方式可供选择。

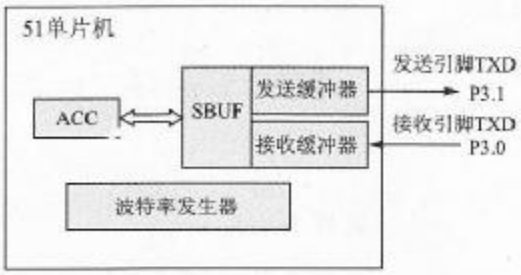


图 7-4　51 单片机串行接口的构成

表 7-1　51 单片机串行数据接口的工作方式

工作方式	SM0	SM1	说　明	波特率
工作方式 0	0	0	移位寄存器(用于 I/O 扩展)	$f_{osc}/12$
工作方式 1	0	1	10 位异步收发	波特率可变，由 T1 控制
工作方式 2	1	0	11 位异步收发	$f_{osc}/64$ 或 $f_{osc}/32$
工作方式 3	1	1	11 位异步收发	波特率可变，由 T1 控制

可以使用 SCON 和 PCON 这两个控制字来设置 51 单片机的串行数据接口的工作方式。这两个控制字的格式分别如图 7-5 和图 7-6 所示。

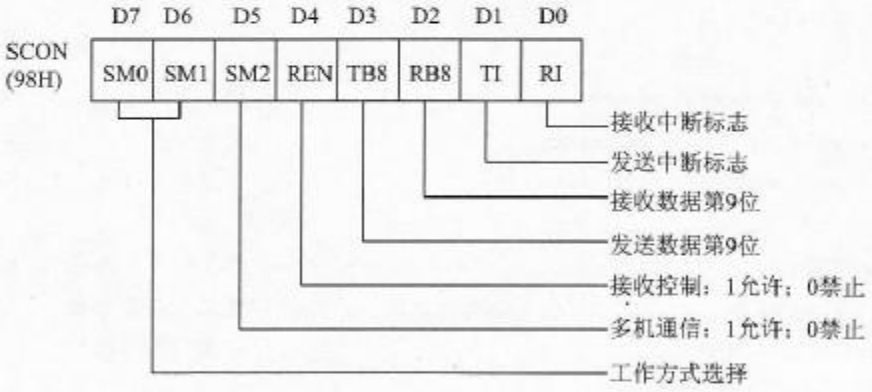


图 7-5　串行接口 SCON 控制字格式

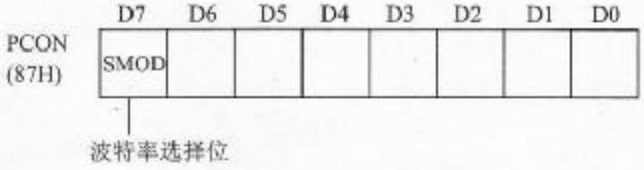


图 7-6　PCON 控制字的格式

当采用工作方式 0 时：

$$波特率=\frac{f_{OSC}}{12}$$

当采用工作方式 2 时：

$$波特率=f_{OSC}\frac{2^{SMOD}}{64}$$

当采用工作方式 1 和方式 3 时：

$$波特率=f_{OSC}\frac{2^{SMOD}}{32\times12}\left(\frac{1}{2^K-X}\right)$$

式中：X——定时器 T1 定时预置初值；

K——定时器的位数(8、13 或 16)。

2. 应用程序

下面的应用程序中设置了 3 个任务，每个任务通过串行接口滚动输出一个字符串。

```
//*****************************************************************
//                    串口应用程序清单
//*****************************************************************

#include <includes.h>

void Task1(void *ppdata) reentrant;                    //定义任务 1
void Task2(void *ppdata) reentrant;                    //定义任务 2
void Task3(void *ppdata) reentrant;                    //定义任务 3

OS_STK Task1Stk[64];                                   //定义任务 1 堆栈
OS_STK Task2Stk[64];                                   //定义任务 2 堆栈
OS_STK Task3Stk[64];                                   //定义任务 3 堆栈

//主函数 ------------------------------------------------------------------
void main(void)
{
    OSInit();

    InitTimer0();                                      //初始化系统时钟
    InitSerial();                                      //串口初始化
    InitSerialBuffer();
```

```
    OSTaskCreate(Task1, (void * )0, &Task1Stk [0],2);
    OSTaskCreate(Task2, (void * )0, &Task2Stk [0],3);
    OSTaskCreate(Task3, (void * )0, &Task3Stk [0],4);

    OSStart();
}

//任务 1-----------------------------------------------------------------
void Task1(void * ppdata) reentrant
{
    ppdata = ppdata;

    ET0 = 1;

    clrscr();
    PrintStr("\n\t\t*******************************\n");
    PrintStr("\t\t*          Hello! The world.          *\n");
    PrintStr("\t\t*******************************\n\n\n");

    for(;;)
    {
        PrintStr("\t\t\tThis is Task1.\n");
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}

//任务 2-----------------------------------------------------------------
void Task2(void * ppdata) reentrant
{
    ppdata = ppdata;

    for(;;)
    {
        PrintStr("\t\t\t\tThis is Task2.\n");
        OSTimeDly(3 * OS_TICKS_PER_SEC);
    }
}
```

```
//任务 3-----------------------------------------------------------------
void Task3(void * ppdata) reentrant
{
    ppdata = ppdata;

    for(;;)
    {
        PrintStr("\t\t\t\t\tThis is Task3.\n");
        OSTimeDly(6 * OS_TICKS_PER_SEC);
    }
}


//*****************************************************************
//                          serial.h 文件
//*****************************************************************
void InitSerial() reentrant;                          //串口初始化
void InitSerialBuffer(void) reentrant;                //串口缓冲区初始化
void PrintChar(unsigned char ch) reentrant;           //显示字符
void PrintStr(unsigned char * str) reentrant;         //显示字符串
void clrscr() reentrant;                              //清屏
void serial(void) reentrant;                          //串口中断服务子程序

//收发数据缓冲区长度
#define LenTxBuf        2000
#define LenRxBuf        50

#define MaxLenStr       100


//*****************************************************************
//                        serial.c 文件
//*****************************************************************
INT8U xdata TxBuf[LenTxBuf],RxBuf[LenRxBuf];          //收发缓冲区实体
INT8U xdata * inTxBuf, * outTxBuf, * inRxBuf, * outRxBuf;   //收发缓冲区读/写指针
//串口初始化程序----------------------------------------------------------
void InitSerial() reentrant
```

```
{
        TMOD = TMOD&0x0F;
        TMOD = TMOD|0x20;
        TL1 = 0xFD;
        TH1 = 0xFD;                                     //19 200, 22.118 4 MHz
        SCON = 0x50;
        PCON = 0x00;
        TR1 = 1;                                        //启动定时器 T1
}

//中断时调用的串口任务函数----------------------------------------------------
void serial(void) reentrant
{
        INT8U  * t;

        if(TI){
                TI = 0;
                if(inTxBuf == outTxBuf)
                {
                    TIflag = 1;
                    return;
                }                                       //TxBuf 空
                SBUF = * outTxBuf;
                outTxBuf ++ ;
                if(outTxBuf == TxBuf + LenTxBuf)
                    outTxBuf = TxBuf;
            }
        if(RI){
                RI = 0;
                t = inRxBuf;t ++ ;
                if(t == RxBuf + LenRxBuf)
                    t = RxBuf;
                if(t == outRxBuf)
                    return;                             //RxBuf 满
                * inRxBuf = SBUF;
                inRxBuf = t;
            }
}
```

```
//显示字符串函数-----------------------------------------------------------------
void PrintStr(unsigned char *str) reentrant
{
        INT8U i;
        INT8U j;
        INT8U ch;

        OS_ENTER_CRITICAL();                          //入临界区
        for(i=0;i<MaxLenStr;i++)
        {
                ch=*(str+i);
                if(ch=='\0')
                    break;
                else
                    if(ch=='\n')
                    {
                        PrintChar(10);
                        PrintChar(13);
                    }
                    else
                        if(ch=='\t'){
                        for(j=0;j<TABNum;j++)
                                PrintChar(' ');
                }
                else PrintChar(ch);
        }
        OS_EXIT_CRITICAL();                           //出临界区
}

//显示字符函数-----------------------------------------------------------------
void PrintChar(unsigned char ch) reentrant
{
        INT8U  *t;

        OS_ENTER_CRITICAL();                          //入临界区
        t=inTxBuf;t++;
        if(t==TxBuf+LenTxBuf)
```

上述应用程序在 KEIL51 中运行的结果如图 7－7 所示。

图 7－7　在 KEIL51 中运行的结果

7.4　小　结

- 多任务操作系统中，系统所提供的函数应该能允许同时被多个任务所调用，而不会通过函数中变量的耦合引起任务之间的相互干扰。这样的函数叫做可重入函数。
- 应该使用硬件定时器来作为时钟中断源。这个定时器可以是与微处理器集成在一个芯片上的，也可以是分立的。定时中断的频率以 10～100 Hz 为宜。
- 不同的处理器支持堆栈的增长方向也不同，因此在对 μC/OS－II 进行移植时，一定要正确地定义堆栈的格式。

7.5　练习题

1. 什么叫可重入函数？
2. 为什么在把 μC/OS－II 向 51 单片机上移植时要设立系统堆栈和任务堆栈映像？

第8章　在 ARM7 上移植 μC/OS－II

目前，在嵌入式处理器芯片中，以 ARM7 为核的处理器是人们应用较多的一种。由于它具有多种工作模式，并且支持两种不同的指令集：标准 32 位 ARM 指令集和 16 位 Thumb 指令集，因而在把 μC/OS－II 向 ARM 移植时要考虑一些比较特殊的问题。本章对在 ARM7 上移植 μC/OS－II 做了比较详细的介绍。

本章的主要内容有：

- C 编译器的选择及 ARM 工作模式的选择；
- 不依赖工作模式的系统底层函数的实现方法；
- 为充分发挥 ARM 的功能而给 μC/OS－II 添加的 4 个系统底层函数。

8.1　移植规划

8.1.1　编译器的选择

目前，适用于 ARM 处理器核的 C 编译器有很多种，例如 SDT、ADS、IAR、TASKING 和 GCC 等。其中人们使用比较多的是 SDT、ADS 和 GCC。

SDT 和 ADS 都是由 ARM 公司自己开发的，其中 ADS 是 SDT 的升级版，而且以后 ARM 公司不再支持 SDT，所以进行 μC/OS－II 向 ARM7 的移植时最好采用 ADS 编译器。本章介绍的移植代码就是使用 ADS 编译器来编译的，并在 MagicARM2200 教学实验开发平台调试通过。有关 MagicARM2200 教学实验开发平台的详细情况请参见附录 E。

8.1.2　ARM7 工作模式的选择

ARM7 处理器核具有用户、系统、管理、中止、未定义、中断和快中断 7 种模式。由于管理、中止、未定义、中断和快中断这 5 个模式与异常相关，不太适合 μC/OS－II 任务使用。而系统模式除了是特权模式外，其他与用户模式一样，因此，μC/OS－II 任务可以使用的模式应该是用户模式和系统模式。

在下面的移植中，是按照 μC/OS－II 任务的缺省模式为 ARM 用户模式，任务的可选模式

是为 ARM 系统模式来设计的。

为了实现任务工作模式的切换，在移植时还给 μC/OS－II 添加了两个用于模式切换的系统函数：ChangeToSYSMode()和 ChangeToUSRMode()。

8.2 移 植

8.2.1 文件 OS_CPU.H 的编写

1. 不依赖于编译的数据类型

μC/OS－II 不使用 C 语言中的 short、int、long 等与处理器类型有关数据的类型，而代之以移植性强的整数数据类型，这样既直观又便于移植，不过这就成了必须移植的代码。根据 ADS 编译器的特性，在文件 OS_CPU.H 中这些数据类型的定义如下：

```
typedef  unsigned char     BOOLEAN;
typedef  unsigned char     INT8U;
typedef  signed char       INT8S;
typedef  unsigned short    INT16U;
typedef  signed short      INT16S;
typedef  unsigned int      INT32U;
typedef  signed int        INT32S;
typedef  float             FP32;
typedef  double            FP64;
typedef  INT32U            OS_STK;
```

2. 利用软中断实现底层接口

由于 ARM7 处理器核允许用户任务使用两种不同的处理器模式：用户模式和系统模式，而且在不同的模式下应用程序使用系统资源时是具有不同的访问控制权限的。

于是，怎样使在不同的 ARM 工作模式下调用 μC/OS－II 底层接口函数不受访问权限的限制，就是在移植时必须考虑的一个问题。解决这个问题的方法之一就是使用 ARM7 的软中断 SWI。

按 ADS 编译器的规定，用户可使用关键字__swi 作为前缀来声明一个利用软中断的调用，其格式如下：

__swi(功能号) 返回值类型 名称(参数列表);

其中关键字__swi 后面括号中的字段叫做软中断的功能编号。系统就是根据这个编号在软中断服务程序中来确定应执行的程序段的。显而易见，括号后面的格式与声明一个函数原型的

格式一样，也可以有参数和返回值。因此，可使用户在用户应用程序中像调用一个普通函数那样实现一个软中断调用。如果在移植时把 μC/OS - II 需要规避由于 ARM 处在不同工作模式时所造成的访问限制的系统函数代码编写在软中断服务程序中，就可以解决上述问题了。

μC/OS - II 中须用软中断实现的函数及其软中断功能号分配见表 8 - 1。

表 8 - 1 须用软中断来实现的函数

功能号	函数名	说 明
0x00	void OS_TASK_SW(void)	任务级任务切换函数
0x01	void _OSStartHighRdy(void)	运行优先级最高的任务，由 OSStartHighRdy()产生
0x02	void OS_ENTER_CRITICAL(void)	关中断
0x03	void OS_EXIT_CRITICAL(void)	开中断
0x80	void ChangeToSYSMode(void)	任务切换到系统模式
0x81	void ChangeToUSRMode(void)	任务切换到用户模式
0x82	void TaskIsARM(INT8U prio)	任务代码是 ARM 代码
0x83	void TaskIsTHUMB(INT8U prio)	任务代码是 Thumb 代码

在文件 OS_CPU.H 中，表 8 - 1 中函数的声明代码如下：

```
__swi(0x00) void OS_TASK_SW(void);              //任务级任务切换函数
__swi(0x01) void _OSStartHighRdy(void);         //运行优先级最高的任务
__swi(0x02) void OS_ENTER_CRITICAL(void);       //关中断
__swi(0x03) void OS_EXIT_CRITICAL(void);        //开中断
__swi(0x80) void ChangeToSYSMode(void);         //任务切换到系统模式
__swi(0x81) void ChangeToUSRMode(void);         //任务切换到用户模式
__swi(0x82) void TaskIsARM(INT8U prio);         //任务代码是 ARM 代码
__swi(0x83) void TaskIsTHUMB(INT8U prio);       //任务代码是 Thumb 代码
```

在上面声明的函数中，需要注意的是，后面的 4 个函数不是 μC/OS - II 的原有函数，而是在移植时由移植者根据 ARM 的需要而编写并添加到 μC/OS - II 中的函数。

实际上，这些带前缀__swi 的函数，其实不是通常意义上的函数，而只是可以让用户在应用程序中使用这些“函数名”去引发一个携带功能号的软中断 SWI，并且可根据中断功能号去执行对应的中断服务程序段（这个中断服务程序段相当于“函数体”）。由于可向这些中断服务程序段传递参数，并且也可以有返回值，因此也可把它们看作函数。

3. OS_STK_GROWTH

虽然 ARM 处理器对堆栈向上及向下的两种增长方式都给予了支持，但由于编译器 ADS 仅支持堆栈从上往下长，并且必须是满递减堆栈，所以在文件中用来定义堆栈增长方式的常量

OS_STK_GROWTH 的值应该为 1，即

```
#define  OS_STK_GROWTH     1
```

8.2.2 文件 OS_CPU_C.C 的编写

1. 任务堆栈初始化函数 OSTaskStkInit()

在编写任务堆栈初始化函数 OSTaskStkInit()之前，必须先根据处理器的结构和特点确定任务的堆栈结构。本移植的堆栈结构见图 8－1。

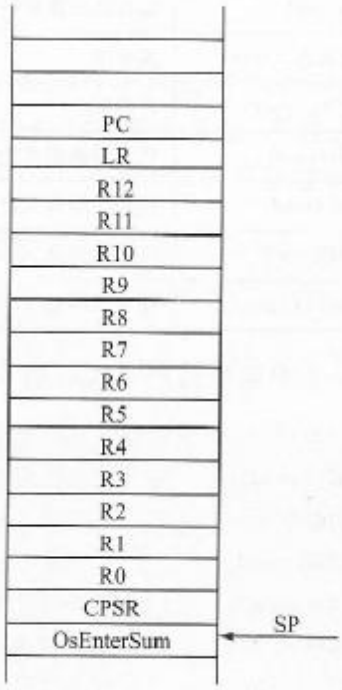


图 8－1 任务堆栈结构

根据图 8－1，很容易写出函数 OSTaskStkInit()的代码。函数 OSTaskStkInt()的代码清单如下：

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

```
    opt = opt;
    stk = ptos;                          //获取堆栈指针
    /* 建立任务环境,ADS1.2 使用满递减堆栈
    *stk = (OS_STK) task;                //pc
    * --stk = (OS_STK) task;             //lr
    * --stk = 0;                         //R12
    * --stk = 0;                         //R11
    * --stk = 0;                         //R10
    * --stk = 0;                         //R9
    * --stk = 0;                         //R8
    * --stk = 0;                         //R7
    * --stk = 0;                         //R6
    * --stk = 0;                         //R5
    * --stk = 0;                         //R4
    * --stk = 0;                         //R3
    * --stk = 0;                         //R2
    * --stk = 0;                         //R1
    * --stk = (unsigned int) pdata;      //R0,第一个参数使用 R0 传递
    * --stk = (USER_USING_MODE|0x00);    //spsr,允许 IRQ、FIQ 中断
    * --stk = 0;                         //关中断计数器 OsEnterSum
    return (stk);
}
```

堆栈中的 OsEnterSum 比较特别,它不是 CPU 的寄存器,而是一个全局变量,目的是用它来保存开关中断的次数。这样,用户在任务中就不必过分考虑开关中断对其他任务的影响。

2. OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()

μC/OS - II 分别使用宏 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()来关中断和开中断。在 ARM 处理器核中可利用改变程序状态寄存器 CPSR 中的相应控制位实现。由于使用了软件中断,程序状态寄存器 CPSR 保存到程序状态保存寄存器 SPSR 中,软件中断退出时会将 SPSR 恢复到 CPSR 中,所以程序只要改变程序状态保存寄存器 SPSR 中的相应的控制位就可以了。

关中断宏 OS_ENTER_CRITICAL()的实现代码如下:

```
__asm
{
    MRS     R0,SPSR
    ORR     R0,R0,#NoInt
    MSR     SPSR_c,R0
}
OsEnterSum++;
```

开中断宏 OS_EXIT_CRITICAL()的实现代码如下：

```
if ( -- OsEnterSum == 0)
{
    __asm
    {
        MRS     R0,SPSR
        BIC     R0,R0,#NoInt
        MSR     SPSR_c,R0
    }
}
```

3. 处理器模式转换函数 ChangeToSYSMode()和 ChangeToUSRMode()

本移植方案根据 ARM 核的特点和移植目标，移植时在 μC/OS－II 中增加了 4 个系统函数。其中，两个是用来转换处理器模式的函数 ChangeToSYSMode()和 ChangeToUSRMode()；两个是设置任务的初始指令集的函数 TaskIsARM()和 TaskIsTHUMB()。

处理器模式转换函数 ChangeToSYSMode()和 ChangeToUSRMode()使用软件中断功能 0x80 和 0x81 实现，其中函数 ChangeToSYSMode()把当前任务转换到系统模式。

```
__asm
{
    MRS     R0,SPSR
    BIC     R0,R0,#0x1F
    ORR     R0,R0,#SYS32Mode
    MSR     SPSR_c,R0
}
```

函数 ChangeToUSRMode()把当前任务转换到用户模式。

```
__asm
{
    MRS     R0,SPSR
    BIC     R0,R0,#0x1F
    ORR     R0,R0,#USR32Mode
    MSR     SPSR_c,R0
}
```

它们可以在任何情况下使用。它们改变程序状态保留寄存器 SPSR 的相应位段，而程序状态保留寄存器会在软件中断退出时复制到程序状态寄存器 CPSR，任务的处理器模式就改变了。

为使用户调用它们时不受处理器模式的限制，它们都是通过软件中断服务程序来实现的。处理器模式转换函数 ChangeToSYSMode()的软件中断功能编号为 0x80；函数 ChangeToUSRMode()的软件中断功能编号为 0x81。

4. 设置任务的初始指令集函数 TaskIsARM()和 TaskIsTHUMB()

函数 TaskIsARM()和 TaskIsTHUMB()被用来在一个 μC/OS - II 任务运行之前，指定该任务使用的指令集。

其中函数 TaskIsARM()用于声明一个待运行任务所使用的指令集是 ARM 指令集；而函数 TaskIsTHUMB()用于声明一个待运行任务所使用的指令集是 Thumb 指令集。在调用这两个函数时，都需要传递一个惟一的参数：被指定指令集的任务的优先级别。

指定用户任务初始指令集为 ARM 指令集的函数 TaskIsARM()在软中断服务程序中对应的代码段如下：

```
if (Regs[0] <= OS_LOWEST_PRIO)
{
    ptcb = OSTCBPrioTbl[Regs[0]];
    if (ptcb != NULL)
    {
        ptcb -> OSTCBStkPtr[1] &= ~(1<<5);
    }
}
```

指定用户任务初始指令集为 Thumb 指令集的函数 TaskIsTHUMB()在软中断服务程序中对应的代码段如下：

```
if (Regs[0] <= OS_LOWEST_PRIO)
{
    ptcb = OSTCBPrioTbl[Regs[0]];
    if (ptcb != NULL)
    {
        ptcb -> OSTCBStkPtr[1] |= (1<<5);
    }
}
```

由于 ARM 是用处理器的寄存器 CPSR 的 T 标志位来指示和设置系统当前指令集的，而 μC/OS - II 任务在创建之后未运行之前，是把寄存器 CPSR 的值存放在任务堆栈中的，所以函数 TaskIsARM()和 TaskIsTHUMB()的核心工作就是修改存放在任务堆栈中的 CPSR 的 T 标志位。其过程为：首先，程序判断传递的参数(任务的优先级)是否在允许的范围内；然后获取任务的任务控制块 TCB 的指针，如果指针指向有效 TCB，则修改在任务堆栈中存储的

CPSR 的 T 标志位。

函数 TaskIsARM()和 TaskIsTHUMB()的软件中断功能号分别为 0x82 和 0x83。

值得注意的是，由于要求这两个函数必须在相应的任务建立后但还没有运行时调用，所以在低优先级任务中创建高优先级任务就要十分小心；否则会出现问题。此时，最好按下面提供的三种方法来做：

- 高优先级任务使用默认的指令集；
- 改变函数 OSTaskCreateHook()使任务默认不是处于就绪状态，建立任务后调用函数 OSTaskResume()来使任务进入就绪状态；
- 建立任务时禁止任务切换，调用函数 TaskIsARM()或 TaskIsTHUMB()后再允许任务切换。

5. 软件中断服务程序的 C 语言部分

前面曾经谈到，为了使 μC/OS－II 的一些底层系统函数在调用时与处理器工作模式无关，所以在移植时这些函数是通过软件中断来实现的。该软件中断服务程序代码如下：

```
void SWI_Exception(
            int SWI_Num,    //软中断功能号
            int *Regs       //Regs 为指向堆栈中保存寄存器的值的指针
            )
{
OS_TCB    *ptcb;
    switch(SWI_Num)
    {     //宏 OS_ENTER_CRITICAL()的代码段 ----------------------------------
        case 0x02:
            __asm
            {
                MRS     R0,SPSR
                ORR     R0,R0,#NoInt
                MSR     SPSR_c,R0
            }
            OsEnterSum++;
            break;
        //宏 OS_EXIT_CRITICAL()的代码段 ----------------------------------
        case 0x03:
            if(--OsEnterSum == 0)
            {
                __asm
```

```
        {
            MRS     R0,SPSR
            BIC     R0,R0,#NoInt
            MSR     SPSR_c,R0
        }
    }
    break;
//函数 ChangeToSYSMode()的代码段----------------------------------
case 0x80:
    __asm
    {
        MRS     R0,SPSR
        BIC     R0,R0,#0x1F
        ORR     R0,R0,#SYS32Mode
        MSR     SPSR_c,R0
    }
    break;
//函数 ChangeToUSRMode()的代码段----------------------------------
case 0x81:
    __asm
    {
        MRS     R0,SPSR
        BIC     R0,R0,#0x1F
        ORR     R0,R0,#USR32Mode
        MSR     SPSR_c,R0
    }
    break;
//函数 TaskIsARM()的代码段----------------------------------------
case 0x82:
    if (Regs[0] <= OS_LOWEST_PRIO)
    {
        ptcb = OSTCBPrioTbl[Regs[0]];
        if (ptcb != NULL)
        {
            ptcb -> OSTCBStkPtr[1] &= ~(1<<5);
        }
    }
```

```
            break;
        //函数 TaskIsTHUMB()的代码段----------------------------------------
        case 0x83:
            if (Regs[0] <= OS_LOWEST_PRIO)
            {
                ptcb = OSTCBPrioTbl[Regs[0]];
                if (ptcb != NULL)
                {
                    ptcb -> OSTCBStkPtr[1] |= (1<<5);
                }
            }
            break;
        default:
            break;
    }
}
```

从以上的代码中可以看出，程序中使用一个 switch 开关结构把软件中断功能号为 0x02、0x03、0x80、0x81、0x82、0x83 的 6 个组织到一个中断服务程序 SWI_Exception()中了。

参照表 8－1 也可发现，软中断功能号为 0x00、0x01 的函数并没有在这里实现，而是在 OS_CPU_A.S 文件中实现的。详情请参考 8.2.3 小节。

6. OSStartHighRdy()

函数 OSStartHighRdy()的代码清单如下：

```
void OSStartHighRdy(void)
{
    _OSStartHighRdy();
}
```

由表 8－1 可知，这是调用软中断的 0x01 号功能，该功能是在文件 OS_CPU_A.S 中用汇编语言实现的。详情请参考 8.2.3 小节。

7. 钩子函数

μC/OS－II 为了用户在系统函数中书写自己的代码而预置了一些函数名带有 Hook 字样的钩子函数，这些函数在移植时可全为空函数。

8.2.3 文件 OS_CPU_A.S 的编写

1. 软件中断服务程序的汇编语言部分

软中断服务程序的汇编语言部分程序清单如下：

```
SoftwareInterrupt
        LDR      SP, StackSvc;------------------------------------------ (1)
        STMFD  SP!, {R0-R3, R12, LR};------------------------------------ (2)
        MOV    R1, SP;---------------------------------------------------- (3)
        MRS    R3, SPSR;-------------------------------------------------- (4)
        TST    R3, #T_bit;------------------------------------------------ (5)
        LDRNEH R0, [LR, #-2];------------------------------------------- (6)
        BICNE  R0, R0, #0xFF00;------------------------------------------ (7)
        LDREQ  R0, [LR, #-4];------------------------------------------- (8)
        BICEQ  R0, R0, #0xFF000000;-------------------------------------- (9)
        CMP    R0, #1;---------------------------------------------------(10)
        LDRLO  PC, =OSIntCtxSw;----------------------------------------- (11)
        LDREQ  PC, =__OSStartHighRdy;----------------------------------(12)
        BL     SWI_Exception;--------------------------------------------(13)
        LDMFD  SP!, {R0-R3, R12, PC}^;----------------------------------- (14)
```

ARM 处理器要求软中断的功能号必须包含在 SWI 指令中，这样应用程序就可在执行 SWI 指令时，通过读取该条指令的相应位段来获得功能号。

由于 ARM 处理器核两个指令集的指令长度不同，在不同的指令集中 SWI 指令的功能号所处的位段也不同，因此在上面的代码中先用(4)、(5)两条指令判断处理器是处在什么指令集状态。如果是 Thumb 指令集状态，则通过 (6)、(7)两条指令获取得软中断功能号；如果是 ARM 指令集状态，则通过(9)、(10)两条指令取得软中断功能号。

然后，通过指令(10)判断软中断功能号是 0 还是 1。如果是 0，就跳转到(11)(任务切换函数处)，执行 OS_TASK_SW()；如果是 1，就跳转到(12)，执行_OSStartHighRdy()。

其他功能就给软件中断的 C 语言处理函数处理。它有两个参数：第一个就是功能号，存于 R0 中；第二个是保存参数和返回值的指针，也就是堆栈中存储用户函数 R0～R3 的位置，实质就是当前堆栈指针的值，它存于 R1 中，见代码清单中的指令(3)。

2. OS_TASK_SW()和 OSIntCtxSw()

任务级任务切换函数 OS_TASK_SW()及中断级任务切换函数 OSIntCtxSw()均用汇编语言编写。

因为在进行任务切换时，被切换的任务可能正处于系统模式，也可能正处于用户模式；可能使用 ARM 指令集，也可能使用 Thumb 指令集；又由于只有用系统模式的 SPSR 保存任务的 CPSR，才能保证正确的任务切换，所以一定要在 ARM 的系统模式下进行任务切换(见下面代码清单的第(37)条指令)。

OS_TASK_SW()和 OSIntCtxSw()的代码如下：

```
OSIntCtxSw
        ;保存任务环境
        LDR     R2,[SP,#20]                 ;获取 PC----------------------------(1)
        LDR     R12,[SP,#16]                ;获取 R12---------------------------(2)
        MRS     R0,CPSR                     ;------------------------------(3)

        MSR     CPSR_c,#(NoInt | SYS32Mode) ;------------------------------(4)
        MOV     R1,LR                       ;------------------------------(5)
        STMFD   SP!,{R1-R2}                 ;保存 LR、PC------------------------(6)
        STMFD   SP!,{R4-R12}                ;保存 R4～R12----------------------(7)

        MSR     CPSR_c,R0                   ;------------------------------(8)
        LDMFD   SP!,{R4-R7}                 ;获取 R0～R3-----------------------(9)
        ADD     SP,SP,#8                    ;出栈 R12、PC----------------------(10)
        MSR     CPSR_c,#(NoInt | SYS32Mode) ;------------------------------(11)
        STMFD   SP!,{R4-R7}                 ;保存 R0～R3-----------------------(12)
        LDR     R1,=OsEnterSum              ;获取 OsEnterSum-------------------(13)
        LDR     R2,[R1]                     ;------------------------------(14)
        STMFD   SP!,{R2,R3}                 ;保存 CPSR、OsEnterSum-------------(15)
       ;保存当前任务堆栈指针到当前任务的 TCB
        LDR     R1,=OSTCBCur                ;------------------------------(16)
        LDR     R1,[R1]                     ;------------------------------(17)
        STR     SP,[R1]                     ;------------------------------(18)

        BL      OSTaskSwHook                ;调用钩子函数----------------------(19)
       ;OSPrioCur <= OSPrioHighRdy
        LDR     R4,=OSPrioCur               ;------------------------------(20)
        LDR     R5,=OSPrioHighRdy           ;------------------------------(21)
        LDRB    R6,[R5]                     ;------------------------------(22)
        STRB    R6,[R4]                     ;------------------------------(23)
       ;OSTCBCur <= OSTCBHighRdy
        LDR     R6,=OSTCBHighRdy            ;------------------------------(24)
        LDR     R6,[R6]                     ;------------------------------(25)
        LDR     R4,=OSTCBCur                ;------------------------------(26)
        STR     R6,[R4]                     ;------------------------------(27)
```

```
OSIntCtxSw_1
        ;获取新任务堆栈指针
        LDR     R4,[R6];17 寄存器 CPSR,OsEnterSum,R0~R12,LR,SP---------------- (28)
        ADD     SP,R4,#68                   ;-----------------------------(29)
        LDR     LR,[SP,#-8]                 ;-----------------------------(30)
        MSR     CPSR_c,#(NoInt | SVC32Mode) ;进入管理模式-------------------(31)
        MOV     SP,R4                       ;设置堆栈指针--------------------(32)

        LDMFD   SP!,{R4,R5}                 ;CPSR,OsEnterSum----------------- (33)
        LDR     R3,=OsEnterSum              ;恢复新任务的 OsEnterSum-------------(34)
        STR     R4,[R3]                     ;-----------------------------(35)
        MSR     SPSR_cxsf,R5                ;恢复 CPSR-----------------------(36)
        LDMFD   SP!,{R0-R12,LR,PC}^         ;运行新任务--------------------- (37)
```

3. 启动最高优先级就绪任务函数 OSStartHighRdy()

由 8.2.2 小节知道，函数 OSStartHighRdy()是通过调用__OSStartHighRdy 来启动最高优先级就绪任务的，而__OSStartHighRdy 是必须用汇编语言来编写的。其代码清单如下：

```
__OSStartHighRdy
    MSR     CPSR_c, #(NoInt | SYS32Mode);---------------------------------- (1)
    LDR     R4, =OSRunning;------------------------------------------------ (2)
    MOV     R5,#1;--------------------------------------------------------(3)
    STRB    R5,[R4];------------------------------------------------------(4)
    BL      OSTaskSwHook;-------------------------------------------------(5)
    LDR     R6, =OSTCBHighRdy;--------------------------------------------- (6)
    LDR     R6,[R6];------------------------------------------------------(7)
    B       OSIntCtxSw_1;-------------------------------------------------(8)
```

8.2.4 关于中断及时钟节拍

在本移植中，只使用了 ARM 的 IRQ 中断。由于不同的 ARM 芯片的中断系统并不完全一样，因此不可能编写出对所有使用 ARM 核的处理器通用的中断及时钟节拍移植代码。

但是，为了使用户可用 C 语言编写中断服务程序时不必为处理器的硬件区别而困扰，这里还是根据 μC/OS - II 对中断服务程序的要求以及 ARM7 体系结构特点和 ADS 编译器特点，编写了一个适用于所有基于 ARM7 核处理器的汇编宏。这个宏实现了 μC/OS - II for ARM7 中断服务程序的汇编语言代码与 C 语言函数代码之间的通用接口。其代码如下：

```
MACRO
$AIRQ_Label HANDLER $IRQ_Exception_Function
  EXPORT    $IRQ_Label                    ;输出的标号---------------------------(1)
  IMPORT    $IRQ_Exception_Function       ;引用的外部标号-----------------------(2)
$IRQ_Label
  SUB       LR, LR, #4                    ;计算返回地址-------------------------(3)
  STMFD     SP!, {R0-R3, R12, LR}         ;保存任务环---------------------------(4)
  MRS       R3, SPSR                      ;保存状态-----------------------------(5)
  STMFD     SP, {R3, SP, LR}^             ;保存用户状态的R3、SP、LR-------------(6)
; OSIntNesting++
  LDR       R2, =OSIntNesting             ;-----------------------------------(7)
  LDRB      R1, [R2]                      ;-----------------------------------(8)
  ADD       R1, R1, #1                    ;-----------------------------------(9)
  STRB      R1, [R2]                      ;-----------------------------------(10)

  SUB       SP, SP, #4*3                  ;-----------------------------------(11)

  MSR       CPSR_c, #(NoInt | SYS32Mode)  ;切换到系统模式-----------------------(12)
  CMP       R1, #1                        ;-----------------------------------(13)
  LDREQ     SP, =StackUsr                 ;-----------------------------------(14)

  BL        $IRQ_Exception_Function       ;调用C语言的中断处理程序---------------(15)
  MSR       CPSR_c, #(NoInt | SYS32Mode)  ;切换到系统模式-----------------------(16)
  LDR       R2, =OsEnterSum;OsEnterSum使OSIntExit退出时关中断-------------------(17)
  MOV       R1, #1                        ;-----------------------------------(18)
  STR       R1, [R2]                      ;-----------------------------------(19)

  BL        OSIntExit                     ;-----------------------------------(20)

  LDR       R2, =OsEnterSum               ;因为服务程序退出,所以OsEnterSum=0--------(21)
  MOV       R1, #0                        ;-----------------------------------(22)
  STR       R1, [R2]                      ;-----------------------------------(23)

  MSR       CPSR_c, #(NoInt | IRQ32Mode)  ;切换回IRQ模式-----------------------(24)
  LDMFD     SP, {R3, SP, LR}^             ;恢复用户状态的R3,SP,LR---------------(25)

  LDR       R0, =OSTCBHighRdy             ;-----------------------------------(26)
```

```
    LDR      R0, [R0]                    ;---------------------------------- (27)
    LDR      R1, = OSTCBCur              ;---------------------------------- (28)
    LDR      R1, [R1]                    ;---------------------------------- (29)
    CMP      R0, R1                      ;---------------------------------- (30)

    ADD      SP, SP, #4 * 3              ;---------------------------------- (31)
    MSR      SPSR_cxsf, R3               ;---------------------------------- (32)
    LDMEQFD  SP!, {R0-R3, R12, PC}^      ;不进行任务切换---------------------- (33)
    LDR      PC, = OSIntCtxSw            ;进行任务切换-------------------------(34)
MEND
```

有了这个宏，中断服务程序的 C 语言部分的编写很简单。其示意代码如下：

```
void ISR(void)
{
    OS_ENTER_CRITICAL()-------------------------------------------------- (1)
    清除中断源;---------------------------------------------------------- (2)
    通知中断控制器中断结束;---------------------------------------------(3)
    开中断; OS_EXIT_CRITICAL();-----------------------------------------(4)
    用户处理程序;-------------------------------------------------------- (5)
}
```

因为在系统响应了中断并进入中断服务程序之后，中断计数器 OsEnterSum 的值一定是 0，如果在此之后服务程序调用了 μC/OS - II 系统服务函数（它们一般会成对调用 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()），则很可能在系统没有清中断源情况下就打开中断，造成处理器中断系统的异常而使应用程序工作异常，所以在中断服务程序中必须设置一个临界段，并且在这个临界段中处理清除中断源事宜，见示意代码中的(1)～(4)。

8.3 移植 μC/OS - II 到 LPC2000

目前，由于 ARM 公司的政策，基于 ARM7 处理器核的各种处理器芯片在存储系统、片内外设设备、中断系统等方面都存在或多或少的差异，再加上嵌入式系统的具体应用项目对资源的要求不一样，因此移植时，在前面介绍的 μC/OS - II 在 ARM7 上移植的通用代码的基础上，还需要有一些与上述问题相关的代码由用户自行编写。如果使用 ADS1.2 这个集成开发环境来编译代码，则还需要用户或厂家编写部分启动代码。由于启动代码与具体的 ARM 器件关联很大，限于篇幅，本书不作介绍，只以 PHILIPS 公司的 LPC2200 系列芯片为例，介绍相对比较通用的汇编语言和移值代码。

8.3.1 挂接 SWI 软件中断

将软中断异常处理程序挂接到内核是通过修改启动代码来实现的，见下面程序清单的(3)和(11)。

```
Reset
        LDR         PC,ResetAddr;                                   (1)
        LDR         PC,UndefinedAddr;                               (2)
        LDR         PC,SWI_Addr;                                    (3)
        LDR         PC,PrefetchAddr;                                (4)
        LDR         PC,DataAbortAddr;                               (5)
        DCD         0xB9205F80;                                     (6)
        LDR         PC,[PC, # -0xFF0];                              (7)
        LDR         PC,FIQ_Addr;                                    (8)

ResetAddr           DCD     ResetInit;                              (9)
UndefinedAddr       DCD     Undefined;                              (10)
SWI_Addr            DCD     SoftwareInterrupt;                      (11)
PrefetchAddr        DCD     PrefetchAbort;                          (12)
DataAbortAddr       DCD     DataAbort;                              (13)
nouse               DCD     0;                                      (14)
IRQ_Addr            DCD     IRQ_Handler;                            (15)
FIQ_Addr            DCD     FIQ_Handler;                            (16)
```

8.3.2 中断及时钟节拍中断

编写中断服务程序代码比较简单，按照 8.2.4 小节编写 C 语言处理函数即可。关键在于把程序与芯片的相关中断源挂接，使芯片在产生相应的中断后，会调用相应的处理程序。这需要做以下两方面工作。

1. 增加汇编语言接口的支持

方法是在文件 IRQ.S 中适当位置添加如下代码：

```
×××_Handler     HANDLER Xxx_Exception
```

在实际编写代码时，其中的×××应该替换为用户自己所需要的字符串。

2. 初始化向量中断控制器

初始化向量中断控制器示意代码如下：

```
VICVectAddrX = (uint32) × × × _Handler ;
VICVectCntlX = (0x20 | Y);
VICIntEnable = 1 << Y;
```

其中 X 为分配给中断的优先级，Y 为中断的通道号。

至于时钟节拍服务中断服务程序的编写，除了在时钟节拍服务中断服务程序中必须调用函数 OSTimeTick()外，与普通中断服务没有什么区别，这里不再介绍。

8.3.3 一个基于 μC/OS - II 和 ARM 的应用程序实例

这是一个基于 μC/OS - II 和 ARM 应用程序实例。该应用程序运行后，用户每按一下按键 KEY1，则会使蜂鸣器发出两声鸣叫。这个应用程序中共有两个任务：Task1 和 Task2。用 Task1 作为启动任务并处理按键事务；用任务 Task2 来处理蜂鸣器的鸣叫。应用程序主要的代码清单如下：

```
//主函数代码-----------------------------------------------------------
#include "config.h"
OS_STK        TaskStartStk[TASK_STK_SIZE];        //任务 Task1 的任务堆栈
OS_STK        TaskStk[TASK_STK_SIZE];             //任务 Task2 的任务堆栈
int main (void)
{
    OSInit();                                     //初始化 μC/OS - II 系统
    OSTaskCreate(Task1,                           //创建任务 Task1
                 (void * )0,
                 &TaskStartStk[TASK_STK_SIZE - 1],
                 0);
    OSStart();                                    //启动 μC/OS - II 任务管理
    return 0;
}
//任务 Task1-----------------------------------------------------------
void  Task1(void * pdata)
{
    pdata = pdata;
    TargetInit();                                 //目标板初始化
    for ( ;; )
    {
        OSTimeDly(OS_TICKS_PER_SEC / 50);
        if (GetKey() != KEY1)
        {
```

```
            continue;                          //按键不是 KEY1,则不理会
        }
        OSTimeDly(OS_TICKS_PER_SEC / 50);      //延时 20 ms,用于按键去抖
        if (GetKey() != KEY1)                  //确认按键是否仍然是 KEY1
        {
            continue;
        }
        OSTaskCreate(Task2,                    //创建任务 Task2
                     (void *)0,
                     &TaskStk[TASK_STK_SIZE - 1],
                     10);
        while (GetKey() != 0)                  //等待按键被释放
        {
            OSTimeDly(OS_TICKS_PER_SEC / 50);  //延时 20 ms
        }
    }
}
//任务 Task2-----------------------------------------------------------
void  Task2(void * pdata)
{
    pdata = pdata;
    BeeMoo();                                  //使蜂鸣器鸣叫
    OSTimeDly(OS_TICKS_PER_SEC / 8);           //延时
    BeeNoMoo();                                //关闭蜂鸣器
    OSTimeDly(OS_TICKS_PER_SEC / 4);           //延时
    BeeMoo();                                  //使蜂鸣器鸣叫
    OSTimeDly(OS_TICKS_PER_SEC / 8);           //延时
    BeeNoMoo();                                //关闭蜂鸣器
    OSTaskDel(OS_PRIO_SELF);                   //任务删除自身
}
//---------------------------------------------------------------------
```

从以上代码中可以看到，应用程序用任务 Task1 来完成按键的识别、消抖工作，并在确认按键 KEY1 被按下时创建能使蜂鸣器鸣叫的任务 Task2；Task2 在完成蜂鸣器的两次鸣叫之后，就把任务自身删除。

8.4 小 结

- 为使在不同的 ARM 工作模式下调用 μC/OS - II 底层接口函数不受访问权限的限制，在移植时使用了 ARM 的软中断技术。
- 根据 ARM 核的特点和移植目标，移植时在 μC/OS - II 中增加了 4 个系统函数。其中，两个是用来转换处理器模式的函数 ChangeToSYSMode()和 ChangeToUSRMode()；两个是设置任务的初始指令集的函数 TaskIsARM()和 TaskIsTHUMB()。
- 为了使用户可用 C 语言编写中断服务程序时不必为处理器的硬件区别而困扰，根据 μC/OS - II 对中断服务程序的要求以及 ARM7 体系结构特点和 ADS 编译器特点，编写了一个适用于所有基于 ARM7 核处理器的汇编宏。这个宏实现了 μC/OS - II for ARM7 中断服务程序的汇编语言代码与 C 语言函数代码之间通用接口。

8.5 练习题

1. 移植 μC/OS - II 到 ARM7 为何使用 SWI 软件中断异常接口？
2. 写出 ADS 中关键字__swi 的具体用法。
3. 移植代码为何要增加 ChangeToSYSMode()、ChangeToUSRMode()、TaskIsARM()和 TaskIsTHUMB()这 4 个函数？它们如何使用？
4. 在 LPC2200 上编写一个简单的基于 μC/OS - II 的程序。

第9章 μC/OS－II 可裁剪性的实现

在实际应用程序的设计中，一般不会用到 μC/OS－II 系统提供全部函数，所以 μC/OS－II 允许用户根据实际需要对 μC/OS－II 进行裁剪，即只选用应用程序用到的功能，而不需要的功能则通过条件编译命令把它们裁剪掉（实质上就是令编译器不编译它们）。于是就需要在 μC/OS－II 的配置文件 OS_CFG. H 中，对相关的配置常量进行相应的设置。

本章的主要内容有：

- 介绍用户在设计应用程序时，为选用 μC/OS－II 提供的服务而需要在 OS_CFG. H 中定义的配置常量。

9.1 文件 OS_CFG. H 中用于系统裁剪的常量

本节介绍每个在配置文件 OS_CFG. H 中用"#define constant"定义的常量，介绍的顺序与它们在文件 OS_CFG. H 中出现的顺序相同。

编译系统在对应用程序进行编译时，会依据用户在配置文件中各配置常量的值有选择地进行编译，从而生成裁剪后的目标代码。

表 9－1 列出了用编译常量控制的 μC/OS－II 函数。

表 9－1 μC/OS－II 函数和相关的常量

函 数	使用该函数需要置 1 的常量	与该函数相关的其他常量
OSInit()	无	OS_MAX_EVENTS OS_Q_EN and OS_MAX_QS OS_MEM_EN OS_TASK_IDLE_STK_SIZE OS_TASK_STAT_EN OS_TASK_STAT_STK_SIZE
OSSchedLock()	无	无
OSSchedUnlock()	无	无

续表 9－1

函　数	使用该函数需要置 1 的常量	与该函数相关的其他常量
OSStart()	无	无
OSStatInit()	OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_EN	OS_TICKS_PER_SEC
OSVersion()	无	无
OSIntEnter()	无	无
OSIntExit()	无	无
OSMboxAccept()	OS_MBOX_EN	无
OSMboxCreate()	OS_MBOX_EN	OS_MAX_EVENTS
OSMboxPend()	OS_MBOX_EN	无
OSMboxPost()	OS_MBOX_EN	无
OSMboxQuery()	OS_MBOX_EN	无
OSMemCreate()	OS_MEM_EN	OS_MAX_MEM_PART
OSMemGet()	OS_MEM_EN	无
OSMemPut()	OS_MEM_EN	无
OSMemQuery()	OS_MEM_EN	无
OSQAccept()	OS_Q_EN	无
OSQCreate()	OS_Q_EN	OS_MAX_EVENTS OS_MAX_QS
OSQFlush()	OS_Q_EN	无
OSQPend()	OS_Q_EN	无
OSQPost()	OS_Q_EN	无
OSQPostFront()	OS_Q_EN	无
OSQQuery()	OS_Q_EN	无
OSSemAccept()	OS_SEM_EN	无
OSSemCreate()	OS_SEM_EN	OS_MAX_EVENTS
OSSemPend()	OS_SEM_EN	无
OSSemPost()	OS_SEM_EN	无
OSSemQuery()	OS_SEM_EN	无
OSTaskChangePrio()	OS_TASK_CHANGE_PRIO_EN	OS_LOWEST_PRIO

续表 9－1

函　数	使用该函数需要置 1 的常量	与该函数相关的其他常量
OSTaskCreate()	OS_TASK_CREATE_EN	OS_MAX_TASKS OS_LOWEST_PRIO
OSTaskCreateExt()	OS_TASK_CREATE_EXT_EN	OS_MAX_TASKS OS_STK_GROWTH OS_LOWEST_PRIO
OSTaskDel()	OS_TASK_DEL_EN	OS_LOWEST_PRIO
OSTaskDelReq()	OS_TASK_DEL_EN	OS_LOWEST_PRIO
OSTaskResume()	OS_TASK_SUSPEND_EN	OS_LOWEST_PRIO
OSTaskStkChk()	OS_TASK_CREATE_EXT_EN	OS_LOWEST_PRIO
OSTaskSuspend()	OS_TASK_SUSPEND_EN	OS_LOWEST_PRIO
OSTaskQuery()		OS_LOWEST_PRIO
OSTimeDly()	无	无
OSTimeDlyHMSM()	无	OS_TICKS_PER_SEC
OSTimeDlyResume()	无	OS_LOWEST_PRIO
OSTimeGet()	无	无
OSTimeSet()	无	无
OSTimeTick()	无	无
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

9.2　配置常量的解释

9.2.1　OS_MAX_EVENTS

配置常量 OS_MAX_EVENTS 用来定义应用系统中可以具有事件控制块的最大数量。一旦定义了常量 OS_MAX_EVENTS 的值，那么在应用程序中所应用事件的总数目就不能大于这个值。

例如，某应用系统中需要使用 10 个消息邮箱、5 个消息队列以及 3 个信号量，则 OS_MAX_EVENTS 至少应该定义为 18。

9.2.2 OS_MAX_MEM_PARTS

配置常量 OS_MAX_MEM_PARTS 用来定义系统中具有内存控制块的最大数量。

如果应用程序要使用动态内存，那么内存控制块的配置常数 OS_MAX_MEM_PARTS 最小应该设置为 2。为了使编译器能够对系统的内存管理函数进行编译，常量 OS_MEM_EN 也要置为 1。

9.2.3 OS_MAX_QS

配置常量 OS_MAX_QS 定义系统中具有消息队列的最大数目。

如果应用程序要使用消息队列，那么 OS_MAX_ QS 最小应该设置为 2。为了使编译器能够对系统的管理消息队列的函数进行编译，常量 OS_Q_EN 也要同时置 1。

9.2.4 OS_MAX_MEM_TASKS

配置常量 OS_MAX_MEM_TASKS 用来定义应用程序中可以具有任务的最大数目。由于 μC/OS－II 保留了两个系统使用的任务（统计任务和空闲任务），所以常量 OS_MAX_MEM_TASKS 的值不能大于 62。

如果设定 OS_MAX_MEM_TASKS 刚好等于所需任务数，则建立新任务时要注意检查是否超过限定。如果 OS_MAX_MEM_TASKS 的值设定得太大，则会造成内存的无谓浪费。

9.2.5 OS_LOWEST_PRIO

配置常量 OS_LOWEST_PRIO 用来设定系统中任务的最低优先级别（最大优先级数目）。

设定适当的 OS_LOWEST_PRIO 值，可以节省用于任务控制块的内存。

μC/OS－II 中优先级别从 0（最高优先级）～63（最低优先级）。μC/OS－II 保留了两个优先级别系统自用：OS_LOWEST_PRIO 和 OS_LOWEST_PRIO－1。其中 OS_LOWEST_PRIO 留给系统的空闲任务 OSTaskIdle()；OS_LOWEST_PRIO－1 留给统计任务 OSTaskStat()。从 0～OS_LOWEST_PRIO－2 为用户任务可使用的优先级别。

需要注意的是，OS_LOWEST_PRIO 和 OS_MAX_TASKS 是相互独立的两个配置常量。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，用户任务的优先级别可以是 0～30。当然，用 OS_LOWEST_PRIO 来设定的优先级别也要够用。例如，设 OS_MAX_TASKS 为 20，而 OS_LOWEST_PRIO 为 10，优先级别就不够用了。

9.2.6 OS_TASK_IDLE_STK_SIZE

配置常量 OS_TASK_IDLE_STK_SIZE 用来设置 μC/OS－II 中空闲任务堆栈的容量。

在设置这个常量时，要注意堆栈容量的单位不是字节，而是 OS_STK（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）。

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，以及可能出现的中断嵌套情况。

9.2.7 OS_TASK_STAT_EN

配置常量 OS_TASK_STAT_EN 用来设定系统是否使用 μC/OS－II 中的统计任务 OSTaskStat()及其初始化函数。若设为 1，则使用统计任务。统计任务每秒运行一次，计算当前系统 CPU 使用率并把结果保存在 8 位全局变量 OSCPUUsage 中。每次运行，OSTaskStat()都将调用函数 OSTaskStatHook()，用户自定义的统计功能可以放在这个函数中。

统计任务 OSTaskStat()的优先级总是自动设为 OS_LOWEST_PRIO－1。

当 OS_TASK_STAT_EN 设为 0 时，μC/OS－II 在系统初始化时，全局变量 OSCPUUsage、OSIdleCtrMax、OSIdleCtrRun 和 OSStatRdy 都将不被声明，以节省内存空间。

9.2.8 OS_TASK_STAT_STK_SIZE

配置常量 OS_TASK_STAT_STK_SIZE 用来设定 μC/OS－II 统计任务的任务堆栈容量。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，以及如下操作：

- 进行 32 位算术运算所需的堆栈空间；
- 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，判断实际的堆栈使用，则用户需要设 OS_TASK_CREATE_EXT_EN 为 1，并使用函数 OSTaskCreateExt()建立任务。

9.2.9 OS_CPU_HOOKS_EN

配置常量 OS_CPU_HOOKS_EN 用来设定是否在文件 OS_CPU_C.C 中实现各钩子函数（Hook Function）。如果要实现钩子函数，则常量 OS_CPU_HOOKS_EN 必须设置为 1。

μC/OS－II 中提供了如下 5 个对外接口函数：

- OSTaskCreateHook()；
- OSTaskDelHook()；
- OSTaskStatHook()；
- OSTaskSwHook()；
- OSTimeTickHook()。

这 5 个钩子函数既可在文件 OS_CPU_C.C 中声明，也可在用户代码中声明。

9.2.10 OS_MBOX_EN

配置常量 OS_MBOX_EN 用来控制应用程序是否使用 μC/OS－II 的消息邮箱函数及其相关数据结构。如果要使用，则必须将其设为 1；如果不使用，则设为 0，关闭此常量以节省内存。

9.2.11 OS_MEM_EN

配置常量 OS_MEM_EN 用来控制应用程序是否使用 μC/OS－II 的内存块管理函数及其相关数据结构。如果要使用，则必须将其设为 1；如果不使用，则设为 0，关闭此常量以节省内存。

9.2.12 OS_Q_EN

配置常量 OS_Q_EN 用来控制应用程序是否使用 μC/OS－II 中的消息队列函数及其相关数据结构。如果要使用，则必须将其设为 1；如果不使用，则设为 0，关闭此常量以节省内存。

9.2.13 OS_SEM_EN

配置常量 OS_SEM_EN 用来控制应用程序是否使用 μC/OS－II 中的信号量管理函数及其相关数据结构。如果要使用，则设为 1；如果不使用，则关闭此常量以节省内存。

9.2.14 OS_TASK_CHANGE_PRIO_EN

配置常量 OS_TASK_CHANGE_PRIO_EN 用来控制应用程序是否使用 μC/OS－II 的改变任务优先级别函数 OSTaskChangePrio()。如果要使用，则应将其设为 1；如果在应用程序中不需要改变运行任务的优先级，则应将此常量设置为 0，以节省内存。

9.2.15 OS_TASK_CREATE_EN

配置常量 OS_TASK_CREATE_EN 用来控制应用程序是否使用 μC/OS－II 的 OSTaskCreate()函数。如果使用，则应将其设置为 1；如果不使用，则应将其设置为 0，以节省内存。

在 μC/OS－II 中推荐用户使用函数 OSTaskCreateExt()来建立任务。

应该注意的是，OS_TASK_CREATE_EN 和 OS_TASK_CREATE_EXT_EN 至少有一个要为 1。

9.2.16 OS_TASK_CREATE_EXT_EN

配置常量 OS_TASK_CREATE_EXT_EN 用来控制是否使用 μC/OS－II 的函数 OSTaskCreateExt()。如果使用，则应将其设为 1；如果不使用，则应将常量其设置为 0，以节省内存。

注意，在应用程序中如果要使用堆栈检查函数 OSTaskStkChk()，则必须使用函数 OSTaskCreateExt()建立任务。

9.2.17 OS_TASK_DEL_EN

配置常量 OS_TASK_DEL_EN 用来控制是否使用 μC/OS－II 的删除任务函数 OSTaskDel()。如果使用，则应将其设为 1；如果不使用，则应将其设为 0，以节省内存。

9.2.18 OS_TASK_SUSPEND_EN

配置常量 OS_TASK_SUSPEND_EN 用来控制是否使用 μC/OS－II 的任务挂起和唤醒函数 OSTaskSuspend()和 OSTaskResume()。如果使用，则应将其设为 1；如果不使用，则应将其设为 0，以节省内存。

9.2.19 OS_TICKS_PER_SEC

配置常量 OS_TICKS_PER_SEC 用来表示调用 OSTimeTick()函数的次数。

在函数 OSStatInit()、OSTaskStat()和 OSTimeDlyHMSM()中都会用到 OS_TICKS_PER_SEC。用户需要在自己的初始化程序中保证 OSTimeTick()按所设定的频率调用。

附录 A　文件 PC.C 中的函数

为使 μC/OS-II 在 PC 机上运行时可充分使用 PC 机的一些功能，在文件 PC.H 和 PC.C 中提供了 3 类以"PC_"为前缀的功能函数：字符显示、运行时间测量及其他服务。这些函数封装了 PC 机的一些操作，可以直接在 μC/OS-II 中使用。为了节省篇幅，这里只介绍本书用到的一些函数。

A.1　字符显示函数

A.1.1　显示一个字符的函数 PC_DispChar()

函数原型如下：

```
void PC_DispChar (INT8U x,              //待显示字符在屏幕上横坐标 x
                  INT8U y,              //待显示字符在屏幕上纵坐标 y
                  INT8U c,              //待显示的字符
                  INT8U color           //字符颜色
                  );
```

A.1.2　清屏幕一列显示的函数 PC_DispClrCol()

函数原型如下：

```
void PC_DispClrCol (INT8U x,            //待清除列的横坐标 x
                    INT8U color         //填充颜色
                    );
```

A.1.3　清屏幕一行显示的函数 PC_DispClrRow()

函数原型如下：

```
void PC_DispClrRow ( INT8U y,           //待清行的纵坐标 y
                     INT8U color        //填充颜色
                     );
```

A.1.4 清屏函数 PC_DispClrScr()

函数原型如下：

```
void PC_DispClrScr (INT8U color          //填充颜色
                    );
```

A.1.5 显示字符串函数 PC_DispStr()

函数原型如下：

```
void PC_DispStr ( INT8U x,                //字符串起始位置的横坐标 x
                  INT8U y,                //字符串起始位置的纵坐标 y
                  INT8U * s,              //字符串指针
                  INT8U color             //字符串颜色
                  );
```

A.1.6 颜色常量的定义

颜色常量的定义如表 A－1 和 A－2 所列。

表 A－1 前景色的常量

常 量	颜 色	常 量	颜 色
DISP_FGND_BLACK	黑	DISP_FGND_DARK_GRAY	深灰
DISP_FGND_BLUE	蓝	DISP_FGND_LIGHT_BLUE	浅蓝
DISP_FGND_GREEN	绿	DISP_FGND_LIGHT_GREEN	浅绿
DISP_FGND_CYAN	青	DISP_FGND_LIGHT_CYAN	浅青
DISP_FGND_RED	红	DISP_FGND_LIGHT_REN	浅红
DISP_FGND_PURPLE	紫	DISP_FGND_LIGHT_PURPLE	浅紫
DISP_FGND_BROWN	褐	DISP_FGND_YELLOW	黄
DISP_FGND_LIGHT_GRAY	浅灰	DISP_FGND_WHITE	白

表 A－2 背景色的常量

常 量	颜 色	常 量	颜 色
DISP_BGND_BLACK	黑	DISP_BGND_RED	红
DISP_BGND_BLUE	蓝	DISP_BGND_PURPLE	紫
DISP_BGND_GREEN	绿	DISP_BGND_BROWN	褐
DISP_BGND_CYAN	青	DISP_BGND_LIGHT_GRAY	深灰

A.2 保存和恢复 DOS 环境的函数

在 PC 机上运行和调试基于 μC/OS-II 的应用程序时，运行 μC/OS-II 操作系统之前，需要在应用程序中先行保护 DOS 操作系统的环境；退出 μC/OS-II 时，还要恢复 DOS 环境。

A.2.1 保存 DOS 环境的函数 PC_DOSSaveReturn()

函数原型如下：

```
void PC_DOSSaveReturn (void);
```

A.2.2 恢复 DOS 环境的函数 PC_DOSReturn()

函数原型如下：

```
void PC_DOSReturn (void);
```

A.3 设置和获取中断向量的函数

A.3.1 设置中断向量的函数 PC_VectSet()

在 PC 机上运行 μC/OS-II 之前，要在 PC 的中断向量表中设置 μC/OS-II 任务切换的中断向量，这时要用到函数 PC_VectSet()。该函数的原型如下：

```
void PC_VectSet (INT8U vect,            //中断向量码
                 void (* isr)(void)     //中断服务程序指针
                 );
```

A.3.2 获取中断向量的函数 PC_VectGet()

函数原型如下：

```
void * PC_VectGet (INT8U vect);
```

附录 B　μC/OS－II 中使用的数据类型

```
typedef unsigned    char    BOOLEAN;
typedef unsigned    char    INT8U;      /*8 位无符号整数*/
typedef signed      char    INT8S;      /*8 位有符号整数*/
typedef unsigned    int     INT16U;     /*16 位无符号整数*/
typedef signed      int     INT16S;     /*16 位有符号整数*/
typedef unsigned    long    INT32U;     /*32 位有符号整数*/
typedef signed      long    INT32S;     /*32 位有符号整数*/
typedef float               FP32;       /*单精度浮点数*/
typedef double              FP64;       /*双精度浮点数*/
```

附录 C　μC/OS－II 在 80x86 上的移植

本书第 7 章之前的所有例题都是在 PC 中运行的，更准确地说，是在移植到 80x86 系列微处理器的 μC/OS－II 平台上运行的。因此，在应用程序的主函数中，除了要用函数 OSInit() 初始化的环境之外，还要使用 PC_VectSet(uCOS, OSCtxSw) 来安装 μC/OS－II 任务切换的中断向量；而在起始任务的代码中要使用 PC_VectSet(0x08, OSTickISR) 来安装 μC/OS－II 系统时钟中断向量，并且要使用 PC_SetTickRate(OS_TICKS_PER_SEC) 来设置 μC/OS－II 系统时钟周期（见下面代码中的黑体文字部分）。

```
void  main (void)
{
      ……
    OSInit();
    PC_DOSSaveReturn();                          //保存 DOS 环境
    PC_VectSet(uCOS, OSCtxSw);                   //安装 μC/OS－II 中断向量
    OSTaskCreate(……);
    OSStart();
}


void  MyTask (void * pdata)
{
    ……
    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                 //安装时钟中断向量
    PC_SetTickRate(OS_TICKS_PER_SEC);            //设置时钟频率
    OS_EXIT_CRITICAL();
    ……
    for (;;)
    {……}
}
```

为了使读者充分理解上述代码的意义，本附录将简单介绍把 μC/OS－II 移植到 80x86 上时的有关部分。

C.1 任务切换函数 OSCtxSw()

OSCtxSw()是任务级的任务切换函数，它被 μC/OS－II 任务调度器 OSSched()在任务切换宏 OS_TASK_SW()中调用，其工作就是保存被中止运行任务的断点和恢复待运行任务的断点并启动它。

C.1.1 任务切换函数 OSCtxSw()

任务切换函数 OSCtxSw()的工作就是保存被中止运行任务的断点和恢复待运行任务的断点并启动它。因此，在这个函数中主要是一系列的压栈和出栈操作。由于 C 语言没有对 CPU 寄存器操作的指令，所以该函数只能用汇编语言来编写。它在文件 OS_CPU_A. ASM 中的代码如下：

```
_OSCtxSw    PROC    FAR
;
            PUSHA;------------------------------------------------ (1)
            PUSH    ES
            PUSH    DS
;
            MOV     AX, SEG _OSTCBCur
            MOV     DS, AX
;
            LES     BX, DWORD PTR DS:_OSTCBCur;------------------------------ (2)
            MOV     ES:[BX + 2], SS
            MOV     ES:[BX + 0], SP
;
            CALL    FAR PTR _OSTaskSwHook;---------------------------------- (3)
;
            MOV     AX, WORD PTR DS:_OSTCBHighRdy + 2;------------------------- (4)
            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;----------------------------(5)
```

```
            MOV     BYTE PTR DS:_OSPrioCur, AL
;
            LES     BX, DWORD PTR DS:_OSTCBHighRdy;----------------------------(6)

            MOV     SS, ES:[BX+2]
            MOV     SP, ES:[BX]
;
            POP     DS;--------------------------------------------------------(7)
            POP     ES
            POPA
;
            IRET;------------------------------------------------------------(8)
;
_OSCtxSw    ENDP
```

下面按代码后面的标号，将代码简单介绍如下：

(1) 把被中止运行任务的在 CPU 其他寄存器中的断点数据保存到当前任务堆栈之中。

(2) 把指向新的堆栈结构的指针保存到当前任务控制块 TCB 中。

(3) 调用钩子函数 OSTaskSwHook()。应用程序设计人员可以在其中编写一些自己的代码。

(4) 把待运行任务的任务控制块 TCB 复制到 OSTCBCur 中。

(5) 把待运行任务的优先级别 Prio(任务的标识)复制到全局变量 OSPrioCur 中。

(6) 恢复待运行任务在上一次被中止运行时所保存的任务堆栈指针。

(7) 从待运行任务的任务堆栈中把断点数据恢复到 CPU 的各个寄存器。

(8) 中断返回，并恢复待运行任务的程序计数器和程序状态字。这条指令被执行后，CPU 即按当前程序计数器的指向运行待运行的新任务。

从上述代码中可以看到，任务切换函数 OSCtxSw()实现任务切换是靠中断返回指令 IRET 来实现的，因此 OSCtxSw()应该是一个中断服务程序。

C.1.2 任务切换宏 OS_TASK_SW()

前面提到，进行任务切换的关键是要实现任务断点指针的切换，这个工作是要靠中断技术来实现的。因此，任务切换宏 OS_TASK_SW()的全部工作就是想办法主动引发一次中断，然后在中断服务程序中完成任务切换的其他工作。由于 80x86 具有内部中断(软中断)指令，因此在宏中引发一次中断是相当容易的，即在宏中封装内部中断指令 INT 即可。在文件 OS_CPU.H 中有关任务切换宏 OS_TASK_SW()的代码如下：

```
#define  uCOS            0x80                    //定义常量 uCOS 并且值为 0x80
#define  OS_TASK_SW()  asm  INT   uCOS           //引发 INT 0x80 内部中断
```

由此可见，在任务调度函数 OSSched()中调用一次任务切换宏 OS_TASK_SW()就执行了一次指令：

```
INT  0x80
```

即引发了一个内部中断，中断向量码为 0x80。

之所以选择中断向量码为 0x80，是因为在 80x86 系统中 DOS 和 BIOS 都没有用到它是一个允许用户使用的向量码(大多数如此，不过如果用户在 PC 机上使用 μC/OS－II，最好先查看一下系统的中断向量表，以确定该向量码真的没被使用；如果已被占用，那么只好再寻找一个未被使用的中断向量码)。

任务调度器 OSSched()、任务切换宏 OS_TASK_SW()、内部中断指令 INT 0x80 和任务切换函数(中断服务程序)之间的关系见图 C－1。

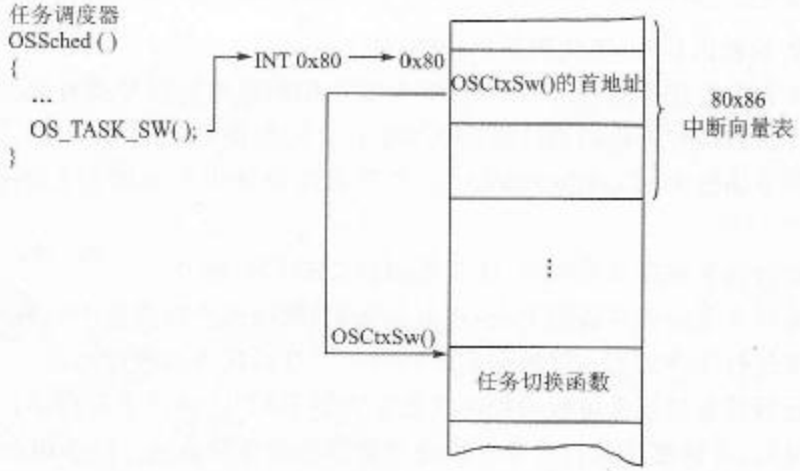


图 C－1　任务切换宏与任务切换函数之间的关系

既然 μC/OS－II 的任务切换是在中断中完成的，那么应用程序就应在创建任务和调用系统函数 OSStart()启动 μC/OS－II 之前，向系统的中断向量表安装任务切换函数 OSCtxSw()的中断向量。安装 μC/OS－II 中断向量的代码如下：

```
PC_VectSet(uCOS, OSCtxSw);                    //安装 μC/OS－II 中断向量
```

其中参数 uCOS 为中断向量码；OSCtxSw 为中断向量。

C.1.3　中断级任务切换函数 OSIntCtxSw()

μC/OS－II 规定，在所有中断服务程序将要结束时都要调用中断服务函数 OSIntExit()。

该函数的源代码如下：

```
void  OSIntExit (void)
{
        ......
      if ((OSIntNesting == 0) && (OSLockNesting == 0))
      {
       ......
            {
            ......
            OSIntCtxSw();
            }
      }
       ......
}
```

从退出中断服务函数 OSIntExit()的源代码中可以看到，该函数要调用中断时执行任务切换的函数 OSIntCtxSw()，这个函数也是用汇编语言编写在文件 OS_CPU_A. ASM 中的。

由于这个函数用在中断服务程序的末尾，保护被中止运行程序断点的工作已经在中断服务程序的前段完成了，因此它的工作只是恢复待运行任务的断点。

函数 OSIntCtxSw()的代码如下：

```
_OSIntCtxSw  PROC   FAR
;
             CALL   FAR PTR _OSTaskSwHook;
             MOV    AX, SEG _OSTCBCur
             MOV    DS, AX
;
             MOV    AX, WORD PTR DS:_OSTCBHighRdy + 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
             MOV    BYTE PTR DS:_OSPrioCur, AL
;
             LES    BX, DWORD PTR DS:_OSTCBHighRdy
```

```
                MOV         SS, ES:[BX + 2]
                MOV         SP, ES:[BX]
;
                POP         DS
                POP         ES
                POPA
;
                IRET
;
_OSIntCtxSw ENDP
```

C.2 μC/OS－II 的系统时钟

在 PC 机上移植和运行 μC/OS－II 要注意其与 PC 中其他操作系统的关系(主要是与 DOS 操作系统之间的关系)。不能因为安装了 μC/OS－II 而使其他系统不能正常使用。

C.2.1 PC 中 DOS 的系统时钟

在 PC 中,DOS 系统时钟的周期为 54.93 ms(18.206 48 Hz)。

DOS 操作系统的系统时钟是由 PC 上配置的硬件定时器产生的。在硬件定时器的每个定时周期结束时,硬件定时器便向 CPU 申请一次中断,在中断服务程序中处理成 DOS 所需要的系统时钟,同时处理 DOS 系统有关时间的一些管理事务。在 80x86 系统的中断向量表中,产生 DOS 系统时钟的硬件定时器所占用的中断向量码为 0x08。

C.2.2 PC 中 μC/OS－II 的系统时钟

在把 μC/OS－II 向 80x86 移植时,根据实时系统的要求,把 μC/OS－II 的系统时钟的频率定为 200 Hz,显然频率比 DOS 的时钟频率高得多。因此借用产生 DOS 系统时钟的定时器的定时信号来产生 μC/OS－II 系统时钟。即截取产生 DOS 系统时钟的硬件定时器的定时中断,先行产生 μC/OS－II 系统时钟,然后对产生 μC/OS－II 时钟进行 11 分频再产生 DOS 的时钟。即

$$\text{DOS 的时钟频率}=\frac{200\ \text{Hz}}{11}=18.181\ 818\ 18\ \text{Hz}$$

取 18.206 48 Hz,以不影响 DOS 的时钟和 DOS 的运行。

为此,μC/OS－II 的应用程序应该在适当的时机,把 μC/OS－II 时钟中断服务程序的中断向量 OSTickISR 安装到 80x86 中断向量表里,而且该位置的中断向量码应该为 0x08,而把

0x08 中原来的 DOS 的时钟中断向量转存到中断向量表的另一个空白位置 0x81 中。这个工作是由 PC 功能函数 PC_DOSSaveReturn()来执行的。PC_DOSSaveReturn()功能函数的源代码如下：

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

    PC_ExitFlag = FALSE; //-----------------------------------------(1)
    OSTickDOSCtr = 1;    //-----------------------------------------(2)
    PC_TickISR = PC_VectGet(VECT_TICK);    //----------------------------(3)

    PC_VectSet(VECT_DOS_CHAIN,PC_TickISR);    //----------------------------(4)

    setjmp(PC_JumpBuf);//-----------------------------------------(5)
    if (PC_ExitFlag == TRUE)    //-----------------------------------(6)
    {
        OS_ENTER_CRITICAL();
        PC_SetTickRate(18);//-----------------------------------------(7)
        OS_EXIT_CRITICAL();
        PC_VectSet(VECT_TICK, PC_TickISR);    //----------------------------(8)
        PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);//-------------------(9)
        exit(0);   //-----------------------------------------------(10)
    }
}
```

下面按上面代码中的标号，对 PC_DOSSaveReturn()函数进行说明。

(1) 设定返回 DOS 的标志 PC_ExitFlag 为 FALSE，即不返回 DOS 环境。

(2) 把计数器 OSTickDOSCtr 初值设置为 1。

(3) 获得 DOS 时钟中断向量。其中，常量 VECT_TICK＝0x08。

(4) 把 DOS 时钟中断向量转移到中断向量表的 VECT_DOS_CHAIN 位置。其中常量 VECT_DOS_CHAIN＝0x81。

(5) 调用函数 setjmp()把 CPU 寄存器中内容保存到结构体 PC_JumpBuf 中。

(6) 判断 PC_ExitFlag，由于 PC_ExitFlag 为 FALSE，因此函数返回。

C.2.3 μC/OS－II 系统时钟中断服务程序

μC/OS－II 系统时钟中断服务程序一般应该用汇编语言编写，其示意性代码如下：

```
void OSTickISR(void)
{
    保护断点；
    OSIntNesting++;
    If(OSIntNesting==1)
    {
        OSTCBCur->OSTCBStkPtr=SS:SP;
    }
    OSTickDOSCtr--;
    If(OSTickDOSCtr==0)
    {
        OSTickDOSCtr=11;
        INT 81H;
    }
    else
    {
        把 EOI 送往 PIC;
    }
    OSTimeTick();
    OSINTExit();
    恢复断点；
    中断返回；
}
```

从上面黑体字的代码段中可以看到，除了处理 μC/OS－II 的时间管理事务之外，每发生 11 次 μC/OS－II 系统时钟中断，μC/OS－II 系统的时钟中断服务程序 OSTickISR()都用"INT 81H"指令引发一次 DOS 系统时钟的中断，以让 DOS 处理自己的时间事务，而在其他次数只简单地用 EOI 指令清除 PC 机中断控制器的中断标志。

C.2.4 在任务中安装 μC/OS－II 系统时钟的中断向量

由上可知，在运行 μC/OS－II 之前必须调用 PC 功能函数 PC_DOSSaveReturn()，中断向量 0x08 的位置腾空，以便安装 μC/OS－II 的时钟中断向量。而安装 μC/OS－II 时钟中断向量的工作，一般是在 μC/OS－II 运行的第一个任务(起始任务)中用如下代码来完成的：

```
PC_VectSet(0x08, OSTickISR);                    //安装时钟中断向量
PC_SetTickRate(OS_TICKS_PER_SEC);               //设置时钟频率
```

并紧接着用 PC 功能函数来设置 μC/OS－II 的系统时钟频率。

DOS 环境与 μC/OS－II 环境下 PC 机的中断向量表之间的比较如图 C－2 所示。

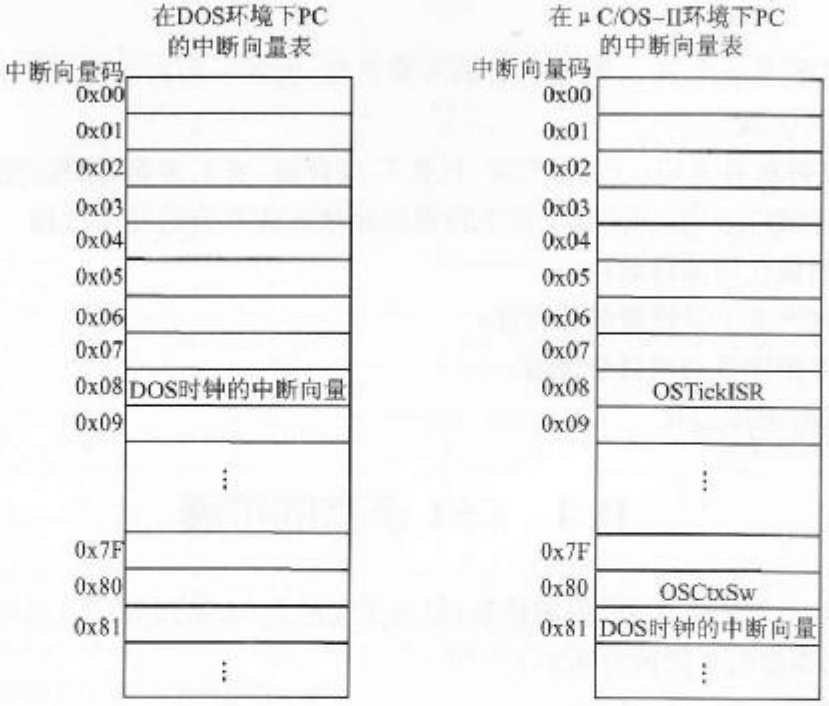


图 C－2　DOS 环境与 μC/OS－II 环境下 PC 中断向量表的比较

C.2.5　在任务中由 μC/OS－II 返回 DOS 环境

当 μC/OS－II 的任务结束需要使 PC 机返回 DOS 时，要调用 PC 功能函数 PC_DOSReturn()。该函数的源代码如下：

```
void PC_DOSReturn (void)
{
    PC_ExitFlag = TRUE;
    longjmp(PC_JumpBuf, 1);
}
```

附录 D　C51 开发工具 μVision2 简介

μVision2 IDE 是一个基于 Windows 的开发平台，包含一个高效的编辑器、一个项目管理器和一个 MAKE 工具。

μVision2 支持所有 KEIL 8051 工具，包括 C 编译器、宏汇编器、链接/定位器以及目标代码到 HEX 的转换器。μVision2 通过以下特性加速嵌入式系统的开发过程：

- 全功能的源代码编辑器；
- 用来配置开发工具设置的器件库；
- 创建和维护项目的项目管理器；
- 集成的 MAKE 工具。

D.1　C51 语言的扩展

虽然 C51 是一个兼容 ANSI 的编译器，但为了支持 8051 系列 MCU，还是加入了一些扩展的内容。C51 编译器的扩展内容包括：

- 数据类型；
- 存储器类型；
- 指针；
- 重入函数；
- 中断服务程序；
- 实时操作系统。

D.1.1　数据类型

本 C51 编译器支持表 D-1 列出的各种规格的数据类型。除了这些数据类型以外，变量可以组合成结构、联合及数组。除非特别说明，这些变量都可以用指针存取。

这些 sbit、sfr 和 sfr16 类型的数据使用户能够操作 8051 系列 MCU 所提供的特殊功能寄存器。例如：

```
sfr P0 = 0x80; /* Define 8051 P0 SFR */
```

表 D-1 数据类型

数据类型	位 数	字节数	数值范围
bit	1		0、1
signed char	8	1	−128～127
unsigned char	8	1	0～255
enum	16	2	−32 768～32 767
signed short	16	2	−32 768～32 767
unsigned short	16	2	0～65 535
signed int	16	2	−32 768～32 767
unsigned int	16	2	0～65 535
signed long	32	4	−2 147 483 648～2 147 483 647
unsigned long	32	4	0～4 294 967 295
float	32	4	±1.175 494E−38～±3.402 823+38
sbit	1		0、1
sfr	8	1	0～255
sfr16	16	2	0～65 535

注：bit、sbit、sfr 和 sfr16 为 8051 硬件、C51 及 C251 编译器所特有，它们不是 ANSI C 的一部分，也不能用指针对它们进行存取。

D.1.2 存储器类型

本 C51 编译器支持 8051 及其派生类型的结构，能够访问 8051 的所有存储器空间。具有表 D-2 列出的存储器类型的变量都可被分配到某个特定的存储器空间。

表 D-2 变量的存储器类型

存储器类型	说 明
code	程序空间(64 KB)；通过"MOVC @A+DPTR"访问
data	直接访问的内部数据存储器；访问速度最快(128 字节)
idata	间接访问的内部数据存储器；可以访问所有内部存储器空间(256 字节)
bdata	可位寻址的内部数据存储器；可以字节或位方式访问(16 字节)
xdata	外部数据存储器(64 KB)；通过"MOVX @DPTR"访问
pdata	分页的外部数据存储器(256 字节)；通过"MOVX @Rn"访问

由于访问内部数据存储器将比访问外部数据存储器快得多，因此应该把频繁使用的变量尽可能地放置在内部数据存储器中。变量的存储类型在定义变量时定义，例如：

```
char data var1;
char code text[] = "ENTER PARAMETER";
unsigned long xdata array[100];
float idata x,y,z;
unsigned int pdata dimension;
unsigned char xdata vector[10][4][4];
char bdata flags;
```

如果在变量的定义中没有定义存储器类型，系统将自动选用默认的存储器类型。默认的存储器类型适用于所有全局变量和静态变量，还有不能分配在寄存器中的函数参数和局部变量。默认的存储器类型由编译器的参数 SMALL、COMPACT 及 LARGE 决定。

D.1.3 存储模式

存储模式决定了默认的存储器类型，此存储器类型将应用于函数参数、局部变量和定义时未包含存储器类型的变量。

1. SMALL 模式

所有变量都默认在 8051 的内部数据存储器中。这与用 data 显式定义变量起到相同的作用。

2. COMPACT 模式

此模式中，所有变量都默认在 8051 的外部数据存储器的一页中。地址的高字节往往通过 Port 2 输出，其值必须在程序的初始化代码中设置，这与用 pdata 显式定义变量起到相同的作用。此模式最多只能提供 256 字节的变量。

3. LARGE 模式

在 LARGE 模式下，所有变量都默认在外部存储器中(xdata)。这与用 xdata 显式定义变量起到相同的作用。数据指针 DPTR 用来寻址。通过 DPTR 进行存储器的访问的效率很低，特别是在对一个大于 1 字节的变量进行操作时尤为明显。此数据访问类型比 SMALL 和 COMPACT 模式需要更多的代码。

4. 存储模式的设置

在 KEIL 开发环境中，选择菜单 Project|Options for Target 'XXXX'选项，在对话框的 Target 选项卡的 Memory Model 下拉列表中进行选择，如图 D－1 所示。

D.1.4 指　针

C51 编译器支持用星号(*)进行指针声明，并可用指针完成在标准 C 语言中的所有操作。

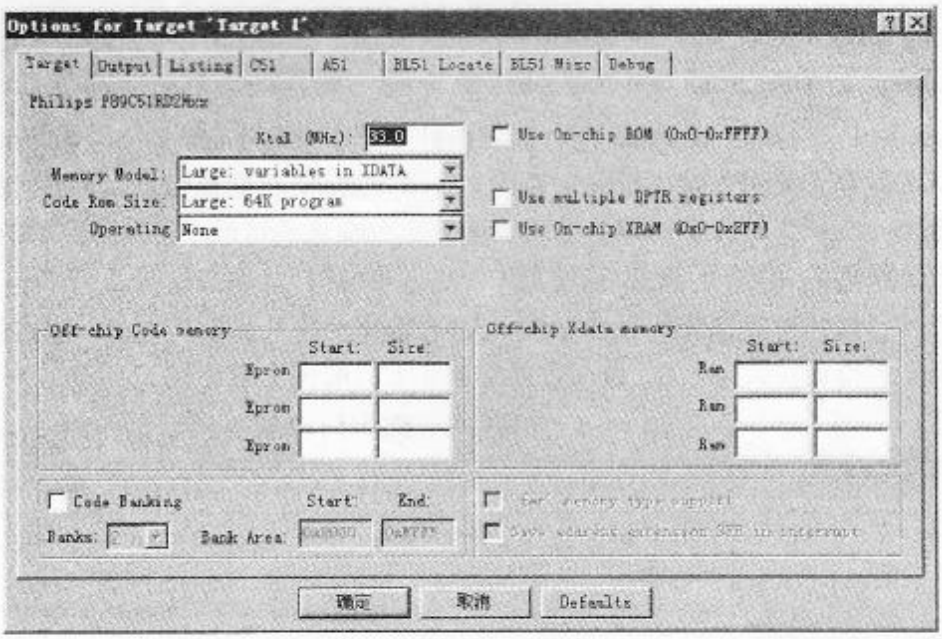


图 D-1 Options for Target 对话框

但是，由于 8051 及其派生系列所具有的独特结构，C51 编译器支持两种不同类型的指针：存储器指针和通用指针。

1. 通用指针

通用或未定型的指针的声明与标准 C 语言中一样。例如：

```
char * s;          /* string ptr */
int * numptr;      /* int ptr */
long * state;      /* long ptr */
```

通用指针总是需要 3 字节来存储：第 1 字节用来表示存储器类型；第 2 字节是指针的高字节；第 3 字节是指针的低字节。

通用指针可用来访问所有类型的变量，而不管变量存储在哪个存储空间中，因此许多库函数都使用通用指针。通过使用通用指针，一个函数可以访问数据，而不用考虑它存储在什么存储器中。

通用指针很方便，但是也很慢。在所指向目标的存储空间不明确的情况下，它们用得最多。

2. 存储器指针

存储器指针在定义时，需要用一个存储器类型进行说明，定义后该指针总是指向这个存储类型说明的特定存储器空间。例如：

```
char data *str;          /* 指向内部 RAM */
int xdata *numtab;       /* 指向外部 RAM */
long code *powtab;       /* 指向程序存储区 */
```

指向 idata、data、bdata 和 pdata 的存储器指针用 1 字节保存；指向 code 和 xdata 的存储器指针用 2 字节保存。使用存储器指针比通用指针效率要高，速度要快。当然，存储器指针的使用不是很方便。在所指向目标的存储空间明确并不会变化的情况下，它们用得最多。

3. 存储器指针和通用指针的比较

使用存储器指针可以显著地提高 8051 C 语言程序的运行速度。表 D－3 中的示例程序说明了使用不同的指针在代码长度、占用数据空间和运行时间上的不同。

表 D－3　不同存储类型指针的比较

项　目	idata 类型指针	xdata 类型指针	通用指针
C 源程序	idata *ip; char val; val = *ip;	char xdata *xp; har val; val = *xp;	char *p; char val; val = *xp;

编译后的代码	MOV R0,ip
	MOV val,@R0
		MOV DPL,
		xp +1 MOV DPH,xp
		MOV A,@DPTR
		MOV val,A
			MOV R1,p + 2
			MOV R2,p + 1
			MOV R3,p
			CALL CLDPTR

指针大小	1 byte	2 bytes	3 bytes
代码长度	4 bytes	9 bytes	11 bytes + library call
执行时间	4 cycles	7 cycles	13 cycles

D.1.5　可重入函数

可重入函数是多个进程可以同时调用函数，即当一个重入函数被调用运行时，另外一个进程可能中断此运行过程，然后再次调用此重入函数。

通常情况下，由于 C51 函数的参数和局部变量是存储在固定的地址单元中，所以 C51 编译的函数是不可重入的。若需要一个重入函数，则在函数声明时要在函数的后面用 reentrant 关键字来修饰。例如：

```
int calc (char i, int b) reentrant
{
    int x;
    x = table[i];
    return (x * b);
}
```

对每一个重入函数来说，根据存储模式，重入堆栈被安置在内部或外部单元中。

D.1.6 与汇编语言的接口

可以在 C 语言程序中调用汇编语言程序，反之亦然。函数参数通过 CPU 寄存器传递，或使用 NOREGPARMS 参数指示编译器通过固定的存储器传递。从函数返回的值总是通过 CPU 寄存器传递。除了直接产生目标代码外，还可用 SRC 编译参数指示编译器产生汇编源代码文件(供 A51 汇编器使用)。例如下面的 C 语言源代码：

```
unsigned int asmfunc1 (unsigned int arg)
{
    return (1 + arg);
}
```

用 SRC 指示 C51 编译器编译后，会产生以下汇编文件：

```
? PR? _asmfunc1? ASM1      SEGMENT CODE
PUBLIC              asmfunc1
        RSEG      ? PR? _asmfunc1? ASM1
        USING 0
 asmfunc1:
; ---- Variable 'arg? 00' assigned to Register 'R6/R7' ----
        MOV A,R7            ; load LSB of the int
        ADD A,#01H          ; add 1
        MOV R7,A            ; put it back into R7
        CLR A
        ADDC A,R6           ; add carry & R6
        MOV R6,A
? C0001:
        RET                 ; return result in R6/R7
```

也可用 #pragma asm 和 #pragma endasm 预处理指示器在 C 语言程序中插入汇编指令。

D.1.7 库函数

C51 编译器包含有 ANSI 标准的 7 个不同的编译库，从而满足不同功能的需要，如表 D-4 所列。

表 D－4 库函数

库文件	说 明	库文件	说 明
C51S. LIB	小模式库，不支持浮点运算	C51L. LIB	大模式库，不支持浮点运算
C51FPS. LIB	小模式库，支持浮点运算	C51FPL. LIB	大模式库，支持浮点运算
C51C. LIB	紧凑模式库，不支持浮点运算	80C751. LIB	Philips 8xC751 及其派生系列使用的库
C51FPC. LIB	紧凑模式库，支持浮点运算		

注：与硬件相联系的输入/输出操作的库函数模块的源代码文件位于\KEIL\C51\LIB 文件夹中。可利用这些文件来修改库，以适应目标板上的任何器件的输入/输出操作。

D. 2 创建项目

在 μVision2 创建一个应用项目的步骤如下：

① 启动 μVision2，新建一个项目文件并从器件库中选择一个器件；

② 新建一个源文件并把它加入到项目中；

③ 增加并配置选择的器件的启动代码；

④ 针对目标硬件设置工具选项；

⑤ 编译项目并生成可以编程 PROM 的 HEX 文件。

D. 2. 1 启动 μVision2 并创建一个项目

μVision2 是一个标准 Windows 应用程序，直接双击程序图标就可以启动它。要新建一个项目文件，选择 Project|New Project 选项，在打开的标准对话框输入项目文件名和存储目录。

选择 Project |Select Device for Target 选项，为项目选择要使用的 CPU 型号。在弹出的对话框的列表框中为器件数据库，如图 D－2 所示。用户要在这个数据库中选择所需要的 MCU。

一旦从器件库中选择了一个 CPU，就可以在项目窗口的 Books 页打开此 CPU 的用户手册。这些用户手册是 KEIL 开发工具(光盘)中的一部分。

D. 2. 2 新建一个源文件

选择 File|New 选项，将打开一个空的编辑窗口，让用户输入源代码。当把此文件另存为 *.C 的文件后，μVision2 将高亮显示 C 语言语法字符。

一旦创建了源文件，就可以把它加入到项目中。把源文件加入到项目中的一般方法是，右击 Project 窗口 Files 页中的文件组，弹出快捷菜单，选择菜单中的 Add Files 选项，打开一个标准的文件对话框，在对话框中选择要加入的文件。

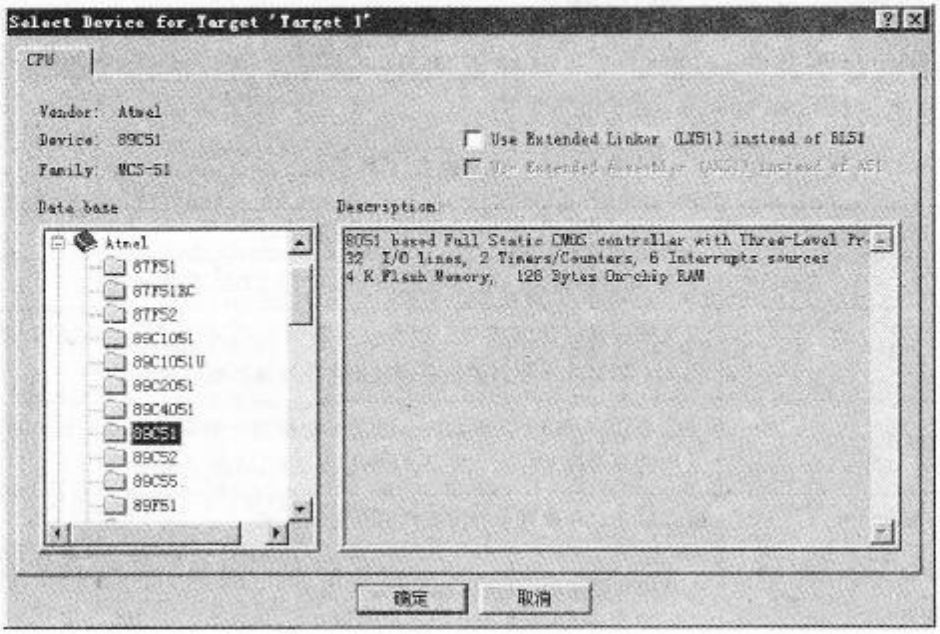


图 D-2 Select Device for Target 对话框

D.2.3 增加和配置初始化代码

文件 STARTUP.A51 是为大多数不同的 8051 CPU 准备的启动代码。启动代码清除数据存储器并初始化硬件和重入函数堆栈指针。另外，一些 8051 派生产品要求初始化 CPU 来适应设计中的相应硬件。例如，Philips 公司的 8051RD+提供的片上 xdata RAM 应该在启动代码中启用。假如需要修改启动文件来适应目标硬件，则应把文件 STARTUP.A51 复制一份到项目文件夹中。

为选择的 CPU 的配置文件创建一个文件组是一个良好的习惯。用 Project | Targets, Groups, Files…打开对话框来添加一个名为 System Files 的文件组到目标中。在此对话框中，单击 Add Files to Group…按钮，把文件 STARTUP.A51 添加到项目中。项目窗口的文件页列出了项目的所有条目。

在项目窗口中双击文件名 STARTUP.A51，就可在编辑器中打开这个文件，并配置初始化代码。

如果使用所选择器件的片上 RAM，在启动代码中的设置必须匹配 Options-Target 对话框中的设置。

D.2.4 为目标设置工具选项

在对话框的各个页中，可以定义与目标硬件及所选器件的片上元件相关的所有参数。表 D－5 说明了目标对话框的一些选项。

表 D－5 目标设置工具选项

选 项	说 明
Xtal	定义 CPU 时钟，对于大多数应用与实际的 XTAL 频率相同
Memory Model	定义编译器的存储模式。对于一个新的应用，默认的是 SMALL 模式。参见 D.1.2 和 D.1.3 小节关于存储模式和存储器类型的讨论
Allocate On-chip Use multiple DPTR registers	定义在启动代码中使能的片上元器件的使用。如果要使用片上 xdata RAM，那么应该在文件 STARTUP.A51 中使能 XRAM 的访问
Off-chip … Memory	在此定义目标硬件上的所有外部存储器区域
Code Banking Xdata Banking	为代码和数据的分段(Banking)定义参数

注：有些选项只有在使用 LX51 链接器时才有用。LX51 链接器只在 PK51 中提供。

D.2.5 创建项目并生成 HEX 文件

通常情况下，在 Options Target 对话框中的设置已经足够开始设计应用程序了。通过单击工具条上的 Build 目标的图标，可以编译所有源文件并生成应用。当程序中有语法错误时，μVision2 将在 Output Window Build 页显示这些错误和告警信息。双击一个信息，将打开此信息对应的文件并定位到语法错误处。

在调试完应用程序后，需要创建一个 HEX 文件来烧录片子或软件模拟。当 Options for Target | Output 中的 Create HEX Fi 选项被选中时，μVision2 每进行一次 Build 都生成 HEX 文件。如果选项 Options for Target|Output 中的 Run User Program #1 也被选中而且已定义，那么在生成 HEX 文件后，将自动运行此处定义的操作，例如对 PROM 器件编程。

D.3 常用的菜单选项

D.3.1 视图菜单 View

视图菜单部分选项及说明见表 D－6。

表 D-6 视图菜单部分选项

菜单选项	说 明
Disassembly Window	显示/隐藏反汇编窗口
Watch & Call Stack Window	显示/隐藏观察和堆栈窗口
Memory Window	显示/隐藏存储器窗口
Code Coverage Window	显示/隐藏代码报告窗口
Analyzer Window	显示/隐藏性能分析窗口
Symbol Window	显示/隐藏字符变量窗口
Serial Window #1	显示/隐藏串口 1 的观察窗口

D.3.2 项目菜单 Project

项目菜单部分选项及说明见表 D-7。

表 D-7 项目菜单部分选项

菜单选项	说 明
New Project…	创建新项目
Target Environment	定义工具、包含文件和库的路径
Build Target	编译修改过的文件并生成应用
Rebuild Target	重新编译所有文件并生成应用
Translate…	编译当前文件
Stop Build	停止生成应用的过程

D.3.3 调试菜单 Debug

调试菜单部分选项及说明见表 D-8。

表 D-8 调试菜单部分选项

菜单选项	说 明
Start/Stop	开始/停止调试模式
Go	运行程序，直到遇到一个断点
Step	单步执行程序，遇到子程序则进入

续表 D－8

菜单选项	说　明
Step over	单步执行程序，跳过子程序
Stop Running	停止程序运行
Step out of	执行到当前函数的结束
Insert/Remove Breakpoint	设置/取消当前行的断点
Enable/Disable Breakpoint	使能/禁止当前行的断点
Disable All Breakpoints	禁止所有断点
Kill All Breakpoints	取消所有断点
Show Next Statement	显示下一条指令
Enable/Disable Trace Recording	使能/禁止程序运行轨迹的标识
View Trace Records	显示程序运行过的指令

D.3.4　外围器件菜单 Peripherals

外围器件菜单部分选项及说明见表 D－9。

表 D－9　外围器件菜单部分选项

菜单选项	说　明
Reset CPU	复位 CPU
Interrupt	打开片上外围器件的设置对话框 对话框的种类及内容依赖于选择的 CPU
I/O-Ports	打开 I/O 并行接口信息对话框
Serial	打开串行接口信息对话框
Timer	打开定时器信息对话框

附录 E　MagicARM2200 教学实验开发平台简介

MagicARM2200 实验箱是由广州周立功单片机发展有限公司开发的一款可使用 μC/OS－II 和 μCLinux 双操作系统、集众多功能于一身的 ARM 教学实验开发平台。

实验箱的主板上带有充足的存储资源(PSRAM、NAND FLASH、NOR FLASH 和 EEPROM 等),具有以太网接口、MODEM 接口、CAN 接口、IDE 硬盘接口、CF 卡接口、SD 卡接口、打印机接口,5.2 英寸 320×240 彩色液晶屏(带触摸屏),可使用 JTAG 仿真调试。灵活的跳线选择(I/O 与功能电路的连接)、外设 PACK 和 GPIO 输出接口,非常适用于教学实验。其主要功能特点如下:

- 支持 μCLinux、μC/OS－II 双操作系统,搭建用户系统平台更方便。
- 完全自主设计的软硬件,拥有自主版权的 JTAG 仿真技术,支持 ADS1.2 集成开发环境及 PHILIPS 公司所有通用 ARM 微控制器的仿真和开发。
- 采用 DeviceARM2200 核心板,标准配置为 PHILIPS 公司的 ARM7TDMI-S 微控制器 LPC2290、2 MB NOR FLASH(BootLoader)、8 MB RAM、16 MB NAND FLASH 和 256 字节 EEPROM。
- 标配 5.2 英寸 320×240 彩色液晶屏,带触摸屏,人机界面更华丽。
- 全面支持 6 种型号的 144 PIN ARM7 微控制器:LPC2200 (LPC2210/2212/2214/2290/2292/2294)。
- 灵活的跳线器选择,让用户可以重新分配 I/O 资源。
- 具有外设 PACK 和 GPIO 输出接口,方便用户连接外部电路的开发和使用。
- 具有 ISP1161 电路,支持 USB1.1 HOST/Device 固件编程与驱动程序的开发。
- 具有 SD 卡接口,支持 SD/MMC 卡读/写。
- 具有 DM9000E 网卡芯片,支持 10/100M 以太网通信。具有 IDE 硬盘接口、CF 存储卡接口。
- 可与标准 MODEM 直接接口,进行远程数据通信。
- 具有 RS232 转换电路,可与上位机进行串行通信,完成 UART 通信实验。
- 包含 2 路 CAN 接口,方便组装现场总线。
- 多种可选配置适配器:

- GPS 模块，进行地理定位实验；
- GPRS 模块，进行 GPRS 实验；
- PDIUSBD12 USB1.1 Device PCAK；
- ISP1181 USB1.1 Device PACK；
- ISP1160 USB1.1 HOST PACK。

- 使用 ZLG7290 扩展了 8 位动态 LED 数码管和一个 16 键小键盘。
- 8 个独立 LED，1 个独立按键，1 个蜂鸣器。
- 包含 1 路 RS485 接口电路，可做 RS485 通信实验。
- 具有打印机接口电路，提供打印机软件包。
- 具有 PS/2 键盘、鼠标接口，扩展系统的输入设备。
- 串行 DAC 电路，可同时输出 4 路 D/A 信号。
- 多种免费商业化软件包及其详细的开发文档：
  - 移植 μC/OS－II 到 ARM7 软件包；
  - 移植 μCLinux 到 LPC2200 软件包；
  - 数据队列软件包；
  - 串口驱动软件包；
  - MODEM 接口软件包；
  - SPI 总线软件包；
  - I^2C 总线软件包；
  - 支持 RS485/422 MODBUS RTU 协议软件包；
  - CAN-bus 协议软件包；
  - 打印机接口软件包；
  - ZLG/GPS 软件包（选配）；
  - ZLG/GPRS 软件包（选配）；
  - ZLG/FS 文件管理系统软件包；
  - MiniGUI 图形用户界面学习版软件包（μC/OS－II & μCLinux）；
  - ZLG/IP TCP/IP 软件包；
  - ZLG/PPP 协议软件包；
  - ZLG/CF CF 卡及 IDE 硬盘软件包；
  - ZLG/SD SD/MMC 读/写软件包；
  - ZLG/FFS 支持写平衡的 NAND FLASH 驱动软件包；
  - ZLG/HOST&Device 固件程序及其驱动程序软件包；
  - ZLG/FTP-S 嵌入式 FTP 服务器软件包；
  - ZLG/BOOT BootLoader 程序。

- 提供 μCLinux 驱动程序编程实验，μCLinux 应用程序编程实例。
- 自主设计的 ZLG/BOOT BootLoader 程序，使用 ZLG/FS 文件管理系统，支持 FTP 下载，下载程序速度更快。
- 提供基于 μC/OS－II & μCLinux 的实验例子，如串口通信、I^2C 存储器读/写、MODEM 通信等。
- 提供 MiniGUI for μCLinux 移植实验、MiniGUI 应用实例；免费提供 MiniGUI-STR 软件包。
- 提供基于 PC 的人机界面软件，方便调试实时时钟、串口通信等功能。
- 可进行 GPIO 的控制实验，如 LED 闪烁控制、按键输入、蜂鸣器控制等。
- 可进行外部中断实验，学习向量中断控制器(VIC)。
- 使用板内的 I^2C 器件，完成 I^2C 总线的实验。
- 定时器控制实验，实时时钟控制实验。
- WDT 及低功耗控制实验。
- EMC 设置实验，REMAP 操作实验，ADC 数据采集实验。

大量的实验程序，详细的配置教材(高等学校嵌入式系统通用教材——ARM 嵌入式系统系列教材一套共 10 本。其中 2 本为标准配置，北京航空航天大学出版社针对 SmartARM2200&MagicARM2200 教学实验平台配套编写了《ARM 嵌入式 μCLinux 系统构建与驱动开发范例》与《嵌入式 MiniGUI 初步与应用开发范例》2 本专著，以及完整的多媒体教学课件。