

UC/OS ii : 源码公开的实时嵌入式操作系统



Powered by xiaoguo's publishing studio
QQ:8204136

目 录

译者序
序
引言



第1章 范例	1
1.0 安装μC/OS-II	1
1.1 INCLUDES.H	3
1.2 与编译器无关的数据类型	3
1.3 全局变量	4
1.4 OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()	6
1.5 基于PC的服务	6
1.6 应用μC/OS-II的范例	8
1.7 例1	10
1.8 例2	15
1.9 例3	22
第2章 实时系统概念	29
2.0 前后台系统(Foreground/Background System)	29
2.1 代码的临界区	30
2.2 资源	30
2.3 共享资源	31
2.4 多任务	31
2.5 任务	31
2.6 任务切换	33
2.7 内核(Kernel)	33
2.8 调度(Scheduler)	34
2.9 非占先式内核	34
2.10 占先式内核	35
2.11 可重入性(Reentrancy)	36
2.12 时间片轮番调度法	38
2.13 任务优先级	38
2.14 静态优先级	38
2.15 动态优先级	39

2.16	优先级反转	39
2.17	任务优先级分配	41
2.18	互斥条件	43
2.19	死锁(或抱死) (Deadlock (or Deadly Embrace))	50
2.20	同步	51
2.21	事件标志(Event Flag)	53
2.22	任务间的通信	53
2.23	消息邮箱	54
2.24	消息队列(Message Queue)	55
2.25	中断	56
2.26	中断延迟	57
2.27	中断响应	57
2.28	中断恢复时间(Interrupt Recovery).....	58
2.29	中断延迟、响应和恢复.....	59
2.30	中断处理时间	59
2.31	非屏蔽中断(NMI)	60
2.32	时钟节拍(Clock Tick)	63
2.33	对存储器的需求	65
2.34	使用实时内核的优缺点.....	66
2.35	实时系统小结	66
 第 3 章 内核结构		68
3.0	临界区(Critical Section).....	68
3.1	任务	69
3.2	任务状态	70
3.3	任务控制块 (Task Control Blocks, OS_TCB)	72
3.4	就绪表 (Ready List)	75
3.5	任务调度 (Task Scheduling)	78
3.6	给调度器上锁和开锁(Locking and UnLocking the Scheduler).....	80
3.7	空闲任务(Idle Task)	81
3.8	统计任务	82
3.9	μ C/OS 中的中断处理.....	86
3.10	时钟节拍	90
3.11	μ C/OS- II 初始化	93
3.12	μ C/OS- II 的启动	95
3.13	获取当前 μ C/OS- II 的版本号	97
3.14	OSEvent???()函数	98

第 4 章 任务管理	99
4.0 建立任务, OSTaskCreate()	100
4.1 建立任务, OSTaskCreateExt()	104
4.2 任务堆栈	108
4.3 堆栈检验, OSTaskStkChk()	110
4.4 删 除任务, OSTaskDel()	113
4.5 请求删除任务, OSTaskDelReq()	116
4.6 改变任务的优先级, OSTaskChangePrio()	119
4.7 挂起任务, OSTaskSuspend()	122
4.8 恢复任务, OSTaskResume()	124
4.9 获得有关任务的信息, OSTaskQuery()	125
第 5 章 时间管理	127
5.0 任务延时函数, OSTimeDly()	127
5.1 按时分秒延时函数 OSTimeDlyHMSM()	129
5.2 让处在延时期的任务结束延时, OSTimeDlyResume()	131
5.2 系统时间, OSTimeGet()和 OSTimeSet()	132
第 6 章 任务之间的通信与同步	134
6.0 事件控制块 ECB	135
6.1 初始化一个事件控制块, OSEventWaitListInit()	139
6.2 使一个任务进入就绪态, OSEventTaskRdy()	139
6.3 使一个任务进入等待某事件发生状态, OSEventTaskWait()	141
6.4 由于等待超时而将任务置为就绪态, OSEventTO()	142
6.5 信号量	143
6.6 邮箱	150
6.7 消息队列	158
第 7 章 内存管理	174
7.0 内存控制块	175
7.1 建立一个内存分区, OSMemCreate()	176
7.2 分配一个内存块, OSMemGet()	179
7.3 释放一个内存块, OSMemPut()	180
7.4 查询一个内存分区的状态, OSMemQuery()	181
7.5 使用内存分区	182
7.6 等待一个内存块	184

第 8 章 移植μC/OS-II	186
8.0 开发工具	187
8.1 目录和文件	188
8.2 INCLUDES.H	189
8.3 OS_CPU.H	189
8.4 OS_CPU_A.ASM	193
8.5 OS_CPU_C.C	198
第 9 章 μC/OS-II 在 80x86 上的移植	207
9.0 开发工具	209
9.1 目录和文件	209
9.2 INCLUDES.H 文件	209
9.3 OS_CPU.H 文件	210
9.4 OS_CPU_A.ASM	214
9.5 OS_CPU_C.C	224
9.6 内存占用	228
9.7 运行时间	231
第 10 章 从μC/OS 升级到μC/OS-II	242
10.0 目录和文件	242
10.1 INCLUDES.H	243
10.2 OS_CPU.H	243
10.3 OS_CPU_A.ASM	246
10.4 OS_CPU_C.C	248
10.5 总结	253
第 11 章 参考手册	255
第 12 章 配置手册	315
附录 A 源代码范例	322
A.0 例 1	322
A.1 例 2	327
A.2 例 3	337
A.3 PC 服务	350

附录 B μC/OS-II 与处理器类型无关的源代码.....	362
B.0 uCOS_II.C.....	362
B.1 uCOS_II.H	363
B.2 OS_CORE.C	374
B.3 OS_MBOX.C	393
B.4 OS_MEM.C	398
B.5 OS_Q.C	403
B.6 OS_SEM.C	413
B.7 OS_TASK.C.....	419
B.8 OS_TIME.C	435
	
附录 C 80x86 源代码在实模式、大模式下编译.....	440
C.0 OS_CPU_A.ASM	440
C.1 OS_CPU_C.C.....	447
C.2 OS_CPU.H	450
附录 D HPLISTC 和 TO	453
D.0 HPLISTC	453
D.1 TO	454
附录 E 参考文献	456
附录 F 使用许可证 (License) 和 μCOS-II 网站	458

第1章

范例



在这一章里将提供三个范例来说明如何使用μC/OS-II。笔者之所以在本书一开始就写这一章是为了让读者尽快开始使用μC/OS-II。在开始讲述这些例子之前，笔者想先说明一些在这本书里的约定。

这些例子曾经用 Borland C/C++ 编译器 (V3.1) 编译过，用选择项产生 Intel/AMD 80186 处理器（大模式下编译）的代码。这些代码实际上在 Intel Pentium II PC (300MHz) 上运行和测试过，Intel Pentium II PC 可以看成是特别快的 80186。笔者选择 PC 做为目标系统是由于以下几个原因：首先也是最为重要的，以 PC 做为目标系统比起以其他嵌入式环境，如评估板，仿真器等，更容易进行代码的测试，不用不断地烧写 EPROM，不断地向 EPROM 仿真器中下载程序等等。用户只需要简单地编译、链接和执行。其次，使用 Borland C/C++ 产生的 80186 的目标代码（实模式，在大模式下编译）与所有 Intel、AMD、Cyrix 公司的 80x86 CPU 兼容。

1.0 安装μC/OS-II

本书附带一张软盘包括了所有我们讨论的源代码。读者应当具备 80x86，Pentium，或者 Pentium-II 处理器的 PC 机，并且运行 DOS 或 Windows 95 操作系统。至少需要 5Mb 硬盘空间来安装μC/OS-II。请按照以下步骤安装：

1. 进入到 DOS (或在 Windows 95 下打开 DOS 窗口) 并且指定 C: 为默认驱动器。
2. 将磁盘插入到 A: 驱动器。
3. 键入 “A: INSTALL[drive]”。

注意[drive]是读者安装μC/OS-II 的目标磁盘的盘符。

INSTALL.BAT 是一个 DOS 的批处理文件，位于磁盘的根目录下。它会自动在读者指定的目标驱动器中建立\SOFTWARE 目录，并且将 uCOS-II.EXE 文件从 A: 驱动器复制到 \SOFTWARE 并运行。μC/OS-II 将在\SOFTWARE 目录下添加所有的目录和文件。完成之后 INSTALL.BAT 将删除 uCOS-II.EXE，并且将目录改为\SOFTWARE\uCOS-II\x86L，第一个例子就存放在这里。

在安装之前请一定阅读一下 `READ.ME` 文件。当 `INSTALL.BAT` 已经完成时，用户的目标目录下应该有以下子目录：

\SOFTWARE

这是根目录，所有软件相关的文件都放在这个目录下。



\SOFTWARE\BLOCKS

子程序模块目录。笔者将例子中 μC/OS-II 用到的与 PC 相关的函数模块编译以后放在这个目录下。

\SOFTWARE\HPLISTC

这个目录中存放的是与范例 HPLIST 相关的文件（请看附录 D）。`HPLIST.C` 存放在 `\SOFTWARE\HPLISTC\SOURCE` 目录下。DOS 下的可执行文件 (`HPLIST.EXE`) 存放在 `\SOFTWARE\TO\EXE` 中。

\SOFTWARE\TO

这个目录中存放的是和范例 TO 相关的文件（请看附录 D）。源文件 `TO.C` 存放在 `\SOFTWARE\TO\SOURCE` 中，DOS 下的可执行文件 (`TO.EXE`) 存放在 `\SOFTWARE\TO\EXE` 中。注意 TO 需要一个 `TO.TBL` 文件，它必须放在根目录下。用户可以在 `\SOFTWARE\TO\EXE` 目录下找到 `TO.TBL` 文件。如果要运行 `TO.EXE`，必须将 `TO.TBL` 复制到根目录下。

\SOFTWARE\uCOS-II

与 μC/OS-II 相关的文件都放在这个目录下。

\SOFTWARE\uCOS-II\EX1_x86L

这个目录里包括例 1 的源代码（参见 1.7 节），可以在 DOS（或 Windows 95 下的 DOS 窗口）下运行。

\SOFTWARE\uCOS-II\EX2_x86L

这个目录里包括例 2 的源代码（参见 1.8 节），可以在 DOS（或 Windows 95 下的 DOS 窗口）下运行。

\SOFTWARE\uCOS-II\EX3_x86L

这个目录里包括例 3 的源代码（参见 1.9 节），可以在 DOS（或 Windows 95 下的 DOS 窗口）下运行。

\SOFTWARE\uCOS-II\x86L

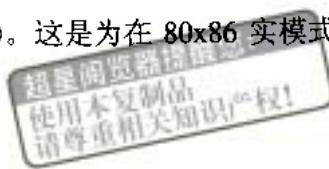
这个目录下包括与处理器类型相关的代码（也就是移植）。这是为在 80x86 实模式、大模式处理器上运行uC/OS-II 必需的一些代码。

\SOFTWARE\uCOS-II\SOURCE

这个目录里包括与处理器类型无关的源代码。这些代码完全可移植到其他架构的处理器上。

1.1 INCLUDES.H

用户将注意到本书中所有的*.C 文件都包括了以下定义：



```
typedef unsigned char  BOOLEAN;
typedef unsigned char  BYTE;
```

INCLUDE.H 可以使用户不必在工程项目中每个*.C 文件中都考虑需要什么样的头文件。换句话说，INCLUDE.H 是主头文件。这样做唯一的缺点是，INCLUDES.H 中许多头文件在一些*.C 文件的编译中是不需要的。这意味着逐个编译这些文件花费了额外的时间。这虽有些不便，但代码的可移植性却增加了。本书中所有的例子使用一个共同的头文件 INCLUDES.H，3 个副本分别存放在\SOFTWARE\uCOS-INEX1_x86L，\SOFTWARE\uCOS-INEX2_x86L，以及\SOFTWARE\uCOS-INEX3_x86L 中。当然可以重新编辑 INCLUDES.H 以添加用户自己的头文件。

1.2 与编译器无关的数据类型

因为不同的微处理器有不同的字长，uC/OS-II 的移植文件包括很多类型定义以确保可移植性（参见\SOFTWARE\uCOS-INEx86L\OS_CPU.H，它是针对 80x86 的实模式，在大模式下编译）。uC/OS-II 不使用 C 语言中的 short,int,long 等数据类型的定义，因为它们与处理器类型有关，隐含着不可移植性。笔者代之以移植性强的整数数据类型，这样，既直观又可移植，如程序清单 1.1 所示。为了方便起见，还定义了浮点数数据类型，虽然uC/OS-II 中没有使用浮点数。

程序清单 1.1 可移植型数据类型

```

typedef signed char INT8U;
typedef unsigned int INT16U;
typedef signed int INT16S;
typedef unsigned long INT32U;
typedef signed long INT32S;
typedef float FF32;
typedef double FF64;

#define BYTE 8
#define WORD 16
#define INT16 16
#define INT16D 16
#define LONG 32
#define ULONG 32

```

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

以 INT16U 双字类型为例，它代表 16 位无符号整数数据类型。 μ C/OS-II 和用户的应用代码可以定义这种类型的数据，范围从 0 到 65535。如果将 μ C/OS-II 移植到 32 位处理器中，那就意味着 INT16U 不再不是一个无符号整型数据，而是一个无符号短整型数据。然而无论将 μ C/OS-II 用到哪里，都会当作 INT16U 处理。程序清单 1.1 是以 Borland C/C++ 编译器为例，为 80x86 提供的定义语句。为了和 μ C/OS 兼容，还定义了 BYTE，WORD，LONG 以及相应的无符号变量。这使得用户可以不作任何修改就能将 μ C/OS 的代码移植到 μ C/OS-II 中。之所以这样做是因为笔者觉得这种新的数据类型定义有更多的灵活性，也更加易读易懂。对一些人来说，WORD 意味着 32 位数，而此处却意味着 16 位数。这些新的数据类型应该能够消除此类含混不清。

1.3 全局变量

以下是如何定义全局变量。众所周知，全局变量应该是得到内存分配且可以被其他模块通过 C 语言中 **extern** 关键字调用的变量。因此，必须在.C 和.H 文件中声明。这种重复的声明很容易导致错误。以下讨论的方法只需用在头文件中声明一次。虽然有些费解，但用户一旦掌握，使用起来却很灵活。程序清单 1.2 中的声明将出现在定义全局变量的所有.H 头文件中。

程序清单 1.2 定义全局宏

```
#define _xxx_EXT  
#else  
#define _xxx_EXT extern  
#endif
```

第1章 范例

```
#define _xxx_GLOBALS  
#include "tipcdata.h"
```

```
#ifdef _xxx_GLOBALS  
#define OS_EXT  
#else  
#define OS_EXT extern  
#endif  
  
OS_EXT INT32U OSIdleCtry;  
OS_EXT INT32U OSIdleCtryRun;  
OS_EXT INT32U OSIdleCtryRun;
```

```
当编译器处理 uCOS_II.C 时，  
#define OS_GLOBALS  
#include "tipcdata.h"  
  
INT32U OSIdleCtry;  
INT32U OSIdleCtryRun;  
INT32U OSIdleCtryRun;
```

_xxx_EXT 的前缀。xxx 代表模块的名字。该模块的.C

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

_xxx（在相应.H 文件中可以找到）为空，（因为给每个全局变量分配内存空间。而当编译器处理其他.C 文件时，**xxx_GLOBAL** 没有定义，**xxx_EXT** 被定义为 **extern**，这样用户就可以调用外见C/OS-II.H，其中包括以下定义：

同时，COS-II.C 中有以下定义：

当编译器处理 COS-II.C 时，它使得头文件变成如下所示，因为 **OS_EXT** 要被设置为空。

这样编译器就会将这些全局变量分配在内存中。当编译器处理其他.C 文件时，头文件

```
extern INT32 OS_ENTER_Critical();
extern INT32 OS_EXIT_Critical();
```

变成了如下的样子，因为 OS_GLOBAL 没有定义，所以 OS_EXT 被定义为 extern。



在这种情况下，不产生内存分配，而任何.C 文件都可以使用这些变量。这样只需在.H

OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()

ICAL()和 OS_EXIT_CRITICAL()两个宏，贯穿本书的所有源代码。OS_ENTER_CRITICAL()关中断；而 OS_EXIT_CRITICAL()开中断。关中断和开中断是为了保护临界区代码。这些代码很显然与处理器有关。关于宏的定义可以在 OS_CPU.H 中找到。9.3.2 节将详细讨论定义这些宏的两种方法。

程序清单 1.3 存取临界区的宏

用户的应用代码可以使用这两个宏来开中断和关中断。很明显，关中断会影响中断延迟，所以要特别小心。用户还可以用信号量来保护临界区（critical section）。

1.5 基于 PC 的服务

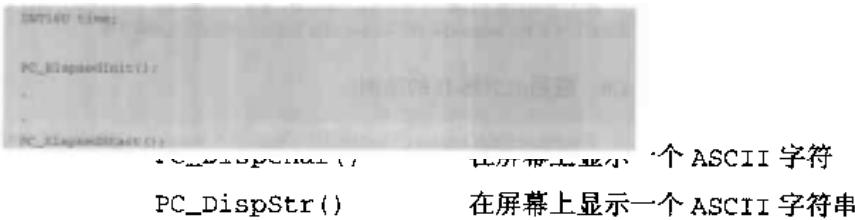
PC.C 文件和 PC.H 文件（在\SOFTWARE\BLOCKS\PC\SOURCE 目录下）是笔者在范例中使用到的一些基于 PC 的服务程序。与μC/OS-II 以前的版本（即 μC/OS）不同，笔者

希望集中这些函数以避免在各个例子中都重复定义，也更容易适应不同的编译器。PC.C 包括字符显示、时间度量和其他各种服务。所有的函数都以 PC_ 为前缀。

1.5.1 字符显示

为了性能更好，显示函数直接向显示内存区中写数据。在 VGA 显示器中，显示内存从绝对地址 0x000B8000 开始（或用段、偏移量表示则为 B800:0000）。在单色显示器中，用户可以把#define constant DISP_BASE 从 0xB800 改为 0xB000。

PC.C 中的显示函数用 x 和 y 坐标来直接向显示内存中写 ASCII 字符。PC 的显示可以达到 25 行 80 列一共 2000 个字符。每个字符需要两个字节来显示。第一个字节是用户想要显示的字符，第二个字节用来确定前景色和背景色。前景色用低四位来表示，背景色用第 4 位到 6 位来表示。最高位表示这个字符是否闪烁，(1) 表示闪烁，(0) 表示不闪烁。用 PC.H 中 #defien constants 定义前景和背景色，PC.C 包括以下四个函数：



1.5.2 花费时间的测量

时间测量函数主要用于测试一个函数的运行花了多少时间。测量时间是用 PC 的 82C54 定时器 2。被测的程序代码是放在函数 PC_ElapsedStart() 和 PC_ElapsedStop() 之间来测量的。在用这两个函数之前，应该调用 PC_ElapsedInit() 来初始化，它主要是计算运行这两个函数本身所附加的时间。这样，PC_ElapsedStop() 函数中返回的数值就是准确的测量结果了。注意，这两个函数都不具备可重入性，所以，必须小心，不要有多个任务同时调用这两个函数。程序清单 1.4 说明了如何测量 PC_DisplayChar() 的执行时间。注意，时间是以 μ s 为单位的。

程序清单 1.4 测量代码执行时间

1.5.3 其他函数

μ C/OS-II 的应用程序和其他 DOS 应用程序是一样的，换句话说，用户可以像在 DOS 下编译其他单线程的程序一样编译和链接用户程序。所生成的 .EXE 程序可以在 DOS 下装载和运行，当然应用程序应该从 main() 函数开始。因为 μ C/OS-II 是多任务操作系统，而且为每个任务开辟一个堆栈，所以单线程的 DOS 环境应该保存，在退出 μ C/OS-II 程序时返回到 DOS。调用 PC_DOSSaveReturn() 可以保存当前 DOS 环境，而调用 PC_DOSReturn() 可以返回到 DOS。PC.C 中使用 ANSI C 的 setjmp(), longjmp() 函数来分别保存和恢复 DOS 环境。Borland C/C++ 编译库提供这些函数，多数其他的编译程序也应有这类函数。

应该注意到，无论是应用程序的错误，还是只调用 exit(0) 而没有调用 PC_DOSReturn() 函数，都会使 DOS 环境被破坏，从而导致 DOS 或 Windows 95 下的 DOS 窗口崩溃。

调用 PC_GetDateTime() 函数可得到 PC 中的日期和时间，并且以 ASCII 字符串形式返回。格式是 MM-DD-YY HH:MM:SS，用户需要 19 个字符来存放这些数据。该函数使用了 Borland C/C++ 的 gettime() 和 getdate() 函数，其他 DOS 环境下的 C 编译应该也有类似函数。

PC_GetKey() 函数检查是否有按键被按下。如果有按键被按下，函数返回其值。这个函数使用了 Borland C/C++ 的 kbhit() 和 getch() 函数，其他 DOS 环境下的 C 编译应该也有类似函数。

函数 PC_SetTickRate() 允许用户为 μ C/OS-II 定义频率，以改变时钟节拍的速率。在 DOS 下，每秒产生 18.20648 次时钟节拍，或每隔 54.925ms 一次。这是因为 82C54 定时器芯片没有初始化，而使用默认值 65535 的结果。如果初始化为 58659，那么时钟节拍的速率就会精确地为 20.000Hz。笔者决定将时钟节拍设得更快一些，用的是 200Hz（实际上是 199.9966Hz）。注意 OS_CPU_A.ASM 中的 OSTickISR() 函数将会每 11 个时钟节拍调用一次 DOS 中的时钟节拍处理，这是为了保证在 DOS 下时钟的准确性。如果用户希望将时钟节拍的速度设置为 20Hz，就必须这样做。在返回 DOS 以前，要调用 PC_SetTickRate()，并设置 18 为目标频率，PC_SetTickRate() 就会知道用户要设置为 18.2Hz，并且会正确设置 82C54。

PC.C 中最后两个函数是获取和设置中断向量，笔者是用 Borland C/C++ 中的库函数来完成的，当然 PC_VectGet() 和 PC_VectSet() 很容易改写，以适用于其他编译器。

1.6 应用 μ C/OS-II 的范例

本章中的例子都用 Borland C/C++ 编译器编译通过，是在 Windows 95 的 DOS 窗口下编

Code generation	
Model	Large
Options	Treat enum as int
Assume SS Equals DS	Default for memory model
Advanced code generation	
Floating point	Rounding:
Instruction set	I386
Options	Generate underscores Debug info in COFF Fast floating point
Optimizations	
Optimizations	Global register allocation Invariant code motion Induction variables Loop optimization Supress redundant loads Copy propagation Dead code elimination Jump optimization In-line intrinsic functions Automatic Optimize globally Speed
Register variables	
Common subexpressions	
Optimize for	

子目录下找到。实际上这些代码是在 Borland IDE 编译的，编译时的选项如表 1.1 所示：

1.1 的编译选项

郑重提醒您：
使用本复制品
请尊重相关知识产权！

笔者的 Borland C/C++ 编译器安装在 C:\CPP 目录下，如果用户的编译器是在不同的目录下，可以在 Options/Directories 菜单下改变路径。

μC/OS-II 是一个可裁剪的操作系统，这意味着用户可以去掉不需要的服务。代码的削减可以通过设置 OS_CFG.H 中的#define OS_???_EN 为 0 来实现。一旦 OS_??? 为 0 用户不需要的服务代码就不生成。本章的范例就用这种功能，所以每个例子都定义了不同的 OS_???_EN。

1.7 例 1

第一个范例可以在\SOFTWARE\μCOS_INEX1_x86L 目录下找到，它有 13 个任务（包括 μC/OS-II 的空闲任务）。μC/OS-II 增加了两个内部任务：空闲任务和一个计算 CPU 利用率的任务。例 1 建立了 11 个其他任务。TaskStart()任务是在函数 main()中建立的；它的功能是建立其他任务，并且在屏幕上显示如下统计信息：

- 每秒钟任务切换次数；
- CPU 利用百分率；
- 寄存器切换次数；
- 目前日期和时间；
- μC/OS-II 的版本号。

```
void main(void)
{
    /* Clear the screen */
    PC_SetColor(0x00, 0x00, 0x00);
    OSInit();                                /* (1) */
    PC_DOSSaveScreen();                      /* (2) */
    PC_WaitScreen(OSOS, 0x00);               /* (3) */
}
```

决定是否返回到 DOS。

Task(); 每个任务在屏幕上随机的位置显示一个 0

1.7.1 main()

例 1 基本上和最初 μC/OS 中的第一个例子做一样的事，但是笔者整理了其中的代码，并且在屏幕上加了彩色显示。同时笔者使用原来的数据类型 (UBYTE, UWORD 等) 来说明 μC/OS-II 向下兼容。

main()程序从清整个屏幕开始，为的是保证屏幕上不留有以前的 DOS 下的显示[程序清单 1.5(1)]。注意，笔者定义了白色的字符和黑色的背景色。既然要清屏幕，所以可以只定义背景色而不定义前景色，但是这样在退回 DOS 之后，用户就什么也看不见了。这也是为什么总要定义一个可见的前景色。

μC/OS-II 要用户在使用任何服务之前先调用 OSInit() [程序清单 1.5(2)]。它会建立两个任务：空闲任务和统计任务，前者在没有其他任务处于就绪态时运行；后者计算 CPU 的利用率。

程序清单 1.5 main()

```
TaskHandle = OSSemCreate(1);  
OSTaskCreate(TaskStart,  
    void *10,  
    void *14TaskStart,(void *_OSR, _OSR-1),  
    0);  
OSStart();
```

第1章 范例



当前 DOS 环境是通过调用 PC_DOSSaveReturn() [程序清单 1.5(3)] 来保存的。这使得用户可以返回到没有运行 μC/OS-II 以前的 DOS 环境。跟随清单 1.6 中的程序可以看到 PC_DOSSaveReturn() 做了很多事情。PC_DOSSaveReturn() 首先设置 PC_ExitFlag 为 FALSE [程

然后初始化 OSTickDOSCr 为 1 [程序清单 1.6(2)]，而 0 将使得这个变量在 OSTickISR() 中减 1 后变为 1；的时钟节拍处理程序入口指针 (tick handler) 存储器 [程序清单 1.6(3)-(4)]，以便为 μC/OS-II 的时钟节拍处理程序所调用。

。设有 PC_DOSSaveReturn() 调用 longjmp() [程序清单 1.6(5)]，它将处理器状态（即所有寄存器的值）存入被称为 PC_JumpBuf 的结构之中。保存处理器的全部寄存器，可以使程序返回到 PC_DOSSaveReturn()，并且在调用 Setjmp() 之后立即执行。因为 PC_ExitFlag 被初始化为 FALSE [程序清单 1.6(1)]。PC_DOSSaveReturn() 跳过 if 状态语句 [程序清单 1.6(6)-(9)]，回到 main() 函数。如果用户想要返回到 DOS，可以调用 PC_DOSReturn() [程序清单 1.7]，它设置 PC_ExitFlag 为 TRUE，并且执行 longjmp() 语句 [程序清单 1.7(2)]，这时处理器将跳回 PC_DOSSaveReturn() [在调用 setjmp() 之后] [程序清单 1.6(5)]，此时 PC_ExitFlag 为 TRUE，故 if 语句以后的代码将得以执行。PC_DOSSaveReturn() 将时钟节拍改为 18.2Hz [程序清单 1.6(6)]，恢复 PC 时钟节拍中断服务 [程序清单 1.6(7)]，清屏幕 [程序清单 1.6(8)]，通过 exit(0) 返回 DOS [程序清单 1.6(9)]。

程序清单 1.6 保存 DOS 环境

```
void PC_DOSSaveReturn()  
{  
    PC_ExitFlag = FALSE; // (1)  
    OSTickDOSCr = 1; // (2)  
    PC_JumpBuf = PC_SetupJumpBuf(); // (3)  
  
    OS_ENTER_CRITICAL();  
    PC_SetISR(OSTICK_ISR, PC_TickISR); // (4)  
    OS_EXIT_CRITICAL();  
}
```

。设有 PC_DOSSaveReturn() 调用 longjmp() [程序清单 1.6(5)]，它将处理器状态（即所有寄存器的值）存入被称为 PC_JumpBuf 的结构之中。保存处理器的全部寄存器，可以使程序返回到 PC_DOSSaveReturn()，并且在调用 Setjmp() 之后立即执行。因为 PC_ExitFlag 被初始化为 FALSE [程序清单 1.6(1)]。PC_DOSSaveReturn() 跳过 if 状态语句 [程序清单 1.6(6)-(9)]，回到 main() 函数。如果用户想要返回到 DOS，可以调用 PC_DOSReturn() [程序清单 1.7]，它设置 PC_ExitFlag 为 TRUE，并且执行 longjmp() 语句 [程序清单 1.7(2)]，这时处理器将跳回 PC_DOSSaveReturn() [在调用 setjmp() 之后] [程序清单 1.6(5)]，此时 PC_ExitFlag 为 TRUE，故 if 语句以后的代码将得以执行。PC_DOSSaveReturn() 将时钟节拍改为 18.2Hz [程序清单 1.6(6)]，恢复 PC 时钟节拍中断服务 [程序清单 1.6(7)]，清屏幕 [程序清单 1.6(8)]，通过 exit(0) 返回 DOS [程序清单 1.6(9)]。

```
SetInputJumpBuf();          (54)
if (PC_TaskFlag == TRUE) {
    OS_Setup_CRITICAL();
    PC_SetTask(OSTaskID);
    PC_VectSet(VECTC_TASK, PC_TaskID);
    OS_EXIT_CRITICAL();
    PC_Dispatcher(OSPF_PEND_WRITE + OSPF_WIND_BLOCK);
} else {
    (55)
}
```

公开的实时嵌入式操作系统

```
void PC_DOSReturn_VOID()
{
    PC_TaskFlag = TRUE;
    Longjmp(PC_JmpBuf, 1);           (56)
}
```



程序清单 1.7 设置返回 DOS

现在回到 main()这个函数，在程序清单 1.5 中，main()调用 PC_VectSet()来设置μC/OS-II 中的 CPU 寄存器切换。任务级的 CPU 寄存器切换通过向相应的向量地址发出 80x86INT 指令来实现。笔者使用向量 0x80（即 128），因为它未被 DOS 和 BIOS 使用。

这里用了一个信号量来保护 Borland C/C++库中产生随机数的函数[程序清单 1.5(5)]，之所以使用信号量保护一下，是因为笔者不知道这个函数是否具备可重入性，笔者假设其不具备，初始化将信号量设置为 1，意思是在某一时刻只有一个任务可以调用随机数产生函数。

在开始多任务之前，笔者建立了一个叫做 TaskStart()的任务[程序清单 1.5(6)]，在用 OOSStart()启动多任务之前用户至少要先建立一个任务，这一点非常重要[程序清单 1.5(7)]。如果不这样做用户的的应用程序将会崩溃。实际上，如果用户要计算 CPU 的利用率时，也需要先建立一个任务。μCOS-II 的统计任务要求在整个一秒钟内没有任何其他任务运行。如果用户在启动多任务之前要建立其他任务，必须保证用户的任务代码监控全局变量 OSSStatRdy 和延时程序 [即调用 OSTimeDly()] 的执行，直到这个变量变成 TRUE。这表明 μC/OS-II 的 CPU 利用率统计函数已经采集到了数据。

1.7.2 TaskStart()

例 1 中的主要工作由 TaskStart()来完成。TaskStart()函数的示意代码如程序清单 1.8 所

示。TaskStart()首先在屏幕顶端显示一个标识，说明这是例1 [程序清单1.8(1)]。然后关闭中断，以改变时钟节拍的中断服务程序（ISR）的入口地址，让其指向μC/OS-II的时钟节拍处

理的18.2Hz变为200Hz [程序清单1.8(3)]。在处理初始化前，当然不希望程序被中断！注意main()这个时候，并没有将中断向量设置成μC/OS-II的时钟，必须在第一个任务中打开时钟节拍中断。

```
void TaskStart(void *data)
{
    /* Prevent compiler warning by assigning "data" to itself;
       Display banner identifying this as EXAMPLE #1; */
    OS_ENTER_CRITICAL();
    PC_SetVectBase(OS_ClockISR);           /* (1) */
    PC_SetTickRate(200);                  /* (2) */
    OS_EXIT_CRITICAL();                   /* (3) */

    /* Initialize the statistic task by calling "osInitStat()"; */
    OSInitStat();                         /* (4) */

    /* Create 10 Identical tasks; */
    Create_10_Identical_tasks();          /* (5) */

    /* TOP (1) */
    Display_the_number_of_tasks_created();
    Display_the_X_and_CPU_speed();
    Display_the_number_of_task_switches_in_1_second();
    Display_uC/OS-II's_version_number();
    TE(key_was_pressed);
    if (key_pressed_were_the_ESCAPE_key) {
        PC_000Return();
    }
}

/* Delay for 1 second. */
Delay_for_1_second();

```

在建立其他任务之前，必须调用OSStatInit() [程序清单1.8(4)]来确定用户的PC速度，

如程序清单 1.9 所示。在一开始，OSStatInit()就将自身延时了两个时钟节拍，这样它就可以与时钟节拍中断同步[程序清单 1.9(1)]。因此，OSStatInit()必须在时钟节拍启动之后调用；否则，用户的应用程序就会崩溃。当μC/OS-II 调用 OSStatInit()时，一个 32 位的计数器 OSIdleCtr 被清为 0 [程序清单 1.9(2)]，并产生另一个延时，这个延时使 OSStatInit()挂起。此

时 μC/OS-II 没有别的任务可以执行。它只能执行空闲任务（μC/OS-II 的内部任务）。空闲 OSIdleCtr[程序清单 1.9(3)]。1 秒以后，μC/OS-II 继续测量 OSIdleMax 中[程序清单 1.9(4)]。OSIdleMax 现而当用户再增加其他应用代码时，空闲任务就不会可能达到那样多的记数（如果用户程序每秒复位 μC/OS-II 调用 OSStatTask()函数产生的任务来完成，y 置为 TRUE[程序清单 1.9(5)]，表示 μC/OS-II 将统计

CPU 的利用率。

程序清单 1.9 测试 CPU 速度

```
void OSStatInit(void)
{
    OSInit();
    OS_ENTER_CRITICAL();
    OSIdleCtr = 0;
    OS_EXIT_CRITICAL();
    OSIdleMax = OS_IDLE_MAX;
    OS_IDLECYCLE = OS_IDLECYCLE;
    OSStatInit = TRUE;
    OS_LEAVE_CRITICAL();
}
```

1.7.3 TaskN()

OSStatInit()将返回到 TaskStart()。现在，用户可以建立 10 个同样的任务（所有任务共享同一段代码）。所有任务都由 TaskStart()中建立，由于 TaskStart()的优先级为 0（最高），新任务建立后不进行任务调度。当所有任务都建立完成后，TaskStart()将进入无限循环之中，在屏幕上显示统计信息，并检测是否有 ESC 键按下，如果没有按键输入，则延时一秒开始下一次循环；如果在这期间用户按下了 ESC 键，TaskStart()将调用 PC_DOSReturn()返回 DOS 系统。

程序清单 1.10 给出了任务的代码。任务一开始，调用 OSSemPend()获取信号量

RandomSem [程序清单 1.10(1)] (译注 1), 然后调用 Borland C/C++ 的库函数 random() 来获得一个随机数 [程序清单 1.10(2)]。此外设 random() 函数是不可重入的, 所以 10 个任务将轮

x 和 y 坐标后 [程序清单 1.10(3)], 任务释放信号号 (0~9, 任务建立时的标识) [程序清单 1.10(4)]。1.10(5)], 等待进入下一次循环。系统中每个任务 2000 次。

显示数字的任务

```
void Task(void *data)
{
    int x;
    int y;

    while(1)
    {
        OSWaitSemaphore(RandomSem, 0, &err);
        x = random(10);
        y = random(14);
        OSReleaseSemaphore(RandomSem);
        PC_DisplayChar(x, y + 5, *(char *)data, DCF_PEND_CRT8_CURSOR);
        continue();
    }
}
```

1.8 例 2

例 2 使用了带扩展功能的任务建立函数 OSTaskCreateExt(), 和 μC/OS-II 的堆栈检查操作功能 (译注 2)。当用户不知道应该给每个任务分配多少堆栈空间时, 堆栈检查功能是很有效的。在这个例子里, 先分配足够的堆栈空间给任务, 然后用堆栈检查操作看看任务到底需要多少堆栈空间。显然, 任务要运行足够长时间, 并要考虑各种情况才能得到正确数据。最后决定的堆栈大小还要考虑系统今后的扩展, 一般多分配 10%~25% 或者更多。如

译注 1: 也就是禁止其他任务运行这段代码。

译注 2: 要使用堆栈检查操作必须用 OSTaskCreateExt() 建立任务。

果系统对稳定性要求高，则应该考虑多分配一倍以上。

μCOS-II 的堆栈检查功能要求任务建立时堆栈全部赋零。OSTaskCreateExt()可以执行此项操作（设置选项 OS_TASK_OPT_STK_CHK 和 OS_TASK_OPT_STK_CLR 打开此项操作）。如果任务运行过程中要进行建立、删除任务的操作，应该设置好上述的选项，确保任务建立后堆栈是清空的。同时要意识到 OSTaskCreateExt()进行堆栈清零操作是一项很费时的工作，而且取决于堆栈的大小。执行堆栈检查操作的时候，μC/OS-II 从栈底向栈顶搜索非 0 元素（参看图 1.1），同时用一个计数器记录 0 元素空闲空间的个数。

例 2 的磁盘文件为\SOFTWARE\μC/OS-II\EX2_x86L，它包含 9 个任务。加上 μCOS-II 本身的两个任务：空闲任务（idle task）和统计任务。与例 1一样，TaskStart()由 main()函数建立，其功能是建立其他任务并在屏幕上显示如下的统计数据：

- 每秒钟任务切换的次数；
- CPU 利用率的百分比；
- 当前日期和时间；
- μCOS-II 的版本号。

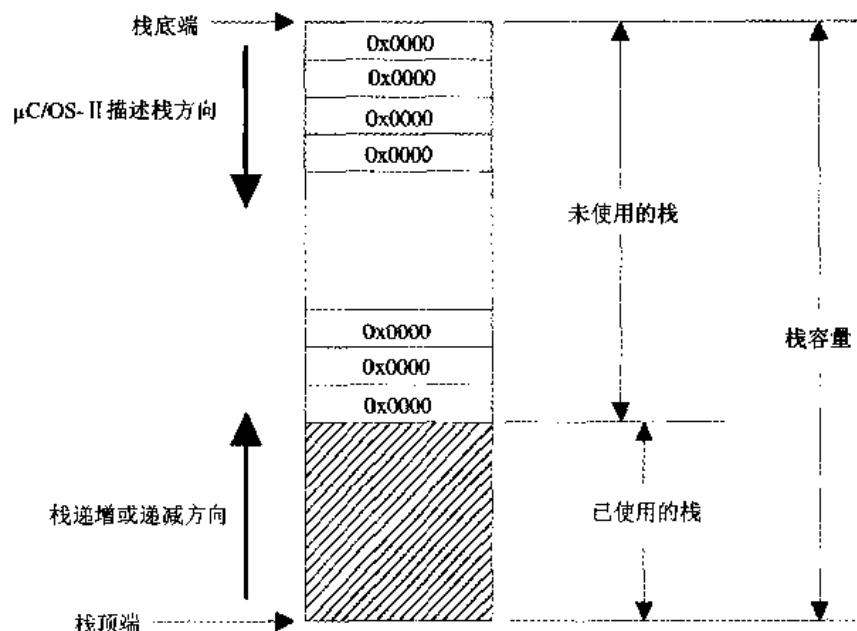


图1.1 μC/OS-II 堆栈检查

1.8.1 main()

例 2 的 main() 函数和例 1 的看起来差不多（参见程序清单 1.11），但是有两处不同。第一，main() 函数调用 PC_ElapsedInit() [程序清单 1.11(1)] 来初始化已用时间测量函数，以记录 OSTaskStkChk() 的执行时间。第二，所有的任务都使用 OSTaskCreateExt() 函数来建立任务[程

ate()], 这使得每一个任务都可进行堆栈检查。



```
void main (void)
{
    PC_DispClear((GSEG_BLOCK_0WHITE + GSEG_BLOCK_BLACK));
    OSInit();
    PC_DOSSaveReturn();
    PC_WaitForIC00, OSCallw();
    PC_DispatchInit(); (1)
    OSTaskCreateExt (TaskStart, (2)
        (void *)0,
        &TaskStartExtTCB, STK_SIZE-1),
        TASK_STK0,
        TASK_STK1,
        &TaskStartExtTCB[0],
        TASK_STK2,
        TASK_STK3,
        (void *)0,
        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
    OSStart();
}
```

void TaskStart (void *Param)

除了 OSTaskCreate()函数的四个参数外, OSTaskCreateExt()还需要五个参数(一共 9 个): 任务的 ID, 一个指向任务堆栈栈底的指针, 堆栈的大小(以堆栈单元为单位, 80X86 中为字), 一个指向用户定义的 TCB 扩展数据结构的指针, 和一个用来对任务确定不同操作的变量。该变量的一个选项就是用来设定是否允许检查 uC/OS-II 堆栈。例 2 中并没有用到 TCB 扩展数据结构指针。

1.8.2 TaskStart()

程序清单 1.12 列出了 TaskStart()的示意代码。前五项操作和例 1 中相同。TaskStart()建立了两个邮箱, 分别提供给任务 4 和任务 5[程序清单 1.12(1)]。除此之外, 还建立了一个专门显示时间和日期的任务。

程序清单 1.12 TaskStart() 的示意代码

```

Prevent compiler warning by assigning 'data' to itself.
Display a banner and non-changing text;
Install uC/OS-II's tick handler;
Change the tick rate to 200 Hz;
Initialize the statistics tasks;
Create 2毫秒任务，由任务#4和#5使用; (1)
Create a task that will display the date and time on the screen; (2)
Create 5 application tasks;
ISR (1) {
    Display tasks running;
    Display CPU usage in %;
    Display Position: switches per second;
    Clear the context switch counter;
    Display uC/OS-II's version;
    If (key was pressed) {
        If (key pressed was the ESCAPE key) {
            Return to DOS;
        }
    }
    Delay for 2 seconds;
}

```

超星浏览器提醒您：
 使用本复制品
 请尊重相关知识产权！

```

void Task1(void *pdata)
{
    OSInit();
    OS_StackData(data);
}

```

1.8.3 TaskN()

任务1将检查其他七个任务堆栈的大小，同时记录OSTackStkChk()函数的执行时间[程序清单1.13(1)~(2)]，并与堆栈大小一起显示出来。注意所有堆栈的大小都是以字节为单位的。任务1每秒执行10次[程序清单1.13(3)]（间隔100ms）。

程序清单1.13 例2的任务1

```

INT16U    time;
INT8U    i;
char     s[80];

pdata = pdata;
for (;;) {
    for (i = 0; i < 7; i++) {
        PC_ElapsedStart();                                (1)
        err = OSTaskStkChk(TASK_START_PRIO+i, &data);
        time = PC_ElapsedStop();                          (2)
        if (err == OS_NO_ERR) {
            sprintf(s, "%3ld    %3ld    %3ld    %5d",
                    data.OSFree + data.OSUsed,
                    data.OSFree,
                    data.OSUsed,
                    time);
            PC_DispStr(19, 12+i, s, DISP_FGND_YELLOW);
        }
    }
    OSTimeDlyHMSM(0, 0, 0, 100);                      (3)
}
}

```

程序清单 1.14 所示的任务 2 在屏幕上显示一个顺时针旋转的指针（译注 3），每 200ms 旋转一格。

程序清单 1.14 任务 2

```

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

```

译注 3：用横线、斜线等字符表示。

```
PC_DisPlayChar(78, 15, '^', DISP_FONT_WHITE + DISP_BACK_BLACK);
OSTimeDelay(100);
PC_DisPlayChar(78, 15, '\\', DISP_FONT_WHITE + DISP_BACK_BLACK);
OSTimeDelay(100);
```

公开的实时嵌入式操作系统

```
void Task3 (void *data)
{
    char dummy[500];
    OSTimeDelay(1);
    data = data;
    for (i = 0; i < 499; i++)
        dummy[i] = '^';
    for (i = 0; i < 10; i++)
        PC_DisPlayChar(78, 15, '^', DISP_FONT_WHITE + DISP_BACK_BLACK);
    OSTimeDelay(20);
    PC_DisPlayChar(78, 15, '\\', DISP_FONT_WHITE + DISP_BACK_BLACK);
    OSTimeDelay(20);
    PC_DisPlayChar(78, 15, '^', DISP_FONT_WHITE + DISP_BACK_BLACK);
    OSTimeDelay(20);
    PC_DisPlayChar(78, 15, '\\', DISP_FONT_WHITE + DISP_BACK_BLACK);
    OSTimeDelay(20);
```



任务 2 相同的一个旋转指针，但是旋转的方向不数组，将堆栈填充掉，使得 OSTaskStkChk() 只需花是当堆栈已经快满的时候。

任务 4（程序清单 1.16）向任务 5 发送消息并等待确认[程序清单 1.16(1)]。发送的消息是一个指向字符的指针。每当任务 4 从任务 5 收到确认[程序清单 1.16(2)]，就将传递的 ASCII 码加 1 再发送[程序清单 1.16(3)]，结果是不断的传送“ABCDEFG…”。

```

void Task4(void *data)
{
    char *msg;
    INT32U err;

    data = data;
    msg = "A";
    for (i = 1;
        while (msg != 'E') {
            OMBoxPost(0xb000, (void *)msg);
            OMBoxPost(0xb000, 0, &err);
            msg++;
            if (msg == 'E')
                break;
        }
}

```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

```

void Task5(void *data)
{
    char *msg;
    INT32U err;

    data = data;
    for (i = 1;
        msg = (char *)OMBoxRead(0xb000, 0, &err);
        PC_Dispatch(70, 18, *msg, 0x00_F000_YELLOW+0x00_0000_RED); //2
        OMBoxSync(0, 0, 1); //3
        OMBoxPost(0xb000, (void *)i); //4
}

```

当任务 5（程序清单 1.17）接收消息[程序清单 1.17(1)]（发送的字符）后，就将字符显示到屏幕上[程序清单 1.17(2)]，然后延时 1 秒[程序清单 1.17(3)]，再向任务 4 发送确认信息。

程序清单 1.17 任务 5

```

void Task3(void *param)
{
    struct time now;
    struct date today;
    char s[48];

    data = data2;
    for(;;)
    {
        PC_GetDateTime(&now);
        PC_DisplayStr(0, 24, s, DISP_PDISP_NLCN + DISP_DISP_CNNF);
        OSWaitAsync(OS_TICKS_PER_SEC);
    }
}

```

当前日期和时间，每秒更新一次。



```
void main(void)
```

1.9 例 3

例 3 中使用了许多 uC/OS-II 提供的附加功能。任务 3 使用了 OSTaskCreateExt()中 TCB 的扩展数据结构，用户定义的任务切换对外接口函数[OSTaskSwHook()]，用户定义的统计任务（statistic task）的对外接口函数[OSTaskStatHook()]以及消息队列。例 3 的磁盘文件是\SOFTWARE\uCOS-II\EX3_x86L，它包括 9 个任务。除了空闲任务（idle task）和统计任务，还有 7 个任务。与例 1、例 2 一样，TaskStart()由 main()函数建立，其功能是建立其他任务，并显示统计信息。

1.9.1 main()

main()函数（程序清单 1.19）和例 2 中的差不多，不同的是在用户定义的 TCB 扩展数据结构中可以保存每个任务的名称[程序清单 1.19(1)]（扩展结构的声明在 INCLUDES.H 中定义，也可参看程序清单 1.20）。笔者定义了 30 个字节来存放任务名（包括空格）[程序清单 1.20(1)]。本例中没有用到堆栈检查操作，TaskStart()中禁止该操作[程序清单 1.19(2)]。

程序清单 1.19 例 3 的 main()函数

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

```

PC_Dispatcher((DISP_PDMH_MACHINE + DISH_SOME_BLOCK),
    0);
PC_DOSBaseReset();
PC_VectSet((COS, 0x000000));
PC_DispatchInit();

strcpy(taskUserData[TASK_START_ID].TaskName, "StartTask");           (1)
GetTaskUserData(taskUserData,
    TASK_ID,
    TASK_START_ID,
    &taskStartData[0],
    TASK_STW_SIZE,
    &taskUserData[TASK_START_ID],
    0);                      (2)

taskStartData;

```

```

typedef struct {
    char TaskName[10];
    INT16 TaskID;
    INT16 TaskStartTime;
    INT16 TaskDPCExecution;
} TASK_USER_DATA;

```

程序清单 1.20 TCB 扩展数据结构

1.9.2 任务

TaskStart()的示意代码如程序清单 1.21 所示，与例 2 有 3 处不同：

- 为任务 1、2、3 建立了一个消息队列[程序清单 1.21(1)]；
- 每个任务都有一个名字，保存在任务的 TCB 扩展数据结构中[程序清单 1.21(2)]；
- 禁止堆栈检查。

```

void TaskStart(void *data)
{
    Prevent compiler warning by assigning "data" to itself;
    Display a banner and non-changing text;
    Install software tick handler;
    Change the tick rate to 200 Hz;
    Initialize the statistics task;
    Create a message queue;                                (1)
    Create a task that will display the date and time on the screen;
    Create 5 application tasks with a name stored in the TCBs user... (2)
    for (i=1;
        Display tasks running;
        Display CPU usage in %;
        Display #context switches per second;
        Clear the context switch counter;
        Display software ticks remaining;
        if (key_was_pressed) {
            if (Key pressed was the ESCAPE key) {
                Return to DOS;
            }
        }
        Delay for 1 second;
    )
}

```



```

void Task1(void *data)
{
    char one = '1';
    char two = '2';
    char three = '3';
}

```

任务 1 向消息队列发送一个消息[程序清单 1.22(1)]，然后延时等待消息发送完成[程序清单 1.22(2)]。这段时间可以让接收消息的任务显示收到的消息。发送的消息有三种。

程序清单 1.22 任务 1

```
data = data;
for (i=0; i<5; i++)
    OSQPost(MsgQueue, (void *)&data); //1
    OSQPost(MsgQueue, (void *)&data); //2
    OSQPost(MsgQueue, (void *)&data); //3
    OSQPost(MsgQueue, (void *)&data); //4
    OSQPost(MsgQueue, (void *)&data); //5
```

第1章 范例

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

```
void Task2(void *data)
{
    char *msg;
    char *err;
    data = data;
    for (i=0; i<5; i++)
        msg = (char *)OSQRead(MsgQueue, 5, 4000); //1
        PC_Display(10, 14, *msg, 0xFFFF_FFFFFF_0000_0000); //2
        OSQRead(MsgQueue, 5, 0, 900); //3
    }
```

任务 2 处于等待消息的挂起状态，且不设定最大等待时间[程序清单 1.23(1)]。所以任务 2 将一直等待直到收到消息。当收到消息后，任务 2 显示消息并且延时 500ms[程序清单 1.23(2)]，延时的时间可以使任务 3 检查消息队列。

程序清单 1.23 任务 2

任务 3 同样处于等待消息的挂起状态，但是它设定了等待结束时间 250ms[程序清单 1.24(1)]。如果有消息来到，任务 3 将显示消息号[程序清单 1.24(3)]，如果超过了等待时间，任务 3 就显示“T”（意为 timeout）[程序清单 1.24(2)]。

程序清单 1.24 任务 3

```

(
    INT8U *msg;
    INT8U err;

    data = data;
    for (;;) {
        msg = (INT8U *)OSQPend(MsgQueue, OS_TICKS_PER_SEC/4, &err); (1)
        if (err == OS_TIMEOUT) {
            PC_DispChar(70,15,'T',DISP_FGND_YELLOW+DISP_BGND_RED); (2)
        } else {
            PC_DispChar(70,15,*msg,DISP_FGND_YELLOW+DISP_BGND_BLUE); (3)
        }
    }
)

```

任务 4 的操作只是从邮箱发送[程序清单 1.25(1)]和接收[程序清单 1.25(2)]，这使得用户可以测量任务在自己 PC 上执行的时间。任务 4 每 10ms 执行一次[程序清单 1.25(3)]。

程序清单 1.25 任务 4

```

void Task4 (void *data)
{
    OS_EVENT *mbox;
    INT8U err;

    data = data;
    mbox = OSMBboxCreate((void *)0);
    for (;;) {
        OSMBboxPost(mbox, (void *)1); (1)
        OSMBboxPend(mbox, 0, &err); (2)
        OSTimeDlyHMSM(0, 0, 0, 10); (3)
    }
}

```

任务 5 除了延时一个时钟节拍以外什么也不做[程序清单 1.26(1)]。注意所有的任务都

```
void Task1(void *data)
{
    data = data;
    for(;;)
        OSTimeDly(3);
}
```

或者事件的发生而让出 CPU。如果始终占用 CPU，



同样，TaskClk()函数（程序清单 1.18）显示当前日期和时间。

1.9.3 注意

有些程序的细节只有请你仔细读一读 EX3L.C 才能理解。EX3L.C 中有 OSTaskSwHook()函数的代码，该函数用来测量每个任务的执行时间，可以用来统计每一个任务的调度频率，也可以统计每个任务运行时间的总和。这些信息将存储在每个任务的 TCB 扩展数据结构中。每次任务切换的时候 OSTaskSwHook()都将被调用。

每次任务切换发生的时候，OSTaskSwHook()先调用 PC_ElapsedStop()函数[程序清单 1.27(1)]来获取任务的运行时间[程序清单 1.27(1)]，PC_ElapsedStop()要和 PC_ElapsedStart()一起使用，上述两个函数用到了 PC 的定时器 2 (timer 2)。其中 PC_ElapsedStart()功能为启动定时器开始记数；而 PC_ElapsedStop()功能为获取定时器的值，然后清零，为下一次计数做准备。从定时器取得的计数将拷贝到 time 变量[程序清单 1.27(1)]。然后 OSTaskSwHook()调用 PC_ElapsedStart()重新启动定时器做下一次计数[程序清单 1.27(2)]。需要注意的是，系统启动后，第一次调用 PC_ElapsedStart()是在初始化代码中，所以第一次任务切换调用 PC_ElapsedStop()所得到的计数值没有实际意义，但这没有什么影响。如果任务分配了 TCB 扩展数据结构[程序清单 1.27(4)]，其中的计数器 TaskCtr 进行累加[程序清单 1.27(5)]。TaskCtr 可以统计任务被切换的频繁程度，也可以检查某个任务是否在运行。TaskExecTime [程序清单 1.27(6)]用来记录函数从切入到切出的运行时间，TaskTotExecTime[程序清单 1.27(7)]记录任务总的运行时间。统计每个任务的上述两个变量，可以计算出一段时间内各个任务占用 CPU 的百分比。OSTaskStatHook()函数会显示这些统计信息。

程序清单 1.27 用户定义的 OSTaskSwHook()

```
INT16U time;
TASK_USER_DATA *puser;
```

```
time = PC_ElapsedTime();  
PC_ElapsedStart();  
puser = OSTaskCur->OSTaskData;  
if (puser != (void *)0) {  
    puser->TaskCTerminate++;  
    puser->TaskRunTime += time;  
    puser->TaskTotalTime += time;  
}
```

公开的实时嵌入式操作系统



```
void OSTaskStatHook (void)  
{  
    char s[80];  
    INT32U i;  
    INT32U total;  
    INT32U per;  
  
    total = 0;  
    for (i = 0; i < 5; i++) {  
        total += TaskUserdata[i].TaskTotalTime;  
        DispTaskStat(i);  
    }  
    if (total > 0) {
```

将调用对外接口函数 OSTaskStatHook()（设置 N 为 1 允许对外接口函数）。统计任务每秒运行一次，本例中 OSTaskStatHook()用来计算并显示各任务占用 CPU 的情况。

OSTaskStatHook()函数中首先计算所有任务的运行时间[程序清单 1.28(1)]，DispTaskStat()用来将数字显示为 ASCII 字符[程序清单 1.28(2)]。然后计算每个任务运行时间的百分比[程序清单 1.28(3)]，并显示在合适的位置上 [程序清单 1.28(4)]。

程序清单 1.28 用户定义的 OSTaskStatHook()

```

for (i = 0; i < 7; i++) {
    pct = 100 * TaskUserData[i].TaskTotExecTime / total;      (3)
    sprintf(s, "t%d %d", pct);
    PC_DispStr(62, i + 11, s, DISP_FGND_YELLOW);           (4)
}
}

if (total > 10000000000L) {
    for (i = 0; i < 7; i++) {
        TaskUserData[i].TaskTotExecTime = 0L;
    }
}
}

```

从上面的代码中，我们可以看到在显示任务执行时间时，如果总执行时间超过了10000000000L，那么将把所有任务的执行时间都清零。这样做的原因是，如果显示的数据过大，将无法正常显示。

任务名	CPU使用率
任务1	10%
任务2	15%
任务3	20%
任务4	25%
任务5	30%
任务6	35%
任务7	40%

从上面的输出结果中，我们可以看到每行显示了任务名和CPU使用率。

（例程2.1）完成CPU使用率显示——终端台启动（源码）

从上面的输出结果中，我们可以看到每行显示了任务名和CPU使用率。

第2章

实时系统概念

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

实时系统的特点是，如果逻辑和时序出现偏差将会引起严重后果的系统。有两种类型的实时系统：软实时系统和硬实时系统。在软实时系统中系统的宗旨是使各个任务运行得越快越好，并不要求限定某一任务必须在多长时间内完成。

在硬实时系统中，各任务不仅要执行无误而且要做到准时。大多数实时系统是二者的结合。实时系统的应用涵盖的领域十分广泛，而多数实时系统又是嵌入式的。这意味着计算机内置在系统内部，用户看不到有个计算机在系统里面。以下是一些嵌入式系统的例子：

过程控制

食品加工
化工厂
汽车业
发动机控制
防抱死系统（ABS）
办公自动化
传真机
复印机
计算机外设
打印机
计算机终端
扫描仪
调制解调器

通信类

交换机
路由器
机器人
航空航天
飞机管理系统
武器系统
喷气发动机控制
民用消费品
微波炉
洗碗机
洗衣机
温度调节器

实时应用软件的设计一般比非实时应用软件设计难一些。本章讲述实时系统的概念。

2.0 前后台系统（Foreground/Background System）

不太复杂的小系统一般设计成如图 2.1 所示的样子。这种系统可称为前后台系统或超循环系统（super-loop）。应用程序是一个无限的循环，循环中调用相应的函数完成相应的

操作，这部分可以看成后台行为（background）。中断服务程序处理异步事件，这部分可以看成前台行为（foreground）。后台也可以叫做任务级，前台也叫中断级。时间相关性很强的关键操作（critical operation）一定是靠中断服务程序来保证的。因为中断服务提供的信息一直要等到后台程序走到该处理这个信息这一步时才能得到处理，这种系统在处理信息的及时性上，比实际可以做到的要差。这个指标称作任务级响应时间。最坏情况下的任务级响应时间取决于整个循环的执行时间。因为循环的执行时间不是常数，程序经过某一特定部分的准确时间也是不能确定的。进而，如果程序修改了，循环的时序也会受到影响。

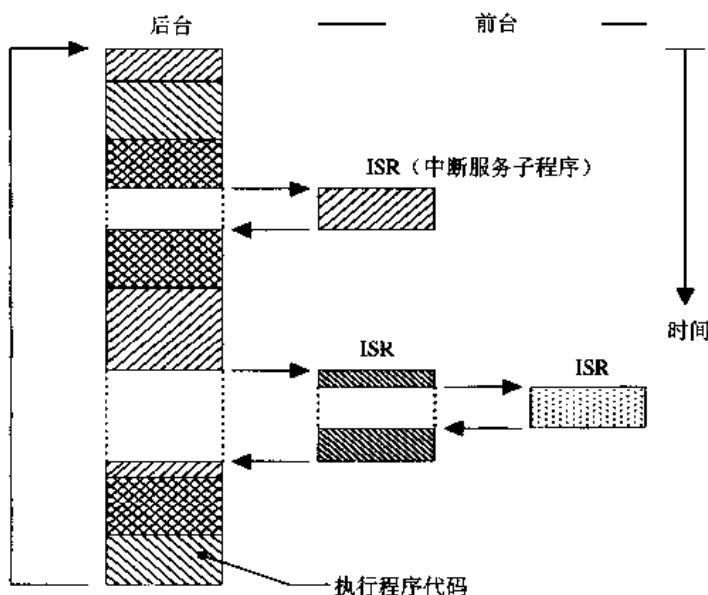


图2.1 前后台系统

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

2.1 代码的临界区

代码的临界区也称为临界区，指处理时不可分割的代码。一旦这部分代码开始执行，则不允许任何中断打入。为确保临界区代码的执行，在进入临界区之前要关中断，而临界区代码执行完以后要立即开中断。（参阅 2.03 节“共享资源”。）

2.2 资源

任何为任务所占用的实体都可称为资源。资源可以是输入输出设备，例如打印机、键

盘、显示器，资源也可以是一个变量、一个结构或一个数组等。



2.3 共享资源

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

2.4 多任务

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

2.5 任务

一个任务，也称作一个线程，是一个简单的程序，该程序可以认为 CPU 完全只属该程序自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间（如图 2.2 所示）。

典型地，每个任务都是一个无限的循环。每个任务都处在以下 5 种状态之一的状态下，这 5 种状态是休眠态、就绪态、运行态、挂起态（等待某一事件发生）和被中断态（参见图 2.3）休眠态相当于该任务驻留在内存中，但并不被多任务内核所调度。就绪态意味着该任务已经准备好，可以运行了，但由于该任务的优先级比正在运行的任务的优先级低，还暂时不能运行。运行态的任务是指该任务掌握了 CPU 的控制权，正在运行中。挂起状态也可以叫做等待事件态 WAITING，指该任务在等待，等待某一事件的发生，（例如等待某外设的 I/O 操作，等待某共享资源由暂不能使用变成能使用状态，等待定时脉冲的到来或等待超时信号的到来以结束目前的等待，等等）。最后，发生中断时，CPU 提供相应的中断服务，原来正在运行的任务暂不能运行，就进入了被中断状态。图 2.3 表示 μC/OS-II 中一些函数提供的服务，这些函数使任务从一种状态变到另一种状态。

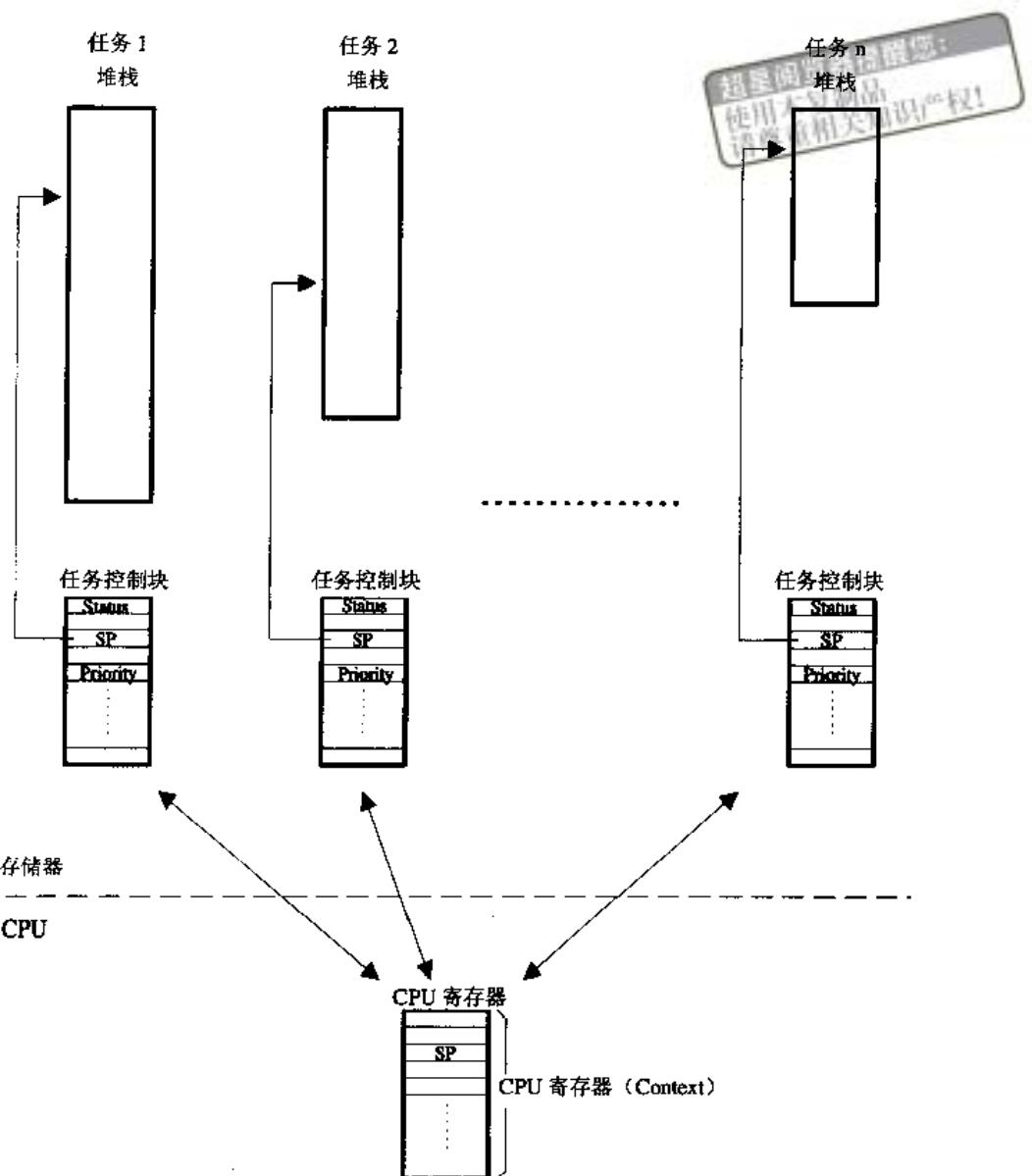


图2.2 多任务

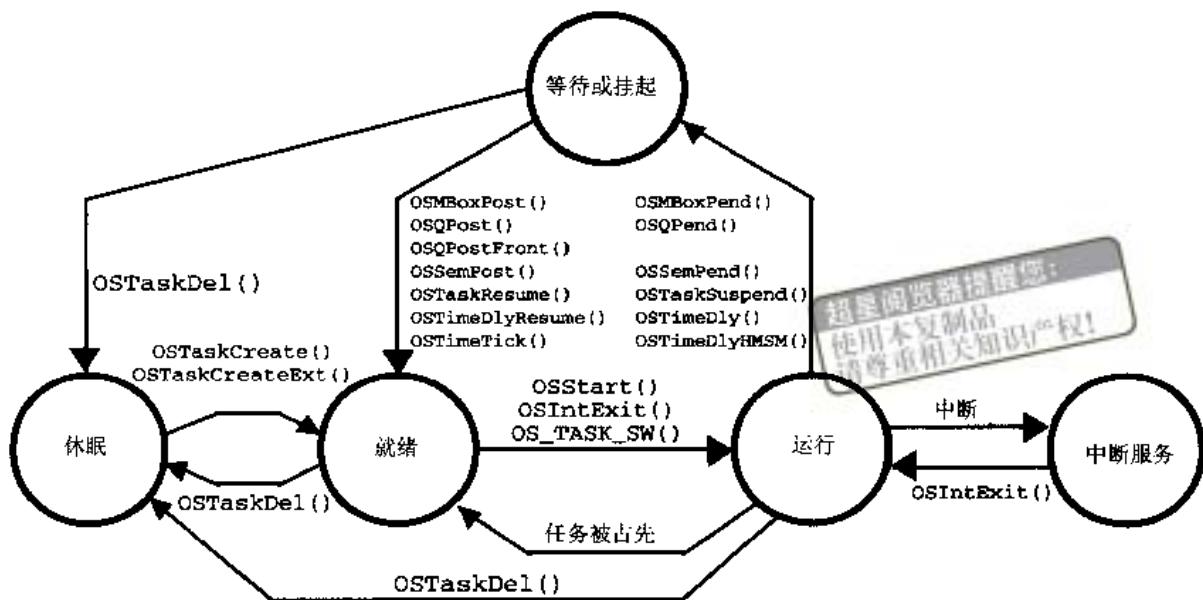


图2.3 任务的状态

2.6 任务切换

当多任务内核决定运行另外的任务时，它保存正在运行任务的当前状态（context），即 CPU 寄存器中的全部内容。这些内容保存在任务的当前状态保存区（task's context storage area），也就是任务自己的栈区之中（见图 2.2）。入栈工作完成以后，就把下一个将要运行的任务的当前状态从该任务的栈中重新装入 CPU 的寄存器，并开始下一个任务的运行。这个过程就称为任务切换（译注 4）。任务切换过程增加了应用程序的额外负载。CPU 的内部寄存器越多，额外负载就越重。做任务切换所需要的时间取决于 CPU 有多少寄存器要入栈。实时内核的性能不应该以每秒钟能做多少次任务切换来评价。

2.7 内核（Kernel）

多任务系统中，内核负责管理各个任务，或者说为每个任务分配 CPU 时间，并且负责任务之间的通信。内核提供的基本服务是任务切换。之所以使用实时内核可以大大简化应用系统的设计，是因为实时内核允许将应用分成若干个任务，由实时内核来管理它们。内核本身也增加了应用程序的额外负载，代码空间增加 ROM 的用量，内核本身的数据结构

译注 4：context switch 又称 task switch，在有的书中翻译成上下文切换，实际含义是任务切换，或 CPU 寄存器内容切换。

增加了 RAM 的用量。但更主要的是，每个任务要有自己的栈空间，这一块占起内存来是相当厉害的。内核本身对 CPU 的占用时间一般在 2 到 5 个百分点之间。

单片机一般不能运行实时内核，因为单片机的 RAM 很有限。通过提供必不可少的系统服务，诸如信号量管理，邮箱、消息队列、延时等，实时内核使得 CPU 的利用更为有效。一旦读者用实时内核做过系统设计，将决不再想返回到前后台系统。

2.8 调度 (Scheduler)

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

2.9 非占先式内核

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

非占先式内核的一个优点是响应中断快。在讨论中断响应时会进一步涉及这个问题。在任务级，非占先式内核允许使用不可重入函数。函数的可重入性以后会讨论。每个任务都可以调用非可重入性函数，而不必担心其他任务可能正在使用该函数，从而造成数据的破坏。因为每个任务要运行到完成时才释放 CPU 的控制权。当然该不可重入型函数本身不得有放弃 CPU 控制权的企图。

使用非占先式内核时，任务级响应时间比前后台系统快得多。此时的任务级响应时间取决于最长的任务执行时间。

非占先式内核的另一个优点是，几乎不需要使用信号量保护共享数据。运行着的任务占有 CPU，而不必担心被别的任务抢占。但这也不是绝对的，在某种情况下，信号量还是用得着的。处理共享 I/O 设备时仍需要使用互斥型信号量。例如，在打印机的使用上，仍需要满足互斥条件。图 2.4 示意非占先式内核的运行情况，任务在运行过程之中，[L2.4(1)] 中断来了，如果此时中断是开着的，CPU 由中断向量[图 2.4(2)]进入中断服务子程序，中断服务子程序做事件处理[图 2.4(3)]，使一个有更高级的任务进入就绪态。中断服务完成以后，中断返回指令[图 2.4(4)]，使 CPU 回到原来被中断的任务，接着执行该任务的代码[图 2.4(5)]



直到该任务完成，调用一个内核服务函数以释放 CPU 控制权，由内核将控制权交给那个优先级更高的、并已进入就绪态的任务[图 2.4(6)]，这个优先级更高的任务才开始处理中断服务程序标识的事件[图 2.4(7)]。

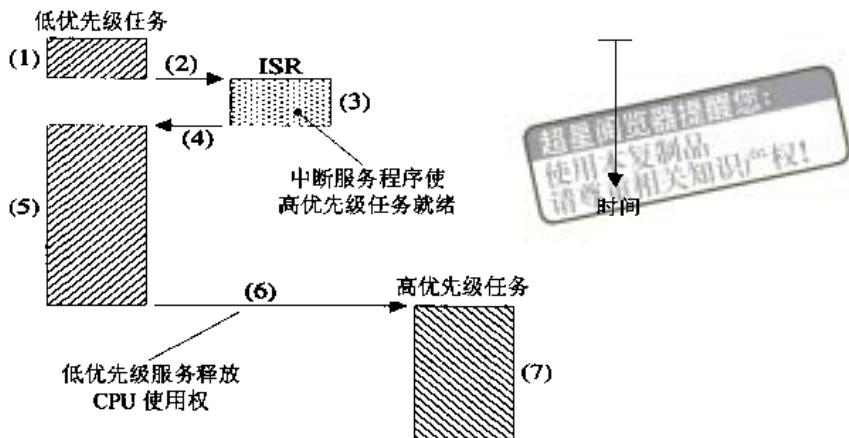


图2.4 非占先式内核

非占先式内核的最大缺陷在于其响应时间。高优先级的任务已经进入就绪态，但还不能运行，也许要等很长时间，直到当前运行着的任务释放 CPU。与前后系统一样，非占先式内核的任务级响应时间是不确定的，不知道什么时候最高优先级的任务才能拿到 CPU 的控制权，完全取决于应用程序什么时候释放 CPU。

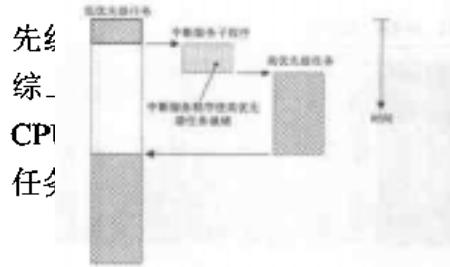
总之，非占先式内核允许每个任务运行，直到该任务自愿放弃 CPU 的控制权。中断可以打入运行着的任务。中断服务完成以后将 CPU 控制权还给被中断了的任务。任务级响应时间要大大好于前后系统，但仍是不可知的，商业软件几乎没有非占先式内核。

2.10 占先式内核

当系统响应时间很重要时，要使用占先式（preemptive）内核。因此，μC/OS-II 以及绝大多数商业上销售的实时内核都是占先式内核。最高优先级的任务一旦就绪，总能得到 CPU 的控制权。当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前任务的 CPU 使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了 CPU 的控制权。如果是中断服务子程序使一个高优先级的任务进入就绪态，中断完成时，中断了的任务被挂起，优先级高的那个任务开始运行。如图 2.5 所示。

使用占先式内核，最高优先级的任务什么时候可以执行，可以得到 CPU 的控制权是可知的。使用占先式内核使得任务级响应时间得以最优化。

使用占先式内核时，应用程序不应直接使用不可重入型函数。调用不可重入型函数时，要满足互斥条件，这一点可以用互斥型信号量来实现。如果调用不可重入型函数时，低优



先级任务剥夺，不可重入型函数中的数据有可能被破坏。
绪态的高优先级的任务先运行，中断服务程序可以抢占
上此时优先级最高的任务运行（不一定是那个被中断了的
了最优化，而且是可知的。 μ C/OS-II 属于占先式内核。



```
void interrupt(uchar *data, uchar *addr)
{
    while (*data++ < *addr++)
        ;
    *data = 0x55;
}
```

图2.5 占先式内核

2.11 可重入性 (Reentrancy)

可重入型函数可以被一个以上的任务调用，而不必担心数据的破坏。可重入型函数任何时候都可以被中断，一段时间以后又可以运行，而相应数据不会丢失。可重入型函数或者只使用局部变量，即变量保存在 CPU 寄存器中或堆栈中。如果使用全局变量，则要对全局变量予以保护。程序 2.1 是一个可重入型函数的例子。

程序清单 2.1 可重入型函数

函数 `Strcpy()` 做字符串复制。因为参数是存在堆栈中的，故函数 `strcpy()` 可以被多个任务互相破坏对方的指针。

示。`swap()` 是一个简单函数，它使函数的两个形式是占先式内核，中断是开着的，`Temp` 定义为整数

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

程序清单 2.2 不可重入型函数



程序员打算让 `swap()` 函数可以为任何任务所调用，如果一个低优先级的任务正在执行 `swap()` 函数，而此时中断发生了，于是可能发生的事情如图 2.6 所示。[图 2.6(1)] 表示中断发生时 `Temp` 已被赋值 1，中断服务子程序使更优先级的任务就绪，当中断完成时[图 2.6(2)]，内核（假定使用的是 μC/OS-II）使高优先级的那个任务得以运行[图 2.6(3)]，高优先级的任务调用 `swap()` 函数是 `Temp` 赋值为 3。这对该任务本身来说，实现两个变量的交换是没有问题的，交换后 Z 的值是 4，X 的值是 3。然后高优先级的任务通过调用内核服务函数中的延迟一个时钟节拍[图 2.6(4)]，释放了 CPU 的使用权，低优先级任务得以继续运行[图 2.6(5)]。注意，此时 `Temp` 的值仍为 3！在低优先级任务接着运行时，Y 的值被错误地赋值为 3，而不是正确值 1。

请注意，这只是一个简单的例子，如何能使代码具有可重入性一看就明白。然而有些情况下，问题并不那么容易解决。应用程序中的不可重入函数引起的错误很可能在测试时发现不了，直到产品到了现场问题才出现。如果在多任务上你还是把新手，使用不可重入型函数时，千万要当心。

使用以下技术之一即可使 `swap()` 函数具有可重入性：

- 把 `Temp` 定义为局部变量；
- 调用 `swap()` 函数之前关中断，调动后再开中断；
- 用信号量禁止该函数在使用过程中被再次调用。

如果中断发生在 `swap()` 函数调用之前或调用之后，两个任务中的 x, y 值都会是正确的。

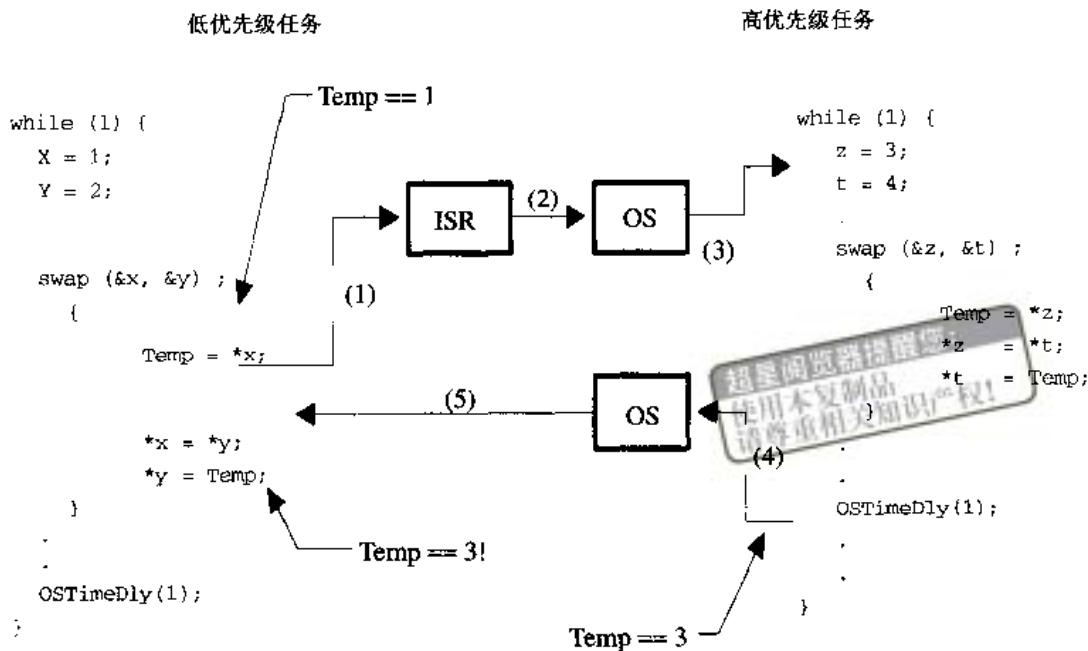


图2.6 非重入函数

2.12 时间片轮番调度法 (round-robin scheduling)

当两个或两个以上任务有同样优先级，内核允许一个任务运行事先确定的一段时间，叫做时间额度 (*quantum*)，然后切换给另一个任务，也叫做时间片调度 (*time slicing*)。内核在满足以下条件时，把 CPU 控制权交给下一个任务就绪态的任务：

- 当前任务已无事可做；
- 当前任务在时间片还没结束时已经完成了。

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

2.13 任务优先级

每个任务都有其优先级 (priority)。任务越重要，赋予的优先级应越高。

2.14 静态优先级

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



2.15 动态优先级

应用程序执行过程中，任务的优先级是可变的，则称之为动态优先级。实时内核应当避免出现优先级反转问题。

2.16 优先级反转

使用实时内核，优先级反转问题是实时系统中出现得最多的问题。图 2.7 解释优先级反转是如何出现的。如图，任务 1 优先级高于任务 2，任务 2 优先级高于任务 3。任务 1 和任务 2 处于挂起状态，等待某一事件的发生，任务 3 正在运行如[图 2.7(1)]。此时，任务 3 要使用其共享资源。使用共享资源之前，首先必须得到该资源的信号量（Semaphore）（见 2.18.04 信号量）。任务 3 得到了该信号量，并开始使用该共享资源[图 2.7(2)]。由于任务 1 优先级高，它等待的事件到来之后剥夺了任务 3 的 CPU 使用权[图 2.7(3)]，任务 1 开始运行[图 2.7(4)]。运行过程中任务 1 也要使用那个任务 3 正在使用着的资源，由于该资源的信号量还被任务 3 占用着，任务 1 只能进入挂起状态，等待任务 3 释放该信号量[图 2.7(5)]。

优先级反转

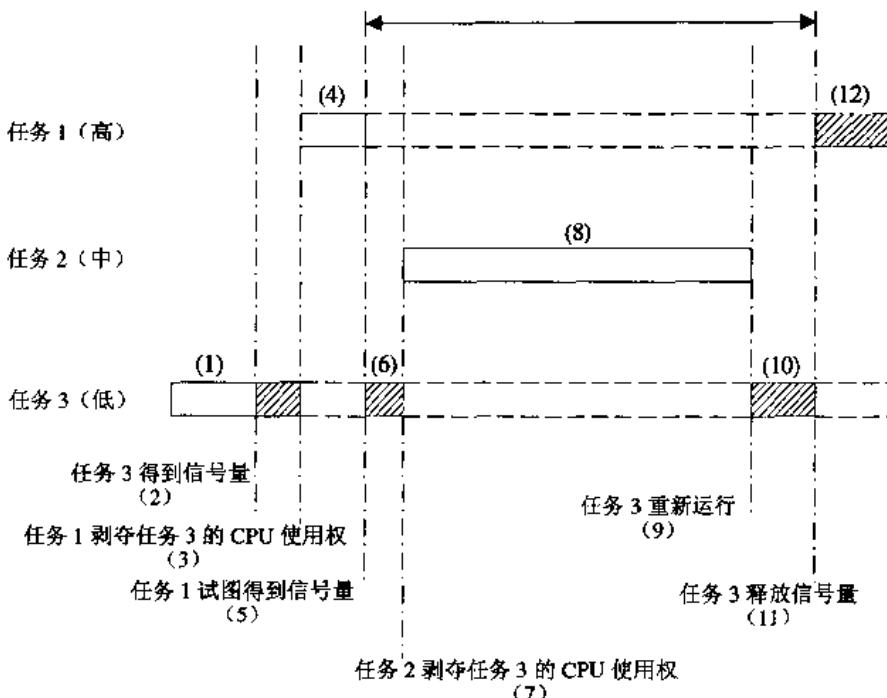


图 2.7 优先级反转问题

任务3得以继续运行[图2.7(6)]。由于任务2的优先级高于任务3，当任务2等待的事件发生后，任务2剥夺了任务3的CPU的使用权[图2.7(7)]并开始运行。处理它该处理的事件[图2.7(8)]，直到处理完之后将CPU控制权还给任务3[图2.7(9)]。任务3接着运行[图2.7(10)]，直到释放那个共享资源的信号量[图2.7(11)]。直到此时，由于实时内核知道有个高优先级的任务在等待这个信号量，内核进行任务切换，使任务1得到该信号量并接着运行[图2.7(12)]。

在这种情况下，任务1优先级实际上降到了任务3的优先级水平。因为任务1要等到任务3释放占有的那个共享资源。由于任务2剥夺任务3的CPU使用权，使任务1的状况更加恶化，任务2使任务1增加了额外的延迟时间。任务1和任务2的优先级发生了反转。

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

图2.8解释如果内核支持优先级继承的话，在上述例子中会是怎样一个过程。任务3在

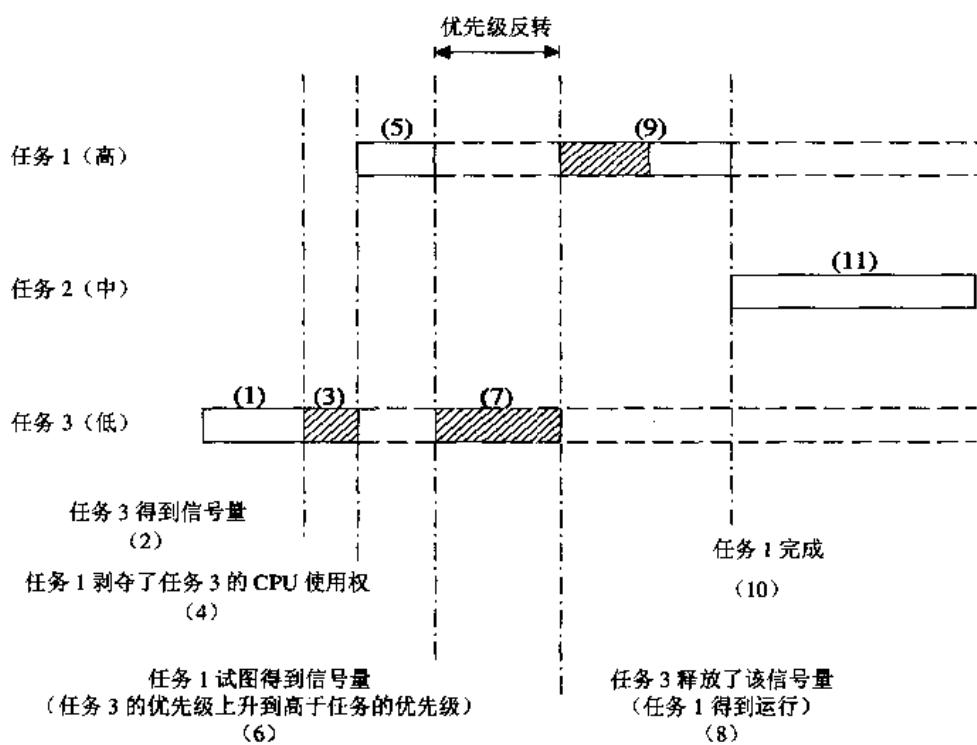


图2.8 支持优先级继承的内核



运行[图 2.8(1)]，任务 3 申请信号量以获得共享资源使用权[图 2.8(2)]，任务 3 得到并开始使用共享资源[图 2.8(3)]。后来 CPU 使用权被任务 1 剥夺[图 2.8(4)]，任务 1 开始运行[图 2.8(5)]，任务 1 申请共享资源信号量[图 2.8(6)]。此时，内核知道该信号量被任务 3 占用了，而任务 3 的优先级比任务 1 低，内核于是将任务 3 的优先级升至与任务 1 一样，然而回到任务 3 继续运行，使用该共享资源[图 2.8(7)]，直到任务 3 释放共享资源信号量[图 2.8(8)]。这时，内核恢复任务 3 本来的优先级并把信号量交给任务 1，任务 1 得以顺利运行。[图 2.8(9)]，任务 1 完成以后[图 2.8(10)]那些任务优先级在任务 1 与任务 3 之间的任务例如任务 2 才能得到 CPU 使用权，并开始运行[图 2.8(11)]。注意，任务 2 在从[图 2.8(3)]到[图 2.8(10)]的任何一刻都有可能进入就绪态，并不影响任务 1、任务 3 的完成过程。在某种程度上，任务 2 和任务 3 之间也还是有不可避免的优先级反转。

2.17 任务优先级分配

给任务定优先级可不是件小事，因为实时系统相当复杂。许多系统中，并非所有的任务都至关重要。不重要的任务自然优先级可以低一些。实时系统大多综合了软实时和硬实时这两种需求。软实时系统只是要求任务执行得尽量快，并不要求在某一特定时间内完成。硬实时系统中，任务不但要执行无误，还要准时完成。

一项有意思的技术可称之为单调执行率调度法 RMS (Rate Monotonic Scheduling)，用于分配任务优先级。这种方法基于哪个任务执行的次数最频繁，执行最频繁的任务优先级最高。见图 2.9。

RMS 做了一系列假设：

- 所有任务都是周期性的；
- 任务间不需要同步，没有共享资源，没有任务间数据交换等问题；
- CPU 必须总是执行那个优先级最高且处于就绪态的任务。换句话说，要使用抢占式调度法。

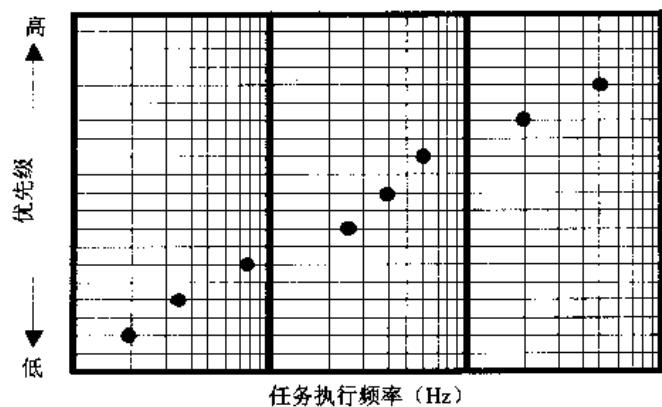


图2.9 基于任务执行频繁度的优先级分配法



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

$$\sum_i \frac{E_i}{T_i} \leq n(2^{1/n} - 1) \quad [2.1]$$

这里 E_i 是任务 i 的最长执行时间， T_i 是任务 i 的执行周期。换句话说， E_i/T_i 是任务 i 所占的 CPU 使用率。对于无穷多个任务，根据 RMS 定理，要使所有任务都满足硬实时条件，所有任务的 CPU 使用率之和必须小于 70% ！请注意，这是指有时间条件要求的任务，如果所有任务的 CPU 使用率之和等于 70% ，使得 CPU 的利用率达到 100% ，那么程序就没有了修改的余地，也没法增加新功能了。

作为系统设计的一条原则，CPU 利用率应小于 60% 到 70% 。

RMS 认为最高执行率的任务具有最高的优先级，但最某些情况下，最高执行率的任务并非是最重要的任务。如果实际应用都真的像 RMS 说的那样，也就没有什么优先级分配可讨论了。然而讨论优先级分配问题，RMS 无疑是一个有意思的起点。

表 2.1

基于任务到 CPU 最高允许使用率

2.18 互斥条件

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

- 关中断；
- 使用测试并置位指令；
- 禁止做任务切换；



2.18.1 关中断和开中断

处理共享数据时保证互斥，最简便快捷的办法是关中断和开中断。如示意性代码程序

```
2.18.1
void function(void)
{
    OS_ENTER_CRITICAL();
    /*在这里处理共享数据*/
    OS_EXIT_CRITICAL();
}
```

μC/OS-II在处理内部变量和数据结构时就是使用的这种手段，即使不是全部，也是绝大部分。实际上μC/OS-II提供两个宏调用，允许用户在应用程序的C代码中关中断然后再开中断：`OS_ENTER_CRITICAL()`和`OS_EXIT_CRITICAL()`[参见 8.3.2]，这两个宏调用的用法见程序清单 2.4。

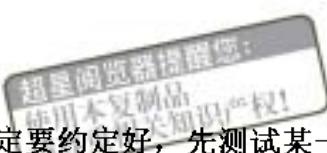
程序清单 2.4 利用μC/OS-II宏调用关中断和开中断

可是，必须十分小心，关中断的时间不能太长。因为它影响整个系统的中断响应时间，即中断延迟时间。当改变或复制某几个变量的值时，应想到用这种方法来做。这也是在中断服务子程序中处理共享变量或共享数据结构的唯一方法。在任何情况下，关中断的时间都要尽量短。

如果使用某种实时内核，一般地说，关中断的最长时间不超过内核本身的关中断时间，就不会影响系统中断延迟。当然得知道内核里中断关了多久。凡好的实时内核，厂商都提

供这方面的数据。总而言之，要想出售实时内核，时间特性最重要。

2.18.2 测试并置位



共享一个资源时，一定要约定好，先测试某一全程共享资源打交道。为防止另一任务也要使用该资源，常称作测试并置位（Test-And-Set），或称作 TAS。会被中断的指令，或者是在程序中关中断做 TAS 操作。

程序清单 2.5 利用测试并置位处理共享资源

有的微处理器有硬件的 TAS 指令（如 Motorola 68000 系列，就有这条指令）。

2.18.3 禁止,然后允许任务切换

如果任务不与中断服务子程序共享变量或数据结构，可以使用禁止、然后允许任务切换（参见 3.6 节）。如程序清单 2.6 所示，以μC/OS-II 的使用为例，两个或两个以上的任务可以共享数据而不发生竞争。注意，此时虽然任务切换是禁止了，但中断还是开着的。如果这时中断来了，中断服务子程序会在这一临界区内立即执行。中断服务子程序结束时，尽管有优先级高的任务已经进入就绪态，内核还是返回到原来被中断了的任务。直到执行完给任务切换开锁函数 OSSchedUnlock()，内核再看有没有优先级更高的任务被中断服务子程序激活而进入就绪态，如果有，则做任务切换。虽然这种方法是可行的，但应该尽量避免禁止任务切换之类操作，因为内核最主要的功能就是做任务的调度与协调。禁止任务切换显然与内核的初衷相违。应该使用下述方法。

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

开锁的方法实现数据共享



2.18.4 信号量 (Semaphore)

信号量（译注 5）是 60 年代中期 Edgser Dijkstra 发明的。信号量实际上是一种约定机制，在多任务内核中普遍使用信号量用于：

- 控制共享资源的使用权（满足互斥条件）；
- 标志某事件的发生；
- 使两个任务的行为同步。

信号像是一把钥匙，任务要运行下去，得先拿到这把钥匙。如果信号已被别的任务占用，该任务只得被挂起，直到信号被当前使用者释放。换句话说，申请信号的任务是在说：“把钥匙给我，如果谁正在用着，我只好等！”信号是只有两个值的变量，信号量是计数式的。只取两个值的信号是只有两个值 0 和 1 的量，因此也称之为信号量。计数式信号量的值可以是 0 到 255 或 0 到 65535，或 0 到 4294967295，取决于信号量规约机制使用的是 8 位、16 位还是 32 位。到底是几位，实际上是取决于用的哪种内核。根据信号量的值，内核跟踪那些等待信号量的任务。

一般地说，对信号量只能实施三种操作：初始化 (INITIALIZE)，也可称作建立 (CREATE)；等信号 (WAIT) 也可称作挂起 (PEND)；给信号 (SIGNAL) 或发信号 (POST)。信号量初始化时要给信号量赋初值，等待信号量的任务表 (Waiting list) 应清为空。

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

译注 5：信号与信号量在英文中都叫做 semaphore，并不加以区分，但它有两种类型，二进制型 (binary) 和计数器型 (counting)。本书中的二进制型信号量实际上是只取两个值 0 和 1 的信号量。实际上这个信号量只有一位，这种信号量翻译为信号更为贴切。而二进制信号量通常指若干位的组合。而本书中解释为事件标志的置位与清除（见 2.21）。

任务以发信号操作（SIGNAL）释放信号量。如果没有任务在等待信号量，信号量的值仅仅是简单地加 1。如果有任务在等待该信号量，那么就会有一个任务进入就绪态，信号量的值也就不加 1。于是钥匙给了等待信号量的诸任务中的一个任务。至于给了那个任务，要看内核是如何调度的。收到信号量的任务可能是以下两者之一。

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

有的内核只选择后者，首次申请共享信号量初始化时选定上述两种方法中的一种。但μC/OS-II 内核则选择前者。

E任务比当前运行的任务优先级高（假设，是当前任务主任务）。则内核做任务切换（假设，使用的是抢占式任务被挂起。直到又变成就绪态中优先级最高任务。如何用信号量处理共享数据。要与同一共享数据打 iPend()。处理完共享数据以后再调用释放信号量函数 OSSemPost()。这两个函数将在以后的章节中描述。要注意的是，在使用信号量之前，一定要对该信号量做初始化。作为互斥条件，信号量初始化为 1。使用信号量处理共享数据不增加中断延迟时间，如果中断服务程序或当前任务激活了一个高优先级的任务，高优先级的任务立即开始执行。

程序清单 2.7 通过获得信号量处理共享数据

当诸任务共享输入输出设备时，信号量特别有用。可以想象，如果允许两个任务同时给打印机送数据时会出现什么现象。打印机会打出相互交叉的两个任务的数据。例如任务 1 要打印“*I am Task!*”，而任务 2 要打印“*I am Task2!*”可能打印出来的结果是：“*I Ia amm T Tasask k1!2!*”。

在这种情况下，使用信号量并给信号量赋初值 1（用二进制信号量）。规则很简单，要想使用打印机的任务，先要得到该资源的信号量。图 2.10 两个任务竞争得到排它性打印机使用权，图中信号量用一把钥匙表示，想使用打印机先要得到这把钥匙。

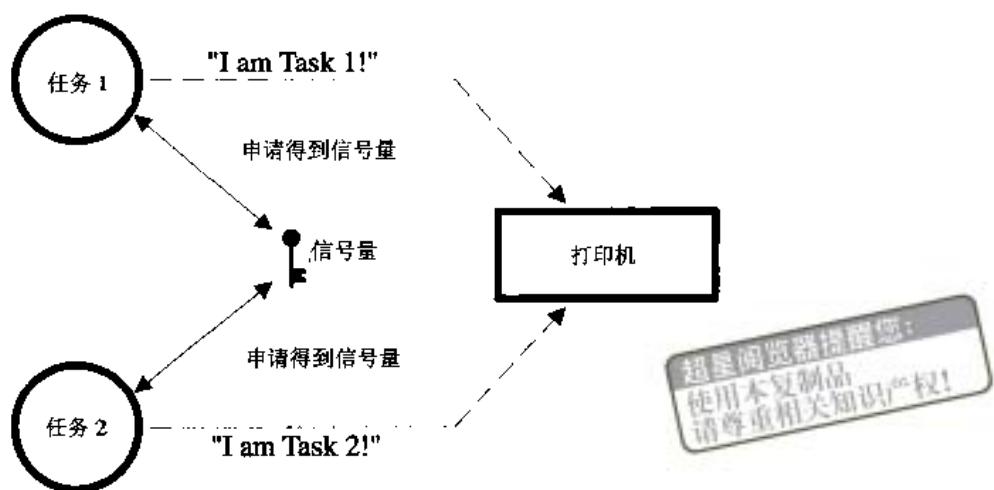


图2.10 用获取信号量来得到打印机使用权

上例中，每个任务都知道有个信号表示资源可不可以使用。要想使用该资源，要先得
到这个信号。然而在此情况下，最好把信号量藏起来，各个任务在同某一资源打交道时，
只能通过一个公共的入口函数（如图 2.10 所示）来访问该信号量。例如，多任务共享一个 RS-232C 外设接口，各
任务要送命令给接口另一端的设备并接收该设备的回应。如图 2.11 所示。

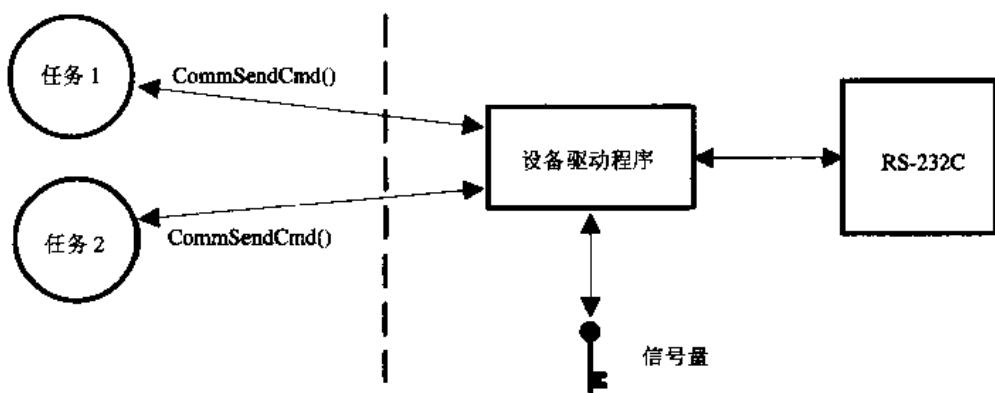


图2.11 在任务级看不到隐含的信号量

调用向串行口发送命令的函数 CommSendCmd(), 该函数有三个形式参数：Cmd 指向
送出的 ASCII 码字符串命令。Response 指向外设回应的字符串。timeout 指设定的时间间隔。
如果超过这段时间外设还不响应，则返回超时错误信息。函数的示意代码如程序清单 2.8
所示。

程序清单 2.8 隐含的信号量

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

实时系统概念

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

```
BUF *BufReq(void)
{
    BUR *pBur;

    Acquire a semaphore;
    Disable interrupts;
    pLE = ->BufFreeList;
    BufFreeList = pLE->BufNext;
    Enable interrupts;
    return pLE;
}
```

述函数。设信号量初值为 1，表示允许使用。初始时的。第一个调用 `CommSendCmd()` 函数的任务申请并等待响应。而另一个任务也要送命令，此时外设信号量重新被释放。第二个任务看起来同调用了一个普通函数一样，只不过这个函数在没有完成其相应功能时不返回。当第一个任务释放了那个信号量，第二个任务得到了该信号量，第二个任务才能使用 RS-232 口。

计数式信号量用于某资源可以同时为几个任务所用。例如，用信号量管理缓冲区阵列 (buffer pool)，如图 2.12 所示。缓冲区阵列中共有 10 个缓冲区，任务通过调用申请缓冲区函数 `BufReq()` 向缓冲区管理方申请得到缓冲区使用权。当缓冲区使用权还不再需要时，通过调用释放缓冲区函数 `BufRel()` 将缓冲区还给管方。函数示意码如程序清单 2.9 所示。

程序清单 2.9 用信号量管理缓冲区

```

void BufSel(OSIF *pcr)
{
    Disable_Interrupts();
    pcr->student = MulticastList;
    BufFreeList = pcr;
    Enable_Interrupts();
    Release_Semaphores();
}

```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

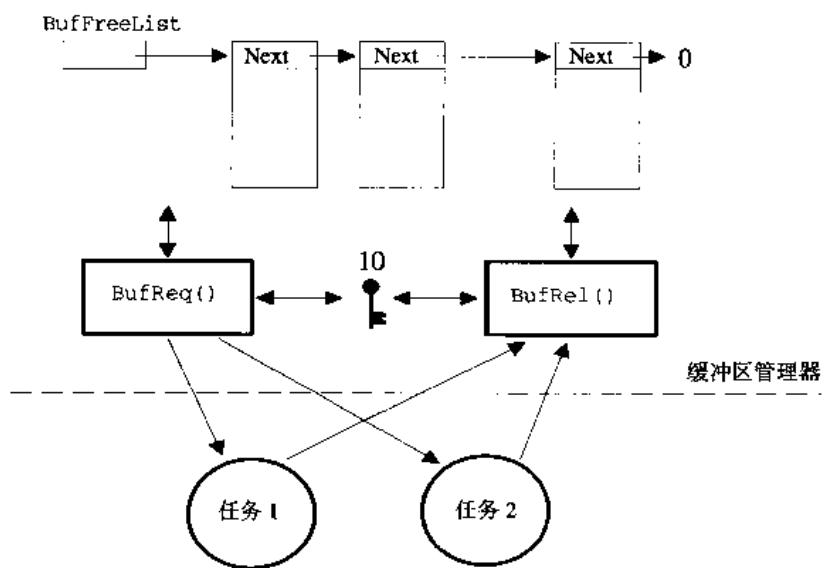


图2.12 计数式信号量的用法

缓冲区阵列管理方满足前十个申请缓冲区的任务，就好像有 10 把钥匙可以发给诸任务。当所有的钥匙都用完了，申请缓冲区的任务被挂起，直到信号量重新变为有效。缓冲区管理程序在处理链表指针时，为满足互斥条件，中断是关掉的（这一操作非常快）。任务使用完某一缓冲区，通过调用缓冲区释放函数 `BufRel()` 将缓冲区还给系统。系统先将该缓冲区指针插入到空闲缓冲区链表（Linked list）中然后再给信号量加 1 或释放该信号量。这一过程隐含在缓冲区管理程序 `BufReq()` 和 `BufRel()` 之中，调用这两个函数的任务不用管函数内部的详细过程。

信号量常被用过了头。处理简单的共享变量也使用信号量则是多余的。请求和释放信号量的过程是要花相当的时间的。有时这种额外的负荷是不必要的。用户可能只需要关中断、开中断来处理简单共享变量，以提高效率（参见 2.18.1）。假如两个任务共享一个 32 位的整数变量，一个任务给这个变量加 1，另一个任务给这个变量清 0。如果注意到不管哪种操作，对微处理器来说，只花极短的时间，就不会使用信号量来满足互斥条件了。每个

任务只需操作这个任务前关中断，之后再开中断就可以了。然而，如果这个变量是浮点数，而相应微处理器又没有硬件的浮点协处理器，浮点运算的时间相当长，关中断时间长了会影响中断延迟时间，这种情况下就有必要使用信号量了。

超星阅读器
使用本资源请
尊重相关知识产权！

2.19 死锁

死锁 (dead lock) 也称作抱死 (deadly embrace)，指两个任务无限期地互相等待对方控制着的资源。设任务 T1 正独享资源 R1，任务 T2 在独享资源 R2，而此时 T1 又要独享 R2，T2 也要独享 R1，于是哪个任务都没法继续执行了，发生了死锁。最简单的防止发生死锁的方法是让每个任务：

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

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

2.20 同步

可以利用信号量使某任务与中断服务程序同步（或者是与另一个任务同步，这两个任务间没有数据交换）。如图 2.13 所示。注意，图中用一面旗帜，或称作一个标志表示信号量。这个标志表示某一事件的发生（不再是一把用来保证互斥条件的钥匙）。用来实现同步机制的信号量初始化成 0，信号量用于这种类型同步的称作单向同步 (unilateral rendezvous)。一个任务做 I/O 操作，然后等信号回应。当 I/O 操作完成，中断服务程序（或另外一个任务）发出信号，该任务得到信号后继续往下执行。

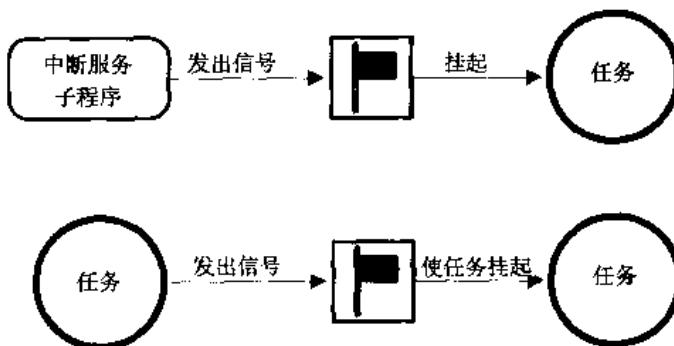
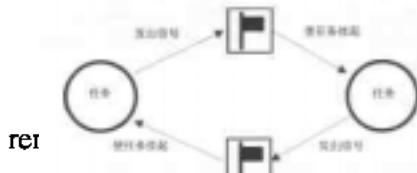


图2.13 用信号量使任务与中断服务同步



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

- 发信号给等待事件发生的任务中优先级最高的任务；
- ~~当且仅是发生过事件的那个任务。~~



示识别事件发生的中断服务或任务也可以是多个。

同步它们的行为。如图 2.14 所示。这叫做双向同步 (bilateral synchronization) 类似，只是两个任务要相互同步。

运行到某一处的第一个任务发信号给第二个任务 [程序清单 2.10(1)]，然后等待信号返回 [程序清单 2.10(2)]。同样，当第二个任务运行到某一处时发信号给第一个任务 [程序清单 2.10(3)] 等待返回信号 [程序清单 2.10(4)]。至此，两个任务实现了

用双向同步，因为在中断服务中不可能等一个信号。

```

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

```

图2.14 两个任务用信号量彼此同步的行为

程序清单 2.10 双向同步

```

task(1)
{
    for (i=1; i<5; i++)
        Perform operation;
    Signal task #1;          (3)
    Wait for signal from task #1; (4)
    Continue operation;
}

```



2.21 事件标志

当某任务要与多个事件同步时，要使用事件标志（event flag）。若任务需要与任何事件之一发生同步，可称为独立型同步（disjunctive synchronization，即逻辑或关系）。任务也可以与若干事件都发生了同步，称之为关联型（conjunctive synchronization，逻辑与关系）。独立型及关联型同步如图 2.15 所示。

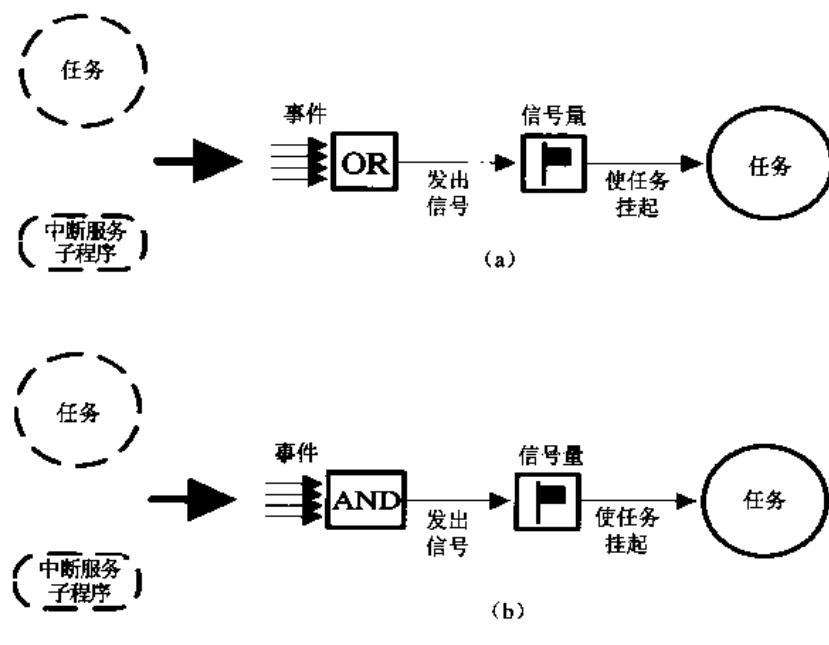


图2.15 独立型及关联型同步

可以用多个事件的组合发信号给多个任务。如图 2.16 所示，典型地，8 个、16 个或 32 个事件可以组合在一起，取决于用的哪种内核。每个事件占 1 位（bit），以 32 位的情况为

多。任务或中断服务可以给某一位置位或复位，当任务所需的事件都发生了，该任务继续执行，至于哪个任务该继续执行了，是在一组新的事件发生时确定的。也就是在事件位置位时做判断。

内核支持事件标志，提供事件标志置位、事件标志清零和等待事件标志等服务。事件标志可以是独立型或组合型。 μ C/OS-II 目前不支持事件标志。

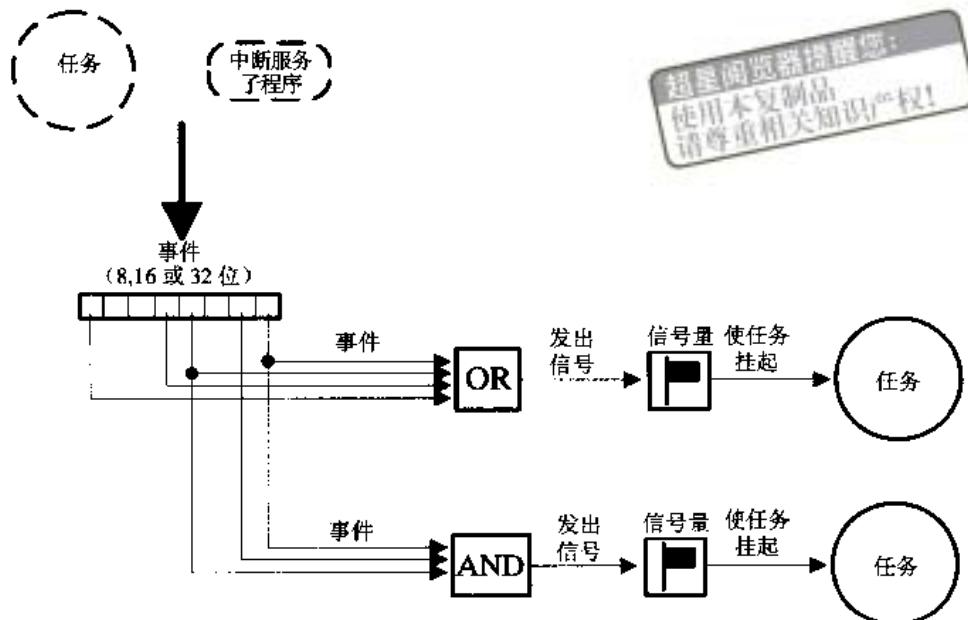


图2.16 事件标志

2.22 任务间的通信

有时很需要任务间的或中断服务与任务间的通信。这种信息传递被称为任务间的通信 (inter task communication)。任务间信息的传递有两个途径：通过全程变量或发消息给另一个任务。

用全程变量时，必须保证每个任务或中断服务程序独享该变量。中断服务中保证独享的惟一办法是关中断。如果两个任务共享某变量，各任务实现独享该变量的办法可以是关中断再开中断，或使用信号量（如前面提到的那样）。请注意，任务只能通过全程变量与中断服务程序通信，而任务并不知道什么时候全程变量被中断服务程序修改了，除非中断程序以信号量方式向任务发信号或者是该任务以查询方式不断周期性地查询变量的值。要避免这种情况，用户可以考虑使用邮箱或消息队列。

2.23 消息邮箱

通过内核服务可以给任务发送消息。典型的消息邮箱（message mail box）也称作消息交换（message exchange），是用一个指针型变量，通过内核服务，一个任务或一个中断服务程序可以把一则消息（即一个指针）放到邮箱里去。同样，一个或多个任务可以通过内核服务接收这则消息。发送消息的任务和接收消息的任务约定，该指针指向的内容就是那则消息。

每个邮箱有相应的正在等待消息的任务列表，要得到消息的任务会因为邮箱是空的而被挂起，且被记录到等待消息的任务表中，直到收到消息。一般地说，内核允许用户定义等待超时，等待消息的时间超过了，仍然没有收到该消息，这任务进入就绪态，并返回出错信息，报告等待超时错误。消息放入邮箱后，或者是把消息传给等待消息的任务表中优先级最高的那个任务（基于优先级），或者是将消息传给最先开始等待消息的任务（基于先进先出）。图 2.17 示意把消息放入邮箱。用一个 I 字表示邮箱，旁边的小砂漏表示超时计时器，计时器旁边的数字表示定时器设定值，即任务最长可以等多少个时钟节拍（Clock Tick），关于时钟节拍以后会讲到。

内核一般提供以下邮箱服务：

- 邮箱内消息的内容初始化，邮箱里最初可以有，也可以没有消息；
- 将消息放入邮箱（POST）；
- 等待有消息进入邮箱（PEND）；
- 如果邮箱内有消息，就接受这则消息。如果邮箱里没有消息，则任务并不被挂起（ACCEPT），用返回代码表示调用结果，是收到了消息还是没有收到消息。

消息邮箱也可以当作只取两个值的信号量来用。邮箱里有消息，表示资源可以使用，而空邮箱表示资源已被其他任务占用。

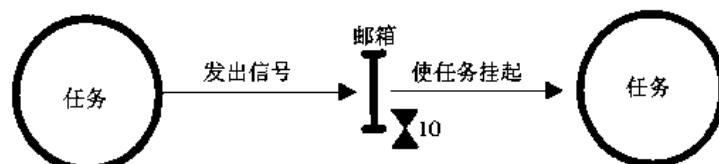


图2.17 消息邮箱

2.24 消息队列

消息队列（message queue）用于给任务发消息。消息队列实际上是邮箱阵列。通过内

核提供的服务，任务或中断服务子程序可以将一条消息（该消息的指针）放入消息队列。同样，一个或多个任务可以通过内核服务从消息队列中得到消息。发送和接收消息的任务约定，传递的消息实际上是传递的指针指向的内容。通常，先进入消息队列的消息先传给任务，也就是说，任务先得到的是最先进入消息队列的消息，即先进先出原则（FIFO）。然而μC/OS-II也允许使用后进先出方式（LIFO）。

像使用邮箱那样，当一个以上的任务要从消息队列接收消息时，每个消息队列有一张等待消息任务的等待列表（waiting list）。如果消息队列中没有消息，即消息队列是空，等待消息的任务就被挂起并放入等待消息任务列表中，直到有消息到来。通常，内核允许等待消息的任务定义等待超时的时间。如果限定时间内任务没有收到消息，该任务就进入就绪态并开始运行，同时返回出错代码，指出出现等待超时错误。一旦一则消息放入消息队列，该消息将传给等待消息的任务中优先级最高的那个任务，或是最先进入等待消息任务列表的任务。图 2.18 示意中断服务子程序如何将消息放入消息队列。图中两个人写的 I 表示消息队列，“10”表示消息队列最多可以放 10 条消息，沙漏旁边的 0 表示任务没有定义超时，将永远等下去，直至消息的到来。

典型地，内核提供的消息队列服务如下：

- 消息队列初始化。队列初始化时总是清为空；
- 放一则消息到队列中去（Post）；
- 等待一则消息的到来（Pend）；
- 如果队列中有消息则任务可以得到消息，但如果此时队列为空，内核并不将该任务挂起（Accept）。如果有消息，则消息从队列中取走。没有消息则用特别的返回代码通知调用者，队列中没有消息。



图2.18 消息队列

2.25 中断

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

- 在前后台系统中，程序回到后台程序；
- 对非占先式内核而言，程序回到被中断了的任务；
- 对占先式内核而言，让进入就绪态的优先级最高的任务开始运行。

中断使得 CPU 可以在事件发生时才予以处理，而不必让微处理器连续不断地查询 (polling) 是否有事件发生。通过两条特殊指令：关中断 (disable interrupt) 和开中断 (enable interrupt) 可以让微处理器不响应或响应中断。在实时环境中，关中断的时间应尽量的短。关中断影响中断延迟时间（见 2.26）。关中断时间太长可能会引起中断丢失。微处理器一般允许中断嵌套，也就是说在中断服务期间，微处理器可以识别另一个更重要的中断，并服务于那个更重要的中断，如图 2.19 所示。

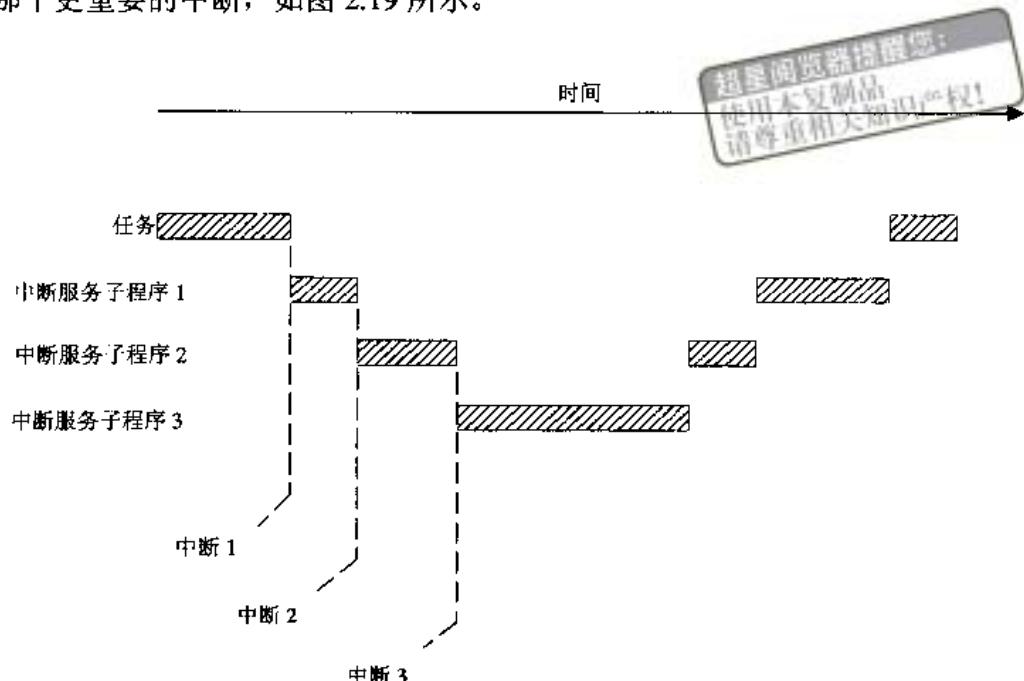


图2.19 中断嵌套

2.26 中断延迟

可能实时内核最重要的指标就是中断关了多长时间。所有实时系统在进入临界区代码段之前都要关中断，执行完临界代码之后再开中断。关中断的时间越长，中断延迟就越长。中断延迟由表达式[2.2]给出。

$$\begin{aligned} \text{中断延迟} = & \text{ 关中断的最长时间} \\ & + \text{ 开始执行中断服务子程序的第一条指令的时间} \end{aligned} \quad [2.2]$$

2.27 中断响应

中断响应定义为从中断发生到开始执行用户的中断服务子程序代码来处理这个中断的

时间。中断响应时间包括开始处理这个中断前的全部开销。典型地，执行用户代码之前要保护现场，将 CPU 的各寄存器推入堆栈。这段时间将被记作中断响应时间。

对前后台系统，保存寄存器以后立即执行用户代码，中断响应时间由表达[2.3]给出。

$$\begin{aligned} \text{中断响应时间} = & \text{ 中断延迟} \\ & + \text{ 保存 CPU 内部寄存器的时间} \end{aligned} \quad [2.3]$$

对于非占先式内核，微处理器保存内部寄存器以后，用户的中断服务子程序代码全立即得到执行。非占先式内核的中断响应时间由表达式[2.4]给出。

$$\begin{aligned} \text{中断响应时间} = & \text{ 中断延迟} \\ & + \text{ 保存 CPU 内部寄存器的时间} \end{aligned} \quad [2.4]$$

对于占先式内核，则要先调用一个特定的函数，该函数通知内核即将进行中断服务，使得内核可以跟踪中断的嵌套。对于 μC/OS-II 说来，这个函数是 OSIntEnter()，占先式内核的中断响应时间由表达式[2.5]给出。

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

中断响应时间是系统在最坏情况下的响应中断的时间，某系统 100 次中有 99 次在 50μs 之内响应中断，只有一次响应中断的时间是 250μs，只能认为中断响应时间是 250μs。

2.28 中断恢复时间

中断恢复时间（interrupt recovery）的定义是微处理器返回到被中断了的程序代码所需要的时间。在前后台系统中，中断恢复时间很简单，只包括恢复 CPU 内部寄存器值的时间和执行中断返回指令的时间。中断恢复时间由表达式[2.6]给出。

$$\begin{aligned} \text{中断恢复时间} = & \text{ 恢复 CPU 内部寄存器值的时间} \\ & + \text{ 执行中断返回指令的时间} \end{aligned} \quad [2.6]$$

和前后台系统一样，非占先式内核的中断恢复时间也很简单，只包括恢复 CPU 内部寄存器值的时间和执行中断返回指令的时间，如表达式[2.7]所示。

$$\begin{aligned} \text{中断恢复时间} = & \text{ 恢复 CPU 内部寄存器值的时间} \\ & + \text{ 执行中断返回指令的时间} \end{aligned} \quad [2.7]$$

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

$$\begin{aligned} \text{中断恢复时间} = & \text{ 判定是否有优先级更高的任务进入了就绪态的时间} \\ & + \text{ 恢复那个优先级更高任务的 CPU 内部寄存器的时间} \\ & + \text{ 执行中断返回指令的时间} \end{aligned} \quad [2.8]$$

2.29 中断延迟、响应和恢复

图2.20到图2.22示意前后台系统、非占先式内核、占先式内核相应的中断延迟、响应和恢复过程。

注意，对于占先式实时内核，中断返回函数将决定是返回到被中断的任务（图2.22A），还是让那个优先级最高任务运行。是中断服务子程序使那个优先级更高的任务进入了就绪态（图2.22B）。在后一种情况下，恢复中断的时间要稍长一些，因为内核要做任务切换。在本书中，我做了一张执行时间表，此表多少可以衡量执行时间的不同，假定μC/OS-II是在33MHz Intel 80186微处理器上运行的。此表可以使读者看到做任务切换的时间开销（见表9.3，在33MHz 80186上μC/OS-II服务的执行时间）。

2.30 中断处理时间

虽然中断服务的处理时间应该尽可能的短，但是对处理时间并没有绝对的限制。不能说中断服务必须全部小于100μs, 500μs或1ms。如果中断服务是在任何给定的时间开始，且中断服务程序代码是应用程序中最重要的代码，则中断服务需要多长时间就应该给它多长时间。然而在大多数情况下，中断服务子程序应识别中断来源，从叫中断的设备取得数据或状态，并通知真正做该事件处理的那个任务。当然应该考虑到是否通知一个任务去做事件处理所花的时间比处理这个事件所花的时间还多。在中断服务中通知一个任务做时间处理（通过信号量、邮箱或消息队列）是需要一定时间的，如果事件处理需花的时间短于给一个任务发通知的时间，就应该考虑在中断服务子程序中做事件处理并在中断服务子程

序中开中断，以允许优先级更高的中断打入并优先得到服务。

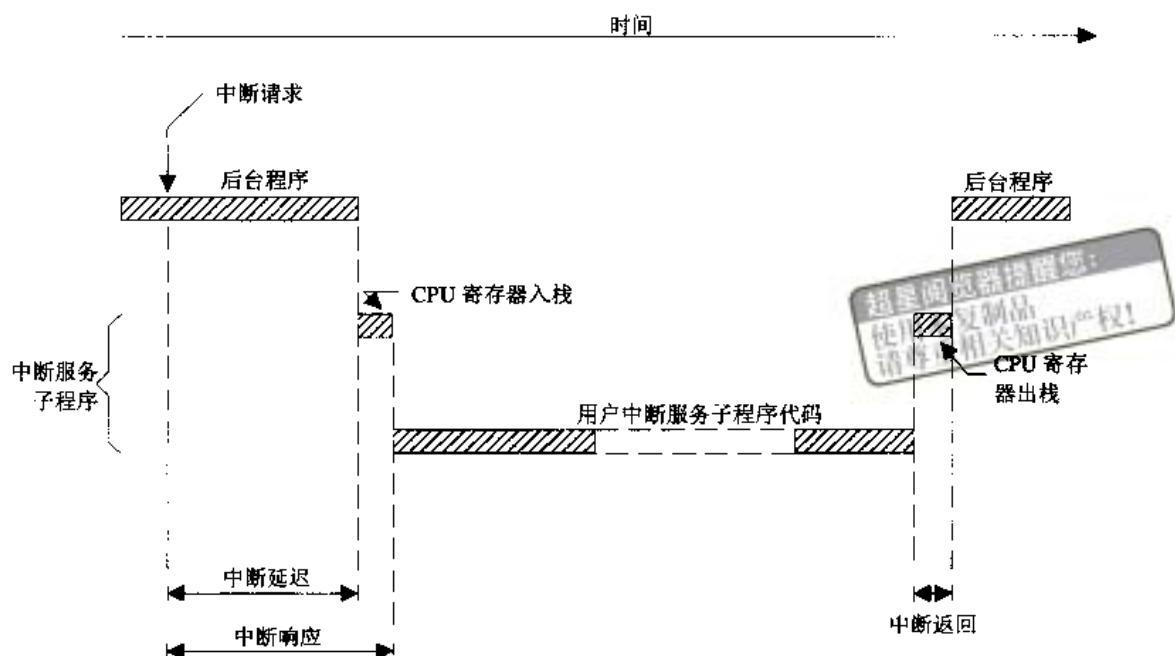


图2.20 中断延迟、响应和恢复（前后台模式）

2.31 非屏蔽中断（NMI）

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

$$\begin{aligned} \text{中断延迟时间} = & \text{ 指令执行时间中最长的那个时间} \\ & + \text{ 开始做非屏蔽中断服务的时间} \end{aligned} \quad [2.9]$$

$$\begin{aligned} \text{中断响应时间} = & \text{ 中断延迟时间} \\ & + \text{ 保存 CPU 寄存器花的时间} \end{aligned} \quad [2.10]$$

$$\begin{aligned} \text{中断恢复时间} = & \text{ 恢复 CPU 寄存器的时间} \\ & + \text{ 执行中断返回指令的时间} \end{aligned} \quad [2.11]$$

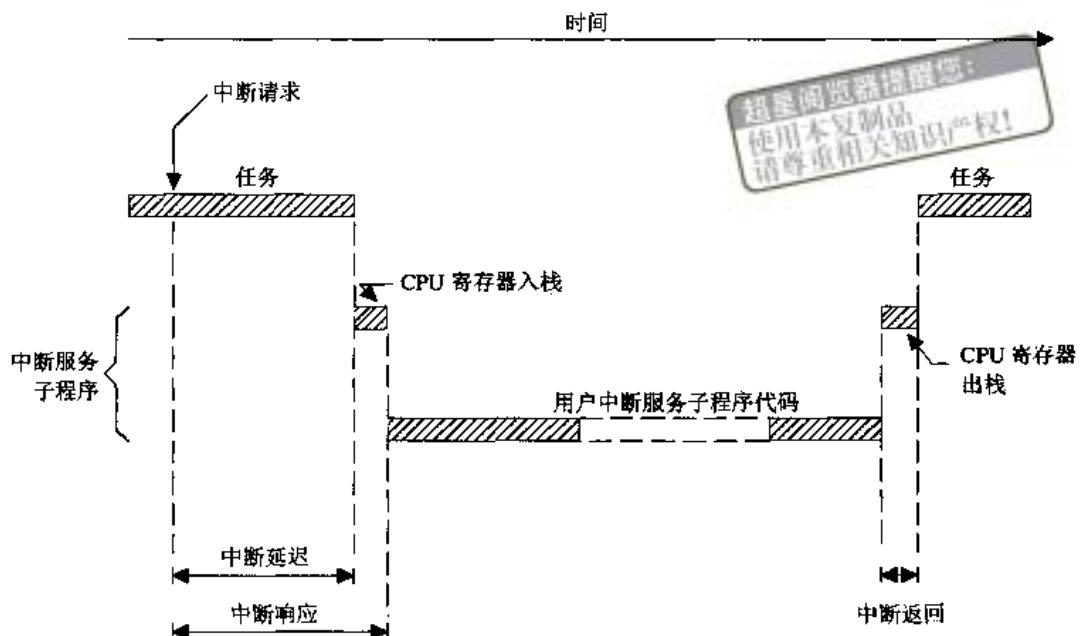


图2.21 中断延迟、响应和恢复（非抢占式内核）

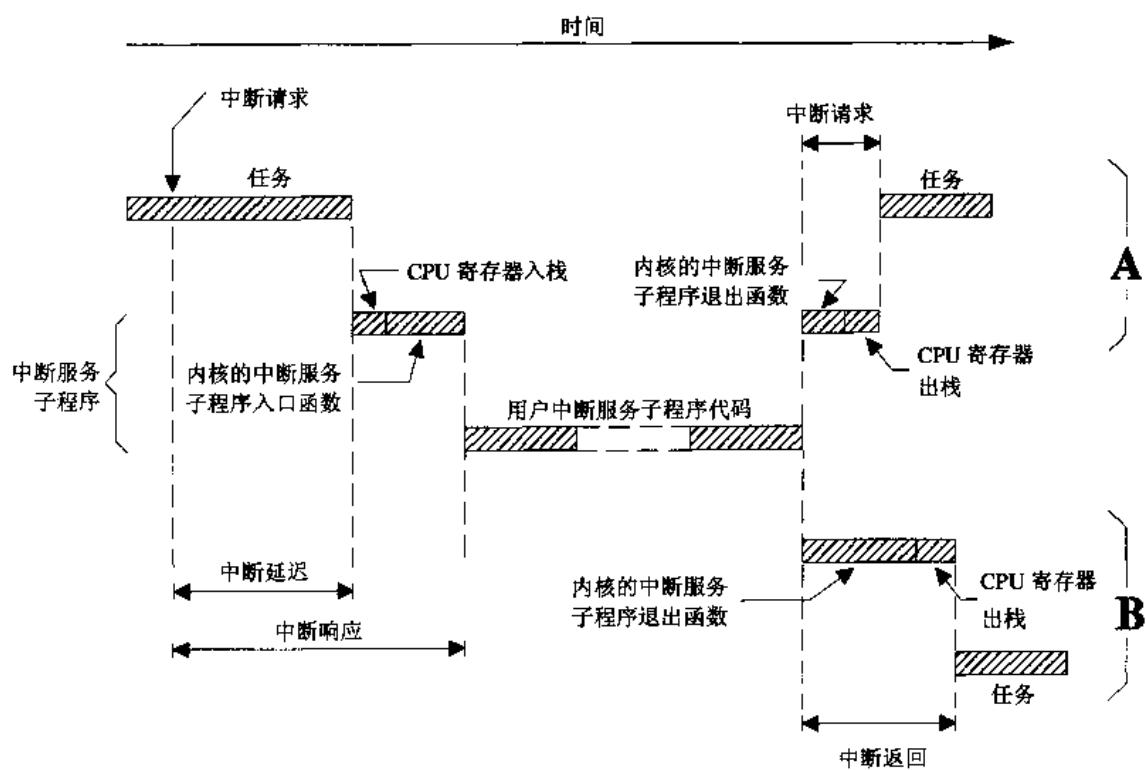


图2.22 中断延迟、响应和恢复（抢占式内核）

在一项应用中，我将非屏蔽中断用于可能每 $150\mu s$ 发生一次的中断。中断处理时间在 $80\sim125\mu s$ 之间。所使用的内核的关中断时间是 $45\mu s$ 。可以看出，如果使用可屏蔽中断的话，中断响应会推迟 $20\mu s$ 。

在非屏蔽中断的中断服务子程序中，不能使用内核提供的服务，因为非屏蔽中断是关不掉的，故不能在非屏蔽中断处理中处理临界区代码。然而向非屏蔽中断传送参数或从非屏蔽中断获取参数还是可以进行的。参数的传递必须使用全程变量，全程变量的位数必须是一次读或写能完成的，即不应该是两个分离的字节，要两次读或写才能完成。

非屏蔽中断可以用增加外部电路的方法禁止掉，如图 2.23 所示。假定中断源和非屏蔽中断都是正逻辑，用一个简单的“与”门插在中断源和微处理器的非屏蔽中断输入端之间。向输出口（Output Port）写 0 就将中断关了。不一定要以这种关中断方式来使用内核服务，但可以用这种方式在中断服务子程序和任务之间传递参数（大的、多字节的，一次读写不能完成的变量）。

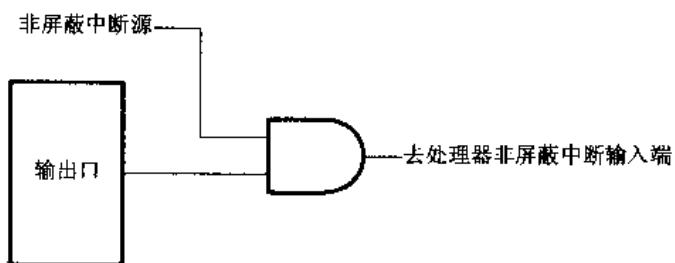


图 2.23 非屏蔽中断的禁止

假定非屏蔽中断服务子程序每 40 次执行中有一次要给任务发信号，如果非屏蔽中断 $150\mu s$ 执行一次，则每 $6ms$ ($40 \times 150\mu s$) 给任务发一次信号。在非屏蔽中断服务子程序中，不能使用内核服务给任务发信号，但可以使用如图 2.24 所示的中断机制。即用非屏蔽中断产生普通可屏蔽中断的机制。在这种情况下，非屏蔽中断通过某一输出口产生硬件中断（置输出口为有效电平）。由于非屏蔽中断服务通常具有最高的优先级，在非屏蔽中断服务过程中不允许中断嵌套，普通中断一直要等到非屏蔽中断服务子程序运行结束后才能被识别。在非屏蔽中断服务子程序完成以后，微处理器开始响应这个硬件中断。在这个中断服务子程序中，要清除中断源（置输出口为无效电平），然后用信号量去唤醒那个需要唤醒的任务。任务本身的运行时间和信号量的有效时间都接近 $6ms$ ，实时性得到了满足。

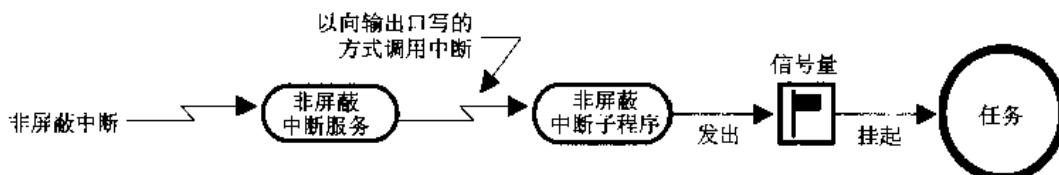


图 2.24 从非屏蔽中断给任务发信号

2.32 时钟节拍

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

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

图 2.25 到 图 2.27 示意任务将自身延迟一个时钟节拍的时序。阴影部分是各部分程序的执行时间。请注意，相应的程序运行时间是长短不一的，这反映了程序中含有循环和条件转移语句（即 if/else, switch, ?: 等语句）的典型情况。时间节拍中断服务子程序的运行时间也是不一样的。尽管在图中画得有所夸大。

第一种情况如图 2.25 所示，优先级高的任务和中断服务超前于要求延时一个时钟节拍的任务运行。可以看出，虽然该任务想要延时 20ms，但由于其优先级的缘故，实际上每次延时多少是变化的，这就引起了任务执行时间的抖动。

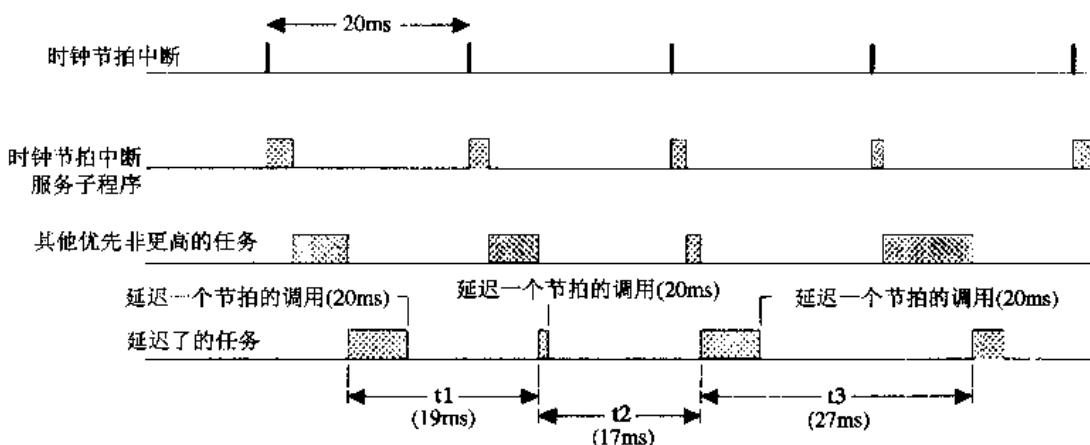


图2.25 将任务延迟一个时钟节拍（第一种情况）

第二种情况，如图 2.26 所示，所有高优先级的任务和中断服务的执行时间略微小于一个时钟节拍。如果任务将自己延时一个时钟节拍的请求刚好发生在下一个时钟节拍之前，这个任务的再次执行几乎是立即开始的。因此，如果要求任务的延迟至少为一个时钟节拍的话，则要多定义一个延时时钟节拍。换句话说，如果想要将一个任务至少延迟 5 个时钟节拍的话，得在程序中延时 6 个时钟节拍。

第三种情况，如图 2.27 所示，所有高优先级的任务加上中断服务的执行时间长于一个时钟节拍。在这种情况下，拟延迟一个时钟节拍的任务实际上在两个时钟节拍后开始

运行，引起了延迟时间超差。这在某些应用中或许是可以的，而在多数情况下是不可接受的。

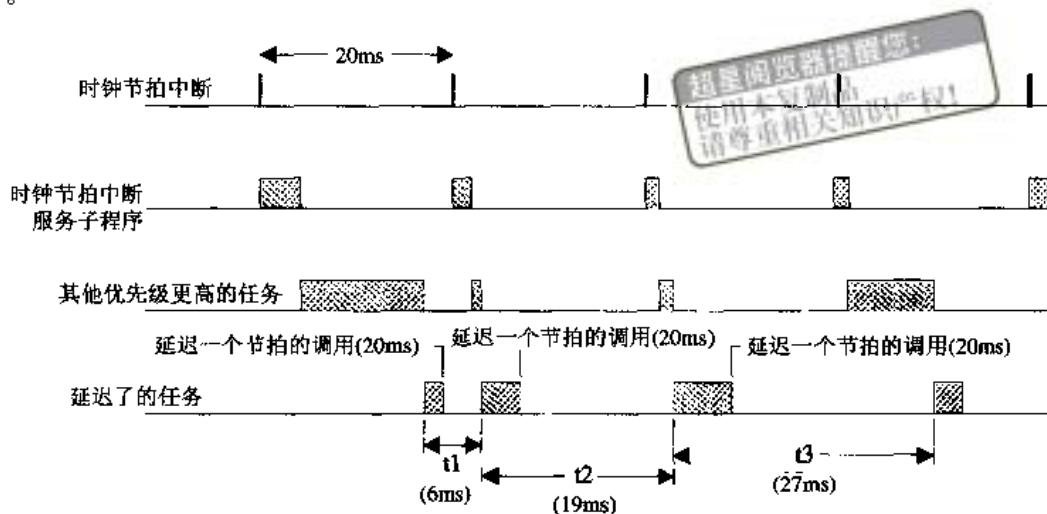


图2.26 将任务延迟一个时钟节拍（第二种情况）

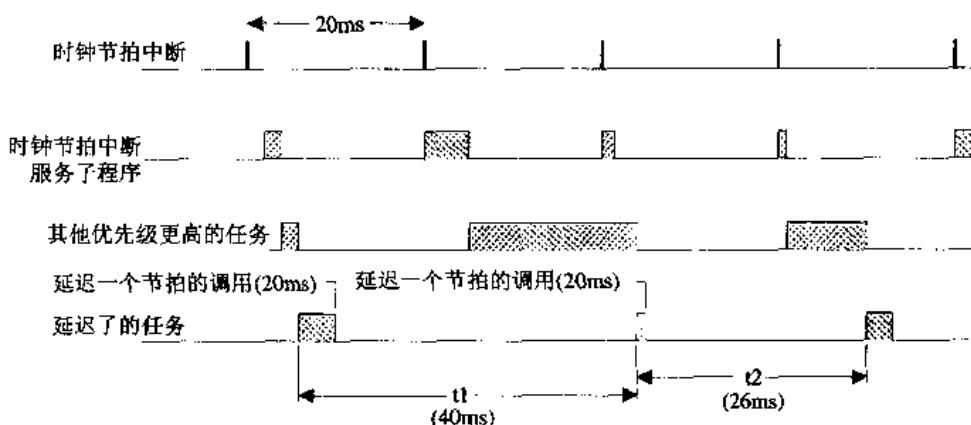


图2.27 将任务延迟一个时钟节拍（第三种情况）

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

- 增加微处理器的时钟频率；
- 增加时钟节拍的频率；
- 重新安排任务的优先级；
- 避免使用浮点运算（如果非使用不可，尽量用单精度数）；
- 使用能较好地优化程序代码的编译器；
- 时间要求苛刻的代码用汇编语言写；

- 如果可能，用同一家族的更快的微处理器做系统升级。如从 8086 向 80186 升级，从 68000 向 68020 升级等。
- 不管怎么样，抖动总是存在的。



2.33 对存储器的需求

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

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

因为每个任务都是独立运行的，必须给每个任务提供单独的栈空间（RAM）。应用程
序设计人员决定分配给每个任务多少栈空间时，应该尽可能使之接近实际需求量（有时，
这是相当困难的一件事）。栈空间的大小不仅仅要计算任务本身的需求（局部变量、函数调
用等等），还需要计算最多中断嵌套层数（保存寄存器、中断服务程序中的局部变量等）。
根据不同的目标微处理器和内核的类型，任务栈和系统栈可以是分开的。系统栈专门用于
处理中断级代码。这样做有许多好处，每个任务需要的栈空间可以大大减少。内核的另
一个应该具有的性能是，每个任务所需的栈空间大小可以分别定义（μC/OS-II 可以做到）。相
反，有些内核要求每个任务所需的栈空间都相同。所有内核都需要额外的栈空间以保证内
部变量、数据结构、队列等。如果内核不支持单独的中断用栈，总的 RAM 需求由表达式[2.13]
给出。

$$\begin{aligned} \text{RAM 总需求} = & \text{ 应用程序的 RAM 需求} \\ & + (\text{任务栈需求} + \text{最大中断嵌套栈需求}) * \text{任务数} \end{aligned} \quad [2.13]$$

如果内核支持中断用栈分离，总 RAM 需求量由表达式[2.14]给出：

$$\begin{aligned} \text{RAM 总需求} = & \text{ 应用程序的 RAM 需求} \\ & + \text{内核数据区的 RAM 需求} \\ & + \text{各任务栈需求之总和} \\ & + \text{最多中断嵌套之栈需求} \end{aligned} \quad [2.14]$$

除非有特别大的 RAM 空间可以所用，对栈空间的分配与使用要非常小心。为减少应

用程序需要的 RAM 空间，对每个任务栈空间的使用都要非常小心，特别要注意以下几点：

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



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

2.34 使用实时内核的优缺点

实时内核也称为实时操作系统或 RTOS。它的使用使得实时应用程序的设计和扩展变得容易，不需要大的改动就可以增加新的功能。通过将应用程序分割成若干独立的任务，RTOS 使得应用程序的设计过程大为简化。使用可剥夺性内核时，所有时间要求苛刻的事件都得到了尽可能快捷、有效的处理。通过有效的服务，如信号量、邮箱、队列、延时、超时等，RTOS 使得资源得到更好的利用。

如果应用项目对额外的需求可以承受，应该考虑使用实时内核。这些额外的需求是：内核的价格，额外的 ROM/RAM 开销，2 到 4 百分点的 CPU 额外负荷。

还没有提到的一个因素是使用实时内核增加的价格成本。在一些应用中，价格就是一切，以至于对使用 RTOS 连想都不敢想。

当今有 80 个以上的 RTOS 商家，生产面向 8 位、16 位、32 位甚至是 64 位的微处理器的 RTOS 产品。一些软件包是完整的操作系统，不仅包括实时内核，还包括输入输出管理、视窗系统（用于显示）、文件系统、网络、语言接口库、调试软件、交叉平台编译（Cross-Platform compiler）。RTOS 的价格从 70 美元到 30000 美元。RTOS 制造商还可能索取每个目标系统的版权使用费。就像从 RTOS 商家那买一个芯片安装到每一个产品上，然后一同出售。RTOS 商家称之为硅片软件（Silicon Software）。每个产品的版权费从 5 美元到 250 美元不等。同如今的其他软件包一样，还得考虑软件维护费，这部分开销为每年还得花 100 到 5000 美元！

2.35 实时系统小结

三种类型的实时系统归纳于表 2.2 中，这三种实时系统是：前后台系统，非占先式内核和占先式内核。

表 2.2

实时系统小结

	前后台系统	非占先式内核	占先式内核
中断延迟时间	MAX (最长指令时间, 用户中断禁用时间) + 指向ISR的向量	MAX (最长指令时间, 用户中断禁用时间, 内核中断禁用时间) + 指向ISR的向量	MAX (最长指令时间, 用户中断禁用时间, 内核中断禁用时间) + 指向ISR的向量
中断响应时间	中断延迟 + 保存 CPU状态时间	中断延迟 + 保存CPU状态时间	中断延迟 + 保存CPU状态时间 + 内核ISR进入时间
中断恢复时间	后台恢复状态时间 + 中断返回时间	任务状态恢复时间 + 中断返回时间	寻找最高优先级任务的时间 + 最高优先级任务的恢复时间 + 中断返回时间
任务响应时间	后台	最长任务时间 + 寻找最高优先级任务的时间 + 任务切换	寻找最高优先级任务的时间 + 任务切换
ROM大小	应用程序代码	应用程序代码 + 内核代码	应用程序代码 + 内核代码
RAM大小	应用程序代码	应用程序代码 + 内核 RAM + SUM (任务堆栈 + MAX (ISR堆栈))	应用程序代码 + 内核 RAM + SUM (任务堆栈 + MAX (ISR堆栈))
可获得的服务	应用程序代码 must 必须提供	有	有

第3章



内核结构

本章给出μC/OS-II的主要结构概貌。读者将学习以下一些内容：

- μC/OS-II是怎样处理临界区代码的；
- 什么是任务，怎样把用户的任务交给μC/OS-II；
- 任务是怎样调度的；
- 应用程序CPU的利用率是多少，μC/OS-II是怎样知道的；
- 怎样写中断服务子程序；
- 什么是时钟节拍，μC/OS-II是怎样处理时钟节拍的；
- μC/OS-II是怎样初始化的；
- 怎样启动多任务。

本章还描述以下函数，这些服务于应用程序：

- OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()
- OSInit()
- OSStart()
- OSIntEnter() 和 OSIntExit()
- OSSchedLock() 和 OSSchedUnlock()
- OSVersion()

3.0 临界区

和其他内核一样，μC/OS-II为了处理临界区（critical section）代码需要关中断，处理完毕后再开中断。这使得μC/OS-II能够避免同时有其他任务或中断服务进入临界区代码。关中断的时间是实时内核开发商应提供的最重要的指标之一，因为这个指标影响用户系统对实时事件的响应速度。μC/OS-II努力使关中断时间降至最短，但就使用μC/OS-II而言，关中断的时间很大程度上取决于微处理器的架构以及编译器所生成的代码质量。

微处理器一般都有关中断/开中断指令，用户使用的C语言编译器必须有某种机制能够在C中直接实现关中断/开中断地操作。某些C编译器允许在用户的C源代码中插入汇编

语言的语句，这使插入微处理器指令来关中断/开中断很容易实现。而有的编译器把从 C 语言中关中断/开中断放在语言的扩展部分。 μ C/OS-II 定义两个宏（macro）来关中断和开中断，以便避开不同 C 编译器厂商选择不同的方法来处理关中断和开中断。 μ C/OS-II 中的这两个宏调用分别是：OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()。因为这两个宏的定义取决于所用的微处理器，故在文件 OS_CPU.H 中可以找到相应宏定义。每种微处理器

```
void YourTask(void *pdata)
{
    for(;;) {
        /* 用户代码 */
        /* 调用  $\mu$ C/OS-II 的某种系统服务 */
        OSTaskSwitch();
        OSTimeDly();
        OSTimeDly();
        OSTaskDel(OS_PRIO_BLOCK);
        OSTaskSuspend(OS_PRIO_BLOCK);
        OSTimeDly();
        OSTaskDelete();
        /* 用户代码 */
    }
}
```

序清单 3.1(2)], 如程序清单 3.1 所示。一个任务看类型，有形式参数变量，但是任务是绝不会返回的。1(1)]。

程序清单 3.1 任务是一个无限循环

不同的是，当任务完成以后，任务可以自我删除，如程序清单 3.2 所示。注意任务代码并非真的删除了， μ C/OS-II 只是简单地不再理会这个任务了，这个任务的代码也不会再运行，如果任务调用了 OSTaskDel(), 这个任务绝不会返回什么。



形式参数变量[程序清单 3.1(1)]是由用户代码在第一次执行的时候带入的。请注意，该变量的类型是一个指向 void 的指针。这是为了允许用户应用程序传递任何类型的数据给任务。这个指针好比一辆万能的车子，如果需要的话，可以运载一个变量的地址，或一个结构，甚至是一个函数的地址。也可以建立许多相同任务，所有任务都使用同一个函数（或者说同一个任务代码程序），参见第 1 章的例 1。例如，用户可以将四个串行口安排成每个串行口都是一个单独的任务，而每个任务的代码实际上是相同的。并不需要将代码复制四次，用户可以建立一个任务，向这个任务传入一个指向某数据结构的指针变量，这个数据结构定义串行口的参数（波特率、I/O 口地址、中断向量号等）。

μ C/OS-II 可以管理多达 64 个任务，但目前版本的 μ C/OS-II 有两个任务已经被系统占用了。作者保留了优先级为 0、1、2、3、OS_LOWEST_PRIO-3、OS_LOWEST_PRIO-2、OS_LOWEST_PRIO-1 以及 OS_LOWEST_PRIO 这 8 个任务以被将来使用。OS_LOWEST_PRIO 是作为常数在 OS_CFG.H 文件中用#define constant 定义的。因此用户可以有高达 56 个应用任务。必须给每个任务赋以不同的优先级，优先级可以从 0 到 OS_LOWEST_PRIO-2，优先级号越低，任务的优先级越高。 μ C/OS-II 总是运行进入就绪态的优先级最高的任务。目前版本的 μ C/OS-II 中，任务的优先级号就是任务编号 (ID)。优先级号(或任务的 ID 号)也被一些内核服务函数调用，如改变优先级函数 OSTaskChangePrio()，以及任务删除函数 OSTaskDel()。

为了使 μ C/OS-II 能管理用户任务，用户必须在建立一个任务的时候，将任务的起始地址与其他参数一起传给下面两个函数中的一个：OSTaskCreate 或 OSTaskCreateExt()。OSTaskCreateExt()是 OSTaskCreate()的扩展，扩展了一些附加的功能。这两个函数的解释见第 4 章。

3.2 任务状态

图 3.1 是 μ C/OS-II 控制下的任务状态转换图。在任一时刻，任务的状态一定是这五种状态之一。

休眠态 (DORMANT) 指任务驻留在程序空间之中，还没有交给 μ C/OS-II 管理（见程序清单 3.1 或程序清单 3.2）。把任务交给 μ C/OS-II 是通过调用下述两个函数：OSTaskCreate()

或 OSTaskCreateExt()。当任务一旦建立，这个任务就进入就绪态准备运行。任务的建立可以在多任务运行开始之前，也可以动态地被一个运行着的任务建立。如果一个任务是被另一个任务建立的，而这个任务的优先级高于建立它的那个任务，则这个刚刚建立的任务将立即得到 CPU 的控制权。一个任务可以通过调用 OSTaskDel()返回到休眠态，或通过调用该函数让另一个任务进入休眠态。

调用 OSStart()可以启动多任务。OSStart()函数运行进入就绪态的优先级最高的任务。只有当所有优先级高于它的任务转为等待状态，或者是被删除了，就绪的任务才能进入运行态。

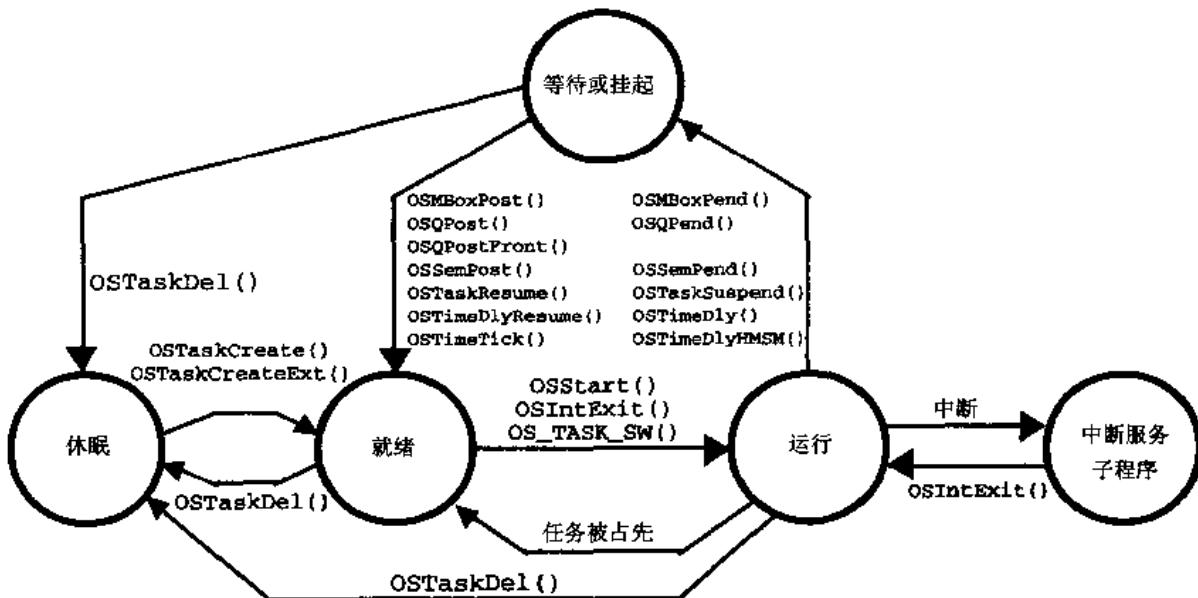


图3.1 任务的状态

正在运行的任务可以通过调用两个函数将自身延迟一段时间，这两个函数是 OSTimeDly() 或 OSTimeDlyHMSM()。这个任务于是进入等待状态，等待这段时间过去，下一个优先级最高的、并进入了就绪态的任务立刻被赋予了 CPU 的控制权。等待的时间过去以后，系统服务函数 OSTimeTick()使延迟了的任务进入就绪态（见 3.10 节）。

正在运行的任务期待某一事件的发生时也要等待，手段是调用以下 3 个函数之一：OSSemPend()，OSMboxPend()，或 OSQPend()。调用后任务进入了等待状态（WAITING）。当任务因等待事件被挂起（Pend），下一个优先级最高的任务立即得到了 CPU 的控制权。当事件发生了，被挂起的任务进入就绪态。事件发生的报告可能来自另一个任务，也可能来自中断服务子程序。

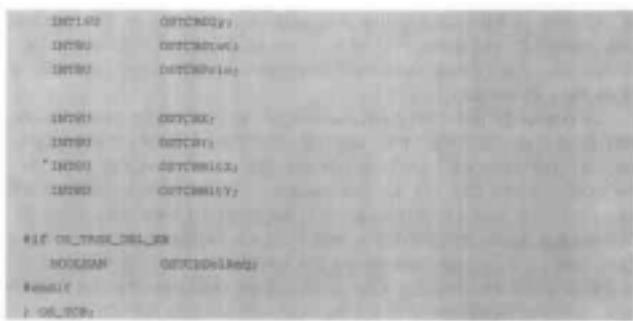
正在运行的任务是可以被中断的，除非该任务关中断，或者μC/OS-II关中断。被中断了的任务就进入了中断服务子程序（ISR）。响应中断时，正在执行的任务被挂起，中断服务子程序控制了 CPU 的使用权。中断服务子程序可能会报告一个或多个事件的发生，而使一个或多个任务进入就绪态。在这种情况下，从中断服务子程序返回之前，μC/OS-II要判

定，被中断的任务是否还是就绪态任务中优先级最高的。如果中断服务子程序使一个优先级更高的任务进入了就绪态，则新进入就绪态的这个优先级更高的任务将得以运行，否则原来被中断了的任务还会继续运行。

当所有的任务都处于等待事件发生或等待延迟时间结束的状态时，μC/OS-II 执行空闲任务（idle task），执行 OSTaskIdle() 函数。



control blocks) OS_TCB 将被赋值（程序清单 3.3）。PU 使用权被剥夺时，μC/OS-II 用它来保存该任务。任务控制块能确保任务从被中断的那一点丝毫不 M 中。读者将会注意到笔者在组织这个数据结构立的时候，OS_TCB 就被初始化了（见第 4 章）。



.OSTCBStkPtr 是指向当前任务栈顶的指针。**μC/OS-II** 允许每个任务有自己的栈，尤为重要的是，每个任务的栈的容量可以是任意的。有些商业内核要求所有任务栈的容量都一样，除非用户写一个复杂的接口函数来改变它。这种限制浪费了 RAM，当各任务需要的栈空间不同时，也必须按任务中预计的最大栈空间来分配栈空间。**OSTCBStkPtr** 是 **OS_TCB** 数据结构中惟一的一个能用汇编语言来处置的变量（在任务切换段的代码 **Context-switching code** 之中）。把 **OSTCBStkPtr** 放在数据结构的最前面，使得从汇编语言中处理这个变量时较为容易。

.OSTCBExtPtr 指向用户定义的任务控制块扩展。用户可以扩展任务控制块而不必修改 **μC/OS-II** 的源代码。**.OSTCBExtPtr** 只在函数 **OstaskCreateExt()** 中使用，故使用时要将 **OS_TASK_CREATE_EN** 设为 1，以允许建立任务函数的扩展。例如用户可以建立一个数据结构，这个数据结构包含每个任务的名字，或跟踪某个任务的执行时间，或者跟踪切换到某个任务的次数（见第 1 章的例 3）。注意，笔者将这个扩展指针变量放在紧跟着堆栈指针的位置，为的是当用户需要在汇编语言中处理这个变量时，从数据结构的起始地址算偏移量比较方便。

.OSTCBStkBottom 是指向任务栈底的指针。如果微处理器的栈指针是递减的，即栈存储器从高地址向低地址方向分配，则 **OSTCBStkBottom** 指向任务使用的栈空间的最低地址。类似地，如果微处理器的栈是从低地址向高地址递增型的，则 **OSTCBStkBottom** 指向任务可以使用的栈空间的最高地址。函数 **OSTaskStkChk()** 要用到变量 **OSTCBStkBottom**，在运行中检验栈空间的使用情况。用户可以用它来确定任务实际需要的栈空间。这个功能只有当用户在任务建立时允许使用 **OSTaskCreateExt()** 函数时才能实现。这就要求用户将 **OS_TASK_CREATE_EXT_EN** 设为 1，以便允许该功能。

.OSTCBStkSize 存有栈中可容纳的指针元数目而不是用字节（Byte）表示的栈容量总

数。也就是说，如果栈中可以保存 1000 个入口地址，每个地址宽度是 32 位的，则实际栈容量是 4000 字节。同样是 1000 个入口地址，如果每个地址宽度是 16 位的，则总栈容量只有 2000 字节。在函数 OSStakChk() 中要调用 OSTCBStkSize。同理，若使用该函数的话，要将 OS_TASK_CREATE_EXT_EN 设为 1。

.OSTCBOpt 把“选择项”传给 OSTaskCreateExt()，只有在用户将 OS_TASK_CREATE_EXT_EN 设为 1 时，这个变量才有效。**μC/OS-II** 目前只支持 3 个选择项（见 uCOS_II.H）：OS_TASK_OPT_STK_CHK，OS_TASK_OPT_STK_CLR 和 OS_TASK_OPT_SAVE_FP。OS_TASK_OPT_STK_CHK 用于告知 TaskCreateExt()，在任务建立的时候任务栈检验功能得到了允许。OS_TASK_OPT_STK_CLR 表示任务建立的时候任务栈要清零。只有在用户需要有栈检验功能时，才需要将栈清零。如果不定义 OS_TASK_OPT_STK_CLR，而后又建立、删除了任务，栈检验功能报告的栈使用情况将是错误的。如果任务一旦建立就决不会被删除，而用户初始化时，已将 RAM 清过零，则 OS_TASK_OPT_STK_CLR 不需要再定义，这可以节约程序执行时间。传递了 OS_TASK_OPT_STK_CLR 将增加 TaskCreateExt() 函数的执行时间，因为要将栈空间清零。栈容量越大，清零花的时间越长。最后一个选择项 OS_TASK_OPT_SAVE_FP 通知 TaskCreateExt()，任务要做浮点运算。如果微处理器有硬件的浮点协处理器，则所建立的任务在做任务调度切换时，浮点寄存器的内容要保存。

.OSTCBId 用于存储任务的识别码。这个变量现在没有使用，留给将来扩展用。

.OSTCBNext 和 **.OSTCBPrev** 用于任务控制块 OS_TCBs 的双重链接，该链表在时钟节拍函数 OSTimeTick() 中使用，用于刷新各个任务的任务延迟变量 OSTCBDly，每个任务的任务控制块 OS_TCB 在任务建立的时候被链接到链表中，在任务删除的时候从链表中被删除。双重连接的链表使得任一成员都能被快速插入或删除。

.OSTCBEEventPtr 是指向事件控制块的指针，后面的章节中会有所描述（见第 6 章）。

.OSTCBMsg 是指向传给任务的消息的指针。用法将在后面的章节中提到（见第 6 章）。

.OSTCBDly 当需要把任务延时若干时钟节拍时要用到这个变量，或者需要把任务挂起一段时间以等待某事件的发生，这种等待是有超时限制的。在这种情况下，这个变量保存的是任务在等待事件发生过程中允许挂起的最多时钟节拍数。如果这个变量为 0，表示任务不延时，或者表示等待事件发生的时间没有限制。

.OSTCBStat 是任务的状态字。当 .OSTCBStat 为 0，任务进入就绪态。可以给 .OSTCBStat 赋其他的值，在文件 uCOS_II.H 中有关于这个值的描述。

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

.OSTCBX, **.OSTCBY**, **.OSTCBBitX** 和 **.OSTCBBitY** 用于加速任务进入就绪态的过程或进入等待事件发生状态的过程（值得注意的是，这些值的计算应避免在运行时进行）。这些值是在任务建立时，或者是在改变任务优先级时算好的。这些值的算法见程序清单 3.4。

```

OSTCB[<-->] = priority == 3;
OSTCB[<-->] = OSMaxPriority(priority >= 3);
OSTCB[<-->] = priority & 0x0f;
OSTCB[<-->] = OSMinPriority(priority & 0x0f);

```

江万江河外 OS_TCB 与几个成员的算法



.OSTCBDelReq 是一个布尔量，用于表示该任务是否需要删除，用法将在后面的章节中描述（见第 4 章）。

应用程序中可以有的最多任务数（OS_MAX_TASKS）是在文件 OS_CFG.H 中定义的。这个数也是μC/OS-II 分配给用户程序的任务控制块 OS_TCBs 的最大数目。将 OS_MAX_TASKS 的数目设置为用户应用程序实际需要的任务数可以减小 RAM 的需求量。所有的任务控制块 OS_TCBs 都是放在任务控制块列表数组 OSTCBtbl[] 中的。请注意，μC/OS-II 分配给系统任务 OS_N_SYS_TASKS 若干个任务控制块，见文件 μC/OS-II.H，供其内部使用。目前，一个用于空闲任务，另一个用于任务统计（如果 OS_TASK_STAT_EN 是设为 1 的）。在 μC/OS-II 初始化的时候，如图 3.2 所示，所有任务控制块 OS_TCBs 被链接成空任务控制块的单向链表。当任务一旦建立，空任务控制块指针 OSTCBFreeList 指向的任务控制块便赋给了该任务，然后 OSTCBFreeList 的值调整为指向下一个空的任务控制块。一旦任务被删除，它的任务控制块就会回到空任务控制块链表中。

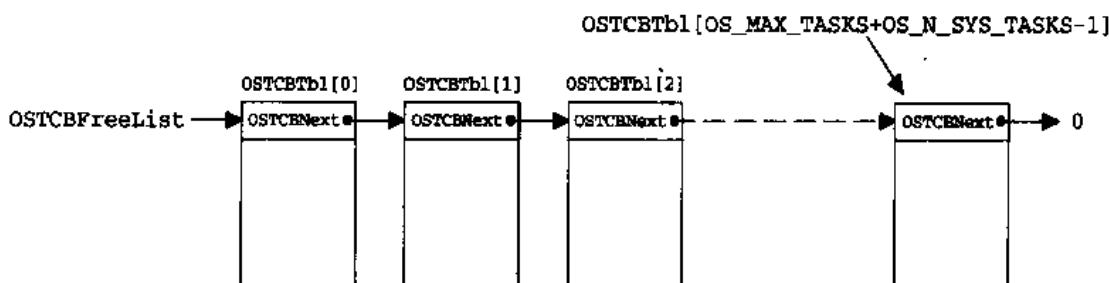


图3.2 空任务列表

3.4 就绪表

每个任务被赋予不同的优先级等级，从 0 级到最低优先级 OS_LOWEST_PRIO，包括 0 和 OS_LOWEST_PRIO 在内（见文件 OS_CFG.H）。当 μC/OS-II 初始化的时候，最低优先级 OS_LOWEST_PRIO 总是被赋给空闲任务 idle task。注意，最多任务数目 OS_MAX_TASKS 和最低优先级数是没有关系的。用户应用程序可以只有 10 个任务，而仍然可以有 32 个优先级的级别（如果用户将最低优先级数设为 31 的话）。

每个任务的就绪态标志都放入就绪表（ready list）中，就绪表中有两个变量 OSRdyGrp 和 OSRdyTbl[]。在 OSRdyGrp 中，任务按优先级分组，8 个任务为一组。OSRdyGrp 中的每一位表示 8 组任务中每一组中是否有进入就绪态的任务。任务进入就绪态时，就绪表 OSRdyTbl[] 中的相应元素的相应位也置位。就绪表 OSRdyTbl[] 数组的大小取决于 OS_LOWEST_PRIO（见文件 OS_CFG.H）。当用户的应用程序中任务数目比较少时，减少 OS_LOWEST_PRIO 的值可以降低 μC/OS-II 对 RAM（数据空间）的需求量。

为确定下次该哪个优先级的任务运行了，内核调度器总是将 OS_LOWEST_PRIO 在就绪表中相应空节的相应位置 1。OSRdyGrp 和 OSRdyTbl[] 之间的关系见图 3.3，是按以下规

OSRdyGrp = (OSMapTbl[prio >> 3])
OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 7] << 1

当 OSRdyTbl[0]中的任何一位是 1 时，OSRdyGrp 的第 0 位置 1。

下标	位掩码（二进制）	
0	00000001	↓， OSRdyGrp 的第 1 位置 1。
1	00000010	↓， OSRdyGrp 的第 2 位置 1。
2	00000000	↓， OSRdyGrp 的第 3 位置 1。
3	00000000	↓， OSRdyGrp 的第 4 位置 1。
4	00000000	↓， OSRdyGrp 的第 5 位置 1。
5	00000000	
6	00000000	
7	10000000	

当 OSRdyTbl[6]中的任何一位是 1 时，OSRdyGrp 的第 6 位置 1。

当 OSRdyTbl[7]中的任何一位是 1 时，OSRdyGrp 的第 7 位置 1。

程序清单 3.5 中的代码用于将任务放入就绪表。prio 是任务的优先级。

程序清单 3.5 使任务进入就绪态

表 3.1

OSMapTbl[]的值

读者可以看出，任务优先级的低三位用于确定任务在总就绪表 OSRdyTbl[]中的所在位。

接下去的三位用于确定是在 OSRdyTbl[]数组的第几个元素。OSMapTbl[]是在 ROM 中的（见文件 OS_CORE.C）位掩码，用于限制 OSRdyTbl[]数组的元素下标在 0 到 7 之间，见表 3.1。

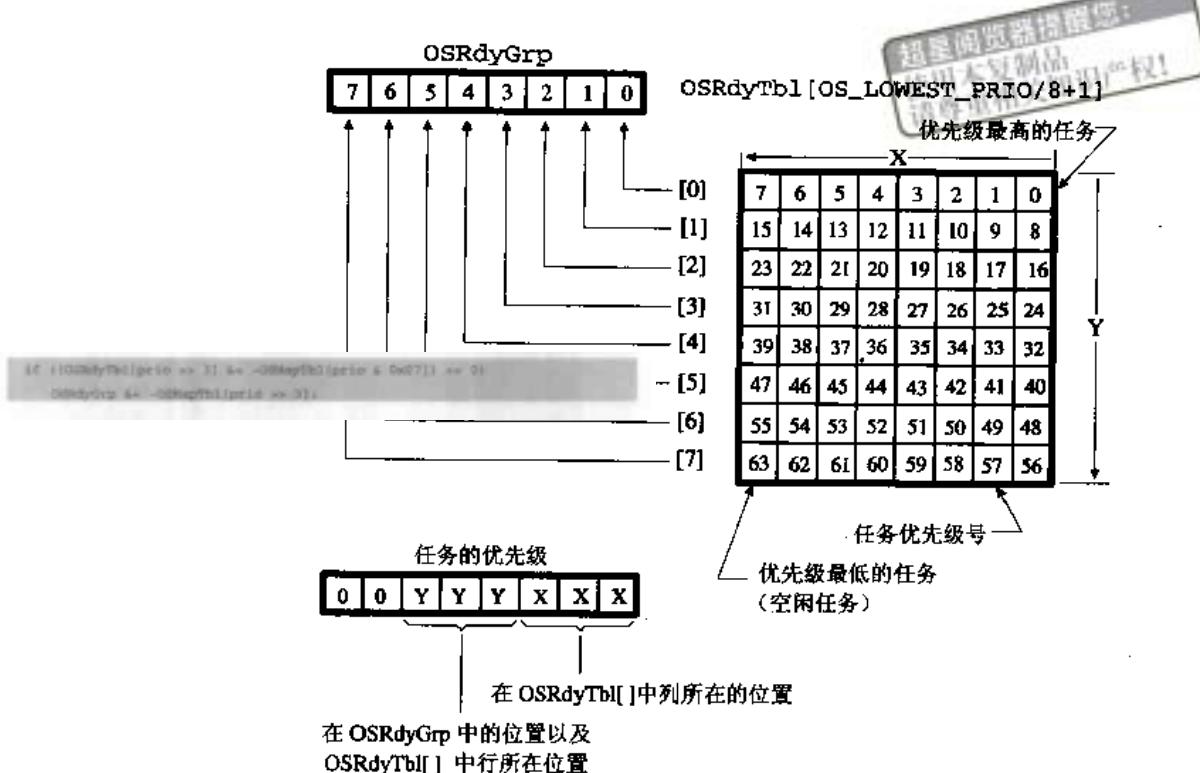


图3.3 μ C/OS-II 就绪表

如果一个任务被删除了，则用程序清单 3.6 中的代码做求反处理。

程序清单 3.6 从就绪表中删除一个任务

以上代码将就绪任务表数组 OSRdyTbl[]中相应元素的相应位清零，而对于 OSRdyGrp，只有当被删除任务所在任务组中全组任务一个都没有进入就绪态时，才将相应位清零。也就是说 OSRdyTbl[prio>>3]所有的位都是零时，OSRdyGrp 的相应位才清零。为了找到那个进入就绪态的优先级最高的任务，并不需要从 OSRdyTbl[0]开始扫描整个就绪任务表，只需要查另外一张表，即优先级判定表 OSUnMapTbl([256])（见文件 OS_CORE.C）。OSRdyTbl[]中每个字节的 8 位代表这一组的 8 个任务哪些进入就绪态了，低位的优先级高于高位。利用这个字节为下标来查 OSUnMapTbl 这张表，返回的字节就是该组任务中就绪态任务中优先级最高的那个任务所在的位置。这个返回值在 0 到 7 之间。确定进入就绪态的优先级最高的任务是用以下代码完成的，如程序清单 3.7 所示。

```

y = OSUnMapTbl[OSRdyGrp];
x = OSUnMapTbl[OSRdyTbl[y]];
prior = ty << 31 * x;

```

程序清单 3.7 找出进入就绪态的优先级最高的任务



例如，如果 OSRdyGrp 的值为二进制 01101000，查 OSUnMapTbl[OSRdyGrp]得到的值是 3，它相应于 OSRdyGrp 中的第 3 位 bit3，这里假设最右边的一位是第 0 位 bit0。类似地，

则 OSUnMapTbl[OSRdyTbl[3]]的值是 2，即第 2 位 bit2 (即 $2^3 \times 8 + 2$)。利用这个优先级的值。查任务控制块优先级的值，再与 OSUnMapTbl[OSRdyTbl[prior]]的值进行比较，如果大于，则将 OSUnMapTbl[OSRdyTbl[prior]]的值赋给 OSRdyGrp，则优先级最高的那个任务就找到了。

```

void OSSelect(void)
{
    OS_ENTER_CRITICAL();
    if (OSIntNesting > OSIntNesting) ++ OSIntNesting; // 1
    y = OSUnMapTbl[OSRdyGrp]; // 2
    OSRdyGrp |= (INTVAL(ty) << 31 * x); // 3
    if (OSRdyGrp >= OSRdyGrp) // 4
        OSRdyGrp = OSRdyGrp | OSRdyGrp; // 5
    OSUnMapTbl[OSRdyTbl[y]] = prior; // 6
    OSIntNesting++; // 7
    OS_TASK_SW(); // 8
}

OS_EXIT_CRITICAL();

```

优先级最高的那一个。确定哪个任务优先级最高，是由调度器 (scheduler) 完成的。任务级的调度是由函数 OSSched()完成的。中断级的调度是由另一个函数 OSIntExt()完成的，这个函数将在以后描述。OSSched()的代码如程序清单 3.8 所示。

程序清单 3.8 任务调度器

μ C/OS-II任务调度所花的时间是常数，与应用程序中建立的任务数无关。如程序清单中[程序清单 3.8(1)]条件语句的条件不满足，任务调度函数 OSSched()将退出，不做任务调度。这个条件是：如果在中断服务子程序中调用 OSSched()，此时中断嵌套层数 OSIntNesting>0，或者由于用户至少调用了一次给任务调度上锁函数 OSSchedLock()，使 OSLockNesting>0。如果不是在中断服务子程序调用 OSSched()，并且任务调度是允许的，即没有上锁，则任务调度函数将找出那个进入就绪态且优先级最高的任务[程序清单 3.8(2)]，进入就绪态的任务在就绪任务表中有相应的位置位。一旦找到那个优先级最高的任务，OSSched()检验这个优先级最高的任务是不是当前正在运行的任务，以此来避免不必要的任务调度[程序清单 3.8(3)]。注意，在 μ C/OS 中曾经是先得到 OSTCBHighRdy 然后和 OSTCBCur 做比较。因为这个比较是两个指针型变量的比较，在 8 位和一些 16 位微处理器中这种比较相对较慢。而在 μ C/OS-II 中是两个整数的比较。并且，除非用户实际需要做任务切换，在查任务控制块优先级表 OSTCBPrioTbl[] 时，不需要用指针变量来查 OSTCBHighRdy。综合这两项改进，即用整数比较代替指针的比较和当需要任务切换时再查表，使得 μ C/OS-II 比 μ C/OS 在 8 位和一些 16 位微处理器上要更快一些。

为实现任务切换，OSTCBHighRdy 必须指向优先级最高的那个任务控制块 OS_TCB，这是通过将以 OSPrioHighRdy 为下标的 OSTCBPrioTbl[] 数组中的那个元素赋给 OSTCBHighRdy 来实现的[程序清单 3.8(4)]。接着，统计计数器 OSCtxSwCtr 加 1，以跟踪任务切换次数[程序清单 3.8(5)]。最后宏调用 OS_TASK_SW()来完成实际上的任务切换[程序清单 3.8(6)]。

任务切换很简单，由以下两步完成，将被挂起任务的微处理器寄存器推入堆栈，然后将较高优先级的任务的寄存器值从栈中恢复到寄存器中。在 μ C/OS-II 中，就绪任务的栈结构总是看起来跟刚刚发生过中断一样，所有微处理器的寄存器都保存在栈中。换句话说， μ C/OS-II 运行就绪态的任务所要做的一切，只是恢复所有的 CPU 寄存器并运行中断返回指令。为了做任务切换，运行 OS_TASK_SW()，人为模仿了一次中断。多数微处理器提供有软中断指令或者陷阱指令 TRAP 来实现上述操作。中断服务子程序或陷阱处理（trap handler），也称作异常处理（exception handler），必须提供中断向量给汇编语言函数 OSCtxSw()。OSCtxSw()除了需要 OSTCBHighRdy 指向即将被挂起的任务，还需要让当前任务控制块 OSTCBCur 指向即将被挂起的任务，参见第 8 章，有关于 OSCtxSw()的更详尽的解释。

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

3.6 给调度器上锁和开锁

给调度器上锁的函数 OSSchedlock()（程序清单 3.9）用于禁止任务调度，直到任务完成后调用给调度器开锁函数 OSSchedUnlock()为止（程序清单 3.10）。调用 OSSchedlock()的任务保持对 CPU 的控制权，尽管有个优先级更高的任务进入了就绪态。当然，此时中断是可以被识别的，中断服务程序也能执行（假设中断是开着的）。OSSchedlock()和 OSSchedUnlock()必须成对使用。变量 OSLockNesting 跟踪 OSSchedLock()函数被调用的次数，以允许嵌套的函数包含临界区代码。这段代码其他任务不得干预。 μ C/OS-II 允许嵌套零时，调度重新得到允许。函数 OSSchedLock()和 OSSchedUnlock()它们影响 μ C/OS-II 对任务的正常管理。

OSSchedUnlock()调用 OSSched() [程序清单 3.10(2)]。

调度器上锁的期间，可能有什么事件发生了并使一个更高优先级的任务进入就绪态。

应用程序不得使用任何能将现行任务挂起的系统调用。也就是说，用户程序不得调用 OSMboxPend()、OSQPend()、OSSemPend()、OSTaskSuspend(OS_PRIO_SELF)、OSTimeDly()或 OSTimeDlyHMSM()，直到 OSLockNesting 为零为止。由于调度器上了锁，用户就锁住了系统，任何其他任务都不能运行。

当低优先级的任务要发消息给多任务的邮箱、消息队列、信号量时（见第 6 章），用户不希望高优先级的任务在邮箱、队列和信号量没有得到消息之前就取得了 CPU 的控制权，此时，用户可以使用 OSSchedLock()。

程序清单 3.9 给调度器上锁

程序清单 3.10 给调度器开锁

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

```
if (osRunning == TRUE) {
    OS_ENTER_CRITICAL();
    if (DELOCKwaiting > 8) {
        DELOCKwaiting--;
        if ((DELOCKwaiting + osLockWaiting) == 8) {           (1)
            OS_EXIT_CRITICAL();
            osSleep();                                         (2)
        } else {
            OS_EXIT_CRITICAL();
        }
    } else {
        OS_EXIT_CRITICAL();
    }
}
```

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

3.7 空闲任务

μ C/OS-II 总是建立一个空闲任务 (idle task)，这个任务在没有其他任务进入就绪态时投入运行。这个空闲任务[OSTaskIdle()]永远设为最低优先级，即 OS_LOWEST_PRIO。空闲任务 OSTaskIdle()什么也不做，只是在不停地给一个 32 位的名叫 OSIdleCtr 的计数器加 1，统计任务（见 3.8）使用这个计数器以确定现行应用软件实际消耗的 CPU 时间。程序清单 3.11 是空闲任务的代码。在计数器加 1 前后，中断是先关掉再开启的，因为 8 位以及大多数 16 位微处理器的 32 位加 1 需要多条指令，要防止高优先级的任务或中断服务子程序从中打入。空闲任务不能被应用软件删除。

程序清单 3.11 μ C/OS-II 的空闲任务

3.8 统计任务



μC/OS-II 有一个提供运行时间统计的任务。这个任务叫做 OSTaskStat(), 如果用户将系

统启动参数 OS_TASK_STAT_EN (见文件 OS_CFG.H) 设为 1, 这个任务就会建立。一旦次 (见文件 OS_CORE.C), 计算当前的 CPU 利用率。应用程序使用了多少 CPU 时间, 用百分比表示, 这里, 精度是 1 个百分点。

是, 用户必须在初始化时建立一个惟一的任务, 在 CORE.C)。换句话说, 在调用系统启动函数 OSStart() 时, 在这个任务中调用系统统计初始化函数 OSTaskStat()。程序清单 3.12 是统计任务的示意性代码。

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

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

因为用户的应用程序必须先建立一个起始任务[TaskStart()], 当主程序 main() 调用系统

启动函数 OSStart()的时候，μC/OS-II 只有 3 个要管理的任务：TaskStart()、OSTaskIdle()和 OSTaskStat()。请注意，任务 TaskStart()的名称是无所谓，叫什么名字都可以。因为 μC/OS-II 已经将空闲任务的优先级设为最低，即 OS_LOWEST_PRIO，统计任务的优先级设为次低，OS_LOWEST_PRIO-1。启动任务 TaskStart()总是优先级最高的任务。

图 3.4 解释初始化统计任务时的流程。用户必须首先调用的是 μC/OS-II 中的系统初始化函数 OSInit()，该函数初始化 μC/OS-II [图 3.4(2)]。有的处理器（例如 Motorola 的 MC68HC11），不需要“设置”中断向量，中断向量已经在 ROM 中有了。用户必须调用 OSTaskCreat()或者 OSTaskCreatExt()以建立 TaskStart() [图 3.4(3)]。进入多任务的条件准备好了以后，调用系统启动函数 OSStart()。这个函数将使 TaskStart()开始执行，因为 TaskStart()是优先级最高的任务 [图 3.4(4)]。

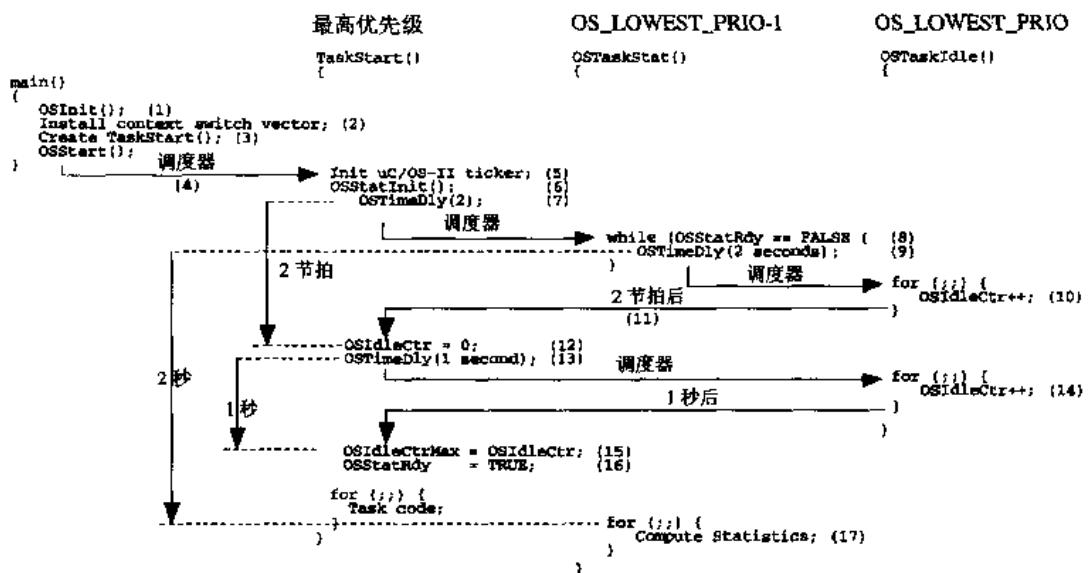


图3.4 统计任务的初始化

TaskStart()负责初始化和启动时钟节拍[图 3.4(5)]。在这里启动时钟节拍是必要的，因为用户不会希望在多任务还没有开始时就接收到时钟节拍中断。接下去 TaskStart()调用统计初始化函数 OSStatInit() [图 3.4(6)]。统计初始化函数 OSStatInit() 将统计在没有其他应用任务运行时，空闲计数器（OSIdleCtr）的计数有多快。奔腾 II 微处理器以 333MHz 运行时，加 1 操作可以使该计数器的值达到每秒 15 000 000 次，这个值离 32 位计数器的溢出极限值 4 294 967 296 还差得远。不过微处理器越来越快，用户要注意这里可能会是将来的一个潜在问题。

系统统计初始化任务函数 OSStatInit() 调用延迟函数 OSTimeDly() 将延时 2 个时钟节拍以停止自身的运行 [图 3.4(7)]。这是为了使 OSStatInit() 与时钟节拍同步。μC/OS-II 然后选下一个优先级最高的进入就绪态的任务运行，这恰好是统计任务 OSTaskStat()。读者会在后面读到 OSTaskStat() 的代码，但粗看一下，OSTaskStat() 所要做的第一件事就是查看统

计任务就绪标志是否为“假”，如果是的话，也要延时两个时钟节拍[图 3.4(8)]。一定会是这样，因为标志 OSStatRdy 已被 OSInit()函数初始化为“假”，所以实际上 OSTaskStat 也将自己推入休眠态(Sleep)两个时钟节拍[图 3.4(9)]。于是任务切换到空闲任务，OSTaskIdle()开始运行，这是惟一一个就绪态任务了。CPU 处在空闲任务 OSTaskIdle 中，直到 TaskStart()一个时钟节拍之后，TaskStart()恢复运行[图 3.4(11)]。

OSIdleCtr 被清零[图 3.4(12)]。然后，OSStatInit()他进入就绪态的任务，OSTaskIdle()又获得了 CPU TaskStart()继续运行，还是在 OSStatInit()中，空闲计数器最大值 OSIdleCtrMax 中[图 3.4(15)]。

StatRdy 设为“真”[图 3.4(16)]，以此来允许两个的利用率。

统计任务的初始化函数 OSStatInit()的代码如程序清单 3.13 所示。

程序清单 3.13 统计任务的初始化

```
void OSStatInit(void)
{
    OSIdleCtr = 0;
    OS_ENTER_CRITICAL();
    OSStatRdy = FALSE;
    OS_EXIT_CRITICAL();
    OSIdleCtrMax = OSIdleCtr;
    OSStatRdy = TRUE;
    OS_EXIT_CRITICAL();
}
```

统计任务 OSStat()的代码程序清单 3.14 所示。在前面一段中，已经讨论了为什么要等待统计任务就绪标志 OSStatRdy[程序清单 3.14(1)]。这个任务每秒执行一次，以确定所有应用程序中的任务消耗了多少 CPU 时间。当用户的应用程序代码加入以后，运行空闲任务的 CPU 时间就少了，OSIdleCtr 就不会像原来什么任务都不运行时有那么多计数。OSIdleCtr 的最大计数值是 OSStatInit()在初始化时保存在 OSIdleCtrMax 中的。CPU 利用率（表达式 [3.1]）则保存在变量 OSCPUUsage[程序清单 3.14(2)]中：

$$\text{OSCPUUsage} (\%) = 100 \left(1 - \frac{\text{OSIdleCtr}}{\text{OSIdleCtrMax}} \right) \quad [3.1]$$

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

```
void OSTaskStatHook(void *pdata)
{
    OS TICKS usage;
    OSIdleCtrMax = 0;
    OSIdleCtr = 0;
    OSIdleCtrRun = 0;
    OSIdleCtrMax = OS_TICKS(MIN_IDLE);
    OSIdleCtrRun = OS_IDLECLOCK;
    OSIdleCtr += OS_IDLECLOCK;
    OS_EXIT_CRITICAL();
    if (OSIdleCtr > OS_IDLECLOCK) {
        usage = (OS_TICKS(L1EL - 1000 * Ticks) / OSIdleCtrMax);
        if (usage > 100) {
            OSCPUUsage = 100;
        } else if (usage < 0) {
            OSCPUUsage = 0;
        } else {
            OSCPUUsage = usage;
        }
    } else {
        OSCPUUsage = 0;
    }
}
```

用任务统计外界接入函数 OSTaskStatHook() [程序 3.1]，这个函数能使统计任务得到扩展。这样，用户，每个任务执行时间的百分比以及其他信息（参见

超星阅览器提供
 使用本复制品
 请尊重相关知识产权!

用户中断服务子程序：	
保存全部 CPU 寄存器；	(1)
调用 OSIntEnter 或 OSIntNesting 增加 1；	(2)
执行用户代码做中断服务；	(3)
调用 OSIntLeave(1)；	(4)
恢复所有 CPU 寄存器；	(5)
执行中断返回指令；	(6)

μ C/OS 中，中断服务子程序要用汇编语言来写。然而，如果用户使用的 C 语言编译器支持在线汇编语言的话，用户可以直接将中断服务子程序代码放在 C 语言的程序文件中。中断服务子程序的示意代码如程序清单 3.15 所示。

程序清单 3.15 μ C/OS-II 中的中断服务子程序

用户代码应该将全部 CPU 寄存器推入当前任务栈[程序清单 3.15(1)]。注意，有些微处理器，例如 Motorola68020（及 68020 以上的微处理器），做中断服务时使用另外的堆栈。 μ C/OS-II 可以用在这类微处理器中，当任务切换时，寄存器是保存在被中断了的那个任务的栈中的。

μ C/OS-II 需要知道用户在做中断服务，故用户应该调用 OSIntEnter()，或者将全局变量 OSIntNesting[程序清单 3.15(2)]直接加 1，如果用户使用的微处理器有存储器直接加 1 的单条指令的话。如果用户使用的微处理器没有这样的指令，就必须先将 OSIntNesting 读入寄存器，再将寄存器加 1，然后再写回到变量 OSIntNesting 中去，在这种情况下调用 OSIntEnter()更简便。OSIntNesting 是共享资源。OSIntEnter()把上述三条指令用开中断、关中断保护起来，以保证处理 OSIntNesting 时的排它性。直接给 OSIntNesting 加 1 比调用 OSIntEnter()快得多，可能时，直接加 1 更好。要当心的是，在有些情况下，从 OSIntEnter()返回时，会把中断开了。遇到这种情况，在调用 OSIntEnter()之前要先清中断源，否则，中断将连续反复发生，用户应用程序就会崩溃！

上述两步完成以后，用户可以开始服务于申请中断的设备了[程序清单 3.15(3)]。这完

全取决于应用程序。 μ C/OS-II 允许中断嵌套，因为 μ C/OS-II 跟踪嵌套层数 OSIntNesting。然而，为允许中断嵌套，在多数情况下，用户应在开中断之前先清中断源。

调用脱离中断函数 OSIntExit() [程序清单 3.15(4)] 标志着中断服务子程序的终结，OSIntExit() 将中断嵌套层数计数器减 1。当嵌套计数器减到零时，所有中断，包括嵌套的中断就都完成了，此时 μ C/OS-II 要判定有没有优先级较高的任务被中断服务子程序（或任一嵌套的中断）唤醒了。如果有优先级高的任务进入了就绪态， μ C/OS-II 就返回到那个高优先级的任务，OSIntExit() 返回到调用点 [程序清单 3.15(5)]。保存的寄存器的值是在这时恢复的，然后是执行中断返回指令 [程序清单 3.16(6)]。注意，如果调度被禁止了 (OSIntNesting>0)， μ C/OS-II 将被返回到被中断了的任务。

以上描述的详细解释如图 3.5 所示。中断来到了 [图 3.5(1)] 但还不能被 CPU 识别，也许是因为中断被 μ C/OS-II 或用户应用程序关了，或者是因为 CPU 还没执行完当前指令。一旦 CPU 响应了这个中断 [图 3.5(2)]，CPU 的中断向量（至少大多数微处理器是如此）跳转到中断服务子程序 [图 3.5(3)]。如上所述，中断服务子程序保存 CPU 寄存器（也叫做 CPU context） [图 3.5(4)]，一旦做完，用户中断服务子程序通知 μ C/OS-II 进入中断服务子程序了，办法是调用 OSIntEnter() 或者给 OSIntNesting 直接加 1 [图 3.5(5)]。然后用户中断服务代码开始执行 [图 3.5(6)]。用户中断服务中做的事要尽可能地少，要把大部分工作留给任务去做。中断服务子程序通知某任务去做事的手段是调用以下函数之一：OSMboxPost()，OSQPost()，OSQPostFront()，OSSemPost()。中断发生并由上述函数发出消息时，接收消息的任务可能是，也可能不是挂起在邮箱、队列或信号量上的任务。用户中断服务完成以后，要调用 OSIntExit() [图 3.5(7)]。从时序图上可以看出，对被中断了的任务说来，如果没有高优先级的任务被中断服务子程序激活而进入就绪态，OSIntExit() 只占用很短的运行时间。进而，在这种情况下，CPU 寄存器只是简单地恢复 [图 3.5(8)] 并执行中断返回指令 [图 3.5(9)]。如果中断服务子程序使一个高优先级的任务进入了就绪态，则 OSIntExit() 将占用较长的运行时间，因为这时要做任务切换 [图 3.5(10)]。新任务的寄存器内容要恢复并执行中断返回指令 [图 3.5(12)]。

进入中断函数 OSIntEnter() 的代码如程序清单 3.16 所示，从中断服务中退出函数 OSIntExit() 的代码如程序清单 3.17 所示。如前所述，OSIntEnter() 所做的事是非常少的。

OSIntExit() 看起来非常像 OSSched()。但有三点不同。第一点，OSIntExit() 使中断嵌套层数减 1 [程序清单 3.17(2)] 而调度函数 OSSched() 的调度条件是：中断嵌套层数计数器和锁定嵌套计数器 (OSLockNesting) 二者都必须是零。第二个不同点是，OSRdyTbl[] 所需的检索值 Y 是保存在全局变量 OSIntExitY 中的 [程序清单 3.17(3)]。这是为了避免在任务栈中安排局部变量。这个变量在哪儿和中断任务切换函数 OSIntCtxSw() 有关（见 9.4.3）。最后一点，如果需要做任务切换，OSIntExit() 将调用 OSIntCtxSw() [程序清单 3.17(4)] 而不是像在 OSSched() 函数中那样调用 OS_TASK_SW()。

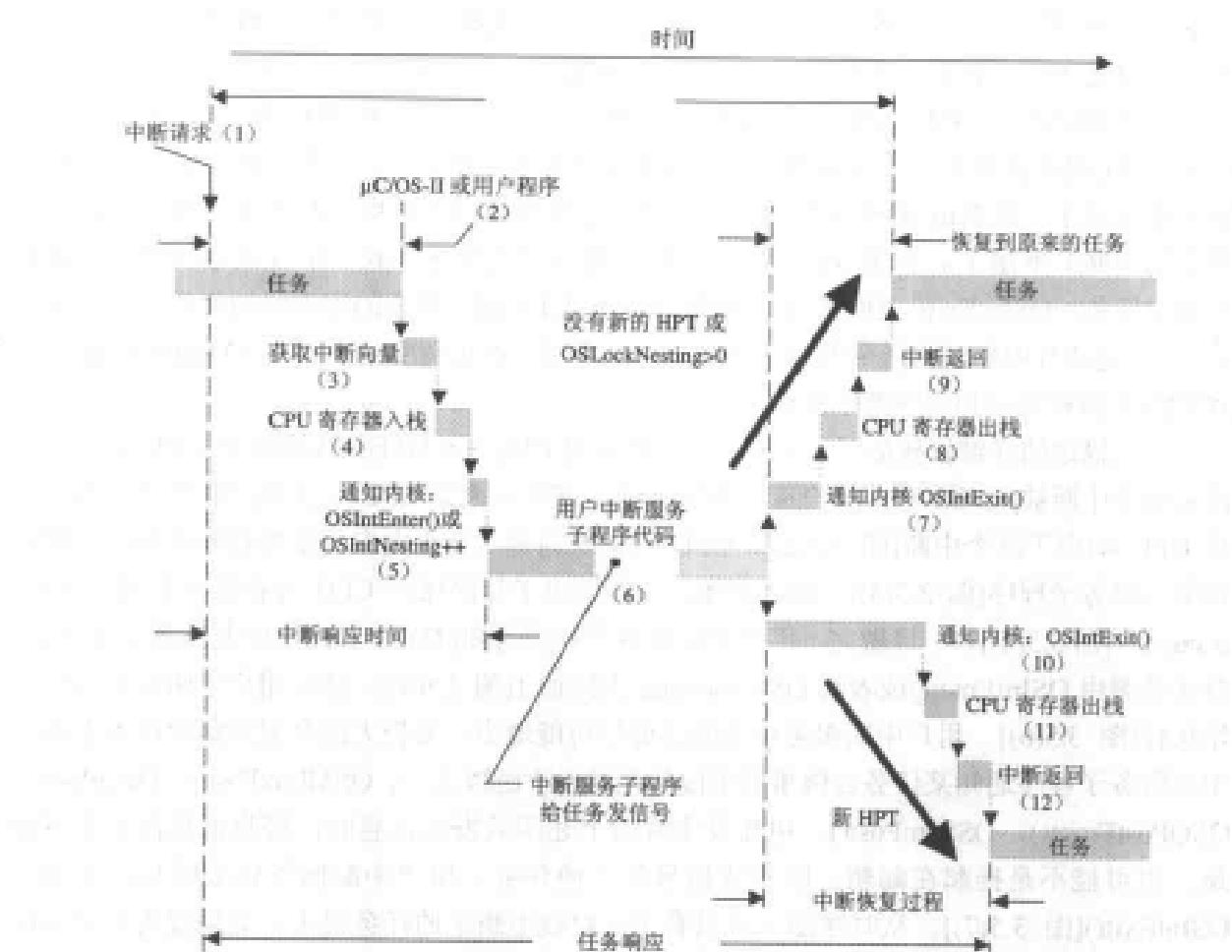


图3.5 中断服务

程序清单 3.16 通知μC/OS-II，中断服务子程序开始了

```
void OSIntEnter (void)
{
    OS_ENTER_CRITICAL();
    OSIntNesting++;
    OS_EXIT_CRITICAL();
}
```

程序清单 3.17 通知μC/OS-II，脱离了中断服务

```
void OSIntExit (void)
{}
```

```

OS_ENTER_CRITICAL();
if ((--OSIntesting < OSLOCKING) == 0) {
    OSINTERRUPT = OSINTERRUPT | OSREADYINT;
    OSINTLOCKED = (INT8U)(OSINTLOCK << 3) +
        OSINTLOCKED | OSREADYINT | OSINTLOCKED;
    OSCLRRC = 0;
    OSINTCAPTURE = 0;
}

OS_EXIT_CRITICAL();

```

第 3 章 内核结构

超星阅读器提醒您：
 使用本复制品
 请尊重相关知识产权！

调用中断切换函数 OSIntCtxSw()而不调用任务切换函数 OS_TASK_SW(), 有两个原因, 首先是, 如程序清单中程序清单 3.5(1)和图 3.6(1)所示, 一半的工作, 即 CPU 寄存器入栈的工作已经做完了。第二个原因是, 在中断服务子程序中调用 OSIntExit()时, 将返回地址推入了堆栈[程序清单 3.15(4)和图 3.6(2)]。OSIntExit()中的进入临界区函数 OS_ENTER_CRITICAL()或许将 CPU 的状态字也推入了堆栈[程序清单 3.7(1)和图 3.6(3)]。这取决于中断是怎么被关掉的(见第 8 章)。最后, 调用 OSIntCtxSw()时的返回地址又被推入了堆栈[程序清单 3.17(4)和图 3.1(4)], 除了栈中不相关的部分, 当任务挂起时, 栈结构应该与μC/OS-II 所规定的完全一致。OSIntCtxSw()只需要对栈指针做简单的调整, 如图 3.6(5)所示。换句话说, 调整栈结构要保证所有挂起任务的栈结构看起来是一样的。

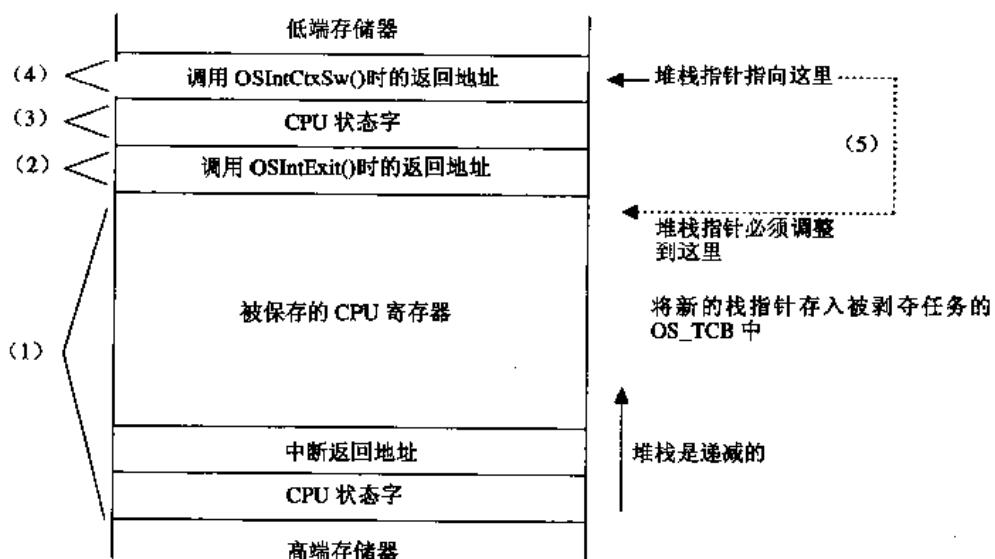


图3.6 中断中的任务切换函数OSIntCtxSw()调整栈结构

有的微处理器，像 Motorola 68HC11 中断发生时 CPU 寄存器是自动入栈的，且要想允许中断嵌套的话，在中断服务子程序中要重新开中断，这可以视作一个优点。确实，如果用户不需要通知任务自身进入了中断服务，只要不在 IntEnter()或 OSIntNesting 加 1。程序清单 3.18 中的示意代码表示了这种情况。一个任务和这个中断服务子程序通信的惟一方法是通过全程变量。

程序清单 3.18 Motorola 68HC11 中的中断服务子程序

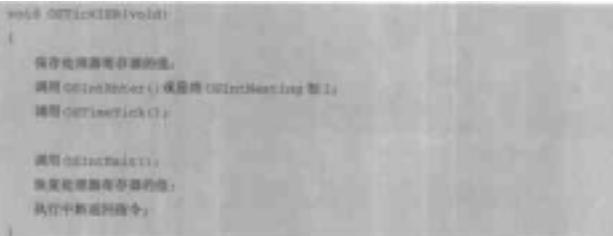
```
void main(void)
{
    Init();
    /* 初始化 μC/OS-II */
    /* 调用程序初始化代码 ... */
    /* ... 通过调用 OSTaskCreate() 创建至少一个任务 ... */
}
```

μC/OS 需要用户提供周期性信号源，用于实现时间延时和确认超时。节拍率应在每秒 10 次到 100 次之间，或者说 10 到 100Hz。时钟节拍率越高，系统的额外负荷就越重。时钟节拍的实际频率取决于用户应用程序的精度。时钟节拍源可以是专门的硬件定时器，也可以是来自 50/60Hz 交流电源的信号。

用户必须在多任务系统启动以后再启动时钟节拍计时，也就是在调用 OSStart()之后。换句话说，在调用 OSStart()之后做的第一件事是初始化定时器中断。通常，容易犯的错误是将允许时钟节拍器中断放在系统初始化函数 OSInit()之后，在调启动多任务系统启动函数 OSStart()之前，如程序清单 3.19 所示。

程序清单 3.19 启动时钟节拍计时的不正确做法

使用本复制品
请尊重相关知识产权！



可能在μC/OS-II 启动第一个任务之前发生，此时用户应用程序有可能会崩溃。

中断服务子程序中调用 OSTimeTick()实现的。时

钟节拍服务必须在启动前尽早中断处理的规则。时钟节拍中断服务子程序的示意代码如程序清单 3.20 所示。这段代码必须用汇编语言编写，因为在 C 语言里不能直接处理 CPU 的寄存器。

程序清单 3.20 时钟节拍中断服务子程序的示意代码

时钟节拍函数 OSTimeTick()的代码如程序清单 3.21 所示。OSTimtick()以调用可由用户定义的时钟节拍外连函数 OSTimTickHook()开始，这个外连函数可以将时钟节拍函数 OSTimtick()予以扩展[程序清单 3.2(1)]。笔者决定首先调用 OSTimTickHook()是打算在时钟节拍中断服务一开始就给用户一个可以做点儿什么的机会，因为用户可能会有一些时间要求苛刻的工作要做。OSTimtick()中量大的工作是给每个用户任务控制块 OS_TCB 中的时间延时项 OSTCBDly 减 1（如果该项不为零的话）。OSTimTick()从 OSTCBLList 开始，沿着 OS_TCB 链表做，一直做到空闲任务[程序清单 3.21(3)]。当某任务的任务控制块中的时间延时项 OSTCBDly 减到了零，这个任务就进入了就绪态[程序清单 3.21(5)]。而被任务挂起的函数 OSTaskSuspend()挂起的任务则不会进入就绪态[程序清单 3.21(4)]。OSTimTick()的执行时间直接与应用程序中建立了多少个任务成正比。

```

void OSTimeTick (void)
{
    OS_TCB *ptcb;
    OS_TICKS iTicks;
    ptcb = OS_CURRENT();
    while (ptcb->OSTICKTICK <= OS_IDLE_PRIO) {
        OS_ENTER_CRITICAL();
        if (ptcb->OSTICKTICK >= 0) {
            if ((ptcb->OSTICKTICK & OS_STM_PUSHPULL)) {
                OSAddDep      += ptcb->OSTICKTICK;
                OSAddDep(ptcb->OSTICKTICK) += ptcb->OSTICKTICK;
            }
            else {
                ptcb->OSTICKTICK += 1;
            }
        }
        ptcb = ptcb->OSTICKNEXT;
        OS_EXIT_CRITICAL();
    }
    OS_ENTER_CRITICAL();
    OSTIME++;
    OS_EXIT_CRITICAL();
}

```

|公开的实时嵌入式操作系统

ck() 的一个节拍服务

超星阅读器
使用本复制品
请尊重相关知识产权!

OSTimeTick()还通过调用 OSTime() [程序清单 3.21(7)] 累加从开机以来的时间，用的是一个无符号 32 位变量。注意，在给 OSTime 加 1 之前使用了关中断，因为多数微处理器给 32 位数加 1 的操作都得使用多条指令。

中断服务子程序似乎就得写这么长，如果用户不喜欢将中断服务程序写这么长，可以在任务级调用 OSTimeTick()，如程序清单 3.22 所示。要想这么做，得建立优先级一个高于应用程序中所有其他任务的任务。时钟节拍中断服务子程序利用信号量或邮箱发信号给这个高优先级的任务。

```
void TickTask (void *pdata)
{
    pdata = pdata;
    for (i=1;
        COMboxPend51...); /* 循环从时钟节拍中断服务程序发来的信号。*/
        OSTimeTick();
    }
}
```

3 章 内核结构

k() 进行时钟节拍服务

```
void OSTickISR(void)
{
    /* 保存处理器寄存器的值。
    使用 OSIntService() 或函数 OSIntPriority 时。*/
    /* 发送一个“空”消息(例如，(void *)1)到时钟节拍的邮箱。*/
    /* 调用 OSIntService();*/
    /* 恢复处理器寄存器的值。*/
    /* 执行中断返回指令。*/
}
```



用户当然需要先建立一个邮箱（初始化成 NULL）用于发信号给上述任何告知时钟节拍中断已经发生了（程序清单 3.23）。

程序清单 3.23 时钟节拍中断服务函数 OSTickISR()进行节拍服务

3.11 μC/OS-II 初始化

在调用 μC/OS-II 的其他服务之前，μC/OS-II 要求用户首先调用系统初始化函数 OSInit()。OSInit()初始化 μC/OS-II 所有的变量和数据结构（见 OS_CORE.C）。

OSInit()建立空闲任务，这个任务总是处于就绪态的。空闲任务 OSTaskIdle() 的优先级总是设成最低，即 OS_LOWEST_PRIO。如果统计任务允许 OS_TASK_STAT_EN 和任务建立扩展允许都设为 1，则 OSInit()还得建立统计任务 OSTaskStat()并且让其进入就绪态。OSTaskStat 的优先级总是设为 OS_LOWEST_PRIO-1。

图 3.7 表示调用 OSInit()之后，一些 μC/OS-II 变量和数据结构之间的关系。其解释是基于以下假设的：

- 在文件 OS_CFG.H 中，OS_TASK_STAT_EN 是设为 1 的；
- 在文件 OS_CFG.H 中，OS_LOWEST_PRIO 是设为 63 的；
- 在文件 OS_CFG.H 中，最多任务数 OS_MAX_TASKS 是设成大于 2 的。

以上两个任务的任务控制块（OS_TCB）是用双向链表链接在一起的。OSTCBLList 指向这个链表的起始处。当建立一个任务时，这个任务总是被放在这个链表的起始处。换句话说，OSTCBLList 总是指向最后建立的那个任务。链的终点指向空字符 NULL（也就是零）。

因为这两个任务都处在就绪态，在就绪任务表 OSRdyTbl[] 中的相应位是设为 1 的。还有，因为这两个任务的相应位是在 OSRdyTbl[] 的同一行上，即属同一组，故 OSRdyGrp 中只有 1 位是设为 1 的。

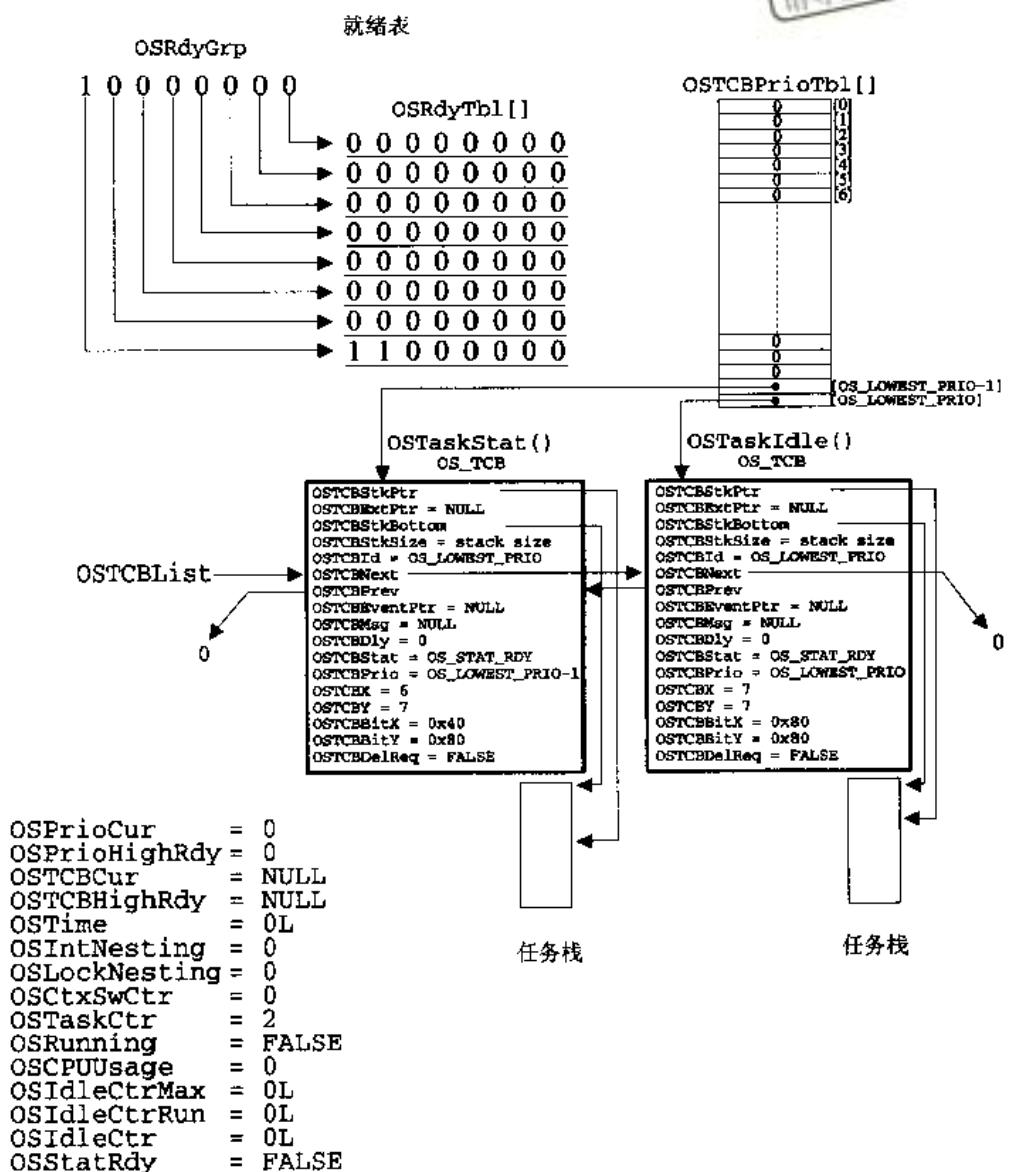


图3.7 调用OSInit()之后的数据结构

μ C/OS-II还初始化了4个空数据结构缓冲区，如图3.8所示。每个缓冲区都是单向链表，允许 μ C/OS-II从缓冲区中迅速得到或释放一个其中的元素。注意，在空缓冲区中空任务控制块的数目取决于最多任务数OS_MAX_TASKS，这个最多任务数是在OS_CFG.H文件中定义的。 μ C/OS-II自动安排总的系统任务数OS_N_SYS_TASKS（见文件 μ COS_II.H）。控制块OS_TCB的数目也就自动确定了。当然，包括足够的任务控制块分配给统计任务和空闲任务。指向空事件表OSEventFreeList和空队列表OSFreeList的指针将在第6章中讨论。指向空存储区的指针表OSMemFreeList将在第7章中讨论。

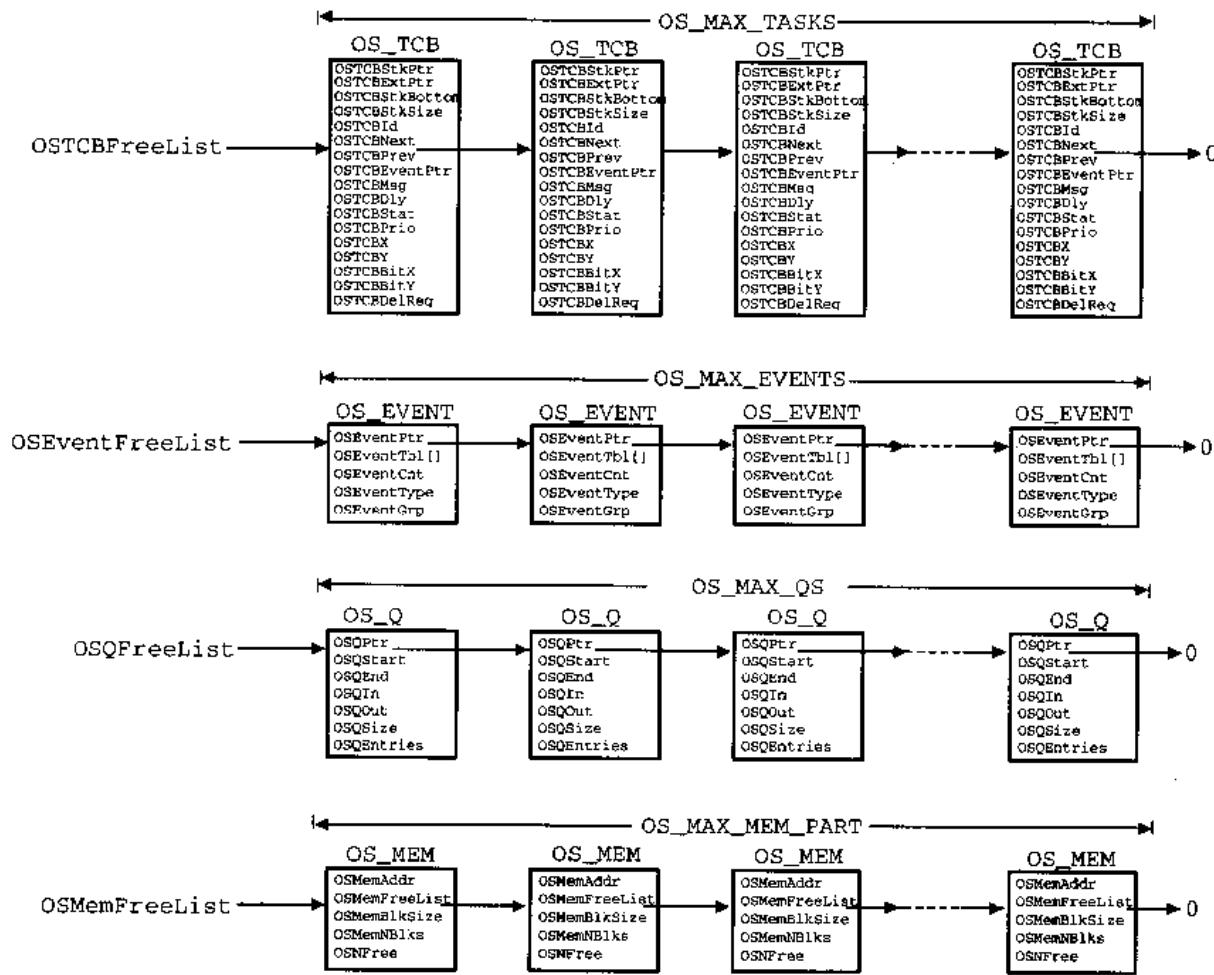


图3.8 空缓冲区

3.12 μ C/OS-II的启动

多任务的启动是用户通过调用OSStart()实现的。然而，启动 μ C/OS-II之前，用户至少要建立一个应用任务，如程序清单3.24所示。

```

void main(void)
{
    OSInit(); /* 初始化,见OS-11 */

    /* 通过调用 OSNewCreate() 或 OSTaskCreateStd() 创建至少一个任务。 */
    OSStart(); /* 不要多任务调度(OSStart())永远不会返回。*/
}

```



```

void OSStart(void)
{
    int80 yr;
    int80 xr;

    if (running == FALSE) {
        /* OSInit();
        * OSNewCreate(OSTaskCreateStd()); */
        OSStartHighRdy = (INT80)(yr << 32 + xr);
        OSReturn = xr;
        OSReturnHighRdy = OSStartHighRdy;
        OSReturnLowRdy = OSStartHighRdy;
        OSReturn = OSStartHighRdy;
        OSStartHighRdy();
    }
}

```

示。当调用 OSStart()时，OSStart()从任务就绪表中选取一个任务控制块[程序清单 3.25(1)]。然后，OSStart()调用 OSStartHighRdy() [程序清单 3.25(2)]（见汇编语言文件 `OS_CPU_A.ASM`），这个文件与选择的微处理器有关。实质上，函数 OSStartHighRdy() 是将任务栈中保存的值弹回到 CPU 寄存器中，然后执行一条中断返回指令，中断返回指令强制执行该任务代码。9.4.1 节将详细介绍对于 80x86 微处理器如何实现。注意，OSStartHighRdy() 将永远不返回到 OSStart()。

程序清单 3.25 启动多任务

多任务启动以后变量与数据结构中的内容如图 3.9 所示。这里笔者假设用户建立的任务优先级为 6，注意，OSTaskCtr 指出已经建立了 3 个任务。OSRunning 已设为“真”，指出多任务已经开始，OSPrioCur 和 OSPrioHighRdy 存放的是用户应用任务的优先级，OSTCBCur 和 OSTCBHighRdy 二者都指向用户任务的任务控制块。

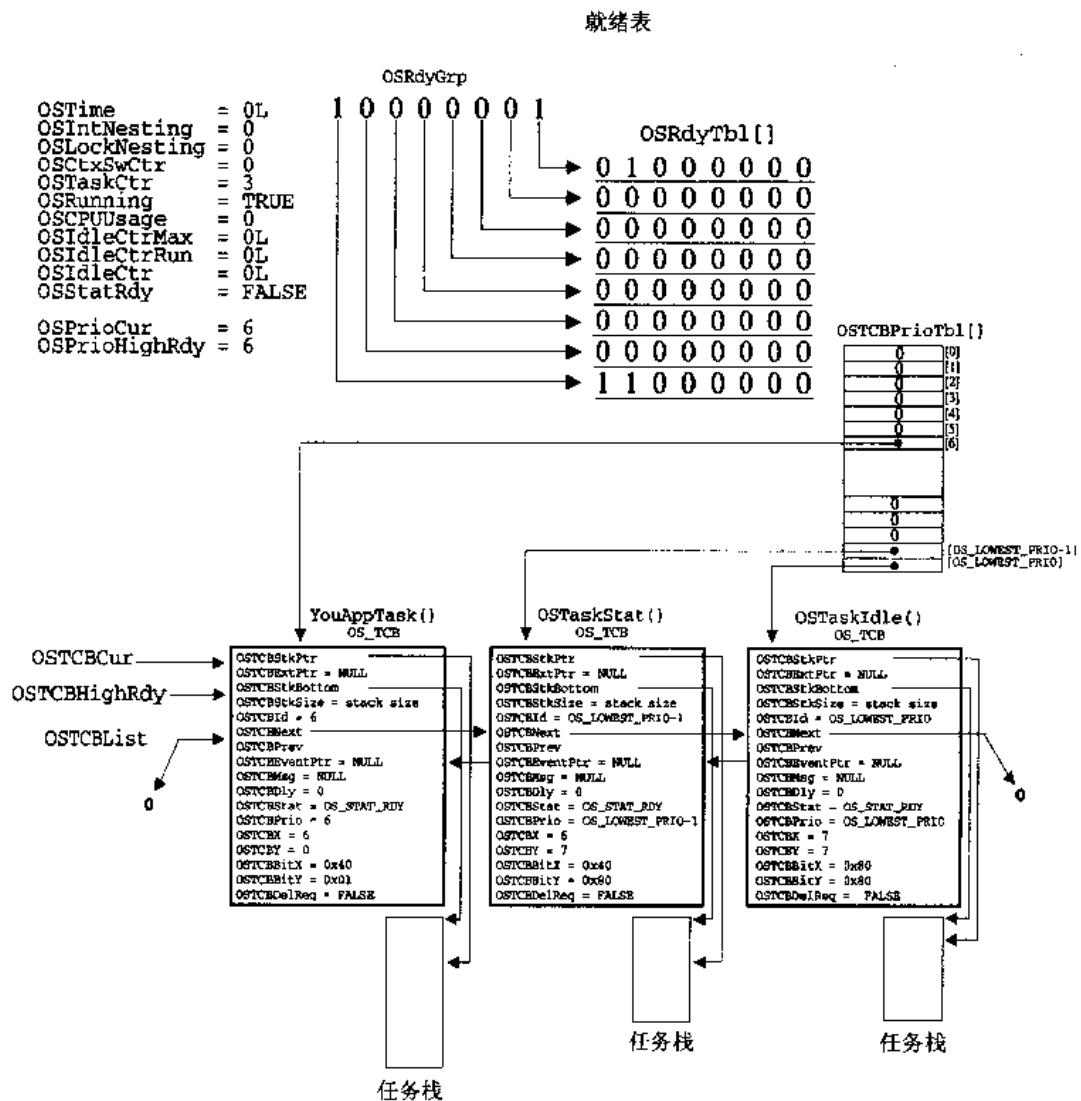


图3.9 调用OSStart()以后的变量与数据结构

3.13 获取当前μC/OS-II 的版本号

应用程序调用 OSVersion()(程序清单 3.26)可以得到当前μC/OS-II 的版本号。OSVersion()函数返回版本号值乘以 100。换言之，200 表示版本号 2.00。



为找到μC/OS-II 的最新版本以及如何做版本升级，用户可以与出版商联系，或者查看 μC/OS-II 的正式网站：www.uCOS-II.com。

3.14 OSEvent???()函数

读者或许注意到有 4 个 OS_CORE.C 中的函数没有在本章中提到。这 4 个函数是 OSEventWaitListInit(), OSEventTaskRdy(), OSEventTaskWait(), OSEventTO()。这几个函数是放在文件 OS_CORE.C 中的，而对如何使用这个函数的解释见第 6 章。

第4章

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

任务管理

可以是一个无限的循环，也可以在执行完一次后被
下是被真正的删除了，而只是μC/OS-II不再理会该
任务看起来与任何 C 函数一样，具有一个返回类
型和一个参数，只是它从不返回。任务的返回类型必须被定义成 void 型。在本章中所提到
前述，任务必须是以下两种结构之一：

```
void YourTask(void *pdata)
{
    for(;;)
    {
        /* 用户代码 */
        // 使用μC/OS-II 的准备例程之一
        OSOsTask();
        OSqHead();
        OSsemPend();
        OStaskAble(OS_PRIO_BLOCK);
        OStaskSuspend(OS_PRIO_BLOCK);
        OSrWait();
        OSrReady(OS_BLOCK);
        /* 用户代码 */
    }
}
```

```
void YourTask(void *pdata)
{
    /* 用户代码 */
    OStaskAble(OS_PRIO_BLOCK);
}
```

或：

本章所讲的内容包括如何在用户的应用程序中建立任务、删除任务、改变任务的优先级、挂起和恢复任务，以及获得有关任务的信息。

μC/OS-II可以管理多达64个任务，并从中保留了四个最高优先级和四个最低优先级的任务供自己使用，所以用户可以使用的只有56个任务。任务的优先级越高，反映优先级的值则越低。在最新的μC/OS-II版本中，任务的优先级数也可作为任务的标识符使用。

4.0 建立任务，OSTaskCreate()

```
INT32 OSTaskCreate (void (*task)(void *pdata), void *pdata, OS_STK *ptos, INT32 prio)
{
    void *pgm;
    OS_TCB *p_tcb;
    OS_PRIO *prio;
    OS_CRITICAL_BLOCK *p_cblk;

    if (prio < OS_LOWEST_PRIO) {
        return OS_PRIO_INVALID;
    }

    OS_ENTER_CRITICAL();
    if (OSTaskCreateExt(prio) == OS_TASK_INVALID) {
        OSLeaveCritical(p_cblk);
        OS_EXIT_CRITICAL();
        pgm = (void *)OSTaskCreateExt(task, pdata, ptos, prio);
        if (pgm != OS_INVALID) {
            p_tcb = OSGetTask(prio, pgm, OS_PRIO_ID(prio, 0, 0));
            if (p_tcb != OS_INVALID) {
                p_tcb->p_task = task;
                p_tcb->p_data = pdata;
                p_tcb->ptos = ptos;
                p_tcb->prio = prio;
            }
        }
    }
    OS_EXIT_CRITICAL();
}
```

必须要先建立任务。用户可以通过传递任务地址建立任务：OSTaskCreate()或OSTaskCreateExt()。向下兼容的，OSTaskCreateExt()是OSTaskCreate()两个函数中的任何一个都可以建立任务。任务可在任务的执行过程中被建立。在开始多任务调度(即任务。任务不能由中断服务程序（ISR）来建立。所述。从中可以知道，OSTaskCreate()需要四个参数：task是任务代码的指针，pdata是当任务开始执行时传递给任务的参数的指针，ptos是分配给任务的堆栈的栈顶指针（参看4.2节），prio是分配给任务的优先级。

程序清单4.1 OSTaskCreate()

```

OS_ENTER_CRITICAL();
OSTaskCreate();
OSTaskCreateExt(OSTCBPriority(prio));
OS_EXIT_CRITICAL();
LT((OSPriority)-1);
OSTaskDelete();
}
}
else {
OS_ENTER_CRITICAL();
OSFreeTaskTbl(prio) = OS_PCB * 10;
OS_EXIT_CRITICAL();
}
return (err);
}
else {
OS_EXIT_CRITICAL();
err = OS_PRIO_EXIST;
}
}

```



`OSTaskCreate()`一开始先检测分配给任务的优先级是否有效[程序清单 4.1(1)]。任务的优先级必须在 0 到 `OS_LOWEST_PRIO` 之间。接着, `OSTaskCreate()`要确保在规定的优先级上还没有建立任务[程序清单 4.1(2)]。在使用μC/OS-II 时, 每个任务都有特定的优先级。如果某个优先级是空闲的, μC/OS-II 通过放置一个非空指针在 `OSTCBPrioTbl[]` 中来保留该优先级[程序清单 4.1(3)]。这就使得 `OSTaskCreate()`在设置任务数据结构的其他部分时能重新允许中断[程序清单 4.1(4)]。

然后, `OSTaskCreate()`调用 `OSTaskStkInit()`[程序清单 4.1(5)], 它负责建立任务的堆栈。该函数是与处理器的硬件体系相关的函数, 可以在 `OS_CPU_C.C` 文件中找到。有关实现 `OSTaskStkInit()` 的细节可参看第 8 章。如果已经有人在你用的处理器上成功地移植了μC/OS-II, 而你又得到了他的代码, 就不必考虑该函数的实现细节了。`OSTaskStkInit()` 函数返回新的堆栈栈顶 (`psp`), 并被保存在任务的 `OS_TCB` 中。注意用户得将传递给 `OSTaskStkInit()` 函数的第四个参数 `opt` 置 0, 因为 `OSTaskCreate()` 与 `OSTaskCreateExt()` 不同, 它不支持用户为任务的创建过程设置不同的选项, 所以没有任何选项可以通过 `opt` 参数传递给 `OSTaskStkInit()`。

μC/OS-II 支持的处理器的堆栈既可以自上(高地址)往下(低地址)递减也可以自下往上递增。用户在调用 `OSTaskCreate()` 的时候必须知道堆栈是递增的还是递减的(参看所用处理器的 `OS_CPU.H` 中的 `OS_STACK_GROWTH`), 因为用户必须得把堆栈的栈顶传递给

OSTaskCreate(), 而栈顶可能是堆栈的最高地址（堆栈从上往下递减），也可能是最低地址（堆栈从下往上递增）。

一旦 OSTaskStkInit()函数完成了建立堆栈的任务，OSTaskCreate()就调用 OSTCBInit() [程序清单 4.1(6)] 从空闲的 OS_TCB 池中获得并初始化一个 OS_TCB。OSTCBInit() 的代码如 C 文件中而不是 OS_TASK.C 文件中。OSTCBInit() > S_TCB[程序清单 4.2(1)]，如果 OS_TCB 池中有空始化[程序清单 4.2(3)]。注意一旦 OS_TCB 被分配，即使这时内核又创建了其他的任务，这些新任务也

，所以 OSTCBInit()在这时就可以允许中断，并继

```
INTRO OSTCBInit (OS_TCB *ptcb, OS_STK *pstk, OS_STK *pmon, INT16U id,
                  INT16U stk_size, void *pmon, INT16U opct)

OS_TCB *ptccb;

OS_ENTER_CRITICAL();
ptcb = OSTCBFreeList;
IF (ptcb != (OS_TCB *)0) { (1)
    OSTCBFreeList = ptcb->OSTCBNext;
    OS_EXIT_CRITICAL(); (2)
    ptcb->OSTCBNext = ptccb; (3)
    ptcb->OSTCBMon = (INT16U)pmon;
    ptcb->OSTCBStat = OS_STAT_IDLE;
    ptcb->OSTCBBody = id;
}
ELSE OS_Task_Create_Error;
ptcb->OSTCBMonPtc = pstk;
ptcb->OSTCBIdleSize = stk_size;
ptcb->OSTCBCurrentSize = pmon;
ptcb->OSTCBOpct = opct;
ptcb->OSTCBId = id;
else
    pstk = pmon;
    stk_size = stk_size;
}

```

```

    pbos      = pbos;
    opt       = opt;
    id        = id;

#endif

#if OS_TASK_DEL_EN
    ptcb->OSTCDBDelReq = OS_NO_ERR;
#endif

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

#if OS_MBOX_EN || (OS_Q_EN && (OS_MAX_OS >= 2)) || OS_SEM_EN
    ptcb->OSTCBEEventPtr = (OS_EVENT *)0;
#endif

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

    OS_ENTER_CRITICAL();                                (4)
    OSTCBPrioTbl[prio] = ptcb;                        (5)
    ptcb->OSTCBNext = OSTCBLList;
    ptcb->OSTCBPrev = (OS_TCB *)0;
    if (OSTCBLList != (OS_TCB *)0) {
        OSTCBLList->OSTCBPrev = ptcb;
    }
    OSTCBLList = ptcb;
    OSRdyGrp |= ptcb->OSTCBBitY;                      (6)
    OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);                                (7)

} else {
    OS_EXIT_CRITICAL();
    return (OS_NO_MORE_TCB);
}
}

```

当 OSTCBInit()需要将 OS_TCB 插入到已建立任务的 OS_TCB 的双向链表中时[程序清单 4.2(5)]，它就禁止中断[程序清单 4.2(4)]。该双向链表开始于 OSTCBLList，而一个新任务的 OS_TCB 常常被插入到链表的表头。最后，该任务处于就绪状态[程序清单 4.2(6)]，并且 OSTCBInit()向它的调用者[OSTaskCreate()]返回一个代码表明 OS_TCB 已经被分配和初始化了[程序清单 4.2(7)]。

现在，我可以继续讨论 OSTaskCreate()（程序清单 4.1）函数了。从 OSTCBInit()返回后，OSTaskCreate()要检验返回代码[程序清单 4.1(7)]，如果成功，就增加 OSTaskCtr[程序清单 4.1(8)]，OSTaskCtr 用于保存产生的任务数目。如果 OSTCBInit()返回失败，就置 OSTCBPrioTbl [prio] 的入口为 0[程序清单 4.1(12)]以放弃该任务的优先级。然后，OSTaskCreate()调用 OSTaskCreateHook()[程序清单 4.1(9)]，OSTaskCreateHook()是用户自己定义的函数，用来扩展 OSTaskCreate()的功能。例如，用户可以通过 OSTaskCreateHook()函数来初始化和存储浮点寄存器、MMU 寄存器的内容，或者其他与任务相关的内容。一般情况下，用户可以在内存中存储一些针对用户的应用程序的附加信息。OSTaskCreateHook()既可以在 OS_CPU_C.C 中定义（如果 OS_CPU_HOOKS_EN 置 1），也可以在其他地方定义。注意，OSTaskCreate()在调用 OSTaskCreateHook()时，中断是关掉的，所以用户应该使 OSTaskCreateHook()函数中的代码尽量简化，因为这将直接影响中断的响应时间。OSTaskCreateHook()在被调用时会收到指向任务被建立时的 OS_TCB 的指针。这意味着该函数可以访问 OS_TCB 数据结构中的所有成员。

如果 OSTaskCreate()函数是在某个任务的执行过程中被调用（即 OSRunning 置为 True[程序清单 4.1(10)]），则任务调度函数会被调用[程序清单 4.1(11)]来判断是否新建立的任务比原来的任务有更高的优先级。如果新任务的优先级更高，内核会进行一次从旧任务到新任务的任务切换。如果在多任务调度开始之前[即用户还没有调用 OSStart()]，新任务就已经建立了，则任务调度函数不会被调用。

4.1 建立任务，OSTaskCreateExt()

用 OSTaskCreateExt() 函数来建立任务会更加灵活，但会增加一些额外的开销。OSTaskCreateExt()函数的代码如程序清单 4.3 所示。

我们可以看到 OSTaskCreateExt()需要 9 个参数！前 4 个参数（task, pdata, ptos 和 prio）与 OSTaskCreate()的 4 个参数完全相同，连先后顺序都一样。这样做的目的是为了使用户能够更容易地将用户的程序从 OSTaskCreate()移植到 OSTaskCreateExt()上去。

id 参数为要建立的任务创建一个特殊的标识符。该参数在 μC/OS 以后的升级版本中可能会用到，但在 μC/OS-II 中还未使用。这个标识符可以扩展 μC/OS-II 功能，使它可以执行的任务数超过目前的 64 个。但在这里，用户只要简单地将任务的 id 设置成与任务的优先级一样的值就可以了。

pbos 是指向任务的堆栈栈底的指针，用于堆栈的检验。

stk_size 用于指定堆栈成员的数目。也就是说，如果堆栈的入口宽度为 4 字节宽，那么 stk_size 为 10000 是指堆栈有 40000 个字节。该参数与 pbos 一样，也用于堆栈的检验。

pext 是指向用户附加的数据域的指针，用来扩展任务的 OS_TCB。例如，用户可以为

例 3)，或是在任务切换过程中将浮点寄存器的内

顷，指定是否允许堆栈检验，是否将堆栈清零，任
文件中有一个所有可能选项(OS_TASK_OPT_STK_
_TASK_OPT_SAVE_FP) 的常数表。每个选项占有
户在使用时只需要将以上 OS_TASK_OPT_???选项

```
OSNewTaskCreate(void *TTask, void *pb0,
    void *pdata,
    OS_STK *pTOS,
    INT8U *pri,
    INT16U *L,
    OS_STK *pExt,
    INT32U stk_size,
    void *pExt,
    INT32U opt);

void *pExt;
INT8U *pri;
INT16U *L;
OS_STK *pExt;
}

if (pri < os_lowest_prio) {                                (1)
    return OS_PRIO_INVALID;
}

OS_ENTER_CRITICAL();                                         (2)
if (OSTCB[pri].opt == OS_PRIO_INVALID) {                   (3)
    OSTCB[pri].opt = OS_PRIO_INVALID;
    OS_EXIT_CRITICAL();                                     (4)
```

```

    if (opt & OS_TASK_OPT_STK_CHK) {                                (5)
        if (opt & OS_TASK_OPT_STK_CLR) {
            Pfill = pbos;
            for (i = 0; i < stk_size; i++) {
                #if OS_STK_GROWTH == 1
                *pfill++ = (OS_STK)0;
                #else
                *pfill-- = (OS_STK)0;
                #endif
            }
        }
        psp = (void *)OSTaskStkInit(task, pdata, ptos, opt);      (6)
        err = OSTCBInit(prio, psp, pbos, id, stk_size, pext, opt); (7)
        if (err == OS_NO_ERR) {                                       (8)
            OS_ENTER_CRITICAL();
            OSTaskCtr++;
            OSTaskCreateHook(OSTCBPrioTbl[prio]);                  (10)
            OS_EXIT_CRITICAL();
            if (!OSRunning) {                                      (11)
                OSSched();                                     (12)
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;                   (13)
            OS_EXIT_CRITICAL();
        }
        return (err);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    }
}

```

OSTaskCreateExt()一开始先检测分配给任务的优先级是否有效[程序清单 4.3(1)]。任务的优先级必须在 0 到 OS_LOWEST_PRIO 之间。接着，OSTaskCreateExt()要确保在规定的

优先级上还没有建立任务[程序清单 4.3(2)]。在使用μC/OS-II 时，每个任务都有特定的优先级。如果某个优先级是空闲的，μC/OS-II 通过放置一个非空指针在 OSTCBPrioTbl[] 中来保留该优先级[程序清单 4.3(3)]。这就使得 OSTaskCreateExt() 在设置任务数据结构的其他部分时能重新允许中断[程序清单 4.3(4)]。

为了对任务的堆栈进行检验[参看 4.3 节]，用户必须在 opt 参数中设置 OS_TASK_OPT_STK_CHK 标志。堆栈检验还要求在任务建立时堆栈的存储内容都是 0（即堆栈已被清零）。为了在任务建立的时候将堆栈清零，需要在 opt 参数中设置 OS_TASK_OPT_STK_CLR。当以上两个标志都被设置好后，OSTaskCreateExt() 才能将堆栈清零[程序清单 4.3(5)]。

接着，OSTaskCreateExt() 调用 OSTaskStkInit()[程序清单 4.3(6)]，它负责建立任务的堆栈。该函数是与处理器的硬件体系相关的函数，可以在 OS_CPU_C.C 文件中找到。有关实现 OSTaskStkInit() 的细节可参看第 8 章。如果已经有人在你用的处理器上成功地移植了 μC/OS-II，而你又得到了他的代码，就不必考虑该函数的实现细节了。OSTaskStkInit() 函数返回新的堆栈栈顶(psp)，并被保存在任务的 OS_TCB 中。

μC/OS-II 支持的处理器的堆栈既可以从上（高地址）往下（低地址）递减也可以从下往上递增（参看 4.2）。用户在调用 OSTaskCreateExt() 的时候必须知道堆栈是递增的还是递减的（参看用户所用处理器的 OS_CPU.H 中的 OS_STACK_GROWTH），因为用户必须得把堆栈的栈顶传递给 OSTaskCreateExt()，而栈顶可能是堆栈的最低地址（当 OS_STK_GROWTH 为 0 时），也可能是最高地址（当 OS_STK_GROWTH 为 1 时）。

一旦 OSTaskStkInit() 函数完成了建立堆栈的任务，OSTaskCreateExt() 就调用 OSTCBInit() [程序清单 4.3(7)]，从空闲的 OS_TCB 缓冲池中获得并初始化一个 OS_TCB。OSTCBInit() 的代码在 OSTaskCreate() 中曾描述过（参看 4.0 节），从 OSTCBInit() 返回后，OSTaskCreateExt() 要检验返回代码[程序清单 4.3(8)]，如果成功，就增加 OSTaskCtr[程序清单 4.3(9)]，OSTaskCtr 用于保存产生的任务数目。如果 OSTCBInit() 返回失败，就置 OSTCBPrioTbl[prio] 的入口为 0[程序清单 4.3(13)] 以放弃对该任务优先级的占用。然后，OSTaskCreateExt() 调用 OSTaskCreateHook() [程序清单 4.3(10)]，OSTaskCreateHook() 是用户自己定义的函数，用来扩展 OSTaskCreateExt() 的功能。OSTaskCreateHook() 可以在 OS_CPU_C.C 中定义（如果 OS_CPU_HOOKS_EN 置 1），也可以在其他地方定义（如果 OS_CPU_HOOKS_EN 置 0）。注意，OSTaskCreateExt() 在调用 OSTaskCreateHook() 时，中断是关掉的，所以用户应该使 OSTaskCreateHook() 函数中的代码尽量简化，因为这将直接影响中断的响应时间。OSTaskCreateHook() 被调用时会收到指向任务被建立时的 OS_TCB 的指针。这意味着该函数可以访问 OS_TCB 数据结构中的所有成员。

如果 OSTaskCreateExt() 函数是在某个任务的执行过程中被调用的（即 OSRunning 置为 True[程序清单 4.3(11)]），以任务调度函数会被调用[程序清单 4.3(12)] 来判断是否新建立的任务比原来的任务有更高的优先级。如果新任务的优先级更高，内核会进行一次从旧任务到新任务的任务切换。如果在多任务调度开始之前[即用户还没有调用 OSStart()]，新任务就已经建立了，则任务调度函数不会被调用。

4.2 任务堆栈



static OS_STK MyTaskStack[stack_size];

或：

程序清单 4.5 静态堆栈

OS_STK *MyTaskStack(stack_size);

必须声明为 OS_STK 类型，并且由连续的内存空间（在编译的时候分配）也可以动态地分配堆栈空间（在运行的时候分配）。静态堆栈声明如程序清单 4.4 和 4.5 所示，这两种声明应放置在函数的外面。

程序清单 4.4 静态堆栈

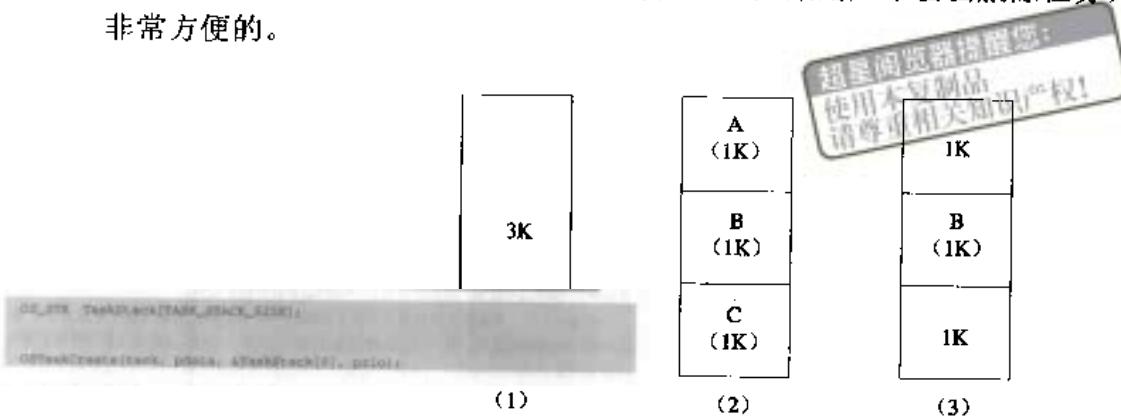
```
OS_STK *pstk;
stack = (OS_STK *)malloc(stack_size);
if (stack != (OS_STK *)100) /* 确认 malloc() 能得到足够内存空间 */
    CreateTheTask();
}
```

用户可以用 C 编译器提供的 malloc() 函数来动态地分配堆栈空间，如程序清单 4.6 所示。在动态分配中，用户要时刻注意内存碎片问题。特别是当用户反复地建立和删除任务时，内存堆中可能会出现大量的内存碎片，导致没有足够大的一块连续内存区域可用作任务堆栈，这时 malloc() 便无法成功地为任务分配堆栈空间。

程序清单 4.6 用 malloc() 为任务分配堆栈空间

图 4.1 表示了一块能被 malloc() 动态分配的 3K 字节的内存堆 [图 4.1(1)]。为了讨论问题方便，假定用户要建立三个任务（任务 A、B 和 C），每个任务需要 1K 字节的空间。设第一个 1K 字节给任务 A，第二个 1K 字节给任务 B，第三个 1K 字节给任务 C [图 4.1(2)]。然后，用户的应用程序删除任务 A 和任务 C，用 free() 函数释放内存到内存堆中 [图 4.1(3)]。现在，用户的内存堆虽有 2K 字节的自由内存空间，但它是不连续的，所以用户不能建立

另一个需要 2K 字节内存的任务（即任务 D）。如果用户不会去删除任务，使用 `malloc()` 是非常方便的。



4.1 内存碎片

从上（高地址）往下（低地址）递减也可以从下往上递增（参看 4.2 节）。用户在调用 `OSTaskCreate()` 或 `OSTaskCreateExt()` 的时候必须知道堆栈是递增还是递减的，因为用户必须得把堆栈的栈顶传递给以上两个函数，当 `OS_CPU.H` 文件中的 `OS_STK_GROWTH` 置为 0 时，用户需要将堆栈的最低内存地址传递给任务创建函数，如程序清单 4.7 所示。

程序清单 4.7 堆栈从下往上递增

当 `OS_CPU.H` 文件中的 `OS_STK_GROWTH` 置为 1 时，用户需要将堆栈的最高内存地址传递给任务创建函数，如程序清单 4.8 所示。

程序清单 4.8 堆栈从上往下递减

这个问题会影响代码的可移植性。如果用户想将代码从支持往下递减堆栈的处理器中移植到支持往上递增堆栈的处理器中的话，用户得使代码同时适应以上两种情况。在这种特殊情况下，程序清单 4.7 和 4.8 可重新写成如程序清单 4.9 所示的形式。

```
OS_STK_TaskStack(TASK_STACK_SIZE);
#LY OS_STK_GROWTH == 0
OSTaskCreate(task, pdata, &taskStack[0], prio);
forSee
OSTaskCreateExt(task, pdata, &taskStack[TASK_STACK_SIZE-1], prio);
Handle
```



任务所需的堆栈的容量是由应用程序指定的。用户在指定堆栈大小的时候必须考虑用户的任务所调用的所有函数的嵌套情况，任务所调用的所有函数会分配的局部变量的数目，以及所有可能的中断服务程序嵌套的堆栈需求。另外，用户的堆栈必须能存储所有的 CPU 寄存器。

4.3 堆栈检验，OSTaskStkChk()

有时候决定任务实际所需的堆栈空间大小是很有必要的。因为这样用户就可以避免为任务分配过多的堆栈空间，从而减少自己的应用程序代码所需的 RAM（内存）数量。 μ C/OS-II 提供的 OSTaskStkChk() 函数可以为用户提供这种有价值的信息。

在图 4.2 中，笔者假定堆栈是从上往下递减的（即 OS_STK_GROWTH 被置为 1），但以下的讨论也同样适用于从下往上递增的堆栈[图 4.2(1)]。 μ C/OS-II 是通过查看堆栈本身的内容来决定堆栈的方向的。只有内核或是任务发出堆栈检验的命令时，堆栈检验才会被执行，它不会自动地不断检验任务的堆栈使用情况。在堆栈检验时， μ C/OS-II 要求在任务建立的时候堆栈中存储的必须是 0 值（即堆栈被清零）[图 4.2(2)]。另外， μ C/OS-II 还需要知道堆栈栈底（BOS）的位置和分配给任务的堆栈的大小[图 4.2(2)]。在任务建立的时候，BOS 的位置及堆栈的大小这两个值存储在任务的 OS_TCB 中。

为了使用 μ C/OS-II 的堆栈检验功能，用户必须要做以下几件事情：

- 在 OS_CFG.H 文件中设 OS_TASK_CREATE_EXT 为 1。
- 用 OSTaskCreateExt() 建立任务，并给予任务比实际需要更多的内存空间。
- 在 OSTaskCreateExt() 中，将参数 opt 设置为 OS_TASK_OPT_STK_CHK+OS_TASK_OPT_STK_CLR。注意如果用户的程序启动代码清除了所有的 RAM，并且从未删除过已建立了的任务，那么用户就不必设置选项 OS_TASK_OPT_STK_CLR 了。这样就会减少 OSTaskCreateExt() 的执行时间。
- 将用户想检验的任务的优先级作为 OSTaskStkChk() 的参数并调用之。

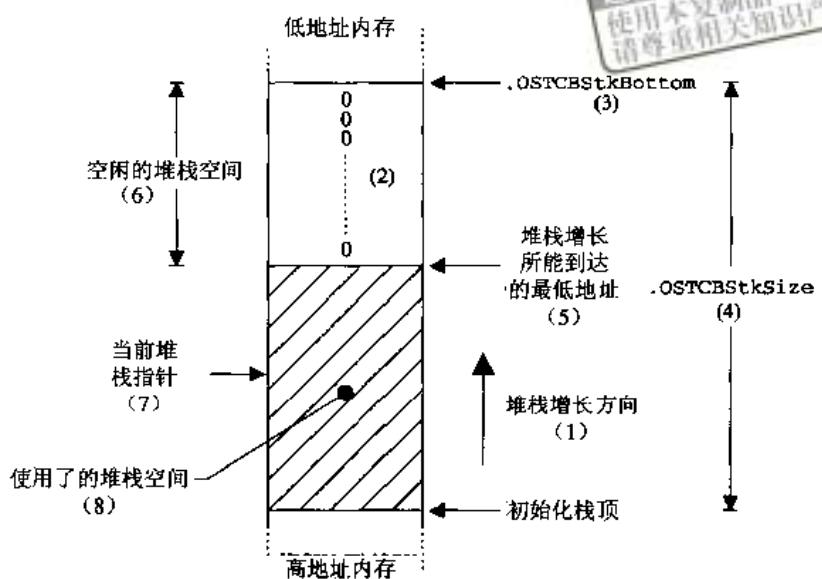


图4.2 堆栈检验

`OSTaskStkChk()`顺着堆栈的栈底开始计算空闲的堆栈空间大小，具体实现方法是统计存储内容为 0 的连续堆栈地址的数目，直到发现内容不为 0 的堆栈[图 4.2(5)]。注意堆栈地址的存储内容在进行检验时使用的是堆栈的数据类型（参看 `OS_CPU.H` 中的 `OS_STK`）。换句话说，如果堆栈的地址有 32 位宽，对 0 值的比较也是按 32 位完成的。所用的堆栈的空间大小是指从用户在 `OSTaskCreateExt()` 中定义的堆栈大小中减去了存储内容为 0 的连续堆栈地址以后的大小。`OSTaskStkChk()` 实际上把空闲堆栈的字节数和已用堆栈的字节数放置在 `OS_STK_DATA` 数据结构中（参看 `μCOS_II.H`）。注意在某个给定的时间，被检验任务的堆栈指针可能会指向最初的堆栈栈顶（`TOS`）与堆栈最深处之间的任何位置[图 4.2(7)]。每次在调用 `OSTaskStkChk()` 的时候，用户也可能会因为任务还没触及堆栈的最深处而得到不同的堆栈的空闲空间数。

用户应该使自己的应用程序运行足够长的时间，并且经历最坏的堆栈使用情况，这样才能得到正确的数。一旦 `OSTaskStkChk()` 提供给用户最坏情况下堆栈的需求，用户就可以重新设置堆栈的最终容量了。为了适应系统以后的升级和扩展，用户应该多分配 10%~100% 的堆栈空间。在堆栈检验中，用户所得到的只是一个大致的堆栈使用情况，并不能说明堆栈使用的全部实际情况。

`OSTaskStkChk()` 函数的代码如程序清单 4.10 所示。`OS_STK_DATA`（参看 `μCOS_II.H`）数据结构用来保存有关任务堆栈的信息。笔者打算用一个数据结构来达到两个目的。第一，把 `OSTaskStkChk()` 当作是查询类型的函数，并且使所有的查询函数用同样的方法返回，即返回查询数据到某个数据结构中。第二，在数据结构中传递数据使得笔者可以在不改变 `OSTaskStkChk()` 的 API（应用程序编程接口）的条件下为该数据结构增加其他域，从而扩展 `OSTaskStkChk()` 的功能。现在，`OS_STK_DATA` 只包含两个域：`OSFree` 和 `OSUsed`。从代码中用户可看到，通过指定执行堆栈检验的任务的优先级可以调用 `OSTaskStkChk()`。如果用

户指定 OS_PRIO_SELF[程序清单 4.10(1)]，那么就表明用户想知道当前任务的堆栈信息。当然，前提是任务已经存在[程序清单 4.10(2)]。要执行堆栈检验，用户必须已用

至传递了选项 OS_TASK_OPT_CHK[程序清单 4.10(3)]。TaskStkChk()就会像前面描述的那样从堆栈栈底开始遍历。最后，存储在 OS_STK_DATA 中的信息就被确定的是堆栈的实际空闲字节数和已被占用的字节数。堆栈的实际大小（用字节表示）就是该项之和。

```
INT8U OSTaskStkChk( INT8U prio, OS_STK_DATA *pdata)
{
    OS_TCB *p_tcb;
    OS_STK *p_stk;
    INT32U free;
    INT32U used;
    INT32U size;
    INT32U offset;

    pdata->OSFree = 0;
    pdata->OSUsed = 0;
    if (prio > OS_LOWEST_PRIO || prio < OS_HIGHEST_PRIO) {
        return OS_PRIO_INVALID;
    }

    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                                (1)
        p_tcb = OS_tcbFromPriority;
        if (p_tcb == OS_tcb + 10) {                            (2)
            OS_EXIT_CRITICAL();
            return OS_TASK_NOT_EXIST;
        }
        if (p_tcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) {          (3)
            OS_ENTER_CRITICAL();
            return OS_TASK_OPT_ERR;
        }
    }
    offset = -OS_STK_SIZE;
    free = 0;
    used = 0;
}
```

```

    size = p->OS->OSTaskSize();
    pOK = p->OS->OSTaskOK();
    OS_EXIT_CRITICAL();
    if( OS_STK_GROWTH == 1 )
        while( *pstk++ == 0 ) ;
    else
        while( *pstk-- == 0 ) ;
    if( p->OS->OSTaskFree + free * sizeof(OS_STK) <= p->OS->OSTaskUsed + task - free * sizeof(OS_STK) )
        return( OS_NO_ERROR );
}
#endif;

```

超星浏览器提醒您：
 使用本复制品
 请尊重相关知识产权！

```

OSMU_OSTaskDel( OSMUS ptoi )
{
    OS_TCB *ptcb;
    OS_EVENT *pevent;
}

```

4.4 删除任务，OSTaskDel()

有时候删除任务是很有必要的。删除任务,是说任务将返回并处于休眠状态(参看 3.2 节), 并不是说任务的代码被删除了, 只是任务的代码不再被μC/OS-II 调用。通过调用 OSTaskDel()就可以完成删除任务的功能(如程序清单 4.11 所示)。OSTaskDel()一开始应确保用户所要删除的任务并非是空闲任务, 因为删除空闲任务是不允许的[程序清单 4.11(1)]。不过, 用户可以删除统计任务[程序清单 4.11(2)]。接着, OSTaskDel()还应确保用户不是在 ISR 例程中去试图删除一个任务, 因为这也是不被允许的[程序清单 4.11(3)]。调用此函数的任务可以通过指定 OS_PRIO_SELF 参数来删除自己[程序清单 4.11(4)]。接下来 OSTaskDel()会保证被删除的任务是确实存在的[程序清单 4.11(3)]。如果指定的参数是 OS_PRIO_SELF 的话, 这一判断过程(任务是否存在)自然是可以通过的, 但笔者不准备为这种情况单独写一段代码, 因为这样只会增加代码并延长程序的执行时间。

程序清单 4.11 删除任务

```

if (prio == OS_IDLE_PRIO) {                                (1)
    return (OS_TASK_DEL_IDLE);
}

if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {      (2)
    return (OS_PRIO_INVALID);
}

OS_ENTER_CRITICAL();
if (OSIntNesting > 0) {                                    (3)
    OS_EXIT_CRITICAL();
    return (OS_TASK_DEL_ISR);
}

if (prio == OS_PRIO_SELF) {                                (4)
    Prio = OSTCBCur->OSTCBPrio;
}

if (!ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) {          (5)
    if ((OSRdyTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0) { (6)
        OSRdyGrp &= ~ptcb->OSTCBBity;
    }

    if ((pevent = ptcb->OSTCBEVENTPTR) != (OS_EVENT *)0) { (7)
        if ((pevent->OSEventTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0) {
            pevent->OSEventGrp &= ~ptcb->OSTCBBity;
        }
    }
}

Ptcb->OSTCBDly = 0;                                         (8)
Ptcb->OSTCBSstat = OS_STAT_RDY;                            (9)
OSLockNesting++;                                           (10)
OS_EXIT_CRITICAL();                                         (11)
OSDummy();                                                 (12)
OS_ENTER_CRITICAL();                                         (13)
OSLockNesting--;                                           (14)
OSTaskDelHook(ptcb);                                       (15)
OSTaskCtr--;
OSTCBPrioTbl[prio] = (OS_TCB *)0;
if (ptcb->OSTCBPrev == (OS_TCB *)0) {                      (16)
    ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
}

```

```

    OSCTCBList      = pc_cb->OSCTCBList;
} else {
    pc_cb->OSTCBPrev = pc_cb->OSTCBNext;
    pc_cb->OSTCBNext = pc_cb->OSTCBPrev;
    pc_cb->OSTCBWaitList = pc_cb;
    OS_EXIT_CRITICAL();
    OSTaskDel();
    return (OS_NO_ERR);
} else {
    OS_EXIT_CRITICAL();
    return (OS_TASK_DEL_ERR);
}

```

超星阅览器提醒您：
 使用本复制品
 请尊重相关知识产权！

一旦所有条件都满足了，该任务的 OS_TCB 就会从所有可能的μC/OS-II 的数据结构中被移去。OSTaskDel()分两步完成该移去任务以减少中断响应时间。首先，如果任务处于就绪表中，它会直接被移去[程序清单 4.11(6)]。如果任务处于邮箱、消息队列或信号量的等待表中，它就从自己所处的表中被移去[程序清单 4.11(7)]。接着，OSTaskDel()将任务的时钟延迟数清零，以确保自己重新允许中断的时候，ISR 例程不会使该任务就绪[程序清单 4.11(8)]。最后，OSTaskDel()置任务的.OSTCBStat 标志为 OS_STAT_RDY。注意，OSTaskDel()并不是试图使任务处于就绪状态，而是阻止其他任务或 ISR 例程让该任务重新开始执行（即避免其他任务或 ISR 调用 OSTaskResume() [程序清单 4.11(9)]）。这种情况是有可能发生的，因为 OSTaskDel()会重新打开中断，而 ISR 可以让更高优先级的任务处于就绪状态，这就可能会使用户想删除的任务重新开始执行。如果不想置任务的.OSTCBStat 标志为 OS_STAT_RDY，就只能清除 OS_STAT_SUSPEND 位了（这样代码可能显得更清楚，更容易理解一些），但这样会使得处理时间稍长一些。

要使被删除的任务不会被其他的任务或 ISR 置于就绪状态，因为该任务已从就绪任务表中删除了，它不是在等待事件的发生，也不是在等待延时期满，不能重新被执行。为了达到删除任务的目的，任务被置于休眠状态。正因为这样，OSTaskDel()必须得阻止任务调度程序[程序清单 4.11(10)]在删除过程中切换到其他的任务中去，因为如果当前的任务正在被删除，它不可能被再次调度！接下来，OSTaskDel()重新允许中断以减少中断的响应时间[程序清单 4.11(11)]。这样，OSTaskDel()就能处理中断服务了，但由于它增加了 OSLockNesting，ISR 执行完后会返回到被中断任务，从而继续任务的删除工作。注意 OSTaskDel()此时还没有完全完成删除任务的工作，因为它还需要从 TCB 链表中解开该任务的 OS_TCB，并将该

OS_TCB 返回到空闲 OS_TCB 表中。

另外需要注意的是，笔者在调用 OS_EXIT_CRITICAL()函数后，马上调用了 OSDummy() [程序清单 4.11(12)]，该函数并不会进行任何实质性的工作。这样做只是因为想确保处理器在中断允许的情况下至少执行一个指令。对于许多处理器来说，执行中断允许指令会强制 CPU 禁止中断直到下个指令结束！Intel 80x86 和 Zilog Z-80 处理器就是如此工作的。开中断后马上关中断就等于从来没开过中断，当然这会增加中断的响应时间。因此调用 OSDummy()确保在再次禁止中断之前至少执行了一个调用指令和一个返回指令。当然，用户可以用宏定义将 OSDummy()定义为一个空操作指令（译注 1），这样调用 OSDummy()就等于执行了一个空操作指令，会使 OSTaskDel()的执行时间稍微缩短一点。但笔者认为这种宏定义是没价值的，因为它会增加移植 μC/OS-II 的工作量。

现在，OSTaskDel()可以继续执行删除任务的操作了。在 OSTaskDel()重新关中断后，它通过锁定嵌套计数器 (OSLockNesting) 减一以重新允许任务调度 [程序清单 4.11(13)]。接着，OSTaskDel()调用用户自定义的 OSTaskDelHook()函数 [程序清单 4.11(14)]，用户可以在这里删除或释放自定义的 TCB 附加数据域。然后，OSTaskDel()减少 μCOS-II 的任务计数器。OSTaskDel()简单地将指向被删去的任务的 OS_TCB 的指针指向 NULL [程序清单 4.11(15)]，从而达到将 OS_TCB 从优先级表中移去的目的。再接着，OSTaskDel()将被删除的任务的 OS_TCB 从 OS_TCB 双向链表中移去 [程序清单 4.11(16)]。注意，没有必要检验 ptcb->OSTCBNext==0 的情况，因为 OSTaskDel()不能删除空闲任务，而空闲任务就处于链表的末端 (ptcb->OSTCBNext==0)。接下来，OS_TCB 返回到空闲 OS_TCB 表中，并允许其他任务的建立 [程序清单 4.11(17)]。最后，调用任务调度程序来查看在 OSTaskDel()重新允许中断的时候 [程序清单 4.11(11)]，中断服务子程序是否曾使更高优先级的任务处于就绪状态 [程序清单 4.11(18)]。

4.5 请求删除任务，OSTaskDelReq()

有时候，如果任务 A 拥有内存缓冲区或信号量之类的资源，而任务 B 想删除该任务，这些资源就可能由于没被释放而丢失。在这种情况下，用户可以想法子让拥有这些资源的任务在使用完资源后，先释放资源，再删除自己。用户可以通过 OSTaskDelReq()函数来完成该功能。

发出删除任务请求的任务 (任务 B) 和要删除的任务 (任务 A) 都需要调用 OSTaskDelReq() 函数。任务 B 的代码如程序清单 4.12 所示。任务 B 需要决定在怎样的情况下请求删除任务 [程序清单 4.12(1)]。换句话说，用户的应用程序需要决定在什么样的情况下删除任务。如果任务需要被删除，可以通过传递被删除任务的优先级来调用 OSTaskDelReq() [程序清单 4.12(2)]。如果要被删除的任务不存在 (即任务已被删除或是还没被建立)，OSTaskDelReq() 返回 OS_TASK_NOT_EXIST。如果 OSTaskDelReq() 的返回值为 OS_NO_ERR，则表明请求

译注 1：例如 MC68HC08 指令中的 NOP 指令。

已被接受但任务还没被删除。用户可能希望任务 B 等到任务 A 删除了自己以后才继续进行

通过让任务 B 延时一定时间来达到这个目的[程序
如果需要，用户可以延时得更长一些。当任务 A
返回值成为 OS_TASK_NOT_EXIST，此时循环结

务（任务 B）

```
void RequestTask(void *pdata)
{
    int32 error;
    pdata = pdata;
    for(;;) {
        /* 检测循环代码 */
        if (!taskIsDeleted()) "需要被删除" { (1)
            while (osTaskDelete(OS_TASK_ID_SELF) != OS_TASK_DELETE_ERROR) (2)
                osDelay(1); (3)
        }
        /* 应用程序代码 */
    }
}
```

```
void TaskToDelete(void *pdata)
{
    int32 error;
    pdata = pdata;
    for(;;) {
        /* 检测循环代码 */
        if (OSTaskDelete(OS_TASK_ID_SELF) == OS_TASK_DEL_REQ) (1)
            /* 通知所有占有的资源 */
    }
}
```

程序清单 4.13 需要删除自己的任务（任务 A）

```
释放所有动态内存  
OSTaskDelete(OS_PRIO_SELF);  
else {  
    /* 调用程序代码 */  
}
```



需要删除自己的任务（任务 A）的代码如程序清单 4.13 所示。在 OS_TCB 中存有一个标志，任务通过查询这个标志的值来确认自己是否需要被删除。这个标志的值是通过调用

的。当 OSTaskDelReq() 返回给调用者 OS_TASK_SELF 时，表示已经有另外的任务请求该任务被删除了。在这种情况下，任务会释放所有动态内存[程序清单 4.13(1)]，并且调用 OSTaskDelete(OS_PRIO_SELF)[程序清单 4.13(2)]。前面曾提到过，任务的代码没有被真正执行[程序清单 4.13(3)]。换句话说，就是任务的代码不会再运行了。如果任务的优先级不是 OS_PRIO_SELF，则会调用 OSTaskCreateExt() 函数重新建立该任务。

OSTaskDelReq() 的代码如程序清单 4.14 所示。通常 OSTaskDelReq() 需要检查临界条件。首先，如果正在删除的任务是空闲任务，OSTaskDelReq() 会报错并返回[程序清单 4.14(1)]。接着，它要保证调用者请求删除的任务的优先级是有效的[程序清单 4.14(2)]。如果调用者就是被删除任务本身，存储在 OS_TCB 中的标志将会作为返回值[程序清单 4.14(3)]。如果用户用优先级而不是 OS_PRIO_SELF 指定任务，并且任务是存在的[程序清单 4.14(4)]，OSTaskDelReq() 就会设置任务的内部标志[程序清单 4.14(5)]。如果任务不存在，OSTaskDelReq() 则会返回 OS_TASK_NOT_EXIST，表明任务可能已经删除自己了[程序清单 4.14(6)]。

程序清单 4.14 OSTaskDelReq()

```
INT32 OSTaskDelReq (INT32 prio)  
{  
    OSLOCK();  
    if (prio <= OS_PRIO_SELF) {  
        return OS_TASK_DEL_SELF;  
    }  
    if (prio >= OS_LOWEST_PRIO && prio <= OS_HIGHEST_PRIO) {  
        /*  
         * 释放所有动态内存  
         */  
        OSTaskDelete(OS_PRIO_SELF);  
    }  
}
```

```

picture. OS_PRIO_INVALID);

if (prio == OS_PRIO_SELF) {                                (2)
    OS_ENTER_CRITICAL();
    task = OSTCB[OS_PRIO_SELF];
    OS_EXIT_CRITICAL();
    return (task);
}
else {
    OS_ENTER_CRITICAL();
    if ((prio < OS_PRIO_SELF) || (prio > OS_PRIO_MAX)) { (3)
        p tcb->OSTCBoldPrio = OS_TASK_DFL_PRIO;
        err = OS_NO_PRIV;
    }
    else {
        err = OS_TASK_NOT_EXIST;                            (4)
    }
}
OS_EXIT_CRITICAL();
return (err);
}

```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

4.6 改变任务的优先级，OSTaskChangePrio()

在用户建立任务的时候会分配给任务一个优先级。在程序运行期间，用户可以通过调用 OSTaskChangePrio()来改变任务的优先级。换句话说，就是μC/OS-II 允许用户动态的改变任务的优先级。

OSTaskChangePrio()的代码如程序清单 4.15 所示。用户不能改变空闲任务的优先级[程序清单 4.15(1)]，但用户可以改变调用本函数的任务或者其他任务的优先级。为了改变调用本函数的任务的优先级，用户可以指定该任务当前的优先级或 OS_PRIO_SELF，OSTaskChangePrio()会决定该任务的优先级。用户还必须指定任务的新（即想要的）优先级。因为μC/OS-II 不允许多个任务具有相同的优先级，所以 OSTaskChangePrio()需要检验新优先级是否合法（即不存在具有新优先级的任务）[程序清单 4.15(2)]。如果新优先级是合法的，μC/OS-II 通过将某些东西存储到 OSTCBPrioTbl[newprio]中保留这个优先级[程序清单 4.15(3)]。如此就使得 OSTaskChangePrio()可以重新允许中断，因为此时其他任务已经不可能建立拥有该优先级的任务，也不能通过指定相同的新优先级来调用 OSTaskChangePrio()。接下来 OSTaskChangePrio()可以预先计算新优先级任务的 OS_TCB 中的某些值[程序清单

4.15(4)]。而这些值用来将任务放入就绪表或从该表中移去（参看 3.4 节）。

接着，OSTaskChangePrio()检验目前的任务是否想改变它的优先级[程序清单 4.15(5)]。然后，OSTaskChangePrio()检查想要改变优先级的任务是否存在[程序清单 4.15(6)]。很明显，如果要改变优先级的任务就是当前任务，这个测试就会成功。但是，如果OSTaskChangePrio()想要改变优先级的任务不存在，它必须将保留的新优先级放回到优先级表 OSTCBPrioTbl[]中[程序清单 4.15(17)]，并返回给调用者一个错误码。

现在，OSTaskChangePrio()可以通过插入 NULL 指针将指向当前任务 OS_TCB 的指针从优先级表中移除了[程序清单 4.15(7)]。这就使得当前任务的旧的优先级可以重新使用了。接着，我们检验一下 OSTaskChangePrio()想要改变优先级的任务是否就绪[程序清单 4.15(8)]。如果该任务处于就绪状态，它必须在当前的优先级下从就绪表中移除[程序清单 4.15(9)]，然后在新的优先级下插入到就绪表中[程序清单 4.15(10)]。这儿需要注意的是，

的值[程序清单 4.15(4)]将任务插入就绪表中的。

等待一个信号量、一封邮件或是一个消息队列。如

[程序清单 4.15(8)]，OSTaskChangePrio()就会知道在等待某一事件的发生，OSTaskChangePrio()必须将待队列（在旧的优先级下）中移除。并在新的优先

级下将事件插入到等待队列中[程序清单 4.15(12)]。任务也有可能正在等待延时的期满（参看第 5 章）或是被挂起（参看 4.7）。在这些情况下，从程序清单 4.15(8)到程序清单 4.15(12)这几行可以略过。

接着，OSTaskChangePrio()将指向任务 OS_TCB 的指针存到 OSTCBPrioTbl[]中[程序清单 4.15(13)]。新的优先级被保存在 OS_TCB 中[程序清单 4.15(14)]，重新计算的值也被保存在 OS_TCB 中[程序清单 4.15(15)]。OSTaskChangePrio()完成了关键性的步骤后，在新的优先级高于旧的优先级或新的优先级高于调用本函数的任务的优先级情况下，任务调度程序就会被调用[程序清单 4.15(16)]。

程序清单 4.15 OSTaskChangePrio()

```

if ((oldprio >= OS_LOWEST_PRIO && oldprio != OS_PRIO_SELF) || (1)
    newprio >= OS_LOWEST_PRIO) {
    return (OS_PRIO_INVALID);
}

OS_ENTER_CRITICAL();
if (OSTCBPrioTbl[newprio] != (OS_TCB *)0) { (2)
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
} else {
    OSTCBPrioTbl[newprio] = (OS_TCB *)1; (3)
    OS_EXIT_CRITICAL();
    y = newprio >> 3; (4)
    bity = OSMapTbl[y];
    x = newprio & 0x07;
    bitx = OSMapTbl[x];
    OS_ENTER_CRITICAL();
    if (oldprio == OS_PRIO_SELF) { (5)
        oldprio = OSTCBCur->OSTCBPrio;
    }
    if ((ptcb = OSTCBPrioTbl[oldprio]) != (OS_TCB *)0) { (6)
        OSTCBPrioTbl[oldprio] = (OS_TCB *)0; (7)
        if (OSRdyTbl[ptcb->OSTCBy] & ptcb->OSTCBBitX) { (8)
            if ((OSRdyTbl[ptcb->OSTCBy] & -ptcb->OSTCBBitX) == 0) { (9)
                OSRdyGrp &= -ptcb->OSTCBBitY;
            }
            OSRdyGrp |= bity; (10)
            OSRdyTbl[y] |= bitx;
        } else {
            if ((pevent = ptcb->OSTCBEVENTPTR) != (OS_EVENT *)0) { (11)
                if ((pevent->OSEventTbl[ptcb->OSTCBy] &
                    -ptcb->OSTCBBitX) == 0) {
                    pevent->OSEventGrp &= -ptcb->OSTCBBitY;
                }
                pevent->OSEventGrp |= bity; (12)
                pevent->OSEventTbl[y] |= bitx;
            }
        }
    }
}

```

```

    osTaskSuspend(newprior) = p tcb;
    p tcb->osPriority = newprior;
    p tcb->osCurPrio = p;
    p tcb->osPriority = b prior;
    p tcb->osPriority = b prior;
    os_EXIT_CRITICAL();
    taskend();
    return OS_NO_ERROR;
}
else {
    osTaskSuspend(newprior) = OS_TCB + 1;
    os_EXIT_CRITICAL();
    return OS_PRIO_ERROR;
}

```

超强浏览器提醒您：
使用本复制品
请尊重相关知识产权！

4.7 挂起任务，OSTaskSuspend()

有时候将任务挂起是很有用的。挂起任务可通过调用 OSTaskSuspend()函数来完成。被挂起的任务只能通过调用 OSTaskResume()函数来恢复。任务挂起是一个附加功能。也就是说，如果任务在被挂起的同时也在等待延时的期满，那么，挂起操作需要被取消，而任务继续等待延时期满，并转入就绪状态。任务可以挂起自己或者其他任务。

OSTaskSuspend()函数的代码如程序清单 4.16 所示。通常 OSTaskSuspend()需要检验临界条件。首先，OSTaskSuspend()要确保用户的应用程序不是在挂起空闲任务[程序清单 4.16(1)]，接着确认用户指定优先级是有效的[程序清单 4.16(2)]。记住最大的有效的优先级数（即最低的优先级）是 OS_LOWEST_PRIO。注意，用户可以挂起统计任务（statistic）。可能用户已经注意到了，第一个测试[程序清单 4.16(1)]在[程序清单 4.16(2)]中被重复了。笔者这样做是为了能与μC/OS 兼容。第一个测试能够被移除并可以节省一点程序处理的时间，但是，这样做的意义不大，所以笔者决定留下它。

接着，OSTaskSuspend()检验用户是否通过指定 OS_PRIO_SELF 来挂起调用本函数的任务本身[程序清单 4.16(3)]。用户也可以通过指定优先级来挂起调用本函数的任务[程序清单 4.16(4)]。在这两种情况下，任务调度程序都需要被调用。这就是笔者为什么要定义局部变量 self 的原因，该变量在适当的情况下会被测试。如果用户没有挂起调用本函数的任务，

OSTaskSuspend()就没有必要运行任务调度程序，因为正在挂起的是较低优先级的任务。

然后，OSTaskSuspend()检验要挂起的任务是否存在[程序清单 4.16(5)]。如果该任务存在的话，它就会从就绪表中被移除[程序清单 4.16(6)]。注意要被挂起的任务有可能没有在

发生或延时的期满。在这种情况下，要被挂起的任务（即为 0）。再次清除该位，要比先检验该位是否被清除了，所以笔者没有检验该位而直接清除它。现在，在 B 中设置 OS_STAT_SUSPEND 标志了，以表明任务是通过调用 OSTaskSuspend() 被挂起的。现在，OSTaskSuspend() 只有在被挂起的任务是调用本函数时才返回[程序清单 4.16(8)]。

```
INT8U OSTaskSuspend( INT8U prio)
{
    BOOLEAN self;
    OS_TCB *ptcb;

    if (prio == OS_STAT_PRIO) {
        return OS_TASK_SUSPEND_PRIO;
    }

    if (prio >= OS_LOWEST_PRIO && prio <= OS_HIGHEST_PRIO) {
        return OS_PRIO_INVALID;
    }

    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {
        ptcb = OS_tcbCur->OS tcb;
        self = TRUE;
    } else if (prio == OS tcbCur->OS tcbPriority) {
        self = TRUE;
    } else {
        self = FALSE;
    }

    if ((ptcb = OS tcbGetFirst(prio)) == (OS tcb * 1)) {
        OS_EXIT_CRITICAL();
        return OS_TASK_SUSPEND_PRIO;
    } else {
        if (OS tcbGet(ptcb->OS tcb) <= -ptcb->OS tcbLKL) {
            self = FALSE;
        }
    }
}
```

```

OSByOp 4x->ptcb->OSFCBality;
{
    OS_TCB->OSTCBStat |= OS_STAT_SUSPEND;
    OS_EXIT_CRITICAL();
    LT_TaskL = TRUE;
    OSReset();
}
return (OS_NO_ERROR);
}

```



4.17 挂起任务 OSTaskSuspend()

任务只有通过调用 OSTaskResume()才能恢复。

4.17 所示。因为 OSTaskSuspend()不能挂起空闲任务是在恢复空闲任务[程序清单 4.17(1)]。注意，这个

测试也可以确保用户不是在恢复优先级为 OS_PRIO_SELF 的任务 (OS_PRIO_SELF 被定义为 0xFF, 它总是比 OS_LOWEST_PRIO 大)。

要恢复的任务必须是存在的，因为用户需要操作它的任务控制块 OS_TCB[程序清单 4.17(2)]，并且该任务必须是被挂起的[程序清单 4.17(3)]。OSTaskResume()是通过清除 OSTCBStat 域中的 OS_STAT_SUSPEND 位来取消挂起的[程序清单 4.17(4)]。要使任务处于就绪状态，OS_TCBDly 域必须为 0[程序清单 4.17(5)]，这是因为在 OSTCBStat 中没有任何标志表明任务正在等待延时的期满。只有当以上两个条件都满足的时候，任务才处于就绪状态[程序清单 4.17(6)]。最后，任务调度程序会检查被恢复的任务拥有的优先级是否比调用本函数的任务的优先级高[程序清单 4.17(7)]。

程序清单 4.17 OSTaskResume()

```

OS_EXIT_CRITICAL();
secure(OS_TASK_NOTREADY);
}
else {
    if (ptcb->OSTCBstat & OS_STAT_BLOCKED) {
        if ((ptcb->OSTCBstat & ~OS_STAT_SUSPENDED) == OS_STAT_BLOCKED && !4) {
            if (ptcb->OSTCBstat & 8) {
                if (ptcb->OSTCBstat & ~OS_BLOCKED) {
                    osmyth(ptcb->OSTCBstat) |= ptcb->OSTCBstat;
                    OS_EXIT_CRITICAL();
                    break;
                }
            }
            else {
                OS_EXIT_CRITICAL();
            }
        }
        return (OS_NO_ERR);
    }
    else {
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_BLOCKED);
    }
}

```

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

```

OS_TCB MyTaskData;
void MyTask(void *pdata)
{
    /* Task code */
}

```

4.9 获得有关任务的信息，OSTaskQuery()

用户的应用程序可以通过调用 OSTaskQuery()来获得自身或其他应用任务的信息。实际上，OSTaskQuery()获得的是对应任务的 OS_TCB 中内容的拷贝。用户能访问的 OS_TCB 的数据域的多少决定于用户的应用程序的配置（参看 OS_CFG.H）。由于μC/OS-II 是可裁剪的，它只包括那些用户的应用程序所要求的属性和功能。

要调用 OSTaskQuery()，如程序清单 4.18 中所示的那样，用户的应用程序必须要为 OS_TCB 分配存储空间。这个 OS_TCB 与 μC/OS-II 分配的 OS_TCB 是完全不同的数据空间。在调用了 OSTaskQuery()后，这个 OS_TCB 包含了对应任务的 OS_TCB 的副本。用户必须十分小心地处理 OS_TCB 中指向其他 OS_TCB 的指针（即 OSTCBNext 与 OSTCBPrev）；用户不要试图去改变这些指针！一般来说，本函数只用来了解任务正在干什么——本函数是有用的调试工具。

程序清单 4.18 得到任务的信息

```
pdata = ptask;
for (i=0; i<=1; i++) {
    /* 用户代码 */
    user = OSTaskQuery(i, myTaskDetail);
    /* 调用检测代码 */
    /* 用户代码 */
}
```

```
INT32 OSTaskQuery(INT32 prio, OS_TCB *pTask)
{
    OS_TCB *ptask;
    if (prio > OS_LOWEST_PRIO || prio <= OS_HIGHEST_PRIO) {           (1)
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();          (2)
    if (prio == OS_PRIO_BLOCK) {
        prio = OS_PRIO_BLOCKED;
    }
    if ((ptask = OSTCBFirst((prio))) == (OS_TCB *)1) {                   (3)
        OS_EXIT_CRITICAL();
        return (OS_PRIO_BLOCK);
    }
    *pTask = *ptask;
    OS_EXIT_CRITICAL();          (4)
    return (OS_NO_ERROR);
}
```



9 所示。注意，笔者允许用户查询所有的任务，包
需要注意的是不要改变 OSTCBNext 与 OSTCBPrev
用户是否想知道当前任务的有关信息[程序清单
单 4.19(3)]。所有的域是通过赋值语句一次性复制
单 4.19(4)]。这样复制会比较快一点，因为编译器

大多都能够产生内存拷贝指令。

程序清单 4.19 OSTaskQuery()

第 5 章

时间管理



在 3.10 节中曾提到, μC/OS-II (其他内核也一样) 要求用户提供定时中断来实现延时与超时控制等功能。这个定时中断叫做时钟节拍, 它应该每秒发生 10 至 100 次。时钟节拍的实际频率是由用户的应用程序决定的。时钟节拍的频率越高, 系统的负荷就越重。

3.10 节讨论了时钟的中断服务子程序和时钟节拍函数 OSTimeTick——该函数用于通知 μC/OS-II 发生了时钟节拍中断。本章主要讲述五个与时钟节拍有关的系统服务:

- OSTimeDly()
- OSTimeDlyHMSM()
- OSTimeDlyResume()
- OSTimeGet()
- OSTimeSet()

本章所提到的函数可以在 OS_TIME.C 文件中找到。

5.0 任务延时函数, OSTimeDly()

μC/OS-II 提供了这样一个系统服务: 申请该服务的任务可以延时一段时间, 这段时间的长短是用时钟节拍的数目来确定的。实现这个系统服务的函数叫做 OSTimeDly()。调用该函数会使 μC/OS-II 进行一次任务调度, 并且执行下一个优先级最高的就绪态任务。任务调用 OSTimeDly() 后, 一旦规定的时间期满或者有其他的任务通过调用 OSTimeDlyResume() 取消了延时, 它就会马上进入就绪状态。注意, 只有当该任务在所有就绪任务中具有最高的优先级时, 它才会立即运行。

程序清单 5.1 所示的是任务延时函数 OSTimeDly() 的代码。用户的应用程序是通过提供延时的时钟节拍数——一个 1 到 65535 之间的数, 来调用该函数的。如果用户指定 0 值 [程序清单 5.1(1)], 则表明用户不想延时任务, 函数会立即返回到调用者。非 0 值会使得任务延时函数 OSTimeDly() 将当前任务从就绪表中移除 [程序清单 5.1(2)]。接着, 这个延时节拍数会被保存在当前任务的 OS_TCB 中 [程序清单 5.1(3)], 并且通过 OSTimeTick() 每隔一个时钟节拍就减少一个延时节拍数。最后, 既然任务已经不再处于就绪状态, 任务调度程序会

```

void OSTimeDly( uint32 ticks)
{
    if(ticks > 0) {
        OS_ENTER_CRITICAL(); // (1)
        if((OS_TICKS16(OS_TICKS16 - OSTICKY) & OS_TICKS16(X)) == 0) { // (2)
            OS调度 A = <OSTICKY> - OSTICKY;
            OS_WRITE_CRITICAL(); // (3)
            OSTimeDly(1); // (4)
        }
    }
}

```



清楚地认识 0 到一个节拍之间的延时过程是非常重要的。换句话说，如果用户只想延时一个时钟节拍，而实际上是在 0 到一个节拍之间结束延时。即使用户的处理器的负荷不是很重，这种情况依然存在的。图 5.1 详细说明了整个过程。系统每隔 10ms 发生一次时钟节拍中断[图 5.1(1)]。假如用户没有执行其他的中断并且此时的中断是开着的，时钟节拍中断服务就会发生[图 5.1(2)]。也许用户有好几个高优先级的任务（HPT）在等待延时期满，它们会接着执行[图 5.1(3)]。接下来，图 5.1 中所示的低优先级任务（LPT）会得到执行的机会，该任务在执行完后马上调用[图 5.1(4)]所示的 OSTimeDly(1)。 μ C/OS-II 会使该任务处于休眠状态直至下一个节拍的到来。当下一个节拍到来后，时钟节拍中断服务子程序会执行[图 5.1(5)]，但是这一次由于没有高优先级的任务被执行， μ C/OS-II 会立即执行申请延时一个时钟节拍的

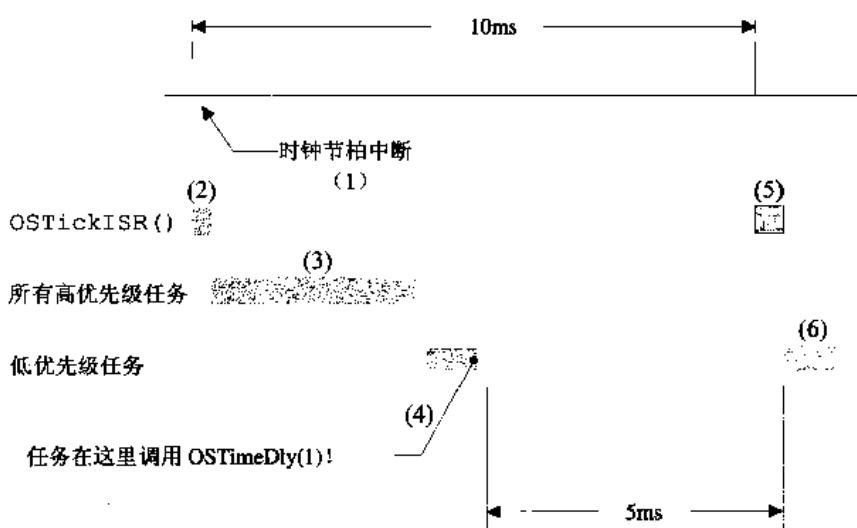


图 5.1 延时详解

5.1(1)]。假如用户没有执行其他的中断并且此时的中断是开着的，时钟节拍中断服务就会发生[图 5.1(2)]。也许用户有好几个高优先级的任务（HPT）在等待延时期满，它们会接着执行[图 5.1(3)]。接下来，图 5.1 中所示的低优先级任务（LPT）会得到执行的机会，该任务在执行完后马上调用[图 5.1(4)]所示的 OSTimeDly(1)。 μ C/OS-II 会使该任务处于休眠状态直至下一个节拍的到来。当下一个节拍到来后，时钟节拍中断服务子程序会执行[图 5.1(5)]，但是这一次由于没有高优先级的任务被执行， μ C/OS-II 会立即执行申请延时一个时钟节拍的

任务[图 5.1(6)]。正如用户所看到的，该任务实际的延时少于一个节拍！在负荷很重的系统中，任务甚至有可能会在时钟中断即将发生时调用 OSTimeDly(1)，在这种情况下，任务几乎没有得到任何延时，因为任务马上又被重新调度了。如果用户的[应用程序至少得延时一个节拍，必须要调用 OSTimeDly\(2\)](#)，指定延时两个节拍！

使用本章相关知识
请尊重相关知识

5.1 按时分秒延时函数 OSTimeDlyHMSM()

OSTimeDly()虽然是一個非常有用的函數，但用户的應用程序需要知道延时时间对应的时钟节拍的数目。用户可以使用定义全局常数 OS_TICKS_PER_SEC（参看 OS_CFG.H）的方法将时间转换成时钟段，但这种方法有时显得比较愚笨。笔者增加了 OSTimeDlyHMSM() 函数后，用户就可以按小时(h)、分(in)、秒(s)和毫秒(ms)来定义时间了，这样会显得更自然些。与 OSTimeDly()一样，调用 OSTimeDlyHMSM() 函数也会使μC/OS-II 进行一次任务调度，并且执行下一个优先级最高的就绪态任务。任务调用 OSTimeDlyHMSM() 后，一旦规定的时间期满或者有其他的任务通过调用 OSTimeDlyResume() 取消了延时（参看 5.2），它就会马上处于就绪态。同样，只有当该任务在所有就绪态任务中具有最高的优先级时，它才会立即运行。

程序清单 5.2 所示的是 OSTimeDlyHMSM() 的代码。从中可以看出，应用程序是通过用小时、分、秒和毫秒指定延时来调用该函数的。在实际应用中，用户应避免使任务延时过长的时间，因为从任务中获得一些反馈行为（如减少计数器，清除 LED 等等）经常是很不错的事。但是，如果用户确实需要延时长时间的话，μC/OS-II 可以将任务延时长达 256 个小时（接近 11 天）。

OSTimeDlyHMSM()一开始先要检验用户是否为参数定义了有效的值[程序清单 5.2(1)]。与 OSTimeDly()一样，即使用户没有定义延时，OSTimeDlyHMSM()也是存在的[程序清单 5.2(9)]。因为μC/OS-II 只知道节拍，所以节拍总数是从指定的时间中计算出来的[程序清单 5.2(3)]。很明显，程序清单 5.2 中的程序并不是十分有效的。笔者只是用这种方法告诉大家一个公式，这样用户就可以知道怎样计算总的节拍数了。真正有意义的只是 OS_TICKS_PER_SEC。[程序清单 5.2(3)]决定了最接近需要延迟的时间的时钟节拍总数。500/OS_TICKS_PER_SECOND 的值基本上与 0.5 个节拍对应的毫秒数相同。例如，若将时钟频率 (OS_TICKS_PER_SEC) 设置成 100Hz(10ms)，4ms 的延时不会产生任何延时！而 5ms 的延时就等于延时 10ms。

μC/OS-II 支持的延时最长为 65535 个节拍。要想支持更长时间的延时，如程序清单 5.2(2) 所示，OSTimeDlyHMSM() 确定了用户想延时多少次超过 65535 个节拍的数目[程序清单 5.2(4)]和剩下的节拍数[程序清单 5.2(5)]。例如，若 OS_TICKS_PER_SEC 的值为 100，用户想延时 15 分钟，则 OSTimeDlyHMSM() 会延时 $15 \times 60 \times 100 = 90000$ 个时钟。这个延时会被分割成两次 32768 个节拍的延时（因为用户只能延时 65535 个节拍而不是 65536 个节拍）和

```

INT32U OPTIMEDLYHMSM(INT32U hours, INT32U minutes, INT32U seconds, INT32U milli)
{
    INT32U ticks;
    INT32U loops;

    if (hours < 0 || minutes < 0 || seconds > 59 || milli > 999) {
        if (minutes + 59) {
            return (OS_TIME_INVALID_MINUTES);
        }
        if (seconds == 59) {
            return (OS_TIME_INVALID_SECONDS);
        }
        if (milli > 999) {
            return (OS_TIME_INVALID_MILLI);
        }
        ticks = (INT32U)hours * 3600L + OS_TICKS_PER_SEC
            + (INT32U)minutes * 60L * OS_TICKS_PER_SEC
            + (INT32U)seconds * OS_TICKS_PER_SEC
            + OS_TICKS_PER_SEC * ((INT32U)milli
            + 500L) / OS_TICKS_PER_SEC / 1000L;
        loops = ticks / 65536L;
        ticks = ticks % 65536L;
        OPTIMEDLY(ticks);
        while (loops > 0) {
            OPTIMEDLY(32768);
            OPTIMEDLY(32768);
            loops--;
        }
        return (OS_TIME_ZERO);
    }
    else {
        return (OS_TIME_ZERO);
    }
}

```

，OSTimeDlyHMSM()首先考虑剩下的节拍，然后[8]（即两个32768个节拍延时）。

超星浏览器提醒您：
 使用本复制品
 请尊重相关知识版权！

由于 OSTimeDlyHMSM()的具体实现方法，用户不能结束延时调用 OSTimeDlyHMSM()要求延时超过 65535 个节拍的任务。换句话说，如果时钟节拍的频率是 100Hz，用户不能让调用 OSTimeDlyHMSM(0, 10, 55, 350)或更长延迟时间的任务结束延时。

5.2 让处在延时期的任务结束延时，OSTimeDlyResume()

μ C/OS-II 允许用户结束延时正处于延时期的任务。延时的任务可以不等待延时期满，而是通过其他任务取消延时来使自己处于就绪态。这可以通过调用 OSTimeDlyResume()和指定要恢复的任务的优先级来完成。实际上，OSTimeDlyResume()也可以唤醒正在等待事
并没有提到过。在这种情况下，等待事件发生的任

单 5.3 所示，它首先要确保指定的任务优先级有效。OSTimeDlyResume()要确认要结束延时的任务是确实存在的[程序清单 5.3(1)]。OSTimeDlyResume()会检验任务是否在等待延时期满[程序清单 5.3(2)]。OSTimeDly 包含非 0 值就表明任务正在等待延时期满，因为任务调用了 OSTimeDly()，OSTimeDlyHMSM()或其他在第六章中所描述的 PEND 函数。

然后延时就可以通过强制命令 OSTCBDly 为 0 来取消延时[程序清单 5.3(4)]。延时的任务有可能已被挂起了，这样的话，任务只有在没有被挂起的情况下才能处于就绪状态[程序清单 5.3(5)]。当上面的条件都满足后，任务就会被放在就绪表中[程序清单 5.3(6)]。这时，OSTimeDlyResume()会调用任务调度程序来看被恢复的任务是否拥有比当前任务更高的优先级[程序清单 5.3(7)]。这会导致任务的切换。

程序清单 5.3 恢复正在延时的任务

```

17. if (ptcb->osState & OS_STATE_SUSPENDED) {
18.     osDlyGrp        |= p tcb->osPriority;
19.     osDlyTsl(ptcb->osCbx) |= p tcb->osPriority;
20.     osExit_critical();
21. } else {
22.     os_exit_critical();
23. }
24. return IOE_NO_BLOCK;
25. } else {
26.     osDlyTsl(ptcb->osCbx);
27.     return IOE_BLOCK_WAIT;
28. }
29. else {
30.     os_exit_critical();
31.     return IOE_TASK_NOT_EXIST;
32. }

```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

INT10H OSTimeGet void

注意，用户的任务有可能是通过暂时等待信号量、邮箱或消息队列来延时自己的（参看第 6 章）。可以简单地通过控制信号量、邮箱或消息队列来恢复这样的任务。这种情况存在的问题是它要求用户分配事件控制块（参看 6.0 节），因此用户的应用程序会多占用一些 RAM。

5.2 系统时间，OSTimeGet()和OSTimeSet()

无论时钟节拍何时发生，μC/OS-II 都会将一个 32 位的计数器加 1。这个计数器在用户调用 OSStart() 初始化多任务和 4 294 967 295 个节拍执行完一遍的时候从 0 开始计数。在时钟节拍的频率等于 100Hz 的时候，这个 32 位的计数器每隔 497 天就重新开始计数。用户可以通过调用 OSTimeGet() 来获得该计数器的当前值。也可以通过调用 OSTimeSet() 来改变该计数器的值。OSTimeGet() 和 OSTimeSet() 两个函数的代码如程序清单 5.4 所示。注意，在访问 OSTime 的时候中断是关掉的。这是因为在大多数 8 位处理器上增加和拷贝一个 32 位的数都需要数条指令，这些指令一般都需要一次执行完毕，而不能被中断打断。

程序清单 5.4 得到和改变系统时间

```

{
    INT32U ticks;

    OS_ENTER_CRITICAL();
    ticks = OSTime;
    OS_EXIT_CRITICAL();
    return (ticks);
}

void OSTimeSet (INT32U ticks)
{
    OS_ENTER_CRITICAL();
    OSTime = ticks;
    OS_EXIT_CRITICAL();
}

```

第6章

超星浏览器提醒您：
 使用本复制品
 请尊重相关知识产权！

任务之间的通信与同步

在μC/OS-II 中，有多种方法可以保护任务之间的共享数据和提供任务之间的通信。在前面的章节中，已经讲到了其中的两种：

一是利用宏 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 来关闭中断和打开中断。当两个任务或者一个任务和一个中断服务子程序共享某些数据时，可以采用这种方法，详见 3.0 节、8.3.2 节及 9.3.2 节。

二是利用函数 OSSchedLock() 和 OSSchedUnlock() 对 μC/OS-II 中的任务调度函数上锁和开锁。用这种方法也可以实现数据的共享，详见 3.6 节。

本章将介绍另外三种用于数据共享和任务通信的方法：信号量、邮箱和消息队列。

图 6.1 介绍了任务和中断服务子程序之间是如何进行通信的。

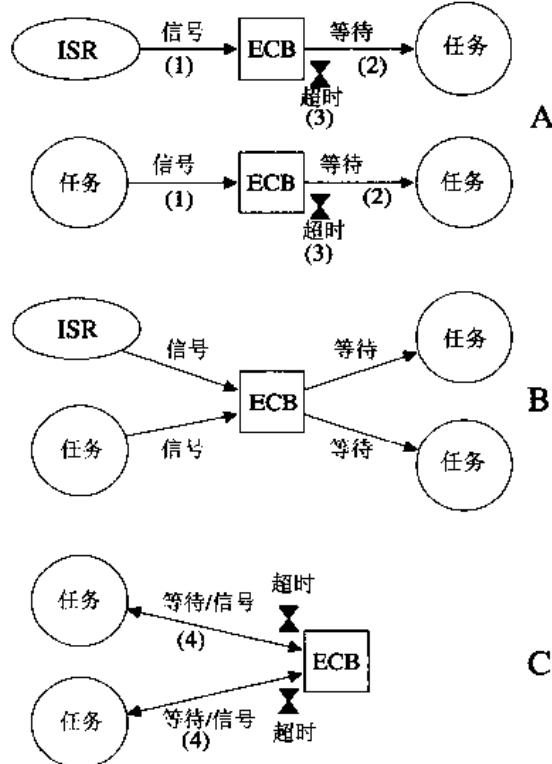


图 6.1 事件控制块的使用

一个任务或者中断服务子程序可以通过事件控制块 (ECB, Event Control Block) 来向另外的任务发信号[图 6.1A(1)]。这里，所有的信号都被看成是事件 (event)。这也说明为什么上面把用于通信的数据结构叫做事件控制块。一个任务还可以等待另一个任务或中断服务子程序给它发送信号[图 6.1A(2)]。这里要注意的是，只有任务可以等待事件发生，中断服务子程序是不能这样做的。对于处于等待状态的任务，还可以给它指定一个最长等待时间，以此来防止因为等待的事件没有发生而无限期地等下去。

多个任务可以同时等待同一个事件的发生[图 6.1B]。在这种情况下，当该事件发生后，

所有等待该事件的任务得到了该事件并进入就绪状态，准备执行。

或者消息队列等。当事件控制块是一个信号量时，

```
typedef struct {
    void *OSEventPtr;           /* 指向消息邮箱或消息队列的指针 */
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; /* 等待任务列表 */
    INT16U OSEventCnt;          /* 计数器(当事件是信号量时) */
    INT8U OSEventType;          /* 时间类型 */
    INT8U OSEventGrp;           /* 等待任务所在组 */
} OS_EVENT;
```

6.0 事件控制块 ECB

μ C/OS-II 通过 μ COS-II.H 中定义的 OS_EVENT 数据结构来维护一个事件控制块的所有信息（程序清单 6.1），也就是本章开篇讲到的事件控制块 ECB。该结构中除了包含了事件本身的定义，如用于信号量的计数器，用于指向邮箱的指针，以及指向消息队列的指针数组等，还定义了等待该事件的所有任务的列表。程序清单 6.1 是该数据结构的定义。

程序清单 6.1 ECB 数据结构

.OSEventPtr 指针，只有在所定义的事件是邮箱或者消息队列时才使用。当所定义的事件是邮箱时，它指向一个消息，而当所定义的事件是消息队列时，它指向一个数据结构，详见 6.6 节和 6.7 节。

.OSEventTbl[] 和 .OSEventGrp 很像前面讲到的 OSRdyTbl[] 和 OSRdyGrp，只不过前两者包含的是等待某事件的任务，而后两者包含的是系统中处于就绪状态的任务（见 3.4 节）。

.OSEventCnt 当事件是一个信号量时，.OSEventCnt 是用于信号量的计数器（见 6.5 节）。

.OSEventType 定义了事件的具体类型。它可以是信号量 (OS_EVENT_SEM)、邮箱

(OS_EVENT_TYPE_MBOX) 或消息队列 (OS_EVENT_TYPE_Q) 中的一种。用户要根据该域的具体值来调用相应的系统函数，以保证对其进行的操作的正确性。

每个等待事件发生的任务都被加入到该事件事件控制块中的等待任务列表中，该列表包括.OSEventGrp 和.OSEventTbl[]两个域。变量前面的[]说明该变量是数据结构的一个域。在这里，所有的任务的优先级被分成 8 组（每组 8 个优先级），分别对应.OSEventGrp 中的 8 位。当某组中有任务处于等待该事件的状态时，.OSEventGrp 中对应的位就被置位。相应地，该任务在.OSEventTbl[]中的对应位也被置位。.OSEventTbl[]数组的大小由系统中任务的最低优先级决定，这个值由 uCOS_II.H 中的 OS_LOWEST_PRIO 常数定义。这样，可以在任务优先级比较少的情况下，减少μC/OS-II 对系统 RAM 的占用量。

当一个事件发生后，该事件的等待事件列表中优先级最高的任务，也即在.OSEventTbl[]中，所有被置 1 的位中，优先级代码最小的任务得到该事件。图 6.2 给出了.OSEventGrp 和.OSEventTbl[]之间的对应关系。该关系可以描述为：

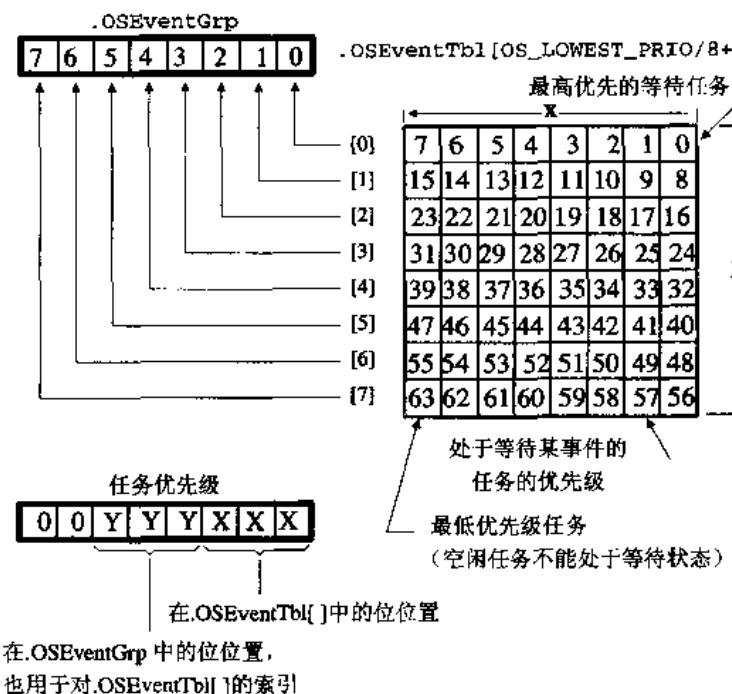


图6.2 事件的等待任务列表

- 当.OSEventTbl[0]中的任何一位为 1 时，.OSEventGrp 中的第 0 位为 1。
- 当.OSEventTbl[1]中的任何一位为 1 时，.OSEventGrp 中的第 1 位为 1。
- 当.OSEventTbl[2]中的任何一位为 1 时，.OSEventGrp 中的第 2 位为 1。
- 当.OSEventTbl[3]中的任何一位为 1 时，.OSEventGrp 中的第 3 位为 1。
- 当.OSEventTbl[4]中的任何一位为 1 时，.OSEventGrp 中的第 4 位为 1。
- 当.OSEventTbl[5]中的任何一位为 1 时，.OSEventGrp 中的第 5 位为 1。
- 当.OSEventTbl[6]中的任何一位为 1 时，.OSEventGrp 中的第 6 位为 1。

```
parent->OSEventGrp    |= OSMapTbl[prio >> 3];
parent->OSEventTbl[prio >> 3] |= OSMapTbl[prio & 0x3];
```

时，.OSEventGrp 中的第 7 位为 1。

下面的代码将一个任务放到事件的等待任务列表中。

程序清单 6.2 将一个任务插入到事件的等待任务列表中

索引	控制码（二进制）
0	0000001
1	0000010
2	0000100
3	0000110
4	0001000
5	0010000
6	0100000
7	1000000

是指向事件控制块的指针。

↑任务到等待任务列表中所花的时间是相同的，和系统中现有多少个任务无关。从图 6.2 中可以看出该算法的原理：任务优先级的最低 3 位决定了该任务在相应的.OSEventTbl[] 中的位置，紧接着的 3 位则决定了该任务优先级

↑用到的查找表 OSMapTbl[]（定义在 OS_CORE.C

↑从任务到事件的映射表 OSUnMapTbl[] 定义在 OS_CORE.C 中。

表 6.1

OSMapTbl[]

从等待任务列表中删除一个任务的算法则正好相反，如程序清单 6.3 所示。

程序清单 6.3 从等待任务列表中删除一个任务

该代码清除了任务在.OSEventTbl[] 中的相应位，并且，如果其所在的组中不再有处于等待该事件的任务时（即.OSEventTbl[prio>>3] 为 0），将.OSEventGrp 中的相应位也清除了。和上面的由任务优先级确定该任务在等待表中的位置的算法类似，从等待任务列表中查找处于等待状态的最高优先级任务的算法，也不是从.OSEventTbl[0] 开始逐个查询，而是采用了查找另一个表 OSUnMapTbl[256]（见文件 OS_CORE.C）。这里，用于索引的 8 位分别代

表对应的 8 组中有任务处于等待状态，其中的最低位具有最高的优先级。用这个值索引，
[OSEventTbl[i].OSEventGrp] (0~7 之间的一个数)。然后利用.OSEventTbl[] 中
可以得到最高优先级任务在组中的位置 (也是 0~
7 之间的一个数)。这样，最终就可以查到处于等待该事件状态的最高优先级任务了。程序
清单 6.4 是该算法的具体实现代码。

程序清单 6.4 在等待任务列表中查找最高优先级的任务

举例来说，如果.OSEventGrp 的值是 01101000 (二进制)，而对应的 OSUnMapTbl [OSEventGrp] 值为 3，说明最高优先级任务所在的组是 3。类似地，如果.OSEventTbl[3] 的值是 11100100 (二进制)，OSUnMapTbl [OSEventTbl[3]] 的值为 2，则处于等待状态的任务的最高优先级是 $3 \times 8 + 2 = 26$ 。

在 μC/OS-II 中，事件控制块的总数由用户所需要的信号量、邮箱和消息队列的总数决定。该值由 OS_CFG.H 中的#define OS_MAX_EVENTS 定义。在调用 OSInit() 时 (见 3.11 节)，所有事件控制块被链接成一个单向链表——空闲事件控制块链表 (图 6.3)。每当建立一个信号量、邮箱或者消息队列时，就从该链表中取出一个空闲事件控制块，并对它进行初始化。因为信号量、邮箱和消息队列一旦建立就不能删除，所以事件控制块也不能放回到空闲事件控制块链表中。

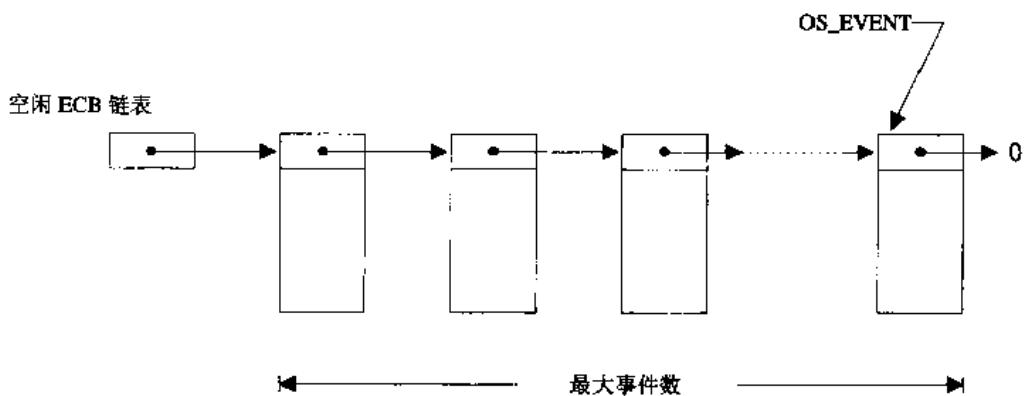


图6.3 空闲事件控制块链表

对于事件控制块进行的一些通用操作包括：

- 初始化一个事件控制块；
- 使一个任务进入就绪态；
- 使一个任务进入等待该事件的状态；

- 因为等待超时而使一个任务进入就绪态。

为了避免代码重复和缩短程代码长度, μC/OS-II 将上面的操作用 4 个系统函数实现, 它们是: OSEventWaitListInit(), OSEventTaskRdy(), OSEventWait() 和 OSEventTO()。

6.1.1 初始化 ECB 块的等待任务列表, OSEventWaitListInit()

 使用本复印件须经相关知识知识产权人授权!

```
void OSEventWaitListInit( OS_EVENT *pevent ) {
```

```
    OS_EVENT_BLOCK *pblock;
```

```
    pevent->OSEventGrp = 0x00;
```

```
    for ( iL = 0; iL < OS_EVENT_TASK_LIST; iL++ ) {
```

```
        pevent->OSEventList[iL] = 0x00;
```

istInit()的源代码。当建立一个信号量、邮箱或者消息队列时, OSSemCreate(), OSMboxCreate(), 或者 OSQCreate()通过调用该函数对等待任务列表进行初始化。该函数初始化一个空的等待任务列表, 其中没有任何任务。该函数的调用参数只有一个, 就是指向需要初始化的事件控制块的指针 pevent。

程序清单 6.5 初始 ECB 块的等待任务列表

6.2 使一个任务进入就绪态, OSEventTaskRdy()

程序清单 6.6 是函数 OSEventTaskRdy()的源代码。当发生了某个事件, 该事件等待任务列表中的最高优先级任务 (Highest Priority Task, HPT) 要置于就绪态时, 该事件对应的 OSSemPost(), OSMboxPost(), OSQPost(), 和 OSQPostFront()函数调用 OSEventTaskRdy()实现该操作。换句话说, 该函数从等待任务队列中删除 HPT 任务, 并把该任务置于就绪态。图 6.4 给出了 OSEventTaskRdy()函数最开始的 4 个动作。

该函数首先计算 HPT 任务在.OSEventTbl[]中的字节索引[程序清单 6.6/ 图 6.4(1)], 其结果是一个从 0 到 OS_LOWEST_PRIO/8+1 之间的数, 并利用该索引得到该优先级任务在.OSEventGrp 中的位屏蔽码[程序清单 6.6/ 图 6.4(2)] (从表 6.1 可以得到该值)。然后, OSEventTaskRdy()函数判断 HPT 任务在.OSEventTbl[]中相应位的位置[程序清单 6.6/ 图

6.4(3)], 其结果是一个从 0 到 OS_LOWEST_PRIO/8+1 之间的数, 以及相应的位屏蔽码[程序清单 6.6/ 图 6.4(4)]。根据以上结果, OSEventTaskRdy()函数计算出 HPT 任务的优先级[程序清单 6.6(5)], 然后就可以从等待任务列表中删除该任务了[程序清单 6.6(6)]。

任务的任务控制块中包含有需要改变的信息。知道了 HPT 任务的优先级, 就可以得到指向该任务的任务控制块的指针[程序清单 6.6(7)]。因为最高优先级任务运行条件已经得到满足, 必须停止 OSTimeTick()函数对.OSTCBDly 域的递减操作, 所以 OSEventTaskRdy()直

接修改该域 or 程序清单 6.6(7) 中任务不再等待该事件的发生, 所以 OSEventTaskRdy()块的指针指向 NULL[程序清单 6.6(9)]。如果当 OSQPost()调用的, 该函数还要将相应的消息传
清单 6.6(10)]。另外, 当 OSEventTaskRdy()被调用
{参数是用于对任务控制块中的位清零的位屏蔽码,
5(11)]。最后, 根据.OSTCBStat 判断该任务是否已
, 则将 HPT 插入到μC/OS-II 的就绪任务列表中[程
事件后不一定进入就绪状态, 也许该任务已经由于

于中断禁止的情况下调用。

程序清单 6.6 使一个任务进入就绪状态

```
void OSEventTaskRdy( os_event_t *pevent, void *msg, OSRSN rsn)
{
    OS_TCB *p_tcb;
    DRTED *t;
    DRTED *y;
    DRTED *x;
    DRTED *bitx;
    DRTED *bity;
    DRTED *prior;

    y = (DRTED *) (pevent->osEventDly);
    bity = (DRTED *) (y+1);
    x = (DRTED *) (pevent->osEventTsl);
    bitx = (DRTED *) (x+1);
    prior = (DRTED *) (x+2);
    if (pevent->osEventTsl[y].pe->bitx == 0) {
        p_tcb = (OS_TCB *) (pevent->osEventGrp & ~0x0f);
        p_tcb->osEventGrp |= 0x01;
    }
}
```

```

ptcb->OSTCBFile=0;
ptcb->OSTCBIdx = 0;
ptcb->OSTCBIndex[0] = OSB_INDEX * 10;
bit (OSB_INDEX & OSB_MAX_INDEX == 2) || OSB_INDEX == 0;
ptcb->OSTCBIndex = msg;
else
    msg = msg;
#endif
ptcb->OSTCBIndex += msg;
if (ptcb->OSTCBIndex == OSB_INDEX_HOL) {
    OSBIndex = 1 + bity;
    OSBIndex1 = 1 + bitx;
}
}

```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

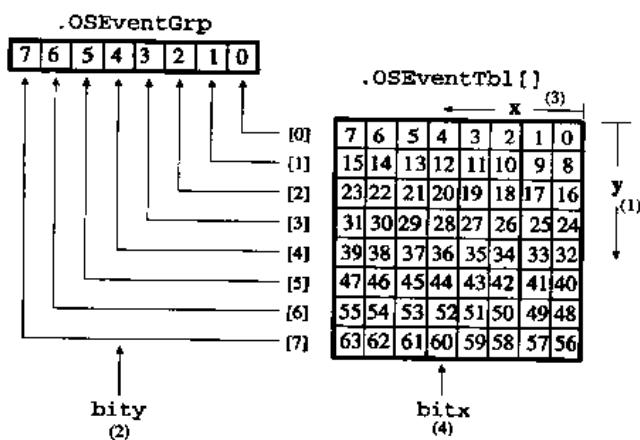


图 6.4 使一个任务进入就绪状态

6.3 使一个任务进入等待某事件发生状态, OSEventTaskWait()

程序清单 6.7 是 OSEventTaskWait()函数的源代码。当某个任务要等待一个事件的发生时, 相应事件的 OSSemPend(), OSMboxPend()或者 OSQPend()函数会调用该函数将当前任务从就绪任务表中删除, 并放到相应事件的事件控制块的等待任务表中。

```
void OSEventTaskWait(10_Event *parent)
{
    OSTCB *OSTCBNow=PLC->parent;
    if ((OSTCBNow->OSTCBEvent->OSEVENT) & ~OSTCBNow->OSTCBEvent->OSEVENT) == 0 {
        OSTCBNow->OSEVENT=0;
        parent->OSEVENT=0;
        parent->OSEVENTS=0;
    }
}
```

鸿公开的实时嵌入式操作系统

态



```
void OSEventTO(10_Event *parent)
{
    if (parent->OSEVENTS & OSEVENT->OSEVENT) && ~OSTCBNow->OSTCBEvent->OSEVENT) == 0 {
        parent->OSEVENTS=0;
        OSEVENT->OSEVENT=0;
        OSEVENT->OSEVENTS=1;
    }
}
```

|块的指针放到该任务的任务控制块中[程序清单 [程序清单 6.7(2)]，并把该任务放到事件控制块的

6.4 由于等待超时而将任务置为就绪态, OSEventTO()

程序清单 6.8 是 OSEventTO()函数的源代码。当在预先指定的时间内任务等待的事件没有发生时, OSTimeTick()函数会因为等待超时而将任务的状态置为就绪。在这种情况下, 事件的 OSSemPend(), OSMboxPend()或者 OSQPend()函数会调用 OSEventTO()来完成这项工作。该函数负责从事件控制块中的等待任务列表里将任务删除[程序清单 6.8(1)], 并把它置成就绪状态[程序清单 6.8(2)]。最后, 从该任务的任务控制块中将指向事件控制块的指针删除[程序清单 6.8(3)]。用户应当注意, 调用 OSEventTO()也应当先关中断。

程序清单 6.8 因为等待超时将任务置为就绪状态

6.5 信号量

μ C/OS-II 中的信号量由两部分组成：一个是信号量的计数值，它是一个 16 位的无符号整数（0 到 65535 之间）；另一个是由等待该信号量的任务组成的等待任务表。用户要在 OS_CFG.H 中将 OS_SEM_EN 开关量常数置成 1，这样 μ C/OS-II 才能支持信号量。

在使用一个信号量之前，首先要建立该信号量，也即调用 OSSemCreate() 函数（见 6.5.1 节），对信号量的初始计数值赋值。该初始值为 0 到 65535 之间的一个数。如果信号量是用来表示一个或者多个事件的发生，那么该信号量的初始值应设为 0。如果信号量是用于对共享资源的访问，那么该信号量的初始值应设为 1（例如，把它当作二值信号量使用）。最后，如果该信号量是用来表示允许任务访问 n 个相同的资源，那么该初始值显然应该是 n，并把该信号量作为一个可计数的信号量使用。

μ C/OS-II 提供了 5 个对信号量进行操作的函数。它们是：OSSemCreate(), OSSemPend(), OSSemPost(), OSSemAccept() 和 OSSemQuery() 函数。图 6.5 说明了任务、中断服务子程序和信号量之间的关系。图中用钥匙或者旗帜的符号来表示信号量：如果信号量用于对共享资源的访问，那么信号量就用钥匙符号。符号旁边的数字 N 代表可用资源数。对于二值信号量，该值就是 1；如果信号量用于表示某事件的发生，那么就用旗帜符号。这时的数字 N 代表事件已经发生的次数。从图 6.5 中可以看出 OSSemPost() 函数可以由任务或者中断服务子程序调用，而 OSSemPend() 和 OSSemQuery() 函数只能有任务程序调用。

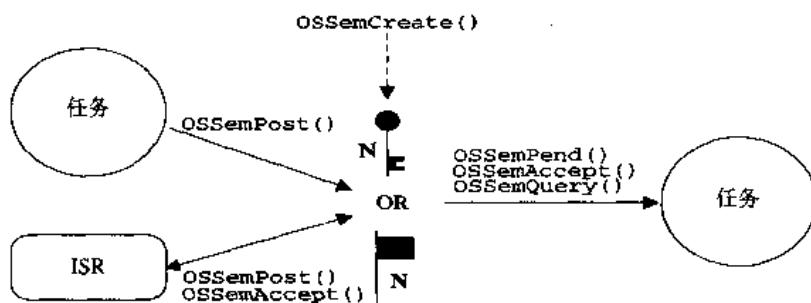


图 6.5 任务、中断服务子程序和信号量之间的关系

6.5.1 建立一个信号量, OSSemCreate()

程序清单 6.9 是 OSSemCreate() 函数的源代码。首先，它从空闲任务控制块链表中得到一个事件控制块[程序清单 6.9(1)]，并对空闲事件控制链表的指针进行适当的调整，使它指向下一个空闲的事件控制块[程序清单 6.9(2)]。如果这时有任务控制块可用[程序清单 6.9(3)]，就将该任务控制块的事件类型设置成信号量 OS_EVENT_TYPE_SEM[程序清单

6.9(4)]。其他的信号量操作函数 OSSem???()通过检查该域来保证所操作的任务控制类型的正确。例如，这可以防止调用 OSSemPost()函数对一个用作邮箱的任务控制块进行操作（见 6.6 节）。接着，用信号量的初始值对任务控制块进行初始化[程序清单 6.9(5)]，并调用 OSEventWaitListInit()函数对事件控制任务控制块的等待任务列表进行初始化（见 6.1 节）[程序清单 6.9(6)]。因为信号量正在被初始化，所以这时没有任何任务等待该信号量。最后，

务控制块的指针。以后对信号量的所有操作，如 OSSemPend()和 OSSemQuery()都是通过该指针完成的。因此，如果系统中没有可用的任务控制块，OSSemCreate()

量一旦建立就不能删除了，因此也就不可能将一链表中。如果有任务正在等待某个信号量，或者删除该任务是很危险的。

```
OS_EVENT *OSSemCreate (INT16U id)
{
    OS_EVENT *pEvent;
    OS_SEM_ID_CRITICAL();
    pEvent = OSEventFreeList;
    if (OSEventFreeList != OS_EVENT_TYPE + 1) {
        OSEventFreeList = OSEventFreeList->OSEventFreeList;
    }
    OS_EXIT_CRITICAL();
    if (pEvent != OS_EVENT_TYPE + 1) {
        pEvent->OSEventType = OS_EVENT_TYPE_SEM;
        pEvent->OSEventID = id;
        OSEventWaitListInit(pEvent);
    }
    return (pEvent);
}
```

6.5.2 等待一个信号量, OSSemPend()

程序清单 6.10 是 OSSemPend()函数的源代码。它首先检查指针 pevent 所指的任务控制

块是否是由 OSSemCreate()建立的[程序清单 6.10(1)]。如果信号量当前是可用的（信号量的计数值大于 0）[程序清单 6.10(2)]，将信号量的计数值减 1[程序清单 6.10(3)]，然后函数将“无错”错误代码返回给它的调用函数。显然，如果正在等待信号量，这时的输出正是我们所希望的，也是运行 OSSemPend()函数最快的路径。

如果此时信号量无效（计数器的值是 0），OSSemPend()函数要进一步检查它的调用函数是不是中断服务子程序[程序清单 6.10(4)]。在正常情况下，中断服务子程序是不会调用 OSSemPend()函数的。这里加入这些代码，只是为了以防万一。当然，在信号量有效的情况下，即使是中断服务子程序调用的 OSSemPend()，函数也会成功返回，不会出任何错误。

如果信号量的计数值为 0，而 OSSemPend()函数又不是由中断服务子程序调用的，则调用 OSSemPend()函数的任务要进入睡眠状态，等待另一个任务（或者中断服务子程序）发出该信号量。OSSemPend()允许用户定义一个最长等待时间作为它的参数，这样可以避免该任务无休止地等待下去。如果该参数值是一个大于 0 的值，那么该任务将一直等到信

0，该任务将一直等待下去。OSSemPend()函数通过 nt 置 1，把任务置于睡眠状态[程序清单 6.10(5)]，清单 6.10(6)]，该值在 OSTimeTick()函数中被逐次递减。注意，OSTimeTick()函数对每个任务的任务控制块的 OSTCBIdle 域做递减操作（只要该域不为 0）（见 3.10 节）。真正将任务置入睡眠状态的操作在 OSEventTaskWait()函数中执行（见 6.3 节）[程序清单 6.10(7)]。

因为当前任务已经不是就绪态了，所以任务调度函数将下一个最高优先级的任务调入，准备运行[程序清单 6.10(8)]。当信号量有效或者等待时间到后，调用 OSSemPend()函数的任务将再一次成为最高优先级任务。这时 OSSched()函数返回。这之后，OSSemPend()要检查任务控制块中的状态标志，看该任务是否仍处于等待信号量的状态[程序清单 6.10(9)]。如果是，说明该任务还没有被 OSSemPost()函数发出的信号量唤醒。事实上，该任务是因为等待超时而由 TimeTick()函数把它置为就绪状态的。这种情况下，OSSemPend()函数调用 OSEventTO()函数将任务从等待任务列表中删除[程序清单 6.10(10)]，并返回给它的调用任务一个“超时”的错误代码。如果任务的任务控制块中的 OS_STAT_SEM 标志位没有置位，就认为调用 OSSemPend()的任务已经得到了该信号量，将指向信号量 ECB 的指针从该任务的任务控制块中删除，并返回给调用函数一个“无错”的错误代码[程序清单 6.10(11)]。

程序清单 6.10 等待一个信号量

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

```

pevt = OS_Join(SemWait, 0);

if (psem->OSEventGrp > 0) {           (2)
    psem->OSEventGrp -= 1;
    OS_EXIT_CRITICAL();
    pevt = OS_RDN_SMP;
}

else if (OSEventGrp > 0) {               (3)
    OS_EXIT_CRITICAL();
    pevt = OS_DNN_PMSL;
}

else {
    OS_ENTER_CRITICAL();                (4)
    if (psem->OSEventGrp <= 0) {        (5)
        OSEventGrp = 0;
        OS_ENTER_CRITICAL();            (6)
        OS_EXIT_CRITICAL();
        pevt = OS_TMRDNN;
    }
}

```

6.5.3 发送一个信号量，OSSemPost()

程序清单 6.11 是 OSSemPost()函数的源代码。它首先检查参数指针 pevent 指向的任务控制块是否是 OSSemCreate()函数建立的[程序清单 6.11(1)]，接着检查是否有任务在等待该信号量[程序清单 6.11(2)]。如果该任务控制块中的.OSEventGrp 域不是 0，说明有任务正在等待该信号量。这时，就要调用函数 OSEventTaskRdy()（见 6.2 节），把其中的最高优先级任务从等待任务列表中删除[程序清单 6.11(3)]并使它进入就绪状态。然后，调用 OSSched()任务调度函数检查该任务是否是系统中的最高优先级的就绪任务[程序清单

6.11(4)]。如果是，这时就要进行任务切换（当 OSSemPost()函数是在任务中调用时），准备执行该就绪任务。如果不是，OSSched()直接返回，调用 OSSemPost()的任务得以继续

号量，该信号量的计数值就简单地加 1[程序清单

```
INT32 OSSemPost( OS_EVENT *pEvent )
{
    OS_EXIT_CRITICAL();                                (1)
    if (pEvent->OSEventType != OS_EVENT_TYPE_SEM ) {   (2)
        OS_EXIT_CRITICAL();                            (3)
        return (OS_ERR_INVALID_TYPE);
    }
    if (pEvent->OSEventGpr) {                          (4)
        OSSEMTaskify(pEvent, (pEvent->OS_PRIO_SEM), (5)
        OS_EXIT_CRITICAL());                           (6)
        OSSched();
        return (OS_NO_ERR);
    }
    else {                                              (7)
        if (pEvent->OSEventCnt < 65535) {             (8)
            pEvent->OSEventCnt++;                     (9)
            OS_EXIT_CRITICAL();                      (10)
            return (OS_NO_ERR);
        }
        else {                                         (11)
            OS_EXIT_CRITICAL();                      (12)
            return (OS_ERR_OVERFLOW);
        }
    }
}
```

的情况。当中断服务子程序调用该函数时，不会发生切换要等到中断嵌套的最外层中断服务子程序调用



6.5.4 无等待地请求一个信号量，OSSemAccept()

当一个任务请求一个信号量时，如果该信号量暂时无效，也可以让该任务简单地返回，而不是进入睡眠等待状态。这种操作是由 OSSemAccept()函数完成的，其源代码见程序清单 6.12。该函数在最开始也是检查参数指针 pevent 指向的事件控制块是否是由

(1)]，接着从该信号量的事件控制块中取出当前计数值是否有效(计数值是否为非 0 值)[程序清单 6.12(3)]。[程序清单 6.12(4)]，然后将信号量的原有计数值返回给要对该返回值进行检查。如果该值是 0，说明该信号量有效，同时该值也暗示着该信号量当前可用的，已经被该调用函数自身占用了一个(该计数值)。求信号量时，只能用 OSSemAccept()而不能用允许等待的。

```
INT140 OSSemAccept( OS_EVENT *pevent )
{
    OS_EVENT_BLOCK event;
    int32_t cnt;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {
        OS_EXIT_CRITICAL();
        return (0);
    }

    cnt = pevent->OSEventData;
    if (cnt > 0) {
        pevent->OSEventData--;
    }
    OS_EXIT_CRITICAL();
    return (cnt);
}
```

程序清单 6.12 无等待地请求一个信号量

6.5.5 查询一个信号量的当前状态，OSSemQuery()

在应用程序中，用户随时可以调用函数 OSSemQuery()（程序清单 6.13）来查询一个信

号量的当前状态。该函数有两个参数：一个是指向信号量对应事件控制块的指针 pevent。该指针是在生产信号量时，由 OSSemCreate()函数返回的；另一个是指向用于记录信号量信息的数据结构 OS_SEM_DATA（见 uCOS-II.H）的指针 pdata。因此，调用该函数前，用户必须先定义该结构变量，用于存储信号量的有关信息。此处之所以使用一个新的数据结构的原因是，OS_SEM_QUERY()函数只处理信号量有关的信息，而不是更一般的 OS_EVENT 数据结构。也就是说，OS_SEM_QUERY()函数只关心信号量的当前状态、等待任务数、等待任务列表等信息，而不关心事件控制块的其他信息，如事件类型、事件计数器等。

OS_SEM_QUERY()函数的实现代码如下所示。从代码中可以看出，该函数首先检查 pevent 指向的事件控制块是否为信号量事件控制块，即 pevent->OSEventType 是否大于 OS_EVENT_TYPE_SEM。如果大于，则返回 OS_ERROR_WRONGTYPE。如果不大于，则将 pevent 指向的事件控制块的 OS_EVENT_TYPE 和 OS_EVENT_PTR 成员分别赋值给 pdata->OSEventType 和 pdata->OSEventPtr。然后，将 pevent 指向的事件控制块的 OS_EVENT_TBL[0]成员赋值给 pEvent。接着，将 pevent 指向的事件控制块的 OS_EVENT_CNT 成员赋值给 pSCnt。最后，将 pevent 指向的事件控制块的 OS_EVENT_WAITING 成员赋值给 pWaitList。至此，OS_SEM_QUERY()函数就将等待任务列表结构拷贝到了 OS_SEM_DATA 结构变量中去。

OS_SEM_QUERY()也是先检查 pevent 指向的事件控制块是否为信号量事件控制块，即 pevent->OSEventType 是否大于 OS_EVENT_TYPE_SEM。如果大于，则返回 OS_ERROR_WRONGTYPE。如果不大于，则将 pevent 指向的事件控制块的 OS_EVENT_TYPE 和 OS_EVENT_PTR 成员分别赋值给 pdata->OSEventType 和 pdata->OSEventPtr。接着，将 pevent 指向的事件控制块的 OS_EVENT_TBL[0]成员赋值给 pEvent。然后，将 pevent 指向的事件控制块的 OS_EVENT_CNT 成员赋值给 pSCnt。最后，将 pevent 指向的事件控制块的 OS_EVENT_WAITING 成员赋值给 pWaitList。至此，OS_SEM_QUERY()函数就将等待任务列表结构拷贝到了 OS_SEM_DATA 结构变量中去。

```

INT8U OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata)
{
    INT8U i;
    INT8U *pEventData;
    INT8U *pEventData;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType > OS_EVENT_TYPE_SEM) {
        OS_EXIT_CRITICAL();
        return OS_ERROR_WRONGTYPE;
    }
    pEventData = pevent->OSEventData;
    pEvent = pEventData->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdata++ = *(pEventData + i);
    }
    pEventData = pevent->OSEventData;
    OS_EXIT_CRITICAL();
    return OS_NO_ERROR;
}

```

6.6 邮箱

邮箱是μC/OS-II 中另一种通信机制，它可以使一个任务或者中断服务子程序向另一个任务发送一个指针型的变量。该指针指向一个包含了特定“消息”的数据结构。为了在μC/OS-II 中使用邮箱，必须将 OS_CFG.H 中的 OS_MBOX_EN 常数置为 1。

使用邮箱之前，必须先建立该邮箱。该操作可以通过调用 OSMboxCreate()函数来完成（见 6.6.1 节），并且要指定指针的初始值。一般情况下，这个初始值是 NULL，但也可以初始化一个邮箱，使其在最开始就包含一条消息。如果使用邮箱的目的是用来通知一个事件的发生（发送一条消息），那么就要初始化该邮箱为 NULL，因为在开始时，事件还没有发生。如果用户用邮箱来共享某些资源，那么就要初始化该邮箱为一个非 NULL 的指针。在这种情况下，邮箱被当成一个二值信号量使用。

μC/OS-II 提供了 5 种对邮箱的操作：OSMboxCreate()，OSMboxPend()，OSMboxPost()，OSMboxAccept()和 OSMboxQuery()函数。图 6.6 描述了任务、中断服务子程序和邮箱之间的关系，这里用符号“I”表示邮箱。邮箱包含的内容是一个指向一条消息的指针。一个邮箱只能包含一个这样的指针（邮箱为满时），或者一个指向 NULL 的指针（邮箱为空时）。从图 6.6 可以看出，任务或者中断服务子程序可以调用函数 OSMboxPost()，但是只有任务可以调用函数 OSMboxPend()和 OSMboxQuery()。

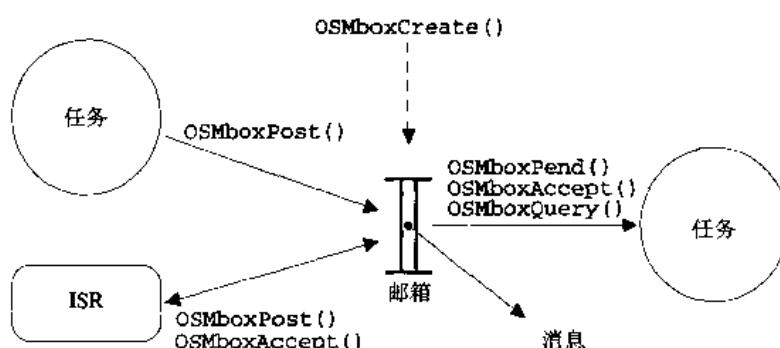


图6.6 任务、中断服务子程序和邮箱之间的关系

6.6.1 建立一个邮箱，OSMboxCreate()

程序清单 6.14 是 OSMboxCreate()函数的源代码，基本上和函数 OSSemCreate()相似。不同之处在于事件控制块的类型被设置成 OS_EVENT_TYPE_MBOX[程序清单 6.14(1)]，以及使用OSEventPtr 域来容纳消息指针，而不是使用OSEventCnt 域[程序清单 6.14(2)]。

OSMboxCreate()函数的返回值是一个指向事件控制块的指针[程序清单 6.14(3)]。这个指针在调用函数 OSMboxPend()，OSMboxPost()，OSMboxAccept()和 OSMboxQuery()时使

```

OSEVENT *OSMboxCreate(OSEvent msg)
{
    OSEVENT *pEvent;
    OS_EVENT_CRITICAL();
    pEvent = OSEventFreeList;
    if (OSEventFreeList == (OSEVENT *) 0) {
        OSEventFreeList = OSEVENT + OSEventFreeList->OSEventFirst;
        pEvent = OSEventFreeList;
    }
    OS_EVENT_CREATE(pEvent);
    if (pEvent != OS_EVENT + 0) {
        pEvent->OSEventType = OS_EVENT_TYPE_MESSAGE;
        pEvent->OSEventPcr = msg;
        OSEventInitList(pEvent);
    }
    return (pEvent);
}

```

中止。该函数用于等待一个邮箱中的句柄。值得注意的是，如果系统中已经没有事件一个 NULL 指针。

如，如果有任务正在等待一个邮箱的信息，这时



6.6.2 等待一个邮箱中的消息，OSMboxPend()

程序清单 6.15 是 OSMboxPend()函数的源代码。同样，它和 OSSemPend()也很相似，因此，在这里只讲述其中的不同之处。OSMboxPend()首先检查该事件控制块是由 OSMboxCreate()函数建立的[程序清单 6.15(1)]。当.OSEventPtr 域是一个非 NULL 的指针时，说明该邮箱中有可用的消息[程序清单 6.15(2)]。这种情况下，OSMboxPend()函数将该域的值复制到局部变量 msg 中，然后将.OSEventPtr 置为 NULL[程序清单 6.15(3)]。这正是我们所期望的，也是执行 OSMboxPend()函数最快的路径。

如果此时邮箱中没有消息是可用的 (.OSEventPtr 域是 NULL 指针)，OSMboxPend()函数检查它的调用者是否是中断服务子程序[程序清单 6.15(4)]。像 OSSemPend()函数一样，

不能在中断服务子程序中调用 OSMboxPend(), 因为中断服务子程序是不能等待的。这里的代码同样是为了以防万一。但是，如果邮箱中有可用的消息，即使从中断服务子程序中调用 OSMboxPend()，任务控制块中的 OSWaitEvent() 会挂起。

```
void *OSBoxRead(OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    void *msg;
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MAILBOX) {           (1)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
    msg = pevent->OSEventData;
    if (msg != (void *)0) {                                         (2)
        pevent->OSEventData = (void *)0;
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERROR;
    } else if (timeout > 0) {                                       (3)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_NO_WAIT;
    } else {
        OS_TICKCur->OSTickStat |= OS_STAT_WAIT;
        OS_TICKCur->OSTickVal += timeout;
        OSEventTaskMail(pevent);
        OS_SCHED();
        OS_EXIT_CRITICAL();
        if (msg == OS_TICKCur->OSTickMsg) { (4)
            *err = OS_ERR_NO_WAIT;
        }
    }
}
```

OSMboxPend()的任务就被挂起，直到邮箱中有了消息其他的任务向该邮箱发送了消息后（或者等待时优先级任务，OSSched()返回。这时，OSMboxPend()任务控制块中[程序清单 6.15(6)]。如果有，那么该函数。

超星奥贝思
使用本复制品
请尊重相关知识产权!

```

OSTCBStat->OSTCBMbox = (void *)0;
OSTCBStat->OSTCBStat = OS_STAT_READY;
OSTCBStat->OSTCBEventPtr = (OS_EVENT *)0;
OSListDelete(OSList);
*err = OS_NO_ERROR;
} else if (OSTCBStat->OSTCBStat & OS_STAT_MBOX) {
    OSEventTo(OSEvent);
    OSListDelete(OSList);
    OSListCreate(OSList);
    mbo = (OS_MBOX *)0;
    *err = OS_TIMEDOUT;
}
else {
    msg = (OS_MESSAGE *)OSEvent->OSEventData;
    pevent->OSEventPtr = (void *)0;
    OSTCBStat->OSTCBEventPtr = (OS_EVENT *)0;
    OSListDelete(OSList);
    *err = OS_NO_ERROR;
}

return (msg);
}

```

在 OSMboxPend()函数中，通过检查任务控制块中的.OSTCBStat 域中的 OS_STAT_MBOX 位，可以知道是否等待超时。如果该域被置 1，说明任务等待已经超时[程序清单 6.15(7)]。这时，通过调用函数 OSEventTo()可以将任务从邮箱的等待列表中删除[程序清单 6.15(8)]。因为此时邮箱中没有消息，所以返回的指针是 NULL[程序清单 6.15(9)]。如果 OS_STAT_MBOX 位没有被置 1，说明所等待的消息已经被发出。OSMboxPend()的调用函数得到指向消息的指针[程序清单 6.15(10)]。此后，OSMboxPend()函数通过将邮箱事件控制块的.OSEventPtr 域置为 NULL 清空该邮箱，并且要将该任务任务控制块中指向邮箱事件控制块的指针删除[程序清单 6.15(12)]。

6.6.3 发送一个消息到邮箱中，OSMboxPost()

程序清单 6.16 是 OSMboxPost()函数的源代码。检查了事件控制块是否是一个邮箱后[程序清单 6.16(1)]，OSMboxPost()函数还要检查是否有任务在等待该邮箱中的消息[程序清单 6.16(2)]。如果事件控制块中的 OSEventGrp 域包含非零值，就暗示着有任务在等待该消息。这时，调用 OSEventTaskRdy()将其中的最高优先级任务从等待列表中删除（见 6.2 节）[程序清单 6.16(3)]，加入系统的就绪任务列表中，准备运行。然后，调用 OSSched()函数[程序

清单 6.16(4)]，检查该任务是否是系统中最高优先级的就绪任务。如果是，执行任务切换[仅当 OSMboxPost()函数是由任务调用时]，该任务得以执行。如果该任务不是最高优先级的任务，OSSched()返回，OSMboxPost()的调用函数继续执行。如果没有任何任务等待该消息，

```
int8_t OSMboxPost( osEventId *pEvent, void *msg )
{
    OS_ENTER_CRITICAL();
    if (pEvent->OSEventType == OS_EVENT_TYPE_BLOCK) {
        OS_EXIT_CRITICAL();
        return OS_NO_EVENT_TYPE;
    }
    if (pEvent->OSEventObj < 0)
        OSEventNotify(pEvent, msg, OS_STAT_BLOCK);
    OS_EXIT_CRITICAL();
    OSSched();
    return OS_NO_ERR;
} else {
    if (pEvent->OSEventObj >= 0)
        OS_EXIT_CRITICAL();
    return OS_NO_ERR;
}
}
```

单 6.16(6)]（假设此时邮箱中的指针不是非 NULL）
OSMboxPend()函数的任务就可以立刻得到该消

断服务子程序中调用的，那么，这时并不发生任
起的任务切换只发生在中断嵌套的最外层中断服
见 3.9 节）。



6.6.4 无等待地从邮箱中得到一个消息，OSMboxAccept()

应用程序也可以以无等待的方式从邮箱中得到消息。这可以通过程序清单 6.17 中的 OSMboxAccept() 函数来实现。OSMboxAccept() 函数开始也是检查事件控制块是否是由

OSMboxAccept(100_EVENT *pevent) {
 void *msg;

 OS_ENTER_CRITICAL();
 if (pevent->uEvent.Type == OS_EVENT_TYPE_MAILBOX) {
 OS_EXIT_CRITICAL();
 return (void *)0;
 }
 msg = pevent->uEvent.Ptr;
 if (msg != (void *)0) {
 pevent->uEvent.Ptr = (void *)0;
 }
 OS_EXIT_CRITICAL();
 return msg; }

OSMboxAccept(100_EVENT *pevent) [程序清单 6.17(1)]。接着，它得到邮箱中的当前内容[程序清单 6.17(2)]。如果邮箱中有消息，就把邮箱清空。如果邮箱中没有消息，就返回一个指向消息的指针被返回给 OSMboxAccept() 的调用函数。如果邮箱中没有消息，那么 OSMboxAccept() 的调用函数必须检查该返回值是否为 NULL。如果返回值不是 NULL，则说明邮箱中有可用的消息。如果该值是非 NULL 值，说明邮箱清空了该消息。中断服务子程序在试图得到一个消息时不能使用 OSMboxPend() 函数。

注意：用户可以用它来清空一个邮箱中现有的内容。

6.6.5 从邮箱中得到消息

6.6.5 查询一个邮箱的状态，OSMboxQuery()

OSMboxQuery() 函数使应用程序可以随时查询一个邮箱的当前状态。程序清单 6.18 是该函数的源代码。它需要两个参数：一个是指向邮箱的指针 pevent。该指针是在建立该

邮箱时，由 OSMboxCreate()函数返回的；另一个是指向用来保存有关邮箱的信息的 OS_MBOX_DATA（见 uCOS_II.H）数据结构的指针 pdata。在调用 OSMboxCreate()函数之前，必须先定义该结构变量，用来保存有关邮箱的信息。之所以定义一个新的数据结

构，是先检查事件控制块是否是邮箱型[程序清单 6.18(2)]和邮箱中的消息[程序清单 6.18(3)]

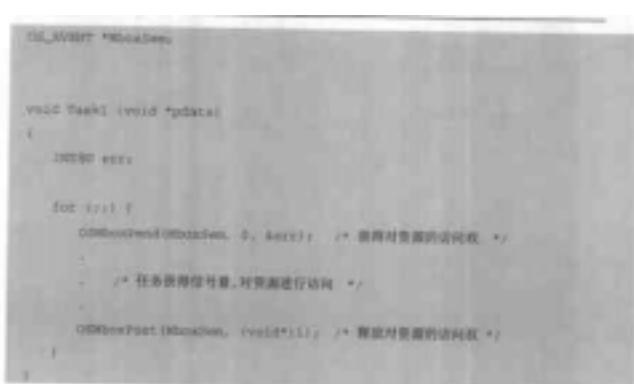
的内容，而非整个 OS_EVENT 数据结构的内
tCnt 和.OSEventType)，而 OS_MBOX_DATA 只
该邮箱现有的等待任务列表 (.OSEventTbl[])

```
INT32 OSBoxQuery(OS_EVENT *pevent, OS_MBOX_DATA *pdata)
{
    OSLOCK();
    if(pdata == NULL) {
        return OS_NO_ERROR;
    }
    if(pevent->OSEventType != OS_EVENT_TYPE_MAILBOX) {
        OS_EXIT_CRITICAL();
        return OS_NO_ERROR;
    }
    pdata->OSEventType = pevent->OSEventType;
    pdata->OSEventCtrl = pevent->OSEventCtrl;
    pdata->OSEventTbl = pevent->OSEventTbl;
    for(i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdata++ = *(pevent++);
    }
    pdata->OSEmail = pevent->OSEmail;
    OS_EXIT_CRITICAL();
    return OS_NO_ERROR;
}
```

是先检查事件控制块是否是邮箱型[程序清单 6.18(2)]和邮箱中的消息[程序清单 6.18(3)]

K_DATA 数据结构。

6.6.6 将邮箱用作二值信号量



的信号量。首先，在初始化时，将邮箱设置为一个二值信号量。任务在运行时，如果需要访问资源，可以调用 `OSMboxPend()` 函数来请求一个信号量，如果成功，返回值为 0；如果失败，返回值为 -1。如果任务获得信号量，对资源进行访问，然后调用 `OSMboxPost()` 函数，将信号量释放，这样其他任务就可以访问该资源了。

6.6.7 用邮箱实现延时，而不使用 OSTimeDly()

邮箱的等待超时功能可以被用来模仿 `OSTimeDly()` 函数的延时，如程序清单 6.20 所示。如果在指定的时间段 `TIMEOUT` 内，没有消息到来，`Task1()` 函数将继续执行。这和 `OSTimeDly(TIMEOUT)` 功能很相似。但是，如果 `Task2()` 在指定的时间结束之前，向该邮箱发送了一个“虚拟”消息，`Task1()` 就会提前开始继续执行。这和调用 `OSTimeDlyResume()` 函数的功能是一样的。注意，这里忽略了对返回的消息的检查，因为此时关心的不是得到了什么样的消息。

```
OS_EVENT *NewTimeOut;

void Task1 (void *pdata)
{
    INT8U ret;

    for (i=1;
        OSCreateTask(0x00000000, 10000, /* 临时任务 */;
        /* 临时结束后执行的代码 */);

    for (j=1;
        OSDeleteTask(0x00000001); /* 取消任务1的定时 */;
        /* 临时结束后执行的代码 */);
}
```

6.7 消息队列

消息队列是μC/OS-II 中另一种通信机制，它可以使一个任务或者中断服务子程序向另一个任务发送以指针方式定义的变量。因具体的应用有所不同，每个指针指向的数据结构变量也有所不同。为了使用μC/OS-II 的消息队列功能，需要在 OS_CFG.H 文件中，将

`OS_Q_EN` 常数设置为 1，并且通过常数 `OS_MAX_QS` 来决定μC/OS-II 支持的最多消息队列数。

在使用一个消息队列之前，必须先建立该消息队列。这可以通过调用 `OSQCreate()` 函数（见 6.7.1 节），并定义消息队列中的单元数（消息数）来完成。

μC/OS-II 提供了 7 个对消息队列进行操作的函数：`OSQCreate()`、`OSQPend()`、`OSQPost()`、`OSQPostFront()`、`OSQAccept()`、`OSQFlush()` 和 `OSQuery()` 函数。图 6.7 是任务、中断服务子程序和消息队列之间的关系。其中，消息队列的符号很像多个邮箱。实际上，我们可以将消息队列看作时多个邮箱组成的数组，只是它们共用一个等待任务列表。每个指针所指向的数据结构是由具体的应用程序决定的。`N` 代表了消息队列中的总单元数。当调用 `OSQPend()` 或者 `OSQAccept()` 之前，调用 `N` 次 `OSQPost()` 或者 `OSQPostFront()` 就会把消息队列填满。从图 6.7 中可以看出，一个任务或者中断服务子程序可以调用 `OSQPost()`、`OSQPostFront()`、`OSQFlush()` 或者 `OSQAccept()` 函数。但是，只有任务可以调用 `OSQPend()` 和 `OSQuery()` 函数。

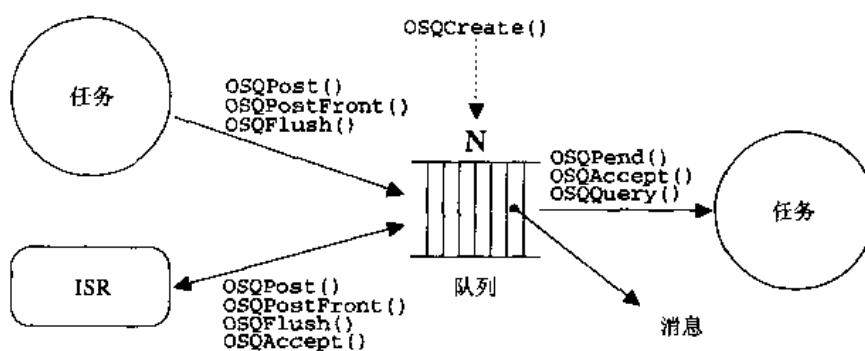


图6.7 任务、中断服务子程序和消息队列之间的关系

图 6.8 是实现消息队列所需要的各种数据结构。这里也需要事件控制块来记录等待任务列表[图 6.8(1)]，而且，事件控制块可以使多个消息队列的操作和信号量操作、邮箱操作相同的代码。当建立了一个消息队列时，一个队列控制块 (`OS_Q` 结构，见 `OS_Q.C` 文件) 也同时被建立，并通过 `OS_EVENT` 中的 `.OSEventPtr` 域链接到对应的事件控制块[图 6.8(2)]。在建立一个消息队列之前，必须先定义一个含有与消息队列最大消息数相同个数的指针数组[图 6.8(3)]。数组的起始地址以及数组中的元素数作为参数传递给 `OSQCreate()` 函数。事实上，如果内存占用了连续的地址空间，也没有必要非得使用指针数组结构。

文件 `OS_CFG.H` 中的常数 `OS_MAX_QS` 定义了在μC/OS-II 中可以使用的最大消息队列数，这个值最小应为 2。μC/OS-II 在初始化时建立一个空闲的队列控制块链表，如图 6.9 所示。

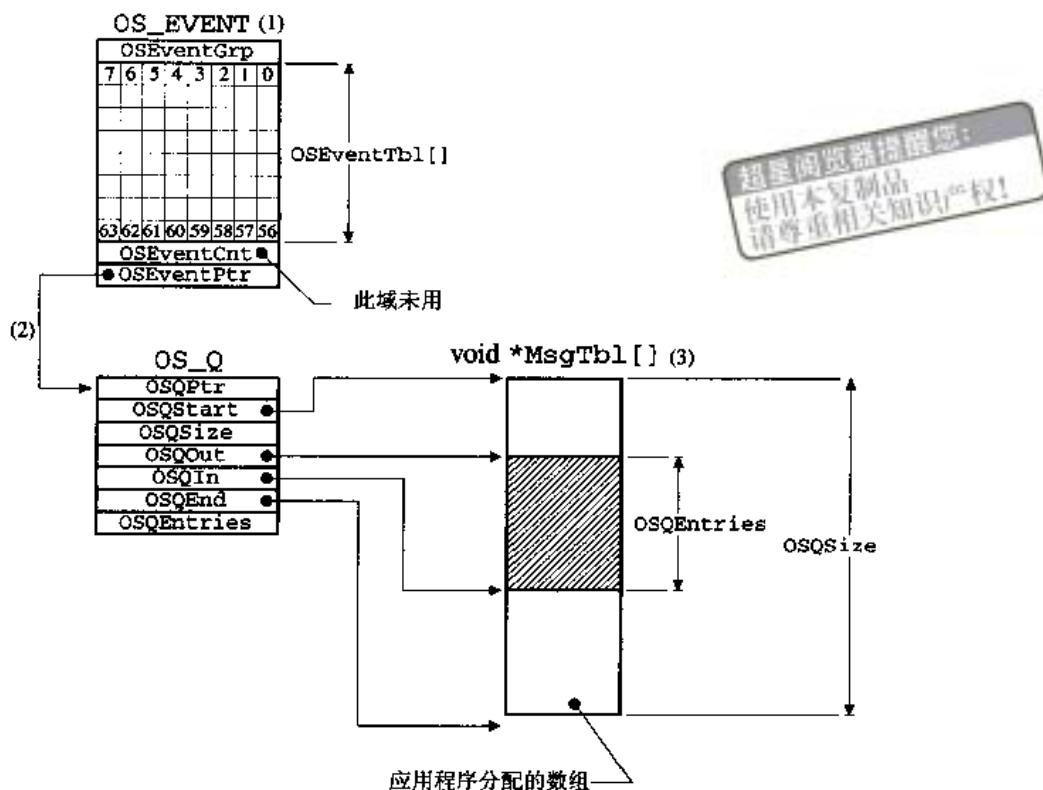


图6.8 用于消息队列的数据结构

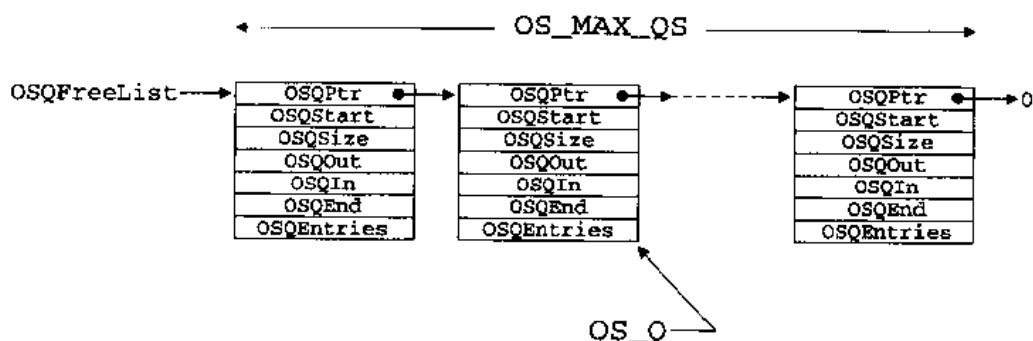


图6.9 空闲队列控制块链表

队列控制块是一个用于维护消息队列信息的数据结构，它包含了以下的一些域。这里，仍然在各个变量前加入一个`[.]`来表示它们是数据结构中的一个域。

.OSQPtr 在空闲队列控制块中链接所有的队列控制块。一旦建立了消息队列，该域就不再有用了。

.OSQStart 是指向消息队列的指针数组的起始地址的指针。用户应用程序在使用消息队列之前必须先定义该数组。

.OSQEnd 是指向消息队列结束单元的下一个地址的指针。该指针使得消息队列构成一个循环的缓冲区。

.OSQIn 是指向消息队列中插入下一条消息的位置的指针。当`.OSQIn` 和`.OSQEnd` 相等时，`.OSQIn` 被调整指向消息队列的起始单元。

.OSQOut 是指向消息队列中下一个取出消息的位置的指针。当`.OSQOut` 和`.OSQEnd` 相等时，`.OSQOut` 被调整指向消息队列的起始单元。

.OSQSize 是消息队列中总的单元数。该值是在建立消息队列时由用户应用程序决定的。在μC/OS-II 中，该值最大可以是 65535。

.OSQEntries 是消息队列中当前的消息数量。当消息队列是空的时，该值为 0。当消息队列满了以后，该值和`.OSQSize` 值一样。在消息队列刚刚建立时，该值为 0。

消息队列最根本的部分是一个循环缓冲区，如图 6.10。其中的每个单元包含一个指针。队列未满时，`.OSQIn` [图 6.10(1)] 指向下一个存放消息的地址单元。如果队列已满(`.OSQEntries` 与`.OSQSize` 相等)，`.OSQIn` [图 6.10(3)] 则与`.OSQOut` 指向同一单元。如果在`.OSQIn` 指向的单元插入新的指向消息的指针，就构成 FIFO (First-In-First-Out) 队列。相反，如果在`.OSQOut` 指向的单元的下一个单元插入新的指针，就构成 LIFO 队列 (Last-In-First-Out) [图 6.10(2)]。当`.OSQEntries` 和`.OSQSize` 相等时，说明队列已满。消息指针总是从`.OSQOut` [图 6.10(4)] 指向的单元取出。指针`.OSQStart` 和`.OSQEnd` [图 6.10(5)] 定义了消息指针数组的头尾，以便在`.OSQIn` 和`.OSQOut` 到达队列的边缘时，进行边界检查和必要的指针调整，实现循环功能。

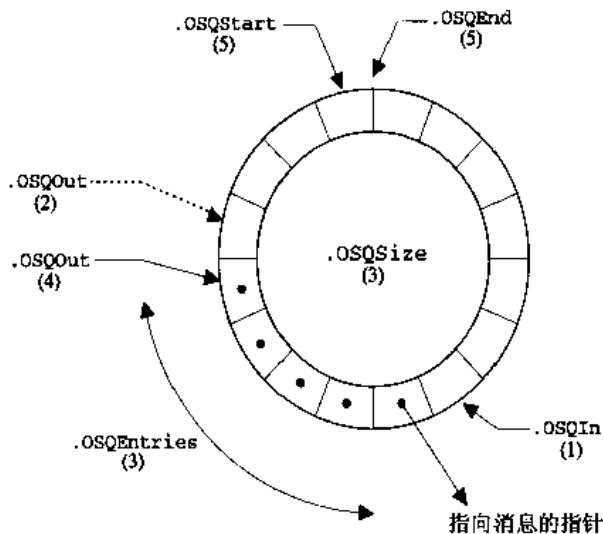


图6.10 消息队列是一个由指针组成的循环缓冲区

6.7.1 建立一个消息队列，OSQCreate()

程序清单 6.21 是 OSQCreate() 函数的源代码。该函数需要一个指针数组来容纳指向各个消息的指针。该指针数组必须声明为 void 类型。

OSQCreate()首先从空闲事件控制块链表中取得一个事件控制块（见图 6.3）[程序清单 6.21(1)]，并对剩下的空闲事件控制块列表的指针做相应的调整，使它指向下一个空闲事件控制块[程序清单 6.21(2)]。接着，OSQCreate()函数从空闲队列控制块列表中取出一个队列控制块[程序清单 6.21(3)]。如果有空闲队列控制块是可以的，就对其进行初始化[程序清单 6.21(4)]。然后该函数将事件控制块的类型设置为 OS_EVENT_TYPE_Q [程序清单 6.21(5)]，并使其.OSEventPtr 指针指向队列控制块[程序清单 6.21(6)]。OSQCreate()还要调用

~~OSEventWaitListInit()~~ 函数对重位任务的等待任务列表初始化（见 6.01 节）[程序清单 6.21(7)]。

显然它的等待任务列表是空的。最后，OSQCreate()将队列控制块的指针[程序清单 6.21(9)]。该指针将在调用 OSQFlush(), OSQAccept()和 OSQQQuery()等消息以被看作是对应消息队列的句柄。值得注意的是，create()函数将返回一个 NULL 指针。如果没有队列资源，OSQCreate()函数将把刚刚取得的事件控制块的指针[程序清单 6.21(8)]。

删除了。试想，如果有任务正在等待某个消息队列是很危险的。

程序清单 6.21 建立一个消息队列

```
OS_EVENT *OSQCreate (void **start, INT16U id)
{
    OS_EVENT *pEvent;
    OS_Q *pq;

    OS_ENTER_CRITICAL();
    pEvent = OSEventFreeList;
    if (OSEventFreeList != (OS_EVENT *)0) {
        OSEventFreeList = OSEventFreeList->OSEventNext;
    }
    OS_EXIT_CRITICAL();
    if (pEvent != (OS_EVENT *)0) {
        OS_ENTER_CRITICAL();
        pq = OSQPfreeList;
        if (OSQPfreeList != (OS_Q *)0) {
            OSQPfreeList = OSQPfreeList->OSQPNext;
        }
    }
}
```

```

OS_EXIT_CRITICAL();
if (pq->OSQEntries >= start) {                                (4)
    pq->OSQFirst = &start;                                         (5)
    pq->OSQLast = &start;                                         (6)
    pq->OSQOut = &start;                                         (7)
    pq->OSQSize = size;
    pq->OSQFreeList = 0;
    pevent->OSEventType = OS_EVENT_TYPE_Q;                         (8)
    pevent->OSEventPte = (void *)OSEventFreeList;                   (9)
    OSEventMailboxExit(pevent);                                      (10)
} else {
    OS_ENTRY_CRITICAL();
    pevent->OSEventPte = (void *)OSEventFreeList;                   (11)
    OSEventFreeList = pevent;
    OS_EXIT_CRITICAL();
    pevent += OS_EVENT_SIZE;                                         (12)
}
return pevent;
}

```

(9)

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

6.7.2 等待一个消息队列中的消息，OSQPend()

程序清单 6.22 是 OSQPend()函数的源代码。OSQPend()函数首先检查事件控制块是否是由 OSQCreate()函数建立的[程序清单 6.22(1)]，接着，该函数检查消息队列中是否有消息可用（即.OSQEntries 是否大于 0）[程序清单 6.22(2)]。如果有，OSQPend()函数将指向消息的指针复制到 msg 变量中，并让.OSQOut 指针指向队列中的下一个单元[程序清单 6.22(3)]，然后将队列中的有效消息数减 1 [程序清单 6.22(4)]。因为消息队列是一个循环的缓冲区，OSQPend()函数需要检查.OSQOut 是否超过了队列中的最后一个单元 [程序清单 6.22(5)]。当发生这种越界时，就要将.OSQOut 重新调整到指向队列的起始单元 [程序清单 6.22(6)]。这是我们调用 OSQPend()函数时所期望的，也是执行 OSQPend()函数最快的路径。

程序清单 6.22 在一个消息队列中等待一条消息

```
void *OSQPend(OS_EVENT *pevent, INT16U timeout, INT32 *err)
```

```

{
    void *msg;
    OS_Q *pq;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) { (1)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
    pq = pevent->OSEventPtr;
    if (pq->OSQEntries != 0) { (2)
        msg = *pq->OSQOut++;
        pq->OSQEntries--;
        if (pq->OSQOut == pq->OSQEnd) { (5)
            pq->OSQOut = pq->OSQStart; (6)
        }
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) { (7)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_PEND_ISR;
    } else {
        OSTCBCur->OSTCBStat |= OS_STAT_Q; (8)
        OSTCBCur->OSTCBDly = timeout;
        OSEventTaskWait(pevent);
        OS_EXIT_CRITICAL();
        OSSched(); (9)
        OS_ENTER_CRITICAL();
        if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) { (10)
            OSTCBCur->OSTCBMsg = (void *)0;
            OSTCBCur->OSTCBStat = OS_STAT_RDY;
            OSTCBCur->OSTCBEVENTPTR = (OS_EVENT *)0; (11)
        }
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    }
}

```

```

1 else if (OSTCBear->OSTCBStat & OS_STAT_Q) {
2     OSEventTo(pevent);
3     OS_EXIT_CRITICAL();
4     msg = "msg->OSQOut++";
5     msg->OSQEventTime++;
6     if (msg->OSQOut == msg->OSQStart) {
7         msg->OSQOut = msg->OSQStart;
8     }
9     OSTCBear->OSTCBEventTime = OS_EVENT + 0;
10    OS_EXIT_CRITICAL();
11    *err = OS_NO_ERR;
12}
13
14return msg;
15

```

超星浏览器提醒您：
 使用本复制品
 请尊重相关知识产权！

如果这时消息队列中没有消息 (.OSEventEntries 是 0), OSQPend()函数检查它的调用者是否是中断服务子程序[程序清单 6.22(7)]。与 OSSemPend()和 OSMboxPend()函数一样, 不能在中断服务子程序中调用 OSQPend(), 因为中断服务子程序是不能等待的。但是, 如果消息队列中有消息, 即使从中断服务子程序中调用 OSQPend()函数, 也一样是成功的。

如果消息队列中没有消息, 调用 OSQPend()函数的任务被挂起[程序清单 6.22(8)]。当有其他的任务向该消息队列发送了消息或者等待时间超时, 并且该任务成为最高优先级任务时, OSSched()返回[程序清单 6.22(9)]。这时, OSQPend()要检查是否有消息被放到该任务的任务控制块中[程序清单 6.22(10)]。如果有, 那么该次函数调用成功, 把任务的任务控制块中指向消息队列的指针删除[程序清单 6.22(17)], 并将对应的消息被返回到调用函数[程序清单 6.22(17)]。

在 OSQPend()函数中, 通过检查任务的任务控制块中的.OSTCBStat 域, 可以知道是否等到时间超时。如果其对应的 OS_STAT_Q 位被置 1, 说明任务等待已经超时[程序清单 6.22(12)]。这时, 通过调用函数 OSEventTo()可以将任务从消息队列的等待任务列表中删除[程序清单 6.22(13)]。这时, 因为消息队列中没有消息, 所以返回的指针是 NULL[程序清单 6.22(14)]。

如果任务控制块标志位中的 OS_STAT_Q 位没有被置 1, 说明有任务发出了一条消息。OSQPend()函数从队列中取出该消息[程序清单 6.22(15)]。然后, 将任务的任务控制中指向



事件控制块的指针删除[程序清单 6.22(16)]。

6.7.3 向消息队列发送一个消息 (FIFO), OSQPost()

程序清单 6.23 是 OSQPost()函数的源代码。在确认事件控制块是消息队列后 [程序清单

在等待该消息队列中的消息[程序清单 6.23(2)]。当，说明该消息队列的等待任务列表中有任务。这时，从列表中取出最高优先级的任务[程序清单 6.23(3)]，**OSSched()** [程序清单 6.23(4)]进行任务的调度。如果该任务里也是最高的，而且 OSQPost()函数不是中断调用的，那么该最高优先级任务被执行。否则的话，OSSched()将返回，OSQPost()继续执行。

[息

```
INT8U OSQPost (OS_EVENT *pevent, void *msg)
{
    OS_Q    *pq;
    OS_EVENT_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE) {
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventType == OS_EVENT_TYPE) {
        OS_ENTER_CRITICAL();
        convertTaskPriority(pevent, msg, OS_STAT_WAIT);
        OS_EXIT_CRITICAL();
        OSSched();
        return (OS_NO_ERROR);
    }
    else {
        pq = pevent->OSEventPtr;
        if (pq->OSQueueType == gpr->OSQueue) {
            OS_EXIT_CRITICAL();
            return (OS_Q_FULL);
        }
        else {
            *pq->OSQueue = msg;
        }
    }
}
```

```
    pq->OSQInList++;
```

```
    if (pq->OSQIn == pq->OSQEnd) {
```

```
        pq->OSQIn = pq->OSQStart;
```

```
}
```

```
OS_BLOCK_INTERRUPT();
```

```
return PQS_NO_ERROR;
```

任务之间的通信与同步



如果没有任务等待该消息队列中的消息，而且此时消息队列未满[程序清单 6.23(5)]，指向该消息的指针被插入到消息队列中[程序清单 6.23(6)]。这样，下一个调用 OSQPend() 函数的任务就可以马上得到该消息。注意，如果此时消息队列已满，那么该消息将由于不

服务子程序调用的，那么即使产生了更高优先级的主任务切换。这个动作一直要等到中断嵌套的最外层中断服务子程序调用 OSIntExit() 函数时才能进行（见 3.9 节）。

6.7.4 向消息队列发送一个消息（后进先出 LIFO），OSQPostFront()

OSQPostFront()函数和 OSQPost()基本上是一样的，只是在插入新的消息到消息队列中时，使用.OSQOut 作为指向下一个插入消息的单元的指针，而不是.OSQIn。程序清单 6.24 是它的源代码。值得注意的是，.OSQOut 指针指向的是已经插入了消息指针的单元，所以再插入新的消息指针前，必须先将.OSQOut 指针在消息队列中前移一个单元。如果.OSQOut 指针指向的当前单元是队列中的第一个单元[程序清单 6.24(1)]，这时再前移就会发生越界，需要特别地将该指针指向队列的末尾[程序清单 6.24(2)]。由于.OSQEnd 指向的是消息队列中最后一个单元的下一个单元，因此.OSQOut 必须被调整到指向队列的有效范围内[程序清单 6.24(3)]。因为 QSQPend() 函数取出的消息是由 OSQPend() 函数刚刚插入的，因此 OSQPostFront() 函数实现了一个 LIFO 队列。

程序清单 6.24 向消息队列发送一条消息（LIFO）

```
OS_EXIT_CRITICAL();
if (pevent->OSEventType == OS_EVENT_TYPE_Q) {
    OS_EXIT_CRITICAL();
    release (OS_EQ_EVENT_TYPE);
}
}
if (pevent->OSEventQref) {
    OSEventTaskify(pevent, msg, OS_STAT_QP);
    OS_EXIT_CRITICAL();
    OSmEnd();
    return (OS_EQ_NOM);
}
else {
    if (pq->OSQueueType) {
        if (pq->OSQueueSize == pq->OSQOut) {
            OS_EXIT_CRITICAL();
            return (OS_EQ_FULL);
        }
        else {
            if (pq->OSQOut == pq->OSQIn) {
                if (pq->OSQOut + msg <= pq->OSQIn)
                    pq->OSQOut = pq->OSQIn;
                else
                    OSQOut++;
            }
            OS_EXIT_CRITICAL();
        }
    }
}
return (OS_EQ_EPRN);
```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

6.7.5 无等待地从一个消息队列中取得消息，OSQAccept()

如果试图从消息队列中取出一条消息，而此时消息队列又为空时，也可以不让调用任务等待而直接返回调用函数。这个操作可以调用 OSQAccept() 函数来完成。程序清单 6.25 是该函数的源代码。OSQAccept() 函数首先查看 pevent 指向的事件控制块是否是由 OSQCreate() 函数建立的[程序清单 6.25(1)]，然后它检查当前消息队列中是否有消息[程序清单 6.25(2)]。如果消息队列中有至少一条消息，那么就从 OSQOut 指向的单元中取出消息[程序清单 6.25(3)]。OSQAccept() 函数的调用函数需要对 OSQAccept() 返回的指针进行检查。

```

void *OSQPNext(OS_EVENT *pevent)
{
    void *msg;
    OS_Q *pq;

    OS_EXIT_CRITICAL();
    if (pevent->OSEventType == OS_EVENT_TYPE_Q) {
        OS_EXIT_CRITICAL();
        return (void *)0;
    }

    pq = pevent->OSEventData.Q;
    if (pq->OSQCount <= 0) {                                (1)
        msg = *(pq->OSQOut++);                            (2)
        pq->OSQInIndex++;                                 (3)
        if (pq->OSQOut == pq->OSQEnd) {
            pq->OSQOut = pq->OSQInIndex;
        }
    } else {
        msg = (void *)0;                                    (4)
    }
    OS_EXIT_CRITICAL();
    return msg;
}

```

空的，其中没有消息可以 [程序清单 6.25(4)]。否则得了一条消息。当中断服务子程序要从消息队列而不能使用 OSQPend()函数。

又一条消息

6.7.6 清空一个消息队列，OSQFlush()

OSQFlush()函数允许用户删除一个消息队列中的所有消息，重新开始使用。程序清单 6.26 是该函数的源代码。和前面的其他函数一样，该函数首先检查 `pevent` 指针是否是执行一个消息队列[程序清单 6.26(1)]，然后将队列的插入指针和取出指针复位，使它们都指向队列起始单元，同时，将队列中的消息数设为 0 [程序清单 6.26(2)]。这里，没有检查该消

```

#define OSQCreate( pEvent, pQueue )
{
    OS_Q *pq;
    OS_ENTER_CRITICAL();
    if (pEvent->OSEventType != OS_EVENT_TYPE_Q) {
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pq = pEvent->OSEvent.Ptr;
    pq->OSQIn = pEvent->OSEvent.In;
    pq->OSQOut = pEvent->OSEvent.Out;
    pq->OSQEntries = 0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}

```

要该等待任务列表不空，.OSQEntries 就一定是 0。此时可以指向消息队列中的任何单元，不一定是起



6.7.7 查询一个消息队列的状态，OSQuery()

OSQuery()函数使用户可以查询一个消息队列的当前状态。程序清单 6.27 是该函数的源代码。OSQuery()需要两个参数：一个是指向消息队列的指针 `pEvent`。它是在建立一个消息队列时，由 OSQCreate()函数返回的；另一个是指向 `OS_Q_DATA`（见 `uCOS_II.H`）数据结构的指针 `pdata`。该结构包含了有关消息队列的信息。在调用 OSQuery()函数之前，必须先定义该数据结构变量。`OS_Q_DATA` 结构包含下面的几个域：

.OSMsg 如果消息队列中有消息，它包含指针`.OSQOut`所指向的队列单元中的内容。如果队列是空的，`.OSMsg` 包含一个 NULL 指针。

.OSNMsgs 是消息队列中的消息数（`.OSQEntries` 的拷贝）。

.OSQSize 是消息队列的总的容量。

.OSEventTbl[] 和 **.OSEventGrp** 是消息队列的等待任务列表。通过它们，OSQuery()的调用函数可以得到等待该消息队列中的消息的任务总数。

OSQuery()函数首先检查 pevent 指针指向的事件控制块是一个消息队列[程序清单

清单 6.27(2)]。如果消息队列中有消息[程序清单 6.27(3)]，那么消息的内容被复制到 OS_Q_DATA 结构中[程序清单 6.27(4)]。最后，将消息队列中的消息清空[程序清单 6.27(5)]。

```
INTRO OSQuery(OS_EVENT *pevent, OS_Q_DATA *pdata)
{
    OS_Q *pq;
    INTRO L;
    INTRO *perr;
    INTRO *pdeact;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType == OS_EVENT_TYPE_Q) { //1
        OS_EXIT_CRITICAL();
        return OS_NO_EVENT_TYPE;
    }

    pdata->OSEventGrp = pevent->OSEventGrp; //2
    perr = &pevent->OSEventError[0];
    pdeact = &pevent->OSEventDeact[0];
    for (L = L + 1 + OS_EVENT_DEL_SIZE; L++) {
        pdata++ = *perr++;
    }

    pq = (OS_Q *)pevent->OSEventPtx;
    if (pq->OSQCount > 0) { //3
        pdata->OSMsg = pq->OSQOut; //4
    } else {
        pdata->OSMsg = (void *)0; //5
    }
    pdata->OSMsgs = pq->OSQCount; //6
    perr->OSError = pq->OSQError;
    OS_EXIT_CRITICAL();
    return OS_NO_ERROR;
}
```

使用本教材请尊重相关版权

6.7.8 使用消息队列读取模拟量的值

在控制系统中，经常要频繁地读取模拟量的值。这时，可以先建立一个定时任务 OSTimeDly()（见 5.0 节），并且给出希望的抽样周期。然后，如图 6.11 所示，让 A/D 采样的任务从一个消息队列中等待消息。该程序最长的等待时间就是抽样周期。当没有其他任务向该消息队列中发送消息时，A/D 采样任务因为等待超时而退出等待状态并进行执行。这就模仿了 OSTimeDly() 函数的功能。

也许，读者会提出疑问，既然 OSTimeDly() 函数能完成这项工作，为什么还要使用消息队列呢？这是因为，借助消息队列，我们可以让其他的任务向消息队列发送消息来终止 A/D 采样任务等待消息，使其马上执行一次 A/D 采样。此外，我们还可以通过消息队列来通知 A/D 采样程序具体对哪个通道进行采样，告诉它增加采样频率等等，从而使得我们的应用更智能化。换句话说，我们可以告诉 A/D 采样程序：“现在马上读取通道 3 的输入值！”之后，该采样任务将重新开始在消息队列中等待消息，准备开始一次新的扫描过程。

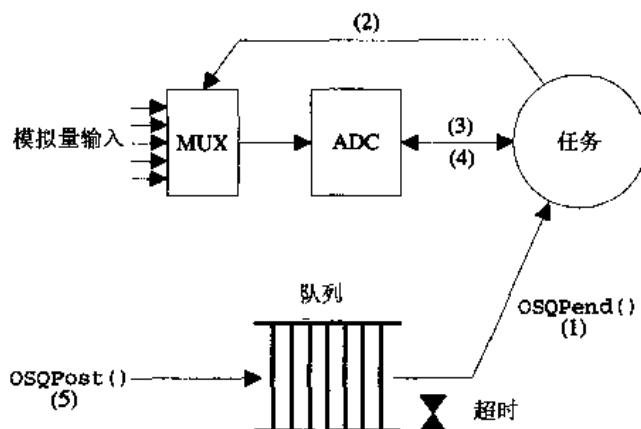


图 6.11 读模拟量输入

6.7.9 使用一个消息队列作为计数信号量

在消息队列初始化时，可以将消息队列中的多个指针设为非 NULL 值（如 void* 1），来实现计数信号量的功能。这里，初始化为非 NULL 值的指针数就是可用的资源数。系统中的任务可以通过 OSQPend() 来请求“信号量”，然后通过调用 OSQPost() 来释放“信号量”，如程序清单 6.28。如果系统中只使用了计数信号量和消息队列，使用这种方法可以有效地节省代码空间。这时将 OS_SEM_EN 设为 0，就可以不使用信号量，而只使用消息队列。值得注意的是，这种方法为共享资源引入了大量的指针变量。也就是说，为了节省代码空间，牺牲了 RAM 空间。另外，对消息队列的操作要比对信号量的操作慢，因此，当用计数信号量同步的信号量很多时，这种方法的效率是非常低的。

程序清单 6.28 使用消息队列作为一个计数信号量

```

OS_EVENT *QSem;
void      *QMsgTbl[N_RESOURCES]

void main (void)
{
    OSInit();

    QSem = OSQCreate(&QMsgTbl[0], N_RESOURCES);
    for (i = 0; i < N_RESOURCES; i++) {
        OSQPost(QSem, (void *)1);
    }

    OSTaskCreate(Task1, ... , ... , ...);

    OSStart();
}

void Task1 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSQPend(&QSem, 0, &err);           /* 得到对资源的访问权 */

        /* 任务获得信号量,对资源进行访问 */

        OSMQPost(QSem, (void *)1);       /* 释放对资源的访问权 */
    }
}

```

第7章

内存管理

超星浏览器提醒您：
未经许可
禁止反编译
请尊重相关知识产权！

我们知道，在ANSI C中可以用malloc()和free()两个函数动态地分配内存和释放内存。但是，在嵌入式实时操作系统中，多次这样做会把原来很大的一块连续内存区域，逐渐地分割成许多非常小而且彼此又不相邻的内存区域，也就是内存碎片。由于这些碎片的大量存在，使得程序到后来连非常小的内存也分配不到。在4.2节的任务堆栈中，我们在讲用malloc()函数来分配堆栈时，曾经讨论过内存碎片的问题。另外，由于内存管理算法的原因，malloc()和free()函数执行时间是不确定的。

在μC/OS-II中，操作系统把连续的大块内存按分区来管理。每个分区中包含有整数个大小相同的内存块，如图7.1。利用这种机制，μC/OS-II对malloc()和free()函数进行了改进，使得它们可以分配和释放固定大小的内存块。这样一来，malloc()和free()函数的执行时间也是固定的了。

如图7.2，在一个系统中可以有多个内存分区。这样，用户的应用程序就可以从不同的内存分区中得到不同大小的内存块。但是，特定的内存块在释放时必须重新放回它以前所属于的内存分区。显然，采用这样的内存管理算法，上面的内存碎片问题就得到了解决。

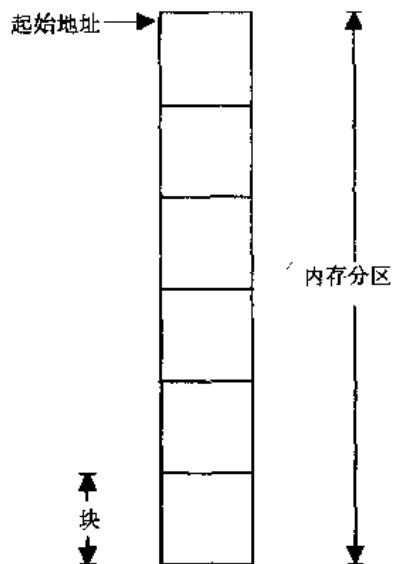
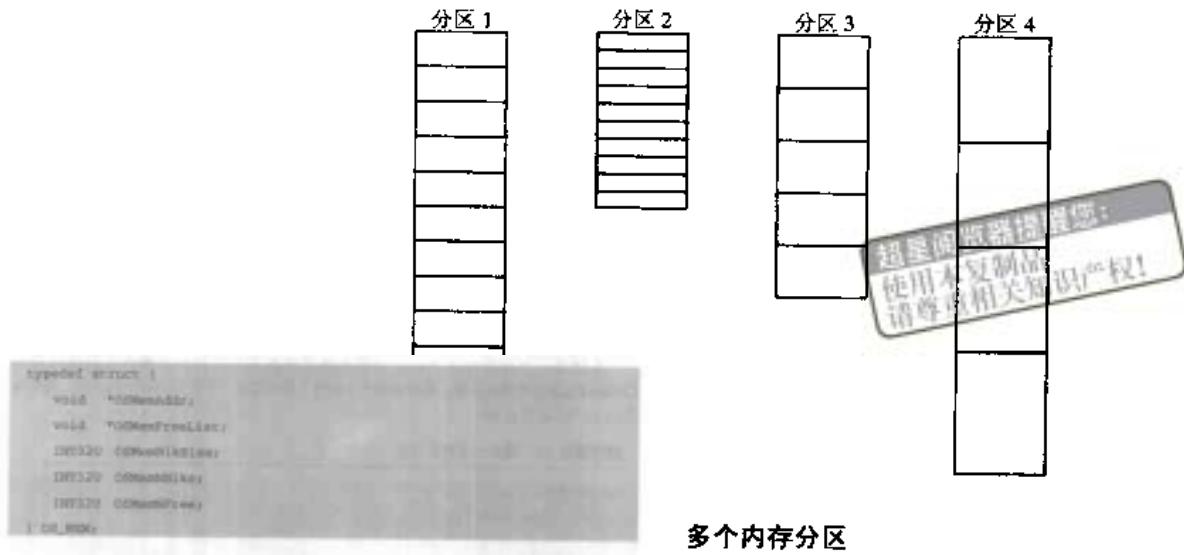


图7.1 内存分区



7.0 内存控制块

为了便于内存的管理，在μC/OS-II 中使用内存控制块（memory control block）的数据结构来跟踪每一个内存分区，系统中的每个内存分区都有它自己的内存控制块。程序清单 7.1 是内存控制块的定义。

程序清单 7.1 内存控制块的数据结构

.OSMemAddr 是指向内存分区起始地址的指针。它在建立内存分区（见 7.1 节）时初始化，在此之后就不能更改了。

.OSMemFreeList 是指向下一个空闲内存控制块或者下一个空闲的内存块的指针，具体含义要根据该内存分区是否已经建立来决定（见 7.1 节）。

.OSMemBlkSize 是内存分区中内存块的大小，是用户建立该内存分区时指定的（见 7.1 节）。

.OSMemNBlnks 是内存分区中总的内存块数量，也是用户建立该内存分区时指定的（见 7.1 节）。

.OSMemNFree 是内存分区中当前可以得空闲内存块数量。

如果要在 μC/OS-II 中使用内存管理，需要在 OS_CFG.H 文件中将开关量 OS_MEM_EN 设置为 1。这样 μC/OS-II 在启动时就会对内存管理器进行初始化[由 OSInit() 调用 OSMemInit() 实现]。该初始化主要建立一个图 7.3 所示的内存控制块链表，其中的常数 OS_MAX_MEM_PART（见文件 OS_CFG.H）定义了最大的内存分区数，该常数值至少应为 2。

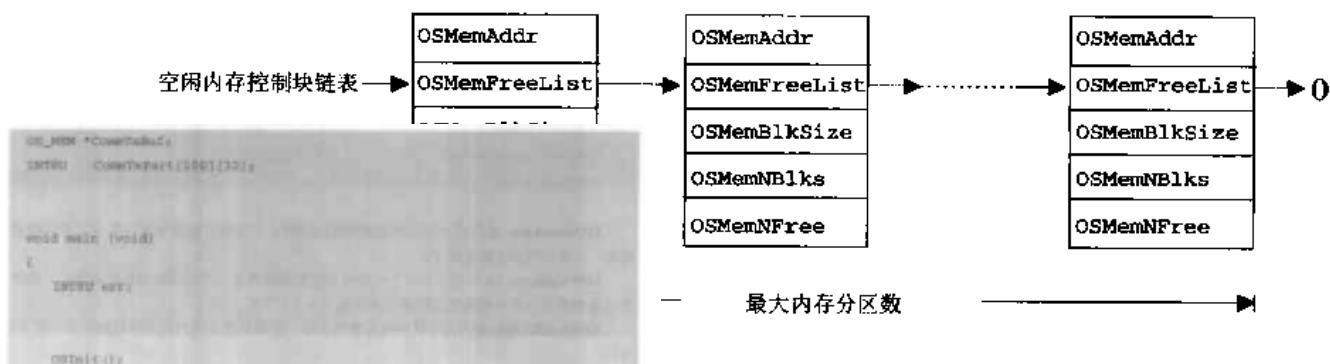


图7.3 空闲内存控制块链表

7.1 建立一个内存分区，OSMemCreate()

在使用一个内存分区之前，必须先建立该内存分区。这个操作可以通过调用 OSMemCreate() 函数来完成。程序清单 7.2 说明了如何建立一个含有 100 个内存块、每个内存块 32 字节的内存分区。

程序清单 7.2 建立一个内存分区

OSMemCreate()

超星阅读器提醒您：
使用本软件
请认真阅读
相关知识版权！

程序清单 7.3 是 OSMemCreate()函数的源代码。该函数共有 4 个参数：内存分区的起始
存块的字节数和一个指向错误信息代码的指针。如
一个 NULL 指针。否则，它将返回一个指向内存控
如 OSMemGet(), OSMemPut(), OSMemQuery()函

```
OS_Mem *OSMemCreate(VOID *addr, INT32U bytes, INT32U minsize, INT32U *err)
```

```
{ OS_Mem *mem;
```

```
INT32U *perr;
```

```
void **p1link;
```

```
INT32U i;
```

if (bytes < 2) {
 *err = OS_MEM_INVALID_BLOCK;
 return (OS_Mem *)0;

if (minsize < 1) {
 *err = OS_MEM_INVALID_BLOCK;

OSMemCreate()从系统中的空闲内存控制块中取得一个内存控制块[程序清单 7.3(3)]，该内
存控制块包含相应内存分区的运行信息。OSMemCreate()必须在有空闲内存控制块可用的情
况下才能建立一个内存分区[程序清单 7.3(4)]。在上述条件均得到满足时，所要建立的内存
分区内的所有内存块被链接成一个单向的链表[程序清单 7.3(5)]。然后，在对应的内存控制
块中填写相应的信息[程序清单 7.3(6)]。完成上述各动作后，OSMemCreate()返回指向该内
存块的指针。该指针在以后对内存块的操作中使用[程序清单 7.3(6)]。

程序清单 7.3 OSMemCreate()

```

    *err = OS_MEM_INVALID_SIZE;
    return ((OS_MEM *)0);
}

OS_ENTER_CRITICAL();
pmem = OSMemFreeList;                                (3)
if (OSMemFreeList != (OS_MEM *)0) {
    OSMemFreeList = (OS_MEM *)OSMemFreeList->OSMemFreeList;
}
OS_EXIT_CRITICAL();
if (pmem == (OS_MEM *)0) {                            (4)
    *err = OS_MEM_INVALID_PART;
    return ((OS_MEM *)0);
}
plink = (void **)addr;                                (5)
pblk = (INT8U *)addr + blksize;
for (i = 0; i < (nblk - 1); i++) {
    *plink = (void *)pblk;
    plink = (void **)pblk;
    pblk = pblk + blksize;
}
*plink = (void *)0;
OS_ENTER_CRITICAL();                                  (6)
pmem->OSMemAddr      = addr;
pmem->OSMemFreeList = addr;
pmem->OSMemNFree     = nblk;
pmem->OSMemNBkls     = nblk;
pmem->OSMemBlkSize   = blksize;
OS_EXIT_CRITICAL();
*err = OS_NO_ERR;
return (pmem);                                       (7)
}

```

图 7.4 是 OSMemCreate() 函数完成后，内存控制块及对应的内存分区和分区内的内存块之间的关系。在程序运行期间，经过多次的内存分配和释放后，同一分区内的各内存块之间的链接顺序会发生很大的变化。



7.2 分配一个内存块，OSMemGet()

应用程序可以调用 OSMemGet() 函数从已经建立的内存分区中申请一个内存块。该函数的惟一参数是指向特定内存分区的指针，该指针在建立内存分区时，由 OSMemCreate() 函数返回。显然，应用程序必须知道内存块的大小，并且在使用时不能超过该容量。例如，如果一个内存分区内的内存块为 32 字节，那么，应用程序最多只能使用该内存块中的 32 字节。当应用程序不再使用这个内存块后，必须及时把它释放，重新放入相应的内存分区中（见 7.3 节）。

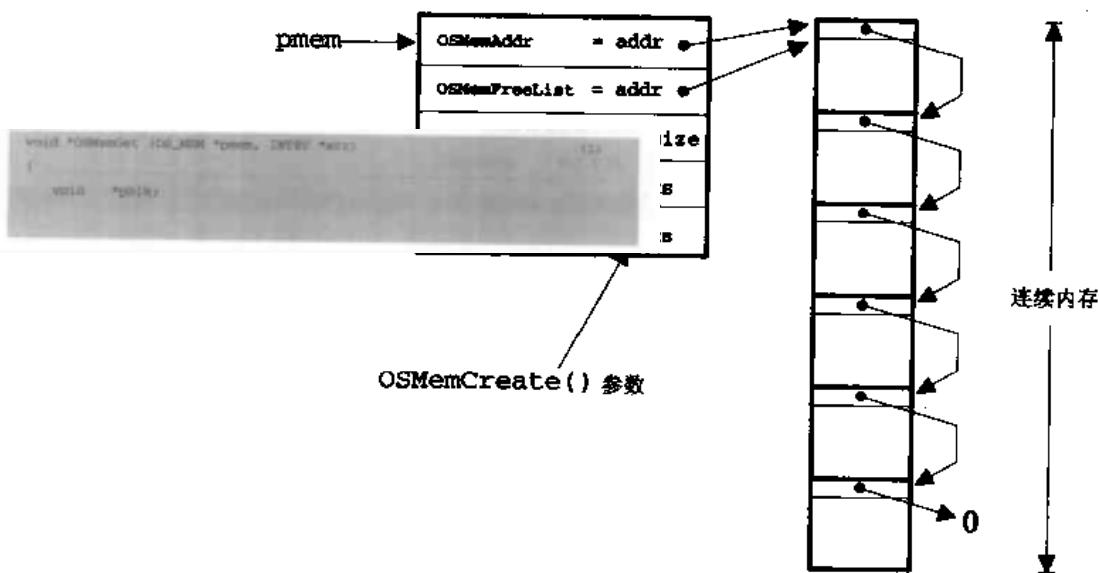


图7.4 OSMemCreate()

程序清单 7.4 是 OSMemGet() 函数的源代码。参数中的指针 pmem 指向用户希望从中分配内存块的内存分区[程序清单 7.4(1)]。OSMemGet() 首先检查内存分区中是否有空闲的内存块[程序清单 7.4(2)]。如果有，从空闲内存块链表中删除第一个内存块[程序清单 7.4(3)]，并对空闲内存块链表作相应的修改 [程序清单 7.4(4)]。这包括将链表头指针后移一个元素和空闲内存块数减1[程序清单 7.4(5)]。最后，返回指向被分配内存块的指针[程序清单 7.4(6)]。

程序清单 7.4 OSMemGet()

```
OS_ENTER_CRITICAL();
if (pmem->OMemFree == 0) {
    pblk = *pmem->OMemFreeList;
    pmem->OMemFreeList = *(void **)pblk;
    pmem->OMemFree++;
    OS_EXIT_CRITICAL();
    ret = OS_NO_ERROR;
    return (void *)pblk;
} else {
    OS_EXIT_CRITICAL();
    ret = OS_NO_FREE_BLOCK;
    return (void *)ret;
}
```

公开的实时嵌入式操作系统



```
ENTRY DOMAIN_OSMEM *pmem, void *pblk;
```

值得注意的是，用户可以在下断点分子程序中调用 OSMemGet()，因为在暂时没有内存块可用的情况下，OSMemGet()不会等待，而是马上返回 NULL 指针。

7.3 释放一个内存块，OSMemPut()

当用户应用程序不再使用一个内存块时，必须及时地把它释放并放回到相应的内存分区中。这个操作由 OSMemPut()函数完成。必须注意的是，OSMemPut()并不知道一个内存块是属于哪个内存分区的。例如，用户任务从一个包含 32 字节内存块的分区中分配了一个内存块，用完后，把它返还给了一个包含 120 字节内存块的内存分区。当用户应用程序下一次申请 120 字节分区中的一个内存块时，它会只得到 32 字节的可用空间，其他 88 字节属于其他的任务，这就有可能使系统崩溃。

程序清单 7.5 是 OSMemPut()函数的源代码。它的第一个参数 pmem 是指向内存控制块的指针，也即内存块属于的内存分区[程序清单 7.5(1)]。OSMemPut()首先检查内存分区是否已满[程序清单 7.5(2)]。如果已满，说明系统在分配和释放内存时出现了错误。如果未满，要释放的内存块被插入到该分区的空闲内存块链表中[程序清单 7.5(3)]。最后，将分区中空闲内存块总数加 1[程序清单 7.5(4)]。

程序清单 7.5 OSMemPut()

超星浏览器提醒您
使用本复制品
请尊重相关知识产权!

```
OS_XPTELR_CREATEAL()
```

- (1) if (pmem->OSMemFree == pmem->OSMemAlloc) {
- OS_EXITC_CREATEAL();
- return OS_MEM_FULL;

- (2) if (id == *pblk) {
- *pblk = pmem->OSMemFreeList;
- pmem->OSMemFreeList = pblk;
- pmem->OSMemFree++;
- OS_EXITC_CREATEAL();
- return OS_NO_ERROR;

```
typedef struct {
    void *pHeader; /* 指向内存分区首地址的指针 */
    void *pFreeList; /* 指向空闲内存块链表首地址的指针 */
    INT32U OSMemAlloc; /* 每个内存块所占的字节数 */
    INT32U OSMemFree; /* 空闲内存块总数 */
    INT32U OSMemUsed; /* 正在使用的内存块总数 */
} OS_MEM_DATA;
```

7.4 查询一个内存分区的状态，OSMemQuery()

在uC/OS-II 中，可以使用 `OSMemQuery()` 函数来查询一个特定内存分区的有关消息。

`OSMemQuery(OS_MemID *pMem, OS_MEM_DATA *pdata)`
存块的大小、可用内存块数和正在使用的内存块数等信息。所有这些信息都放在一个叫 `OS_MEM_DATA` 的数据结构中，如程序清单 7.6 所示。

程序清单 7.6 `OS_MEM_DATA` 数据结构

程序清单 7.7 是 `OSMemQuery()` 函数的源代码，它将指定内存分区的信息复制到 `OS_MEM_DATA` 定义的变量的对应域中。在此之前，代码首先禁止了外部中断，防止复制过程中某些变量值被修改[程序清单 7.7(1)]。由于正在使用的内存块数是由 `OS_MEM_DATA` 中的局部变量计算得到的，所以，可以放在临界区的外面。

程序清单 7.7 `OSMemQuery()`

```

OS_ENTER_CRITICAL();
pdata->OSAdd = pbase->OSMemAdd;
pdata->OSFreeList = pbase->OSMemFreeList;
pdata->OSMemSize = pbase->OSMemAllocSize;
pdata->OSMemAlloc = pbase->OSMemAlloc;
OS_EXIT_CRITICAL();
pdata->OSMem = pdata->OSMem - pdata->OSMemAlloc;
return (OS_NO_ERR);
}

```

公开的实时嵌入式操作系统

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

7.5 使用内存分区

图 7.5 中的例子演示了如何使用μC/OS-II 中的动态分配内存功能，并利用它进行消息传递（见第 6 章）。程序清单 7.8 是这个例子中两个任务的示意代码，其中一些重要代码的标号和图 7.5 中括号内用数字标识的动作是相对应的。

第一个任务读取并检查模拟输入量的值（如气压、温度、电压等），如果其超过了一定的阈值，就向第二个任务发送一个消息。该消息中含有时间信息、出错的通道号和错误代码等可以想像的任何可能的信息。

错误处理程序是该例子的中心。任何任务、中断服务子程序都可以向该任务发送出错消息。错误处理程序则负责在显示设备上显示出错信息，在磁盘上登记出错记录，或者启动另一个任务对错误进行纠正等。

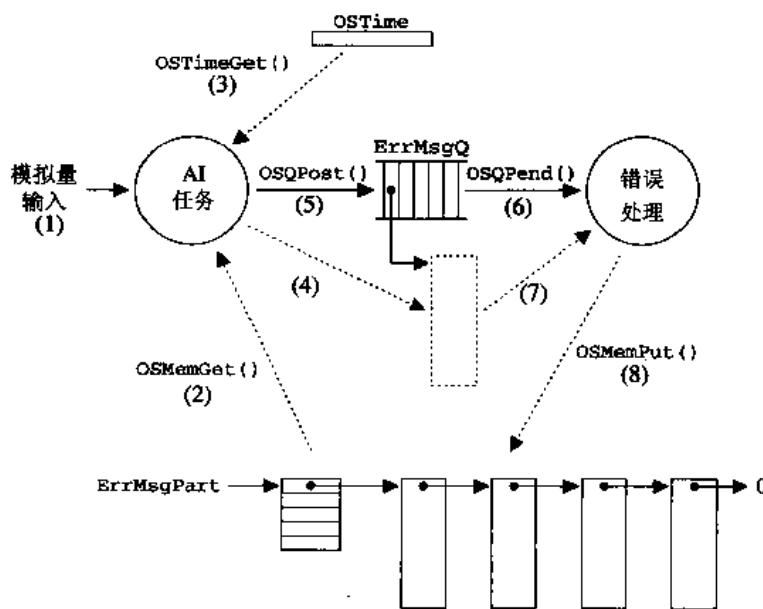


图7.5 使用动态内存分配

程序清单 7.8 内存分配的例子——扫描模拟量的输入和报告出错

```

AnalogInputTask()
{
    for (;;) {
        for (所有的模拟量都有输入) {
            读入模拟量输入值;                                (1)
            if (模拟量超过阈值) {
                得到一个内存块;                                (2)
                得到当前系统时间 (以时钟节拍为单位);          (3)
                将下列各项存入内存块:                          (4)
                    系统时间 (时间戳);
                    超过阈值的通道号;
                    错误代码;
                    错误等级;
                    等等
                向错误队列发送错误消息;                      (5)
                (一个指向包含上述各项的内存块的指针)
            }
        }
    }
}

```

延时任务，直到要再次对模拟量进行采样时为止；

```

ErrorHandlerTask()
{
    for (;;) {
        等待错误队列的消息;                                (6)
        (得到指向有关错误数据的内存块的指针)
        读入消息，并根据消息的内容执行相应的操作;      (7)
        将内存块放回到相应的内存分区中;                  (8)
    }
}

```

7.6 等待一个内存块

有时候，在内存分区暂时没有可用的空闲内存块的情况下，让一个申请内存块的任务等待也是有用的。但是，μC/OS-II 本身在内存管理上并不支持这项功能。如果确实需要，则可以通过为特定内存分区增加信号量的方法，实现这种功能（见 6.5 节）。应用程序为了申请分配内存块，首先要得到一个相应的信号量，然后才能调用 OSMemGet()函数。整个过程见程序清单 7.9。

程序代码首先定义了程序中使用到的各个变量[程序清单 7.9(1)]。该例中，直接使用数字定义了各个变量的大小，实际应用中，建议将这些数字定义成常数。在系统复位时，

```
OS_EVENT *SemaphorePrt;
OS_MEM *Partitionptr;
INT8U Partition[100][32];
OS_STK TaskStk[1000];
```

```
void main (void)
{
    INT8U i;
    OSInit();
}
```

[程序清单 7.9(2)]，然后用内存分区中总的内存块数来接着建立内存分区[程序清单 7.9(4)]和相应的要访问到此为止，我们对如何增加其他的任务也已经很清楚。使用动态内存块，就没有必要使用信号量了。这种情况下，除非我们要实现多任务共享内存，否则连内存分区都不需要。多任务执行从 OSStart()开始[程序清单 7.9(6)]。当一个任务运行时，只有在信号量有效时[程序清单 7.9(7)]，才有可能得到内存块[程序清单 7.9(8)]。一旦信号量有效了，就可以申请内存块并使用它，而没有必要对 OSSemPend()返回的错误代码进行检查。因为在这里，只有当一个内存块被其他任务释放并放回到内存分区后，μC/OS-II 才会返回到该任务去执行。同理，对 OSMemGet()返回的错误代码也无需做进一步的检查（一个任务能得以继续执行，则内存分区中至少有一个内存块是可用的）。当一个任务不再使用某内存块时，只需简单地将它释放并返还到内存分区[程序清单 7.9(9)]，并发送该信号量[程序清单 7.9(10)]。

程序清单 7.9 等待从一个内存分区中分配内存块

```

SemaphorePtr = OSSemCreate(100);          (3)
PartitionPtr = OSMemCreate(Partition, 100, 32, &err);    (4)

OSTaskCreate(Task, (void *)0, &TaskStk[999], &err);      (5)

OSStart();                                (6)
}

void Task (void *pdata)
{
    INT8U err;
    INT8U *pblock;

    for (;;) {
        OSSemPend(SemaphorePtr, 0, &err);           (7)
        pblock = OSMemGet(PartitionPtr, &err);       (8)

        /* 使用内存块 */

        OSMemPut(PartitionPtr, pblock);             (9)
        OSSemPost(SemaphorePtr);                   (10)
    }
}

```

首先，通过调用 OSMemCreate() 函数创建一个内存分区。接着调用 OSSemCreate() 函数创建一个信号量。然后在主函数中调用 OSStart() 函数启动系统。在任务函数中，先调用 OSSemPend() 函数挂起信号量，再调用 OSMemGet() 函数从分区中分配一个内存块。分配成功后，再调用 OSMemPut() 函数将内存块放回分区。最后调用 OSSemPost() 函数发布信号量。

通过上面的示例程序，读者可以知道，在嵌入式系统中，如果想要使用内存分区，那么首先需要创建一个信号量，然后通过信号量来控制对分区的访问。在任务函数中，先调用 OSSemPend() 函数挂起信号量，再调用 OSMemGet() 函数从分区中分配一个内存块。分配成功后，再调用 OSMemPut() 函数将内存块放回分区。最后调用 OSSemPost() 函数发布信号量。

第 8 章

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

移植μC/OS-II

这一章介绍如何将μC/OS-II移植到不同的处理器上。所谓移植，就是使一个实时内核能在某个微处理器或微控制器上运行。为了方便移植，大部分的μC/OS-II代码是用 C 语言写的；但仍需要用 C 和汇编语言写一些与处理器相关的代码，这是因为μC/OS-II在读写处理器寄存器时只能通过汇编语言来实现。由于μC/OS-II在设计时就已经充分考虑了可移植性，所以μC/OS-II的移植相对来说是比较容易的。如果已经有人在你使用的处理器上成功地移植了μC/OS-II，你也得到了相关代码，就不必看本章了。当然，本章介绍的内容将有助于用户了解μC/OS-II中与处理器相关的代码。

要使μC/OS-II正常运行，处理器必须满足以下要求：

1. 处理器的 C 编译器能产生可重入代码。
2. 用 C 语言就可以打开和关闭中断。
3. 处理器支持中断，并且能产生定时中断（通常在 10 至 100Hz 之间）。
4. 处理器支持能够容纳一定量数据（可能是几千字节）的硬件堆栈。
5. 处理器有将堆栈指针和其他 CPU 寄存器读出和存储到堆栈或内存中的指令。

像 Motorola 6805 系列的处理器不能满足上面的第 4 条和第 5 条要求，所以μC/OS-II 不能在这类处理器上运行。

图 8.1 说明了μC/OS-II 的结构以及它与硬件的关系。由于μC/OS-II 为自由软件，当用户用到μC/OS-II 时，有责任公开应用软件和μC/OS-II 的配置代码。本书和磁盘包含了所有与处理器无关的代码和 Intel 80x86 实模式下的与处理器相关的代码（C 编译器大模式下编译）。如果用户打算在其他处理器上使用μC/OS-II，最好能找到一个现成的移植实例，如果没有只好自己编写了。用户可以在正式的μC/OS-II 网站 www.uCOS-II.com 中查找一些移植实例。

如果用户理解了处理器和 C 编译器的技术细节，移植μC/OS-II的工作实际上是非常简单的。前提是你的处理器和编译器满足了μC/OS-II的要求，并且已经有了必要工具。移植工作包括以下几个内容：

- 用#define 设置一个常量的值 (OS_CPU.H);
- 声明 10 个数据类型 (OS_CPU.H);
- 用#define 声明三个宏 (OS_CPU.H);

- 用 C 语言编写六个简单的函数 (OS_CPU_C.C);
- 编写四个汇编语言函数 (OS_CPU_A.ASM)。

根据处理器的不同，一个移植实例可能需要编写或改写 50~300 行的代码，需要的时间从几个小时到一星期不等。

一旦代码移植结束，下一步工作就是测试。测试一个μC/OS-II 这样的多任务实时内核并不复杂。甚至可以在没有应用程序的情况下测试。换句话说，就是让内核自己测试自己。这样做有两个好处：第一，避免使本来就复杂的事情更加复杂；第二，如果出现问题，可以知道问题出在内核代码上而不是应用程序。刚开始的时候可以运行一些简单的任务和时钟节拍中断服务例程。一旦多任务调度成功地运行了，再添加应用程序的任务就是非常简单的工作了。

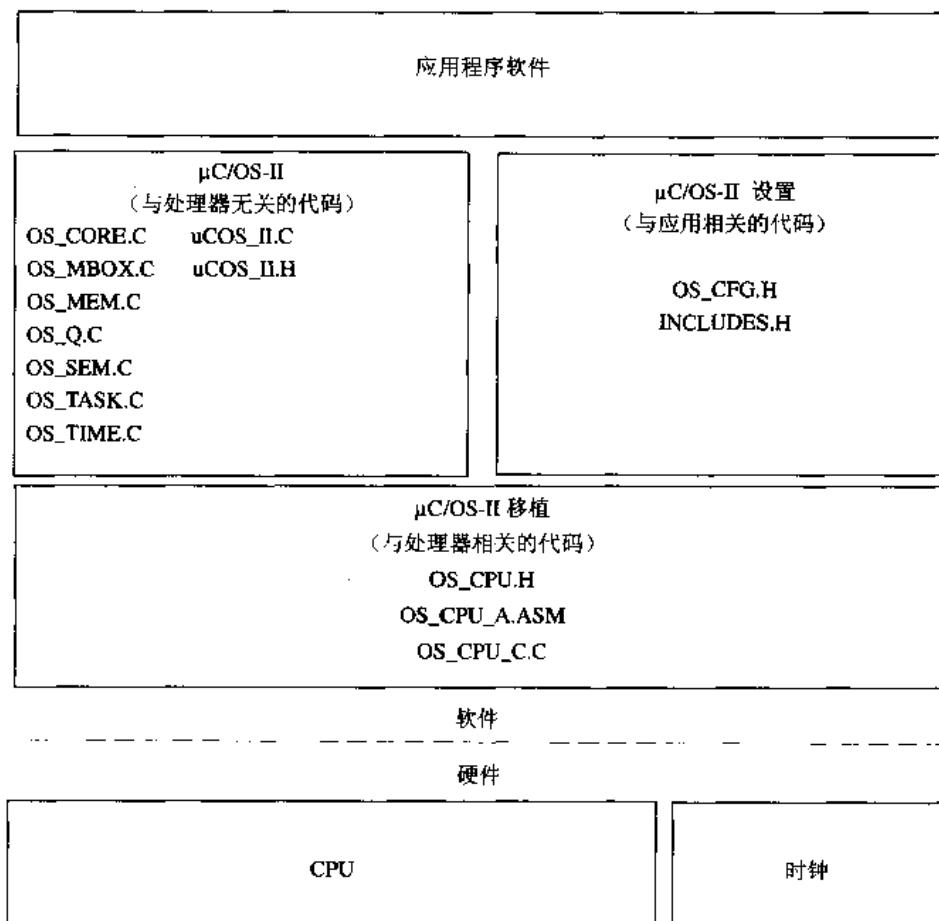


图 8.1 μC/OS-II 硬件和软件体系结构

8.0 开发工具

如前所述，移植μC/OS-II 需要一个 C 编译器，并且是针对用户用的 CPU 的。因为μC/OS-

II是一个占先式内核，用户只有通过 C 编译器来产生可重入代码；C 编译器还要支持汇编语言程序。绝大部分的 C 编译器都是为嵌入式系统设计的，它包括汇编器、连接器和定位器。连接器用来将不同的模块（编译过和汇编过的文件）连接成目标文件。定位器则允许用户将代码和数据放置在目标处理器的指定内存映射空间中。所用的 C 编译器还必须提供一个机制来从 C 中打开和关闭中断。一些编译器允许用户在 C 源代码中插入汇编语言。这就使得插入合适的处理器指令来允许和禁止中断变得非常容易了。还有一些编译器实际上包括了语言扩展功能，可以直接从 C 中允许和禁止中断。



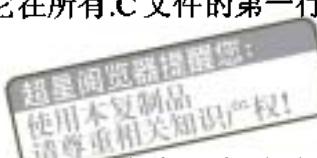
的安装程序，可在硬盘上安装μC/OS-II 和移植实例。我设计了一个连续的目录结构，使得用户更容易移植到其他处理器的移植实例，你可以考虑采取同样的方法。

移植实例应该放在用户的硬盘的\SOFTWARE\μC/OS-II 目录下。各个微处理器或微控制器的移植源代码必须在以下两个或三个文件中找到：OS_CPU.H, OS_CPU_C.C, OS_CPU_A.ASM。汇编语言文件 OS_CPU_A.ASM 是可选择的，因为某些 C 编译器允许用户在 C 语言中插入汇编语言，所以用户可以将所需的汇编语言代码直接放到 OS_CPU_C.C 中。放置移植实例的目录决定于用户所用的处理器，例如在下面的表中所示的放置不同移植实例的目录结构。注意，各个目录虽然针对完全不同的目标处理器，但都包括了相同的文件名。

#include "includes.h"

8.2 INCLUDES.H

在第1章中曾提到过，INCLUDES.H是一个头文件，它在所有.C文件的第一行被包含。



个.C文件不用分别去考虑它实际上需要哪些头文件。它可能会包含一些实际不相关的头文件。这意味着于它增强了代码的可移植性，所以我们还是决定使DES.H来增加自己的头文件，但是用户的头文件必

```
#ifndef OS_CPU_GLOBALS
#define OS_CPU_H
#include "os.h"
#include "os_cpu.h"

/* 定义全局常量
 * (与编译器无关)
 */

/* 定义全局类型
 * (与编译器无关)
 */

#ifndef OS_CPU_SIZEOF_VOID_P
#define OS_CPU_SIZEOF_VOID_P 4
#endif
#ifndef OS_CPU_SIZEOF_CHAR
#define OS_CPU_SIZEOF_CHAR 1
#endif
#ifndef OS_CPU_SIZEOF_SHORT
#define OS_CPU_SIZEOF_SHORT 2
#endif
#ifndef OS_CPU_SIZEOF_INT
#define OS_CPU_SIZEOF_INT 4
#endif
#ifndef OS_CPU_SIZEOF_LONG
#define OS_CPU_SIZEOF_LONG 4
#endif
#ifndef OS_CPU_SIZEOF_FLOAT
#define OS_CPU_SIZEOF_FLOAT 4
#endif
#ifndef OS_CPU_SIZEOF_DOUBLE
#define OS_CPU_SIZEOF_DOUBLE 8
#endif
#ifndef OS_CPU_SIZEOF_POINTER
#define OS_CPU_SIZEOF_POINTER 4
#endif

```

OS_CPU.H包括了用#define 定义的与处理器相关的常量，宏和类型定义。OS_CPU.H的大体结构如程序清单8.1所示。

程序清单8.1 OS_CPU.H

```

typedef unsigned int INT16U; /* 无符号 16 位整数 */
typedef signed int INT16S; /* 有符号 16 位整数 */
typedef unsigned long INT32U; /* 无符号 32 位整数 */
typedef signed long INT32S; /* 有符号 32 位整数 */
typedef float F32; /* 单精度浮点数 */
typedef double D64; /* 双精度浮点数 */

#ifndef _OS_STK_H_
#define _OS_STK_H_ /* 将堆栈 32 位数为 16 位 */

/* 与处理器相关的代码 */

#endif

#define OS_KRTCH_CRITICAL() 777 /* 禁止中断 */
#define OS_EXIT_CRITICAL() 777 /* 允许中断 */
#define OS_STK_GROWTH 1 /* 定义堆栈的增長方向：1-向左，-1-向右 */
#define OS_TASK_SWI 777

```

超星浏览器提醒您：
 使用本复制品
 请尊重相关知识产权！

8.3.1 与编译器相关的数据类型

因为不同的微处理器有不同的字长，所以μC/OS-II 的移植包括了一系列的类型定义以确保其可移植性。尤其是，μC/OS-II 代码从不使用 C 的 short,int 和 long 等数据类型，因为它们是与编译器相关的，不可移植。相反的，我定义的整型数据结构既是可移植的又是直观的[程序清单 8.1(2)]。为了方便，虽然μC/OS-II 不使用浮点数据，但我还是定义了浮点数据类型[程序清单 8.1(2)]。

例如，INT16U 数据类型总是代表 16 位的无符号整数。现在，μC/OS-II 和用户的应用程序就可以估计出声明为该数据类型的变量的数值范围是 0~65535。将μC/OS-II 移植到 32 位的处理器上也就意味着 INT16U 实际被声明为无符号短整型数据结构而不是无符号整型数据结构。但是，μC/OS-II 所处理的仍然是 INT16U。

用户必须将任务堆栈的数据类型告诉给μC/OS-II。这个过程是通过为 OS_STK 声明正确的 C 数据类型来完成的。如果用户的处理器上的堆栈成员是 32 位的，并且用户的编译文件指定整型为 32 位数，那么就应该将 OS_STK 声明为无符号整型数据类型。所有的任务堆栈都必须用 OS_STK 来声明数据类型。

用户所必须要做的就是查看编译器手册，并找到对应于μC/OS-II 的标准 C 数据类型。

8.3.2 OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()

与所有的实时内核一样，μC/OS-II需要先禁止中断再访问代码的临界区，并且在访问完毕后重新允许中断。这就使得μC/OS-II能够保护临界区代码免受多任务或中断服务例程

时内核公司提供的一个重要指标之一，因为它将影响到
虽然μC/OS-II尽量使中断禁止时间达到最短，但是
处理器结构和编译器产生的代码的质量。通常每个
中断，因此用户的C编译器必须要有一定的机制来
直接从C中执行这些操作。有些编译器能够允许用户在C源代码中插入汇编语言声明。这样就使得插入处理器指令来允许和禁止中断变得很容易了。其他一些编译器实际上包括了
禁止中断。为了隐藏编译器厂商提供的具体实现方法，μC/OS-II定义了两个宏来禁止和允许中断：OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL() [程序清单8.1(3)]。

执行这两个宏的第一个也是最简单的方法是在OS_ENTER_CRITICAL()中调用处理器指令来禁止中断，以及在OS_EXIT_CRITICAL()中调用允许中断指令。但是，在这个过程中还存在着小小的问题。如果用户在禁止中断的情况下调用μC/OS-II函数，在从μC/OS-II返回的时候，中断可能会变成是允许的了！如果用户禁止中断就表明用户想在从μC/OS-II函数返回的时候中断还是禁止的。在这种情况下，光靠这种执行方法可能是不够的。

执行OS_ENTER_CRITICAL()的第二个方法是先将中断禁止状态保存到堆栈中，然后禁止中断。而执行OS_EXIT_CRITICAL()的时候只是从堆栈中恢复中断状态。如果用这个方法的话，不管用户是在中断禁止还是允许的情况下调用μC/OS-II服务，在整个调用过程中都不会改变中断状态。如果用户在中断禁止的时候调用μC/OS-II服务，其实用户是在延长应用程序的中断响应时间。用户的应用程序还可以用OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()来保护代码的临界区。但是，用户在使用这种方法的时候还得十分小心，因为如果用户在调用象OSTimeDly()之类的服务之前就禁止中断，很有可能用户的应用程序会崩溃。发生这种情况的原因是任务被挂起直到时间期满，而中断是禁止的，因

而用户不可能获得节拍中断！很明显，所有的 PEND 调用都会涉及到这个问题，用户得十分小心。一个通用的方法是用户应该在中断允许的情况下调用 μC/OS-II 的系统服务！

看用户想牺牲些什么。如果用户并不关心在调用

μC/OS-II 服务后用户的应用程序厅中自动是否是允许的，那么用户应该选择第一种方法执行。

如果用户想在调用 μC/OS-II 服务过程中保持中断禁止状态，那么很明显用户应该选择第二

```
#define OS_ENTER_CRITICAL()    __asm CLI
```

```
#define OS_EXIT_CRITICAL()    __asm STI
```

```
#define OS_ENTER_CRITICAL()    __asm CLI
```

```
#define OS_EXIT_CRITICAL()    __asm STI
```

给用户举个例子吧，通过执行 STI 命令在 Intel 80186 上禁止中断，并用 CLI 命令来允许中断。用户可以用下面的方法来执行这两个宏：



CLI 和 STI 指令都会在两个时钟周期内被马上执行（总共为 4 个周期）。为了保持中断状态，用户需要用下面的方法来执行宏：

在这种情况下，OS_ENTER_CRITICAL() 需要 12 个时钟周期，而 OS_EXIT_CRITICAL() 需要另外的 8 个时钟周期（总共有 20 个周期）。这样，保持中断禁止状态要比简单的禁止/允许中断多花 16 个时钟周期的时间（至少在 80186 上是这样的）。当然，如果用户有一个速度比较快的处理器（如 Intel Pentium II），那么这两种方法的时间差别会很小。

8.3.3 OS_STK_GROWTH

绝大多数的微处理器和微控制器的堆栈是从上往下长的。但是某些处理器是用另外一种方式工作的。μC/OS-II 被设计成两种情况都可以处理，只要在结构常量 OS_STK_GROWTH [程序清单 8.1(4)] 中指定堆栈的生长方式（如下所示）就可以了。

置 OS_STK_GROWTH 为 0 表示堆栈从下往上长。

置 OS_STK_GROWTH 为 1 表示堆栈从上往下长。

8.3.4 OS_TASK_SW()

OS_TASK_SW() [程序清单 8.1(5)] 是一个宏，它是在 μC/OS-II 从低优先级任务切换到最高优先级任务时被调用的。OS_TASK_SW() 总是在任务级代码中被调用的。另一个函数 OSIntExit() 被用来在 ISR 使得更高优先级任务处于就绪状态时，执行任务切换功能。任务切换只是简单的将处理器寄存器保存到将被挂起的任务的堆栈中，并且将更高优先级的任

务从堆栈中恢复出来。

在μC/OS-II中，处于就绪状态的任务的堆栈结构看起来就像刚发生过中断并将所有的寄存器保存到堆栈中的情形一样。换句话说，μC/OS-II要运行处于就绪状态的任务必须要做的事就是将所有处理器寄存器从任务堆栈中恢复出来，并且执行中断的返回。为了切换任务可以通过执行 OS_TASK_SW()来产生中断。大部分的处理器会提供软中断或是陷阱（TRAP）指令来完成这个功能。ISR 或是陷阱处理函数（也叫做异常处理函数）的向量地址必须指向汇编语言函数 OSCTxSw()（参看 8.4.2）。

例如，在Intel或者AMD 80x86处理器上可以使用INT指令。但是中断处理向量需要指向 OSCTxSw()。Motorola 68HC11处理器使用的是SWI指令，同样，SWI的向量地址仍是 OSCTxSw()。还有，Motorola 680x0/CPU32可能会使用16个陷阱指令中的一个。当然，选中的陷阱向量地址还是 OSCTxSw()。

一些处理器如Zilog Z80并不提供软中断机制。在这种情况下，用户需要尽自己的所能将堆栈结构设置成与中断堆栈结构一样。OS_TASK_SW()只会简单的调用 OSCTxSw()而不是将某个向量指向 OSCTxSw()。μC/OS已经被移植到了Z80处理器上，μC/OS-II也同样可以。

8.4 OS_CPU_A.ASM

μC/OS-II的移植实例要求用户编写四个简单的汇编语言函数：

```
OSStartHighRdy()
OSCTxSw()
OSIntCtxSw()
OSTickISR()
```

如果用户的编译器支持插入汇编语言代码的话，用户就可以将所有与处理器相关的代码放到OS_CPU_C.C文件中，而不必再拥有一些分散的汇编语言文件。

8.4.1 OSStartHighRdy()

使就绪状态的任务开始运行的函数叫做OSStart()，如下所示。在用户调用OSStart()之前，用户必须至少已经建立了自己的一个任务[参看 OSTaskCreate()和 OSTaskCreateExt()]。OSStartHighRdy()假设OSTCBHighRdy指向的是优先级最高的任务的任务控制块。前面曾提到过，在μC/OS-II中处于就绪状态的任务的堆栈结构，看起来就像刚发生过中断并将所有的寄存器保存到堆栈中的情形一样。要想运行最高优先级任务，用户所要做的是将所有处理器寄存器按顺序从任务堆栈中恢复出来，并且执行中断的返回。为了简单一点，堆栈指针总是存储在任务控制块（即它的OS_TCB）的开头。换句话说，也就是要恢复的任务堆栈指针总是存储在OS_TCB的0偏址的内存单元中。

```
void OSStartHighRdy (void)
{
    /* 将用户对宏定义 OSTaskSwHook() 的
     * 调用映射到 OSTaskSwHook() 上。
     * 移除任务的悬挂指针用于恢复。
     * 释放锁住 = OSTCBHighRdy->OS_TASK_SW();
     * 从新任务的堆栈中恢复所有处理器寄存器。
     * 从中断堆栈返回。
}

```

注意，`OSStartHighRdy()`必须调用 `OSTaskSwHook()`，因为用户正在进行任务切换的部分工作——用户在恢复最高优先级任务的寄存器。而 `OSTaskSwHook()`可以通过检查 `OSRunning` 来知道是 `OSStartHighRdy()` 在调用它（`OSRunning` 为 `FALSE`）还是正常的任务切换并调用它（`OSRunning` 为 `TRUE`）。

任务级任务恢复之前和调用 `OSTaskSwHook()` 之后设置

```
void OSCtxSw (void)
{
    /* 将用户对宏定义 OSTaskSwHook() 的调用
     * 映射到 OSTaskSwHook() 上。
     * 将当前任务的堆栈指针保存到当前任务的 OS_TCB 中。
     * OSTCBCur->OSTaskSwHook = 挂起指针;
}

```

8.4.2 OSCtxSw()

如前所述，任务级的切换问题是通过发软中断命令或依靠处理器执行陷阱指令来完成的。中断服务例程，陷阱或异常处理例程的向量地址必须指向 `OSCtxSw()`。

如果当前任务调用μC/OS-II 提供的系统服务，并使得更高优先级任务处于就绪状态，μC/OS-II 就会借助上面提到的向量地址找到 `OSCtxSw()`。在系统服务调用的最后，μC/OS-II 会调用 `OSSched()`，并由此来推断当前任务不再是要运行的最重要的任务了。`OSSched()` 先将最高优先级任务的地址装载到 `OSTCBHighRdy` 中，再通过调用 `OS_TASK_SW()` 来执行软中断或陷阱指令。注意，变量 `OSTCBCur` 早就包含了指向当前任务的任务控制块（`OS_TCB`）的指针。软中断（或陷阱）指令会强制一些处理器寄存器（比如返回地址和处理器状态字）到当前任务的堆栈中，并使处理器执行 `OSCtxSw()`。`OSCtxSw()` 的原型如程序清单 8.2 所示。这些代码必须写在汇编语言中，因为用户不能直接从 C 中访问 CPU 寄存器。注意在 `OSCtxSw()` 和用户定义的函数 `OSTaskSwHook()` 的执行过程中，中断是禁止的。

程序清单 8.2 `OSCtxSw()` 的原型

```

移植用户是它的 OSIntCtxSw()。
OSIntCbxr = OSIntCbxmbr;
OSIntLcbr = OSIntLcmbr;
将所有需要恢复的任务的堆栈指针;
堆栈指针 = OSIntLcmbr - OSIntCbxmbr;
将所有处理器寄存器头断任务的堆栈中恢复起来;
执行中断返回指令;

```

8.4.3 OSIntCtxSw()

OSIntExit()通过调用 OSIntCtxSw()来在 ISR 中执行切换功能。因为 OSIntCtxSw()是在 ISR 中被调用的，所以可以断定所有的处理器寄存器都被正确地保存到了被中断的任务的堆栈之中。实际上除了我们需要的东西外，堆栈结构中还有其他的一些东西。OSIntCtxSw()必须要清理堆栈，这样被中断的任务的堆栈结构内容才能满足我们的需要。

要想了解 OSIntCtxSw()，用户可以看看μC/OS-II 调用该函数的过程。用户可以参看图 8.2 来帮助理解下面的描述。假定中断不能嵌套（即 ISR 不会被中断），中断是允许的，并且处理器正在执行任务级的代码。当中断来临的时候，处理器会结束当前的指令，识别中断并且初始化中断处理过程，包括将处理器的状态寄存器和返回被中断的任务的地址保存到堆栈中 [图 8.2(1)]。至于究竟哪些寄存器保存到了堆栈上，以及保存的顺序是怎样的，并不重要。

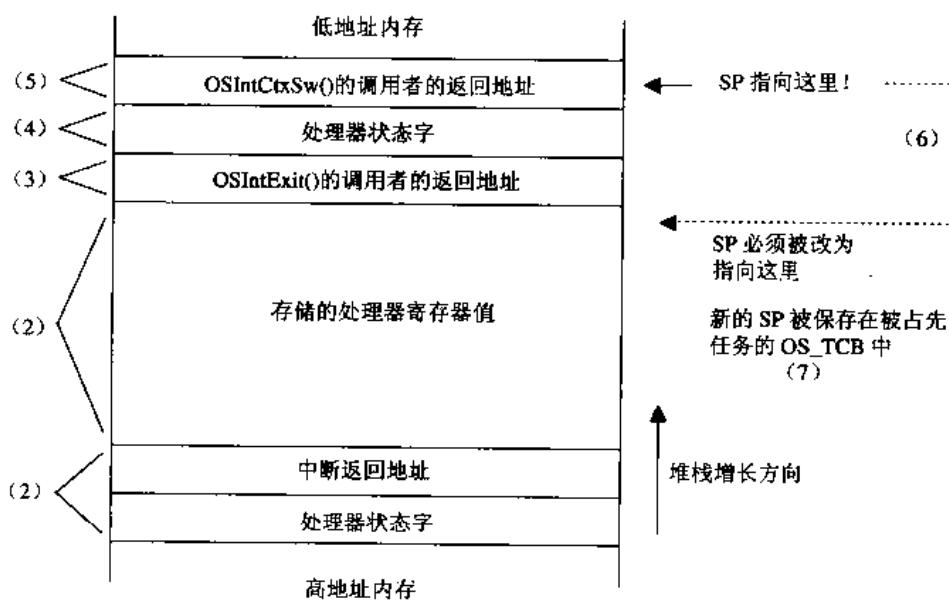


图8.2 在ISR执行过程中的堆栈内容

接着，CPU 会调用正确的 ISR。μC/OS-II 要求用户的 ISR 在开始时要保存剩下的处理

器寄存器[图 8.2(2)]。一旦寄存器保存好了，μC/OS-II 就要求用户或者调用 OSIntEnter()，或者将变量 OSIntNesting 加 1。在这个时候，被中断任务的堆栈中只包含了被中断任务的寄存器内容。现在，ISR 可以执行中断服务了。并且如果 ISR 发消息给任务[通过调用 OSMboxPost()或 OSQPost()]，恢复任务[通过调用 OSTaskResume()]，或者调用 OSTimeTick()或 OSTimeDlyResume()的话，有可能使更高优先级的任务处于就绪状态。

假设有一个更高优先级的任务处于就绪状态。μC/OS-II 要求用户的 ISR 在完成中断服务的时候调用 OSIntExit()。OSIntExit()会告诉 μC/OS-II 到了返回任务级代码的时间了。调用 OSIntExit()会导致调用者的返回地址被保存到被中断的任务的堆栈中[图 8.2(3)]。

OSIntExit()刚开始时会禁止中断，因为它需要执行临界区的代码。根据 OS_ENTER_CRITICAL()的不同执行过程（参看 8.3.2），处理器的状态寄存器会被保存到被中断的任务的堆栈中[图 8.2(4)]。OSIntExit()注意到由于有更高优先级的任务处于就绪状态，被中断的任务已经不再是要继续执行的任务了。在这种情况下，指针 OSTCBHighRdy 会被指向新任务的 OS_TCB，并且 OSIntExit()会调用 OSIntCtxSw()来执行任务切换。调用 OSIntCtxSw()也同样使返回地址被保存到被中断的任务的堆栈中[图 8.2(5)]。

在用户切换任务的时候，用户口相对某些项（[图 8.2(1)]和[图 8.2(2)]）保留在堆栈中，
[图 8.2(5)]。这是通过调整堆栈指针（加一个数在堆栈指针上）来完成的[图 8.2(6)]。加在堆栈指针上的数必须是明确的，而这个数主要依赖于移植的目标处理器（地址空间可能是 16, 32 或 64 位），所用的编译器，编译器选项，内存模式等等。另外，处理器状态字可能是 8, 16, 32 甚至 64 位宽，并且 OSIntExit()可能会分配局部变量。有些处理器允许用户直接增加常量到堆栈指针中，而有些则不允许。在后一种情况下，可以通过简单的执行一定数量的 pop（出栈）指令来实现相同的功能。一旦堆栈指针完成调整，新的堆栈指针会被保存到被切换出去的任务的 OS_TCB 中[图 8.2(7)]。

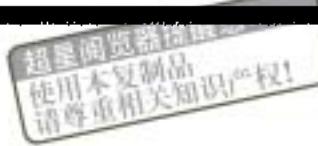
OSIntCtxSw()是 μC/OS-II（和 μC/OS）中唯一的与编译器相关的函数；在我收到的 E-mail 中，关于该函数的 E-mail 明显多于关于 μC/OS 其他方面的。如果在多次任务切换后用户的系统崩溃了，用户应该怀疑堆栈指针在 OSIntCtxSw()中是否正确调整。

OSIntCtxSw()的原型如程序清单 8.3 所示。这些代码必须用汇编语言编写，因为用户不能直接从 C 语言中访问 CPU 寄存器。如果用户的编译器支持插入汇编语言代码的话，用户就可以将 OSIntCtxSw()代码放到 OS_CPU_C.C 文件中，而不放到 OS_CPU_A.ASM 文件中。正如用户所看到的那样，除了第一行以外，OSIntCtxSw()的代码与 OSCtxSw()是一样的。这样在移植实例中，用户可以通过“跳转”到 OSCtxSw()中来减少 OSIntCtxSw()代码量。

程序清单 8.3 OSIntCtxSw()的原型

移植μC/OS-II
OSInit()，
OSInitCreate()函数中嵌入堆栈的多处内容：
将启动任务堆栈的设置移到启动任务的 os_tcb 中：
OSInitCreate->OSTaskStack0 = 堆栈指针；
调用用户定义的 OSTaskStack0()；
OSInitCreate = OSInitCreate();
OSInitCreate = OSInitCreate();
同时需要修改的任务的堆栈指针：
堆栈指针 = OSInitCreate->OSTaskStack0;
将所有忙于调用 OSInitCreate() 的任务从堆栈中恢复出来：
执行中断返回指令；

移植μC/OS-II



```
void main(void)
{
    /* 等待时钟节拍 */
    OSInit();
    /* 调用 μC/OS-II */
    /* 应用的初始化代码 ... */
    /* ... 调用 OSStartCreate() 建立第一个任务 */
}
```

μC/OS-II 要求用户提供一个时钟资源来实现时间的延时和期满功能。时钟节拍应该每秒钟发生 10~100 次。为了完成该任务，可以使用硬件时钟，也可以从交流电中获得 50/60Hz 的时钟频率。

用户必须在开始多任务调度后[即调用 OSStart()后]允许时钟节拍中断。换句话说，就是用户应该在 OSStart()运行后，μC/OS-II 启动运行的第一个任务中初始化节拍中断。通常所犯的错误是在调用 OSInit()和 OSStart()之间允许时钟节拍中断（如程序清单 8.4 所示）。

程序清单 8.4 在不正确的位置启动时钟节拍中断

超星阅览器提醒您：
 使用本复制品
 请尊重相关知识产权！

任务前时钟节拍中断就发生了。在这种情况下，程序也可能会崩溃。

所示。这些代码必须写在汇编语言中，因为用户不能通过单条指令来增加 OSIntNesting，如果用户的处理器可以通过单条指令来增加

OSIntNesting，那么用户就没必要调用 OSIntEnter()了。增加 OSIntNesting 要比通过函数调用和返回快得多。OSIntEnter()只增加 OSIntNesting，并且作为临界区代码中受到保护。

程序清单 8.5 时钟节拍 ISR 的原型

8.5 OS_CPU_C.C

μ C/OS-II 的移植实例要求用户编写 6 个简单的 C 函数：

```
OSTaskStkInit()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskSwHook()
OSTaskStatHook()
OSTimeTickHook()
```

惟一必要的函数是 OSTaskStkInit(), 其他 5 个函数必须得声明但没必要包含代码。

8.5.1 OSTaskStkInit()

OSTaskCreate() 和 OSTaskCreateExt() 通过调用 OSTaskStkInit() 来初始化任务的堆栈结构, 因此, 堆栈看起来就像刚发生过中断并将所有的寄存器保存到堆栈中的情形一样。图 8.3 显示了 OSTaskStkInit() 放到正被建立的任务堆栈中的东西。注意, 在这里我假定了堆栈是从上往下长的。下面的讨论同样适用于从下往上长的堆栈。

在用户建立任务的时候, 用户会传递任务的地址, pdata 指针, 任务的堆栈栈顶和任务的优先级给 OSTaskCreate() 和 OSTaskCreateExt()。虽然 OSTaskCreateExt() 还要求有其他的参数, 但这些参数在讨论 OSTaskStkInit() 的时候是无关紧要的。为了正确初始化堆栈结构, OSTaskStkInit() 只要求刚才提到的前三个参数和一个附加的选项, 这个选项只能在 OSTaskCreateExt() 中使用。

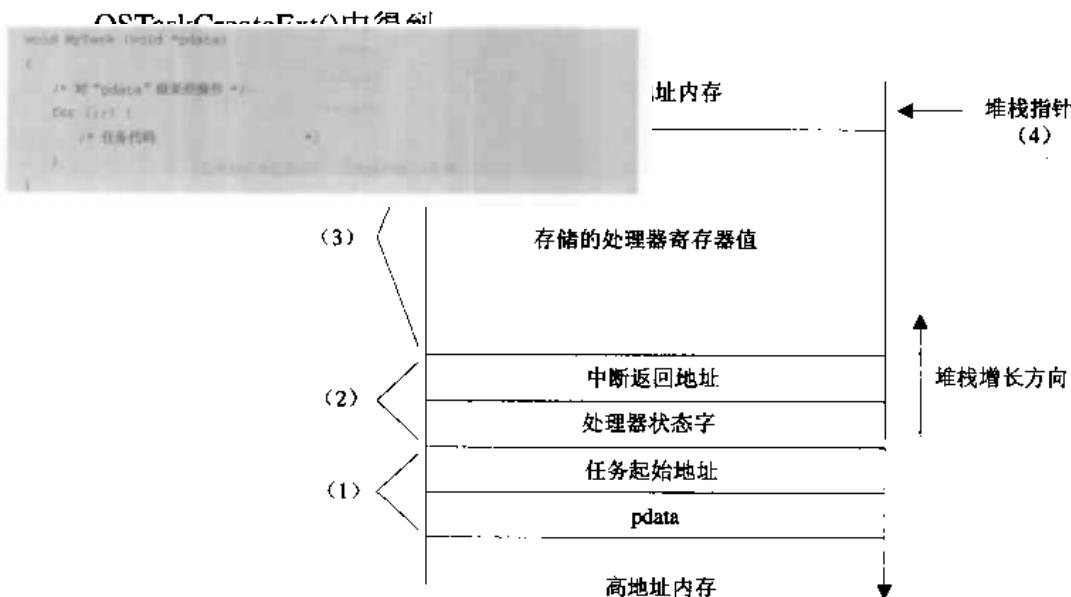


图 8.3 堆栈初始化 (pdata 通过堆栈传递)

回顾一下, 在 μC/OS-II 中, 无限循环的任务看起来就像其他的 C 函数一样。当任务开始被 μC/OS-II 执行时, 任务就会收到一个参数, 好像它被其他的任务调用一样。

如果我想从其他的函数中调用 MyTask(), C 编译器就会先将调用 MyTask() 的函数的返回地址保存到堆栈中，再将参数保存到堆栈中。实际上有些编译器会将 pdata 参数传至一个或多个寄存器中。在后面我会讨论这类情况。假定 pdata 会被编译器保存到堆栈中，OSTaskStkInit() 就会简单的模仿编译器的这种动作，将 pdata 保存到堆栈中[图 8.3(1)]。但是结果表明，与 C 函数调用不一样，调用者的返回地址是未知的。用户所拥有的是任务的开始地址，而不是调用该函数（任务）的函数的返回地址！事实上用户不必太在意这点，因为任务并不希望返回到其他函数中。

这时，用户需要将寄存器保存到堆栈中，当处理器发现并开始执行中断的时候，它会自动地完成该过程的。一些处理器会将所有的寄存器存入堆栈，而其他一些处理器只将部分寄存器存入堆栈。一般而言，处理器至少得将程序计数器的值（中断返回地址）和处理器的状态字存入堆栈[图 8.3(2)]。很明显，处理器是按一定的顺序将寄存器存入堆栈的，而用户在将寄存器存入堆栈的时候也就必须依照这一顺序。

接着，用户需要将剩下的处理器寄存器保存到堆栈中[图 8.3(3)]。保存的命令依赖于用户的处理器是否允许用户保存它们。有些处理器用一个或多个指令就可以马上将许多寄存器都保存起来。用户必须用特定的指令来完成这一过程。例如，Intel 80x86 使用 PUSHA 指令将 8 个寄存器保存到堆栈中。对 Motorola 68HC11 处理器而言，在中断响应期间，所有的寄存器都会按一定顺序自动的保存到堆栈中，所以在用户将寄存器存入堆栈的时候，也必须依照这一顺序。

现在是时候讨论这个问题了：如果用户的 C 编译器将 pdata 参数传递到寄存器中而不是堆栈中该作些什么？用户需要从编译器的文档中找到 pdata 存储在哪个寄存器中。pdata 的内容就会随着这个寄存器的存储被放置在堆栈中。

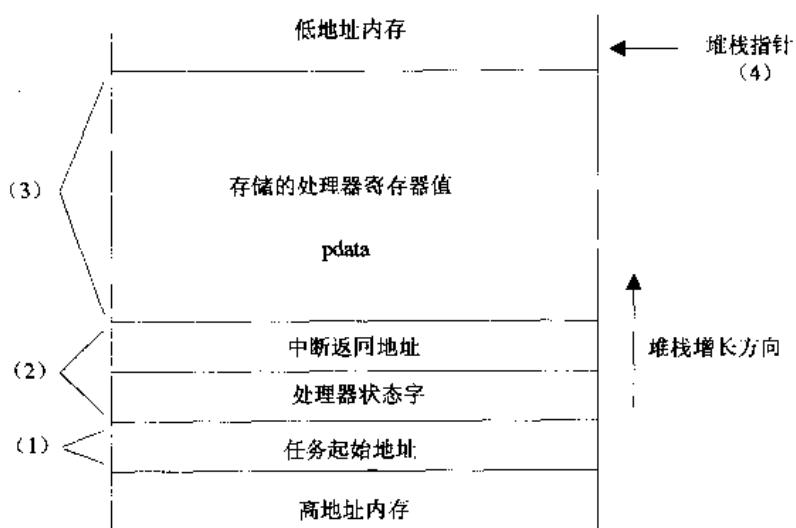


图 8.4 堆栈初始化（pdata 通过寄存器传递）



一旦用户初始化了堆栈，OSTaskStkInit()就需要返回堆栈指针所指的地址[图 8.3(4)]。OSTaskCreate()和OSTaskCreateExt()会获得该地址并将它保存到任务控制块(OS_TCB)中。处理器文档会告诉用户堆栈指针会指向下一个堆栈空闲位置，还是会指向最后存入数据的堆栈单元位置。例如，对 Intel 80x86 处理器而言，堆栈指针会指向最后存入数据的堆栈单元位置，而对 Motorola 68HC11 处理器而言，堆栈指针会指向下一个空闲的位置。

8.5.2 OSTaskCreateHook()

当用 OSTaskCreate()或 OSTaskCreateExt()建立任务的时候就会调用 OSTaskCreateHook()。该函数允许用户或使用用户的移植实例的用户扩展μC/OS-II 的功能。当μC/OS-II 设置完了自己的内部结构后，会在调用任务调度程序之前调用 OSTaskCreateHook()。该函数被调用的时候中断是禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

当 OSTaskCreateHook()被调用的时候，它会收到指向已建立任务的 OS_TCB 的指针，这样它就可以访问所有的结构成员了。当使用 OSTaskCreate()建立任务时，OSTaskCreateHook()的功能是有限的。但当用户使用 OSTaskCreateExt()建立任务时，用户会得到 OS_TCB 中的扩展指针(OSTCBExtPtr)，该指针可用来访问任务的附加数据，如浮点寄存器，MMU 寄存器，任务计数器的内容，以及调试信息。

只用当 OS_CFG.H 中的 OS_CPU_HOOKS_EN 被置为 1 时才会产生 OSTaskCreateHook()的代码。这样，使用用户的移植实例的用户可以在其他的文件中重新定义 hook 函数。

8.5.3 OSTaskDelHook()

当任务被删除的时候就会调用 OSTaskDelHook()。该函数在把任务从μC/OS-II 的内部任务链表中解开之前被调用。当 OSTaskDelHook()被调用的时候，它会收到指向正被删除任务的 OS_TCB 的指针，这样它就可以访问所有的结构成员了。OSTaskDelHook()可以用来检验 TCB 扩展是否被建立(一个非空指针)并进行一些清除操作。OSTaskDelHook()不返回任何值。

只用当 OS_CFG.H 中的 OS_CPU_HOOKS_EN 被置为 1 时才会产生 OSTaskDelHook()的代码。

8.5.4 OSTaskSwHook()

当发生任务切换的时候调用 OSTaskSwHook()。不管任务切换是通过 OSCtxSw()还是 OSIntCtxSw()来执行的都会调用该函数。OSTaskSwHook()可以直接访问 OSTCBCur 和 OSTCBHighRdy，因为它们是全局变量。OSTCBCur 指向被切换出去的任务的 OS_TCB，而 OSTCBHighRdy 指向新任务的 OS_TCB。注意在调用 OSTaskSwHook()期间中断一直是被禁止的。因为代码的多少会影响到中断的响应时间，所以用户应尽量使代码简化。

OSTaskSwHook()没有任何参数，也不返回任何值。

只用当 OS_CFG.H 中的 OS_CPU_HOOKS_EN 被置为 1 时才会产生 OSTaskSwHook() 的代码。

8.5.5 OSTaskStatHook()

OSTaskStatHook()每秒钟都会被 OSTaskStat()调用一次。用户可以用 OSTaskStatHook() 来扩展统计功能。例如，用户可以保持并显示每个任务的执行时间，每个任务所用的 CPU 份额，以及每个任务执行的频率等等。OSTaskStatHook()没有任何参数，也不返回任何值。

只用当 OS_CFG.H 中的 OS_CPU_HOOKS_EN 被置为 1 时才会产生 OSTaskStatHook() 的代码。

8.5.6 OSTimeTickHook()

OSTaskTimeHook()在每个时钟节拍都会被 OSTaskTick()调用。实际上，OSTaskTimeHook() 是在节拍被 μC/OS-II 真正处理，并通知用户的移植实例或应用程序之前被调用的。OSTaskTimeHook()没有任何参数，也不返回任何值。

只用当 OS_CFG.H 中的 OS_CPU_HOOKS_EN 被置为 1 时才会产生 OSTaskTimeHook() 的代码。

以下是这几个函数的详细说明。

OSTaskCreateHook()
void OSTaskCreateHook (OS_TCB *ptcb)

所属文件	调用者	开关量
OS_CPU_C.C	OSTaskCreate() and OSTaskCreateExt()	OS_CPU_HOOKS_EN

无论何时建立任务，在分配好和初始化 TCB 后就会调用该函数，当然任务的堆栈结构也已经初始化好了。OSTaskCreateHook()允许用户用自己的方式来扩展任务建立函数的功能。例如用户可以初始化和存储与任务相关的浮点寄存器，MMU 寄存器以及其他寄存器的内容。通常，用户可以存储用户的应用程序所分配的附加的内存信息。用户还可以通过使用 OSTaskCreateHook()来触发声波器或逻辑分析仪，以及设置断点。

参数

ptcb 是指向所创建任务的任务控制块的指针。

返回值

无



注意事项

的。因此用户应尽量减少该函数中的代码以缩短中

```
void OSTaskCreateExt(OS_TCB *ptcb)
{
    if (ptcb->OSTCBExtPtr != (void *)0)
    /* 重置浮点寄存器的内容... */
    /* ...TCB 扩展域... */
}
```

该例子假定了用户是用 OSTaskCreateExt()建立任务的，因为它希望在任务 OS_TCB 中有.OSTCBExtPtr 域，该域包含了指向浮点寄存器的指针。

```
void OSTaskDelHook(OS_TCB *ptcb)
```

OSTaskDelHook()

```
void OSTaskDelHook (OS_TCB *ptcb)
```

所属文件	调用者	开关量
OS_CPU_C.C	OSTaskDel()	OS_CPU_HOOKS_EN

当用户通过调用 OSTaskDel()来删除任务时都会调用该函数。这样用户就可以处理 OSTaskCreateHook()所分配的内存。OSTaskDelHook()就在 TCB 从 TCB 链中被移除前被调用。用户还可以通过使用 OSTaskDelHook()来触发声波器或逻辑分析仪，以及设置断点。

参数

ptcb 是指向所创建任务的任务控制块的指针。

返回值

无

注意事项

该函数在被调用的时候中断是禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

范例

OSTaskSwHook()**void OSTaskSwHook (void)**

所属文件	调用者	开关量
OS_CPU_C.C	OSCtxSw() and OSIntCtxSw()	OS_CPU_HOOKS_EN

当执行任务切换时都会调用该函数。全局变量 OSTCBHighRdy 指向得到 CPU 的任务的任务的 TCB。OSTaskSwHook()在保存好了任务的堆栈指针后马上被调用。用户可以用该函数来保

存或恢复浮点寄存器或 MMU 寄存器的内容，来得到任务执行时间的轨迹以及任务被切换进来的次数等等。

参数

无

返回值

无

注意事项

该函数在被调用的时候中断是禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

范例***OSTaskStatHook()*****void OSTaskStatHook (void)**

所属文件	调用者	开关量
OS_CPU_C.C	OSTaskStat()	OS_CPU_HOOKS_EN

该函数每秒钟都会被μC/OS-II的统计任务调用。OSTaskStatHook()允许用户加入自己的统计功能。

参数

```
void OSTaskStatHook (void)
```

/* 计算所有任务执行的总时间 */
/* 计算每个任务的执行时间在总时间内所占的百分比 */

无

注意事项

统计任务大概在调用 OSStart()后再过5秒开始执行。注意，当OS_TASK_STAT_EN或者OS_TASK_CREATE_EXT_EN被置为0时，该函数不会被调用。

范例



OSTimeTickHook()

void OSTimeTickHook (void)

所属文件	调用者	开关量
OS_CPU_C.C	OSTimeTick()	OS_CPU_HOOKS_EN

只要发生时钟节拍，该函数就会被OSTimeTick()调用。一旦进入OSTimeTick()就会马上调用OSTimeTickHook()以允许执行用户的应用程序中的与时间密切相关的代码。用户还可以通过使用该函数触发声波器或逻辑分析仪来调试，或者为仿真器设置断点。

参数

无

返回值

无

注意事项

OSTimeTick()通常是被ISR调用的，所以时钟节拍ISR的执行时间会因为用户在该函数中提供的代码而增加。当OSTimeTick()被调用的时候，中断可以是禁止的也可以是允许

```
void offtimeclockless (void)
{
    /* 被禁止数据 */
}
```

怎样进行的。如果中断是禁止的，该函数将会影响

范例



第9章



μC/OS-II 在 80x86 上的移植

本章将介绍如何将μC/OS-II 移植到 Intel 80x86 系列 CPU 上，本章所介绍的移植和代码都是针对 80x86 的实模式的，且编译器在大模式下编译和连接。本章的内容同样适用于下述 CPU：

80186

80286

80386

80486

Pentium

Pentium II

实际上，将要介绍的移植过程适用于所有与 80x86 兼容的 CPU，如 AMD，Cyrix，NEC（V-系列）等等。以 Intel 的 CPU 为例只是因其更典型。80x86 CPU 每年的产量有数百万以上，大部分用于个人计算机，但用于嵌入式系统的数量也在不断增加。最快的处理器（Pentium 系列）将在 2000 年达到 1G 的工作频率。

大部分支持 80x86（实模式）的 C 编译器都提供了不同的内存使用模式，每一种都有不同的内存组织方式，适用于不同规模的应用程序。在大模式下，应用程序和数据最大寻址空间为 1MB，程序指针为 32 位。下一节将介绍为什么 32 位指针只用到了其中的 20 位来寻址（1MB）。

本章所介绍的内容也适用于 8086 处理器，但由于 8086 没有 PUSH A 指令，移植的时候要用几条 PUSH 指令来代替。

图 9.1 显示了工作在实模式下的 80x86 处理器的编程模式。所有的寄存器都是 16 位，在任务切换时需要保存寄存器内容。

80x86 提供了一种特殊的机制，使得用 16 位寄存器可以寻址 1MB 地址空间，这就是存储器分段的方法。内存的物理地址用段地址寄存器和偏移量寄存器共同表示。计算方法是：段地址寄存器的内容左移 4 位（乘以 16），再加上偏移量寄存器（其他 6 个寄存器中的一个，AX，BP，SP，SI，DI 或 IP）的内容，产生可寻址 1MB 的 20 位物理地址。图 9.2 表明了寄存器是如何组合的。段寄存器可以指向一个内存块，称为一个段。一个 16 位的段寄存器可以表示 65536 个不同的段，因此可以寻址 1048576 字节。由于偏移量寄存器也是 16 位的，所以

单个段不能超过64K。实际操作中，应用程序是由许多小于64K的段组成的。

代码段寄存器（CS）指向当前程序运行的代码段起始，堆栈段寄存器（SS）指向程序堆栈段的起始，数据段寄存器指向程序数据区的起始，附加段寄存器（ES）指向一个附加数据存储区。每次CPU寻址的时候，段寄存器中的某一个会被自动选用，加上偏移量寄存器的内容作为物理地址。文献中会经常发现用段地址—偏移量表示地址的方法，例如1000:00FF表示物理地址0x100FF。

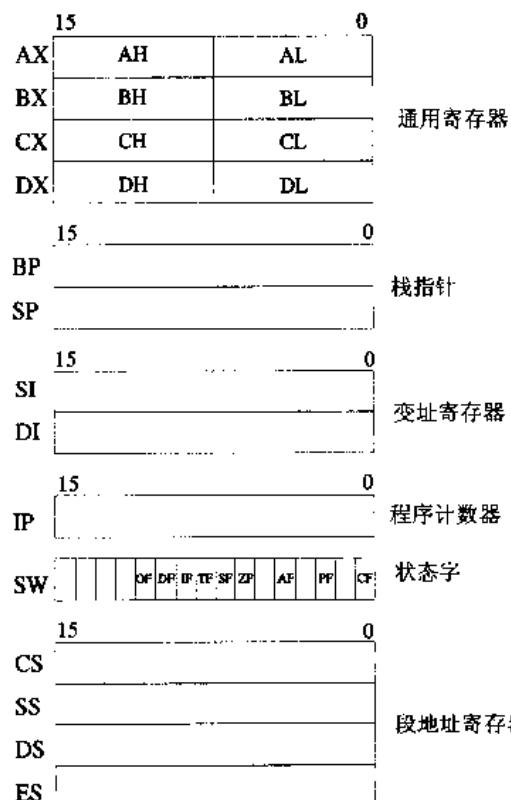


图9.1 80x86 实模式内部寄存器图

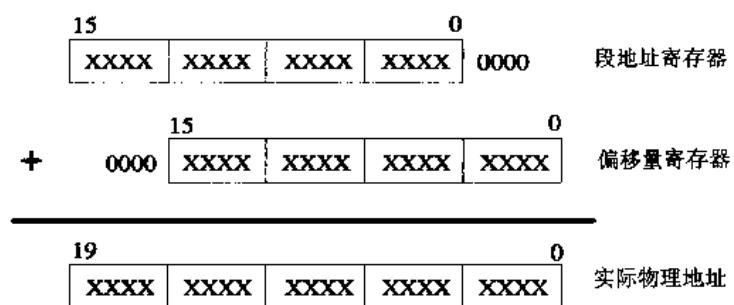


图 9.2 使用段寄存器和偏移量寄存器寻址

9.0 开发工具

笔者采用的是 Borland C/C++ V3.1 和 Borland Turbo Assembler 汇编器完成程序的移植和测试，它可以产生可重入的代码，同时支持在 C 程序中嵌入汇编语句。编译完成后，程序可在 PC 机上运行。本书代码的测试是在一台 Pentium II 计算机上完成的，操作系统是 Microsoft Windows 95。实际上编译器生成的是 DOS 可执行文件，在 Windows 的 DOS 窗口中运行。

只要你用的编译器可以产生实模式下的代码，移植工作就可以进行。如果开发环境不同，就只能麻烦你更改一下编译器和汇编器的设置了。

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <dos.h.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
```

把和硬件相关的，针对 Intel 80x86 的代码安装到
是 80x86 实模式，且在编译器大模式下编译的。

移植部分的代码可在下述文件中找到：OS_CPU.H, OS_CPU_C.C, 和 OS_CPU_A.ASM。

9.2 INCLUDES.H 文件

INCLUDES.H 是主头文件，在所有后缀名为.C 的文件的开始都包含 INCLUDES.H 文件。使用 INCLUDES.H 的好处是，所有的.C 文件都只包含一个头文件，程序简洁，可读性强。缺点是.C 文件可能会包含一些它并不需要的头文件，额外地增加编译时间。与其优点相比，多一些编译时间还是可以接受的。用户可以改写 INCLUDES.H 文件，增加自己的头文件，但必须加在文件末尾。程序清单 9.1 是为 80x86 编写的 INCLUDES.H 文件的内容。

程序清单 9.1 INCLUDES.H

```
#include "soc_swrm.h"
#include "os_cdg.h"
#include "soc_swrm_blockdev_smc_ipc.h"
#include "soc_swrm_usc_ll11uscoce1usc_ll1.h"
```

公开的实时嵌入式操作系统

感谢您：
使用本复印件
请尊重相关知识产权！

```
#define OS_CPSI_GLOBALS
#define OS_CPSI_EXIT
#define
#define OS_CPSI_EXIT return
#endif
/*
 *          目录类型
 *          (与编译器相关的内容)
 */
typedef unsigned short WORD8;
typedef unsigned char INT8U;           /* 无符号 8 位数 */
typedef signed char INT8S;             /* 带符号 8 位数 */
typedef unsigned int INT16U;            /* 无符号 16 位数 */
typedef signed int INT16S;              /* 带符号 16 位数 */
typedef unsigned long INT32U;           /* 无符号 32 位数 */
typedef signed long INT32S;             /* 带符号 32 位数 */
typedef float FP32;                   /* 单精度浮点数 */
typedef double FP64;                  /* 双精度浮点数 */

typedef unsigned int OS_STK;           /* 堆栈入口数据为 16 位 */
```

的常量，宏和结构体的定义。程序清单 9.2 是为

```

#define BYTE    INT8S      /* 以下定义的数据类型是为了与 uC/OS V1.xx 兼容 */
#define UBYTE   INT8U      /* 在 uC/OS-II 中并没有实际的用处 */
#define WORD    INT16S
#define UWORD   INT16U
#define LONG    INT32S
#define ULONG   INT32U

/*
*****
*          Intel 80x86 (实模式, 大模式编译)
*
*方法 #1: 用简单指令开关中断。
*注意, 用方法 1 关闭中断, 从调用函数返回后中断会重新打开!
*注意将文件 OS_CPU_A.ASM 中与 OSIntCtxSw() 相关的常量从 10 改到 8。
*
*
* 方法 #2: 关中断前保存中断被关闭的状态,
* 注意将文件 OS_CPU_A.ASM 中与 OSIntCtxSw() 相关的常量从 8 改到 10。
*
*
*****
*/
#define OS_CRITICAL_METHOD 2

#if OS_CRITICAL_METHOD == 1
#define OS_ENTER_CRITICAL() asm CLI      /* 关闭中断 */
#define OS_EXIT_CRITICAL()  asm STI      /* 打开中断 */
#endif

#if OS_CRITICAL_METHOD == 2
#define OS_ENTER_CRITICAL() asm {PUSHF; CLI} /* 关闭中断 */
#define OS_EXIT_CRITICAL()  asm POPF     /* 打开中断 */
#endif

```

```
28  
/*  
 * Intel 80x86 (实模式, 大段式编译)  
 */  
  
#define OS_STK_SIZE 1 /* 堆栈由高地址向低地址增长 */ (31)  
#define OS_PR 0x80 /* 中断向量 0x80 用于任务切换 */ (41)  
  
#define OS_TASK_SIZE 0x00 0x00 0x00  
  
/*  
 * 全局变量  
 */  
  
OS_CPU_WAIT_HOOK serviceHook; /* 为编程 tick 时钟中断肯定定义的计数器 */ (32)
```

9.3.1 数据类型

由于不同的处理器有不同的字长，μC/OS-II 的移植需要重新定义一系列的数据结构。使用 Borland C/C++ 编译器，整数（int）类型数据为 16 位，长整型（long）为 32 位。为了读者方便起见，尽管 μC/OS-II 中没有用到浮点类型的数，在源代码中笔者还是提供了浮点类型的定义。

由于在 80x86 实模式中堆栈都是按字进行操作的，没有字节操作，所以 Borland C/C++ 编译器中堆栈数据类型 OS_STK 声明为 16 位。所有的堆栈都必须用 OS_STK 声明。

9.3.2 代码临界区

与其他实时系统一样，μC/OS-II 在进入系统临界代码区之前要关闭中断，等到退出临界区后再打开。从而保护核心数据不被多任务环境下的其他任务或中断破坏。Borland C/C++ 支持嵌入汇编语句，所以加入关闭/打开中断的语句是很方便的。μC/OS-II 定义了两个宏用来关闭/打开中断：OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()。此处，笔者为用户提供两种开关中断的方法，如下所述的方法 1 和方法 2。作为一种测试，本书采用了方法 1。当然，你可以自由决定采用那种方法。

方法 1

第一种方法，也是最简单的方法，是直接将 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 定义为处理器的关闭 (CLI) 和打开 (STI) 中断指令。但这种方法有一个隐患，如果在关闭中断后调用 μC/OS-II 函数，当函数返回后，中断将被打开！严格意义上的关闭中断应该是执行 OS_ENTER_CRITICAL() 后中断始终是关闭的，方法 1 显然不满足要求。但方法 1 的最大优点是简单，执行速度快（只有一条指令），在此类操作频繁的时候更为突出。如果在任务中并不在意调用函数返回后是否被中断，推荐用户采用方法 1。此时需要将 OSIntCtxSw() 中的常量由 10 改到 8（见文件 OS_CPU_A.ASM）。

方法 2

执行 OS_ENTER_CRITICAL() 的第二种方法是先将中断关闭的状态保存到堆栈中，然后关闭中断。与之对应的 OS_EXIT_CRITICAL() 的操作是从堆栈中恢复中断状态。采用此方法，不管用户是在中断关闭还是允许的情况下调用 μC/OS-II 中的函数，在调用过程中都不会改变中断状态。如果用户在中断关闭的情况下调用 μC/OS-II 函数，其实是延长了中断响应时间。虽然 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 可以保护代码的临界区。但如此用法要小心，特别是在调用 OSTimeDly() 一类函数之前关闭了中断。此时任务将处于延时挂起状态，等待时钟中断，但此时时钟中断是禁止的！则系统可能会崩溃。很明显，所有的 PEND 调用都会涉及到这个问题，必须十分小心。所以建议用户调用 μC/OS-II 的系统函数之前打开中断。

9.3.3 堆栈增长方向

80x86 处理器的堆栈是由高地址向低地址方向增长的，所以常量 OS_STK_GROWTH 必须设置为 1 [程序清单 9.2(3)]。

9.3.4 OS_TASK_SW()

在 μC/OS-II 中，就绪任务的堆栈初始化应该模拟一次中断发生后的样子，堆栈中应该按进栈次序设置好各个寄存器的内容。OS_TASK_SW() 函数模拟一次中断过程，在中断返回的时候进行任务切换。80x86 提供了 256 个软中断源可供选用，中断服务程序 (ISR)（也称为异常处理过程）的入口指针必须指向汇编函数 OSCtxSw()（请参看文件 OS_CPU_A.ASM）。

由于笔者是在 PC 机上测试代码的，本章的代码用到了中断号 128 (0x80)，因为此中断号是提供给用户使用的 [程序清单 9.2(4)]（译注 1），类似的用户可用中断号还有 0x4B 到 0x5B，0x5D 到 0x66，或者 0x68 到 0x6F。如果用户用的不是 PC，而是其他嵌入式系统，如 80186

译注 1：PC 和操作系统会占用一部分中断资源。

处理器，用户可能有更多的中断资源可供选用。

9.3.5 时钟节拍的发生频率

实时系统中时钟节拍的发生频率应该设置为10到100 Hz。通常（但不是必需的）为了方便计算设为整数。不幸的是，在PC中，系统缺省的时钟节拍频率是18.20648Hz，这对于我们的计算和设置都不方便。本章中，笔者将更改PC的时钟节拍频率到200 Hz（间隔5ms）。一方面200 Hz近似18.20648Hz的11倍，可以经过11次延时再调用DOS中断；另一方面，在DOS中，有些操作要求时钟间隔为54.93ms，我们设定的间隔5ms也可以满足要求。如果你的PC机处理器是80386，时钟节拍最快也只能到200 Hz，而如果是Pentium II处理器，则达到200 Hz以上没有问题。

在文件OS_CPU.H的末尾声明了一个8位变量OSTickDOSCtr，将保存时钟节拍发生的次数，每发生11次，调用DOS的时钟节拍函数一次，从而实现与DOS时钟的同步。OSTickDOSCtr是专门为PC环境而声明的，如果在其他非PC的系统中运行μC/OS-II，就不用这种同步方法，直接设定时钟节拍发生频率就行了。

9.4 OS_CPU_A.ASM

μC/OS-II 的移植需要用户改写OS_CPU_A.ASM中的四个函数：

OSStartHighRdy()

OSCtxSw()

OSIntCtxSw()

OSTickISR()

9.4.1 OSStartHighRdy()

该函数由OSStart()函数调用，功能是运行优先级最高的就绪任务，在调用OSStart()之前，用户必须先调用OSInit()，并且已经至少创建了一个任务[请参考OSTaskCreate()和OSTaskCreateExt()函数]。OSStartHighRdy()默认指针OSTCBHighRdy指向优先级最高就绪任务的任务控制块(OS_TCB) [在这之前OSTCBHighRdy已由OSStart()设置好了]。图9.3给出了由函数OSTaskCreate()或OSTaskCreateExt()创建的任务的堆栈结构。很明显，OSTCBHighRdy->OSTCBSkPtr指向的是任务堆栈的顶端。

函数OSStartHighRdy()的代码见程序清单9.3。

为了启动任务，OSStartHighRdy()从任务控制块(OS_TCB) [程序清单9.3(1)]中找到指向堆栈的指针，然后运行POP DS [程序清单9.3(2)]，POP ES [程序清单9.3(3)]，POPA [程序清单9.3(4)]，和IRET[程序清单9.3(5)]指令。此处笔者将任务堆栈指针保存在任务控制块的开

头，这样使得堆栈指针的存取在汇编语言中更容易操作。

当执行了IRET指令后，CPU会从（SS:SP）指向的堆栈中恢复各个寄存器的值并执行中断前的指令。SS:SP+4指向传递给任务的参数pdata。

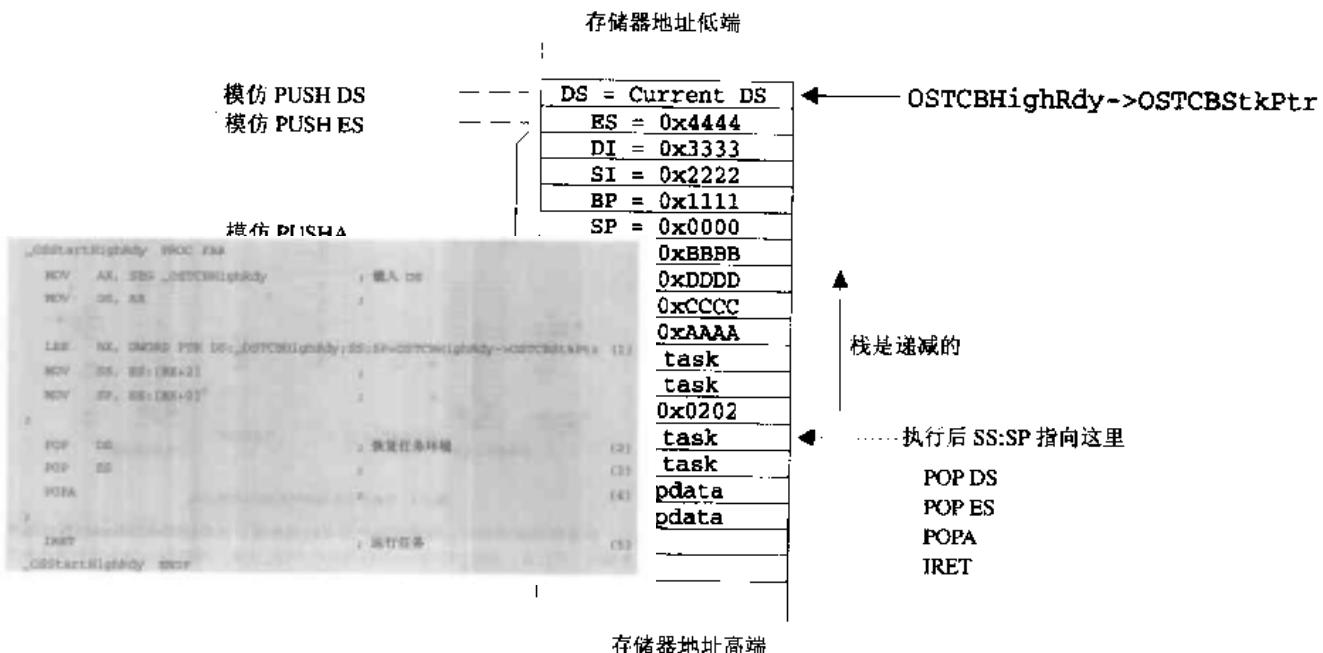


图 9.3 任务创立时的80x86堆栈结构

程序清单 9.3 OSStartHighRdy()

超星阅览器扫描
使用本复制品
请尊重相关知识产权!

9.4.2 OSCtxSw()

OSCtxSw()是一个任务级的任务切换函数[在任务中调用，区别于在中断程序中调用的OSIntCtxSw()]。在80x86系统上，它通过执行一条软中断的指令来实现任务切换。软中断向量指向OSCtxSw()。在μC/OS-II中，如果任务调用了某个函数，而该函数的执行结果可能造成系统任务重新调度（例如试图唤醒了一个优先级更高的任务），则在函数的末尾会调用OSSched()，如果OSSched()判断需要进行任务调度，会找到该任务控制块OS_TCB的地址，并将该地址拷贝到OSTCBHighRdy，然后通过宏OS_TASK_SW()执行软中断进行任务切换。注意到在此过程中，变量OSTCBCur始终包含一个指向当前运行任务OS_TCB的指针。程序清单9.4为OSCtxSw()的代码。

图9.4是任务被挂起或被唤醒时的堆栈结构。在80x86处理器上，任务调用OS_TASK_SW()执行软中断指令后[图9.4/程序清单9.4(1)]，先向堆栈中压入返回地址（段地址和偏移量），然后是状态字寄存器SW。紧接着用PUSHA [图9.4/程序清单9.4(2)]，PUSH ES [图9.4/程序清单9.4(3)]，和 PUSH DS [图9.4/程序清单9.4(4)]保存任务运行环境。最后用OSCtxSw()在任务OS_TCB中保存SS和SP寄存器。

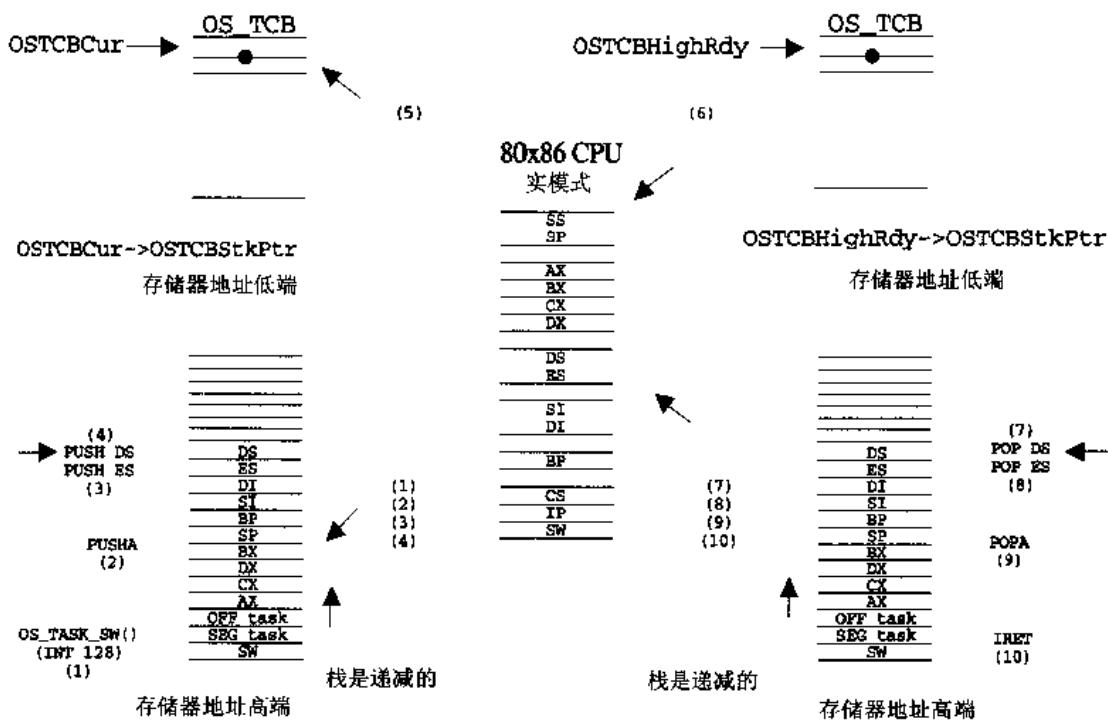


图 9.4 任务级任务切换时的80x86堆栈结构

任务环境保存完后，将调用用户定义的对外接口函数OSTaskSwHook() [程序清单9.4(6)]。请注意，此时OSTCBCur指向当前任务OS_TCB，OSTCBHighRdy指向新任务的

可以访问这两个任务的OS_TCB。如果不使用对外
接口，加快任务切换的速度。

```

;DISCHRM PROC FAR
    PUSHA          ; 保护当前任务环境      (1)
    PUSH BX
    PUSH DX
    POPA           ; (4)

    MOV AX, SS: _OSTCBCur      ; 读入 OS_TCBCur 地址      (5)
    MOV DS, AX
    MOV ES, DS+2, DS
    MOV DS, DS+01, DS

    CALL FAR PTR _OSTaskSwHook      (6)
    MOV AX, WORD PTR DS:_OSTCBHighRdy      ; OSTCBHighRdy->OSTCBHighRdy      (7)
    MOV BX, WORD PTR DS:_OSTCBHighRdy
    MOV WORD PTR DS:_OSTCBCur+2, AX
    MOV WORD PTR DS:_OSTCBCur, BX

    MOV AX, WORD PTR DS:_OSPriority      ; OSPriority = currentPriority      (8)
    MOV BYTE PTR DS:_OSPriority, AX

    LEA BX, SS:WORD PTR DS:_OSTCBHighRdy      ; DS:BX = OSTCBHighRdy->OSTCBHighRdy      (9)
    MOV DS, BX+2
    MOV BP, BX+00

    POP DX          ; 读入新任务的 cur 地址      (10)
    POP BX
    POPA           ; (11)

    INT3           ; 切换新任务      (12)
    DISCHRM ENDP

```

从对外接口函数OSTaskSwHook()返回后，由于任务的更替，变量OSTCBHighRdy被拷贝到OSTCBCur中[程序清单9.4(7)]，同样，OSPriority被拷贝到OSPriority中[程序清单

9.4(8)]。OSCtxSw()将载入新任务的CPU环境，首先从新任务OS_TCB中取出SS和SP寄存器的值[图9.4(6)/程序清单9.4(9)]，然后运行POP DS [图9.4(7)/程序清单9.4(10)]，POP ES [图9.4(8)/程序清单9.4(11)]，POPA [图9.4(9)/程序清单9.4(12)]取出其他寄存器的值，最后用中断返回指令IRET [图9.4(10) / 程序清单9.4(13)]完成任务切换。

需要注意的是在运行OSCtxSw()和OSTaskSwHook()函数期间，中断是禁止的。

9.4.3 OSIntCtxSw()

```
OSINTCTXSW PROC FAR
    ; 调用 OSIntCtxSw 的代码
    ADD SP, 8 ; 为 OS_CRITICAL_METHOD 为 1 释放真正释还原为代码, 参见 OS_CPV.h
    ADD SP, 10 ; 为 OS_CRITICAL_METHOD 为 1 释放真正释还原为代码, 参见 OS_CPV.h
    MOV AX, WORD PTR DS:_OSINTCWSW ; 载入 DS
    MOV DS, AX

    LES BX, DWORD PTR DS:_OSINTCWSW ; OSINTCWSW->OSINTCWSW+8=BP
    MOV BX,[BX+2], BX
    MOV BX,[BX+2], BP

    CALL FAR PTR _OSTaskSwitch
    ; 与 OSIntCtxSw() 的代码相同, 不同之处是, 第一, 由于中断服务程序没有堆栈处理器 (没有PUSHA, PUSH ES或PUSH DS); 第二, 去掉堆栈中一些不需要的内容, 以便堆栈中只包含任务切换过程。
    MOV AX, WORD PTR DS:_OSINTCWSW+2 ; OSINTCWSW -> OSINTCWSW+2
    MOV DS, WORD PTR DS:_OSINTCWSW+2
    MOV WORD PTR DS:_OSINTCWSW+2, AX
    MOV WORD PTR DS:_OSINTCWSW, BX
    RET
```

程序清单 9.5 OSIntCtxSw()

可能会引起任务切换，在中断服务程序的最后会调用OSIntCtxSw()。所以当需要进行任务切换，将调用OSIntCtxSw()。所以函数。由于在调用OSIntCtxSw()之前已经发生了中断，所以保存在被中断任务的堆栈中了。

OSIntCtxSw()的代码相同，不同之处是，第一，由于中断服务程序没有堆栈处理器（没有PUSHA, PUSH ES或PUSH DS）；第二，去掉了堆栈中一些不需要的内容，以便堆栈中只包含任务切换过程。

超星阅览器提醒您：
 使用本复制品
 请尊重相关知识产权！

```

MOV AL, PTR PTR DS:OSTCBHighRdy + OffPrioCur + OffPriority
MOV BX, PTR DS:OSTCBHighRdy, AL

LEA BX, PTR DS:OSTCBHighRdy + CurIP = OSTCBHighRdy->OSTCBIP
MOV BX, BX+BX+2
MOV BX, BX+BX

POP DS
      ; Load new task's context
POP ES
POPA

IRET
      ; Return to new task
  
```

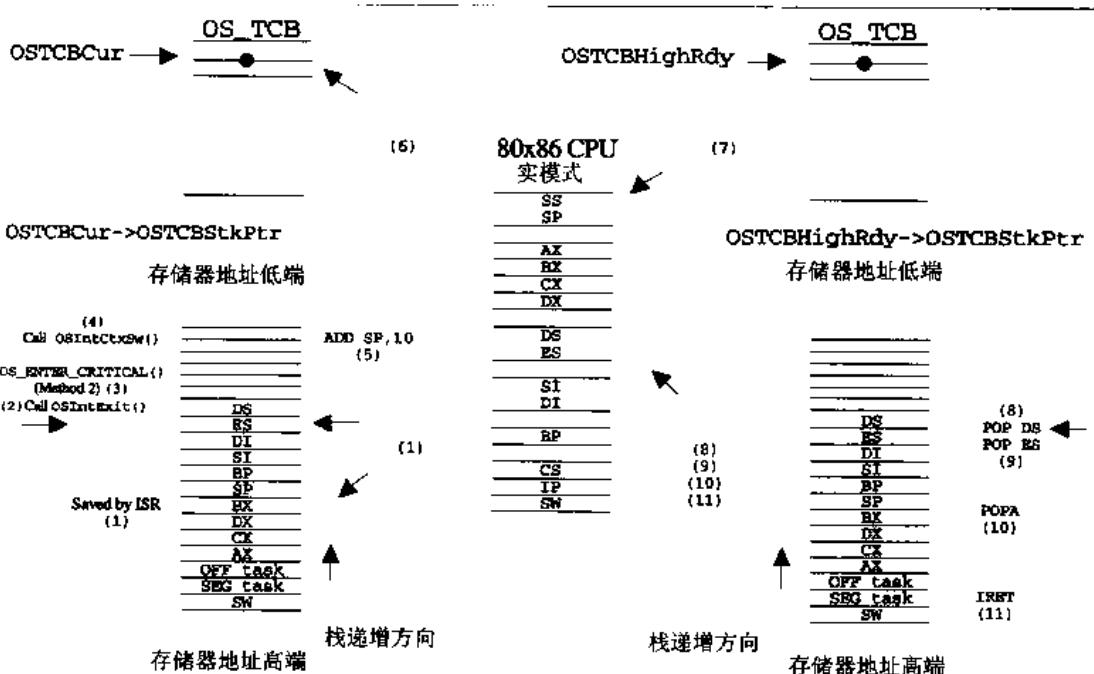


图 9.5 中断级任务切换时的 80x86 堆栈结构

当中断发生后，CPU在完成当前指令后，进入中断处理过程。首先是保存现场，将返回地址压入当前任务堆栈，然后保存状态寄存器的内容。接下来CPU从中断向量处找到中断服务程序的入口地址，运行中断服务程序。在μC/OS-II中，要求用户的中断服务程序在

开头保存CPU其他寄存器的内容[图9.5(1)]。此后，用户必须调用OSIntEnter()或者把全局变量OSIntNesting加1。此时，被中断任务的堆栈中保存了任务的全部运行环境。在中断服务程序中，有可能引起任务就绪状态的改变而需要任务切换，例如调用了OSMboxPost()，OSQPostFront()，OSQPost()或试图唤醒一个优先级更高的任务[调用OSTaskResume()]，还可能调用OSTimeTick()，OSTimeDlyResume()等等。

μC/OS-II要求用户在中断服务程序的末尾调用OSIntExit()，以检查任务就绪状态。在调用OSIntExit()后，返回地址会压入堆栈中[图9.5(2)]。

进入OSIntExit()后，由于要访问临界代码区，首先关闭中断。由于OS_ENTER_CRITICAL()可能有不同的操作（见9.3.2节），状态寄存器SW的内容有可能被压入堆栈[图9.5(3)]。如果确实要进行任务切换，指针OSTCBHighRdy将指向新的就绪任务的OS_TCB，OSIntExit()会调用OSIntCtxSw()完成任务切换。注意，调用OSIntCtxSw()会再一次在堆栈中保存返回地址[图9.5(4)]。在进行任务切换的时候，我们希望堆栈中只保留一次中断发生的任务环境[如图9.5(1)]，而忽略掉由于函数嵌套调用而压入的一系列返回地址[图9.5(2),(3),(4)]。忽略的方法也很简单，只要把堆栈指针加一个固定的值就可以了[图9.5(5)/程序清单9.5(1)]。如果用方法2实现OS_ENTER_CRITICAL()，这个固定值是10；如果用方法1，则是8。实际操作中还与编译器以及编译模式有关。例如，有些编译器会为OSIntExit()在堆栈中分配临时变量，这都会影响具体占用堆栈的大小，这一点需要提醒用户注意。

一旦堆栈指针重新定位后，就被保存到将要被挂起的任务的OS_TCB中[图9.5(6) / 程序清单9.5(2)]。在μC/OS-II中（包括μC/OS），OSIntCtxSw()是惟一一个与编译器相关的函数，也是用户间的最多的。如果你的系统移植后运行一段时间后就会死机，就应该怀疑是OSIntCtxSw()中堆栈指针重新定位的问题。

在当前任务的现场保存完毕后，用户定义的对外接口函数OSTaskSwHook()会被调用[程序清单9.5(3)]。注意到OSTCBCur指向当前任务的OS_TCB，OSTCBHighRdy指向新任务的OS_TCB。在函数OSTaskSwHook()中用户可以访问这两个任务的OS_TCB。如果不使用对外接口函数，请在头文件中关闭相应的开关选项，提高任务切换的速度。

从对外接口函数OSTaskSwHook()返回后，由于任务的更替，变量OSTCBHighRdy被拷贝到OSTCBCur中[程序清单9.5(4)]，同样，OSPrioHighRdy被拷贝到OSPrioCur中[程序清单9.5(5)]。此时，OSIntCtxSw()将载入新任务的CPU环境，首先从新任务OS_TCB中取出SS和SP寄存器的值[图9.5(7) / 程序清单9.5(6)]，然后运行POP DS [图9.5(8) / 程序清单9.5(7)]，POP ES [图9.5(9) / 程序清单9.5(8)]，POPA[图9.5(10)/程序清单9.5(9)]取出其他寄存器的值，最后用中断返回指令IRET [图9.5(11) / 程序清单9.5(10)]完成任务切换。

需要注意的是在运行OSIntCtxSw()和用户定义的OSTaskSwHook()函数期间，中断是禁止的。

9.4.4 OSTickISR()

在9.3.5节中，我们已经提到过实时系统中时钟节拍发生频率的问题，应该在10到100Hz之间。但由于PC环境的特殊性，时钟节拍由硬件产生，间隔54.93ms（18.20648Hz）。我们将时钟节拍频率设为200Hz。PC时钟节拍的中断向量为0x08，μC/OS-II将此向量截取，指向了μC/OS的中断服务函数OSTickISR()，而原先的中断向量保存在中断129（0x81）中。为满足DOS的需要，原先的中断服务还是每隔54.93ms（实际上还要短些）调用一次。图9.6为安装μC/OS-II前后的中断向量表。

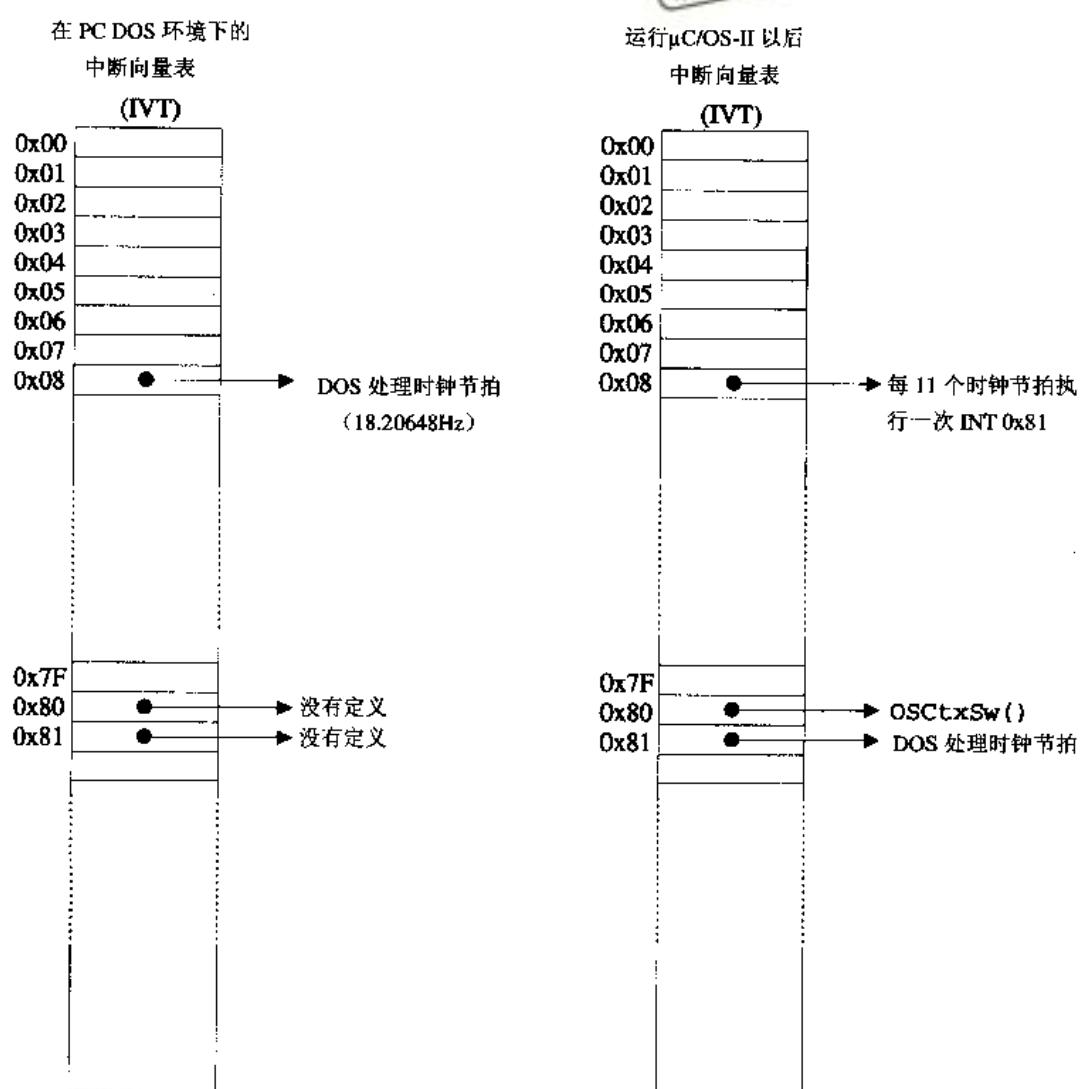


图9.6 PC中断向量表(IVT)

在μC/OS-II中，当调用OSStart()启动多任务环境后，时钟中断的作用是非常重要的。但在PC环境下，启动μC/OS-II之前就已经有时钟中断发生了，实际上我们希望在μC/OS-II初

初始化完成之后再发生时钟中断，调用OSTickISR()。与此相关的有下述过程。

PC_DOSSaveReturn() 函数（参看PC.C）：该函数由main()调用，任务是取得DOS下时钟中断向量，并将其保存在0x81中。

main() 函数：

设定中断向量0x80指向任务切换函数OSCtxSw();

至少创立一个任务；

当初始化工作完成后调用OSStart()启动多任务环境。

第一个运行的任务：

设定中断向量0x08指向函数OSTickISR();

将时钟节拍频率从18.20648改为200Hz。

在程序清单9.6给出了函数OSTickISR()的示意代码。和μC/OS-II中的其他中断服务程序一样，OSTickISR()首先在被中断任务堆栈中保存CPU寄存器的值，然后调用OSIntEnter()。

```
void OSTickISR (void)
{
    Save processor registers;          (1)
    OSIntNesting++;                  (2)
    OSTickDOSCtr++;                 (3)
    if (OSTICKENDOSCTR == 0) {        (4)
        Chain BIOS DOS interrupt on 'TRY RET' instruction;
    } else {

```

OSIntEnter(), 其作用是将记录中断嵌套层数的全局Enter(), 直接将OSIntNesting加1也是允许的。接下来[3]，每发生11次中断，OSTickDOSCtr减到0，则[4]，调用间隔大约是54.93ms。如果不调用DOS

时钟中断函数，则向中断优先级控制器(PIC)发送命令清除中断标志。如果调用了DOS中断，则此项操作可免，因为在DOS的中断程序中已经完成了。随后，OSTickISR()调用OSTimeTick(), 检查所有处于延时等待状态的任务，判断是否有延时结束就绪的任务[程序清单9.6(6)]。在OSTickISR()的最后调用OSIntExit(), 如果在中断中(或其他嵌套的中断)有更高优先级的任务就绪，并且当前中断为中断嵌套的最后一层。OSIntExit()将进行任务调度。注意如果进行了任务调度，OSIntExit()将不再返回调用者，而是用新任务的堆栈中的寄存器数值恢复CPU现场，然后用IRET实现任务切换。如果当前中断不是中断嵌套的最后一层，或中断中没有改变任务的就绪状态，OSIntExit()将返回调用者OSTickISR(), 最后OSTickISR()返回被中断的任务。

程序清单9.7给出了OSTickISR()的完整代码。

程序清单 9.6 OSTickISR()示意代码

```

        Send BOI command to PIC (Priority Interrupt Controller); (5)
    )
    OSTimeTick();                                (6)
    OSIntExit();                                 (7)
    Restore processor registers;                (8)
    Execute a return from interrupt instruction (IRET); (9)
}

```

程序清单 9.7 OSTickISR()

```

_OSTickISR PROC FAR
;
    PUSHA                                     ; 保存被中断任务的 CPU 环境
    PUSH ES
    PUSH DS
;
    MOV AX, SEG _OSTickDOSCtr      ; 载入 DS
    MOV DS, AX
;
    INC BYTE PTR _OSIntNesting     ; 提示 uC/OS-II 进入中断
;
    DEC BYTE PTR DS:_OSTickDOSCtr
    CMP BYTE PTR DS:_OSTickDOSCtr, 0
    JNE SHORT _OSTickISR1         ; 每 11 个时钟节拍(18.206 Hz)调用 DOS 时钟中断
;
    MOV BYTE PTR DS:_OSTickDOSCtr, 11
    INT 081H                         ; 调用 DOS 时钟中断处理过程
    JMP SHORT _OSTickISR2
;
_OSTickISR1:
    MOV AL, 20H                          ; 向中断优先级控制器发送命令, 清除标志位
    MOV DX, 20H
    OUT DX, AL
;
_OSTickISR2:
    CALL FAR PTR _OSTimeTick          ; 调用 OSTimeTick() 函数
;
    CALL FAR PTR _OSIntExit          ; 提示 uC/OS-II 退出中断

```

```

void OSTickISR (void)
{
    save processor registers;          (1)
    OSIntNesting++;                  (2)
    Chain into DOS by executing an 'INT 81H' instruction; (3)
    OSTimeTick();                   (4)
    OSIntExit();                     (5)
    Restore processor registers;     (6)
    Execute a return from interrupt instruction (IRET); (7)
}

```

保持18.20648 Hz)，OSTickISR()函数还可以简化。的示意代码。同样，函数开头要保存所有的CPU[程序清单9.8(2)]。接下来调用DOS的时钟中断[程序清单9.8(3)]。清除中断优先级控制器的操作了，因为DOS的时OSTimeTick()检查任务的延时是否结束[程序清单9.8(4)]，最后调用OSIntExit()[程序清单9.8(5)]。结束部分是恢复CPU寄存器的内容[程序清单9.8(6)]，执行IRET指令返回被中断的任务。如果采用8.2 Hz的OSTickISR()函数，系统初始化过程就不用调用PC_SetTickRate()，同时将文件OS_CFG.H中的常量OS_TICKS_PER_SEC由200改为18。

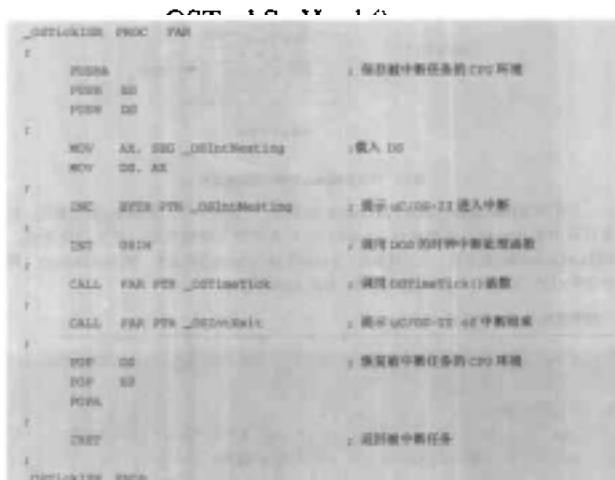
程序清单9.9给出了18.2 Hz OSTickISR()的完整代码。

程序清单 9.8 18.2Hz OSTickISR()伪码

9.5 OS_CPU_C.C

μ C/OS-II 的移植需要用户改写OS_CPU_C.C中的6个函数：

OSTaskStkInit()
OSTaskCreateHook()
OSTaskDelHook()



)函数，其他5个函数需要声明，但不一定有实际内
 S_CPU_C.C中没有给出代码。如果用户需要使用这
 5个函数，可以在头文件中将常量 OS_CPU_HOOKS_EN 设为1，设为0表示
 不使用这些函数。
)函数

9.5.1 OSTaskStkInit()

该函数由OSTaskCreate()或OSTaskCreateExt()调用，用来初始化任务的堆栈。初始状态

的堆栈模拟发生一次中断后的堆栈结构。图9.7说明了OSTaskStkInit()初始化后的堆栈内容。请注意，图中的堆栈结构不是调用OSTaskStkInit()的任务的，而是新创建任务的。

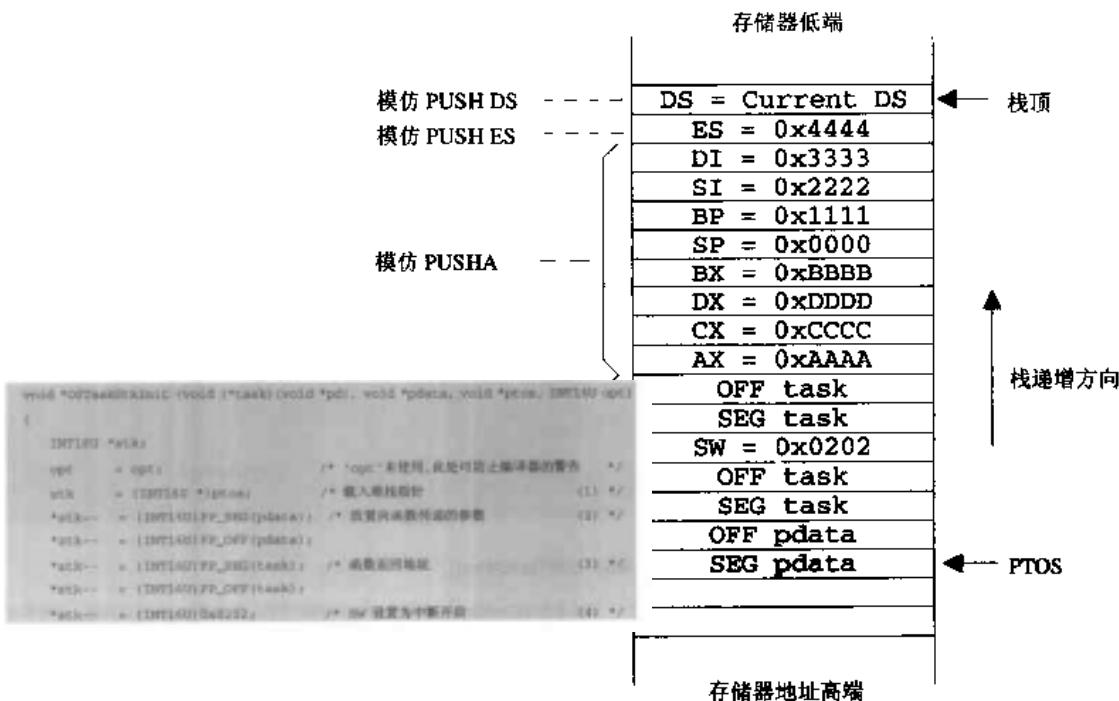


图9.7 传递参数pdata的堆栈初始化结构

当调用OSTaskCreate()或OSTaskCreateExt()创建一个新任务时，需要传递的参数是：任务代码的起使地址，参数指针（pdata），任务堆栈顶端的地址，任务的优先级。OSTaskCreateExt()还需要一些其他参数，但与OSTaskStkInit()没有关系。OSTaskStkInit()（程序清单9.10）只需要以上提到的3个参数（task, pdata, 和ptos）。

程序清单 9.10 OSTaskStkInit()



μC/OS-II 80x86 上的移植

```
*pdata = *(INT16*)DP_BEG->task; /* 指向堆栈底端指向任务代码的指针 */
*pdata = *(INT16)PP_DPF(task); /* AX = 0xAAAA */
*pdata = *(INT16)BnAAAA; /* DS = 0xCCCC */
*pdata = *(INT16)BnCCCC; /* DX = 0xCCCC */
*pdata = *(INT16)Bn0000; /* BX = 0x0000 */
*pdata = *(INT16)Bn0000; /* SP = 0x0000 */
*pdata = *(INT16)Bn1111; /* BP = 0x1111 */
*pdata = *(INT16)Bn2222; /* SI = 0x2222 */
*pdata = *(INT16)Bn3333; /* DI = 0x3333 */
*pdata = *(INT16)Bn4444; /* DS 寄存器 */
*pdata = _DS; /* DS = 0x0000 的 DS 寄存器 */
�DATA (000000, *1850)
```

由于80x86 堆栈是16位宽的（以字为单位）[程序清单9.10(1)]，OSTaskStkInit()将创立一个指向以字为单位的内存区域的指针。同时要求堆栈指针指向空堆栈的顶端。

笔者使用的Borland C/C++编译器配置为用堆栈而不是寄存器来传送参数pdata，此时参数pdata的段地址和偏移量都将被保存在堆栈中[程序清单9.10(2)]。

堆栈中紧接着是任务函数的起始地址[程序清单9.10(3)]，理论上，此处应该为任务的返回地址，但在μC/OS-II中，任务函数必须为无限循环结构，不能有返回点。

返回地址下面是状态字(SW)[程序清单9.10(4)]，设置状态字也是为了模拟中断发生后的堆栈结构。堆栈中的SW初始化为0x0202，这将使任务启动后允许中断发生；如果设为0x0002，则任务启动后将禁止中断。需要注意的是，如果选择任务启动后允许中断发生，则所有的任务运行期间中断都允许；同样，如果选择任务启动后禁止中断，则所有的任务都禁止中断发生，而不能有所选择。

如果确实需要突破上述限制，可以通过参数pdata向任务传递希望实现的中断状态。如果某个任务选择启动后禁止中断，那么其他的任务在运行的时候需要重新开启中断。同时还要修改OSTaskIdle()和OSTaskStat()函数，在运行时开启中断。如果以上任何一个环节出现问题，系统就会崩溃。所以笔者还是推荐用户设置SW为0x0202，在任务启动时开启中断。

堆栈中还要留出各个寄存器的空间，注意寄存器在堆栈中的位置要和运行指令PUSHA、PUSH ES，和PUSH DS和压入堆栈的次序相同。上述指令在每次进入中断服务程序时都会被调用[程序清单9.10(5)]。AX,BX,CX,DX,SP,BP,SI,和DI的次序是和指令PUSHA的压栈次序相同的。如果使用没有PUSHA指令的8086处理器，就要使用多个PUSH指令压入上述寄存器，且顺序要与PUSHA相同。在程序清单9.10中每个寄存器被初始化为不同的值，这是为

了调试方便。Borland编译器支持伪寄存器变量操作，可以用`_DS`关键字取得CPU DS寄存器的值，程序清单9.10(6)处用`_DS`直接把DS寄存器拷贝到堆栈中。

堆栈初始化工作结束后，`OSTaskStkInit()`返回新的堆栈栈顶指针，`OSTaskCreate()`或`OSTaskCreateExt()`将指针保存在任务的`OS_TCB`中。

9.5.2 OTaskCreateHook()

`OS_CPU_C.C`中未定义，此函数为用户定义。

9.5.3 OTaskDelHook()

`OS_CPU_C.C`中未定义，此函数为用户定义。

9.5.4 OTaskSwHook()

`OS_CPU_C.C`中未定义，此函数为用户定义。其用法请参考例3。

9.5.5 OTaskStatHook()

`OS_CPU_C.C`中未定义，此函数为用户定义。其用法请参考例3。

9.5.6 OSTimeTickHook()

`OS_CPU_C.C`中未定义，此函数为用户定义。

9.6 内存占用

表9.1列出了指定初始化常量的情况下，μC/OS-II占用内存的情况，包括数据和程序代码。如果μC/OS-II用于嵌入式系统，则数据指RAM的占用，程序代码指ROM的占用。内存占用的说明清单随磁盘一起提供给用户，在安装μC/OS-II后，查看\SOFTWARE\μCOS-II\Ix836L\DOC\目录下的ROM-RAM.XLS文件。该文件为Microsoft Excel文件，需要用Office 97或更高版本的Excel打开。

表9.1中所列出的内存占用大小都近似为25字节的倍数。笔者所用的Borland C/C++ V3.1设定为编译产生运行速度最快的目标代码，所以表中所列的数字并不是绝对的，但可以给读者一个总的概念。例如，如果不使用消息队列机制，在编译前将`OS_Q_EN`设为0，则编译后的目标代码长度6875字节，可减小大约1475字节。

此外，空闲任务（idle）和统计任务（statistics）的堆栈都设为1024字节（1KB）。根据你自己的要求可以增减。μC/OS-II的数据结构最少需要35字节的RAM。

表 9.1

μC/OS-II 内存占用 (80186)

配置参数	值	代码 (字节)	数据 (字节)
OS_MAX_EVENTS	10		164
OS_MAX_MEM_PART	5		104
OS_MAX_QS	5		124
OS_MAX_TASKS	63		2925
OS_LOWEST_PRIO	63		264
OS_TASK_IDLE_STK_SIZE	512		1024
OS_TASK_STAT_EN	1	325	10
OS_TASK_STAT_STK_SIZE	512		1024
OS_CPU_HOOKS_EN	1		0
OS_MBOX_EN	1	600	(参看 OS_MAX_EVENTS)
OS_MEM_EN	1	725	(参看OS_MAX_MEM_PART)
OS_Q_EN	1	1475	(参看OS_MAX_QS)
OS_SEM_EN	1	475	(参看OS_MAX_EVENTS)
OS_TASK_CHANGE_PRIO_EN	1	450	0
OS_TASK_CREATE_EN	1	225	1
OS_TASK_CREATE_EXT_EN	1	300	0
OS_TASK_DEL_EN	1	550	0
OS_TASK_SUSPEND_EN	1	525	0
μC/OS-II 内核		2700	35
应用程序堆栈	0		0
应用程序的RAM	0		0
总计		8350	5675

表9.2说明了如何裁减μC/OS-II，应用在更小规模的系统上。这样的小系统有16个任务，并且不采用如下功能：

● 邮箱功能 (OS_MBOX_EN 设为 0)

为 0)

_CHANGE_PRIO_EN 设为 0)

create() (OS_TASK_CREATE_EN 设为 0)

及为 0)

SPEND_EN 设为 0)

μC/OS-II 配置

配置参数	值	代码 (字节)	数据 (字节)
OS_MAX_EVENTS	32		164
OS_MAX_MEM_PART	5		0
OS_MAX_QB	3		124
OS_MAX_TASKS	32		792
OS_LOWEST_PRIO	63		164
OS_TASK_IDLE_STK_SIZE	512		1024
OS_TASK_STAT_EN	1	325	10
OS_TASK_STAT_STK_SIZE	512		1024
OS_CPU_HOOKS_EN	1		0
OS_MBOX_EN	0	0	(参考OS_MAX_EVENTS)
OS_MEM_EN	0	0	(参考OS_MAX_MEM_PART)
OS_Q_EN	1	3479	(参考OS_MAX_QB)
OS_SEM_EN	1	475	(参考OS_MAX_EVENTS)
OS_TASK_CHANGE_PRIO_EN	0	0	0
OS_TASK_CREATE_EN	0	0	1
OS_TASK_CREATE_EXT_EN	1	300	0
OS_TASK_DEL_EN	0	0	0
OS_TASK_SUSPEND_EN	0	0	0
μCOS-II 内核		2700	35
应用程序堆栈	0		0
应用程的RAM	0		0
总计		5275	3438



采取上述措施后，程序代码空间可以减小3Kb，数据空间可以减小2,200字节。由于只有16个任务运行，节省了大量用于任务控制块OS_TCB的空间。在80x86的大模式编译条件下，每一个OS_TCB将占用45字节的RAM。

函数	关闭中断时间			最小运行时间			最大运行时间		
	I	C	μs	I	C	μs	I	C	μs
线程管理									
OSCreate()									
OSCreate()	—	—	—	—	—	—	—	—	—
OSScheduler()	4	34	1.8	7	87	2.8	7	87	2.8
OSSchedulerUnlock()	51	587	17.2	13	130	3.9	73	782	23.7
OSReset()	8	9	0.8	35	278	8.4	35	278	8.4
OSThreadExit()	—	—	—	—	—	—	—	—	—
OSYield()	8	9	0.9	2	19	0.6	2	19	0.6
中断管理									
OSIntEnter()	4	42	1.3	4	42	1.3	4	42	1.3
OSIntExit()	58	558	16.9	27	207	6.3	31	534	27.4
OSTimeGet()	38	310	9.4	948	38803	327.4	2364	30620	624.8
内存管理									
OSMemoryAllocate()	15	364	4.9	13	297	7.8	13	297	7.8
OSMemoryCreate()	15	148	4.5	135	939	28.5	115	939	28.5
OSMemoryRead()	68	567	17.2	28	317	9.6	164	1912	57.9
OSMemoryWrite()	84	347	22.6	24	305	9.2	152	1484	45.0
OSMemoryQuery()	120	998	29.9	128	1257	38.1	128	1257	38.1



函数在80186处理器上的运行时间。统计的方法是条汇编指令所需的时钟周期，根据处理器的时钟频率，函数包含有多少条指令，C栏为函数运行需要多少时钟周期，μs栏为函数运行所需的时间。表中有3类时间，分别是在函数中关闭中断、运行时间和在函数中打开中断。如果你不使用80186处理器，表中的数据就没有意义了，但可用来比较不同函数运行时间的相对大小。

MHz 80186 上的执行时间

续表

函数	关闭中断时间			最小运行时间			最大运行时间		
	I	C	μs	I	C	μs	I	C	μs
OSMemPut()	23	282	8.5	12	161	4.9	29	321	9.7
OSMemQuery()	40	400	12.1	45	450	13.6	45	450	13.6
消息队列									
OSQAccept()	34	387	11.7	25	269	8.2	44	479	14.5
OSQCreate()	14	150	4.5	154	1291	39.1	154	1291	39.1
OSQFlush()	18	202	6.1	25	253	7.7	25	253	7.7
OSQPend()	64	620	18.8	45	495	15.0	186	1938	58.7
OSQPost()	98	873	26.5	51	547	16.6	155	1493	45.2
OSQPostFront()	87	788	23.9	44	412	12.5	153	1483	44.9
OSQuery()	128	1100	33.3	137	1171	35.5	137	1171	35.5
信号量管理									
OSSemAccept()	10	113	3.4	16	161	4.9	16	161	4.9
OSSemCreate()	14	140	4.2	98	768	23.3	98	768	23.3
OSSemPend()	58	567	17.2	17	184	5.6	164	1690	51.2
OSSemPost()	87	776	23.5	18	198	6.0	151	1469	44.5
OSSemQuery()	110	882	26.7	116	931	28.2	116	931	28.2
任务管理									
OSTaskChangePrio()	63	567	17.2	178	981	29.7	166	1532	46.4
OSTaskCreate()	57	567	17.2	217	2388	72.4	266	2939	89.1
OSTaskCreateExt()	57	567	17.2	235	2606	79.0	284	3157	95.7
OSTaskDel()	62	620	18.8	116	1206	36.5	165	1757	53.2
OSTaskDelReq()	23	199	6.0	39	330	10.0	39	330	10.0
OSTaskQuery()	84	1025	31.1	95	1122	34.0	95	1122	34.0
OSTaskResume()	27	242	7.3	48	430	13.0	97	981	29.7
OSTaskStkChk()	31	316	9.6	62	599	18.2	62	599	18.2
OSTaskSuspend()	37	352	10.7	63	579	17.5	112	1130	34.2
时钟管理									
OSTimeDly()	57	567	17.2	81	844	25.6	85	871	26.4
OSTimeDlyHMSM()	57	567	17.2	216	2184	66.2	220	2211	67.0
OSTimeDlyResume()	57	567	17.2	23	181	5.5	98	989	30.0
OSTimeGet()	7	57	1.7	14	117	3.5	14	117	3.5
OSTimeSet()	7	61	1.8	11	99	3.0	11	99	3.0
OSTimeTick()	30	310	9.4	900	10257	310.8	1908	19707	597.2

函数	关闭中断时间			最小运行时间			最大运行时间		
	I	C	μs	I	C	μs	I	C	μs
用户定义函数									
OSSchedCreateHook()	0	0	0.0	4	38	1.2	4	38	3.2
OSSchedDeleteHook()	0	0	0.0	4	38	1.2	4	38	1.2
OSSchedStartHook()	0	0	0.0	1	38	0.5	1	38	0.5
OSSchedStopHook()	0	0	0.0	1	38	0.5	1	38	0.5
OSTimeTickHook()	0	0	0.0	1	38	0.5	1	38	0.5

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

以上各表中的时间数据都是假设函数成功运行，正常返回；同时假定处理器工作在最大总线速度。平均来说，80186处理器的每条指令需要10个时钟周期。

对于80186处理器，μC/OS-II中的函数最大的关闭中断时间是33.3μs，约1100个时钟周期。

N/A是指该函数的运行时间长短并不重要，例如一些只执行一次的初始化函数。

如果你用的是x86系列的其他CPU，你可以根据表中每个函数的运行时钟周期项估计当前CPU的执行时间。例如，如果用80486，且知80486的指令平均用2个时钟周期；或者知道80486总线频率为66MHz（比80186的33MHz快2倍），都可以估计出函数在80486上的执行时间。

下面我们将讨论每个函数的关闭中断时间，最大、最小运行时间是如何计算的，以及这样计算的先决条件。

OSSchedUnlock()

最小运行时间是当变量OSLockNesting减为0，且系统中没有更高优先级的任务就绪，OSSchedUnlock()正常结束返回调用者。

最大运行时间是也是当变量OSLockNesting减为0，但有更高优先级的任务就绪，函数中需要进行任务切换。

OSIntExit()

最小运行时间是当变量OSLockNesting减为0，且系统中没有更高优先级的任务就绪，OSIntExit()正常结束返回被中断任务。

最大运行时间是也是当变量OSLockNesting减为0，但有更高优先级的任务就绪，OSIntExit()将不返回调用者，经过任务切换操作后，将直接返回就绪的任务。

OSTickISR()

此函数假定在当前μC/OS-II中运行有最大数目的任务（64个）。

最小运行时间是当64个任务都不在等待延时状态。也就是说，所有的任务都不需要OSTickISR()处理。

最大运行时间是当63个任务(空闲进程不会延时等待)都处于延时状态,此时OSTickISR()需要逐个检查等待中的任务,将计数器减1,并判断是否延时结束。这种情况对于系统是一个很重的负荷。例如在最坏的情况下,设时钟节拍间隔10ms, OSTickISR()需要625μs,占了约6%的CPU利用率。但请注意,此时所有的任务都没有执行,只是内核的开销。

OSMboxPend()

最小运行时间是当邮箱中有消息需要处理的时候。

最大运行时间是当邮箱中没有消息,任务需要等待的时候。此时调用OSMboxPend()的任务将被挂起,进行任务切换。最大运行时间是同一任务执行OSMboxPend()的累计时间,这个过程包括OSMboxPend()查看邮箱,发现没有消息,再调用任务切换函数OSSched(),切换到新任务。当由于某种原因调用OSMboxPend()的任务又被唤醒执行,从OSSched()中返回,发现返回的原因是由于延时结束(译注1),最后返回调用任务。OSMboxPend()的最大运行时间是上述时间的总和。

OSMboxPost()

最小运行时间是当邮箱是空的,没有任务等待消息的时候。

最大运行时间是当消息邮箱中有一个或多个任务在等待消息。此时,消息将发往等待队列中优先级最高的任务,将此任务唤醒执行。最大运行时间是同一任务执行OSMboxPost()的累计时间,这个过程包括任务唤醒等待任务,发送消息,调用任务切换函数OSSched(),切换到新任务。当由于某种原因调用OSMboxPost()的任务又被唤醒执行,从OSSched()中返回,发现返回的原因是由于延时结束(译注2),最后返回调用任务。OSMboxPost()的最大运行时间是上述时间的总和。

OSMemGet()

最小运行时间是当系统中已经没有内存块,OSMemGet()返回错误码。

最大运行时间是OSMemGet()获得了内存块,返回调用者。

OSMemPut()

最小运行时间是当向一个已经排满的内存分区中返回内存块。

最大运行时间是当向一个未排满的内存分区中返回内存块。

OSQPend()

最小运行时间是当消息队列中有消息需要处理的时候。

译注1: 处理延时结束情况下的代码最长。

译注2: 处理延时结束情况下的代码最长。

最大运行时间是当消息队列中没有消息，任务需要等待的时候。此时调用OSQPend()的任务将被挂起，进行任务切换。最大运行时间是同一任务执行OSQPend()的累计时间，这个过程包括OSQPend()查看消息队列，发现没有消息，再调用任务切换函数OSSched()，切换到新任务。当由于某种原因调用OSQPend()的任务又被唤醒执行，从OSSched()中返回，发现返回的原因是由于延时结束（译注3），最后返回调用任务。OSQPend()的最大运行时间是上述时间的总和。

OSQPost()

最小运行时间是当消息队列是空的，没有任务等待消息的时候。

最大运行时间是当消息队列中有一个或多个任务在等待消息。此时，消息将发往等待队列中优先级最高的任务，将此任务唤醒执行。最大运行时间是同一任务执行OSQPost()的累计时间，这个过程包括任务唤醒等待任务，发送消息，调用任务切换函数OSSched()，切换到新任务。当由于某种原因调用OSQPost()的任务又被唤醒执行，从OSSched()中返回，发现返回的原因是由于延时结束（译注4），最后返回调用任务。OSQPost()的最大运行时间是上述时间的总和。

OSQPostFront()

此函数与OSQPost()的过程相同。

OSSemPend()

最小运行时间是当信号量可获取的时候（信号量计数器大于0）。

最大运行时间是当信号量不可得，任务需要等待的时候。此时调用OSSemPend()的任务将被挂起，进行任务切换。最大运行时间是同一任务执行OSSemPend()的累计时间，这个过程包括OSSemPend()查看信号量计数器，发现是0，再调用任务切换函数OSSched()，切换到新任务。当由于某种原因调用OSSemPend()的任务又被唤醒执行，从OSSched()中返回，发现返回的原因是由于延时结束（译注5），最后返回调用任务。OSSemPend()的最大运行时间是上述时间的总和。

OSSemPost()

最小运行时间是当没有任务在等待信号量的时候。

最大运行时间是当有一个或多个任务在等待信号量。此时，等待队列中优先级最高的任务将被唤醒执行。最大运行时间是同一任务执行OSSemPost()的累计时间，这个过程包括任务唤醒等待任务，调用任务切换函数OSSched()，切换到新任务。当由于某种原因调用

译注 3：处理延时结束情况下的代码最长。

译注 4：处理延时结束情况下的代码最长。

译注 5：处理延时结束情况下的代码最长。

超星浏览器提醒您：
使用本资源制品
请尊重相关知识产权！

OSSemPost()的任务又被唤醒执行，从OSSched()中返回，发现返回的原因是由于延时结束（译者6），最后返回调用任务。OSSemPost()的最大运行时间是上述时间的总和。

OSTaskChangePrio()

最小运行时间是当任务被改变的优先级比当前运行任务的低，此时不进行任务切换，直接返回调用任务。

最大运行时间是当任务被改变的优先级比当前运行任务的高，此时将进行任务切换。

OSTaskCreate()

最小运行时间是当调用OSTaskCreate()的任务创建了一个比自己优先级低的任务，此时不进行任务切换。

最大运行时间是当调用OSTaskCreate()的任务创建了一个比自己优先级高的任务，此时将进行任务切换。

上述两种情况都是假定OSTaskCreateHook()不进行任何操作。

OSTaskCreateExt()

最小运行时间是当OSTaskCreateExt()不对堆栈进行清零操作（此项操作是为堆栈检查函数做准备的）。

最大运行时间是当OSTaskCreateExt()需要进行堆栈清零操作。但此项操作的时间取决于堆栈的大小。如果设清除每个堆栈单元（译注7）需要100个时钟周期（3μs），1000字节的堆栈将需要1500μs（1000字节除以2再乘以3μs/每字）。在清除堆栈过程中中断是打开的，可以响应中断请求。

上述两种情况都是假定OSTaskCreateHook()不进行任何操作。

OSTaskDel()

最小运行时间是当被删除的任务不是当前任务，此时不进行任务切换。

最大运行时间是当被删除的任务是当前任务，此时将进行任务切换。

OSTaskDelReq()

该函数很短，几乎没有最小和最大运行时间之分。

OSTaskResume()

最小运行时间是当OSTaskResume()唤醒了一个任务，但该任务的优先级比当前任务低，此时不进行任务切换。

译注 6：处理延时结束情况下的代码最长。

译注 7：堆栈操作以字为单位。

最大运行时间是OSTaskResume()唤醒了一个优先级更高的任务，此时将进行任务切换。

OSTaskStkChk()

OSTaskStkChk()的执行过程是从堆栈的底端开始检查0的个数，估计堆栈所剩的空间。所以最小运行时间是当OSTaskStkChk()检查一个全部占满的堆栈。但实际上这种情况是不允许发生的，这将使系统崩溃。

最大运行时间是当OSTaskStkChk()检查一个全空堆栈，执行时间取决于堆栈的大小。例如检查每个堆栈单元需要80钟周期（ $2.4\mu s$ ），1000字节的堆栈将需要 $1200\mu s$ （1000字节除以2再乘以 $2.4\mu s$ /每字）。再加上其他的一些操作，总共需要大约 $1218\mu s$ 。在检查堆栈过程中中断是打开的，可以响中断请求。

OSTaskSuspend()

最小运行时间是当被挂起的任务不是当前任务，此时不进行任务切换。

最大运行时间是当前任务挂起自己，此时将进行任务切换。

OSTaskQuery()

该函数的运行时间总是一样的。OSTaskQuery()执行的操作是获取任务的任务控制块OS_TCB。如果OS_TCB中包含所有的操作项，需要占用45字节（大模式编译）。

OSTimeDly()

如果延时时间不为0，则OSTimeDly()运行时间总是相同的。此时将进行任务切换。

如果延时时间为0，OSTimeDly()不清除OSRdyGrp中的任务就绪位，不进行延时操作，直接返回。

OSTimeDlyHMSM()

如果延时时间不为0，则OSTimeDlyHMSM()运行时间总是相同的。此时将进行任务切换。此外，OSTimeDlyHMSM()延时时间最好不要超过65536个时钟节拍。也就是说，如果时钟节拍发生的间隔为10ms（频率100Hz），延时时间应该限定在10分55秒350毫秒内。如果超过了上述数值，该任务就不能用OSTimeDlyResume()函数唤醒。

OSTimeDlyResume()

最小运行时间是当被唤醒的任务优先级低于当前任务，此时不进行任务切换。

最大运行时间是当唤醒了一个优先级更高的任务，此时将进行任务切换。

OSTimeTick()

前面我们讨论的OSTickISR()函数其实就是OSTimeTick()与OSIntEnter()、OSIntExit()的

组合。OSTickISR()的时间占用情况就是OSTimeTick()的占用情况。以下讨论假定系统中有μC/OS-II允许的最大数量的任务（64个）。

等待延时状态。也就是说，所有的任务都不需要

闲进程不会延时等待）都处于延时状态，此时将计数器减1，并判断是否延时结束。例如在最

OSTick()需要约600μs，占了6%的CPU利用率。

（按关闭中断时间排序）

函数	关闭中断时间			最小运行时间			最大运行时间		
	T	C	μs	T	C	μs	T	C	μs
OSVersion()	0	0	0.0	2	19	0.8	2	19	0.8
OSStart()	0	0	0.0	35	276	8.4	35	276	8.4
OSSemaphore()	4	34	1.0	2	87	2.6	2	87	2.6
OSMailUser()	4	42	1.3	4	42	1.3	4	42	1.3
OSTimeGet()	2	37	1.7	14	137	3.5	14	137	3.5
OSTimeSet()	7	61	1.8	11	99	3.0	11	99	3.0
OSStatAccept()	10	123	3.4	16	181	4.9	16	181	4.9
OSSendCreate()	14	140	4.2	98	768	23.3	98	768	23.3
OSMutexCreate()	15	148	4.5	135	939	28.5	135	939	28.5
OSQCreate()	16	150	4.5	158	1281	39.1	158	1281	39.1
OSMutexAccept()	15	161	4.9	13	257	7.8	13	257	7.8
OSMemCreate()	21	181	5.5	72	766	23.2	72	766	23.2
OSTaskDelBeg()	23	199	6.0	39	330	10.0	39	330	10.0
OSQFlush()	18	202	6.1	25	253	7.7	25	253	7.7
OSTaskRemove()	27	242	7.3	48	480	13.0	97	981	29.7
OSMemGet()	19	247	7.5	18	172	9.2	33	380	10.6
OSMemPut()	23	282	8.3	12	141	4.9	29	321	8.3
OSTimeTick()	30	303	9.4	900	10257	310.8	1908	19307	397.2
OSTickIRQ()	30	319	9.4	946	10809	327.4	2304	20629	634.8
OSTickSdChk()	31	336	9.8	62	589	18.2	62	589	18.2
OSTaskSuspend()	37	362	10.7	63	579	17.5	132	1130	34.2
OSQAccept()	34	387	11.7	23	269	8.2	44	479	14.5
OSMemQuery()	40	400	12.1	43	490	13.6	45	490	13.6
OSExit()	56	558	16.9	27	207	6.3	57	516	17.4

使用本资料请尊重相关知识产权

续表

函数	关闭中断时间			最小运行时间			最大运行时间		
	I	C	μs	I	C	μs	I	C	μs
OSSchedUnlock()	57	567	17.2	13	130	3.9	73	782	23.7
OSTimeDly()	57	567	17.2	81	844	25.6	85	871	26.4
OSTimeDlyResume()	57	567	17.2	23	181	5.5	98	989	30.0
OSTaskChangePrio()	63	567	17.2	178	981	29.7	166	1532	46.4
OSSemPend()	58	567	17.2	17	184	5.6	164	1690	51.2
OSMboxPend()	68	567	17.2	28	317	9.6	184	1912	57.9
OSTimeDlyHMSM()	57	567	17.2	216	2184	66.2	220	2211	67.0
OSTaskCreate()	57	567	17.2	217	2388	72.4	266	2939	89.1
OSTaskCreateExt()	57	567	17.2	235	2606	79.0	284	3157	95.7
OSTaskDel()	62	620	18.8	116	1206	36.5	165	1757	53.2
OSQPend()	64	620	18.8	45	495	15.0	186	1938	58.7
OSMboxPost()	84	747	22.6	24	305	9.2	152	1484	45.0
OSSemPost()	87	776	23.5	18	198	6.0	151	1469	44.5
OSQPostFront()	87	788	23.9	44	412	12.5	153	1483	44.9
OSQPost()	98	873	26.5	51	547	16.6	155	1493	45.2
OSSemQuery()	110	882	26.7	116	931	28.2	116	931	28.2
OSMboxQuery()	120	988	29.9	128	1257	38.1	128	1257	38.1
OSTaskQuery()	84	1025	31.1	95	1122	34.0	95	1122	34.0
OSQQuery()	128	1100	33.3	137	1171	35.5	137	1171	35.5
OSStatInit()	—	—	—	—	—	—	—	—	—
OSInit()	—	—	—	—	—	—	—	—	—

表 9.5 各函数的执行时间(按最大运行时间排序)

Service	关闭中断时间			最大运行时间			最小运行时间		
	I	C	μs	I	C	μs	I	C	μs
OSVersion()	0	0	0.0	2	19	0.6	2	19	0.6
OSIntEnter()	4	42	1.3	4	42	1.3	4	42	1.3
OSSchedLock()	4	34	1.0	7	87	2.6	7	87	2.6
OSTimeSet()	7	61	1.8	11	99	3.0	11	99	3.0
OSTimeGet()	7	57	1.7	14	117	3.5	14	117	3.5
OSSemAccept()	10	113	3.4	16	161	4.9	16	161	4.9

续表

Service	关闭中断时间			最大运行时间			最小运行时间		
	I	C	μs	I	C	μs	I	C	μs
OSQFlush()	18	202	6.1	25	253	7.7	25	253	7.7
OSMboxAccept()	15	161	4.9	13	257	7.8	13	257	7.8
OSStart()	0	0	0.0	35	278	8.4	35	278	8.4
OSMemPut()	23	282	8.5	12	161	4.9	29	321	9.7
OSTaskDelReq()	23	199	6.0	39	330	10.0	39	330	10.0
OSMemGet()	19	247	7.5	18	172	5.2	33	350	10.6
OSMemQuery()	40	400	12.1	45	450	13.6	45	450	13.6
OSQAccept()	34	387	11.7	25	269	8.2	44	479	14.5
OSIntExit()	56	558	16.9	27	207	6.3	57	574	17.4
OSTaskStkChk()	31	316	9.6	62	599	18.2	62	599	18.2
OSMemCreate()	21	181	5.5	72	766	23.2	72	766	23.2
OSSemCreate()	14	140	4.2	98	768	23.3	98	768	23.3
OSSchedUnlock()	57	567	17.2	13	130	3.9	73	782	23.7
OSTimeDly()	57	567	17.2	81	844	25.6	85	871	26.4
OSSemQuery()	110	882	26.7	116	931	28.2	116	931	28.2
OSMboxCreate()	15	148	4.5	115	939	28.5	115	939	28.5
OSTaskResume()	27	242	7.3	48	430	13.0	97	981	29.7
OSTimeDlyResume()	57	567	17.2	23	181	5.5	98	989	30.0
OSTaskQuery()	84	1025	31.1	95	1122	34.0	95	1122	34.0
OSTaskSuspend()	37	352	10.7	63	579	17.5	112	1130	34.2
OSQuery()	128	1100	33.3	137	1171	35.5	137	1171	35.5
OSMboxQuery()	120	988	29.9	128	1257	38.1	128	1257	38.1
OSQCreate()	14	150	4.5	154	1291	39.1	154	1291	39.1
OSSemPost()	87	776	23.5	18	198	6.0	151	1469	44.5
OSQPostFront()	87	788	23.9	44	412	12.5	153	1483	44.9
OSMboxPost()	84	747	22.6	24	305	9.2	152	1484	45.0
OSQPost()	98	873	26.5	51	547	16.6	155	1493	45.2
OSTaskChangePrio()	63	567	17.2	178	981	29.7	166	1532	46.4
OSSemPend()	58	567	17.2	17	184	5.6	164	1690	51.2
OSTaskDel()	62	620	18.8	116	1206	36.5	165	1757	53.2
OSMboxPend()	68	567	17.2	28	317	9.6	184	1912	57.9
OSQPend()	64	620	18.8	45	495	15.0	186	1938	58.7
OSTimeDlyHMSM()	57	567	17.2	216	2184	66.2	220	2211	67.0

续表

Service	关闭中断时间			最大运行时间			最小运行时间		
	I	C	μs	I	C	μs	I	C	μs
OSTaskCreate()	57	567	17.2	217	2388	72.4	266	2939	89.1
OSTaskCreateExt()	57	567	17.2	235	2606	79.0	284	3157	95.7
OSTimeTick()	30	310	9.4	900	10257	310.8	1908	19707	597.2
OSTickISR()	30	310	9.4	948	10803	327.4	2304	20620	624.8
OSInit()	—	—	—	—	—	—	—	—	—
OSStdInit()	—	—	—	—	—	—	—	—	—

从表中可以看出，关闭中断时间越长，最大运行时间也越长。这说明在移植时，如果希望系统响应时间快，那么必须尽量减少中断的关闭时间。

附录B 参考资料

本书的许多内容取自于“μC/OS-II”、“μC/OS-III”、“μC/OS-IV”等移植手册，以及“μC/OS-II移植指南”、“μC/OS-III移植指南”、“μC/OS-IV移植指南”等移植手册。这些移植指南由瑞典的O.S. Software公司编写，是移植μC/OS系列嵌入式实时操作系统到各种不同处理器平台的权威指南。在编写过程中，我们参考了移植指南中的许多内容，同时对移植指南的内容进行了大量的修改和补充，使其更符合我国的国情，更便于国内读者阅读和使用。希望本书能为我国嵌入式系统的开发提供一些帮助。

移植指南	处理器	移植日期	作者
μC/OS-II移植指南	Intel 8051	1996年1月	赵伟、王海峰
μC/OS-II移植指南	Intel 8086/8088	1996年1月	赵伟、王海峰
μC/OS-II移植指南	Motorola 68000	1996年1月	赵伟、王海峰
μC/OS-II移植指南	TI TMS320C30	1996年1月	赵伟、王海峰

本书的许多内容取自于“μC/OS-II”、“μC/OS-III”、“μC/OS-IV”等移植手册，以及“μC/OS-II移植指南”、“μC/OS-III移植指南”、“μC/OS-IV移植指南”等移植手册。这些移植指南由瑞典的O.S. Software公司编写，是移植μC/OS系列嵌入式实时操作系统到各种不同处理器平台的权威指南。在编写过程中，我们参考了移植指南中的许多内容，同时对移植指南的内容进行了大量的修改和补充，使其更符合我国的国情，更便于国内读者阅读和使用。希望本书能为我国嵌入式系统的开发提供一些帮助。



第 10 章

从μC/OS 升级到μC/OS-II

本章描述如何从μC/OS 升级到μC/OS-II。如果已经将μC/OS 移植到了某类微处理器

;有限。在多数情况下，用户能够在 1 个小时之移植，可隔过本章前一部分直接参阅 10.5 节。

SOFTWARE\uCOS	SOFTWARE\uCOS-II
intel.H	OS_CPU.H
lx86L_A.ASM	OS_CPU_A.ASM
lx86L_C.C	OS_CPU_C.C

10.0 目录和文件

用户首先会注意到的是目录的结构，主目录不再叫 \SOFTWARE\uCOS。而是叫 \SOFTWARE\uCOS-II。所有的μC/OS-II 文件都应放在用户硬盘的\SOFTWARE\uCOS-II 目录下。面向不同的微处理器或微处理器的源代码一定是在以下两个或三个文件中：OS_CPU.H, OS_CPU_C.C, 或许还有 OS_CPU_A.ASM.。汇编语言文件是可有可无的，因为有些 C 编译程序允许使用在线汇编代码，用户可以将这些汇编代码直接写在 OS_CPU_C.C. 中。

与微处理器有关的特殊代码，即与移植有关的代码，在 μC/OS 中是放在用微处理器名字命名的文件中的，例如，Intel 80x86 的实模式（Real Mode），在大模式下编译（Large Model）时，文件名为 Ix86L.H, Ix86L_C.C, 和 Ix86L_A.ASM.

表 10.0

在μC/OS-II 中重新命名的文件

升级可以从这里开始：首先将μC/OS 目录下的旧文件复制到μC/OS-II 的相应目录下，并改用新的文件名，这比重新建立一些新文件要容易许多。表 10.2 给出来几个与移植有关的新旧文件名命名法的例子。

SOFTWARE\μCOS-II\0251.H	SOFTWARE\μCOS-II\0251.H
00251.H	OS_CPU.H
00251.C	OS_CPU.C
SOFTWARE\μCOS-MN86.D	SOFTWARE\μCOS-II\0251.D
MN86.H	OS_CPU.H
MN86.C	OS_CPU.C
SOFTWARE\μCOS-MN86.C	SOFTWARE\μCOS-II\0251.C
MN86.C1.H	OS_CPU.H
MN86.C1.C	OS_CPU.C
SOFTWARE\μCOS-286.D	SOFTWARE\μCOS-II\286.D
Z80.D	OS_CPU.H
Z80_A.ASM	OS_CPU_A.ASM
Z80.C	OS_CPU.C

μC/OS 升级到μC/OS-II

到μC/OS-II，要重新命名的文件



10.1 INCLUDES.H

用户应用程序中的INCLUDES.H 文件要修改。以80x86 实模式，在大模式下编译为例，用户要做如下修改：

- 变目录名μC/OS 为μC/OS-II；
- 变文件名 IX86LH 为 OS_CPU.H；
- 变文件名 UCOSH 为 uCOS_II.H。

新旧文件如程序清单10.1和10.2所示。

10.2 OS_CPU.H

OS_CPU.H 文件中有与微处理器类型及相应硬件有关的常数定义、宏定义和类型定义。

10.2.1 与编译有关的数据类型 s

为了实现 μC/OS-II，用户应定义6个新的数据类型：INT8U、INT8S、INT16U、NT16S、INT32U、和INT32S。这些数据类型有分别表示有符号和无符号8位、16位、32位整数。在μC/OS 中相应的数据类型分别定义为：UBYTE、BYTE、UWORD、WORD、ULONG和LONG。用户所要做的仅仅是复制μC/OS 中数类型并修改原来的UBYTE为INT8U，将BYTE为INT8S，将UWORD修改为INT16U等等，如程序清单10.3所示。

程序清单 10.1 μC/OS 中的 INCLUDES.H

```
/*
*****
*          INCLUDES.H
*****
*/
#include <STDIO.H>
#include <STRING.H>
#include <CTYPE.H>
#include <STDLIB.H>
#include <CONIO.H>
#include <DOS.H>

#include "\SOFTWARE\UCOS\IX86L\IX86L.H"
#include "OS_CFG.H"
#include "\SOFTWARE\UCOS\SOURCE\UCOS.H"
```

程序清单 10.2 μC/OS-II 中的 INCLUDES.H

```
/*
*****
*          INCLUDES.H
*****
*/
#include <STDIO.H>
#include <STRING.H>
#include <CTYPE.H>
#include <STDLIB.H>
#include <CONIO.H>
#include <DOS.H>

#include "\SOFTWARE\uCOS-II\IX86L\OS_CPU.H"
#include "OS_CFG.H"
#include "\SOFTWARE\uCOS-II\SOURCE\uCOS_II.H"
```

10.4.3 数据类型的修改

```
/* μC/OS 数据类型 */
typedef unsigned char OS_BYTE; /* 无符号 8 位数 */
typedef signed char OSINT8; /* 有符号 8 位数 */
typedef unsigned int OSWORD; /* 无符号 16 位数 */
typedef signed int OSINT16; /* 有符号 16 位数 */
typedef unsigned long OSLONG; /* 无符号 32 位数 */
typedef signed long OSINT32; /* 有符号 32 位数 */

/* μC/OS-II 数据类型 */
typedef unsigned char OSQ8U; /* 无符号 8 位数 */
typedef signed char OSQ8S; /* 有符号 8 位数 */
typedef unsigned int OSQ16U; /* 无符号 16 位数 */
typedef signed int OSQ16S; /* 有符号 16 位数 */
typedef unsigned long OSQ32U; /* 无符号 32 位数 */
typedef signed long OSQ32S; /* 有符号 32 位数 */
```

```
#define OS_STK_TYPE OSQ32S /* 在 μC/OS 中 */
#define OS_STK OSQ32S /* 在 μC/OS-II 中 */
```

超星阁浏览器提醒您：
 使用本复制品
 请尊重相关知识产权！

在μC/OS中，任务栈定义为类型OS_STK_TYPE，而在μC/OS-II中任务栈要定义类型OS_STK。为了免于修改所有应用程序的文件，可以在OS_CPU.H中建立两个数据类型，以Intel 80x86为例，如程序清单10.4所示。

程序清单 10.4 μC/OS 和 μC/OS-II 任务栈的数据类型

10.2.2 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()

μC/OS-II和μC/OS一样，分别定义两个宏来开中断和关中断：OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()。在μC/OS向μC/OS-II升级的时候，用户不必动这两个宏。

10.2.3 OS_STK_GROWTH

大多数微处理器和微处理器的堆栈都是由存储器高地址向低地址操作的，然而有些微处理器的工作方式正好相反。μC/OS-II设计成通过定义一个常数OS_STK_GROWTH来处理不同微处理器栈操作的取向：

对栈操作由低地址向高地址增长，设OS_STK_GROWTH为0：

对栈操作由高地址向低地址递减，设OS_STK_GROWTH 为 1。

有些新的常数定义（#define constants）在μC/OS中是没有的，因此要加到OS_CPU.H中去。

10.2.4 OS_TASK_SW()

OS_TASK_SW()是一个宏，从μC/OS升级到μC/OS-II时，这个宏不需要改动。当μC/OS-II从低优先级的任务向高优先级的任务切换时要用到这个宏，OS_TASK_SW()的调用总是出

void OS_Task (void *param)

param = param;

while (1);

因为Intel 80x86的结构特点，在μC/OS中使用过OS_FAR。这个定义语句（#define）在μC/OS-II 中去掉了，因为这条定义使移植变得不方便。因此对于Intel 80x86，如果用户定义在大模式下编译时，所有存储器属性都将为远程（FAR）。

在μC/OS-II中，任务返回值类型定义如程序清单10.5所示。用户可以重新编辑所有OS_FAR的文件，或者在μC/OS-II中将OS_FAR定义为空，去掉OS_FAR，以实现向μC/OS-II的升级。

程序清单 10.5 在 μC/OS 中任务函数的定义

10.3 OS_CPU_A.ASM

移植μC/OS 和μC/OS-II 需要用户用汇编语言写4个相当简单的函数。

OSStartHighRdy()

OSCtxSw()

OSIntCtxSw()

OSTickISR()

10.3.1 OSStartHighRdy()

```
OSStartHighRdy()
CALL OSTaskSwHook();
Set OSRunning to 1;
将 OSTCBHighRdy->OSTCBLinkLst 装入处理器的栈指针;
从栈中弹出所有寄存器的值;
执行中断返回指令;
```

调用OSSTaskSwHook()。OSTaskSwHook()这个函数的栈指针装入CPU之前要先调用OSTaskSwHook()。

OSTaskSwHook()之后立即将OSRunning设为1。程序清单10.6给出OSStartHighRdy()的示意代码。μC/OS只有其中最后三步。

程序清单 10.6 OSStartHighRdy() 的示意代码

```
OSStartHighRdy()
所有处理器寄存器的值装入当前任务栈;
将栈指针装入OSPCur->OSTCBLinkLst;
CALL OSTaskSwHook(); (1)
OSTCBCur = OSTCBHighRdy;
OSPCur = OSPrerigHighRdy; (2)
将 OSTCBHighRdy->OSTCBLinkLst 装入处理器的栈指针;
从栈中弹出所有寄存器的值;
执行中断返回指令;
```

10.3.2 OSCtxSw()

在μC/OS-II中，任务切换要增作两件事，首先，将当前任务栈指针保存到当前任务控制块TCB后要立即调用OSTaskSwHook()。其次，在装载新任务的栈指针之前必须将OSPrerig设为OSPrerigHighRdy。OSCtxSw()的示意代码如程序清单10.7所示。μC/OS-II加上了步骤(1)和(2)。

程序清单 10.7 OSCtxSw() 的示意代码

10.3.3 OSIntCtxSw()

如同上述函数一样，在μC/OS-II中，OSCtxSw()也增加了两件事。首先，将当前任务

后要立即调用OSTaskSwHook()。其次，在装载新任务时调用OSTaskCreateHook()，引起的操作系统的参数，保存到OSTaskCreate()→OSTaskCreateHook()引起的操作系统的参数，从程序中弹出所有寄存器的值，执行中断返回指令。

码

盗墓浏览器提醒您：
使用本复制品
请尊重相关知识产权！

10.3.4 OSTickISR()

在μC/OS-II和μC/OS 中，这个函数的代码是一样的，无须改变。

10.4 OS_CPU_C.C

移植 μC/OS-II 需要用C语言写6个非常简单的函数：

OSTaskStkInit()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskSwHook()
OSTaskStatHook()
OSTimeTickHook()

其中只有一个函数OSTaskStkInit()是必不可少的。其他5个只需定义，而不包括任何代码。

10.4.1 OTaskStkInit()

在μC/OS中，OSTaskCreate()被认为是与使用的微处理器类型有关的函数。实际上这个函数中只有一部分内容是依赖于微处理器类型的。在μC/OS-II中，与使用的微处理器类型有关的那一部分已经从函数OSTaskCreate() 中抽出来了，放在一个叫作OSTaskStkInit()的函

数中。

OSTaskStkInit()口负责设置任务的堆

```
WORD OS_TaskCreate(void (*task)(void *pd), void *pdata, void *stack,
                   WORD stack_size, WORD priority, void *parent_stk,
                   WORD exit_code);
```

使之看起来好像中断刚刚发生过，所有的CPU寄存器，程序清单10.9给出Intel 80x86实模式，在大模式下汇编语言。程序清单10.10是同类微微处理器的μC/OS-II的代码，可以看出：从 [程序清单10.9(1)] OS_EXIT_HOOK 和 OS_EXIT_HOOK()都抽出来并移到了OSTaskStkInit()中。

OSTaskCreate()

郑重申明
该程序为原创作品
未经授权不得使用！

```

    if (err == OS_NO_ERR) {
        if (OSRunning) {
            OSSched();
        }
    } else {
        OSTCBPrioTbl[p] = (OS_TCB *)0;
    }
    return (err);
} else {
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
}

```

程序清单 10.10 μC/OS-II 中的 OSTaskStkInit()

```

void *OSTaskStkInit(void (*task)(void *pd), void *pdata, void *ptos, INT16U opt)
{
    INT16U *stk;

    opt    = opt;
    stk    = (INT16U *)ptos;
    *stk-- = (INT16U)FP_SEG(pdata);
    *stk-- = (INT16U)FP_OFF(pdata);
    *stk-- = (INT16U)FP_SEG(task);
    *stk-- = (INT16U)FP_OFF(task);
    *stk-- = (INT16U)0x0202;
    *stk-- = (INT16U)FP_SEG(task);
    *stk-- = (INT16U)FP_OFF(task);
    *stk-- = (INT16U)0xAAAA;
    *stk-- = (INT16U)0xCCCC;
    *stk-- = (INT16U)0xDDDD;
    *stk-- = (INT16U)0xBBBB;
    *stk-- = (INT16U)0x0000;
    *stk-- = (INT16U)0x1111;
    *stk-- = (INT16U)0x2222;
}

```

```
#include <os.h>
#include <os_cpu_hooks.h>
void _task(void *ptcb)
{
    /* Task code */
}
```

μC/OS 升级到μC/OS-II

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

```
#if OS_CPU_HOOKS_EN
OSTaskCreateHook(OS_HOOK *ptcb)
{
    ptcb = ptcb;
}
#endif
```

OSTaskCreateHook()在μC/OS中没有，如程序清单10.11所示，在由μC/OS 向μC/OS-II 升级时，定义一个空函数就可以了。注意其中的赋值语句，如果不把ptcb赋给ptcb，有些编译器会产生一个警告错误，说定义的ptcb变量没有用到。

程序清单 10.11 μC/OS-II 中的 OSTaskCreateHook()

```
#if OS_CPU_HOOKS_EN
OSTaskDelHook(OS_HOOK *ptcb)
{
    ptcb = ptcb;
}
#endif
```

用户还应该使用条件编译管理指令来处理这个函数。只有在OS_CFG.H 文件中将OS_CPU_HOOKS_EN设为1时，OSTaskCreateHook()的代码才会生成。这样做好处是允许用户移植时可在不同文件中定义钩子函数。

10.4.3 OSTaskDelHook()

OSTaskDelHook() 这个函数在μC/OS中没有，如程序清单10.12所示，从μC/OS 到μC/OS-II，只要简单地定义一个空函数就可以了。注意，如果不用赋值语句将ptcb赋值为ptcb，有些编译程序可能会产生一些警告信息，指出定义的ptcb变量没有用到。

程序清单 10.12 μC/OS-II 中的 OSTaskDelHook()

还是要用条件编译管理指令来处理这个函数。只有把OS_CFG.H文件中的OS_CPU_HOOKS_EN设为1，OSTaskDelHook()的代码才能生成。这样做的好处是允许用户移植时在



OSTaskSwHook()在μC/OS中也不存在。从μC/OS向μC/OS-II升级时，只要简单地定义一个空函数就可以了，如程序清单10.13所示。

程序清单 10.13 μC/OS-II 中的 OSTaskSwHook() 函数



还是要用编译管理指令来处理这个函数。只有把OS_CFG.H文件中的OS_CPU_HOOKS_EN设为1，OSTaskSwHook()的代码才能生成。

10.4.5 OSTaskStatHook()

OSTaskStatHook()在μC/OS中不存在，从μC/OS向μC/OS-II升级时，只要简单地定义一个空函数就可以了，如程序清单10.14所示。

还是要用编译管理指令来处理这个函数。只有把OS_CFG.H文件中的OS_CPU_HOOKS_EN设为1，OSTaskSwHook()的代码才能生成。

程序清单 10.14 μC/OS-II 中的 OSTaskStatHook() 函数

10.4.6 OSTimeTickHook()

OSTimeTickHook()在μC/OS中不存在，从μC/OS向μC/OS-II升级时，只要简单地定义一个空函数就可以了，如程序清单10.15所示。

```
#include "os.h"
#include "os_ticks.h"
#include "os_ticks.h"

```

只有把OS_CFG.H 文件中的OS_CPU_HOOKS_EN 成。

程序清单 10.15 μC/OS-II 中的 OSTimeTickHook()

μCOS	μCOS-II
Processor_name.H	OS_CPU.H
数据类型:	数据类型:
BYTE	INT8U
YTE	INT8S
UWORD	INT16U
WORD	INT16S
ULONG	INT32U
LONG	INT32S
OS_STK_TYPE	OS_STK
OS_ENTER_CRITICAL()	不变
OS_EXIT_CRITICAL()	不变
—	增加了 OS_STK_GROWTH
OS_TASK_SW()	不变
OS_PAR	定义OS_PAR 为空，或返回所有的 OS_PAR
Processor_name.ASM	OS_CPU.ASM
OSTaskSwkHdly()	增加了调用 OSTaskSwHook(); 或 OSRunning = 1(8 bit)
OSChkHdly()	增加了调用 OSTaskSwHook(); 将 OSPrvHighHdly 和 OSPrvCur(8 bit)
OSIntCntr()	增加了调用 OSTaskSwHook(); 将 OSPrvHighHdly 和 OSPrvCur(8 bit)
OSTaskEnd()	不变

级需要改得地方。其中processor_name.?是μC/OS

C/OS-I 要修改的地方

续表

μC/OS	μC/OS-II
<i>Processor_name.C</i>	OS_CPU_C.C
OSTaskCreate()	抽出栈初始部分，放在函数 OSTaskStkInit() 中
—	增加了空函数 OSTaskCreateHook()
—	增加了空函数 OSTaskDelHook()
—	增加了空函数 OSTaskSwHook()
—	增加了空函数 OSTaskStatHook()
—	增加了空函数 OSTimeTickHook()

第六章 总结

本章主要介绍了 μC/OS-II 的移植方法。移植工作从移植到移植完成，大致分为以下三个阶段：

1) 移植前的准备工作：包括移植平台的准备、移植环境的准备。

2) 移植实现：包括移植代码的实现、移植代码的测试与优化。

3) 移植后的维护：包括移植代码的维护、移植环境的维护。

通过本章的学习，读者应该能够掌握 μC/OS-II 的移植方法，从而能够顺利地将 μC/OS-II 移植到自己的平台上。

在学习本章时，读者需要注意以下几点：

1) 移植前的准备工作：包括移植平台的准备、移植环境的准备。

2) 移植实现：包括移植代码的实现、移植代码的测试与优化。

3) 移植后的维护：包括移植代码的维护、移植环境的维护。

通过本章的学习，读者应该能够掌握 μC/OS-II 的移植方法，从而能够顺利地将 μC/OS-II 移植到自己的平台上。

第 11 章

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

参 考 手 册

本章提供了μC/OS-II 的用户指南。每一个用户可以调用的内核函数都按字母顺序加以说明，包括：

- 函数的功能描述
- 函数原型
- 函数名称及源代码
- 函数所使用的常量
- 函数参数
- 函数返回值
- 特殊说明和注意点

OSInit()

```
void OSInit(void);
```

所属文件	调用者	开关量
OS_CORE.C	启动代码	无

OSinit()初始化μC/OS-II，对这个函数的调用必须在调用 OSStart()函数之前，而 OSStart()函数真正开始运行多任务。

参数

无

返回值

无

注意/警告

必须先于 OSStart()函数的调用。

范例

```

OSInit(); /* 初始化μC/OS-II */

OSStart(); /* 启动多任务内核 */

```

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

OSIntEnter()

```
void OSIntEnter ( void ) ;
```

所属文件	调用者	开关量
os.h	中断	无

该函数正在执行，这有助于μC/OS-II掌握中断嵌套层数（OSIntNesting），这样可以避免调用函数所带来的额外的开销。

参数

无

返回值

无

注意/警告

在任务级不能调用该函数。

如果系统使用的处理器能够执行自动的独立执行读取-修改-写入的操作，那么就可以直接递增中断嵌套层数（OSIntNesting），这样可以避免调用函数所带来的额外的开销。

范例一

（Intel 80x86 的实模式，在大模式下编译）

超星阅览器提醒您：
使用本复制品
请尊重相关知识版权！

编译)

OSIntExit()

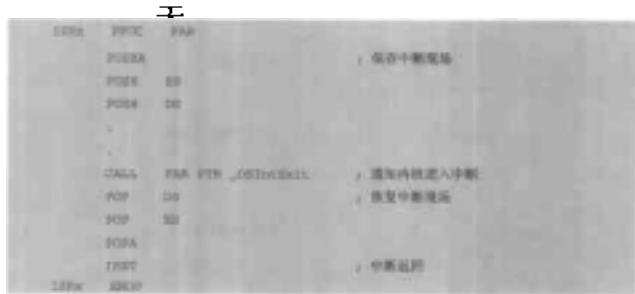
void OSIntExit (void) ;

所属文件	调用者	开关量
OS_CORE.C	中断	无

OSIntExit()通知μC/OS-II一个中断服务已执行完毕，这有助于μC/OS-II掌握中断嵌套

的情况。通常 OSIntExit()和 OSIntEnter()联合使用。当最后一层嵌套的中断执行完毕后，如果有更高优先级的任务准备就绪，μC/OS-II会调用任务调度函数，在这种情况下，中断返回到更高优先级的任务而不是被中断了的任务。

参数



即使没有调用 OSIntEnter()而是使用直接递增 OSIntNesting 的方法，也必须调用 OSIntExit()函数。

范例

(Intel 80x86 的实模式，在大模式下编译)

OSMboxAccept()

```
void *OSMboxAccept (OS_EVENT *pevent) ;
```

所属文件	调用者	开关量
OS_MBOX.C	任务或中断	OS_MBOX_EN

OSMboxAccept()函数查看指定的消息邮箱是否有需要的消息。不同于 OSMboxPend()函数，如果没有需要的消息，OSMboxAccept()函数并不挂起任务。如果消息已经到达，该消息被传递到用户任务并且从消息邮箱中清除。通常中断调用该函数，因为中断不允许挂

起等待消息。

参数

pevent 是指向需要查看的消息邮箱的指针。当建立消息邮箱时，该指针返回到用户程

的指针；如果消息邮箱没有消息，返回空指针。



OSMboxCreate()

OS_EVENT *OSMboxCreate (void *msg) ;

所属文件	调用者	开关量
OS_MBOX.C	任务或启动代码	OS_MBOX_EN

OSMboxCreate()建立并初始化一个消息邮箱。消息邮箱允许任务或中断向其他一个或几个任务发送消息。

参数

mco 参数用来初始化建立的消息邮箱。如果该指针不为空，建立的消息邮箱将含有消

控块的指针。如果没有可用的事件控制块，返

必须先建立消息邮箱，然后使用。

范例

OSMboxPend()

```
void *OSMboxPend ( OS_EVNNT *pevent, INT16U timeout, int8u *err );
```

所属文件	调用者	开关量
OS_MBOX.C	任务	OS_MBOX_EN

OSMboxPend()用于任务等待消息。消息通过中断或另外的任务发送给需要的任务。消息是一个以指针定义的变量，在不同的程序中消息的使用也可能不同。如果调用 OSMboxPend() 函数时消息邮箱已经存在需要的消息，那么该消息被返回给 OSMboxPend() 的调用者，消息

邮箱中清除该消息。如果调用 OSMboxPend() 函数时消息邮箱中没有需要的消息，OSMboxPend() 函数挂起当前任务直到得到需要的消息或超出定义等待超时的时间。如果同时有多个任务等待同一个消息，μC/OS-II 默认最高优先级的任务取得消息并且任务恢复执行。一个由 OSTaskSuspend() 函数挂起的任务也可以接受消息，但这个任务将一直保持挂起状态直到通过调用 OSTaskResume() 函数恢复任务的运行。

参数

pevent 是指向即将接受消息的消息邮箱的指针。该指针的值在建立该消息邮箱时可以得到。[参考 OSMboxCreate() 函数]。

timeout 允许一个任务在经过了指定数目的时钟节拍后还没有得到需要的消息时恢复运行。如果该值为零表示任务将持续的等待消息。最大的等待时间为 65535 个时钟节拍。这个时间长度并不是非常严格的，可能存在一个时钟节拍的误差，因为只有在一个时钟节拍结束后才会减少定义的等待超时时钟节拍。

。 OSMboxPend() 函数返回的错误码可能为下述几

。受。

- **OS_TIMEOUT:** 消息没有在指定的周期数内送到。
- **OS_ERR_PEND_ISR:** 从中断调用该函数。虽然规定了不允许从中断调用该函数，但 μC/OS-II 仍然包含了检测这种情况的功能。
- **OS_ERR_EVENT_TYPE:** pevent 不是指向消息邮箱的指针。

返回值

OSMboxPend() 函数返回接受的消息并将 *err 置为 OS_NO_ERR。如果没有在指定数目的时钟节拍内接受到需要的消息，OSMboxPend() 函数返回空指针并且将 *err 设置为 OS_TIMEOUT。

注意/警告

必须先建立消息邮箱，然后使用。

不允许从中断调用该函数。

范例

```

    posra + pdata);
    for( i=0; i<
        msg = OSMboxRead(Combox, 10, &err);
        if( err == OS_MQ_ERROR ) {
            /* 没有正确的接收 */
        }
    else {
        /* 在规定时间内没有接收到消息 */
    }
}

```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

OSMboxPost()

INT8U OSMboxPost (OS_EVENT *pevent, void *msg) ;

所属文件	调用者	开关量
OS_MBOX.C	任务或中断	OS_MBOX_EN

*OSMboxPost()*函数通过消息邮箱向任务发送消息。消息是一个指针长度的变量，在不同的程序中消息的使用也可能不同。如果消息邮箱中已经存在消息，返回错误码说明消息邮箱已满。*OSMboxPost()*函数立即返回调用者，消息也没有能够发到消息邮箱。如果有任何任务在等待消息邮箱的消息，最高优先级的任务将得到这个消息。如果等待消息的任务优先级比发送消息的任务优先级高，那么高优先级的任务将得到消息而恢复执行，也就是说，将发生一次任务切换。

参数

pevent 是指向即将接受消息的消息邮箱的指针。该指针的值在建立该消息邮箱时可以得到[参考 *OSMboxCreate()*函数]。

msg 是即将实际发送给任务的消息。消息是一个指针长度的变量，在不同的程序中消

息的使用也可能不同。不允许传递一个空指针，因为这意味着消息邮箱为空。

返回值

*OSMboxPost()*函数的返回值为下述之一：

消息邮箱中。

经包含了其他消息，不空。

t 不是指向消息邮箱的指针。



着消息邮箱为空。

```
OSL_WSTR *Combox;
INT8U ComboxPost(OSL_WSTR *pdata)
{
    INT8U ret;
    OS_MBOX_DATA *pdata;
    ret = 0;
    pdata = pdata;
    if (ret) {
        err = OSMboxPost(Combox, (void *)ComboxData);
    }
}
```

OSMboxQuery()

INT8U OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata) ;

所属文件	调用者	开关量
OS_MBOX.C	任务或中断	OS_MBOX_EN

OSMboxQuery()函数用来取得消息邮箱的信息。用户程序必须分配一个 OS_MBOX_DATA 的数据结构，该结构用来从消息邮箱的事件控制块接受数据。通过调用 OSMboxQuery() 函数可以知道有多少个任务在等待消息，还可以检查消息邮箱现

```
void *pMem;
INT8U osEventId(OS_EVENT_TYPE_ECODE);
INT8U osEventData;
```

参数

pevent 是指向即将接受消息的消息邮箱的指针。该指针的值在建立该消息邮箱时可以得到[参考 OSMboxCreate()函数]。

pdata 是指向 OS_MBOX_DATA 数据结构的指针，该数据结构包含下述成员：

```
OS_EVENT *pMem;
void Task(void *pdata)
{
    OS_MBOXDATA *mbox_data;
    mbox_data = pdata;
    for(;;)
    {
        err = OSMboxQuery(pevent, mbox_data);
        if(err == OS_NO_ERR)
        {
            /* 消息邮箱中消息为空 */
            /* 消息邮箱等待队列为空 */
        }
    }
}
```

之一：

- **OS_NO_ERR:** 调用成功。
- **OS_ERR_EVENT_TYPE:** pevent 不是指向消息邮箱的指针。

注意/警告

必须先建立消息邮箱，然后使用。

范例

```
if (err == OS_NO_ERR) {  
    /* 如果 mem->data.blks 为零或为负，这时将忽略此参数 */
```

OSMemCreate()

```
OS_MEM *OSMemCreate( void *addr, INT32U nblks ,INT32U blksize, INT8U *err);
```

所属文件	调用者	开关量
OS_MEM.C	任务或初始代码	OS_MEMORY_EN

OSMemCreate()函数建立并初始化一块内存区。一块内存区包含指定数目的大小确定的内存块。程序可以包含这些内存块并在用完后释放回内存区。

参数

addr 建立的内存区的起始地址。内存区可以使用静态数组或在初始化时使用 **malloc()** 函数建立。

nblks 需要的内存块的数目。每一个内存区最少需要定义两个内存块。

blksize 每个内存块的大小，最少应该能够容纳一个指针。

err 是指向包含错误码的变量的指针。**OSMemCreate()**函数返回的错误码可能为下述几种：

OS_NO_ERR：成功建立内存区。

OS_MEMORY_INVALID_PART：没有空闲的内存区。

OS_MEMORY_INVALID_BLKS：没有为每一个内存区建立至少两个内存块。

OS_MEMORY_INVALID_SIZE：内存块大小不足以容纳一个指针变量。

返回值

OSMemCreate()函数返回指向内存区控制块的指针。如果没有剩余内存区，**OSMemCreate()**函数返回空指针。

注意/警告

必须首先建立内存区，然后使用。

范例

```

void main(void)
{
    DEBInit();
    /* 初始化C/OS-II */
    CommBuf = OSMemCreate(&CommBuf[16], 16, 128, &err);
    OSStart();
    /* 启动多任务内核 */
}

```



OSMemGet()

void *OSMemGet(OS_MEM *pmem, INT8U *err);

所属文件	调用者	开关量
OS_MEM.C	任务或中断	OS_MEM_EN

OSMemGet()函数用于从内存区分配一个内存块。用户程序必须知道所建立的内存块的大小，同时用户程序必须在使用完内存块后释放内存块。可以多次调用 **OSMemGet()**函数。

参数

pmem 是指向内存区控制块的指针，可以从 **OSMemCreate()**函数返回得到。

err 是指向包含错误码的变量的指针。**OSMemGet**（函数返回的错误码可能为下述几种：

- **OS_NO_ERR**: 成功得到一个内存块。
- **OS_MEM_NO_FREE_BLKS**: 内存区已经没有空间分配给内存块。

返回值

OSMemGet()函数返回指向内存区块的指针。如果没有空间分配给内存块，**OSMemGet()**函数返回空指针。

注意/警告

必须首先建立内存区，然后使用。

```

void Task(void *pdata)
{
    INT8U *pbuf;

    pdata = pdata;
    for (;;) {
        msg = OSMemGet(CommMem, 4096);
        if (msg != (INT8U *)0) {
            /* 内存块已分配 */
        }
    }
}

```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

OSMemPut()

INT8U OSMemPut(OS_MEM *pmem, void *pbblk);

所属文件	调用者	开关量
OS_MEM.C	任务或中断	OS_MEM_EN

OSMemPut()函数释放一个内存块，内存块必须释放回原先申请的内存区。

参数

pmem 是指向内存区控制块的指针，可以从 **OSMemCreate()**函数返回得到。

pbblk 是指向将被释放的内存块的指针。

返回值

OSMemPut()函数的返回值为下述之一：

OS_NO_ERR：成功释放内存块

OS_MEM_FULL：内存区已经不能再接受更多释放的内存块。这种情况说明用户程序

出现了错误，因为释放的内存块比 `OSMemGet()` 函数得到的多。

```
OS_MEM *ComMem;
INT8U *ComData;
void task(void *pdata)
{
    INT8U *err;

    pdata = pdata;
    for(;;)
    {
        err = OSMemPut(ComMem, (void *)ComData);
        if (err == OS_MEN_FREE) /* 释放内存 */
    }
}
```



OSMemQuery()

`INT8U OSMemQuery(OS_MEMORY *pmem, OS_MEMORY_DATA *pdata);`

所属文件	调用者	开关量
OS_MEMORY.C	任务或中断	OS_MEMORY_EN

`OSMemQuery()` 函数得到内存区的信息。该函数返回 `OS_MEMORY` 结构包含的信息，但使用了一个新的 `OS_MEMORY_DATA` 的数据结构。`OS_MEMORY_DATA` 数据结构还包含了正被使用的内存块数目的域。

参数

`pmem` 是指向内存区控制块的指针，可以从 `OSMemCreate()` 函数返回得到。

```

void *OSMemAlloc(          /* 指向内存区起始地址的指针
    void *OSMemList;        /* 指向空闲内存块列表起始地址的指针
    INT32U OSAllocSize;     /* 每个内存块的大小
    INT32U OSAllocTotal;    /* 该内存区的内存块总数
    INT32U OSAllocUsed;     /* 使用的内存块数
)

```

据结构的指针，该数据结构包含了以下的域：

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

```

OS_MEM *OSMemAlloc(
    void TaskCreate *pTask,
    INT32U err,
    OS_MEM_DATA *mem_data,
    pTask->pdata,
    Err (err) {
        err = OSMemQuery (mem_data, mem_data);
    }
)

```

NO_ERR。

必须首先建立内存区，然后使用。

范例

OSQAccept()
void *OSQAccept(OS_EVENT *pevent);

所属文件	调用者	开关量
OS_Q.C	任务或中断	OS_Q_EN

OSQAccept()函数检查消息队列中是否已经有需要的消息。不同于 OSQPend()函数，如果没有需要的消息，OSQAccept()函数并不挂起任务。如果消息已经到达，该消息被传递到用户任务。通常中断调用该函数，因为中断不允许挂起等待消息。

参数

```
os_event *ComQ;
```

```
void Task (void *pdata)
```

```
{
```

```
    void *msg;
```

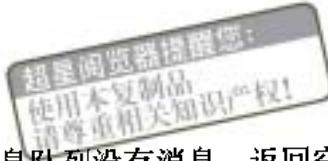
```

    pdata = pdata;
    for (i = 0;
```

```
        msg = OSQAccept(ComQ); /* 检查消息队列 */
    if (msg != (void *)0) { /* 处理接受的消息 */
        /* ... */
    } else { /* 没有消息 */
        /* ... */
    }
}
```

指针。当建立消息队列时，该指针返回到用户程

指针；如果消息队列没有消息，返回空指针。



OSQCreate()

```
OS_EVENT *OSQCreate( void **start, INT8U size);
```

所属文件	调用者	开关量
OS_Q.C	任务或启动代码	OS_Q_EN

OSQCreate()函数建立一个消息队列。任务或中断可以通过消息队列向其他一个或多个任务发送消息。消息的含义是和具体的应用密切相关的。

```
OS_EVENT *ComQ;
void *Data[10];

void main(void)
{
    OSInit();
    /* 初始化OS */

    ComQ = OSQCreate(10, 10);
    /* 建立消息队列 */

    OSInit();
    /* 启动多任务内存 */
}
```

（内存区是一个指针数组。

队列事件控制块的指针。如果没有空余的事件空闲

注意/警告

必须先建立消息队列，然后使用。

范例

OSQFlush()

INT8U *SOQFlush (OS_EVENT *pevent) ;

所属文件	调用者	开关量
OS_Q.C	任务或中断	超星 OS_Q_EN 使用方法参见手册 请尊重相关知识产权

OSQFlush()函数清空消息队列并且忽略发送往队列的所有消息。不管队列中是否有消息，这个函数的执行时间都是相同的。

该指针的值在建立该队列时可以得到[参考

空；

- **OS_ERR_EVENT_TYPE:** 试图消除不是消息队列的对象。

注意/警告

必须先建立消息队列，然后使用。

范例

OSQPend()

```
void *OSQPend( OS_EVENT *pevent, INT16U timeout, INT8U *err);
```

所属文件	调用者	开关量
OS_Q.C	任务	OS_Q_EN

OSQPend()函数用于任务等待消息。消息通过中断或另外的任务发送给需要的任务。消息是一个以指针定义的变量，在不同的程序中消息的使用也可能不同。如果调用 **OSQPend()** 函数时队列中已经存在需要的消息，那么该消息被返回给 **OSQPend()** 函数的调用者，队列中清除该消息。如果调用 **OSQPend()** 函数时队列中没有需要的消息，**OSQPend()** 函数挂起当前任务直到得到需要的消息或超出定义的超时时间。如果同时有多个任务等待同一个消息，μC/OS-II 默认最高优先级的任务取得消息并且任务恢复执行。一个由 **OSTaskSuspend()** 函数挂起的任务也可以接受消息，但这个任务将一直保持挂起状态直到通过调用 **OSTaskResume()** 函数恢复任务的运行。

参数

pevent 是指向即将接受消息的队列的指针。该指针的值在建立该队列时可以得到[参考 **OSMboxCreate()** 函数]。

timeout 允许一个任务在经过了指定数目的时钟节拍后还没有得到需要的消息时恢复运行状态。如果该值为零表示任务将持续的等待消息。最大的等待时间为 65535 个时钟节拍。这个时间长度并不是非常严格的，可能存在一个时钟节拍的误差，因为只有在一个时钟节拍结束后才会减少定义的等待超时时钟节拍。

err 是指向包含错误码的变量的指针。**OSQPend()** 函数返回的错误码可能为下述几种：

- **OS_NO_ERR:** 消息被正确的接受。
- **OS_TIMEOUT:** 消息没有在指定的周期数内送到。
- **OS_ERR_PEND_ISR:** 从中断调用该函数。虽然规定了不允许从中断调用该函数，但 μC/OS-II 仍然包含了检测这种情况的功能。
- **OS_ERR_EVENT_TYPE:** **pevent** 不是指向消息队列的指针。

返回值

OSQPend() 函数返回接受的消息并将 ***err** 置为 **OS_NO_ERR**。如果没有在指定数目的时钟节拍内接受到需要的消息，**OSQPend()** 函数返回空指针并且将 ***err** 设置为 **OS_TIMEOUT**。

注意/警告

必须先建立消息邮箱，然后使用。

不允许从中断调用该函数。

```

OS_EVENT *ComQ;
void *msg;

pdata = pdata;
Evt_1111 = 1;

msg = OSQPend(ComQ, 100, &err);
if (err == os_NO_BLOCK) {
    /* 在指定时间内没有接收到消息 */
} else {
    /* 在指定的时间内没有使用到指定的消息 */
}

```



OSQPost()

INT8U OSQPost(OS_EVENT *pevent, void *msg);

所属文件	调用者	开关量
OS_Q.C	任务或中断	OS_Q_EN

OSQPost()函数通过消息队列向任务发送消息。消息是一个指针长度的变量，在不同的程序中消息的使用也可能不同。如果队列中已经存满消息，返回错误码。**OSQPost()**函数立即返回调用者，消息也没有能够发到队列。如果有任何任务在等待队列中的消息，最高优



先级的任务将得到这个消息。如果等待消息的任务优先级比发送消息的任务优先级高，那么高优先级的任务将得到消息而恢复执行，也就是说，发生了一次任务切换。消息队列是先入先出（FIFO）机制的，先进入队列的消息先被传递给任务。

参数

pevent 是指向即将接受消息的消息队列的指针。该指针的值在建立该队列时可以得到[参考 OSQCreate()函数]。

msg 是即将实际发送给任务的消息。消息是一个指针长度的变量，在不同的程序中一个空指针。

```
OS_EVENT *ComQ;
INT32 ComQueue(100);

void ComTask0(void *pdata)
{
    INT32 msg;
    pdata = pdata;
    msg = 1;
    :
    /* 消息队列中。
       满。
    */
    if (msg == OSQPut(ComQ, msg) != OS_ERROR) {
        /* 将消息放入消息队列 */
    }
}
```

■消息队列中。

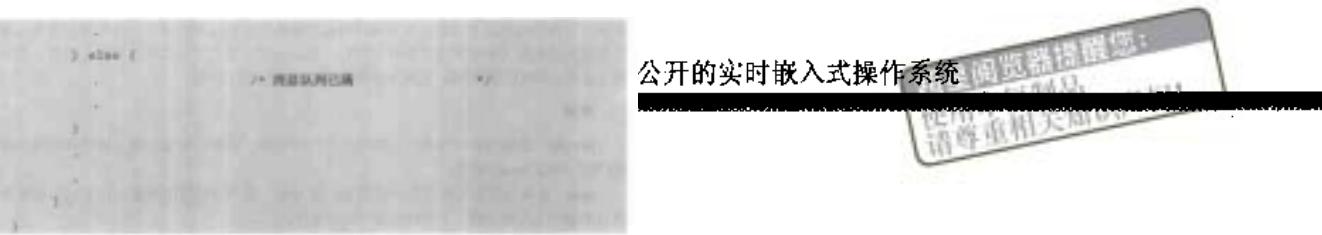
■满。

nt 不是指向消息队列的指针。

必须先建立消息队列，然后使用。

不允许传递一个空指针。

范例

***OSQPostFront()***

```
INT8U OSQPostFront(OS_EVENT *pevent, void *msg);
```

所属文件	调用者	开关量
OS_Q.C	任务或中断	OS_Q_EN

OSQPostFront()函数通过消息队列向任务发送消息。**OSQPostFront()**函数和**OSQPost()**函数非常相似，不同之处在于**OSQPostFront()**函数将发送的消息插到消息队列的最前端。也就是说，**OSQPostFront()**函数使得消息队列按照后入先出（LIFO）的方式工作，而不是先入先出（FIFO）。消息是一个指针长度的变量，在不同的程序中消息的使用也可能不同。如果队列中已经存满消息，返回错误码。**OSQPost()**函数立即返回调用者，消息也没能发到队列。如果有任何任务在等待队列中的消息，最高优先级的任务将得到这个消息。如果等待消息的任务优先级比发送消息的任务优先级高，那么高优先级的任务将得到消息而恢复执行，也就是说，发生了一次任务切换。

参数

pevent 是指向即将接受消息的消息队列的指针。该指针的值在建立该队列时可以得到[参考**OSQCreate()**函数]。

msg 是即将实际发送给任务的消息。消息是一个指针长度的变量，在不同的程序中消息的使用也可能不同。不允许传递一个空指针。

返回值

OSQPost()函数的返回值为下述之一：

- **OS_NO_ERR**: 消息成功的放到消息队列中。
- **OS_MBOX_FULL**: 消息队列已满。
- **OS_ERR_EVENT_TYPE**: **pevent** 不是指向消息队列的指针。

```

OS_EVENT *ComQ;
INT8U    ComQout(100);

void ComQData(OS_Q_DATA *pdata)
{
    INT8U  ret;
    pdata = pdata;
    for (zz) {
        ret = OSQPGetFront(&ComQ, (void *)ComQdata(zz));
        if (ret == OS_NO_BLOCK) {
            /* 将消息放入消息队列 */
        }
        else {
            /* 消息队列已满 */
        }
    }
}

```

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

OSQuery()

INT8U OSQuery(OS_EVENT *pevent, OS_Q_DATA *pdata);

所属文件	调用者	开关量
OS_Q.C	任务或中断	OS_Q_EN

OSQuery()函数用来取得消息队列的信息。用户程序必须建立一个 OS_Q_DATA 的数据结构，该结构用来保存从消息队列的事件控制块得到的数据。通过调用 OSQuery()函数

可以知道任务是否在等待消息、有多少个任务在等待消息、队列中有多少消息以及消息队列中消息的大小。

通过调用 OSQuery() 函数，可以得到即将被传递给任务的消息的信息。

```
void *OSQuery(OSQHandle hOSQ,
              OSQEventTl *pevent,
              OSQSize *osqsize,
              OSQEventTl *osqevent,
              OSQEventSize *osqeventsizes);
```



pevent 是指向即将接受消息的消息邮箱的指针。该指针的值在建立该消息邮箱时可以得到[参考 OSQCreate() 函数]。

pdata 是指向 OS_Q_DATA 数据结构的指针，该数据结构包含下述成员：

```
OS_QDATA *pdata;
void Task(void *pdata);
OS_Q_DATA qdata;
OSMID osmid;
OSQEvent osqevent;
OSQEventSize osqeventsizes;
```

- OS_ERR_EVENT_TYPE : pevent 不是指向消息队列的指针。

注意/警告

必须先建立消息队列，然后使用。

范例

***OSSchedLock()*****void OSSchedLock(void);**

所属文件	调用者	开关量
OS_CORE.C	任务或中断	N/A

```
void Task3(void *pdata)
{
    pdata = pdata;
    for(;;)
        OSSchedLock(); // 禁止任务调度
}
```

只有使用配对的函数 OSSchedUnlock()才能重新开启任务调度。调用该函数的任务独占 CPU，不管有没有其他高优先级任务被接受和执行（中断必须允许）。OSSchedLock()函数和 OSSchedUnlock()函数必须配对使用。 μ C/OS-II 可以支持多达 254 层的 OSSchedLock()函数嵌套，必须调用同样次数的 OSSchedUnlock()函数才能恢复任务调度。

参数

无

返回值

无

注意/警告

任务调用了 OSSchedLock()函数后，决不能再调用可能导致当前任务挂起的系统函数：OSTimeDly(), OSTimeDlyHMSM(), OSSemPend(), OSMboxPend(), OSQPend()。因为任务调度已经被禁止，其他任务不能运行，这会导致系统死锁。

范例

OSSchedLock();

/* 不允许被中断的执行代码 */
/* 恢复任务调度 */**OSSchedUnlock()****void OSSchedUnlock(void);**

所属文件	调用者	开关量
	任务或中断	N/A

OSSchedUnlock()函数恢复任务调度。

```
void TaskA(void *pdata)
{
    pdata = pdata;
    for (;;) {
        OSSchedLock(); /* 禁止任务调度 */
        /* 不允许被中断的执行代码 */
        OSSchedUnlock(); /* 恢复任务调度 */
    }
}
```

无

注意/警告

任务调用了 OSSchedLock()函数后，决不能再调用可能导致当前任务挂起的系统函数：OSTimeDly(), OSTimeDlyHMSM(), OSSemPend(), OSMboxPend(), OSQPend()。因为任务调度已经被禁止，其他任务不能运行，这会导致系统死锁。

范例

OSSemAccept()

```
INT16U *OSSemAccept (OS_EVENT *pevent) ;
```

所属文件	调用者	开关量
OS_SEM.C	任务或中断	OS_SEM_EN

OSSemAccept()函数查看设备是否就绪或事件是否发生。与 OSSemPend()函数不同，如果设备没有就绪，OSSemAccept()函数并不挂起任务。中断调用该函数来查询信号量。



量。当建立信号量时，该指针返回到用户程序[参

·信号量的值大于零，说明设备就绪，这个值被返
用 OSSemAccept()函数时，设备信号量的值等于

零，说明设备没有就绪，返回零。

注意/警告

必须先建立信号量，然后使用。

范例

OSSemCreate()**OS_EVENT *OSSemCreate (WORD value);**

所属文件	调用者	开关量
OS_SEM.C	任务或启动代码	OS_SEM_EN

一个信号量。信号量的作用如下：
中断同步。

value 参数是建立的信号量的初始值，可以取 0 到 65535 之间的任何值。

返回值

OSSemCreate()函数返回指向分配给所建立的消息邮箱的事件控制块的指针。如果没有可用的事件控制块，OSSemCreate()函数返回空指针。

注意/警告

必须先建立信号量，然后使用。

范例

OSSemPend()

```
void OSSemPend( OS_EVNNT *pevent, INT16U timeout, int8u *err );
```

所属文件	调用者	开关量
OS_SEM.C	任务	OS_SEM_EN

OSSemPend()函数用于任务试图取得设备的使用权，任务需要和其他任务或中断同步，任务需要等待特定事件的发生等场合。如果任务调用 OSSemPend()函数时，信号量的值大于零，OSSemPend()函数递减该值并返回该值。如果调用时信号量等于零，OSSemPend()函数将任务加入该信号量的等待队列。OSSemPend()函数挂起当前任务直到其他的任务或中断置起信号量或超出等待的预期时间。如果在预期的时钟节拍内信号量被置起，μC/OS-II默认最高优先级的任务取得信号量恢复执行。一个被 OSTaskSuspend()函数挂起的任务也可以接受信号量，但这个任务将一直保持挂起状态直到通过调用 OSTaskResume()函数恢复任务的运行。

参数

pevent 是指向信号量的指针。该指针的值在建立该信号量时可以得到[参考 OSSemCreate()函数]。

timeout 允许一个任务在经过了指定数目的时钟节拍后还没有得到需要的信号量时恢复运行状态。如果该值为零表示任务将持续的等待信号量。最大的等待时间为 65535 个时钟节拍。这个时间长度并不是非常严格的，可能存在一个时钟节拍的误差，因为只有在一个时钟节拍结束后才会减少定义的等待超时时钟节拍。

err 是指向包含错误码的变量的指针。OSSemPend()函数返回的错误码可能为下述几种：

- **OS_NO_ERR:** 信号量不为零。
- **OS_TIMEOUT:** 信号量没有在指定的周期数内置起。
- **OS_ERR_PEND_ISR:** 从中断调用该函数。虽然规定了不允许从中断调用该函数，但μC/OS-II仍然包含了检测这种情况的功能。
- **OS_ERR_EVENT_TYPE:** pevent 不是指向信号量的指针。

返回值

无

```

OS_EVENT *DiagEvent;

void DiagTask(void *pdata)
{
    INT8U acc;
    pdata = pdata;
    for(;;);
}

OSSemPost(DiagEvent, 0, acc);
/* 只有信号量置0，该任务才能执行 */
}

```

超星浏览器提醒您：
 使用本复制品
 请尊重相关知识版权！

OSSemPost()

INT8U OSSemPost (OS_EVENT *pevent) ;

所属文件	调用者	开关量
OS_SEM.C	任务或中断	OS_SEM_EN

*OSSemPost()*函数置起指定的信号量。如果指定的信号量是零或大于零, *OSSemPost()*函数递增该信号量并返回。如果有任何任务在等待信号量, 最高优先级的任务将得到信号量并进入就绪状态。任务调度函数将进行任务调度, 决定当前运行的任务是否仍然为最高优先级的就绪状态的任务。

参数

pevent 是指向信号量的指针。该指针的值在建立该信号量时可以得到[参考 *OSSemCreate()*函数]。

返回值

*OSSemPost()*函数的返回值为下述之一:

- OS_NO_ERR：信号量成功的置起。

```
OS_EVENT *pEvent;
void TaskX(void *pdata)
{
    INT8U err;
    OSSEM_DATA *pdata = (OSSEM_DATA *)pdata;
    for(;;) {
        err = OSSemPost(pEvent);
        if (err == OS_NO_ERR) { /* 信号量置起 */
            /* 信号量置出 */
        }
    }
}
```

不是指向信号量的指针。



OSSemQuery()

INT8U OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata);

所属文件	调用者	开关量
OS_SEM.C	任务或中断	OS_SEM_EN

`OSSemQuery()` 函数用于获取某个信号量的信息。使用 `OSSemQuery()` 之前，应用程序需要先创立类型为 `OS_SEM_DATA` 的数据结构，用来保存从信号量的事件控制块中取得的数据。使用 `OSEventTbl` 可以得知其大小，以及有多少任务位于信号量的任务等待队列中（通过信号量的标识号码。`OSEventTbl[]` 域的大小由语句：
`#define constant OS_EVENT_TBL_SIZE` 定义（参阅文件 `uCOS_II.H`）。

参数

`pevent` 是一个指向信号量的指针。该指针在信号量建立后返回调用程序[参见 `OSSemCreate()` 函数]。

`pdata` 是一个指向数据结构 `OS_SEM_DATA` 的指针，该数据结构包含下述域：

```
OS_SEM_DATA *sem_data;
OS_EVENT *event;
void Task(void *pdata);
```

OS_SEM_DATA

```
OS_SEM_DATA sem_data;
OS_EVENT event;
void Task(void *pdata);
```

OS_EVENT

```
OS_EVENT *event;
```

void Task(void *pdata);

```
OS_SEM_DATA sem_data;
OS_EVENT *event;
void Task(void *pdata);
```

- `OS_NO_ERR` 表示调用成功。
- `OS_ERR_EVENT_TYPE` 表示未向信号量传递指针。

注意/警告

被操作的信号量必须是已经建立了的。

范例

在本例中，应用程序检查信号量，查找等待队列中优先级最高的任务。

```
pdata = pdata;
for (j=1; j<=1000; j++) {
    if (mem_data == OSInitQuery(DispMem, mem_data)) {
        if (mem_data <= 0x300000) {
            if (mem_data >= 0x400000) {
                if (mem_data <= 0x500000) {
                    if (mem_data >= 0x600000) {
                        if (mem_data <= 0x700000) {
                            if (mem_data >= 0x800000) {
                                if (mem_data <= 0x900000) {
                                    if (mem_data >= 0xA00000) {
                                        if (mem_data <= 0xB00000) {
                                            if (mem_data >= 0xC00000) {
                                                if (mem_data <= 0xD00000) {
                                                    if (mem_data >= 0xE00000) {
                                                        if (mem_data <= 0xF00000) {
                                                            if (mem_data >= 0x1000000) {
                                                                if (mem_data <= 0x1100000) {
                                                                    if (mem_data >= 0x1200000) {
................................................................
```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

OSStart()

void OSStart(void);

所属文件	调用者	开关量
OS_CORE.C	只能是初始化代码	无

OSStart()启动μC/OS-II 的多任务环境。

参数

无

返回值

无

注意/警告

在调用 OSStart()之前必须先调用 OSInit ()。在用户程序中 OSStart()只能被调用一次。第二次调用 OSStart()将不进行任何操作。

范例

```
void main(void)
{
    /* 用户代码 */
    OSInit();
    /* 初始化 CPOS-RTOS */
    /* 用户代码 */
    OSStart();
    /* 启动多任务环境 */
}
```



OSStatInit()

```
void OSStatInit(void);
```

调用者	开关量
是初始化代码	OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_EN

OSStatInit()仅当系统中仅有其他任务运行时，32位计数器所能达到的最大值。OSStatInit()的调用时机是当多任务环境已经启动，且系统中只有一个任务在运行。也就是说，该函数只能在第一个被建立并运行的任务中调用。

参数

无

返回值

无

注意/警告

无

范例

超星阅览器提醒您：
 使用本复制品
 请尊重相关知识产权！

OSTaskChangePrio()

INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio);

所属文件	调用者	开关量
OS_TASK.C	任务	OS_TASK_CHANGE_PRIO_EN

OSTaskChangePrio()改变一个任务的优先级。

oldprio 是任务原先的优先级。

newprio 是任务的新优先级。

返回值

OSTaskChangePrio()的返回值为下述之一：

- OS_NO_ERR：任务优先级成功改变。
- OS_PRIO_INVALID：参数中的任务原先优先级或新优先级大于或等于 OS_LOWEST_PRIO。
- OS_PRIO_EXIST：参数中的新优先级已经存在。
- OS_PRIO_ERR：参数中的任务原先优先级不存在。

注意/警告

参数中的新优先级必须是没有使用过的，否则会返回错误码。在 OSTaskChangePrio() 中存在。

范例

超星浏览器提醒您：
 使用本复制品
 请尊重相关知识产权！

OSTaskCreate()

```
INT8U OSTaskCreate(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio);
```

调用者	开关量
或初始化代码	无

任务的建立可以在多任务环境启动之前，也可以在正常运行时。一个任务必须为无限循环结构（如下所示），且不能有返回点。

*OSTaskCreate()*是为与先前的μC/OS 版本保持兼容，新增的特性在 *OSTaskCreateExt()*函数中。

无论用户程序中是否产生中断，在初始化任务堆栈时，堆栈的结构必须与 CPU 中断后寄存器入栈的顺序结构相同。详细说明请参考所用处理器的手册。

参数

task 是指向任务代码的指针。

pdata 指向一个数据结构，该结构用来在建立任务时向任务传递参数。下例中说明μC/OS 中的任务结构以及如何传递参数 **pdata**：

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

ptos 为指向任务堆栈栈顶的指针。任务堆栈用来保存局部变量，函数参数，返回地址以及任务被中断时的 CPU 寄存器内容。任务堆栈的大小决定于任务的需要及预计的中断嵌套层数。计算堆栈的大小，需要知道任务的局部变量所占的空间，可能产生嵌套调用的函数，及中断嵌套所需空间。如果初始化常量 OS_STK_GROWTH 设为 1，堆栈被设为从内存高地址向低地址增长，此时 **ptos** 应该指向任务堆栈空间的最高地址。反之，如果 OS_STK_GROWTH 设为 0，堆栈将从内存的低地址向高地址增长。

prio 为任务的优先级。每个任务必须有一个唯一的优先级作为标识。数字越小，优先级越高。

OSTaskCreate()的返回值为下述之一：

- OS_NO_ERR：函数调用成功。
- OS_PRIO_EXIST：具有该优先级的任务已经存在。
- OS_PRIO_INVALID：参数指定的优先级大于 OS_LOWEST_PRIO。
- OS_NO_MORE_TCB：系统中没有 OS_TCB 可以分配给任务了。

注意/警告

任务堆栈必须声明为 OS_STK 类型。

在任务中必须调用μC/OS 提供的下述过程之一：延时等待、任务挂起、等待事件发生（等待信号量，消息邮箱、消息队列），以使其他任务得到 CPU。

用户程序中不能使用优先级 0, 1, 2, 3, 以及 OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2, OS_LOWEST_PRIO-1, OS_LOWEST_PRIO。这些优先级 μC/OS 系统保留，其余的 56 个优先级提供给应用程序。

范例 1

本例中，传递给任务 Task1()的参数 **pdata** 不使用，所以指针 **pdata** 被设为 NULL。注意到程序中设定堆栈向低地址增长，传递的栈顶指针为高地址 &Task1Stk [1023]。如果在你的程序中设定堆栈向高地址增长，则传递的栈顶指针应该为 &Task1Stk [0]。

```

void main(void)
{
    INT8U err;

    OSInit(); /* 初始化 μC/OS-II */

    OSTaskCreate(Task1, /* 任务名 */ (void *)0,
                 &Task1Stk[1023], /* 任务堆栈 */
                 25); /* 优先级 */

    OSStart(); /* 启动多任务环境 */
}

void Task1(void *pdata)
{
    pdata = pdata;
    for (;;) {
        /* 任务代码 */
    }
}

```

范例 2

你可以创立一个通用的函数，多个任务可以共享一个通用的函数体，例如一个处理串行通信口的函数。传递不同的初始化数据（端口地址、波特率）和指定不同的通信口就可以作为不同的任务运行。

```

OS_STK *Comm1Stk[1024];
COMM_DATA Comm1Data; /* 包含 COMM 口初始化数据的数据结构 */
/* 通道 1 的数据 */

OS_STK *Comm2Stk[1024];
COMM_DATA Comm2Data; /* 包含 COMM 口初始化数据的数据结构 */

```

```

    /* 通道 2 的数据 */
    . . .
}

void main(void)
{
    INT8U err;
    . . .
    OSInit();           /* 初始化 C/OS-II */
    . . .
    OSTaskCreate(CommTask,
        (void *)&Comm1Data,
        &Comm1Stk[1023],
        25);
    OSTaskCreate(CommTask,
        (void *)&Comm2Data,
        &Comm2Stk[1023],
        26);
    . . .
    OSStart();          /* 启动多任务环境 */
}

void CommTask(void *pdata) /* 通信任务 */
{
    for (;;) {
        . . .
        /* 任务代码 */
        . . .
    }
}

```

OSTaskCreateExt()

```

INT8U OSTaskCreateExt(void (*task)(void *pd), void *pdata,
OS_STK *ptos, INT8U prio, INT16U id, OS_STK *pbos, INT32U stk_size, void *pext, INT16U opt);

```

所属文件	调用者	开关量
OS_TASK.C	任务或初始化代码	无

OSTaskCreateExt()建立一个新任务。与 OSTaskCreate()不同的是，OSTaskCreateExt()允许用户设置更多的细节内容。任务的建立可以在多任务环境启动之前，也可以在正在运行的任务中建立。由于新任务和原有任务不能共存，所以不能建立新任务。一个任务必须为无限循环结构（如下所示）。

用来在建立任务时向任务传递参数。下例说明了 pdata（译注 1）。

```
void TaskCreateExt(pdata)
{
    /* 对参数 pdata 进行操作，例如 pdata = pdata */
    /* 任务函数体，总是为无限循环结构 */
}

/* 任务中必须调用如下的函数：
 * OSTaskCreate()
 * OSTaskDel()
 * OSOpenEvent()
 * OSOpenEventEx()
 * OSTimeDly()
 * OSTimeDlyHMSM()
 * OSTaskSleep()
 * OSTaskDelay()
 */


```

ptos 为指向任务堆栈栈顶的指针。任务堆栈用来保存局部变量，函数参数，返回地址以及中断时的 CPU 寄存器内容。任务堆栈的大小决定于任务的需要及预计的中断嵌套层数。计算堆栈的大小，需要知道任务的局部变量所占的空间，可能产生嵌套调用的函数，及中断嵌套所需空间。如果初始化常量 OS_STK_GROWTH 设为 1，堆栈被设为向低端增长（从内存高地址向低地址增长）。此时 **ptos** 应该指向任务堆栈空间的最高地址。反之，

译注 1：如果在程序中不使用参数 **pdata**，为了避免在编译中出现“参数未使用”的警告信息，可以写一句 **pdata = pdata**。

如果 OS_STK_GROWTH 设为 0，堆栈将从低地址向高地址增长。

prio 为任务的优先级。每个任务必须有一个惟一的优先级作为标识。数字越小，优先级越高。

id 是任务的标识，目前这个参数没有实际的用途，但保留在 OSTaskCreateExt() 中供今后扩展，应用程序中可设置 id 与优先级相同。

pbos 为指向堆栈底端的指针。如果初始化常量 OS_STK_GROWTH 设为 1，堆栈被设为从内存高地址向低地址增长。此时 **pbos** 应该指向任务堆栈空间的最低地址。反之，如果 OS_STK_GROWTH 设为 0，堆栈将从低地址向高地址增长。**pbos** 应该指向堆栈空间的最高地址。参数 **pbos** 用于堆栈检测函数 OSTaskStkChk()。

stk_size 指定任务堆栈的大小。其单位由 OS_STK 定义：当 OS_STK 的类型定义为 INT8U、INT16U、INT32U 的时候，**stk_size** 的单位分别为字节（8 位）、字（16 位）和双字（32 位）。

pext 是一个用户定义数据结构的指针，可作为 TCB 的扩展。例如，当任务切换时，用户定义的数据结构中可存放浮点寄存器的数值，任务运行时间，任务切入次数等等信息。

opt 存放与任务相关的操作信息。**opt** 的低 8 位由 μC/OS 保留，用户不能使用。用户可以使用 **opt** 的高 8 位。每一种操作由 **opt** 中的一位或几位指定，当相应的位被置位时，表示选择某种操作。当前的 μC/OS 版本支持下列操作：

- OS_TASK_OPT_STK_CHK：决定是否进行任务堆栈检查。
- OS_TASK_OPT_STK_CLR：决定是否清空堆栈。
- OS_TASK_OPT_SAVE_FP：决定是否保存浮点寄存器的数值。此项操作仅当处理器有浮点硬件时有效。保存操作由硬件相关的代码完成。

其他操作请参考文件 uCOS_II.H。

返回值

OSTaskCreateExt() 的返回值为下述之一：

- OS_NO_ERR：函数调用成功。
- OS_PRIO_EXIST：具有该优先级的任务已经存在。
- OS_PRIO_INVALID：参数指定的优先级大于 OS_LOWEST_PRIO。
- OS_NO_MORE_TCB：系统中没有 OS_TCB 可以分配给任务了。

注意/警告

任务堆栈必须声明为 OS_STK 类型。

在任务中必须进行 μC/OS 提供的下述过程之一：延时等待、任务挂起、等待事件发生（等待信号量，消息邮箱、消息队列），以使其他任务得到 CPU。

用户程序中不能使用优先级 0, 1, 2, 3，以及 OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2, OS_LOWEST_PRIO-1, OS_LOWEST_PRIO。这些优先级 μC/OS 系统保留，其余 56 个优先级提供给应用程序。

范例 1

```

typedef struct {           /* 用户定义的数据结构 (1) */
    char TaskName[20];
    INT16U TaskID;
    INT16U TaskPriority;
    INT32D TaskCreateTime;
    INT32D TaskLastExecuteTime;
} TASK_USER_DATA;

OS_STK TaskStack1Stack;           /* 堆栈向低地址增长 (2) */
TASK_USER_DATA TaskUserDatas;    /* 使用本章知识实现! */

void main(void)
{
    OSInit();                  /* 初始化 C/OS-II */
    strcpy(TaskUserDatas.TaskName, "MyTaskName"); /* 命名任务 (2) */
    Task = OSTaskCreateExt(Task,
                           (void *)0,
                           TaskStack1,
                           /* 堆栈向低地址增长 (3) (3) */
                           10,
                           OSTaskCreateExt,
                           (void *)TaskUserDatas,
                           /* (3) 的扩展 */           /* (3) */
                           );
}

```

结构 `TASK_USER_DATA` (1), 在其中保存了任
标准库函数 `strcpy()` 初始化 (2)。在本例中, 允许
`OSTaskStkChk()` 函数。本例中设定堆栈向低地址方
`WTH` 设为 1。程序注释中的 `TOS` 意为堆栈顶端
(Bottom-Of-Stack)。

```

OS_TASK_OPT_STK_CHK);           /* 允许堆栈检查      (4) */

...
OSStart();                      /* 启动多任务环境      */
}

void Task(void *pdata)
{
    pdata = pdata;                /* 此句可避免编译中的警告信息      */
    for (;;) {
        .
        .
        .
    }
}

```

范例 2

本例中创立的任务将运行在堆栈向高地址增长的处理器上(1), 例如 Intel 的 MCS-251。此时 OS_STK_GROWTH 设为 0。在本例中, 允许堆栈检查操作(2), 程序可以调用 OSTaskStkChk()函数。程序注释中的 TOS 意为堆栈顶端 (Top-Of-Stack), BOS 意为堆栈底顶端 (Bottom-Of-Stack)。

```

OS_STK *TaskStk[1024];

void main(void)
{
    INT8U err;
    .
    .
    .
    OSInit();                  /* 初始化 C/OS-II      */
    .
    .
    .
    err = OSTaskCreateExt(Task,
                          (void *)0,
                          &TaskStk[0],            /* 堆栈向高地址增长(TOS) (1) */
                          10,
                          10,
                          10,
                          10);
}

```

```

�任务数(1000),      /* 延长向高地址增长 (800) (11)*
1024,
void *10;
OS_TASK_DEFN(OSK_CRS);
...
osTaskInit();          /* 启动多任务环境 */
}

void Task(void *pdata)
{
    pdata = pdata;           /* 此句可避免编译时出现警告信息 */
    for(;;);
    /* 在此处 */
}

```



OSTaskDel()

INT8U OSTaskDel (INT8U prio);

所属文件	调用者	开关量
OS_TASK.C	只能是任务	OS_TASK_DEL_EN

*OSTaskDel()*函数删除一个指定优先级的任务。任务可以传递自己的优先级给 *OSTaskDel()*，从而删除自身。如果任务不知道自己的优先级，还可以传递参数 *OS_PRIO_SELF*。被删除的任务将回到休眠状态。任务被删除后可以用函数 *OSTaskCreate()*或 *OSTaskCreateExt()*重新建立。

参数

prio 为指定要删除任务的优先级，也可以用参数 *OS_PRIO_SELF* 代替，此时，下一个优先级最高的就绪任务将开始运行。

返回值

*OSTaskDel()*的返回值为下述之一：

- *OS_NO_ERR*: 函数调用成功。
- *OS_TASK_DEL_IDLE*: 错误操作，试图删除空闲任务。
- *OS_TASK_DEL_ERR*: 错误操作，指定要删除的任务不存在。
- *OS_PRIO_INVALID*: 参数指定的优先级大于 *OS_LOWEST_PRIO*。
- *OS_TASK_DEL_ISR*: 错误操作，试图在中断处理程序中删除任务。



μC/OS 中的空闲任务。

小心，此时，为安全起见可以用另一个函数



OSTaskDelReq()

INT8U OSTaskDel (INT8U prio);

所属文件	调用者	开关量
OS_TASK.C	只能是任务	OS_TASK_DEL_EN

OSTaskDelReq() 函数请求一个任务删除自身。通常 OSTaskDelReq() 用于删除一个占有系统资源的任务（例如任务建立了信号量）。对于此类任务，在删除任务之前应当先释放任务占用的系统资源。具体的做法是：在需要被删除的任务中调用 OSTaskDelReq() 检测是否有其他任务的删除请求，如果有，则释放自身占用的资源，然后调用 OSTaskDel() 删除自身。例如，假设任务 5 要删除任务 10，而任务 10 占有系统资源，此时任务 5 不能直接调用 OSTaskDel(10) 删除任务 10，而应该调用 OSTaskDelReq(10) 向任务 10 发送删除请求。在任务 10 中调用 OSTaskDelReq(OS_PRIO_SELF)，并检测返回值。如果返回 OS_TASK_DEL_

REQ，则表明有来自其他任务的删除请求，此时任务 10 应该先释放资源，然后调用 OSTaskDel(OS_PRIO_SELF)删除自己。任务 5 可以循环调用 OSTaskDelReq(10)并检测返回值，如果返回 OS_TASK_NOT_EXIST，表明任务 10 已经成功删除。

参数

prio 为要求删除任务的优先级。如果参数为 OS_PRIO_SELF，则表示调用函数的任务正在查询是否有来自其他任务的删除请求。

返回值

```
void TaskDelete(void *pTask) /* 任务优先级 5 */
{
    int32 i;
    for (i = 0; i < 10; i++) {
        if (OSTaskDelReq(i) == OS_PRIO_SELF) {
            if (i == 10) {
                while (OSTaskDelReq(10) != OS_TASK_NOT_EXIST)
                    OSTimeDly(1);
            }
        }
    }
}
```

任务记录。

的任务不存在。发送删除请求的任务可以等待此返回

作，试图删除空闲任务。

的优先级大于 OS_LOWEST_PRIO 或没有设定

- **OS_TASK_DEL_REQ:** 当前任务收到来自其他任务的删除请求。

注意/警告

OSTaskDelReq()将判断用户是否试图删除μC/OS 中的空闲任务。

范例

```

void tasktopanelled(void *pdata) /* 任务优先级 10 */
{
    OS_TCB *pdata;
    for (i=0; i<OS_TaskCount(); i++)
        if (OSTaskGetPriority(OS_PRIO_SELF) == OS_TaskGetPriority(i))
            /* 将此任务占用的系统资源 */
            /* 释放给分配的内容 */
            OSTaskDelete(OS_PRIO_SELF);
}

```



OSTaskQuery()

INT8U OSTaskQuery (INT8U prio, OS_TCB *pdata);

所属文件	调用者	开关量
OS_TASK.C	任务或中断	无

OSTaskQuery()用于获取任务信息，函数返回任务 TCB 的一个完整的拷贝。应用程序必须建立一个 OS_TCB 类型的数据结构容纳返回的数据。需要提醒用户的是，在对任务 OS_TCB 对象中的数据操作时要小心，尤其是数据项 OSTCBNext 和 OSTCBPrev。它们分别指向 TCB 链表中的后一项和前一项。

参数

prio 为指定要获取 TCB 内容的任务优先级，也可以指定参数 OS_PRIO_SELF，获取调用任务的信息。

pdata 指向一个 OS_TCB 类型的数据结构，容纳返回的任务 TCB 的一个拷贝。

返回值

OSTaskQuery()的返回值为下述之一：

- OS_NO_ERR：函数调用成功。
- OS_PRIO_ERR：参数指定的任务非法。
- OS_PRIO_INVALID：参数指定的优先级大于 OS_LOWEST_PRIO。

据成员取决于下述开关量在初始化时的设定（参见

```
void Task(void *pdata)
{
    OS_TCB task_data;
    OS_PRIO prio;
    void *pentry;
    OS_PRIO priority;

    pdata = pentry; /* 根据成员取决于下述开关量在初始化时的设定 (参见 */

    /* ... */

    prio = OSTaskQuery(OS_PRIO_SELF, &task_data);
    if (prio == OS_PRIO_ERR) {
        pentry = task_data.OSPriority; /* 读取 OS 的就绪就绪结构的优先级 */
        priority = task_data.OSPriority; /* 读取任务优先级 */
    }
}
```



OSTaskResume()**INT8U OSTaskResume (INT8U prio);**

所属文件	调用者	开关量
OS_TASK.C	只能是任务	OS_TASK_SUSPEND_EN

OSTaskResume()唤醒一个用 OSTaskSuspend()函数挂起的任务。OSTaskResume()也是惟一能“解挂”挂起任务的函数。

参数

```
void TaskResume(void *pdata)
{
    OS_PRIO prio;
    TaskID task;

    if ((pdata == NULL) || (task = OSTaskGetTaskID(pdata)) == OS_INVALID_TASK_ID)
        return;

    if (OSTaskSuspend(task, &prio) != OS_SUCCESS)
        return;

    if (prio < OS_PRIO_INVALID)
        OSTaskResume(task, prio);
    else
        OSTaskResume(task, OS_PRIO_INVALID);
}
```



一：

■ 唤醒的任务不存在。

■ 要唤醒的任务不在挂起状态。

- OS_PRIO_INVALID: 参数指定的优先级大于或等于 OS_LOWEST_PRIO。

注意/警告

无

范例

OSTaskStkChk()

```
INT8U OSTaskStkChk ( INT8U prio, OS_STK_DATA *pdata);
```

所属文件	调用者	开关量
OS_TASK.C	只能是任务	OS_TASK_CREATE_EXT

OSTaskStkChk() 检查任务堆栈状态，计算指定任务堆栈中的未用空间和已用空间。使用 `OSTaskStkChk()` 函数要求所检查的任务是被 `OSTaskCreateExt()` 函数建立的，且 `opt` 参数

包含 `OS_TASK_OPT_STK_CHK`。

计算堆栈未用空间的方法是从堆栈底端向顶端逐个字节比较，检查堆栈中 0 的个数，直到一个非 0 的数值出现。这种方法的前提是堆栈建立时已经全部清零。要实现清零操作，需要在任务建立初始化堆栈时设置 `OS_TASK_OPT_STK_CLR` 为 1。如果应用程序在初始化时已经将全部 RAM 清零，且不进行任务删除操作，也可以设置 `OS_TASK_OPT_STK_CLR` 为 0，这将加快 `OSTaskCreateExt()` 函数的执行速度。

参数

`prio` 为指定要获取堆栈信息的任务优先级，也可以指定参数 `OS_PRIO_SELF`，获取调用任务本身的信息。

`pdata` 指向一个类型为 `OS_STK_DATA` 的数据结构，其中包含如下信息：

返回值

`OSTaskStkChk()` 的返回值为下述之一：

- `OS_NO_ERR`: 函数调用成功。
- `OS_PRIO_INVALID`: 参数指定的优先级大于 `OS_LOWEST_PRIO`，或未指定 `OS_PRIO_SELF`。
- `OS_TASK_NOT_EXIST`: 指定的任务不存在。
- `OS_TASK_OPT_ERR`: 任务用 `OSTaskCreateExt()` 函数建立的时候没有指定 `OS_TASK_OPT_STK_CHK` 操作，或者任务是用 `OSTaskCreate()` 函数建立的。

注意/警告

函数的执行时间是由任务堆栈的大小决定的，事先不可预料。

在应用程序中可以把 `OS_STK_DATA` 结构中的数据项 `OSFree` 和 `OSUsed` 相加，可得到堆栈的大小。

虽然原则上该函数可以在中断程序中调用，但由于该函数可能执行很长时间，所以实际中不提倡这种做法。

```
void Task (void *pdata)
{
    OS_PKT_DATA *pkt_data;
    INT32U      pkt_size;

    for (i=0; i<5; i++)
    {
        /* ... */

        pkt = OSReadPkt(pData, pkt_data);
        if (pkt == OS_WK_BLOCK)
        {
            pkt_size = pkt_data.OSFree + pkt_data.OSNeed;
        }
        else
        {
            /* ... */
        }
    }
}
```



OSTaskSuspend()

INT8U OSTaskSuspend (INT8U prio);

所属文件	调用者	开关量
OS_TASK.C	只能是任务	OS_TASK_SUSPEND_EN

OSTaskSuspend()无条件挂起一个任务。调用此函数的任务也可以传递参数 OS_PRIO_SELF，挂起调用任务本身。当前任务挂起后，只有其他任务才能唤醒。任务挂起后，系统会重新进行任务调度，运行下一个优先级最高的就绪任务。唤醒挂起任务需要调用函数 OSTaskResume()。

任务的挂起是可以叠加到其他操作上的。例如，任务被挂起时正在进行延时操作，那么任务的唤醒就需要两个条件：延时的结束以及其他任务的唤醒操作。又如，任务被挂起时正在等待信号量，当任务从信号量的等待对列中清除后也不能立即运行，而必须等到唤醒操作后。

参数

prio 为指定要获取挂起的任务优先级，也可以指定参数 OS_PRIO_SELF，挂起任务本身。此时，下一个优先级最高的就绪任务将运行。

返回值

OSTaskSuspend()的返回值为下述之一：

- **OS_NO_ERR:** 函数调用成功。
- **OS_TASK_SUSPEND_IDLE:** 试图挂起μC/OS-II 中的空闲任务 (Idle task)。此为非法操作。

内优先级大于 **OS_LOWEST_PRIO** 或没有设定

挂起的任务不存在。

esume()应该成对使用。

OSTaskResume()唤醒。

范例

```
void Task1(void *data)
{
    int16 acc;
    /* ... */

    acc = OSTaskSuspend(OS_NO_ERR, IDLE); /* 挂起当前任务 */
    /* 通过任务名或句柄挂起任务时，任务可继续运行 */
    /* ... */
}
```



OSTimeDly()

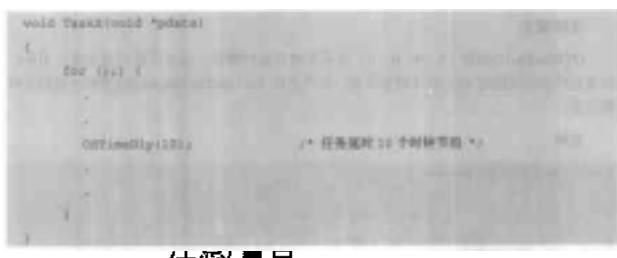
void OSTimeDly (INT16U ticks);

所属文件	调用者	开关量
OS_TIMC.C	只能是任务	无

OSTimeDly()将一个任务延时若干个时钟节拍。如果延时时间大于 0，系统将立即进行

任务调度。延时时间的长度可从 0 到 65535 个时钟节拍。延时时间 0 表示不进行延时，函数将立即返回调用者。延时的具体时间依赖于系统每秒钟有多少时钟节拍（由文件 SO_CFG.H 中的常量 OS_TICKS_PER_SEC 设定）。

参数



注意到延时时间 0 表示不进行延时操作，而立即返回调用者。为了确保设定的延时时间，建议用户设定的时钟节拍数加 1。例如，希望延时 10 个时钟节拍，可设定参数为 11。

范例

```
OSTimeDlyHMSM()
void OSTimeDlyHMSM( INT8U hours, INT8U minutes, INT8U seconds, INT8U milli);
```

所属文件	调用者	开关量
OS_TIMC.C	只能是任务	无

OSTimeDlyHMSM()将一个任务延时若干时间。延时的单位是小时、分、秒、毫秒。所以使用 OSTimeDlyHMSM()比 OSTimeDly()更方便。调用 OSTimeDlyHMSM()后，如果延时时间不为 0，系统将立即进行任务调度。

参数

hours 为延时小时数，范围从 0~255。

minutes 为延时分钟数，范围从 0~59。

seconds 为延时秒数，范围从 0~59。

milli 为延时毫秒数，范围从 0~999。需要说明的是，延时操作函数都是以时钟节拍为单位的。实际的延时时间是时钟节拍的整数倍。例如系统每次时钟节拍间隔是 10ms，如果设定延时为 5ms，将不产生任何延时操作，而设定延时 15ms，实际的延时是两个时钟节拍，也就是 20ms。

返回值

```
void TaskDly(void *pdata)
{
    OSTimeDlyHMSM(0, 0, 0); /* 任务睡眠 1 秒 */
}
```

之一：

参数错误，分钟数大于 59。

参数错误，秒数大于 59。

参数错误，毫秒数大于 999。

- OS_TIME_ZERO_DLY：四个参数全为 0。

注意/警告

OSTimeDlyHMSM(0, 0, 0, 0) 表示不进行延时操作，而立即返回调用者。另外，如果延时总时间超过 65535 个时钟节拍，将不能用 OSTimeDlyResume() 函数终止延时并唤醒任务。

范例

OSTimeDlyResume()

void OSTimeDlyResume(INT8U prio);

所属文件	调用者	开关量
OS_TIMC.C	只能是任务	无

OSTimeDlyResume() 唤醒一个用 **OSTimeDly()** 或 **OSTimeDlyHMSM()** 函数延时的任务。

参数

prio 为指定要唤醒任务的优先级。

返回值

OSTimeDlyResume() 的返回值为下述之一：

```
void TaskHandle *pdata;
{
    DATA_TYPE err;
    pdata = pHandle;
    if (err)
    {
        err = OSTimeDlyResume(10); // 唤醒优先级为 10 的任务
        if (err == OS_PRIO_BLOCKED) // 任务被唤醒
    }
}
```

的优先级大于 **OS_LOWEST_PRIO**。

的任务不在延时状态。

的任务不存在。

去唤醒一个设置了等待超时操作，并且正在等待事
结束等待，除非的确希望这么做。

OSTimeDlyResume() 函数不能唤醒一个用 **OSTimeDlyHMSM()** 延时，且延时时间总计超过 65535 个时钟节拍的任务。例如，如果系统时钟为 100Hz，**OSTimeDlyResume()** 不能唤醒延时 **OSTimeDlyHMSM(0, 10, 55, 350)** 或更长时间的任务（译注 2）。

范例

译注 2：**OSTimeDlyHMSM(0, 10, 55, 350)** 共延时 [10 minutes *60 + (55+0.35) seconds] *100 =65535 次时钟节拍。

OSTimeGet()

INT32U OSTimeGet (void);



所属文件	调用者	开关量
OS_TIMC.C	任务或中断	无

OSTimeGet()获取当前系统时钟数值。系统时钟是一个 32 位的计数器，记录系统上电

```
void Task0(void *pdata)
{
    OSInit();
    for(;;)
    {
        ticks = OSTimeGet(); // 获取当前系统时钟的值
    }
}
```

注意/警告

无

范例

OSTimeSet()

void OSTimeSet (INT32U ticks);

所属文件	调用者	开关量
OS_TIMC.C	任务或中断	无

OSTimeSet()设置当前系统时钟数值。系统时钟是一个32位的计数器，记录系统上电后或时钟重新设置后的时钟计数。

参数

```
void TaskSwitch (void)
{
    for (i=1; i<=1000000; i++)
        OSTimeSet (DLY); /* 调整系统时钟 */
}
```

节拍数。



无

范例

OSTimeTick()

void OSTimeTick (void);

所属文件	调用者	开关量
OS_TIMC.C	任务或中断	无

每次时钟节拍，μC/OS-II 都将执行 **OSTimeTick()** 函数。**OSTimeTick()** 检查处于延时状态的任务是否达到延时时间 [用 **OSTimeDly()** 或 **OSTimeDlyHMSM()** 函数延时]，或正在等待事件的任务是否超时。

参数

无

返回值

无

注意/警告

```
TICKISRPROC FAR
PUSHA                                ; 保存 CPU 寄存器内容
PUSH BX
PUSH DX

INC WORD PTR _OSTimeLasting ; 调用 μC/OS-II 进入中断处理程序
CALL FAR PTR _OSTimeTick      ; 调用时钟节拍处理函数
                                ; 用户代码被调用中断标志

CALL FAR PTR _OSBAUDINIT ; 调用 μC/OS-II 退出中断处理程序
POP DX
POP BX
POPA

IRET                                ; 中断返回

TICKISRENDPROC
```

任务数直接相关，在任务或中断中都可以调用。
高（优先级数字很小），这是因为 OSTimeTick() 负



OSVersion()

INT16U OSVersion (void);

所属文件	调用者	开关量
OS_CORE.C	任务或中断	无

OSVersion() 获取当前 μC/OS-II 的版本。

参数

无

返回值

当前版本，格式为 x.yy，返回值为乘以 100 后的数值。例如当前版本 2.00，则返回 200。

```
void TaskX(void *pdata)
{
    int160 ms_version;
    for(;;) {
        ms_version = (char*)ms_version; /* 把 ms_version 的版本 */
    }
}
```



OS_ENTER_CRITICAL()
OS_EXIT_CRITICAL()

所属文件	调用者	开关量
OS_CPU.C	任务或中断	无

*OS_ENTER_CRITICAL()*和*OS_EXIT_CRITICAL()*为定义的宏，用来关闭、打开 CPU 的中断。

参数

无

返回值

无

注意/警告

*OS_ENTER_CRITICAL()*和*OS_EXIT_CRITICAL()*必须成对使用。

范例

```
void TaskX(void *pdata)
```

```
{  
    for(;;) {  
        /*  
         * 在此期间，不能有中断发生  
         */  
        OS_ENTER_CRITICAL(); /* 关闭中断 */  
  
        /* 进入核心代码 */  
  
        OS_EXIT_CRITICAL(); /* 打开中断 */  
  
        /*  
         * 在此期间，不能有中断发生  
         */  
    }  
}
```

从上面的代码中可以看出，如果在进入内核代码之前没有调用 OS_ENTER_CRITICAL()，那么在进入内核代码之后就不能再调用 OS_EXIT_CRITICAL()。也就是说，在调用 OS_EXIT_CRITICAL() 之前必须先调用 OS_ENTER_CRITICAL()。如果在调用 OS_EXIT_CRITICAL() 之前没有调用 OS_ENTER_CRITICAL()，那么在调用 OS_EXIT_CRITICAL() 之后就不能再调用 OS_ENTER_CRITICAL()。也就是说，在调用 OS_EXIT_CRITICAL() 之前必须先调用 OS_ENTER_CRITICAL()。

第 12 章

超量浏览器提醒您：
 使用本复制品
 请尊重相关知识产权！

配置手册

类型	宏	其他常量
常量		
OSInit()	—	OS_MAX_EVENTS OS_Q_EN and OS_MAX_QS OS_MMEN OS_TASK_IDLE_STK_SIZE OS_TASK_STAT_EN OS_TASK_STAT_STK_SIZE
OSSchdLock()	—	—
OSSchdUnlock()	—	—
OSStart()	—	—
OSStartExt()	OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_EN	OS_TICKS_PER_SEC
OSVersion()	—	—
中断处理		
OSIntEnter()	—	—
OSIntExit()	—	—
消息队列		
OSMboxAccept()	OS_MBOX_EN	—
OSMboxCreate()	OS_MBOX_EN	OS_MAX_EVENTS

置项。由于μC/OS-II 向用户提供源代码，初始化配置项都在文件 OS_CFG.H 中。用户的工程文件组中都

义的常量，介绍的顺序和它们在 OS_CFG.H 中出现时的μC/OS-II 函数。“类型”为函数所属的类型，“置项”为与这个函数有关的其他控

注意编译工程文件时要包含 OS_CFG.H，使定义的常量生效。

表 12.1 μC/OS-II 函数和相关常量

续表

类 型	置1	其他常量
OSMboxPend()	OS_MBOX_EN	—
OSMboxPost()	OS_MBOX_EN	—
OSMboxQuery()	OS_MBOX_EN	—
内存块管理		
OSMemCreate()	OS_MEM_EN	OS_MAX_MEM_PART
OSMemGet()	OS_MEM_EN	—
OSMemPut()	OS_MEM_EN	—
OSMemQuery()	OS_MEM_EN	—
消息队列		
OSQAccept()	OS_Q_EN	—
OSQCreate()	OS_Q_EN	OS_MAX_EVENTSOS_MAX_QS
OSQFlush()	OS_Q_EN	—
OSQPend()	OS_Q_EN	—
OSQPost()	OS_Q_EN	—
OSQPostFront()	OS_Q_EN	—
OSQuery()	OS_Q_EN	—
信号量管理		
OSSemAccept()	OS_SEM_EN	—
OSSemCreate()	OS_SEM_EN	OS_MAX_EVENTS
OSSemPend()	OS_SEM_EN	—
OSSemPost()	OS_SEM_EN	—
OSSemQuery()	OS_SEM_EN	—
任务管理		
OSTaskChangePrio()	OS_TASK_CHANGE_PRIO_EN	OS_LOWEST_PRIO
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
时钟管理		
OSTimeDly()	—	—

未经许可
使用本复制品
请尊重相关知识产权!

形 型	定 义	其 他 介 绍
OSTimeDlyHMSM()	—	OS_TICKS_PER_SEC
OSTimeDlyResource()	—	OS_LOWEST_PRIO
OSTimeGet()	—	—
OSTimeSet()	—	—
OSTimeTick()	—	—
用 P 定义函数		
OSTaskCreateHook()	OS_CPU_HOOKS_EN	—
OSTaskDeleteHook()	OS_CPU_HOOKS_EN	—
OSTaskSethook()	OS_CPU_HOOKS_EN	—
OSTaskSwHook()	OS_CPU_HOOKS_EN	—
OSTimeTickHook()	OS_CPU_HOOKS_EN	—

OS_MAX_EVENTS

OS_MAX_EVENTS 定义系统中最大的事件控制块的数量。系统中的每一个消息邮箱，消息队列，信号量都需要一个事件控制块。例如，系统中有 10 个消息邮箱，5 个消息队列，3 个信号量，则 **OS_MAX_EVENTS** 最小应该为 18。只要程序中用到了消息邮箱，消息队列或是信号量，则 **OS_MAX_EVENTS** 最小应该设置为 2。

OS_MAX_MEM_PARTS

OS_MAX_MEM_PARTS 定义系统中最大的内存块数，内存块将由内存管理函数操作（定义在文件 **OS_MEM.C** 中）。如果要使用内存块，**OS_MAX_MEM_PARTS** 最小应该设置为 2，常量 **OS_MEM_EN** 也要同时置 1。

OS_MAX_QS

OS_MAX_QS 定义系统中最大的消息队列数。要使用消息队列，常量 **OS_Q_EN** 也要同时置 1。如果要使用消息队列，**OS_MAX_QS** 最小应该设置为 2。

OS_MAX_TASKS

OS_MAX_MEM_TASKS 定义用户程序中最大的任务数。**OS_MAX_MEM_TASKS** 不能大于 62，这是由于μC/OS-II 保留了两个系统使用的任务。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。而 OS_MAX_MEM_TASKS 设定的太大则会浪费内存。

OS_LOWEST_PRIO

OS_LOWEST_PRIO 设定系统中的任务最低优先级（最大优先级数）。设定 OS_LOWEST_PRIO 可以节省用于任务控制块的内存。 μC/OS-II 中优先级数从 0（最高优先级）到 63（最低优先级）。设定 OS_LOWEST_PRIO 小于 63 意味着不会建立优先级数大于 OS_LOWEST_PRIO 的任务。<μC/OS-II 中保留两个优先级系统自用：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()]。用户任务的优先级可以从 0 到 OS_LOWEST_PRIO-2。OS_LOWEST_PRIO 和 OS_MAX_TASKS 之间没有什么关系。例如，可以设 OS_MAX_TASKS 为 10 而 OS_LOWEST_PRIO 为 32。此时系统最多可有 10 个任务，用户任务的优先级可以是 0 到 30。当然，OS_LOWEST_PRIO 设定的优先级也要够用，例如设 OS_MAX_TASKS 为 20，而 OS_LOWEST_PRIO 为 10，优先级就不够用了。

OS_TASK_IDLE_STK_SIZE

OS_TASK_IDLE_STK_SIZE 设置 μC/OS-II 中空闲任务堆栈的容量。注意堆栈容量的单位不是字节，而是 OS_STK（译注 1）。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。

OS_TASK_STAT_EN

OS_TASK_STAT_EN 设定系统是否使用 μC/OS-II 中的统计任务（statistic task）及其初始化函数。如果设为 1，则使用统计任务 OSTaskStat()。统计任务每秒运行一次，计算当前系统 CPU 使用率，结果保存在 8 位变量 OSCPUUsage 中。每次运行，OSTaskStat()都将调用 OSTaskStatHook() 函数，用户自定义的统计功能可以放在这个函数中。详细情况请参考 OS_CORE.C 文件。统计任务 OSTaskStat() 的优先级总是设为 OS_LOWEST_PRIO-1。

当 OS_TASK_STAT_EN 设为 0 的时候，全局变量 OSCPUUsage, OSIdleCtrMax, OSIdleCtrRun 和 OSStatRdy 都不声明，以节省内存空间。

译注 1：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。

OS_TASK_STAT_STK_SIZE

OS_TASK_STAT_STK_SIZE 设置μC/OS-II 中统计任务堆栈的容量。注意单位不是字节，而是 OS_STK (译注 2)。统计任务堆栈的容量取决于所使用的处理器类型，以及如下的操作：

- 进行 32 位算术运算所需的堆栈空间。
- 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，用户需要设 OS_TASK_CREATE_EXT_EN 为 1，并使用 OSTaskCreateExt() 函数建立任务。

OS_CPU_HOOKS_EN

此常量设定是否在文件 OS_CPU_C.C 中声明对外接口函数 (hook function)，设为 1 为声明。μC/OS-II 中提供了 5 个对外接口函数，可以在文件 OS_CPU_C.C 中声明，也可以在用户自己的代码中声明：

- OSTaskCreateHook()
- OSTaskDelHook()
- OSTaskStatHook()
- OSTaskSwHook()
- OSTimeTickHook()

OS_MBOX_EN

OS_MBOX_EN 控制是否使用μC/OS-II 中的消息邮箱函数及其相关数据结构，设为 1 为使用。如果不使用，则关闭此常量节省内存。

OS_MEM_EN

OS_MEM_EN 控制是否使用μC/OS-II 中的内存块管理函数及其相关数据结构，设为 1 为使用。如果不使用，则关闭此常量节省内存。

译注 2：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。

OS_Q_EN



OS_Q_EN 控制是否使用μC/OS-II 中的消息队列函数及其相关数据结构，设为 1 为使用。如果不使用，则关闭此常量节省内存。如果 OS_Q_EN 设为 0，则语句#define constant OS_MAX_QS 无效。

OS_SEM_EN

OS_SEM_EN 控制是否使用μC/OS-II 中的信号量管理函数及其相关数据结构，设为 1 为使用。如果不使用，则关闭此常量节省内存。

OS_TASK_CHANGE_PRIO_EN

此常量控制是否使用μC/OS-II 中的 OSTaskChangePrio() 函数，设为 1 为使用。如果在应用程序中不需要改变运行任务的优先级，则将此常量设为 0 节省内存。

OS_TASK_CREATE_EN

此常量控制是否使用μC/OS-II 中的 OSTaskCreate() 函数，设为 1 为使用。在μC/OS-II 中推荐用户使用 OSTaskCreateExt() 函数建立任务。如果不使用 OSTaskCreate() 函数，将 OS_TASK_CREATE_EN 设为 0 可以节省内存。注意 OS_TASK_CREATE_EN 和 OS_TASK_CREATE_EXT_EN 至少有一个要为 1，当然如果都使用也可以。

OS_TASK_CREATE_EXT_EN

此常量控制是否使用μC/OS-II 中的 OSTaskCreateExt() 函数，设为 1 为使用。该函数为扩展的，功能更全的任务建立函数。如果不使用该函数，将 OS_TASK_CREATE_EXT_EN 设为 0 可以节省内存。注意，如果要使用堆栈检查函数 OSTaskStkChk()，则必须用 OSTaskCreateExt() 建立任务。

OS_TASK_DEL_EN

此常量控制是否使用μC/OS-II 中的 OSTaskDel() 函数，设为 1 为使用。如果在应用程序中不使用删除任务函数，将 OS_TASK_DEL_EN 设为 0 可以节省内存。

OS_TASK_SUSPEND_EN

此常量控制是否使用μC/OS-II 中的 OSTaskSuspend()和 OSTaskResume()函数，设为 1 为使用。如果在应用程序中不使用任务挂起-唤醒函数，将 OS_TASK_SUSPEND_EN 设为 0 可以节省内存。

OS_TICKS_PER_SEC

此常量标识调用 OSTimeTick()函数的频率。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（译注 3）。在函数 OSStatInit(), OSTaskStat()和 OSTimeDlyHMSM()中都会用到 OS_TICKS_PER_SEC。

译注 3：即系统硬件定时器中断发生的频率。

附录 A

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

源代码范例

A.0 例 1

A.0.1 EX1L.C

```
/*
***** uC/OS-II
* The Real-Time Kernel
*
* (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
* All Rights Reserved
*
* V2.00
*
* EXAMPLE #1
*****
*/
#include "includes.h"

/*
***** CONSTANTS
*****
*/
#define TASK_STK_SIZE      512 /* Size of each task's stacks (# of WORDs) */
#define N_TASKS             10  /* Number of identical tasks */

/*
*****
```

附录 A 源代码范例

```
*                                VARIABLES
*****
*/
OS_STK    TaskStk[N_TASKS][TASK_STK_SIZE]; /* Tasks stacks
OS_STK    TaskStartStk[TASK_STK_SIZE];
char      TaskData[N_TASKS];                  /* Parameters to pass to each task */
OS_EVENT   *RandomSem;

/*
*****
*          FUNCTION PROTOTYPES
*****
*/
void  Task(void *data);           /* Function prototypes of tasks           */
void  TaskStart(void *data);     /* Function prototypes of Startup task   */

/*
*****
*          MAIN
*****
*/
void main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK); /* Clear the screen      */
    OSInit();                                         /* Initialize uC/OS-II   */
    PC_DOSSaveReturn();                               /* Save environment to return to DOS */
    PC_VectSet(uCOS, OSCTxSw); /* Install uC/OS-II's context switch vector */
    RandomSem = OSSemCreate(1);                      /* Random number semaphore */
    OSTaskCreate(TaskStart, (void *)0, (void *)&TaskStartStk[TASK_STK_SIZE - 1], 0);
    OSStart();                                         /* Start multitasking      */
}

/*
*****
*          STARTUP TASK
*****
*/
void TaskStart (void *data)
{
    UBYTE i;
    char  s[100];
```



```

WORD key;

data = data; /* Prevent compiler warning */

PC_DispStr(26, 0, "uC/OS-II, The Real-Time Kernel", DISP_FGND_WHITE+DISP_BGND_RED+DISP_BLINK);
PC_DispStr(33, 1, "Jean J. Labrosse", DISP_FGND_WHITE);
PC_DispStr(36, 3, "EXAMPLE #1", DISP_FGND_WHITE);

OS_ENTER_CRITICAL();
PC_VectSet(0x08, OSTickISR); /* Install uC/OS-II's clock tick ISR */
PC_SetTickRate(OS_TICKS_PER_SEC); /* Reprogram tick rate */
OS_EXIT_CRITICAL();

PC_DispStr(0, 22, "Determining CPU's capacity ...", DISP_FGND_WHITE);
OSStatInit(); /* Initialize uC/OS-II's statistics */
PC_DispClrLine(22, DISP_FGND_WHITE + DISP_BGND_BLACK);

for (i = 0; i < N_TASKS; i++) { /* Create N_TASKS identical tasks */
    TaskData[i] = '0' + i; /* Each task will display its own letter */
    OSTaskCreate(Task, (void *)&TaskData[i], (void *)&TaskStk[i][TASK_STK_SIZE - 1], i + 1);
}
PC_DispStr(0, 22, "#Tasks : xxxxx CPU Usage: xxx %", DISP_FGND_WHITE);
PC_DispStr(0, 23, "#Task switch/sec: xxxxx", DISP_FGND_WHITE);
PC_DispStr(28, 24, "<-PRESS 'ESC' TO QUIT->", DISP_FGND_WHITE + DISP_BLINK);
for (;;) {
    sprintf(s, "%5d", OSTaskCtr); /* Display #tasks running */
    PC_DispStr(18, 22, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
    sprintf(s, "%3d", OSCPUUsage); /* Display CPU usage in % */
    PC_DispStr(36, 22, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
    sprintf(s, "%5d", OSCtxSwCtr); /* Display #context switches per second */
    PC_DispStr(18, 23, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
    OSCtxSwCtr = 0;

    sprintf(s, "V%3.2f", (float)OSVersion() * 0.01); /* Display version number as Vx.yy */
    PC_DispStr(75, 24, s, DISP_FGND_YELLOW + DISP_BGND_BLUE);
    PC_GetDateTime(s); /* Get and display date and time */
    PC_DispStr(0, 24, s, DISP_FGND_BLUE + DISP_BGND_CYAN);

    if (PC.GetKey(&key) == TRUE) { /* See if key has been pressed */
        if (key == 0x1B) { /* Yes, see if it's the ESCAPE key */
            PC_DOSReturn(); /* Return to DOS */
        }
    }
}

```



附录 A 源代码范例

```
OSTimeDlyHMSM(0, 0, 1, 0);           /* Wait one second */  
}  
}  
  
/*  
*****  
*          TASKS  
*****  
*/  
  
void Task (void *data)  
{  
    UBYTE x;  
    UBYTE y;  
    UBYTE err;  
  
    for (;;) {  
        OSSemPend(RandomSem, 0, &err); /* Acquire semaphore to perform random numbers */  
        x = random(80);           /* Find X position where task number will appear */  
        y = random(16);           /* Find Y position where task number will appear */  
        OSSemPost(RandomSem);    /* Release semaphore */  
        /* Display the task number on the screen */  
        PC_DispChar(x, y + 5, *(char *)data, DISP_FGND_LIGHT_GRAY);  
        OSTimeDly(1);           /* Delay 1 clock tick */  
    }  
}
```



A.0.2 INCLUDES.H (例 1)

```
/*  
*****  
*          uC/OS-II  
*          The Real-Time Kernel  
*  
*          (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL  
*          All Rights Reserved  
*  
*          MASTER INCLUDE FILE  
*****  
*/  
  
#include <stdio.h>  
#include <string.h>  
#include <ctype.h>
```

```
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <setjmp.h>

#include "\software\ucos-ii\ix86l\os_cpu.h"
#include "os_cfg.h"
#include "\software\blocks\pc\source\pc.h"
#include "\software\ucos-ii\source\ucos_ii.h"
```



A.0.3 OS_CFG.H (例 1)

```
/*
*****
*          uC/OS-II
*          The Real-Time Kernel
*
*      (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*          All Rights Reserved
*
*          Configuration for Intel 80x86 (Large)
*
* File : OS_CFG.H
* By   : Jean J. Labrosse
*****
*/
/*
*****
*          uC/OS-II CONFIGURATION
*****
*/
#define OS_MAX_EVENTS    2 /* Max. number of event control blocks in your application ... */
/* ...MUST be>= 2 */ /* */
#define OS_MAX_MEM_PART  2 /* Max. number of memory partitions ... */
/* ... MUST be >= 2 */ /* */
#define OS_MAX_QS        2 /* Max. number of queue control blocks in your application ... */
/* ... , MUST be >= 2 */ /* */
#define OS_MAX_TASKS     11 /* Max. Number of tasks in your application... */
/* ... MUST be >= 2 */ /* */
#define OS_LOWEST_PRIO  12 /* Defines the lowest priority that can be assigned ... */
/* ... MUST NEVER be higher than 63! */ /*
```

```

#define OS_TASK_IDLE_STK_SIZE 512 /* Idle task stack size (# of 16-bit wide entries) */

#define OS_TASK_STAT_EN    1      /* Enable (1) or Disable(0) the statistics task */
#define OS_TASK_STAT_STK_SIZE 512 /* Statistics task stack size (# of 16-bit wide entries)

#define OS_CPU_HOOKS_EN     1      /* uC/OS-II hooks are found in the processor port files */
#define OS_MBOX_EN          0      /* Include code for MAILBOXES */
#define OS_MEM_EN           0 /* Include code for MEMORY MANAGER (fixed sized memory blocks) */
#define OS_Q_EN              0 /* Include code for QUEUES */
#define OS_SEM_EN            1 /* Include code for SEMAPHORES */
#define OS_TASK_CHANGE_PRIO_EN 0 /* Include code for OSTaskChangePrio() */
#define OS_TASK_CREATE_EN    1 /* Include code for OSTaskCreate() */
#define OS_TASK_CREATE_EXT_EN 0 /* Include code for OSTaskCreateExt() */
#define OS_TASK_DEL_EN       0 /* Include code for OSTaskDel() */
#define OS_TASK_SUSPEND_EN   0 /* Include code for OSTaskSuspend() and OSTaskResume() */

#define OS_TICKS_PER_SEC     200   /* Set the number of ticks in one second */

```

A.1 例 2

A.1.1 EX2L.C

```

/*
*****uC/OS-II
*The Real-Time Kernel
*
*(c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*All Rights Reserved
*
*V2.00
*
*EXAMPLE #2
*****
*/
#include "includes.h"

/*
*****CONSTANTS
*****
```

```

/*
#define   TASK_STK_SIZE    512    /* Size of each task's stacks (# of WORDS)      */

#define   TASK_START_ID      0    /* Application tasks IDs                         */
#define   TASK_CLK_ID         1
#define   TASK_1_ID           2
#define   TASK_2_ID           3
#define   TASK_3_ID           4
#define   TASK_4_ID           5
#define   TASK_5_ID           6

#define   TASK_START_PRIO     10 /* Application tasks priorities                  */
#define   TASK_CLK_PRIO        11
#define   TASK_1_PRIO          12
#define   TASK_2_PRIO          13
#define   TASK_3_PRIO          14
#define   TASK_4_PRIO          15
#define   TASK_5_PRIO          16

/*
***** VARIABLES *****
*/
OS_STK  TaskStartStk[TASK_STK_SIZE]; /* Startup    task stack      */
OS_STK  TaskClkStk[TASK_STK_SIZE];   /* Clock      task stack      */
OS_STK  Task1Stk[TASK_STK_SIZE];    /* Task #1    task stack      */
OS_STK  Task2Stk[TASK_STK_SIZE];    /* Task #2    task stack      */
OS_STK  Task3Stk[TASK_STK_SIZE];    /* Task #3    task stack      */
OS_STK  Task4Stk[TASK_STK_SIZE];    /* Task #4    task stack      */
OS_STK  Task5Stk[TASK_STK_SIZE];    /* Task #5    task stack      */

OS_EVENT *AckMbox;                /* Message mailboxes for Tasks #4 and #5 */
OS_EVENT *TxMbox;

/*
***** FUNCTION PROTOTYPES *****
*/
void    TaskStart(void *data);      /* Function prototypes of tasks      */
void    TaskClk(void *data);

```



附录 A 源代码范例

```
void      Task1(void *data);
void      Task2(void *data);
void      Task3(void *data);
void      Task4(void *data);
void      Task5(void *data);

/*
***** MAIN *****
*/
void main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE);      /* Clear the screen */

    OSInit();                            /* Initialize uC/OS-II */

    PC_DOSSaveReturn();                 /* Save environment to return to DOS */

    PC_VectSet(uCOS, OSCtxSw);          /* Install uC/OS-II's context switch vector */

    PC_ElapsedInit();                  /* Initialized elapsed time measurement */

    OSTaskCreateExt(TaskStart, (void *)0, &TaskStartStk[TASK_STK_SIZE-1], TASK_START_PRIO,
                    TASK_START_ID, &TaskStartStk[0], TASK_STK_SIZE, (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

    OSStart();                          /* Start multitasking */
}

/*
***** STARTUP TASK *****
*/
void TaskStart (void *data)
{
    char   s[80];
    INT16S key;

    data = data;                      /* Prevent compiler warning */
}

超星阅览器提醒您：  
使用本复制品  
请尊重相关知识产权！
```

```

PC_DispStr(26, 0, "uC/OS-II, The Real-Time Kernel", DISP_FGND_WHITE + DISP_BGND_RED +
DISP_BLINK);

PC_DispStr(33, 1, "Jean J. Labrosse", DISP_FGND_WHITE);
PC_DispStr(36, 3, "EXAMPLE #2", DISP_FGND_WHITE);

PC_DispStr(0, 9, "Task           Total Stack   Free Stack Used Stack ExecTime (uS)",

DISP_FGND_WHITE);
PC_DispStr(0,10,"----- ----- ----- ----- ----- ----- ----- -----", DISP_FGND_WHITE);
PC_DispStr(0, 12, "TaskStart():", DISP_FGND_WHITE);
PC_DispStr(0, 13, "TaskClk() :", DISP_FGND_WHITE);
PC_DispStr(0, 14, "Task1()  :", DISP_FGND_WHITE);
PC_DispStr(0, 15, "Task2()  :", DISP_FGND_WHITE);
PC_DispStr(0, 16, "Task3()  :", DISP_FGND_WHITE);
PC_DispStr(0, 17, "Task4()  :", DISP_FGND_WHITE);
PC_DispStr(0, 18, "Task5()  :", DISP_FGND_WHITE);
PC_DispStr(28, 24, "<-PRESS 'ESC' TO QUIT->", DISP_FGND_WHITE + DISP_BLINK);

OS_ENTER_CRITICAL();          /* Install uC/OS-II's clock tick ISR */
PC_VectSet(0x08, OSTickISR);
PC_SetTickRate(OS_TICKS_PER_SEC); /* Reprogram tick rate */
OS_EXIT_CRITICAL();

PC_DispStr(0, 22, "Determining CPU's capacity ...", DISP_FGND_WHITE);
OSStatInit();                /* Initialize uC/OS-II's statistics */
PC_DisClearLine(22, DISP_FGND_WHITE + DISP_BGND_BLACK);

AckMbox = OSMsgBoxCreate((void *)0); /* Create 2 message mailboxes . */
TxMbox = OSMsgBoxCreate((void *)0);

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, (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, (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, (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);

```

附录 A 源代码范例

```
OSTaskCreateExt(Task4, (void *)0, &Task4Stk[TASK_STK_SIZE-1], TASK_4_PRIO,
                TASK_4_ID, &Task4Stk[0], TASK_STK_SIZE, (void *)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

OSTaskCreateExt(Task5, (void *)0, &Task5Stk[TASK_STK_SIZE-1], TASK_5_PRIO,
                TASK_5_ID, &Task5Stk[0], TASK_STK_SIZE, (void *)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

PC_DispStr(0, 22, "#Tasks      : xxxxx CPU Usage: xxx %", DISP_FGND_WHITE);
PC_DispStr(0, 23, "#Task switch/sec: xxxxx", DISP_FGND_WHITE);

for (;;) {
    sprintf(s, "%5d", OSTaskCtr); /* Display #tasks running */
    PC_DispStr(18, 22, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
    sprintf(s, "%3d", OSCPUUsage); /* Display CPU usage in % */
    PC_DispStr(36, 22, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
    sprintf(s, "%5d", OSCtxSwCtr); /* Display #context switches per second */
    PC_DispStr(18, 23, s, DISP_FGND_BLUE + DISP_BGND_CYAN);

    OSCtxSwCtr = 0; /* Clear context switch counter */

    sprintf(s, "V%3.2f", (float)OSVersion() * 0.01);
    PC_DispStr(75, 24, s, DISP_FGND_YELLOW + DISP_BGND_BLUE);

    if (PC_GetKey(&key)) { /* See if key has been pressed */
        if (key == 0x1B) { /* Yes, see if it's the ESCAPE key */
            PC_DOSReturn(); /* Yes, return to DOS */
        }
    }

    OSTimeDly(OS_TICKS_PER_SEC); /* Wait one second */
}

/*
*****
*          TASK #1
*
* Description: This task executes every 100 ms and measures the time it takes to perform stack
*   checking for each of the 5 application tasks. Also, this task displays the statistics related
*   to
*   each task's stack usage.
*****
*/
```

```
void Task1 (void *pdata)
{
    TNT8U      err;
    OS_STK_DATA data;           /* Storage for task stack data */
    INT16J     time;            /* Execution time (in uS) */
    INT8U      i;
    char       s[80];

    pdata = pdata;
    for (;;) {
        for (i = 0; i < 7; i++) {
            PC_ElapsedStart();
            err = OSTaskStkChk(TASK_START_PRIO+i, &data);
            time = PC_ElapsedStop();
            if (err == OS_NO_ERR) {
                sprintf(s, "%3ld      %3ld      %3ld      %5d",
                        data.OSFree + data.OSUsed,
                        data.OSFree,
                        data.OSUsed,
                        time);
                PC_DispStr(19, 12+i, s, DISP_FGND_YELLOW);
            }
        }
        OSTimeDlyHMSM(0, 0, 0, 100);          /* Delay for 100 ms */
    }
}

/*
*****
*               TASK #2
*
* Description: This task displays a clockwise rotating wheel on the screen.
*****
*/
void Task2 (void *data)
{
    data = data;
    for (;;) {
        PC_DispChar(70, 15, '|', DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
        PC_DispChar(70, 15, '/', DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
        PC_DispChar(70, 15, '-', DISP_FGND_WHITE + DISP_BGND_RED);
    }
}
```

附录 A 源代码范例

```
OSTimeDly(10);
PC_DispChar(70, 15, '\\', DISP_FGND_WHITE + DISP_BGND_RED);
OSTimeDly(10);
}

}

/*
*****
*          TASK #3
*
* Description: This task displays a counter-clockwise rotating wheel on the screen.
*
* Note(s)    : I allocated 100 bytes of storage on the stack to artificially 'eat' up stack space.
*****
*/
void Task3 (void *data)
{
    char    dummy[500];
    INT16U i;

    data = data;
    for (i = 0; i < 499; i++) {           /* Use up the stack with 'junk'           */
        dummy[i] = '?';
    }
    for (;;) {
        PC_DispChar(70, 16, '|', DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '\\', DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '--', DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '/' , DISP_FGND_WHITE + DISP_BGND_BLUE);
    }
}
```

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

    data = data;
    txmsg = 'A';
    for (;;) {
        while (txmsg <= 'Z') {
            OSMboxPost(TxMbox, (void *)&txmsg); /* Send message to Task #5 */
            OSMboxPend(AckMbox, 0, &err); /* Wait for acknowledgement from Task #5 */
            txmsg++; /* Next message to send */
        }
        txmsg = 'A'; /* Start new series of messages */
    }
}

/*
***** TASK #5 *****
*
* Description: This task displays messages sent by Task #4. When the message is displayed,
*      Task #5 acknowledges Task #4.
*****
*/
void Task5 (void *data)
{
    char *rxmsg;
    INT8U err;

    data = data;
    for (;;) {
        rxmsg = (char *)OSMboxPend(TxMbox, 0, &err); /* Wait for message from Task #4 */
        PC_DispChar(70, 18, *rxmsg, DISP_FGND_YELLOW + DISP_BGND_RED);
        OSTimeDlyHMSM(0, 0, 1, 0); /* Wait 1 second */
        OSMboxPost(AckMbox, (void *)1); /* Acknowledge reception of msg */
    }
}
*/
```

```
*****
*          CLOCK TASK
*****
*/
void TaskClk (void *data)
{
    struct time now;
    struct date today;
    char      s[40];

    data = data;
    for (;;) {
        PC_GetDateTime(s);
        PC_DispStr(0, 24, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

A.1.2 INCLUDES.H (例 2)

```
/*
*****
*          uC/OS-II
*          The Real-Time Kernel
*
*          (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*          All Rights Reserved
*
*          MASTER INCLUDE FILE
*****
*/
#include <stdio.h>
#include <string.h>
#include <cctype.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <setjmp.h>

#include "\software\ucos-i3\ix86l\os_cpu.h"
#include "os_cfg.h"
#include "\software\blocks\pc\source\pc.h"
```

```
#include    "\software\ucos-ii\source\ucos_ii.h"
```

A.1.3 OS_CFG.H (例 2)

```
/*
***** uC/OS-II
* The Real-Time Kernel
*
* (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
* All Rights Reserved
*
* Configuration for Intel 80x86 (Large)
*
* File : OS_CFG.H
* By   : Jean J. Labrosse
***** */

/*
***** uC/OS CONFIGURATION
***** */

#define OS_MAX_EVENTS    20 /* Max. number of event control blocks in your application ... */
/* ... MUST be >= 2 */
#define OS_MAX_MEM_PART  10 /* Max. number of memory partitions ... */
/* ... MUST be >= 2 */
#define OS_MAX_SEM       5 /* Max. number of semaphores in your application ... */
```

```

#define OS_MEM_EN          0 /* Include code for MEMORY MANAGER (fixed sized memory blocks) */
#define OS_Q_EN            0 /* Include code for QUEUES */
#define OS_SEM_EN          0 /* Include code for SEMAPHORES */
#define OS_TASK_CHANGE_PRIO_EN 0 /* Include code for OSTaskChangePrio() */
#define OS_TASK_CREATE_EN   0 /* Include code for OSTaskCreate() */
#define OS_TASK_CREATE_EXT_EN 1 /* Include code for OSTaskCreateExt() */
#define OS_TASK_DEL_EN      0 /* Include code for OSTaskDel() */
#define OS_TASK_SUSPEND_EN  0 /* Include code for OSTaskSuspend() and OSTaskResume() */

#define OS_TICKS_PER_SEC    200 /* Set the number of ticks in one second */

```

A.2 例 3

A.2.1 EX3L.C

```

/*
***** uC/OS-II *****
*           The Real-Time Kernel
*
*       (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*           All Rights Reserved
*
*           v2.00
*
*           EXAMPLE #3
***** .
*/
#define EX3_GLOBALS
#include "includes.h"

/*
***** CONSTANTS *****
*/
#define TASK_STK_SIZE 512 /* Size of each task's stacks (# of WORDs) */
#define TASK_START_ID 0 /* Application tasks */
#define TASK_CLK_ID 1

```

```

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

#define      TASK_START_PRIO    10      /* Application tasks priorities */
#define      TASK_CLK_PRIO       11
#define      TASK_1_PRIO         12
#define      TASK_2_PRIO         13
#define      TASK_3_PRIO         14
#define      TASK_4_PRIO         15
#define      TASK_5_PRIO         16

#define      MSG_QUEUE_SIZE     20      /* Size of message queue used in example */

/*
***** VARIABLES *****
*/
OS_STK      TaskStartStk[TASK_STK_SIZE];   /* Startup task stack */
OS_STK      TaskClkStk[TASK_STK_SIZE];   /* Clock task stack */
OS_STK      Task1Stk[TASK_STK_SIZE];    /* Task #1 task stack */
OS_STK      Task2Stk[TASK_STK_SIZE];    /* Task #2 task stack */
OS_STK      Task3Stk[TASK_STK_SIZE];    /* Task #3 task stack */
OS_STK      Task4Stk[TASK_STK_SIZE];    /* Task #4 task stack */
OS_STK      Task5Stk[TASK_STK_SIZE];    /* Task #5 task stack */

OS_EVENT    *MsgQueue;                  /* Message queue pointer */
void        *MsgQueueTbl[20];           /* Storage for messages */

/*
***** FUNCTION PROTOTYPES *****
*/
void        TaskStart(void *data);        /* Function prototypes of tasks */
void        TaskClk(void *data);
void        Task1(void *data);
void        Task2(void *data);
void        Task3(void *data);

```



附录 A 源代码范例

```
void        Task4(void *data);
void        Task5(void *data);

/*
*****MAIN*****
*/
void main (void)
{
    PC_DispClrScr(DISP_BGND_BLACK);      /* Clear the screen */

    OSInit();                            /* Initialize uC/OS-II */

    PC_DOSSaveReturn();                 /* Save environment to return to DOS */

    PC_VectSet(uCOS, OSCtxSw);         /* Install uC/OS-II's context switch vector */

    PC_ElapsedInit();                  /* Initialized elapsed time measurement */

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

/*
*****STARTUP TASK*****
*/
void TaskStart (void *data)
{
    char s[80];
    INT16S key;

    data = data;                      /* Prevent compiler warning */

    PC_DispStr(26, 0, "uC/OS-II, The Real-Time Kernel", DISP_FGND_WHITE + DISP_BGND_RED +
DISP_BLINK);
    PC_DispStr(33, 1, "Jean J. Labrosse", DISP_FGND_WHITE);
    PC_DispStr(36, 3, "EXAMPLE #3", DISP_FGND_WHITE);
    PC_DispStr(0,9,"Task Name          Counter  Exec.Time(uS)  Tot.Exec.Time(uS)  %Tot.");
}
```



```
DISP_FGND_WHITE);

PC_DispStr(0, 10, "-----");
PC_DispStr(0, 10, ", DISP_FGND_WHITE);

PC_DispStr(0, 22, "Determining CPU's capacity ...", DISP_FGND_WHITE);
PC_DispStr(28, 24, "<-PRESS 'ESC' TO QUIT->", DISP_FGND_WHITE + DISP_BLINK);

OS_ENTER_CRITICAL(); /* Install uC/OS-II's clock tick ISR */
PC_VectSet(0x08, OSTickISR);
PC_SetTickRate(OS_TICKS_PER_SEC); /* Reprogram tick rate */
OS_EXIT_CRITICAL();

OSStatInit();

MsgQueue = OSQCreate(&MsgQueueTbl[0], MSG_QUEUE_SIZE); /* Create a message queue */

strcpy(TaskUserData[TASK_CLK_ID].TaskName, "Clock Task");
OSTaskCreateExt(TaskClk, (void *)0, &TaskClkStk[TASK_STK_SIZE-1], TASK_CLK_PRIO,
    TASK_CLK_ID, &TaskClkStk[0], TASK_STK_SIZE, &TaskUserData[TASK_CLK_ID], 0);

strcpy(TaskUserData[TASK_1_ID].TaskName, "MsgQ Tx Task");
OSTaskCreateExt(Task1, (void *)0, &Task1Stk[TASK_STK_SIZE-1], TASK_1_PRIO,
    TASK_1_ID, &Task1Stk[0], TASK_STK_SIZE, &TaskUserData[TASK_1_ID], 0);

strcpy(TaskUserData[TASK_2_ID].TaskName, "MsgQ Rx Task #1");
OSTaskCreateExt(Task2, (void *)0, &Task2Stk[TASK_STK_SIZE-1], TASK_2_PRIO,
    TASK_2_ID, &Task2Stk[0], TASK_STK_SIZE, &TaskUserData[TASK_2_ID], 0);

strcpy(TaskUserData[TASK_3_ID].TaskName, "MsgQ Rx Task #2");
OSTaskCreateExt(Task3, (void *)0, &Task3Stk[TASK_STK_SIZE-1], TASK_3_PRIO,
    TASK_3_ID, &Task3Stk[0], TASK_STK_SIZE, &TaskUserData[TASK_3_ID], 0);

strcpy(TaskUserData[TASK_4_ID].TaskName, "MboxPostPendTask");
OSTaskCreateExt(Task4, (void *)0, &Task4Stk[TASK_STK_SIZE-1], TASK_4_PRIO,
    TASK_4_ID, &Task4Stk[0], TASK_STK_SIZE, &TaskUserData[TASK_4_ID], 0);

strcpy(TaskUserData[TASK_5_ID].TaskName, "TimeDlyTask");
OSTaskCreateExt(Task5, (void *)0, &Task5Stk[TASK_STK_SIZE-1], TASK_5_PRIO,
    TASK_5_ID, &Task5Stk[0], TASK_STK_SIZE, &TaskUserData[TASK_5_ID], 0);

PC_DispStr(0, 22, "#Tasks : xxxx CPU Usage: xxx %", DISP_FGND_WHITE);
PC_DispStr(0, 23, "#Task switch/sec: xxxx", DISP_FGND_WHITE);
for (;;) {
    sprintf(s, "%5d", OSTaskCtr); /* Display #tasks running */
    PC_DispStr(18, 22, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
    sprintf(s, "%3d", OSCPUUsage); /* Display CPU usage in % */
    PC_DispStr(22, 23, s, DISP_FGND_WHITE + DISP_BLINK);
}
```

附录 A 源代码范例

```
PC_DispStr(36, 22, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
sprintf(s, "%5d", OSCTxSwCtr); /* Display #context switches per second */
PC_DispStr(18, 23, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
sprintf(s, "V%3.2f", (float)OSVersion() * 0.01);
PC_DispStr(75, 24, s, DISP_FGND_YELLOW + DISP_BGND_BLUE);

OSCTxSwCtr = 0; /* Clear the context switch counter */

if (PC.GetKey(&key)) { /* See if key has been pressed */
    if (key == 0x1B) { /* Yes, see if it's the ESCAPE key */
        PC_DOSReturn(); /* Yes, return to DOS */
    }
}

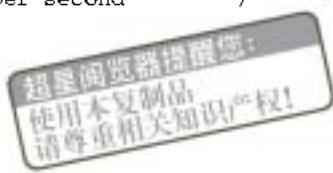
OSTimeDly(OS_TICKS_PER_SEC); /* Wait one second */

}

/*
*****
*          TASK #1
*
* Description: This task executes 5 times per second but doesn't do anything.
*****
*/
void Task1 (void *data)
{
    char one  = '1';
    char two  = '2';
    char three = '3';

    data = data;
    for (;;) {
        OSQPost(MsgQueue, (void *)&one);
        OSTimeDlyHMSM(0, 0, 1, 0); /* Delay For 1 second */
        OSQPost(MsgQueue, (void *)&two);
        OSTimeDlyHMSM(0, 0, 0, 500); /* Delay for 500 mS */
        OSQPost(MsgQueue, (void *)&three);
        OSTimeDlyHMSM(0, 0, 1, 0); /* Delay for 1 second */
    }
}
/*

```



```
*****
*                                TASK #2
*
* Description: This task waits for messages sent by task #1.
*****
*/
void Task2 (void *data)
{
    INT8U *msg;
    INT8U err;

    data = data;
    for (;;) {
        msg = (INT8U *)OSQPend(MsgQueue, 0, &err); /* Wait forever for message */
        PC_DispChar(70, 14, *msg, DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDlyHMSM(0, 0, 0, 500);           /* Delay for 500 mS */
    }
}

/*
*****
*                                TASK #3
*
* Description: This task waits for up to 500 mS for a message sent by task #1.
*****
*/
void Task3 (void *data)
{
    INT8U *msg;
    INT8J err;

    data = data;
    for (;;) {
        msg = (INT8J *)OSQPend(MsgQueue, OS_TICKS_PER_SEC/4, &err);/* Wait up to 250 mS for a msg */
        if (err == OS_TIMEOUT) {
            PC_DispChar(70, 15, 'T', DISP_FGND_YELLOW + DISP_BGND_RED);
        } else {
            PC_DispChar(70, 15, *msg, DISP_FGND_YELLOW + DISP_BGND_BLUE);
        }
    }
}
```

附录 A 源代码范例

```
/*
***** TASK #4 *****
* Description: This task posts a message to a mailbox and then immediately reads the message.
***** */

void Task4 (void *data)
{
    OS_EVENT *mbox;
    INT8U     err;

    data = data;
    mbox = OSMboxCreate((void *)0);
    for (;;) {
        OSMboxPost(mbox, (void *)1);          /* Send message to mailbox */
        OSMboxPending(mbox, 0, &err);         /* Get message from mailbox */
        OSTimeDlyHMSM(0, 0, 0, 10);          /* Delay 10 mS */
    }
}

/*
***** TASK #5 *****
* Description: This task simply delays itself. We basically want to determine how long OSTimeDly()
*      takes to execute.
***** */

void Task5 (void *data)
{
    data = data;
    for (;;) {
        OSTimeDly(1);
    }
}

/*
***** DISPLAY TASK RELATED STATISTICS *****
*/
```



```

void DispTaskStat (INT8U id)
{
    char s[80];

    sprintf(s, "%-18s %5u      %5u      %10ld",
        TaskUserData[id].TaskName,
        TaskUserData[id].TaskCtr,
        TaskUserData[id].TaskExecTime,
        TaskUserData[id].TaskTotExecTime);
    PC_DispStr(0, id + 11, s, DISP_FGND_LIGHT_GRAY);
}

/*
***** CLOCK TASK *****
*/
void TaskClk (void *data)
{
    char s[40];

    data = data;
    for (;;) {
        PC_GetDateTime(s);
        PC_DispStr(0, 24, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
        OSTimeDlyHMSM(0, 0, 0, 100); /* Execute every 100 mS */
    }
}

/*
***** TASK CREATION HOOK *****
*
* Description: This function is called when a task is created.
*
* Arguments : ptcb    is a pointer to the task being created.
*
* Note(s)   : 1) Interrupts are disabled during this call.
*/
void OSTaskCreateHook (OS_TCB *ptcb)

```



附录A 源代码范例



```
{  
    ptcb = ptcb; /* Prevent compiler warning  
}  
  
/*  
*****  
*  
*           TASK DELETION HOOK  
*  
*  
* Description: This function is called when a task is deleted.  
*  
* Arguments : ptcb     is a pointer to the task being created.  
*  
* Note(s)   : 1) Interrupts are disabled during this call.  
*****  
*/  
void OSTaskDelHook (OS_TCB *ptcb)  
{  
    ptcb = ptcb; /* Prevent compiler warning */  
}  
  
/*  
*****  
*  
*           TASK SWITCH HOOK  
*  
*  
* Description: This function is called when a task switch is performed. This allows you to perform  
*      other operations during a context switch.  
*  
* Arguments : none  
*  
* Note(s): 1) Interrupts are disabled during this call.  
*          2) It is assumed that the global pointer 'OSTCBHighRdy' points to the TCB of the  
*              task that will be 'switched in' (i.e. the highest priority task) and, 'OSTCBCur'  
*              points to the task being switched out (i.e. the preempted task).  
*****  
*/  
void OSTaskSwHook (void)  
{  
    INT16U      time;  
    TASK_USER_DATA *puser;  
    time = PC_ElapsedStop(); /* This task is done */  
    PC_ElapsedStart(); /* Start for next task */  
    puser = OSTCBCur->OSTCBExtPtr; /* Point to used data */  
    if (puser != (void *)0)  
        puser->TaskCtr++; /* Increment task counter */  
}
```

```

    puser->TaskExecTime -= time; /* Update the task's execution time */
    puser->TaskTotExecTime += time; /* Update the task's total execution time */
}

/*
*****
*          STATISTIC TASK HOOK
*
* Description: This function is called every second by uC/OS-II's statistics task. This allows
*               your application to add functionality to the statistics task.
*
* Arguments : none
*****
*/
void OSTaskStatHook (void)
{
    char s[80];
    INT8U i;
    INT32U total;
    INT8U pct;
    total = 0L; /* Totalize TOT. EXEC. TIME for each task */
    for (i = 0; i < 7; i++) {
        total += TaskUserData[i].TaskTotExecTime;
        DispTaskStat(i); /* Display task data */
    }
    if (total > 0) {
        for (i = 0; i < 7; i++) { /* Derive percentage of each task */
            pct = 100 * TaskUserData[i].TaskTotExecTime / total;
            sprintf(s, "%3d %%", pct);
            PC_DispStr(62, i + 11, s, DISP_FGND_YELLOW);
        }
    }
    if (total > 1000000000L) { /* Reset total time counters at 1 billion */
        for (i = 0; i < 7; i++) {
            TaskUserData[i].TaskTotExecTime = 0L;
        }
    }
}

/*
*****
*          TICK HOOK
*
* Description: This function is called every tick.
*/

```

```

/*
 * Arguments : none
 *
 * Note(s)   : 1) Interrupts may or may not be are ENABLED during this call.
 ****
 */
void OSTimeTickHook (void)
{
}

```



A.2.2 INCLUDES.H (例 3)

```

/*
***** uC/OS-II
*          The Real-Time Kernel
*
*      (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*          All Rights Reserved
*
*          MASTER INCLUDE FILE
*****
*/
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <setjmp.h>

#include "\software\ucos-ii\ix86l\os_cpu.h"
#include "os_cfg.h"
#include "\software\blocks\pc\source\pc.h"
#include "\software\ucos-ii\source\ucos_ii.h"

#ifndef EX3_GLOBALS
#define EX3_EXT
#else
#define EX3_EXT extern
#endif

/*

```

```
*****
*          DATA TYPES
*****
*/
typedef struct {
    char    TaskName[30];
    INT16U TaskCtr;
    INT16U TaskExecTime;
    INT32U TaskTotExecTime;
} TASK_USER_DATA;

/*
*****
*          VARIABLES
*****
*/
EX3_EXT TASK_USER_DATA TaskUserData[10];

/*
*****
*          FUNCTION PROTOTYPES
*****
*/
void DispTaskStat(INT8U id);
```



A.2.3 OS_CFG.H (例 3)

```
/*
*****
*          uC/OS-II
*          The Real-Time Kernel
*
*          (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*          All Rights Reserved
*
*          Configuration for Intel 80x86 (Large)
*
* File : OS_CFG.H
* By   : Jean J. Labrosse
*****
*/

```

附录 A 源代码范例

```
/*
***** uC/OS-II CONFIGURATION *****
*/
#define OS_MAX_EVENTS    20 /* Max. number of event control blocks in your application ... */
                           /* ... MUST be >= 2 */
#define OS_MAX_MEM_PART 10 /* Max. number of memory partitions ... */
                           /* ... MUST be >= 2 */
#define OS_MAX_QS        5  /* Max. number of queue control blocks in your application ... */
                           /* ... MUST be >= 2 */
#define OS_MAX_TASKS    32 /* Max. number of tasks in your application... */
                           /* ... MUST be >= 2 */

#define OS_LOWEST_PRIO  63 /* Defines the lowest priority that can be assigned ... */
                           /* ... MUST NEVER be higher than 63! */

#define OS_TASK_IDLE_STK_SIZE 512 /* Idle task stack size (# of 16-bit wide entries) */

#define OS_TASK_STAT_EN     1      /* Enable (1) or Disable(0) the statistics task */
#define OS_TASK_STAT_STK_SIZE 512/* Statistics task stack size (# of 16-bit wide entries) */

#define OS_CPU_HOOKS_EN     0      /* uC/OS-II hooks are NOT found in the processor port files */
#define OS_MBOX_EN          1      /* Include code for MAILBOXES */
#define OS_MEM_EN           0      /* Include code for MEMORY MANAGER (fixed sized memory blocks)
*/
#define OS_Q_EN              1      /* Include code for QUEUES */
#define OS_SEM_EN             0      /* Include code for SEMAPHORES */
#define OS_TASK_CHANGE_PRIO_EN 0      /* Include code for OSTaskChangePrio() */
#define OS_TASK_CREATE_EN     0      /* Include code for OSTaskCreate() */
#define OS_TASK_CREATE_EXT_EN 1      /* Include code for OSTaskCreateExt() */
#define OS_TASK_DEL_EN       0      /* Include code for OSTaskDel() */
#define OS_TASK_SUSPEND_EN   0      /* Include code for OSTaskSuspend() and OSTaskResume() */

#define OS_TICKS_PER_SEC    200 /* Set the number of ticks in one second */
```

A. 3 PC 服务

A.3.1 PC.C

```
/*
*****PC SUPPORT FUNCTIONS*****
*
* (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
* All Rights Reserved
*
* File : PC.C
* By   : Jean J. Labrosse
***** */

#include "includes.h"

/*
*****CONSTANTS*****
*/
#define DISP_BASE          0xB800 /* Base segment of display (0xB800=VGA, 0xB000=Mono) */
#define DISP_MAX_X         80    /* Maximum number of columns */
#define DISP_MAX_Y         25    /* Maximum number of rows */

#define TICK_T0_8254_CWR   0x43  /* 8254 PIT Control Word Register address. */
#define TICK_T0_8254_CTR0  0x40  /* 8254 PIT Timer 0 Register address. */
#define TICK_T0_8254_CTR1  0x41  /* 8254 PIT Timer 1 Register address. */
#define TICK_T0_8254_CTR2  0x42  /* 8254 PIT Timer 2 Register address. */

#define TICK_T0_8254_CTR0_MODE3 0x36 /* 8254 PIT Binary Mode 3 for Counter 0 control word. */
#define TICK_T0_8254_CTR2_MODE0 0xB0 /* 8254 PIT Binary Mode 0 for Counter 2 control word. */
#define TICK_T0_8254_CTR2_LATCH 0x80 /* 8254 PIT Latch command control word */

#define VECT_TICK           0x08  /* Vector number for 82C54 timer tick */
#define VECT_DOS_CHAIN      0x81  /* Vector number used to chain DOS */

/*
*****LOCAL GLOBAL VARIABLES*****
*/

```

附录 A 源代码范例

```
static INT16U PC_ElapsedOverhead;
static jmp_buf PC_JumpBuf;
static BOOLEAN PC_ExitFlag;
void (*PC_TickISR)(void);

/*
*****  

*          CLEAR SCREEN  

*  

* Description : This function clears the PC's screen by directly accessing video RAM instead of  

*      using the BIOS. It assumed that the video adapter is VGA compatible. Video RAM starts  

*      at absolute address 0x000B8000. Each character on the screen is composed of two bytes:  

*      the ASCII character to appear on the screen followed by a video attribute. An attribute  

*      of 0x07 displays the character in WHITE with a black background.  

*  

* Arguments   : color  specifies the foreground/background color combination to use  

*                  (see PC.H for available choices)  

*  

* Returns     : None  

*****  

*/
void PC_DispcLrScr (INT8U color)
{
    INT8U far *pscr;
    INT16U i;

    pscr = MK_FP(DISP_BASE, 0x0000);
    for (i = 0; i < (DISP_MAX_X * DISP_MAX_Y); i++)
        /* PC display has 80 columns and 25 lines           */
        *pscr++ = ' ';                                /* Put ' ' character in video RAM   */
        *pscr++ = color;                            /* Put video attribute in video RAM */

}

/*
*****  

*          CLEAR A LINE  

*  

* Description : This function clears one of the 25 lines on the PC's screen by directly accessing  

*      video RAM instead of using the BIOS. It assumed that the video adapter is  

*      VGA compatible. Video RAM starts at absolute address 0x000B8000. Each character on  

*      the screen is composed of two bytes: the ASCII character to appear on the screen  

*      followed by a video attribute. An attribute of 0x07 displays the character in WHITE  

*      with a black background.
*
```

```
* Arguments : y           corresponds to the desired line to clear. Valid line numbers are from
*                           0 to 24. Line 0 corresponds to the topmost line.
*
*           color      specifies the foreground/background color combination to use
*                           (see PC.H for available choices)
*
* Returns   : None
*****
*/
void PC_DispClrLine (INT8U y, INT8U color)
{
    INT8U far *pscr;
    INT8U     i;

    pscr = MK_FP(DISP_BASE, (INT16U)y * DISP_MAX_X * 2);
    for (i = 0; i < DISP_MAX_X; i++) {
        *pscr++ = ' ';           /* Put ' ' character in video RAM */
        *pscr++ = color;         /* Put video attribute in video RAM */
    }
}

/*
*****
*           DISPLAY A SINGLE CHARACTER AT 'X' & 'Y' COORDINATE
*
* Description : This function writes a single character anywhere on the PC's screen. This function
*               writes directly to video RAM instead of using the BIOS for speed reasons. It assumed
*               that the video adapter is VGA compatible. Video RAM starts at absolute address
*               0x000B8000. Each character on the screen is composed of two bytes: the ASCII
*               character to appear on the screen followed by a video attribute. An attribute of
*               0x07 displays the character in WHITE with a black background.
*
* Arguments : x corresponds to the desired column on the screen. Valid columns numbers are
*               from 0 to 79. Column 0 corresponds to the leftmost column.
*               y corresponds to the desired row on the screen. Valid row numbers are from 0 to 24.
*               Line 0 corresponds to the topmost row.
*               c   is the ASCII character to display. You can also specify a character with a
*                   numeric value higher than 128. In this case, special character based graphics
*                   will be displayed.
*               color specifies the foreground/background color to use (see PC.H for available choices)
*                   and whether the character will blink or not.
*
* Returns   : None
*****
*/
```



附录 A 源代码范例

```
void PC_DispChar (INT8U x, INT8U y, INT8U c, INT8U color)
{
    INT8U far *pscr;
    INT16U offset;

    offset = (INT16U)y * DISP_MAX_X * 2 + (INT16U)x * 2; /* Calculate position on the screen */
/*
    pscr = MK_FP(DISP_BASE, offset);
    *pscr++ = c;                                /* Put character in video RAM */
    *pscr = color;                             /* Put video attribute in video RAM */
*/
/*
***** DISPLAY A STRING AT 'X' & 'Y' COORDINATE *****
* Description : This function writes an ASCII string anywhere on the PC's screen. This function
*               writes directly to video RAM instead of using the BIOS for speed reasons. It assumed
*               that the video adapter is VGA compatible. Video RAM starts at absolute address
*               0x000B8000. Each character on the screen is composed of two bytes: the ASCII character
*               to appear on the screen followed by a video attribute. An attribute of 0x07 displays
*               the character in WHITE with a black background.
*
* Arguments : x corresponds to the desired column on the screen. Valid columns numbers are
*               from 0 to 79. Column 0 corresponds to the leftmost column.
*               y corresponds to the desired row on the screen. Valid row numbers are from 0
*               to 24. Line 0 corresponds to the topmost row.
*               s Is the ASCII string to display. You can also specify a string containing
*               characters with numeric values higher than 128. In this case, special
*               character based graphics will be displayed.
*               color specifies the foreground/background color to use (see PC.H for available
*               choices) and whether the characters will blink or not.
*
* Returns   : None
*****
*/
void PC_DispStr (INT8U x, INT8U y, INT8U *s, INT8U color)
{
    INT8U far *pscr;
    INT16U offset;

    offset = (INT16U)y * DISP_MAX_X * 2 + (INT16U)x * 2; /* Calculate position of 1st character */
/*
    pscr = MK_FP(DISP_BASE, offset);
    while (*s) {

```



```

        *pscr++ = *s++;           /* Put character in video RAM      */
        *pscr++ = color;         /* Put video attribute in video RAM   */
    }

}

/*
***** RETURN TO DOS *****
*
* Description : This function returns control back to DOS by doing a 'long jump' back to the
*               saved location stored in 'PC_JumpBuf'. The saved location was established by
*               the function 'PC_DOSSaveReturn()'. After execution of the long jump, execution
*               will resume at the line following the 'set jump' back in 'PC_DOSSaveReturn()'.

* Setting the flag 'PC_ExitFlag' to TRUE ensures that the 'if' statement
*       in 'PC_DOSSaveReturn()' executes.

* Arguments   : None
*
* Returns     : None
***** SAVE DOS RETURN LOCATION *****
*/
void PC_DOSReturn (void)
{
    PC_ExitFlag = TRUE;          /* Indicate we are returning to DOS      */
    longjmp(PC_JumpBuf, 1);     /* Jump back to saved environment      */
}

/*
***** SAVE DOS RETURN LOCATION *****
*
* Description : This function saves the location of where we are in DOS so that it can be recovered.
*               This allows us to abort multitasking under μC/OS-II and return back to DOS as if we had
*               never left. When this function is called by 'main()', it sets 'PC_ExitFlag' to FALSE
*               so that we don't take the 'if' branch. Instead, the CPU registers are saved in the
*               long jump buffer 'PC_JumpBuf' and we simply return to the caller. If a 'long jump' is
*               performed using the jump buffer then, execution would resume at the 'if' statement and
*               this time, if 'PC_ExitFlag' is set to TRUE then we would execute the 'if' statements and
*               restore the DOS environment.

*
* Arguments   : None
*
* Returns     : None
***** */
void PC_DOSSaveReturn (void)

```

附录 A 源代码范例

```
{  
    PC_ExitFlag = FALSE;           /* Indicate that we are not exiting yet */  
    OSTickDOSCtr = 1;             /* Initialize the DOS tick counter */  
    PC_TickISR = PC_VectGet(VECT_TICK); /* Get MS-DOS's tick vector */  
  
    OS_ENTER_CRITICAL();  
    PC_VectSet(VECT_DOS_CHAIN, PC_TickISR); /* Store MS-DOS's tick to chain */  
    OS_EXIT_CRITICAL();  
  
    setjmp(PC_JumpBuf);          /* Capture where we are in DOS */  
    if (PC_ExitFlag == TRUE) {     /* See if we are exiting back to DOS */  
        OS_ENTER_CRITICAL();  
        PC_SetTickRate(18);        /* Restore tick rate to 18.2 Hz */  
        PC_VectSet(VECT_TICK, PC_TickISR); /* Restore DOS's tick vector */  
        OS_EXIT_CRITICAL();  
        PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK); /* Clear the display */  
        exit(0);                  /* Return to DOS */  
    }  
}  
  
/*  
*****  
*          ELAPSED TIME INITIALIZATION  
*  
* Description : This function initialize the elapsed time module by determining how long  
*                 the START and STOP functions take to execute. In other words, this function  
*                 calibrates this module to account for the processing time of the START and STOP functions.  
*  
* Arguments   : None.  
*  
* Returns     : None.  
*****  
*/  
void PC_ElapsedInit(void)  
{  
    PC_ElapsedOverhead = 0;  
    PC_ElapsedStart();  
    PC_ElapsedOverhead = PC_ElapsedStop();  
}  
  
/*  
*****  
*          INITIALIZE PC'S TIMER #2  
*  
* Description : This function initialize the PC's Timer #2 to be used to measure the time  
*****
```

```

*      between events. Timer #2 will be running when the function returns.
*
* Arguments : None.
*
* Returns   : None.
*****
*/
void PC_ElapsedStart(void)
{
    INT8U data;

    data = (INT8U)inp(0x61);           /* Disable timer #2 */
    data &= 0xFE;
    outp(0x61, data);
    outp(TICK_T0_8254_CWR, TICK_T0_8254_CTR2_MODE0); /* Program timer #2 for Mode 0 */
    outp(TICK_T0_8254_CTR2, 0xFF);
    outp(TICK_T0_8254_CTR2, 0xFF);
    data |= 0x01;                     /* Start the timer */
    outp(0x61, data);

}

/*
*****
* STOP THE PC'S TIMER #2 AND GET ELAPSED TIME
*
* Description : This function stops the PC's Timer #2, obtains the elapsed counts from when
*               it was started and converts the elapsed counts to micro-seconds.
*
* Arguments   : None.
*
* Returns     : The number of micro-seconds since the timer was last started.
*
* Notes      : - The returned time accounts for the processing time of the START and STOP functions.
*               - 54926 represents 54926S-16 or, 0.838097 which is used to convert timer counts to
*                 micro-seconds. The clock source for the PC's timer #2 is 1.19318 MHz (or 0.838097 us)
*****
*/
INT16U PC_ElapsedStop(void)
{
    INT8U data;
    INT8U low;
    INT8U high;
    INT16U cnts;
}

```

附录 A 源代码范例

```
data = inp(0x61);                                /* Disable the timer */  
data &= 0xFE;  
outp(0x61, data);  
  
outp(TICK_T0_8254_CWR, TICK_T0_8254_CTR2_LATCH); /* Latch the timer value */  
low = inp(TICK_T0_8254_CTR2);  
high = inp(TICK_T0_8254_CTR2);  
cnts = (INT16U)0xFFFF - (((INT16U)high << 8) + (INT16U)low); /* Compute time it took for operation */  
/*  
*****  
* GET THE CURRENT DATE AND TIME  
*  
* Description: This function obtains the current date and time from the PC.  
*  
* Arguments : s is a pointer to where the ASCII string of the current date and time will be stored.  
*             You must allocate at least 19 bytes (includes the NUL) of storage in the return  
*             string.  
*  
* Returns   : none  
*****  
*/  
void PC_GetDateTime (char *s)  
{  
    struct time now;  
    struct date today;  
  
    gettimeofday(&now);  
    getdate(&today);  
    sprintf(s, "%02d-%02d-%02d %02d:%02d:%02d",  
           today.da_mon,  
           today.da_day,  
           today.da_year,  
           now.ti_hour,  
           now.ti_min,  
           now.ti_sec);  
}  
  
/*  
*****
```

```
*          CHECK AND GET KEYBOARD KEY
*
* Description: This function checks to see if a key has been pressed at the keyboard and returns
*      TRUE if so. Also, if a key is pressed, the key is read and copied where the argument
*      is pointing to.
*
* Arguments : c      is a pointer to where the read key will be stored.
*
* Returns   : TRUE  if a key was pressed
*              FALSE otherwise
*****
*/
BOOLEAN PC_GetKey (INT16S *c)
{
    if (kbhit()) {           /* See if a key has been pressed */
        *c = getch();         /* Get key pressed */
        return (TRUE);
    } else {
        *c = 0x00;            /* No key pressed */
        return (FALSE);
    }
}

/*
*****
*          SET THE PC'S TICK FREQUENCY
*
* Description: This function is called to change the tick rate of a PC.
*
* Arguments : freq      is the desired frequency of the ticker (in Hz)
*
* Returns   : none
*
* Notes    : 1) The magic number 2386360 is actually twice the input frequency of the 8254 chip
*             which is always 1.193180 MHz.
*             2) The equation computes the counts needed to load into the 8254. The strange equation
*                 is actually used to round the number using integer arithmetic. This is equivalent
*                 to the floating point equation:
*
*                         1193180.0 Hz
*             count = ----- + 0.5
*                         freq
*****
*/
void PC_SetTickRate (TNT16U freq)
```

附录 A 源代码范例



```
/*
 * TNT16U count;

if (freq >= 18) {           /* See if we need to restore the DOS frequency      */
    count = 0;
} else if (freq > 0) {
    /* Compute 8254 counts for desired frequency and ... */
    /* ... round to nearest count                         */
    count = (INT16U)((INT32U)2386360L / freq + 1) >> 1;
} else {
    count = 0;
}

outp(TICK_T0_8254_CWR, TICK_T0_8254_CTR0_MODE3); /* Load the 8254 with desired frequency */
outp(TICK_T0_8254_CTR0, count & 0xFF);           /* Low byte                                */
outp(TICK_T0_8254_CTR0, (count >> 8) & 0xFF);   /* High byte                               */

}

/*
***** OBTAIN INTERRUPT VECTOR
*
* Description: This function reads the pointer stored at the specified vector.
*
* Arguments : vect is the desired interrupt vector number, a number between 0 and 255.
*
* Returns   : none
*****
*/
void *PC_VectGet (INT8U vect)
{
    return (getvect(vect));
}

/*
***** INSTALL INTERRUPT VECTOR
*
* Description: This function sets an interrupt vector in the interrupt vector table.
*
* Arguments : vect is the desired interrupt vector number, a number between 0 and 255.
*             isr  is a pointer to a function to execute when the interrupt or exception occurs.
*
* Returns   : none
*/
```



```
*****
*/
void PC_VectSet (INT8U vect, void (*isr)(void))
{
    setvect(vect, (void interrupt (*)())isr);
}
```

A.3.2 PC.H

```
/*
*****
*          PC SUPPORT FUNCTIONS
*
*      (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*          All Rights Reserved
*
* File : PC.H
* By   : Jean J. Labrosse
*****
*/
/*          CONSTANTS
*          COLOR ATTRIBUTES FOR VGA MONITOR
*
* Description: These #defines are used in the PC_Disp???() functions. The 'color' argument in
*              these function MUST specify a 'foreground' color, a 'background' and whether the
*              display will blink or not. If you don't specify a background color, BLACK is assumed.
*              You would specify a color combination as follows:
*
*          PC_DispChar(0, 0, 'A', DISP_FGND_WHITE + DISP_BGND_BLUE + DISP_BLINK);
*
*          To have the ASCII character 'A' blink with a white letter on a blue background.
*****
*/
#define DISP_FGND_BLACK      0x00
#define DISP_FGND_BLUE       0x01
#define DISP_FGND_GREEN      0x02
#define DISP_FGND_CYAN       0x03
#define DISP_FGND_RED        0x04
#define DISP_FGND_PURPLE     0x05
#define DISP_FGND_BROWN       0x06
#define DISP_FGND_LIGHT_GRAY 0x07
#define DISP_FGND_DARK_GRAY   0x08
```

附录 A 源代码范例



```
#define DISP_FGND_LIGHT_BLUE      0x09
#define DISP_FGND_LIGHT_GREEN     0x0A
#define DISP_FGND_LIGHT_CYAN      0x0B
#define DISP_FGND_LIGHT_RED       0x0C
#define DISP_FGND_LIGHT_PURPLE    0x0D
#define DISP_FGND_YELLOW          0x0E
#define DISP_FGND_WHITE           0x0F

#define DISP_BGND_BLACK           0x00
#define DISP_BGND_BLUE            0x10
#define DISP_BGND_GREEN           0x20
#define DISP_BGND_CYAN            0x30
#define DISP_BGND_RED             0x40
#define DISP_BGND_PURPLE          0x50
#define DISP_BGND_BROWN           0x60
#define DISP_BGND_LTGRAY          0x70

#define DISP_BLINK                 0x80

/*
*****
*          FUNCTION PROTOTYPES
*****
*/
void PC_DispClrScr(INT8U bgnd_color);
void PC_DispClrLine(INT8U y, INT8U bgnd_color);
void PC_DispChar(INT8U x, INT8U y, INT8U c, INT8U color);
void PC_DispStr(INT8U x, INT8U y, INT8U *s, INT8U color);

void PC_DOSReturn(void);
void PC_DOSSaveReturn(void);

void PC_ElapsedInit(void);
void PC_ElapsedStart(void);
INT16U PC_ElapsedStop(void);

void PC_GetDateTime(char *s);
BOOLEAN PC.GetKey(INT16S *c);

void PC_SetTickRate(INT16U freq);

void *PC_VectGet(INT8U vect);
void PC_VectSet(INT8U vect, void (*isr)(void));
```



附录 B

μC/OS-II 与处理器类型无关的源代码

B.0 uCOS_II.C

```
/*
*****  
*          uC/OS-II  
*          The Real-Time Kernel  
*  
*          (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL  
*          All Rights Reserved  
*  
*          V2.00  
*  
* File : uCOS_II.C  
* By  : Jean J. Labrosse  
*****  
*/  
  
#define OS_GLOBALS      /* Declare GLOBAL variables */  
#include "includes.h"  
  
#define OS_MASTER_FILE /* Prevent the following files from including includes.h*/  
#include "\software\ucos-ii\source\os_core.c"  
#include "\software\ucos-ii\source\os_mbox.c"  
#include "\software\ucos-ii\source\os_mem.c"  
#include "\software\ucos-ii\source\os_q.c"  
#include "\software\ucos-ii\source\os_sem.c"  
#include "\software\ucos-ii\source\os_task.c"  
#include "\software\ucos-ii\source\os_time.c"
```



B.1 uCOS_II.H

```

/*
***** uC/OS-II
* The Real-Time Kernel
*
* (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
* All Rights Reserved
*
* V2.00
*
* File : uCOS_II.H
* By   : Jean J. Labrosse
***** */

/*
***** MISCELLANEOUS
***** */

#define OS_VERSION      200 /* Version of uC/OS-II (Vx.yy multiplied by 100) */

#ifndef OS_GLOBALS
#define OS_EXT
#else
#define OS_EXT extern
#endif

#define OS_PRIO_SELF    0xFF /* Indicate SELF priority */

#if OS_TASK_STAT_EN
#define OS_N_SYS_TASKS     2      /* Number of system tasks */
#else
#define OS_N_SYS_TASKS     1
#endif

#define OS_STAT_PRIO      (OS_LOWEST_PRIO - 1) /* Statistic task priority */
#define OS_IDLE_PRIO       (OS_LOWEST_PRIO)      /* IDLE      task priority */

#define OS_EVENT_TBL_SIZE ((OS_LOWEST_PRIO) / 8 + 1) /* Size of event table */

```

```

#define OS_RDY_TBL_SIZE ((OS_LOWEST_PRIO) / 8 + 1) /* Size of ready table */

#define OS_TASK_IDLE_ID      65535 /* I.D. numbers for Idle and Stat tasks */
#define OS_TASK_STAT_ID      65534

/* TASK STATUS (Bit definition for OSTCBStat) */
#define OS_STAT_RDY          0x00 /* Ready to run */
#define OS_STAT_SEM          0x01 /* Pending on semaphore */
#define OS_STAT_MBOX         0x02 /* Pending on mailbox */
#define OS_STAT_Q             0x04 /* Pending on queue */
#define OS_STAT_SUSPEND      0x08 /* Task is suspended */

#define OS_EVENT_TYPE_MBOX   1
#define OS_EVENT_TYPE_Q       2
#define OS_EVENT_TYPE_SEM    3

/* TASK OPTIONS (see OSTaskCreateExt()) */
#define OS_TASK_OPT_STK_CHK  0x0001 /* Enable stack checking for the task */
#define OS_TASK_OPT_STK_CLR  0x0002 /* Clear the stack when the task is create */
#define OS_TASK_OPT_SAVE_FP   0x0004 /* Save the contents of any floating-point registers */

#ifndef FALSE
#define FALSE                0
#endif

#ifndef TRUE
#define TRUE                 1
#endif

/*
*****
*          ERROR CODES
*****
*/
#define OS_NO_ERR            0
#define OS_ERR_EVENT_TYPE    1
#define OS_ERR_PEND_ISR      2

#define OS_TIMEOUT           10
#define OS_TASK_NOT_EXIST    11

#define OS_MBOX_FULL         20

```

盗版必究
使用本复制品
请尊重相关知识产权!

```
#define OS_O_FULL          30
#define OS_PRIO_EXIST        40
#define OS_PRIO_ERR          41
#define OS_PRIO_INVALID       42
#define OS_SEM_OVF           50
#define OS_TASK_DEL_ERR       60
#define OS_TASK_DEL_IDLE      61
#define OS_TASK_DEL_REQ       62
#define OS_TASK_DEL_ISR       63
#define OS_NO_MORE_TCB        70
#define OS_TIME_NOT_DLY       80
#define OS_TIME_INVALID_MINUTES 81
#define OS_TIME_INVALID_SECONDS 82
#define OS_TIME_INVALID_MILLI   83
#define OS_TIME_ZERO_DLY       84
#define OS_TASK_SUSPEND_PRIO    90
#define OS_TASK_SUSPEND_IDLE     91
#define OS_TASK_RESUME_PRIO     100
#define OS_TASK_NOT_SUSPENDED   101
#define OS_MEM_INVALID_PART      110
#define OS_MEM_INVALID_BLKS       111
#define OS_MEM_INVALID_SIZE       112
#define OS_MEM_NO_FREE_BLKS       113
#define OS_MEM_FULL              114
#define OS_TASK_OPT_ERR          130

/*
***** EVENT CONTROL BLOCK *****
*                                EVENT CONTROL BLOCK
*****
*/
#if (OS_MAX_EVENTS >= 2)
typedef struct {
    void    *OSEventPlr;           /* Pointer to message or queue structure */

```

```

INT8U  OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur */
INT16U OSEventCnt;                  /* Count of used when event is a semaphore */
INT8U  OSEventType;                /* OS_EVENT_TYPE_MBOX, OS_EVENT_TYPE_Q or OS_EVENT_TYPE_SEM */
INT8U  OSEventGrp;                 /* Group corresponding to tasks waiting for event to occur */
} OS_EVENT;
#endif

/*
*****
*          MESSAGE MAILBOX DATA
*****
*/
#endif OS_MBOX_EN

typedef struct {
    void *OSMsg;                      /* Pointer to message in mailbox */
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur */
    INT8U OSEventGrp;                 /* Group corresponding to tasks waiting for event to occur */
} OS_MBOX_DATA;
#endif

/*
*****
*          MEMORY PARTITION DATA STRUCTURES
*****
*/
#endif OS_MEM_EN && (OS_MAX_MEM_PART >= 2)

typedef struct {                      /* MEMORY CONTROL BLOCK */
    void *OSMemAddr;                 /* Pointer to beginning of memory partition */
    void *OSMemFreeList;              /* Pointer to list of free memory blocks */
    INT32U OSMemBlkSize;              /* Size (in bytes) of each block of memory */
    INT32U OSMemNBlks;                /* Total number of blocks in this partition */
    INT32U OSMemNFree;                /* Number of memory blocks remaining in this partition */
} OS_MEM;

typedef struct {
    void *OSAddr;                    /* Pointer to the beginning address of the memory partition */
    void *OSFreeList;                /* Pointer to the beginning of the free list of memory blocks */
    INT32U OSBlkSize;                /* Size (in bytes) of each memory block */
    INT32U OSNBlks;                  /* Total number of blocks in the partition */
    INT32U OSNFree;                  /* Number of memory blocks free */
    INT32U OSNUUsed;                 /* Number of memory blocks used */
} OS_MEM_DATA;

```

盗版必究
使用本复制品
请尊重相关知识产权!

```

#endif

/*
***** MESSAGE QUEUE DATA *****
*/
#endif

#if OS_Q_EN
typedef struct {
    void *OSMsg; /* Pointer to next message to be extracted from queue */
    INT16U OSNMsgs; /* Number of messages in message queue */
    INT16U OSQSize; /* Size of message queue */
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur */
    INT8U OSEventGrp; /* Group corresponding to tasks waiting for event to occur */
} OS_Q_DATA;
#endif

/*
***** SEMAPHORE DATA *****
*/
#endif

#if OS_SEM_EN
typedef struct {
    INT16U OSCnt; /* Semaphore count */
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur */
    INT8U OSEventGrp; /* Group corresponding to tasks waiting for event to occur */
} OS_SEM_DATA;
#endif

/*
***** TASK STACK DATA *****
*/
#endif

#if OS_TASK_CREATE_EXT_EN
typedef struct {
    INT32U OSFree; /* Number of free bytes on the stack */
    INT32U OSUsed; /* Number of bytes used on the stack */
} OS_STK_DATA;
#endif

```

```

/*
***** TASK CONTROL BLOCK *****
*/
typedef struct os_tcb {
    OS_STK      *OSTCBStkPtr;      /* Pointer to current top of stack      */
    #if OS_TASK_CREATE_EXT_EN
        void      *OSTCBExtPtr;      /* Pointer to user definable data for TCB extension */
        OS_STK    *OSTCBStkBottom;   /* Pointer to bottom of stack          */
        TNT32U    OSTCBStkSize;     /* Size of task stack (in bytes)      */
        INT16U    OSTCBOpt;        /* Task options as passed by OSTaskCreateExt() */
        INT16U    OSTCBID;         /* Task ID (0..65535)                */
    #endif
    struct os_tcb *OSTCBNext;    /* Pointer to next      TCB in the TCB list      */
    struct os_tcb *OSTCBPrev;    /* Pointer to previous TCB in the TCB list      */
    #if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
        OS_EVENT   *OSTCBEEventPtr;   /* Pointer to event control block      */
    #endif
    #if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
        void      *OSTCBMsg;        /* Message received from OSMboxPost() or OSQPost() */
    #endif
    INT16U    OSTCBDly;        /* Nbr ticks to delay task or, timeout waiting for event */
    INT8U     OSTCBStat;        /* Task status                      */
    INT8U     OSTCBPrio;        /* Task priority (0 == highest, 63 == lowest) */
    TNT8U    OSTCBX;           /* Bit position in group corresponding to task priority (0..7) */
    INT8U    OSTCBY;           /* Index into ready table corresponding to task priority */
    INT8U    OSTCBBITX;        /* Bit mask to access bit position in ready table */
    INT8U    OSTCBBITY;        /* Bit mask to access bit position in ready group */
    #if OS_TASK_DEL_EN
        BOOLEAN   OSTCBDelReq;     /* Indicates whether a task needs to delete itself */
    #endif
} OS_TCB;

```

```

/*
*****GLOBAL VARIABLES*****
*/
OS_EXT INT32U      OSCtxSwCtr;      /* Counter of number of context switches */

#if (OS_MAX_EVENTS >= 2)
OS_EXT OS_EVENT    *OSEventFreeList;   /* Pointer to list of free EVENT control blocks */
OS_EXT OS_EVENT    OSEventTbl[OS_MAX_EVENTS]; /* Table of EVENT control blocks */
#endif

OS_EXT INT32U      OSIdleCtr;      /* Idle counter */

#if OS_TASK_STAT_EN
OS_EXT INT8S      OSCPUUsage;      /* Percentage of CPU used */
OS_EXT INT32U      OSIdleCtrMax;   /* Maximum value that idle counter can take in 1 sec. */
OS_EXT INT32U      OSIdleCtrRun;   /* Value reached by idle counter at run time in 1 sec. */
OS_EXT BOOLEAN    OSStatRdy;      /* Flag indicating that the statistic task is ready */
#endif

OS_EXT INT8U      OSIntNesting;    /* Interrupt nesting level */

OS_EXT INT8U      OSLockNesting;   /* Multitasking lock nesting level */

OS_EXT INT8U      OSPrioCur;      /* Priority of current task */
OS_EXT INT8U      OSPrioHighRdy;  /* Priority of highest priority task */

OS_EXT INT8U      OSRdyGrp;       /* Ready list group */
OS_EXT INT8U      OSRdyTbl[OS_RDY_TBL_SIZE]; /* Table of tasks which are ready to run */

OS_EXT BOOLEAN    OSRunning;      /* Flag indicating that kernel is running */

#if OS_TASK_CREATE_EN || OS_TASK_CREATE_EXT_EN || OS_TASK_DEL_EN
OS_EXT INT8U      OSTaskCtr;      /* Number of tasks created */
#endif

OS_EXT OS_TCB     *OSTCBCur;      /* Pointer to currently running TCB */
OS_EXT OS_TCB     *OSTCBFreeList;  /* Pointer to list of free TCBs */
OS_EXT OS_TCB     *OSTCBHighRdy;  /* Pointer to highest priority TCB ready to run */
OS_EXT OS_TCB     *OSTCBLList;    /* Pointer to doubly linked list of TCBs */
OS_EXT OS_TCB     *OSTCBPriorTbl[OS_LOWEST_PRIO + 1]; /* Table of pointers to created TCBs */

OS_EXT INT32U      OSTime;        /* Current value of system time (in ticks) */

```

```
extern INT8U const OSMapTbl[]; /* Priority->Bit Mask lookup table */
extern INT8U const OSUnMapTbl[]; /* Priority->Index lookup table */

/*
***** FUNCTION PROTOTYPES
* (Target Independant Functions)
*****
*/

/*
***** MESSAGE MAILBOX MANAGEMENT
*****
*/
#if OS_MBOX_EN
void *OSMboxAccept(OS_EVENT *pevent);
OS_EVENT *OSMboxCreate(void *msg);
void *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
INT8U OSMboxPost(OS_EVENT *pevent, void *msg);
INT8U OSMboxQuery(OS_EVENT *pevent, OS_MBOX_DATA *pdata);
#endif
/*
***** MEMORY MANAGEMENT
*****
*/
#if OS_MEM_EN && (OS_MAX_MEM_PART >= 2)
OS_MEM *OSMemCreate(void *addr, INT32U nblks, INT32U blksize, INT8U *err);
void *OSMemGet(OS_MEM *pmem, INT8U *err);
INT8U OSMemPut(OS_MEM *pmem, void *pblk);
INT8U OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata);
#endif
/*
***** MESSAGE QUEUE MANAGEMENT
*****
*/
#if OS_Q_EN && (OS_MAX_QS >= 2)
void *OSQAccept(OS_EVENT *pevent);
OS_EVENT *OSQCreate(void **start, INT16U size);
INT8U OSQFlush(OS_EVENT *pevent);
#endif
```

```

void      *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
INT8U    OSQPost(OS_EVENT *pevent, void *msg);
INT8U    OSQPostFront(OS_EVENT *pevent, void *msg);
INT8U    OSQuery(OS_EVENT *pevent, OS_Q_DATA *pdata);
#endif

/*
***** SEMAPHORE MANAGEMENT *****
*/
#if      OS_SEM_EN
INT16U  OSSemAccept(OS_EVENT *pevent);
OS_EVENT *OSSemCreate(INT16U value);
void    OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
INT8U    OSSemPost(OS_EVENT *pevent);
INT8U    OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata);
#endif

/*
***** TASK MANAGEMENT *****
*/
#if      OS_TASK_CHANGE_PRIO_EN
INT8U    OSTaskChangePrio(INT8U oldprio, INT8U newprio);
#endif

INT8U    OSTaskCreate(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio);

#if      OS_TASK_CREATE_EXT_EN
INT8U    OSTaskCreateExt(void (*task)(void *pd),
                      void    *pdata,
                      OS_STK *ptos,
                      INT8U   prio,
                      INT16U  id,
                      OS_STK *pbos,
                      INT32U  stk_size,
                      void    *pext,
                      INT16U  opt);
#endif

#if      OS_TASK_DEL_EN
INT8U    OSTaskDel(INT8U prio);
INT8U    OSTaskDelReq(INT8U prio);
#endif

```

```

#if      OS_TASK_SUSPEND_EN
INT8U   OSTaskResume(INT8U prio);
TNT8U   OSTaskSuspend(INT8U prio);
#endif

#if      OS_TASK_CREATE_EXT_EN
INT8U   OSTaskStkChk(INT8U prio, OS_STK_DATA *pdata);
#endif

INT8U   OSTaskQuery(INT8U prio, OS_TCB *pdata);

/*
***** TIME MANAGEMENT *****
*/
void    OSTimeDly(INT16U ticks);
INT8U   OSTimeDlyHMSM(INT8U hours, INT8U minutes, TNT8U seconds, INT16U milli);
INT8U   OSTimeDlyResume(INT8U prio);
INT32U  OSTimeGet(void);
void    OSTimeSet(INT32U ticks);
void    OSTimeTick(void);

/*
***** MISCELLANEOUS *****
*/
void    OSInit(void);

void    OSIntEnter(void);
void    OSIntExit(void);

void    OSSchedLock(void);
void    OSSchedUnlock(void);

void    OSStart(void);

void    OSStatInit(void);

INT16U  OSVersion(void);

```



```

/*
*****INTERNAL FUNCTION PROTOTYPES
*(Your application MUST NOT call these functions)
*****
*/
#endif

#if OS_MBOX_EN || OS_Q_EN || OS_SEM_EN
void OSEventTaskRdy(OS_EVENT *pevent, void *msg, INT8U msk);
void OSEventTaskWait(OS_EVENT *pevent);
void OSEventTO(OS_EVENT *pevent);
void OSEventWaitListInit(OS_EVENT *pevent);
#endif

#if OS_MEM_EN && (OS_MAX_MEM_PART >= 2)
void OSMemInit(void);
#endif

#if OS_Q_EN
void OSQInit(void);
#endif

void OSSched(void);

void OSTaskIdle(void *data);

#if OS_TASK_STAT_EN
void OSTaskStat(void *data);
#endif

INT8U OSTCBInit(INT8U prio, OS_STK *ptos, OS_STK *pbos, INT16U id, INT16U stk_size,
                 void *pext, INT16U opt);

/*
*****FUNCTION PROTOTYPES
*(Target Specific Functions)
*****
*/
void OSCtxSw(void);

void OSIntCtxSw(void);

```



```
void      OSStartHighRdy(void);

void      OSTaskCreateHook(OS_TCB *ptcb);
void      OSTaskDelHook(OS_TCB *ptcb);
void      OSTaskStatHook(void);
void      *OSTaskStkInit(void (*task)(void *pd), void *pdata, void *ptos, INT16U opt);
void      OSTaskSwHook(void);

void      OSTickISR(void);

void      OSTimeTickHook(void);
```



B.2 OS_CORE.C

```
/*
*****+
*          uC/OS-II
*          The Real-Time Kernel
*          CORE FUNCTIONS
*
*          (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*          All Rights Reserved
*
*          V2.00
*
* File : OS_CORE.C
* By   : Jean J. Labrosse
*****
*/
#ifndef OS_MASTER_FILE
#define OS_GLOBALS
#include "includes.h"
#endif

/*
*****+
*          LOCAL GLOBAL VARIABLES
*****
*/
static INT8U OSIntExitY; /* Variable used by 'OSIntExit' to prevent using locals */
```

附录 B μC/OS-II 与处理器类型无关的源代码

```
static OS_STK OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE];      /* Idle task stack */

#ifndef OS_TASK_STAT_EN
static OS_STK OSTaskStatStk[OS_TASK_STAT_STK_SIZE];      /* Statistics task stack */
#endif

static OS_TCB OSTCBTbl[OS_MAX_TASKS + OS_N_SYS_TASKS]; /* Table of TCBs */

/*
*****
*          MAPPING TABLE TO MAP BIT POSITION TO BIT MASK
*
* Note: Index into table is desired bit position, 0..7
*       Indexed value corresponds to bit mask
*****
*/
INT8U const OSMapTbl[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};

/*
*****
*          PRIORITY RESOLUTION TABLE
*
* Note: Index into table is bit pattern to resolve highest priority
*       Indexed value corresponds to highest priority bit position (i.e. 0..7)
*****
*/
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
};
```

```
/*
*****
*          MAKE TASK READY TO RUN BASED ON EVENT OCCURRING
*
* Description: This function is called by other uC/OS-II services and is used to ready a task
*               that was waiting for an event to occur.
*
* Arguments : pevent   is a pointer to the event control block corresponding to the event.
*
*             msg      is a pointer to a message. This pointer is used by message oriented services
*                     such as MAILBOXes and QUEUES. The pointer is not used when called by other
*                     service functions.
*
*             msk      is a mask that is used to clear the status byte of the TCB. For example,
*                     OSSemPost() will pass OS_STAT_SEM, OSMboxPost() will pass OS_STAT_MBOX etc.
*
* Returns   : none
*
* Note      : This function is INTERNAL to uC/OS-II and your application should not call it.
*****
*/
#ifndef OS_EVENT_TASK_RDY
#define OS_EVENT_TASK_RDY
#endif

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void OSEventTaskRdy (OS_EVENT *pevent, void *msg, INT8U msk)
{
    OS_TCB *ptcb;
    INT8U x;
    INT8U y;
    INT8U bitx;
    INT8U bity;
    INT8U prio;

    y = OSUnMapTbl[pevent->OSEventGrp]; /* Find highest prio. task waiting for message */
    bity = OSMapTbl[y];
    x = OSUnMapTbl[pevent->OSEventTbl[y]];
    bitx = OSMapTbl[x];
    prio = (INT8U)((y << 3) + x); /* Find priority of task getting the msg */
    if ((pevent->OSEventTbl[y] &= ~bitx) == 0) { /* Remove this task from the waiting list */
        pevent->OSEventGrp &= ~bity;
    }
    ptcb = OSTCBPrioTbl[prio]; /* Point to this task's OS_TCB */
}
```

附录B μC/OS-II 与处理器类型无关的源代码

```
ptcb->OSTCBDly = 0; /* Prevent OSTimeTick() from readying task */
ptcb->OSTCBEEventPtr = (OS_EVENT *)0; /* Unlink ECB from this task */
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    ptcb->OSTCBMsg = msg; /* Send message directly to waiting task */
#else
    msg = msg; /* Prevent compiler warning if not used */
#endif
    ptcb->OSTCBStat &= ~msk; /* Clear bit associated with event type */
    if (ptcb->OSTCBStat == OS_STAT_RDY) (/* See if task is ready (could be susp'd) */
        OSRdyGrp |= bity; /* Put task in the ready to run list */
        OSRdyTbl[y] |= bitx;
    )
}
#endif

/*
*****
*          MAKE TASK WAIT FOR EVENT TO OCCUR
*
* Description: This function is called by other uC/OS-II services to suspend a task because an
*               event has not occurred.
*
* Arguments : pevent is a pointer to the event control block for which the task will be waiting for.
*
* Returns   : none
*
* Note      : This function is INTERNAL to uC/OS-II and your application should not call it.
*****
*/
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void OSEventTaskWait (OS_EVENT *pevent)
{
    OSTCBCur->OSTCBEEventPtr = pevent; /* Store pointer to event control block in TCB */
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) (/* Task no longer ready */
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    )
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX; /* Put task in waiting list */
    pevent->OSEventGrp |= OSTCBCur->OSTCBBitY;
}
#endif

/*
*****

```

```

*          MAKE TASK READY TO RUN BASED ON EVENT TIMEOUT
*
* Description: This function is called by other uC/OS-II services to make a task ready to run
*               because a timeout occurred.
*
* Arguments : pevent   is a pointer to the event control block which is readying a task.
*
* Returns   : none
*
* Note      : This function is INTERNAL to uC/OS-II and your application should not call it.
*****
```

超星阅读器
使用本资源
请尊重相关知识产权！

```

*/
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void OSEventTO (OS_EVENT *pevent)
{
    if ((pevent->OSEventTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStat     = OS_STAT_RDY;           /* Set status to ready      */
    OSTCBCur->OSTCBEEventPtr = (OS_EVENT *)0;        /* No longer waiting for event */
}
#endif
```

```

/*
*****
```

```

*          INITIALIZE EVENT CONTROL BLOCK'S WAIT LIST
*
* Description: This function is called by other uC/OS-II services to initialize the event wait list.
*
* Arguments : pevent   is a pointer to the event control block allocated to the event.
*
* Returns   : none
*
* Note      : This function is INTERNAL to uC/OS-II and your application should not call it.
*****
```

```

*/
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void OSEventWaitListInit (OS_EVENT *pevent)
{
    INT8U i;

    pevent->OSEventGrp = 0x00;                      /* No task waiting on event */
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
```

```

    pevent->OSEventTbl[i] = 0x00;
}

#endif

/*
*****INITIALIZATION
*
* Description: This function is used to initialize the internals of uC/OS-II and MUST be called
*               prior to creating any uC/OS-II object and, prior to calling OSStart().
*
* Arguments : none
*
* Returns   : none
*****
*/
void OSInit (void)
{
    INT16U i;

    OSTime      = 0L;           /* Clear the 32-bit system clock */
    OSIntNesting = 0;           /* Clear the interrupt nesting counter */
    OSLockNesting = 0;          /* Clear the scheduling lock counter */
#ifndef OS_TASK_CREATE_EN || OS_TASK_CREATE_EXT_EN || OS_TASK_DEL_EN
    OSTaskCtr     = 0;          /* Clear the number of tasks */
#endif
    OSRunning     = FALSE;       /* Indicate that multitasking not started */
    OSIdleCtr     = 0L;          /* Clear the 32-bit idle counter */
#ifndef OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_EN
    OSIdleCtrRun = 0L;
    OSIdleCtrMax = 0L;
    OSStatRdy    = FALSE;       /* Statistic task is not ready */
#endif
    OSCtxSwCtr    = 0;           /* Clear the context switch counter */
    OSRdyGrp     = 0;           /* Clear the ready list */
    for (i = 0; i < OS_RDY_TBL_SIZE; i++) {
        OSRdyTbl[i] = 0;
    }
    OSPrioCur     = 0;
    OSPrioHighRdy = 0;
    OSTCBHighRdy = (OS_TCB *)0;           /* TCB Initialization */
    OSTCBCur     = (OS_TCB *)0;
    OSTCBL.list  = (OS_TCB *)0;
}

```



```

for (i = 0; i < (OS_LOWEST_PRIO + 1); i++) { /* Clear the priority table */
    OSTCBPrioTbl[i] = (OS_TCB *)0;
}
for (i = 0; i < (OS_MAX_TASKS + OS_N_SYS_TASKS - 1); i++) { /* Init. list of free TCBs */
    OSTCBTbl[i].OSTCBNext = &OSTCBTbl[i + 1];
}
OSTCBTbl[OS_MAX_TASKS + OS_N_SYS_TASKS - 1].OSTCBNext = (OS_TCB *)0; /* Last OS_TCB */
OSTCBFreeList = &OSTCBTbl[0];

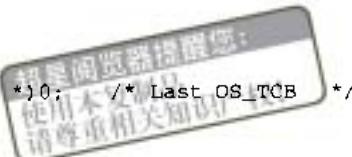
#if OS_MAX_EVENTS >= 2
    for (i = 0; i < (OS_MAX_EVENTS - 1); i++) { /* Init. list of free EVENT control blocks */
        OSEventTbl[i].OSEventPtr = (OS_EVENT *)&OSEventTbl[i + 1];
    }
    OSEventTbl[OS_MAX_EVENTS - 1].OSEventPtr = (OS_EVENT *)0;
    OSEventFreeList = &OSEventTbl[0];
#endif

#if OS_Q_EN && (OS_MAX_QS >= 2)
    OSQInit(); /* Initialize the message queue structures */
#endif

#if OS_MEM_EN && OS_MAX_MEM_PART >= 2
    OSMemInit(); /* Initialize the memory manager */
#endif

#if OS_STK_GROWTH == 1
    #if OS_TASK_CREATE_EXT_EN
        OSTaskCreateExt(OSTaskIdle,
                        (void *)0, /* No arguments passed to OSTaskIdle() */
                        &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE-1], /* Set Top-Of-Stack */
                        OS_IDLE_PRIO, /* Lowest priority level */
                        OS_TASK_IDLE_ID,
                        &OSTaskIdleStk[0], /* Set Bottom-Of-Stack */
                        OS_TASK_IDLE_STK_SIZE,
                        (void *)0, /* No TCB extension */
                        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR); /* Enable stack checking + clear
stack */
    #else
        OSTaskCreate(OSTaskIdle, (void *)0, &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE - 1],
                    OS_IDLE_PRIO);
    #endif
    #else
        #if OS_TASK_CREATE_EXT_EN
            OSTaskCreateExt(OSTaskIdle,
                            (void *)0, /* No arguments passed to OSTaskIdle() */

```



```

    &OSTaskIdleStk[0],           /* Set Top-Of-Stack          */
    OS_IDLE_PRIO,               /* Lowest priority level    */
    OS_TASK_IDLE_ID,
    &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE - 1], /* Set Bottom-Of-Stack   */
    OS_TASK_IDLE_STK_SIZE,
    (void *)0,                  /* No TCB extension        */
    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR); /* Enable stack checking + clear
stack */

#endif
#else
OSTaskCreate(OSTaskIdle, (void *)0, &OSTaskIdleStk[0], OS_IDLE_PRIO);
#endif
#endif

#if OS_TASK_STAT_EN
#if OS_TASK_CREATE_EXT_EN
#if OS_STK_GROWTH == 1
OSTaskCreateExt(OSTaskStat,
    (void *)0,             /* No args passed to OSTaskStat()      */
    &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1], /* Set Top-Of-Stack   */
    OS_STAT_PRIO,           /* One higher than the idle task     */
    OS_TASK_STAT_ID,
    &OSTaskStatStk[0],       /* Set Bottom-Of-Stack      */
    OS_TASK_STAT_STK_SIZE,
    (void *)0,              /* No TCB extension        */
    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR); /* Enable stack checking + clear */

#else
OSTaskCreateExt(OSTaskStat,
    (void *)0,             /* No args passed to OSTaskStat()      */
    &OSTaskStatStk[0],       /* Set Top-Of-Stack      */
    OS_STAT_PRIO,           /* One higher than the idle task     */
    OS_TASK_STAT_ID,
    &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1], /* Set Bottom-Of-Stack   */
    OS_TASK_STAT_STK_SIZE,
    (void *)0,              /* No TCB extension        */
    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR); /* Enable stack checking + clear */

#endif
#endif
#else
#if OS_STK_GROWTH == 1
OSTaskCreate(OSTaskStat,
    (void *)0,             /* No args passed to OSTaskStat()      */
    &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1], /* Set Top-Of-Stack   */
    OS_STAT_PRIO);           /* One higher than the idle task     */
#else
OSTaskCreate(OSTaskStat,
    (void *)0,             /* No args passed to OSTaskStat()      */

```

```

        &OSTaskStatStk[0],      /* Set Top-Of-Stack           */
        OS_STAT_PRIO);         /* One higher than the idle task   */

#endif
#endif
}

/*
*****
*          ENTER ISR
*
* Description: This function is used to notify uC/OS-JT that you are about to service an interrupt
*               service routine (ISR). This allows uC/OS-II to keep track of interrupt nesting and
*               thus only perform rescheduling at the last nested ISR.
*
* Arguments : none
*
* Returns   : none
*
* Notes     : 1) Your ISR can directly increment OSIntNesting without calling this function because
*               OSIntNesting has been declared 'global'. You MUST, however, be sure that the
*               increment is performed 'indivisibly' by your processor to ensure proper access
*               to this critical resource.
*               2) You MUST still call OSIntExit() even though you increment OSIntNesting directly.
*               3) You MUST invoke OSIntEnter() and OSIntExit() in pair. In other words, for every
*                  call to OSIntEnter() at the beginning of the ISR you MUST have a call to OSIntExit()
*                  at the end of the ISR.
*****
*/
void OSIntEnter (void)
{
    OS_ENTER_CRITICAL();
    OSintNesting++;           /* Increment ISR nesting level           */
    OS_EXIT_CRITICAL();
}

/*
*****
*          EXIT ISR
*
* Description: This function is used to notify uC/OS-II that you have completed serviving an ISR.
*               When the last nested ISR has completed, uC/OS-II will call the scheduler to determine
*               whether a new, high-priority task, is ready to run.
*/

```





```

/*
 * Arguments : none
 *
 * Returns   : none
 *
 * Notes      : 1) You MUST invoke OSIntEnter() and OSIntExit() in pair. In other words, for every
 *                  call to OSIntEnter() at the beginning of the ISR you MUST have a call to OSIntExit()
 *                  at the end of the ISR.
 *                  2) Rescheduling is prevented when the scheduler is locked (see OSSchedLock())
 ****
 */

void OSIntExit (void)
{
    OS_ENTER_CRITICAL();
    if ((--OSIntNesting + OSLockNesting) == 0) { /* Reschedule only if all ISRs completed & not
locked */
        OSIntExitY = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((OSIntExitY << 3) + OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
        if (OSPrioHighRdy != OSPrioCur) { /* No context switch if current task is highest ready */

            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++; /* Keep track of the number of context switches */
            OSIntCtxSw(); /* Perform interrupt level context switch */
        }
    }
    OS_EXIT_CRITICAL();
}

/*
 ****
 *
 *          SCHEDULER
 *
 * Description: This function is called by other uC/OS-II services to determine whether a new,
 *              high priority task has been made ready to run. This function is invoked by TASK level
 *              code and is not used to reschedule tasks from ISRs (see OSIntExit() for ISR rescheduling).
 *
 * Arguments : none
 *
 * Returns   : none
 *
 * Notes      : 1) This function is INTERNAL to uC/OS-II and your application should not call it.
 *                  2) Rescheduling is prevented when the scheduler is locked (see OSSchedLock())
 ****

```



```

/*
void OSSched (void)
{
    INT8U y;

    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) { /* Task scheduling must be enabled and not ISR level */
        y          = OSUnMapTbl[OSRdyGrp]; /* Get pointer to highest priority task ready to run */
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) { /* No context switch if current task is highest ready */
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCTxSwCtr++; /* Increment context switch counter */
            OS_TASK_SW(); /* Perform a context switch */
        }
    }
    OS_EXIT_CRITICAL();
}

/*
***** PREVENT SCHEDULING *****
*
* Description: This function is used to prevent rescheduling to take place. This allows your
*               application to prevent context switches until you are ready to permit context switching.
*
* Arguments : none
*
* Returns   : none
*
* Notes     : 1) You MUST invoke OSSchedLock() and OSSchedUnlock() in pair. In other words, for
*               every call to OSSchedLock() you MUST have a call to OSSchedUnlock().
***** */

void OSSchedLock (void)
{
    if (OSRunning == TRUE) { /* Make sure multitasking is running */
        OS_ENTER_CRITICAL();
        OSLockNesting++; /* Increment lock nesting level */
        OS_EXIT_CRITICAL();
    }
}

```

```

/*
***** ENABLE SCHEDULING *****
* Description: This function is used to re-allow rescheduling.
*
* Arguments : none
*
* Returns   : none
*
* Notes      : 1) You MUST invoke OSSchedLock() and OSSchedUnlock() in pair. In other words, for
*               every call to OSSchedLock() you MUST have a call to OSSchedUnlock().
***** */

void OSSchedUnlock (void)
{
    if (OSRunning == TRUE) { /* Make sure multitasking is running */
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) { /* Do not decrement if already 0 */
            OSLockNesting--;
            /* Decrement lock nesting level */
            if ((OSLockNesting | OSIntNesting) == 0) { /* See if scheduling re-enabled and not an ISR */
                OS_EXIT_CRITICAL();
                OSSched(); /* See if a higher priority task is ready */
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}

/*
***** START MULTITASKING *****
* Description: This function is used to start the multitasking process which lets uC/OS-II manages
*               the task that you have created. Before you can call OSStart(), you MUST have called
*               OSInit() and you MUST have created at least one task.
*
* Arguments : none
*

```



```
* Returns : none
*
* Note : OSStartHighRdy() MUST:
*         a) Call OSTaskSwHook() then,
*         b) Set OSRunning to TRUE.
*****
```

```
/*
INT8U y;
INT8U x;

if (OSRunning == FALSE) {
    y      = OSUnMapTbl[OSRdyGrp]; /* Find highest priority's task priority number */
    x      = OSUnMapTbl[OSRdyTbl[y]];
    OSPrioHighRdy = (INT8U)((y << 3) + x);
    OSPrioCur     = OSPrioHighRdy;
    OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy]; /* Point to highest priority task ready to run */
    OSTCBCur     = OSTCBHighRdy;
    OSStartHighRdy(); /* Execute target specific code to start task */
}
}

*/
*****
```

```
*                      STATISTICS INITIALIZATION
*
* Description: This function is called by your application to establish CPU usage by first
*               determining how high a 32-bit counter would count to in 1 second if no other tasks
*               were to execute during that time. CPU usage is then determined by a low priority task
*               which keeps track of this 32-bit counter every second but this time, with other tasks
*               running. CPU usage is determined by:
*
*               OSIdleCtr
*               CPU Usage (%) = 100 * (1 - -----)
*                               OSIdleCtrMax
*
* Arguments : none
*
* Returns : none
*****
```

```
*/
```

```

#if OS_TASK_STAT_EN
void OSStatInit (void)
{
    OSTimeDly(2);                                /* Synchronize with clock tick */
    OS_ENTER_CRITICAL();
    OSIdleCtr = 0L;                               /* Clear idle counter */
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC);                 /* Determine MAX.idle counter value for 1 second */
    OS_ENTER_CRITICAL();
    OSIdleCtrMax = OSIdleCtr;                    /* Store maximum idle counter count in 1 second */
    OSStatRdy = TRUE;
    OS_EXIT_CRITICAL();
}
#endif

/*
***** IDLE TASK *****
*
* Description: This task is internal to uC/OS-II and executes whenever no other higher priority
* tasks executes because they are waiting for event(s) to occur.
*
* Arguments : none
*
* Returns   : none
***** */

void OSTaskIdle (void *pdata)
{
    pdata = pdata;                                /* Prevent compiler warning for not using 'pdata' */
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtr++;
        OS_EXIT_CRITICAL();
    }
}

/*
***** STATISTICS TASK *****
*
* Description: This task is internal to uC/OS-II and is used to compute some statistics about

```

```
*      the multitasking environment. Specifically, OSTaskStat() computes the CPU usage.
*      CPU usage is determined by:
*
*          OSIdleCtr
*      OSCPUUsage = 100 * (1 - -----)    (units are in %)
*                      OSIdleCtrMax
*
* Arguments : pdata      this pointer is not used at this time.
*
* Returns   : none
*
* Notes     : 1) This task runs at a priority level higher than the idle task. In fact, it runs at
*             the next higher priority, OS_IDLE_PRIO-1.
*             2) You can disable this task by setting the configuration #define OS_TASK_STAT_EN to 0.
*             3) We delay for 5 seconds in the beginning to allow the system to reach steady state and
*                 have all other tasks created before we do statistics. You MUST have at least a delay
*                 of 2 seconds to allow for the system to establish the maximum value for the idle
*                 counter.
***** */
*/
#endif OS_TASK_STAT_EN
void OSTaskStat (void *pdata)
{
    INT32U run;
    INT8S usage;

    pdata = pdata; /* Prevent compiler warning for not using 'pdata' */
    while (OSStatRdy == FALSE) {
        OSTimeDly(2 * OS_TICKS_PER_SEC); /* Wait until statistic task is ready */
    }
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtrRun = OSIdleCtr; /* Obtain the of the idle counter for the past second */
        run = OSIdleCtr;
        OSIdleCtr = 0L; /* Reset the idle counter for the next second */
        OS_EXIT_CRITICAL();
        if (OSIdleCtrMax > 0L) {
            usage = (INT8S)(100L - 100L * run / OSIdleCtrMax);
            if (usage > 100) {
                OSCPUUsage = 100;
            } else if (usage < 0) {
                OSCPUUsage = 0;
            } else {

```

```

        OSCPUUsage = usage;
    }
} else {
    OSCPUUsage = 0;
}
OSTaskStatHook();           /* Invoke user definable hook          */
OSTimeDly(OS_TICKS_PER_SEC);/* Accumulate OSIdleCtr for the next second*/
}

}

#endif

/*
*****
*          INITIALIZE TCB
*
* Description: This function is internal to uC/OS-II and is used to initialize a Task Control
*               Block when a task is created (see OSTaskCreate() and OSTaskCreateExt()).
*
* Arguments : prio      is the priority of the task being created
*
*             ptos      is a pointer to the task's top-of-stack assuming that the CPU registers
*                         have been placed on the stack. Note that the top-of-stack corresponds to a
*                         'high' memory location if OS_STK_GROWTH is set to 1 and a 'low' memory
*                         location if OS_STK_GROWTH is set to 0. Note that stack growth is CPU
*                         specific.
*
*             pbos      is a pointer to the bottom of stack. A NULL pointer is passed if called by
*                         'OSTaskCreate()' .
*
*             id       is the task's ID (0..65535)
*
*             stk_size  is the size of the stack (in 'stack units'). If the stack units are INT8Us
*                         then, 'stk_size' contains the number of bytes for the stack. If the stack
*                         units are INT32Us then, the stack contains '4 * stk_size' bytes. The stack
*                         units are established by the #define constant OS_STK which is CPU
*                         specific. 'stk_size' is 0 if called by 'OSTaskCreate()' .
*
*             pext      is a pointer to a user supplied memory area that is used to extend the task
*                         control block. This allows you to store the contents of floating-point
*                         registers, MMU registers or anything else you could find useful during a
*                         context switch. You can even assign a name to each task and store this name
*                         in this TCB extension. A NULL pointer is passed if called by OSTaskCreate().
*
*             opt       options as passed to 'OSTaskCreateExt()' or,

```

```

*                               0 if called from 'OSTaskCreate()'.

*
* Returns     : OS_NO_ERR      if the call was successful
*               OS_NO_MORE_TCB   if there are no more free TCBs to be allocated and thus, the task
*                           cannot be created.
*
* Note        : This function is INTERNAL to uC/OS-II and your application should not call it.
*****  

* /  

TNT8U OSTCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos, INT16U id, INT16U stk_size,
                  void *pext, INT16U opt)
{
    OS_TCB *ptcb;

    OS_ENTER_CRITICAL();
    ptcb = OSTCBFreeList;           /* Get a free TCB from the free TCB list */
    if (ptcb != (OS_TCB *)0) {
        OSTCBFreeList = ptcb->OSTCBNext; /* Update pointer to free TCB list */
        OS_EXIT_CRITICAL();
        ptcb->OSTCBStkPtr = ptos;      /* Load Stack pointer in TCB */
        ptcb->OSTCBPrio = (INT8U)prio; /* Load task priority into TCB */
        ptcb->OSTCBStat = OS_STAT_RDY; /* Task is ready to run */
        ptcb->OSTCBLdy = 0;           /* Task is not delayed */
    }
    #if OS_TASK_CREATE_EXT_EN
        ptcb->OSTCBExt.Ptr = pext;    /* Store pointer to TCB extension */
        ptcb->OSTCBStkSize = stk_size; /* Store stack size */
        ptcb->OSTCBStkBottom = pbos; /* Store pointer to bottom of stack */
        ptcb->OSTCBOpt = opt;        /* Store task options */
        ptcb->OSTCBId = id;         /* Store task ID */
    #else
        pext = pext;                /* Prevent compiler warning if not used */
        stk_size = stk_size;
        pbos = pbos;
        opt = opt;
        id = id;
    #endif
    #if OS_TASK_DEL_EN
        ptcb->OSTCBLdy = OS_NO_ERR;
    #endif
    ptcb->OSTCBY = prio >> 3; /* Pre-compute X, Y, BitX and BitY */
}

```

```

ptcb->OSTCBRbitY = OSMapTbl[ptcb->OSTCBY];
ptcb->OSTCBX     = prio & 0x07;
ptcb->OSTCBBitX   = OSMapTbl[ptcb->OSTCBX];

#if OS_MBOX_EN || (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_SEM_EN
    ptcb->OSTCBEEventPtr = (OS_EVENT *)0; /* Task is not pending on an event */
#endif

#if OS_MBOX_EN || (OS_Q_EN && (OS_MAX_QS >= 2))
    ptcb->OSTCBMsg     = (void *)0;      /* No message received */ */
#endif

OS_ENTER_CRITICAL();
OSTCBPrioTbl[prio] = ptcb;
ptcb->OSTCBNext   = OSTCBLList; /* Link into TCB chain */ *
ptcb->OSTCBPrev   = (OS_TCB *)0;
if (OSTCBLList != (OS_TCB *)0) {
    OSTCBLList->OSTCBPrev = ptcb;
}
OSTCBLList          = ptcb;
OSRdyGrp           |= ptcb->OSTCBBitY; /* Make task ready to run */
OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
OS_EXIT_CRITICAL();
return (OS_NO_ERR);
} else {
    OS_EXIT_CRITICAL();
    return (OS_NO_MORE_TCB);
}
}

/*
*****
*          PROCESS SYSTEM TICK
*
* Description: This function is used to signal to uC/OS-II the occurrence of a 'system tick' (also
*               known as a 'clock tick'). This function should be called by the ticker ISR but, can
*               also be called by a high priority task.
*
* Arguments : none
*
* Returns   : none
*****
*/

```

```

void OSTimeTick (void)
{
    OS_TCB *ptcb;

    OSTimeTickHook(); /* Call user definable hook */
    ptcb = OSTCBList; /* Point at first TCB in TCB list */
    while (ptcb->OSTCBPrio != OS_IDLE_PRIO) { /* Go through all TCBs in TCB list */
        OS_ENTER_CRITICAL();
        if (ptcb->OSTCBDly != 0) { /* Delayed or waiting for event with TO */
            if (--ptcb->OSTCBDly == 0) { /* Decrement nbr of ticks to end of delay */
                if (!(ptcb->OSTCStat & OS_STAT_SUSPEND)) { /* Is task suspended? */
                    OSRdyGrp |= ptcb->OSTCBBitY; /* No, Make task Rdy to Run (timed out) */
                    OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                } else { /* Yes, Leave 1 tick to prevent ... */
                    ptcb->OSTCBDly = 1; /* ... loosing the task when the ... */
                }
            }
        }
        ptcb = ptcb->OSTCBNext; /* Point at next TCB in TCB list */
        OS_EXIT_CRITICAL();
    }

    OS_ENTER_CRITICAL(); /* Update the 32-bit tick counter */
    OSTime++;
    OS_EXIT_CRITICAL();
}

/*
*****
*          GET VERSION
*
* Description: This function is used to return the version number of uC/OS-II. The returned value
* corresponds to uC/OS-II's version number multiplied by 100. In other words, version
* 2.00 would be returned as 200.
*
* Arguments : none
*
* Returns   : the version number of uC/OS-II multiplied by 100.
*****
*/
INT16U OSVersion (void)
{
    return (OS_VERSION);
}

```

B.3 OS_MBOX.C

```

/*
***** uC/OS-II *****

* The Real-Time Kernel
* MESSAGE MAILBOX MANAGEMENT

*
* (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
* All Rights Reserved

*
* V2.00

*
* File : OS_MBOX.C
* By : Jean J. Labrosse
***** */

#ifndef OS_MASTER_FILE
#include "includes.h"
#endif

#if OS_MBOX_EN
/*
***** ACCEPT MESSAGE FROM MAILBOX

*
* Description: This function checks the mailbox to see if a message is available. Unlike OSMboxPending(),
* OSMboxAccept() does not suspend the calling task if a message is not available.
*
* Arguments : pevent      is a pointer to the event control block
*
* Returns   : != (void *)0  is the message in the mailbox if one is available. The mailbox is cleared
*              so the next time OSMboxAccept() is called, the mailbox will be empty.
*              == (void *)0  if the mailbox is empty or if you didn't pass the proper event pointer.
***** */

void *OSMboxAccept (OS_EVENT *pevent)
{
    void *msg;

```





```

OS_ENTER_CRITICAL();
if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) { /* Validate event block type */
    OS_EXIT_CRITICAL();
    return ((void *)0);
}

msg = pevent->OSEventPtr;
if (msg != (void *)0) { /* See if there is already a message */
    pevent->OSEventPtr = (void *)0; /* Clear the mailbox */
}
OS_EXIT_CRITICAL();
return (msg); /* Return the message received (or NULL) */
}

/*
*****
*          CREATE A MESSAGE MAILBOX
*
* Description: This function creates a message mailbox if free event control blocks are available.
*
* Arguments : msg      is a pointer to a message that you wish to deposit in the mailbox. If
*              you set this value to the NULL pointer (i.e. (void *)0) then the mailbox
*              will be considered empty.
*
* Returns   : != (void *)0 is a pointer to the event control block (OS_EVENT) associated with
*              the created mailbox
*              == (void *)0 if no event control blocks were available
*****
*/
OS_EVENT *OSMboxCreate (void *msg)
{
    OS_EVENT *pevent;

    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList; /* Get next free event control block */
    if (OSEventFreeList != (OS_EVENT *)0) { /* See if pool of free ECB pool was empty */
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {
        pevent->OSEventType = OS_EVENT_TYPE_MBOX;
        pevent->OSEventPtr = msg; /* Deposit message in event control block */
        OSEventWaitListInit(pevent);
    }
}

```

```

}

return (pevent);           /* Return pointer to event control block */ }

/*
***** PEND ON MAILBOX FOR A MESSAGE *****
*
* Description: This function waits for a message to be sent to a mailbox
*
* Arguments : pevent   is a pointer to the event control block associated with the desired mailbox
*
*             timeout   is an optional timeout period (in clock ticks). If non-zero, your task will
*                         wait for a message to arrive at the mailbox up to the amount of time
*                         specified by this argument. If you specify 0, however, your task will wait
*                         forever at the specified mailbox or, until a message arrives.
*
*             err       is a pointer to where an error message will be deposited. Possible error
*                         messages are:
*
*                         OS_NO_ERR      The call was successful and your task received a message.
*                         OS_TIMEOUT     A message was not received within the specified timeout
*                         OS_ERR_EVENT_TYPE Invalid event type
*                         OS_ERR_PEND_ISR If you called this function from an ISR and the result
*                                         would lead to a suspension.
*
* Returns   : != (void *)0 is a pointer to the message received
*             == (void *)0 if no message was received or you didn't pass the proper pointer to the
*                         event control block.
***** */

void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    void *msg;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) { /* Validate event block type */
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
    msg = pevent->OSEventPtr;
}

```

```

if (msg != (void *)0) {           /* See if there is already a message      */
    pevent->OSEventPtr = (void *)0;      /* Clear the mailbox                      */
    OS_EXIT_CRITICAL();                /*                                         */
    *err = OS_NO_ERR;
} else if (OSIntNesting > 0) {       /* See if called from ISR ...          */
    OS_EXIT_CRITICAL();                /* ... can't PEND from an ISR           */
    *err = OS_ERR_PEND_ISR;
} else {
    OSTCBCur->OSTCBStat |= OS_STAT_MBOX;      /* Message not available, task will pend */
    OSTCBCur->OSTCBDly = timeout;             /* Load timeout in TCB                   */
    OSEventTaskWait(pevent); /* Suspend task until event or timeout occurs */
    OS_EXIT_CRITICAL();
    OSSched();           /* Find next highest priority task ready to run */
    OS_ENTER_CRITICAL();
    if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) { /* See if we were given the message */
        OSTCBCur->OSTCBMsg = (void *)0; /* Yes, clear message received */
        OSTCBCur->OSTCBStat = OS_STAT_RDY;
        OSTCBCur->OSTCBEVENTPTR = (OS_EVENT *)0; /* No longer waiting for event */
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSTCBCur->OSTCBStat & OS_STAT_MBOX) { /* If status is not OS_STAT_RDY, timed out */
        OSEventTO(pevent);           /* Make task ready                         */
        OS_EXIT_CRITICAL();
        msg = (void *)0;           /* Set message contents to NULL           */
        *err = OS_TIMEOUT;         /* Indicate that a timeout occurred */
    } else {
        msg = pevent->OSEventPtr; /* Message received                      */
        pevent->OSEventPtr = (void *)0; /* Clear the mailbox                     */
        OSTCBCur->OSTCBEVENTPTR = (OS_EVENT *)0;
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    }
}
return (msg);           /* Return the message received (or NULL) */
}

/*
*****
*               POST MESSAGE TO A MAILBOX
*
* Description: This function sends a message to a mailbox
*
* Arguments : pevent is a pointer to the event control block associated with the desired mailbox
*

```

附录B μC/OS-II 与处理器类型无关的源代码

```
*           msg      is a pointer to the message to send. You MUST NOT send a NULL pointer.  
*  
* Returns   : OS_NO_ERR      The call was successful and the message was sent  
*           OS_MBOX_FULL    If the mailbox already contains a message. You can can only send one  
*                           message at a time and thus, the message MUST be consumed before you are  
*                           allowed to send another one.  
*           OS_ERR_EVENT_TYPE If you are attempting to post to a non mailbox.  
*****  
*/  
  
INT8U OSMboxPost (OS_EVENT *pevent, void *msg)  
{  
    OS_ENTER_CRITICAL();  
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) /* Validate event block type */  
        OS_EXIT_CRITICAL();  
        return (OS_ERR_EVENT_TYPE);  
    }  
    if (pevent->OSEventGrp) { /* See if any task pending on mailbox */  
        OSEventTaskRdy(pevent, msg, OS_STAT_MBOX); /* Ready highest priority task waiting on event */  
        OS_EXIT_CRITICAL();  
        OSSched(); /* Find highest priority task ready to run */  
        return (OS_NO_ERR);  
    } else {  
        if (pevent->OSEventPtr != (void *)0) { /* Make sure mailbox doesn't already have a msg */  
            OS_EXIT_CRITICAL();  
            return (OS_MBOX_FULL);  
        } else {  
            pevent->OSEventPtr = msg; /* Place message in mailbox */  
            OS_EXIT_CRITICAL();  
            return (OS_NO_ERR);  
        }  
    }  
}  
  
/*  
*****  
*          QUERY A MESSAGE MAILBOX  
*  
* Description: This function obtains information about a message mailbox.  
*  
* Arguments : pevent      is a pointer to the event control block associated with the desired mailbox  
*  
*           pdata       is a pointer to a structure that will contain information about the message  
*                         mailbox.  
*/
```



```

/*
 * Returns    : OS_NO_ERR           The call was successful and the message was sent
 *              OS_ERR_EVENT_TYPE If you are attempting to obtain data from a non mailbox.
 ****
 */
INT8U OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata)
{
    INT8U i;
    INT8U *psrc;
    INT8U *pdest;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {      /* Validate event block type */
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp; /* Copy message mailbox wait list */
    psrc      = &pevent->OSEventTbl[0];
    pdest     = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdest++ = *psrc++;
    }
    pdata->OSMsg = pevent->OSEventPtr;      /* Get message from mailbox */
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
#endif

```

B.4 OS_MEM.C

```

/*
 ****
 *
 *                      uC/OS-II
 *                      The Real-Time Kernel
 *                      MEMORY MANAGEMENT
 *
 *          (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
 *                      All Rights Reserved
 *
 *                      V2.00
 *

```



```

* File : OS_MEM.C
* By   : Jean J. Labrosse
***** ****
*/
#ifndef OS_MASTER_FILE
#include "includes.h"
#endif

#if OS_MEM_EN && OS_MAX_MEM_PART >= 2
/*
***** ****
*
*           LOCAL GLOBAL VARIABLES
*****
*/
static OS_MEM      *OSMemFreeList;          /* Pointer to free list of memory partitions */
static OS_MEM      OSMemTbl[OS_MAX_MEM_PART]; /* Storage for memory partition manager */

/*
***** ****
*
*           CREATE A MEMORY PARTITION
*
* Description : Create a fixed-sized memory partition that will be managed by uC/OS-II.
* Arguments   : addr     is the starting address of the memory partition
*
*                 nblk     is the number of memory blocks to create from the partition.
*
*                 blksize  is the size (in bytes) of each block in the memory partition.
*
*                 err      is a pointer to a variable containing an error message which will be set by
*                           this function to either:
*                               OS_NO_ERR      if the memory partition has been created correctly.
*                               OS_MEM_INVALID_PART no free partitions available
*                               OS_MEM_INVALID_BLK user specified an invalid number of blocks (must be >= 2)
*                               OS_MEM_INVALID_SIZE user specified an invalid block size
*                                         (must be greater than the size of a pointer)
* Returns    : != (OS_MEM *)0 is the partition was created
*                 == (OS_MEM *)0 if the partition was not created because of invalid arguments or, no
*                           free partition is available.
*****
*/
OS_MEM *OSMemCreate (void *addr, INT32U nblk, INT32U blksize, INT8U *err)
{

```



```

OS_MEM *pmem;
INT8U *pblk;
void **plink;
INT32U i;

if (nblks < 2) { /* Must have at least 2 blocks per partition */
    *err = OS_MEM_INVALID_BLKS;
    return ((OS_MEM *)0);
}

if (blksize < sizeof(void *)) { /* Must contain space for at least a pointer */
    *err = OS_MEM_INVALID_SIZE;
    return ((OS_MEM *)0);
}

OS_ENTER_CRITICAL();
pmem = OSMemFreeList; /* Get next free memory partition */
if (OSMemFreeList != (OS_MEM *)0) { /* See if pool of free partitions was empty */
    OSMemFreeList = (OS_MEM *)OSMemFreeList->OSMemFreeList;
}
OS_EXIT_CRITICAL();
if (pmem == (OS_MEM *)0) { /* See if we have a memory partition */
    *err = OS_MEM_INVALID_PART;
    return ((OS_MEM *)0);
}

plink = (void **)addr; /* Create linked list of free memory blocks */
pblk = (INT8U *)addr + blksize;
for (i = 0; i < (nblks - 1); i++) {
    *plink = (void *)pblk;
    plink = (void **)pblk;
    pblk = pblk + blksize;
}
*plink = (void *)0; /* Last memory block points to NULL */
OS_ENTER_CRITICAL();
pmem->OSMemAddr = addr; /* Store start address of memory partition */
pmem->OSMemFreeList = addr; /* Initialize pointer to pool of free blocks */
pmem->OSMemNFree = nblks; /* Store number of free blocks in MCB */
pmem->OSMemNBElks = nblks;
pmem->OSMemBlkSize = blksize; /* Store block size of each memory blocks */
OS_EXIT_CRITICAL();
*err = OS_NO_ERR;
return (pmem);
}

/*
***** GET A MEMORY BLOCK

```



```

/*
 * Description : Get a memory block from a partition
 *
 * Arguments   : pmem    is a pointer to the memory partition control block
 *
 *           err      is a pointer to a variable containing an error message which will be set by this
 *           function to either:
 *
 *           OS_NO_ERR if the memory partition has been created correctly.
 *           OS_MEM_NO_FREE_BLKS if there are no more free memory blocks to allocate to caller
 *
 * Returns     : A pointer to a memory block if no error is detected
 *           A pointer to NULL if an error is detected
 ****
 */

void *OSMemGet (OS_MEM *pmem, INT8U *err)
{
    void    *pblk;

    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree > 0) { /* See if there are any free memory blocks */
        pblk      = pmem->OSMemFreeList; /* Yes, point to next free memory block */
        pmem->OSMemFreeList = *(void **)pblk; /* Adjust pointer to new free list */
        pmem->OSMemNFree--; /* One less memory block in this partition */
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR; /* No error */
        return (pblk); /* Return memory block to caller */
    } else {
        OS_EXIT_CRITICAL();
        *err = OS_MEM_NO_FREE_BLKS; /* No, Notify caller of empty memory partition */
        return ((void *)0); /* Return NULL pointer to caller */
    }
}

/*
 ****
 *          INITIALIZE MEMORY PARTITION MANAGER
 *
 * Description : This function is called by uC/OS-II to initialize the memory partition manager.
 *           Your application MUST NOT call this function.
 *
 * Arguments   : none
 *

```

```

* Returns    : none
*****
*/
void OSMemInit (void)
{
    OS_MEM *pmem;
    INT16U i;

    pmem = (OS_MEM *)&OSMemTbl[0]; /* Point to memory control block (MCB) */
    for (i = 0; i < (OS_MAX_MEM_PART - 1); i++) { /* Init. list of free memory partitions */
        pmem->OSMemFreeList = (void *)&OSMemTbl[i+1]; /* Chain list of free partitions */
        pmem->OSMemAddr = (void *)0; /* Store start address of memory partition */
        pmem->OSMemNFree = 0; /* No free blocks */
        pmem->OSMemNBlnks = 0; /* No blocks */
        pmem->OSMemBlkSize = 0; /* Zero size */
        pmem++;
    }
    OSMemTbl[OS_MAX_MEM_PART - 1].OSMemFreeList = (void *)0;
    OSMemFreeList = (OS_MEM *)&OSMemTbl[0];
}

/*
***** RELEASE A MEMORY BLOCK *****
* Description : Returns a memory block to a partition
*
* Arguments   : pmem    is a pointer to the memory partition control block
*
*             pbik    is a pointer to the memory block being released.
*
* Returns     : OS_NO_ERR      if the memory block was inserted into the partition
*               OS_MEM_FULL    if you are returning a memory block to an already FULL memory partition
*                           (You freed more blocks than you allocated!)
*****
*/
INT8U OSMemPut (OS_MEM *pmem, void *pbik)
{
    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree >= pmem->OSMemNBlnks) { /* Make sure all blocks not already returned */
        OS_EXIT_CRITICAL();

```

```

    return (OS_MEM_FULL);
}

*(void **)pblk = pmem->OSMemFreeList; /* Insert released block into free block list */
pmem->OSMemFreeList = pblk;
pmem->OSMemNFree++; /* One more memory block in this partition */
OS_EXIT_CRITICAL();
return (OS_NO_ERR); /* Notify caller that memory block was released */
}

/*
***** QUERY MEMORY PARTITION *****
*
* Description : This function is used to determine the number of free memory blocks and the number
*               of used memory blocks from a memory partition.
*
* Arguments   : pmem   is a pointer to the memory partition control block
*
*               pdata   is a pointer to a structure that will contain information about the memory
*                      partition.
*
* Returns     : OS_NO_ERR      Always returns no error.
***** */

INT8U OSMemQuery (OS_MEM *pmem, OS_MEM_DATA *pdata)
{
    OS_ENTER_CRITICAL();
    pdata->OSAddr = pmem->OSMemAddr;
    pdata->OSFreeList = pmem->OSMemFreeList;
    pdata->OSBlkSize = pmem->OSMemBlkSize;
    pdata->OSNBlks = pmem->OSMemNBlks;
    pdata->OSNFree = pmem->OSMemNFree;
    OS_EXIT_CRITICAL();
    pdata->OSNUsed = pdata->OSNBlks - pdata->OSNFree;
    return (OS_NO_ERR);
}
#endif

```

B.5 OS_Q.C

/*

```
*****
*                               uC/OS-II
*                               The Real-Time Kernel
*                               MESSAGE QUEUE MANAGEMENT
*
*   (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*       All Rights Reserved
*
*                               V2.00
*
* File : OS_Q.C
* By   : Jean J. Labrosse
*****
*/
#ifndef OS_MASTER_FILE
#include "includes.h"
#endif

#if OS_Q_EN && (OS_MAX_QS >= 2)
/*
*****
*                               LOCAL DATA TYPES
*****
*/
typedef struct os_q {
    /* QUEUE CONTROL BLOCK */
    struct os_q *OSQPtr;           /* Link to next queue control block in list of free blocks */
    void **OSQStart;              /* Pointer to start of queue data */
    void **OSQEnd;                /* Pointer to end of queue data */
    void **OSQIn;                 /* Pointer to where next message will be inserted in the Q */
    void **OSQOut;                /* Pointer to where next message will be extracted from the Q */
    INT16U OSQSize;               /* Size of queue (maximum number of entries) */
    INT16U OSQEntries;            /* Current number of entries in the queue */
} OS_Q;

/*
*****
*                               LOCAL GLOBAL VARIABLES
*****
*/
static OS_Q *OSQFreeList; /* Pointer to list of free QUEUE control blocks*/
static OS_Q OSQTbl[OS_MAX_QS]; /* Table of QUEUE control blocks */


```



```

/*
*****
*          ACCEPT MESSAGE FROM QUEUE
*
* Description: This function checks the queue to see if a message is available. Unlike OSQPend(),
*               OSQAccept() does not suspend the calling task if a message is not available.
*
* Arguments : pevent      is a pointer to the event control block
*
* Returns   : != (void *)0  is the message in the queue if one is available. The message is removed
*                  from the so the next time OSQAccept() is called, the queue will contain
*                  one less entry.
*                  == (void *)0  if the queue is empty
*                          if you passed an invalid event type
*****
*/
void *OSQAccept (OS_EVENT *pevent)
{
    void *msg;
    OS_Q *pq;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) { /* Validate event block type */
        OS_EXIT_CRITICAL();
        return ((void *)0);
    }
    pq = pevent->OSEventPtr;           /* Point at queue control block */
    if (pq->OSQEntries != 0) {         /* See if any messages in the queue */
        msg = *pq->OSQOut++;          /* Yes, extract oldest message from the queue */
        pq->OSQEntries--;            /* Update the number of entries in the queue */
        if (pq->OSQOut == pq->OSQEnd) { /* Wrap OUT pointer if we are at the end of the queue */
            pq->OSQOut = pq->OSQStart;
        }
    } else {
        msg = (void *)0;                /* Queue is empty */
    }
    OS_EXIT_CRITICAL();
    return (msg);                      /* Return message received (or NULL) */
}

/*

```

```
*****
*          CREATE A MESSAGE QUEUE
*
* Description: This function creates a message queue if free event control blocks are available.
*
* Arguments : start      is a pointer to the base address of the message queue storage area. The
*              storage area MUST be declared as an array of pointers to 'void' as follows
*
*              void *MessageStorage[size]
*
*          size      is the number of elements in the storage area
*
* Returns   : != (void *)0 is a pointer to the event control block (OS_EVENT) associated with the
*              created queue
*              == (void *)0 if no event control blocks were available
*****
*/
OS_EVENT *OSQCreate (void **start, INT16U size)
{
    OS_EVENT *pevent;
    OS_Q    *pq;

    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;           /* Get next free event control block */
    if (OSEventFreeList != (OS_EVENT *)0) { /* See if pool of free ECB pool was empty */
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) { /* See if we have an event control block */
        OS_ENTER_CRITICAL();           /* Get a free queue control block */
        pq = OSQFreeList;
        if (OSQFreeList != (OS_Q *)0) {
            OSQFreeList = OSQFreeList->OSQPtr;
        }
        OS_EXIT_CRITICAL();
        if (pq != (OS_Q *)0) { /* See if we were able to get a queue control block */
            pq->OSQStart      = start;    /* Yes, initialize the queue */
            pq->OSQEnd        = &start[size];
            pq->OSQIn         = start;
            pq->OSQOut        = start;
            pq->OSQSize        = size;
            pq->OSQEntries     = 0;
            pevent->OSEventType = OS_EVENT_TYPE_Q;
        }
    }
}
```



附录B μC/OS-II 与处理器类型无关的源代码

```
    pevent->OSEventPtr = pq;
    OSEventWaitListInit(pevent);
} else { /* No, since we couldn't get a queue control block */
    OS_ENTER_CRITICAL(); /* Return event control block on error */
    pevent->OSEventPtr = (void *)OSEventFreeList;
    OSEventFreeList = pevent;
    OS_EXIT_CRITICAL();
    pevent = (OS_EVENT *)0;
}
}

return (pevent);
}

/*
*****
*          FLUSH QUEUE
*
* Description : This function is used to flush the contents of the message queue.
*
* Arguments   : none
*
* Returns     : OS_NO_ERR      upon success
*                 OS_ERR_EVENT_TYPE If you didn't pass a pointer to a queue
*****
*/
INT8U OSQFlush (OS_EVENT *pevent)
{
    OS_Q *pq;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) /* Validate event block type */
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pq = pevent->OSEventPtr; /* Point to queue storage structure */
    pq->OSQIn = pq->OSQStart;
    pq->OSQOut = pq->OSQStart;
    pq->OSQEntries = 0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```



```
/*
*****QUEUE MODULE INITIALIZATION*****
*
* Description : This function is called by uC/OS-II to initialize the message queue module. Your
*                application MUST NOT call this function.
*
* Arguments   : none
*
* Returns     : none
******/
void OSQInit (void)
{
    INT16U i;

    for (i = 0; i < (OS_MAX_QS - 1); i++) {      /* Init. list of free QUEUE control blocks */
        OSQTbl[i].OSQPtr = &OSQTbl[i+1];
    }
    OSQTbl[OS_MAX_QS - 1].OSQPtr = (OS_Q *)0;
    OSQFreeList           = &OSQTbl[0];
}

/*
*****PEND ON A QUEUE FOR A MESSAGE*****
*
* Description: This function waits for a message to be sent to a queue
*
* Arguments   : pevent      is a pointer to the event control block associated with the desired queue
*
*               timeout     is an optional timeout period (in clock ticks). If non-zero, your task will
*                           wait for a message to arrive at the queue up to the amount of time
*                           specified by this argument. If you specify 0, however, your task will wait
*                           forever at the specified queue or, until a message arrives.
*
*               err        is a pointer to where an error message will be deposited. Possible error
*                           messages are:
*
*                           OS_NO_ERR      The call was successful and your task received a message.
*                           OS_TIMEOUT     A message was not received within the specified timeout
*                           OS_ERR_EVENT_TYPE You didn't pass a pointer to a queue

```



附录 B μC/OS-II 与处理器类型无关的源代码

```
* OS_ERR_PEND_ISR    If you called this function from an ISR and the result
* would lead to a suspension.
*
* Returns : != (void *)0  is a pointer to the message received
*          == (void *)0  if no message was received or you didn't pass a pointer to a queue.
*****  
使用本教材请尊重相关知识产权!  
*****
```

```
void *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    void *msg;
    OS_Q *pq;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) /* Validate event block type */
        OS_EXIT_CRITICAL();
    *err = OS_ERR_EVENT_TYPE;
    return ((void *)0);
}
pq = pevent->OSEventPtr;           /* Point at queue control block */
if (pq->OSQEntries != 0) {         /* See if any messages in the queue */
    msg = *pq->OSQOut++;          /* Yes, extract oldest message from the queue */
    pq->OSQEntries--;             /* Update the number of entries in the queue */
    if (pq->OSQOut == pq->OSQEnd) { /* Wrap OUT pointer if we are at the end of the queue */
        pq->OSQOut = pq->OSQStart;
    }
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
} else if (OSIntNesting > 0) {      /* See if called from ISR ... */
    OS_EXIT_CRITICAL();            /* ... can't PEND from an ISR */
    *err = OS_ERR_PEND_ISR;
} else {
    OSTCBCur->OSTCBStat |= OS_STAT_Q; /* Task will have to pend for a message to be posted */
    OSTCBCur->OSTCBDly = timeout; /* Load timeout into TCB */
    OSEventTaskWait(pevent);       /* Suspend task until event or timeout occurs */
    OS_EXIT_CRITICAL();
    OSSched();                   /* Find next highest priority task ready to run */
    OS_ENTER_CRITICAL();
    if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) /* Did we get a message? */
        OSTCBCur->OSTCBMsg = (void *)0; /* Extract message from TCB (Put there by QPost) */
    OSTCBCur->OSTCBStat = OS_STAT_RDY;
    OSTCBCur->OSTCBEVENTPTR = (OS_EVENT *)0; /* No longer waiting for event */
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
```

```

) else if (OSTCBCur->OSTCBStat & OS_STAT_Q) /* Timed out if status indicates pending on Q */
    OSEventTO(pevent);
    OS_EXIT_CRITICAL();
    msg = (void *)0; /* No message received */ 
    *err = OS_TIMEOUT; /* Indicate a timeout occurred */
} else {
    msg = *pq->OSQOut++; /* Extract message from queue */ 
    pq->OSQEntries--; /* Update the number of entries in the queue */
    if (pq->OSQOut == pq->OSPEnd) /* Wrap OUT pointer if we are at the end of Q */
        pq->OSQOut = pq->OSQStart;
}
OSTCBCur->OSTCBEVENTPTR = (OS_EVENT *)0;
OS_EXIT_CRITICAL();
*err = OS_NO_ERR;
}

return (msg); /* Return message received (or NULL) */
}

```

```

/*
*****
*          POST MESSAGE TO A QUEUE
*
* Description: This function sends a message to a queue
*
* Arguments : pevent      is a pointer to the event control block associated with the desired queue
*
*             msg        is a pointer to the message to send. You MUST NOT send a NULL pointer.
*
* Returns   : OS_NO_ERR      The call was successful and the message was sent
*             OS_Q_FULL     If the queue cannot accept any more messages because it is full.
*             OS_ERR_EVENT_TYPE If you didn't pass a pointer to a queue.
*****
*/

```

```

INT8U OSQPost (OS_EVENT *pevent, void *msg)
{
    OS_Q *pq;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) /* validate event block type */
        OS_EXIT_CRITICAL();
    return (OS_ERR_EVENT_TYPE);
}

```

```

}

if (pevent->OSEventGrp) {      /* See if any task pending on queue      */
    OSEventTaskRdy(pevent, msg, OS_STAT_Q); /* Ready highest priority task waiting on event */
    OS_EXIT_CRITICAL();
    OSSched();           /* Find highest priority task ready to run      */
    return (OS_NO_ERR);
} else {
    pq = pevent->OSEventPtr;           /* Point to queue control block      */
    if (pq->OSQEntries >= pq->OSQSize) { /* Make sure queue is not full      */
        OS_EXIT_CRITICAL();
        return (OS_Q_FULL);
    } else {
        *pq->OSQIn++ = msg;          /* Insert message into queue      */
        pq->OSQEntries++;          /* Update the nbr of entries in the queue */
        if (pq->OSQIn == pq->OSQEnd) { /* Wrap IN ptr if we are at end of queue*/
            pq->OSQIn = pq->OSQStart;
        }
        OS_EXIT_CRITICAL();
    }
    return (OS_NO_ERR);
}

}

/*
*****
*          POST MESSAGE TO THE FRONT OF A QUEUE
*
* Description: This function sends a message to a queue but unlike OSQPost(), the message is posted
*               at the front instead of the end of the queue. Using OSQPostFront() allows you to send
*               'priority' messages.
*
* Arguments : pevent      is a pointer to the event control block associated with the desired queue
*
*             msg       is a pointer to the message to send. You MUST NOT send a NULL pointer.
*
* Returns   : OS_NO_ERR      The call was successful and the message was sent
*             OS_Q_FULL     If the queue cannot accept any more messages because it is full.
*             OS_ERR_EVENT_TYPE If you didn't pass a pointer to a queue.
*****
*/
INT8U OSQPostFront (OS_EVENT *pevent, void *msg)
{
    OS_Q  *pq;

```

```

OS_ENTER_CRITICAL();
if (pevent->OSEventType != OS_EVENT_TYPE_Q) { /* Validate event block type*/
    OS_EXIT_CRITICAL();
    return (OS_ERR_EVENT_TYPE);
}

if (pevent->OSEventGrp) {      /* See if any task pending on queue */
    OSEventTaskRdy(pevent, msg, OS_STAT_Q); /* Ready highest priority task waiting on event */
    OS_EXIT_CRITICAL();
    OSSched();           /* Find highest priority task ready to run      */
    return (OS_NO_ERR);
} else {
    pq = pevent->OSEventPtr;          /* Point to queue control block      */
    if (pq->OSQEntries >= pq->OSQSize) { /* Make sure queue is not full */
        OS_EXIT_CRITICAL();
        return (OS_Q_FULL);
    } else {
        if (pq->OSQOut == pq->OSQStart) { /* Wrap OUT ptr if we are at the 1st queue entry */
            pq->OSQOut = pq->OSQEnd;
        }
        pq->OSQOut--;
        *pq->OSQOut = msg;           /* Insert message into queue      */
        pq->OSQEntries++;          /* Update the nbr of entries in the queue */
        OS_EXIT_CRITICAL();
    }
    return (OS_NO_ERR);
}
}

/*
*****
*               QUERY A MESSAGE QUEUE
*
* Description: This function obtains information about a message queue.
*
* Arguments  : pevent      is a pointer to the event control block associated with the desired mailbox
*
*             pdata       is a pointer to a structure that will contain information about the message
*                         queue.
*
* Returns   : OS_NO_ERR      The call was successful and the message was sent
*             OS_ERR_EVENT_TYPE If you are attempting to obtain data from a non queue.
*****

```

```

/*
INT8U OSQQQuery (OS_EVENT *pevent, OS_Q_DATA *pdata)
{
    OS_Q    *pq;
    INT8U   i;
    INT8U   *psrc;
    INT8U   *pdest;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) /* Validate event block type */
        OS_EXIT_CRITICAL();
    return (OS_ERR_EVENT_TYPE);
}

pdata->OSEventGrp = pevent->OSEventGrp; /* Copy message mailbox wait list */
psrc      = &pevent->OSEventTbl[0];
pdest     = &pdata->OSEventTbl[0];
for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
    *pdest++ = *psrc++;
}
pq = (OS_Q *)pevent->OSEventPtr;
if (pq->OSQEntries > 0) {
    pdata->OSMsg = pq->OSQOut;           /* Get next message to return if available */
} else {
    pdata->OSMsg = (void *)0;
}
pdata->OSNMsgs = pq->OSQEntries;
pdata->OSQSize = pq->OSQSize;
OS_EXIT_CRITICAL();
return (OS_NO_ERR);
}
#endif

```



B.6 OS_SEM.C

```

/*
***** uC/OS-II *****
*          The Real-Time Kernel
*          SEMAPHORE MANAGEMENT
*
*          (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL

```

```
*                               All Rights Reserved
*
*                               V2.00
*
* File : OS_SEM.C
* By   : Jean J. Labrosse
*****
*/
#ifndef OS_MASTER_FILE
#include "includes.h"
#endif

#if OS_SEM_EN
/*
*****
*          ACCEPT SEMAPHORE
*
* Description: This function checks the semaphore to see if a resource is available or, if an event
*               occurred. Unlike OSSemPend(), OSSemAccept() does not suspend the calling task if the
*               resource is not available or the event did not occur.
*
* Arguments : pevent      is a pointer to the event control block
*
* Returns   : > 0      if the resource is available or the event did not occur the semaphore is
*               decremented to obtain the resource.
*             == 0      if the resource is not available or the event did not occur or,
*               you didn't pass a pointer to a semaphore
*****
*/
INT16U OSSemAccept (OS_EVENT *pevent)
{
    INT16U cnt;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) /* Validate event block type*/
        OS_EXIT_CRITICAL();
    return (0);
}
cnt = pevent->OSEventCnt;
if (cnt > 0) /* See if resource is available */
    pevent->OSEventCnt--; /* Yes, decrement semaphore and notify caller */
}

*****
```





```

OS_EXIT_CRITICAL();
return (cnt);                                /* Return semaphore count           */
}

/*
*****CREATE A SEMAPHORE
*
* Description: This function creates a semaphore.
*
* Arguments : cnt      is the initial value for the semaphore. If the value is 0, no resource is
*              available (or no event has occurred). You initialize the semaphore to a
*              non-zero value to specify how many resources are available (e.g. if you have
*              10 resources, you would initialize the semaphore to 10).
*
* Returns   : != (void *)0 is a pointer to the event control block (OS_EVENT) associated with the
*              created semaphore
*             == (void *)0 if no event control blocks were available
*****
*/
OS_EVENT *OSSemCreate (INT16U cnt)
{
    OS_EVENT *pevent;

    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                  /* Get next free event control block      */
    if (OSEventFreeList != (OS_EVENT *)0){/* See if pool of free ECB pool was empty*/
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {          /* Get an event control block            */
        pevent->OSEventType = OS_EVENT_TYPE_SEM;
        pevent->OSEventCnt = cnt;           /* Set semaphore value                 */
        OSEventWaitListInit(pevent);
    }
    return (pevent);
}

/*
*****PEND ON SEMAPHORE
*

```



```

/*
 * Description: This function waits for a semaphore.
 *
 * Arguments : pevent      is a pointer to the event control block associated with the desired
 *              semaphore.
 *
 *           timeout      is an optional timeout period (in clock ticks). If non-zero, your task will
 *                         wait for the resource up to the amount of time specified by this argument.
 *                         If you specify 0, however, your task will wait forever at the specified
 *                         semaphore or, until the resource becomes available (or the event occurs).
 *
 *           err          is a pointer to where an error message will be deposited. Possible error
 *                         messages are:
 *
 *                         OS_NO_ERR      The call was successful and your task owns the resource
 *                                         or, the event you are waiting for occurred.
 *                         OS_TIMEOUT     The semaphore was not received within the specified
 *                                         timeout.
 *                         OS_ERR_EVENT_TYPE If you didn't pass a pointer to a semaphore.
 *                         OS_ERR_PEND_ISR  If you called this function from an ISR and the result
 *                                         would lead to a suspension.
 *
 * Returns   : none
 ****
 */
void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) /* Validate event block type*/
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
    }

    if (pevent->OSEventCnt > 0) /* If sem. is positive, resource available...*/
        pevent->OSEventCnt--; /* ... decrement semaphore only if positive. */
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) /* See if called from ISR ... */
        OS_EXIT_CRITICAL(); /* ... can't PEND from an ISR */
        *err = OS_ERR_PEND_ISR;
    } else /* Otherwise, must wait until event occurs */
        OSTCBCur->OSTCBSStat |= OS_STAT_SEM; /* Resource not available, pend on semaphore */
        OSTCBCur->OSTCBDly = timeout; /* Store pend timeout in TCB */
        OSEventTaskWait(pevent); /* Suspend task until event or timeout occurs */
        OS_EXIT_CRITICAL();
}

```

```

OSSched();           /* Find next highest priority task ready      */
OS_ENTER_CRITICAL();
if (OSTCBCur->OSTCBStat & OS_STAT_SEM) { /* Must have timed out if still waiting for event */
    OSEventTO(pevent);
    OS_EXIT_CRITICAL();
    *err = OS_TIMEOUT;        /* Indicate that didn't get event within TO   */
} else {
    OSTCBCur->OSTCBEVENTPTR = (OS_EVENT *)0;
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
}

/*
*****
*          POST TO A SEMAPHORE
*
* Description: This function signals a semaphore
*
* Arguments : pevent      is a pointer to the event control block associated with the desired
*              semaphore.
*
* Returns   : OS_NO_ERR      The call was successful and the semaphore was signaled.
*              OS_SEM_OVF      If the semaphore count exceeded its limit. In other words, you
*                            have signalled the semaphore more often than you waited on it with
*                            either OSSemAccept() or OSSemPend().
*              OS_ERR_EVENT_TYPE  If you didn't pass a pointer to a semaphore
*****
*/
INT8U OSSemPost (OS_EVENT *pevent)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {      /* Validate event block type */
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {      /* See if any task waiting for semaphore */
        OSEventTaskRdy(pevent, (void *)0, OS_STAT_SEM); /* Ready highest prio task waiting on event */
        OS_EXIT_CRITICAL();
        OSSched();           /* Find highest priority task ready to run      */
        return (OS_NO_ERR);
    } else {

```

```

if (pevent->OSEventCnt < 65535) {           /* Make sure semaphore will not overflow */
    pevent->OSEventCnt++; /* Increment semaphore count to register event */
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
} else {           /* Semaphore value has reached its maximum */
    OS_EXIT_CRITICAL();
    return (OS_SEM_OVF);
}
}

/*
*****
*          QUERY A SEMAPHORE
*
* Description: This function obtains information about a semaphore
*
* Arguments : pevent      is a pointer to the event control block associated with the desired
*              semaphore
*
*             pdata       is a pointer to a structure that will contain information about the
*              semaphore.
*
* Returns   : OS_NO_ERR      The call was successful and the message was sent.
*              OS_ERR_EVENT_TYPE If you are attempting to obtain data from a non semaphore.
*****
*/
INT8U OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata)
{
    INT8U i;
    INT8U *psrc;
    INT8U *pdest;

    OS_ENTRY_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {           /* Validate event block type */
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;                  /* Copy message mailbox wait list */
    psrc = &pevent->OSEventTbl[0];
    pdest = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdest++ = *psrc++;
    }
}

```

```

    pdata->OSCnt      = pevent->OSEventCnt;           /* Get semaphore count */
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}

#endif

```

B.7 OS_TASK.C

```

/*
***** uC/OS-II
*          The Real-Time Kernel
*          TASK MANAGEMENT
*
*          (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*          All Rights Reserved
*
*          V2.00
*
* File : OS_TASK.C
* By   : Jean J. Labrosse
*****
*/
#ifndef OS_MASTER_FILE
#include "includes.h"
#endif

/*
***** LOCAL FUNCTION PROTOTYPES
*****
*/
static void OSDummy(void);

/*
***** DUMMY FUNCTION
*
* Description: This function doesn't do anything. It is called by OSTaskDel() to ensure that
*               interrupts are disabled immediately after they are enabled.
*/

```



```
/*
 * Arguments : none
 *
 * Returns   : none
 ****
 */
static void OSdummy (void)
{
}

/*
*****
*          CHANGE PRIORITY OF A TASK
*
* Description: This function allows you to change the priority of a task dynamically. Note that
*               the new priority MUST be available.
*
* Arguments : oldp    is the old priority
*
*             newp    is the new priority
*
* Returns   : OS_NO_ERR      is the call was successful
*             OS_PRIO_INVALID if the priority you specify is higher than the maximum allowed
*                               (i.e. >= OS_LOWEST_PRIO)
*             OS_PRIO_EXIST   if the new priority already exist.
*             OS_PRIO_ERR     there is no task with the specified OLD priority (i.e. the OLD task does
*                               not exist.
*****
*/
#endif OS_TASK_CHANGE_PRIO_EN
INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio)
{
    OS_TCB    *ptcb;
    OS_EVENT  *pevent;
    INT8U    x;
    INT8U    y;
    INT8U    bitx;
    INT8U    bity;

    if ((oldprio >= OS_LOWEST_PRIO && oldprio != OS_PRIO_SELF) ||
        newprio >= OS_LOWEST_PRIO) {
```

附录B μC/OS-II 与处理器类型无关的源代码

```
return (OS_PRIO_INVALID);
}

OS_ENTER_CRITICAL();

if (OSTCBPrioTbl[newprio] != (OS_TCB *)0) { /* New priority must not already exist */
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
} else {
    OSTCBPrioTbl[newprio] = (OS_TCB *)1; /* Reserve the entry to prevent others */
    OS_EXIT_CRITICAL();
    y = newprio >> 3; /* Precompute to reduce INT. latency */
    bity = OSMapTbl[y];
    x = newprio & 0x07;
    bitx = OSMapTbl[x];
    OS_ENTER_CRITICAL();
    if (oldprio == OS_PRIO_SELF) { /* See if changing self */
        oldprio = OSTCBCur->OSTCBPrio; /* Yes, get priority */
    }
    if ((ptcb = OSTCBPrioTbl[oldprio]) != (OS_TCB *)0) { /* Task to change must exist */
        OSTCBPrioTbl[oldprio] = (OS_TCB *)0; /* Remove TCB from old priority */
        if (OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) { /* If task is ready make it not ready */
            if ((OSRdyTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0) {
                OSRdyGrp &= ~ptcb->OSTCBBitY;
            }
            OSRdyGrp |= bity; /* Make new priority ready to run */
            OSRdyTbl[y] |= bitx;
        } else {
            if ((pevent = ptcb->OSTCBEVENTPTR) != (OS_EVENT *)0) { /* Remove from event wait list */
                if ((pevent->OSEventTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0) {
                    pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
                }
                pevent->OSEventGrp |= bity; /* Add new priority to wait list */
                pevent->OSEventTbl[y] |= bitx;
            }
        }
        OSTCBPrioTbl[newprio] = ptcb; /* Place pointer to TCB @ new priority*/
        ptcb->OSTCBPrio = newprio; /* Set new task priority */
        ptcb->OSTCBY = y;
        ptcb->OSTCBX = x;
        ptcb->OSTCBBitY = bity;
        ptcb->OSTCBBitX = bitx;
        OS_EXIT_CRITICAL();
        OSSched(); /* Run highest priority task ready */
        return (OS_NO_ERR);
    } else {
        OSTCBPrioTbl[newprio] = (OS_TCB *)0; /* Reserve Release the reserved prio. */
    }
}
```

```

    OS_EXIT_CRITICAL();
    return (OS_PRIO_ERR);           /* Task to change didn't exist */
}
}

#endif

/*
*****CREATE A TASK*****
*
* Description: This function is used to have uC/OS-II manage the execution of a task. Tasks can
* either be created prior to the start of multitasking or by a running task. A task
* cannot be created by an ISR.
*
* Arguments : task    is a pointer to the task's code
*
*             pdata   is a pointer to an optional data area which can be used to pass parameters to
*                     the task when the task first executes. Where the task is concerned it thinks
*                     it was invoked and passed the argument 'pdata' as follows:
*
*                     void Task (void *pdata)
*                     {
*                         for (;;) {
*                             Task code;
*                         }
*                     }
*
*             ptos    is a pointer to the task's top of stack. If the configuration constant
*                     OS_STK_GROWTH is set to 1, the stack is assumed to grow downward (i.e. from high
*                     memory to low memory). 'pstk' will thus point to the highest (valid) memory
*                     location of the stack. If OS_STK_GROWTH is set to 0, 'pstk' will point to the
*                     lowest memory location of the stack and the stack will grow with increasing
*                     memory locations.
*
*             prio    is the task's priority. A unique priority MUST be assigned to each task and the
*                     lower the number, the higher the priority.
*
* Returns   : OS_NO_ERR      if the function was successful.
*             OS_PRIO_EXIT    if the task priority already exist
*                           (each task MUST have a unique priority).
*             OS_PRIO_INVALID if the priority you specify is higher than the maximum allowed
*                           (i.e. > OS_LOWEST_PRIO)
*****
```



```

/*
 * OS_TaskCreate - Create a task
 * OS_TaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
 {
    void    *psp;
    INT8U   err;

    if (prio > OS_LOWEST_PRIO) {           /* Make sure priority is within allowable range */
        return (OS_PRIO_INVALID);
    }

    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) /* Make sure task doesn't already exist at this priority */
        OSTCBPrioTbl[prio] = (OS_TCB *)1; /* Reserve the priority to prevent others from doing ... */
                                         /* ... the same thing until task is created.      */
    OS_EXIT_CRITICAL();
    psp = (void *)OSTaskStkInit(task, pdata, ptos, 0); /* Initialize the task's stack          */
    err = OSTCBInit(prio, psp, (void *)0, 0, 0, (void *)0, 0);
    if (err == OS_NO_ERR) {
        OS_ENTER_CRITICAL();
        OSTaskCtr++;           /* Increment the #tasks counter       */
        OSTaskCreateHook(OSTCBPrioTbl[prio]); /* Call user defined hook */
        OS_EXIT_CRITICAL();
        if (OSRunning) {         /* Find highest priority task if multitasking has started */
            OSSched();
        }
    } else {
        OS_ENTER_CRITICAL();
        OSTCBPrioTbl[prio] = (OS_TCB *)0; /* Make this priority available to others */
        OS_EXIT_CRITICAL();
    }
    return (err);
} else {
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
}

#endif

/*
***** CREATE A TASK (Extended Version)
*/

```

```
* Description: This function is used to have μC/OS-II manage the execution of a task. Tasks can
* either be created prior to the start of multitasking or by a running task. A task
* cannot be created by an TSR. This function is similar to OSTaskCreate() except that
* it allows additional information about a task to be specified.
*
* Arguments : task      is a pointer to the task's code
*
*             pdata     is a pointer to an optional data area which can be used to pass parameters to
*                         the task when the task first executes. Where the task is concerned it thinks
*                         it was invoked and passed the argument 'pdata' as follows:
*
*                         void Task (void *pdata)
*                         {
*                             for (;;) {
*                                 Task code;
*                             }
*                         }
*
*             ptos      is a pointer to the task's top of stack. If the configuration constant
*                         OS_STK_GROWTH is set to 1, the stack is assumed to grow downward (i.e. from high
*                         memory to low memory). 'pstk' will thus point to the highest (valid) memory
*                         location of the stack. If OS_STK_GROWTH is set to 0, 'pstk' will point to the
*                         lowest memory location of the stack and the stack will grow with increasing
*                         memory locations. 'pstk' MUST point to a valid 'free' data item.
*
*             prio      is the task's priority. A unique priority MUST be assigned to each task and the
*                         lower the number, the higher the priority.
*
*             id        is the task's ID (0..65535)
*
*             pbos      is a pointer to the task's bottom of stack. If the configuration constant
*                         OS_STK_GROWTH is set to 1, the stack is assumed to grow downward (i.e. from high
*                         memory to low memory). 'pbos' will thus point to the LOWEST (valid) memory
*                         location of the stack. If OS_STK_GROWTH is set to 0, 'pbos' will point to the
*                         HIGHEST memory location of the stack and the stack will grow with increasing
*                         memory locations. 'pbos' MUST point to a valid 'free' data item.
*
*             stk_size  is the size of the stack in number of elements. If OS_STK is set to INT8U,
*                         'stk_size' corresponds to the number of bytes available. If OS_STK is set to
*                         INT16U, 'stk_size' contains the number of 16-bit entries available. Finally, if
*                         OS_STK is set to INT32U, 'stk_size' contains the number of 32-bit entries
*                         available on the stack.
*
*             pext      is a pointer to a user supplied memory location which is used as a TCB extension.
*                         For example, this user memory can hold the contents of floating-point registers
```

附录B μC/OS-II 与处理器类型无关的源代码

```
*           during a context switch, the time each task takes to execute, the number of times
*           the task has been switched-in, etc.
*
*           opt      contains additional information (or options) about the behavior of the task. The
*           LOWER 8-bits are reserved by uC/OS-II while the upper 8 bits can be application
*           specific. See OS_TASK_OPT_??? in uCOS-II.H.
*
* Returns   : OS_NO_ERR      if the function was successful.
*           OS_PRIO_EXIT    if the task priority already exist
*                         (each task MUST have a unique priority).
*           OS_PRIO_INVALID if the priority you specify is higher than the maximum allowed
*                         (i.e. > OS_LOWEST_PRIO)
*****
*/
#endif OS_TASK_CREATE_EXT_EN

INT8U OSTaskCreateExt (void (*task)(void *pd),
                      void     *pdata,
                      OS_STK  *ptos,
                      INT8U    prio,
                      INT16U   id,
                      OS_STK  *pbos,
                      INT32U   stk_size,
                      void     *pext,
                      INT16U   opt)
{
    void    *psp;
    INT8U   err;
    INT16U   i;
    OS_STK  *pfill;

    if (prio > OS_LOWEST_PRIO) {          /* Make sure priority is within allowable range      */
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {/* Make sure task doesn't already exist at this priority */
        OSTCBPrioTbl[prio] = (OS_TCB *)1;    /* Reserve the priority to prevent others from doing ... */
        /* ... the same thing until task is created.          */
        OS_EXIT_CRITICAL();
    }

    if (opt & OS_TASK_OPT_STK_CHK) { /* See if stack checking has been enabled      */
        if (opt & OS_TASK_OPT_STK_CLR) /* See if stack needs to be cleared */
            pfill = pbos;           /* Yes, fill the stack with zeros      */
    }
}
```

```

        for (i = 0; i < stk_size; i++) {
            #if OS_STK_GROWTH == 1
            *pfill++ = (OS_STK)0;
            #else
            *pfill-- = (OS_STK)0;
            #endif
        }
    }

    psp = (void *)OSTaskStkInit(task, pdata, ptos, opt); /* Initialize the task's stack */
    err = OSTCBInit(prio, psp, pbos, id, stk_size, pext, opt);
    if (err == OS_NO_ERR) {
        OS_ENTER_CRITICAL();
        OSTaskCtr++;           /* Increment the #tasks counter      */
        OSTaskCreateHook(OSTCBPrioTbl[prio]); /* Call user defined hook      */
        OS_EXIT_CRITICAL();
        if (OSRunning) {         /* Find highest priority task if multitasking has started */
            OSSched();
        }
    } else {
        OS_ENTER_CRITICAL();
        OSTCBPrioTbl[prio] = (OS_TCB *)0; /* Make this priority available to others */
        OS_EXIT_CRITICAL();
    }
    return (err);
} else {
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
#endif

/*
*****DELETE A TASK*****
*
* Description: This function allows you to delete a task. The calling task can delete itself by
*               its own priority number. The deleted task is returned to the dormant state and can be
*               re-activated by creating the deleted task again.
*
* Arguments : prio   is the priority of the task to delete. Note that you can explicitly delete
*               the current task without knowing its priority level by setting 'prio' to
*               OS_PRIO_SELF.

```

```

*
* Returns    : OS_NO_ERR           if the call is successful
*              OS_TASK_DEL_IDLE   if you attempted to delete uC/OS-II's idle task
*              OS_PRIO_INVALID     if the priority you specify is higher than the maximum allowed
*                                (i.e. >= OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
*              OS_TASK_DEL_ERR      if the task you want to delete does not exist
*              OS_TASK_DEL_ISR      if you tried to delete a task from an ISR
*
* Notes      : 1) To reduce interrupt latency, OSTaskDel() 'disables' the task:
*                  a) by making it not ready
*                  b) by removing it from any wait lists
*                  c) by preventing OSTimeTick() from making the task ready to run.
*                     The task can then be 'unlinked' from the miscellaneous structures in uC/OS-II.
* 2) The function OSDummy() is called after OS_EXIT_CRITICAL() because, on most processors,
*    the next instruction following the enable interrupt instruction is ignored. You can
*    replace OSDummy() with a macro that basically executes a NO OP (i.e. OS_NOP()). The
*    NO OP macro would avoid the execution time of the function call and return.
* 3) An ISR cannot delete a task.
* 4) The lock nesting counter is incremented because, for a brief instant, if the current
*    task is being deleted, the current task would not be able to be rescheduled because
*    it is removed from the ready list. Incrementing the nesting counter prevents another
*    task from being scheduled. This means that the ISR would return to the current task which
*    is being deleted. The rest of the deletion would thus be able to be completed.
*****
```

*/

```

#endif OS_TASK_DEL_EN
INT8U OSTaskDel (INT8U prio)
{
    OS_TCB  *ptcb;
    OS_EVENT *pevent;

    if (prio == OS_IDLE_PRIO) {          /* Not allowed to delete idle task */
        return (OS_TASK_DEL_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {      /* Task priority valid ? */
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSIntNesting > 0) {            /* See if trying to delete from ISR */
        OS_EXIT_CRITICAL();
        return (OS_TASK_DEL_ISR);
    }
}
```

```

if (prio == OS_PRIO_SELF) {           /* See if requesting to delete self */
    prio = OSTCBCur->OSTCBPrio;      /* Set priority to delete to current */
}

if ((ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) {           /* Task to delete must exist */
    if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) /* Make task not ready */
        OSRdyGrp &= ~ptcb->OSTCBBitY;
}

if ((pevent = ptcb->OSTCBEVENTPTR) != (OS_EVENT *)0) { /* If task is waiting on event */
    if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) /* ... remove task from */
        pevent->OSEventGrp &= ~ptcb->OSTCBBitY; /* ... event ctrl block */
}

ptcb->OSTCBDly = 0;           /* Prevent OSTimeTick() from updating */
ptcb->OSTCBStat = OS_STAT_RDY; /* Prevent task from being resumed */
OSLockNesting++;
OS_EXIT_CRITICAL();           /* Enabling INT. ignores next instruc. */
OSDummy();                   /* ... Dummy ensures that INTs will be */
OS_ENTER_CRITICAL();          /* ... disabled HERE! */
OSLockNesting--;
OSTaskDelHook(ptcb);         /* Call user defined hook */
OSTaskCtr--;                 /* One less task being managed */
OSTCBPrioTbl[prio] = (OS_TCB *)0; /* Clear old priority entry */
if (ptcb->OSTCBPrev == (OS_TCB *)0) { /* Remove from TCB chain */
    ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
    OSTCBList            = ptcb->OSTCBNext;
} else {
    ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;
    ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;
}
ptcb->OSTCBNext = OSTCBFreeList; /* Return TCB to free TCB list */
OSTCBFreeList = ptcb;
OS_EXIT_CRITICAL();
OSSched();                  /* Find new highest priority task */
return (OS_NO_ERR);
} else {
    OS_EXIT_CRITICAL();
    return (OS_TASK_DEL_ERR);
}
}

#endif

/*
*****
* REQUEST THAT A TASK DELETE ITSELF

```

```

*
* Description: This function is used to:
*               a) notify a task to delete itself.
*               b) to see if a task requested that the current task delete itself.
*
* This function is a little tricky to understand. Basically, you have a task that needs
* to be deleted however, this task has resources that it has allocated (memory buffers,
* semaphores, mailboxes, queues etc.). The task cannot be deleted otherwise these
* resources would not be freed. The requesting task calls OSTaskDelReq() to indicate
* that the task needs to be deleted. Deleting of the task is however, deferred to the
* task to be deleted. For example, suppose that task #10 needs to be deleted. The
* requesting task example, task #5, would call OSTaskDelReq(10). When task #10 gets
* to execute, it calls this function by specifying OS_PRIO_SELF and monitors the returned
* value. If the return value is OS_TASK_DEL_REQ, another task requested a task delete.
* Task #10 would look like this:
*
*     void Task(void *data)
*     {
*         .
*         .
*         .
*         while (1) {
*             OSTimeDly(1);
*             if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ) {
*                 Release any owned resources;
*                 De-allocate any dynamic memory;
*                 OSTaskDel(OS_PRIO_SELF);
*             }
*         }
*     }
*
* Arguments : prio    is the priority of the task to request the delete from
*
* Returns   : OS_NO_ERR      if the task exist and the request has been registered
*              OS_TASK_NOT_EXIST if the task has been deleted. This allows the caller to know whether
*                           the request has been executed.
*              OS_TASK_DEL_IDLE  if you requested to delete μC/OS-II's idle task
*              OS_PRIO_INVALID  if the priority you specify is higher than the maximum allowed
*                           (i.e. >= OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
*              OS_TASK_DEL_REQ   if a task (possibly another task) requested that the running task
*                           be deleted.
***** */
*/
#endif OS_TASK_DEL_EN
INT8U OSTaskDelReq (INT8U prio)

```

超星阅览器提醒您：
 使用本复制品
 请尊重相关知识产权！

```

(
  BOOLEAN stat;
  INT8U err;
  OS_TCB *ptcb;

  if (prio == OS_IDLE_PRIO) {           /* Not allowed to delete idle task */
    return (OS_TASK_DEL_IDLE);
  }

  if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {      /* Task priority valid ? */
    return (OS_PRIO_INVALID);
  }

  if (prio == OS_PRIO_SELF) {           /* See if a task is requesting to ... */
    OS_ENTER_CRITICAL();                /* ... this task to delete itself */
    stat = OSTCBCur->OSTCBDelReq;     /* Return request status to caller */
    OS_EXIT_CRITICAL();
    return (stat);
  } else {
    OS_ENTER_CRITICAL();
    if ((ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) {        /* Task to delete must exist */
      ptcb->OSTCBDelReq = OS_TASK_DEL_REQ;          /* Set flag indicating task to be DEL. */
      err = OS_NO_ERR;
    } else {
      err = OS_TASK_NOT_EXIST;          /* Task must be deleted */
    }
    OS_EXIT_CRITICAL();
    return (err);
  }
}

#endif

/*
*****
*             RESUME A SUSPENDED TASK
*
* Description: This function is called to resume a previously suspended task. This is the only
*               call that will remove an explicit task suspension.
*
* Arguments : prio   is the priority of the task to resume.
*
* Returns   : OS_NO_ERR           if the requested task is resumed
*             OS_PRIO_INVALID       if the priority you specify is higher than the maximum allowed
*                               (i.e. >= OS_LOWEST_PRIO)
*             OS_TASK_RESUME_PRIO   if the task to resume does not exist
*/

```

```

*           OS_TASK_NOT_SUSPENDED    if the task to resume has not been suspended
***** ****
*/
#endif

#if OS_TASK_SUSPEND_EN
INT8U OSTaskResume (INT8U prio)
{
    OS_TCB    *ptcb;

    if (prio >= OS_LOWEST_PRIO) {          /* Make sure task priority is valid */
        return (OS_PRIO_INVALID);
    }

    OS_ENTER_CRITICAL();
    if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) {          /* Task to suspend must exist */
        OS_EXIT_CRITICAL();
        return (OS_TASK_RESUME_PRIO);
    } else {
        if (ptcb->OSTCBStat & OS_STAT_SUSPEND) {      /* Task must be suspended */
            if (((ptcb->OSTCBStat & ~OS_STAT_SUSPEND) == OS_STAT_RDY) && /* Remove suspension */
                (ptcb->OSTCBDly == 0)) {          /* Must not be delayed */
                OSRdyGrp != ptcb->OSTCBBitY;      /* Make task ready to run */
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                OS_EXIT_CRITICAL();
                OSSched();
            } else {
                OS_EXIT_CRITICAL();
            }
            return (OS_NO_ERR);
        } else {
            OS_EXIT_CRITICAL();
            return (OS_TASK_NOT_SUSPENDED);
        }
    }
}
#endif

/*
***** ****
*           STACK CHECKING
*
* Description: This function is called to check the amount of free memory left on the specified
*               task's stack.
*

```



```

* Arguments : prio      is the task priority
*
*           pdata     is a pointer to a data structure of type OS_STK_DATA.
*
* Returns   : OS_NO_ERR      upon success
*           OS_PRIO_INVALIDD if the priority you specify is higher than the maximum allowed
*                               (i.e. > OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
*           OS_TASK_NOT_EXIST if the desired task has not been created
*           OS_TASK_OPT_ERR   if you did NOT specified OS_TASK_OPT_STK_CHK when the task was created
*****  

*/
#ifndef OS_TASK_CREATE_EXT_EN
INT8U OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata)
{
    OS_TCB *ptcb;
    OS_STK *pchk;
    INT32U free;
    INT32U size;

    pdata->OSFree = 0;                      /* Assume failure, set to 0 size      */
    pdata->OSUsed = 0;
    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) { /* Make sure task priority is valid */
        return (OS_PRIO_INVALIDD);
    }
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {               /* See if check for SELF */
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl(prio);
    if (ptcb == (OS_TCB *)0) {                /* Make sure task exist */
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);
    }
    if ((ptcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) == 0) /* Make sure stack checking option is set */
        OS_EXIT_CRITICAL();
    return (OS_TASK_OPT_ERR);
}
free = 0;
size = ptcb->OSTCBStkSize;
pchk = ptcb->OSTCBStkBottom;
OS_EXIT_CRITICAL();

#if OS_STK_GROWTH == 1
while (*pchk++ == 0) { /* Compute the number of zero entries on the stk */
    free++;
}

```

```

}

#else
    while (*pchk-- == 0) {
        free++;
    }
#endif
    pdata->OSFree = free * sizeof(OS_STK);           /* Compute number of free bytes on the stack */
    pdata->OSUsed = (size - free) * sizeof(OS_STK); /* Compute number of bytes used on the stack */
    return (OS_NO_ERR);
}
#endif

/*
***** SUSPEND A TASK *****
*
* Description: This function is called to suspend a task. The task can be the calling task if the
* priority passed to OSTaskSuspend() is the priority of the calling task or OS_PRIO_SELF.
*
* Arguments : prio      is the priority of the task to suspend. If you specify OS_PRIO_SELF, the
*              calling task will suspend itself and rescheduling will occur.
*
* Returns   : OS_NO_ERR      if the requested task is suspended
*              OS_TASK_SUSPEND_IDLE  if you attempted to suspend the idle task which is not allowed.
*              OS_PRIO_INVALID     if the priority you specify is higher than the maximum allowed
*                                (i.e. >= OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
*              OS_TASK_SUSPEND_PRIO if the task to suspend does not exist
*
* Note      : You should use this function with great care. If you suspend a task that is waiting for
*              an event (i.e. a message, a semaphore, a queue ...) you will prevent this task from
*              running when the event arrives.
*****
*/
#endif

#if OS_TASK_SUSPEND_EN
INT8U OSTaskSuspend (INT8U prio)
{
    BOOLEAN self;
    OS_TCB *ptcb;

    if (prio == OS_IDLE_PRIO) {          /* Not allowed to suspend idle task */
        return (OS_TASK_SUSPEND_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO & prio != OS_PRIO_SELF) { /* Task priority valid ? */

```

```

        return (OS_PRIO_INVALID);

    }

OS_ENTER_CRITICAL();

    if (prio == OS_PRIO_SELF) { /* See if suspend SELF */
        prio = OSTCBCur->OSTCBPrio;
        self = TRUE;
    } else if (prio == OSTCBCur->OSTCBPrio) { /* See if suspending self */
        self = TRUE;
    } else {
        self = FALSE; /* No suspending another task */
    }

    if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) { /* Task to suspend must exist*/
        OS_EXIT_CRITICAL();
        return (OS_TASK_SUSPEND_PRIO);
    } else {
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) { /* Make task not ready */
            OSRdyGrp &= ~ptcb->OSTCBBitY;
        }
        ptcb->OSTCBStat |= OS_STAT_SUSPEND; /* Status of task is 'SUSPENDED' */
        OS_EXIT_CRITICAL();
        if (self == TRUE) { /* Context switch only if SELF */
            OSSched();
        }
        return (OS_NO_ERR);
    }
}

#endif

/*
*****
*          QUERY A TASK
*
* Description: This function is called to obtain a copy of the desired task's TCB.
*
* Arguments : prio      is the priority of the task to obtain information from.
*
* Returns   : OS_NO_ERR      if the requested task is suspended
*             OS_PRIO_INVALID if the priority you specify is higher than the maximum allowed
*                           (i.e. > OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
*             OS_PRIO_ERR     if the desired task has not been created
*****
*/
INT8U OSTaskQuery (INT8U prio, OS_TCB *pdata)

```

```

{
    OS_TCB *ptcb;

    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) /* Task priority valid ?*/
        return (OS_PRIO_INVALID);
    }

    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {           /* See if suspend SELF                      */
        prio = OSTCBCur->OSTCBPrio;
    }
    if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) {      /* Task to query must exist   */
        OS_EXIT_CRITICAL();
        return (OS_PRIO_ERR);
    }
    *pdata = *ptcb;                  /* Copy TCB into user storage area          */
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}

```



B.8 OS_TIME.C

```

/*
*****uC/OS-II
*The Real-Time Kernel
*TIME MANAGEMENT
*
*(c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*All Rights Reserved
*
*V2.00
*
*File : OS_TIME.C
*By   : Jean J. Labrosse
*****
*/
#ifndef OS_MASTER_FILE
#include "includes.h"
#endif

/*

```

```
*****
*          DELAY TASK 'n' TICKS  (n from 0 to 65535)
*
* Description: This function is called to delay execution of the currently running task until the
*               specified number of system ticks expires. This, of course, directly equates to delaying
*               the current task for some time to expire. No delay will result If the specified delay
*               is 0. If the specified delay is greater than 0 then, a context switch will result.
*
* Arguments : ticks      is the time delay that the task will be suspended in number of clock 'ticks'.
*               Note that by specifying 0, the task will not be delayed.
*
* Returns   : none
*****
*/
void OSTimeDly (TNT16U ticks)
{
    if (ticks > 0) {                                /* 0 means no delay!           */
        OS_ENTER_CRITICAL();
        if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) { /* Delay current task */
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBDly = ticks;                /* Load ticks in TCB           */
        OS_EXIT_CRITICAL();
        OSSched();                                    /* Find next task to run!     */
    }
}

/*
*****
*          DELAY TASK FOR SPECIFIED TIME
*
* Description: This function is called to delay execution of the currently running task until
*               some time expires. This call allows you to specify the delay time in HOURS, MINUTES,
*               SECONDS and MILLISECONDS instead of ticks.
*
* Arguments : hours      specifies the number of hours that the task will be delayed (max. is 255)
*             minutes    specifies the number of minutes (max. 59)
*             seconds    specifies the number of seconds (max. 59)
*             milli      specifies the number of milliseconds (max. 999)
*
* Returns   : OS_NO_ERR
*             OS_TIME_INVALID_MINUTES
*             OS_TIME_INVALID_SECONDS
*****
```

```

*          OS_TIME_INVALID_MS
*          OS_TIME_ZERO_DLY
*
* Note(s) : The resolution on the milliseconds depends on the tick rate. For example, you can't
*           do a 10 ms delay if the ticker interrupts every 100 ms. In this case, the delay would
*           be set to 0. The actual delay is rounded to the nearest tick.
***** ****
*/
INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U milli)
{
    INT32U ticks;
    INT16U loops;

    if (hours > 0 || minutes > 0 || seconds > 0 || milli > 0) {
        if (minutes > 59) {
            return (OS_TIME_INVALID_MINUTES); /* Validate arguments to be within range */
        }
        if (seconds > 59) {
            return (OS_TIME_INVALID_SECONDS);
        }
        if (milli > 999) {
            return (OS_TIME_INVALID_MILLI);
        }
        /* Compute the total number of clock ticks required.. */
        /* .. (rounded to the nearest tick) */
        ticks = ((INT32U)hours * 3600L + (INT32U)minutes * 60L + (INT32U)seconds) * OS_TICKS_PER_SEC
            + OS_TICKS_PER_SEC * ((INT32U)milli + 500L / OS_TICKS_PER_SEC) / 1000L;
        loops = ticks / 65536L; /* Compute the integral number of 65536 tick delays*/
        ticks = ticks % 65536L; /* Obtain the fractional number of ticks */
        OSTimeDly(ticks);
        while (loops > 0) {
            OSTimeDly(32768);
            OSTimeDly(32768);
            loops--;
        }
        return (OS_NO_ERR);
    } else {
        return (OS_TIME_ZERO_DLY);
    }
}

/*
***** ****
*             RESUME A DELAYED TASK

```

```

*
* Description: This function is used resume a task that has been delayed through a call to either
*              OSTimedly() or OSTimedlyHMSM(). Note that you MUST NOT call this function to resume a
*              task that is waiting for an event with timeout. This situation would make the task look
*              like a timeout occurred (unless you desire this effect). Also, you cannot resume a task
*              that has called OSTimedlyHMSM() with a combined time that exceeds 65535 clock ticks. In
*              other words, if the clock tick runs at 100 Hz then, you will not be able to resume a
*              delayed task that called OSTimedlyHMSM(0, 10, 55, 350) or higher.
*
*          (10 Minutes * 60 + 55 Seconds + 0.35) * 100 ticks/second.
*
* Arguments : prio      specifies the priority of the task to resume
*
* Returns   : OS_NO_ERR           Task has been resumed
*             OS_PRIO_INVALID        if the priority you specify is higher than the maximum allowed
*                                     (i.e. >= OS_LOWEST_PRIO)
*             OS_TIME_NOT_DLY         Task is not waiting for time to expire
*             OS_TASK_NOT_EXIST       The desired task has not been created
*****
```

```

INT8U OSTimedlyResume (INT8U prio)
{
    OS_TCB *ptcb;
    if (prio >= OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    ptcb = (OS_TCB *)OSTCBPrioTbl[prio]; /* Make sure that task exist */
    if (ptcb != (OS_TCB *)0) {
        if (ptcb->OSTCBDly != 0) { /* See if task is delayed */
            ptcb->OSTCBDly = 0; /* Clear the time delay */
            if (!(ptcb->OSTCBStat & OS_STAT_SUSPEND)) { /* See if task is ready to run */
                OSRdyGrp |= ptcb->OSTCBBitY; /* Make task ready to run */
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
            }
            OS_EXIT_CRITICAL();
            OSSched(); /* See if this is new highest priority */
        } else {
            OS_EXIT_CRITICAL(); /* Task may be suspended */
        }
        return (OS_NO_ERR);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_TIME_NOT_DLY); /* Indicate that task was not delayed */
    }
} else {
    OS_EXIT_CRITICAL();
}
```

```

        return (OS_TASK_NOT_EXIST);           /* The task does not exist      */
    }

}

/*
***** GFT CURRENT SYSTEM TIME
*
* Description: This function is used by your application to obtain the current value of the 32-bit
*               counter which keeps track of the number of clock ticks.
*
* Arguments : none
*
* Returns   : The current value of OSTime
*****
*/
INT32U OSTimeGet (void)
{
    INT32U ticks;

    OS_ENTER_CRITICAL();
    ticks = OSTime;
    OS_EXIT_CRITICAL();
    return (ticks);
}

/*
***** SET SYSTEM CLOCK
*
* Description: This function sets the 32-bit counter which keeps track of the number of clock ticks.
*
* Arguments : ticks      specifies the new value that OSTime needs to take.
*
* Returns   : none
*****
*/
void OSTimeSet (INT32U ticks)
{
    OS_ENTER_CRITICAL();
    OSTime = ticks;
    OS_EXIT_CRITICAL();
}

```

附录 C

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

80x86 源代码在实模式、大模式下编译

C.0 OS_CPU_A.ASM

```
;*****  
;  
;          uC/OS-II  
;          The Real-Time Kernel  
;  
;(c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL  
;          All Rights Reserved  
;  
;  
;          80x86/80x88 Specific code  
;          LARGE MEMORY MODEL  
;  
;          IBM/PC Compatible Target  
;  
;  
; File : OS_CPU_A.ASM  
; By   : Jean J. Labrosse  
;*****  
  
PUBLIC _OSTickISR  
PUBLIC _OSStartHighRdy  
PUBLIC _OSCtxSw  
PUBLIC _OSIntCtxSw  
  
EXTRN _OSIntExit:FAR  
EXTRN _OSTimeTick:FAR  
EXTRN _OSTaskSwHook:FAR  
  
EXTRN _OSIntNesting:BYTE  
EXTRN _OSTickDOSCtr:BYTE  
EXTRN _OSFrioHighRdy:BYTE
```

```

        EXTRN _OSPriCur:BYTE
        EXTRN _OSRunning:BYTE
        EXTRN _OSTCBCir:DWORD
        EXTRN _OSTCBHighRdy:DWORD

.MODEL  LARGE
.CODE
.186
; *****
;           START MULTITASKING
;           void OSStartHighRdy(void)
;
; The stack frame is assumed to look as follows:
;
; OSTCBHighRdy->OSTCBStkPtr --> DS          (Low memory)
;
;             ES
;
;             DI
;
;             SI
;
;             BP
;
;             SP
;
;             BX
;
;             DX
;
;             CX
;
;             AX
;
;             OFFSET of task code address
;
;             SEGMENT of task code address
;
;             Flags to load in PSW
;
;             OFFSET of task code address
;
;             SEGMENT of task code address
;
;             OFFSET of 'pdata'
;
;             SEGMENT of 'pdata'      (High memory)
;
; Note : OSStartHighRdy() MUST:
;         a) Call OSTaskSwHook() then,
;         b) Set OSRunning to TRUE,
;         c) Switch to the highest priority task.
; *****

```

_OSStartHighRdy PROC FAR

```

        MOV    AX, SEG _OSTCBHighRdy    ; Reload DS
        MOV    DS, AX                  ;
;
        CALL   FAR PTR _OSTaskSwHook   ; Call user defined task switch hook
;

```



```

INC    BYTE PTR DS:_OSRunning ; Indicate that multitasking has started
;

LES    BX, DWORD PTR DS:_OSTCBHighRdy ; SS:SP = OSTCBHighRdy->OSTCBStkPtr
MOV    SS, ES:[BX+2]                ;
MOV    SP, ES:[BX+0]                ;
;

POP    DS                         ; Load task's context
POP    ES                         ;
POPA                           ;
;

IRET                           ; Run task

_OSStartHighRdy ENDP

;***** *****
;          PERFORM A CONTEXT SWITCH (From task level)
;          void OSCtxSw(void)

;

; Note(s): 1) Upon entry,
;           OSTCBCur points to the OS_TCB of the task to suspend
;           OSTCBHighRdy points to the OS_TCB of the task to resume

;

; 2) The stack frame of the task to suspend looks as follows:

;

;           SP -> OFFSET of task to suspend      (Low memory)
;           SEGMENT of task to suspend
;           PSW     of task to suspend      (High memory)
;

; 3) The stack frame of the task to resume looks as follows:

;

;           OSTCBHighRdy->OSTCBStkPtr --> DS           (Low memory)
;                           ES
;                           DI
;                           SI
;                           BP
;                           SP
;                           BX
;                           DX
;                           CX
;                           AX
;                           OFFSET of task code address
;                           SEGMENT of task code address
;                           Flags to load in PSW           (High memory)
;***** *****

```

```

(OSCtxSw PROC FAR
;
    PUSHAD          ; Save current task's context
    PUSH  ES          ;
    PUSH  DS          ;
;
    MOV   AX, SEG _OSTCBCur      ; Reload DS in case it was altered
    MOV   DS, AX          ;
;
    LES   BX, DWORD PTR DS:_OSTCBCur    ; OSTCBCur->OSTCBStkPtr = SS:SP
    MOV   ES:[BX+2], SS          ;
    MOV   ES:[BX+0], SP          ;
;
    CALL  FAR PTR _OSTaskSwHook      ; Call user defined task switch hook
;
    MOV   AX, WORD PTR DS:_OSTCBHighRdy+2 ; OSTCBCur = OSTCBHighRdy
    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  ; OSPrioCur = OSPrioHighRdy
    MOV   BYTE PTR DS:_OSPrioCur, AL        ;
;
    LES   BX, DWORD PTR DS:_OSTCBHighRdy  ; SS:SP = OSTCBHighRdy->OSTCBStkPtr
    MOV   SS, ES:[BX+2]          ;
    MOV   SP, ES:[BX]          ;
;
    POP   DS          ; Load new task's context
    POP   ES          ;
    POPAD          ;
;
    IRET          ; Return to new task
;
(OSCtxSw ENDP

;*****
;           PERFORM A CONTEXT SWITCH (From an ISR)
;           void OSIntCtxSw(void)

;
; Note(s): 1) Upon entry,
;           OSTCBCur points to the OS_TCB of the task to suspend
;           OSTCBHighRdy points to the OS_TCB of the task to resume
;
;           2) The stack frame of the task to suspend looks as follows:
```

```
; SP+0 --> OFFSET of return address of OSIntCtxSw() (Low memory)
; +2      SEGMENT of return address of OSIntCtxSw()
; +4      PSW saved by OS_ENTER_CRITICAL() in OSIntExit()
; +6      OFFSET of return address of OSIntExit()
; +8      SEGMENT of return address of OSIntExit()
; +10     DS
;          ES
;          DI
;          SI
;          BP
;          SP
;          BX
;          DX
;          CX
;          AX
;          OFFSET of task code address
;          SEGMENT of task code address
;          Flags to load in PSW           (High memory)
;
;-----3) The stack frame of the task to resume looks as follows.
```

```

LES BX, DWORD PTR DS:_OSTCBCur      ; OSTCBCur->OSTCBStkPtr = SS:SP
MOV ES:[BX+2], SS                  ;
MOV ES:[BX+0], SP                  ;

;
CALL FAR PTR _OSTaskSwHook        ; Call user defined task switch hook

;
MOV AX, WORD PTR DS:_OSTCBHighRdy+2 ; OSTCBCur = OSTCBHighRdy
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 ; OSPrioCur = OSPrioHighRdy
MOV BYTE PTR DS:_OSPrioCur, AL     ;

;
LES BX, DWORD PTR DS:_OSTCBHighRdy ; SS:SP = OSTCBHighRdy->OSTCBStkPtr
MOV SS, ES:[BX+2]                  ;
MOV SP, ES:[BX]                    ;

;
POP DS                           ; Load new task's context
POP ES                           ;
POPA                           ;

;
IRET                           ; Return to new task

;
(OSIntCtxSw ENDP

*****
;                                     HANDLE TICK ISR
;

; Description: This function is called 199.99 times per second or, 11 times faster than the normal
DOS
;           tick rate of 18.20648 Hz. Thus every 11th time, the normal DOS tick handler is called.
;           This is called chaining. 10 times out of 11, however, the interrupt controller on
the PC
;           must be cleared to allow for the next interrupt.
;

; Arguments : none
;

; Returns   : none
;

; Note(s)   : The following C-like pseudo-code describe the operation being performed in the code
below.
;

; Save all registers on the current task's stack;

```

```

; OSIntNesting++;
; OSTickDOSCtr--;
; if (OSTickDOSCtr == 0) {
;     INT 81H;           Chain into DOS every 54.925 mS
;                   (Interrupt will be cleared by DOS)
; } else {
;     Send EOI to PIC;   Clear tick interrupt by sending an End-Of-Interrupt to the
8259
;                           PIC (Priority Interrupt Controller)
;
;     OSTimeTick();      Notify uC/OS-II that a tick has occurred
;     OSIntExit();        Notify uC/OS-II about end of ISR
;     Restore all registers that were save on the current task's stack;
;     Return from Interrupt;
; ****
;

_OSTickISR PROC FAR
    PUSHA             ; Save interrupted task's context
    PUSH   ES
    PUSH   DS
;
    MOV    AX, SEG _OSTickDOSCtr       ; Reload DS
    MOV    DS, AX
;
    INC    BYTE PTR _OSIntNesting     ; Notify uC/OS-II of ISR
;
    DEC    BYTE PTR DS:_OSTickDOSCtr
    CMP    BYTE PTR DS:_OSTickDOSCtr, 0
    JNE    SHORT _OSTickISR1  ; Every 11 ticks (~199.99 Hz), chain into DOS
;
    MOV    BYTE PTR DS:_OSTickDOSCtr, 11
    INT    081H                 ; Chain into DOS's tick ISR
    JMP    SHORT _OSTickISR2
;

_OSTickISR1:
    MOV    AL, 20H               ; Move EOI code into AL.
    MOV    DX, 20H               ; Address of 8259 PIC in DX.
    OUT    DX, AL                ; Send EOI to PIC if not processing DOS timer.
;
_OSTickISR2:
    CALL   FAR PTR _OSTimeTick     ; Process system tick
;
    CALL   FAR PTR _OSIntExit      ; Notify uC/OS-II of end of ISR
;
    POP    DS                   ; Restore interrupted task's context

```

```

    POP    ES
    POPA

;

    IRET           ; Return to interrupted task
;

_OSTickISR ENDP
;

END

```

C.1 OS_CPU_C.C

```

/*
***** uC/OS-II *****

*          The Real-Time Kernel
*
*      (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*          All Rights Reserved
*
*
*          80x86/80x88 Specific code
*          LARGE MEMORY MODEL
*
* File : OS_CPU_C.C
* By   : Jean J. Labrosse
***** */

#define OS_CPU_GLOBALS
#include "includes.h"

/*
***** INITIALIZE A TASK'S STACK
*
* Description: This function is called by either OSTaskCreate() or OSTaskCreateExt() to initialize
*               the stack frame of the task being created. This function is highly processor specific.
*
* Arguments : task      is a pointer to the task code
*
*             pdata     is a pointer to a user supplied data area that will be passed to the task
*               when the task first executes.
*

```

```

*      ptos      is a pointer to the top of stack. It is assumed that 'ptos' points to
*      a 'free' entry on the task stack. If OS_STK_GROWTH is set to 1 then
*      'ptos' will contain the HIGHEST valid address of the stack. Similarly, if
*      OS_STK_GROWTH is set to 0, the 'ptos' will contains the LOWEST valid address
*      of the stack.

*
*      opt       specifies options that can be used to alter the behavior of OSTaskStkInit().
*      (see uCOS_II.H for OS_TASK_OPT_???).

*
* Returns : Always returns the location of the new top-of-stack' once the processor registers have
* been placed on the stack in the proper order.

*
* Note(s) : Interrupts are enabled when your task starts executing. You can change this by setting
* the PSW to 0x0002 instead. In this case, interrupts would be disabled upon task startup.
* The application code would be responsible for enabling interrupts at the beginning of the
* task code. You will need to modify OSTaskIdle() and OSTaskStat() so that they enable
* interrupts. Failure to do this will make your system crash!
*****
```

*/

```

void *OSTaskStkInit (void (*task)(void *pd), void *pdata, void *ptos, INT16U opt)
{
    INT16U *stk;

    opt = opt; /* 'opt' is not used, prevent warning */
```

*/

```

    stk = (INT16U *)ptos; /* Load stack pointer */
```

*/

```

    *stk-- = (INT16U)FP_SEG(pdata); /* Simulate call to function with argument */
```

*/

```

    *stk-- = (INT16U)FP_OFF(pdata);
    *stk-- = (INT16U)FP_SEG(task);
    *stk-- = (INT16U)FP_OFF(task);
    *stk-- = (INT16U)0x0202; /* SW = Interrupts enabled */
```

*/

```

    *stk-- = (INT16U)FP_SEG(task); /* Put pointer to task on top of stack */
```

*/

```

    *stk-- = (INT16U)FP_OFF(task);
    *stk-- = (INT16U)0xAAAA; /* AX = 0xAAAA */
```

*/

```

    *stk-- = (INT16U)0xCCCC; /* CX = 0xCCCC */
```

*/

```

    *stk-- = (INT16U)0xDDDD; /* DX = 0xDDDD */
```

*/

```

    *stk-- = (INT16U)0xB BBBB; /* BX = 0xBB BB */
```

*/

```

    *stk-- = (INT16U)0x0000; /* SP = 0x0000 */
```

*/

```

    *stk-- = (INT16U)0x1111; /* BP = 0x1111 */
```

*/

```

    *stk-- = (INT16U)0x2222; /* SI = 0x2222 */
```

*/

```

    *stk-- = (INT16U)0x3333; /* DI = 0x3333 */
```

*/

```

    *stk-- = (INT16U)0x4444; /* ES = 0x4444 */
```

*/

```

    *stk = _DS; /* DS = Current value of DS */
```

*/

```

    return ((void *)stk);
}
```

```
#if OS_CPU_HOOKS_EN
```

```

/*
***** TASK CREATION HOOK *****
* Description: This function is called when a task is created.
*
* Arguments : ptcb    is a pointer to the task control block of the task being created.
*
* Note(s)   : 1) Interrupts are disabled during this call.
***** */

void OSTaskCreateHook (OS_TCB *ptcb)
{
    ptcb = ptcb;           /* Prevent compiler warning */
}

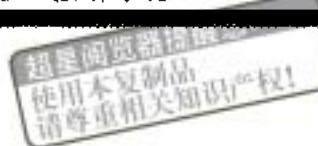
/*
***** TASK DELETION HOOK *****
* Description: This function is called when a task is deleted.
*
* Arguments : ptcb    is a pointer to the task control block of the task being deleted.
*
* Note(s)   : 1) Interrupts are disabled during this call.
***** */

void OSTaskDelHook (OS_TCB *ptcb)
{
    ptcb = ptcb;           /* Prevent compiler warning */
}

/*
***** TASK SWITCH HOOK *****
* Description: This function is called when a task switch is performed. This allows you to perform
*               other operations during a context switch.
*
* Arguments : none
*
* Note(s)   : 1) Interrupts are disabled during this call.
*               2) It is assumed that the global pointer 'OSTCBHighRdy' points to the TCB of the
*                  task that will be 'switched in' (i.e. the highest priority task) and, 'OSTCBCur'
*                  points to the task being switched out (i.e. the preempted task).
***** */

void OSTaskSwHook (void)

```



```
/*
*****
*          STATISTIC TASK HOOK
*
* Description: This function is called every second by uC/OS-II's statistics task. This allows
*               your application to add functionality to the statistics task.
*
* Arguments  : none
*****
*/
void OSTaskStatHook (void)
{

}

/*
*****
*          TICK HOOK
*
* Description: This Function is called every tick.
*
* Arguments  : none
*
* Note(s)    : 1) Interrupts may or may not be ENABLED during this call.
*****
*/
void OSTimeTickHook (void)
{
}
#endif
```

C.2 OS_CPU.H

```
/*
*****
*          uC/OS-II
*          The Real-Time Kernel
*
* (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*               All Rights Reserved
*
*          80x86/80x88 Specific code
*          LARGE MEMORY MODEL
```



```

/*
 * File : OS_CPU.H
 * By   : Jean J. Labrosse
 ****
 */
#ifndef OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT extern
#endif

/*
 ****
 *          DATA TYPES
 *          (Compiler Specific)
 ****
 */

typedef unsigned char BOOLEAN;
typedef unsigned char INT8U;           /* Unsigned 8 bit quantity */
typedef signed  char INT8S;           /* Signed   8 bit quantity */
typedef unsigned int INT16U;          /* Unsigned 16 bit quantity */
typedef signed  int INT16S;           /* Signed   16 bit quantity */
typedef unsigned long INT32U;          /* Unsigned 32 bit quantity */
typedef signed  long INT32S;           /* Signed   32 bit quantity */
typedef float      FP32;             /* Single precision floating point */
typedef double     FP64;              /* Double precision floating point */

typedef unsigned int OS_STK;           /* Each stack entry is 16-bit wide */

#define BYTE      INT8S /* Define data types for backward compatibility ... */
#define UBYTE     INT8U /* ... to uC/OS V1.xx. Not actually needed for ... */
#define WORD      INT16S /* ... uC/OS-II. */
#define UWORD     INT16U
#define LONG      INT32S
#define ULONG     INT32U

/*
 ****
 *          Intel 80x86 (Real-Mode, Large Model)
 *
 * Method #1: Disable/Enable interrupts using simple instructions. After critical section, interrupts
 * will be enabled even if they were disabled before entering the critical section. You MUST
 * change the constant in OS_CPU_A.ASM, function OSIntCtxSw() from 10 to 8.
 */

```

```
/*
 * Method #2: Disable/Enable interrupts by preserving the state of interrupts. In other words, if
 *             interrupts were disabled before entering the critical section, they will be disabled when
 *             leaving the critical section. You MUST change the constant in OS_CPU_A.ASM, function
 *             OSIntCtxSw() from 8 to 10.
 ****
 */
#define OS_CRITICAL_METHOD    2

#if      OS_CRITICAL_METHOD == 1
#define  OS_ENTER_CRITICAL()  asm CLI          /* Disable interrupts      */
#define  OS_EXIT_CRITICAL()   asm STI          /* Enable  interrupts     */
#endif

#if      OS_CRITICAL_METHOD == 2
#define  OS_ENTER_CRITICAL()  asm {PUSHF; CLI}  /* Disable interrupts      */
#define  OS_EXIT_CRITICAL()   asm POPF         /* Enable  interrupts     */
#endif

/*
 ****
 *           Intel 80x86 (Real-Mode, Large Model) Miscellaneous
 ****
 */

#define  OS_STK_GROWTH     1  /* Stack grows from HIGH to LOW memory on 80x86 */

#define  uCOS              0x80  /* Interrupt vector # used for context switch */

#define  OS_TASK_SW()        asm INT  uCOS

/*
 ****
 *           GLOBAL VARIABLES
 ****
 */

OS_CPU_EXT INT8U OSTickDOSCtr; /* Counter used to invoke DOS's tick handler every 'n' ticks */
```

附录 D



HPLISTC 和 TO

HPLISTC 和 TO 是 DOS 下的两个工具程序，为了让用户用起来方便，笔者提供了源代码和可执行代码。

D.0 HPLISTC

HPLISTC 用于在 HP 激光打印机（HP Laserjet）上以压缩模式（每英寸打印 17 个字符）打印 C 语言的源代码文件。在 8.5×11 英寸的纸上（竖式），每行可以打印 132 个字符，用横式可以打印 175 个字符。打印完成以后，打印机回到原来默认的打印模式。

DOS 下的可执行代码是\SOFTWARE\HPLISTC\EXEHPLISC.EXE 源程序见\SOFTWARE\HPLISTC\SOURCE\HPLIST.C。

在每页的页眉上打印当前日期和时间、文件名、扩展名和页号。是否在页眉上打印标题，也可以通过选择项设置。HPLISTC 在打印源代码时寻找两个特别的注释：/*\$TITLE*/ 或 /*\$title*/ 以及 /*\$PAGE*/ 或 /*\$page*/。/*\$TITLE= */ 定义打印格式中的标题，这个标题印在每页的第二行上。从下一页起，打印在页眉上的标题。写法如下：

```
/*$TITLE=Matrox Keyboard Driver*/
```

这样将从下页起在页眉上打印标题 “*Matrox Keyboard Driver*”，直到下次重新设置为止。注释/*PAGE*/强制在此分页。HPLISTC 在打印时不分页，除非注释中出现/*PAGE*/。如果不使用这个注释，打印机在达到打印机每页允许打印的最多行数之后由打印机控制分页，每页纸的页码号打印在页眉上，页码的计算是靠/*PAGE*/注释完成的。

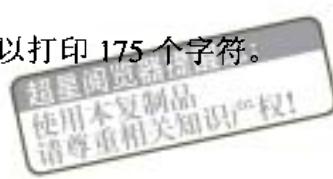
HPLESTC 允许在每一行的前面打印行号，以便阅读，还可以用模式打印，命令格式如下：

```
HPLISTC filename.ext [L | 1] [destination]
```

这里 filename.ext 是拟打印文件的文件名，destination 是输出设备，因为 HPLESTC 是

向标准输出设备输出的，可用输入输出再定向，定向给一个文件，或者是定向给打印机（PRN, LPT1, LPT2），或者定向给串行口（COM1, COM2），办法是在 DOS 下用再定向符 > ； HPLISTC，默认的输出设备是监视器。

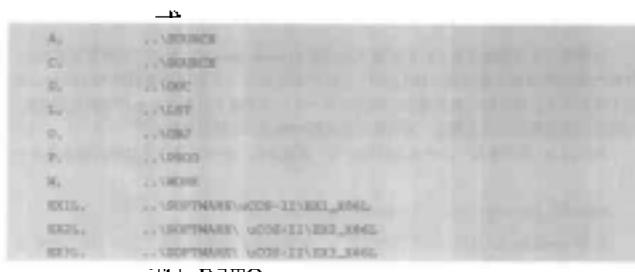
[L|l]意为 L 或者 l，表示使用横模式打印，每行可以打印 175 个字符。



D.1 TO

TO 是 DOS 下的工具程序，可以使用户变更路径时不必使用：

CD path



命令，因为这条命令使得目录之间的换来换去变

就能迅速进入到想去的目录，这里用路径的代名词 name 表示想去的目录。路径和该路径的代名词是放在 ASCII 码文件 TO.TBL 中的，放在当前驱动器的根目录下。TO 扫描文件 TO.TBL，以找到命令行中的代名词，如果代名词存在于文件 TO.TBL 之中，则变换到相应路径，如果代名词在 TO.TBL 中找不到，则显示 Invalid Name。

在 DOS 下的可执行文件在\SOFTWARE\TO\EXETO.EXE 中，路径以及路径的代名词的范例文件是\SOFTWARE\TO\EXETO.TBL，TO 命令的源码见\SOFTWARE\TO\SOURCE\TO.C。TO.TBL 文件的范例和格式如程序清单 D.1 所示。

程序清单 D.1 TO.TBL 文件的范例

```
TABLE, ..\SOFTWARE\WOWENTI\HPLISTC  
APLICTC, ..\SOFTWARE\HPLISTC\SOURCE  
TO, ..\SOFTWARE\UOS-11\SOURCE  
uOS-11, ..\SOFTWARE\UOS-11\SOURCE
```

HPLISPC 和 TO



给 TO.TBL 增加一个路径代名词及路径可以键入：

```
TO name path
```

这里 name 是拟去路径 path 的代名词。这样就免去了重新编辑 TO.TBL 这个文件。如果键入：

```
TO EXIL
```

就进入 \SOFTWARE\UOS-11\EX1_X86L 这个子目录。同样，如果键入：

```
TO hplistc
```

就进入到\SOFTWARE\HPLISTC\SOURCE 子目录。TO.TBL 文件可以很长，但不得重名。不同名的两个代名词可以指向同一个子目录。如果用文本编辑来写 TO.TBL 文件，则所有字母要大写。因为在 DOS 下，键入的命令被转换成了大写。TO 扫描 TO.TBL 文件时，是从头扫描的，为了扫描得快，最常用的子目录和代名词应尽量往前放。

超全网最
使用本复制品
请尊重相关知识产权!

附录 E 参考文献

- Lehoczky, John, Lui Sha, and Ye Ding. 1989. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In: *Proceedings of the IEEE Real-Time Systems Symposium.*, Los Alamitos, California. Piscataway, New Jersey: IEEE Computer Society, p. 166~171.
- Madnick, E. Stuart and John J. Donovan. 1974. *Operating Systems*. New York: McGraw-Hill. ISBN 0-07-039455-5.
- Ripps, David L. 1989. *An Implementation Guide To Real-Time Programming*. Englewood Cliffs, New Jersey: Yourdon Press. ISBN 0-13-451873-X.
- Savitzky, Stephen R. 1985. *Real-Time Microprocessor Systems*. New York: Van Nostrand Reinhold. ISBN 0-442-28048-3.
- Wood, Mike and Tom Barrett . 1990. A Real-Time Primer. *Embedded Systems Programming*, February, p. 20~28.

附录 F



使用许可证 (License) 和 μCOS-II 网站

μCOS-II 的源码和目标码可以在有资质的大学中免费提供给学生，用于非商业性目的。换句话说，μCOS-II 用于教学目的，不需要使用许可证。

如果以盈利为目的，将 μCOS-II 的目标代码嵌入到产品中去，则应得到“目标代码销售许可证”。这是要付钱的，具体价格可用以下地址与作者联系。

如果销售 μCOS-II 的源码，也应得到“源代码销售许可证”，具体价格可与作者联系：

Jean.Labrosse@μCOS-II.com

或者：

Jean J. Labrosse
9540 NW 9th Court
Plantation, FL 33324
U.S.A.
1-954-472-5094 (电话)
1-954-472-7779 (传真)

μCOS-II 网站 (www.μCOS-II.com) 包括：

- ◆ 关于 μCOS 和 μCOS-II 的新闻
- ◆ 维护
- ◆ 移植
- ◆ 常遇问题解答
- ◆ 应用文章
- ◆ 关于嵌入式实时系统的书籍
- ◆ 近期培训班
- ◆ 相关网站

光盘上的内容

本书所附光盘包含μCOS-II 的源代码（大部分用 C 语言编写）、可执行文件（放在/source code 路径下）和 www.uCOS-II.com 网站的最新完全镜像。其中可执行文件由 Borland C++ v3.1 编译。如果用 Microsoft C/C++ 编译，应调整 SP 的偏移常数，并禁用堆栈监测（参见 Read.me 文件）。

浏览网站镜像时，在光盘根目录下找到 index.html 文件，并用 Web 浏览器打开即可。



Powered by xiaoguo's publishing studio
QQ:8204136