第一章：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/AMD80186 处理器（大模式下编译）的代码。这些代码实际上是在 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。

1.00 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。是假定读者在 80x86，Pentium，或者 Pentium-II 处理器上运行 DOS 或 Windows95。至少需要 5Mb 硬盘空间来安装 uC/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』是读者想要将μC/OS－II 安装的目标磁盘的盘符。

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 并且运行。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\EX1_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一下子目录：

- **\SOFTWARE**
  这是根目录，是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，HPLISTC 和 TO)。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，HPLISTC 和 TO)。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 1.07，例 1)，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 1.08，例 2)，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 1.09，例 3)，可以在 DOS（或 Windows 95 下的 DOS 窗口）下运行。

- \SOFTWARE\uCOS-II\Ix86L
  这个目录下包括依赖于处理器类型的代码。此时是为在 80x86 处理器上运行 uC/OS-II 而必须的一些代码，实模式，在大模式下编译。

- \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它架构的处理器上。

1.01 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：

```
#include "includes.h"
```

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要花费额外的时间。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-II\EX1_x86L ， \SOFTWARE\uCOS-II\EX2_x86L ， 以及 \SOFTWARE\uCOS-II\EX3_x86L 中。当然可以重新编辑 INCLUDES.H 以添加用户自己的头文件。

1.02 不依赖于编译的数据类型

因为不同的微处理器有不同的字长，μC/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-II\Ix86L\OS_CPU.H，它是针对 80x86 的实模式，在大模式下编译）。μCOS-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，如表 L1.1 所示。为了方便起见，还定义了浮点数数据类型，虽然μC/OS-II 中没有使用浮点

数。

程序清单 L1.1 可移植型数据类型。

```
Typedef unsigned char  BOOLEAN;
Typedef unsigned char  INT8U;
Typedef signed   char  INT8S;
Typedef unsigned int   INT16U;
Typedef signed   int   INT16S;
Typedef unsigned long  INT32U;
Typedef signed   long  INT32S;
Typedef float         FP32;
Typedef double        FP64;

#define BYTE          INT8S
#define UBYTE         INT8U
#define WORD          INT16S
#define UWORD         INT16U
#define LONG          INT32S
#define ULONG         INT32U
```

以 INT16U 数据类型为例，它代表 16 位无符号整数数据类型。μC/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 65,535。如果将μCO/S-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。然而将无论μC/OS-II 用到哪里,都会当作 INT16U 处理。表 1.1 是以 Borland C/C++编译器为例,为 80x86 提供的定义语句。为了和μC/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μC/OS 的代码移植到μC/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请

1.03 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 文件中定义。这种重复的定义很容易导致错误。以下讨论的方法只需用在头文件中定义一次。虽然有点不易懂,但用户一旦掌握，使用起来却很灵活。表 1.2 中的定义出现在定义所有全局变量的.H 头文件中。

程序清单 L 1.2 定义全局宏。

```
#ifdef   xxx_GLOBALS
```

```
#define  xxx_EXT
#else
#define  xxx_EXT extern
#endif
```

.H 文件中每个全局变量都加上了 xxx_EXT 的前缀。xxx 代表模块的名字。该模块的.C 文件中有以下定义：

```
#define  xxx_GLOBALS
#include "includes.h"
```

当编译器处理.C 文件时，它强制 xxx_EXT（在相应.H 文件中可以找到）为空，（因为 xxx_GLOBALS 已经定义)。所以编译器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部全局变量。为了说明这个概念，可以参见 uC/OS_II.H,其中包括以下定义：

```
#ifdef   OS_GLOBALS
#define  OS_EXT
#else
#define  OS_EXT extern
#endif

OS_EXT  INT32U      OSIdleCtr;
OS_EXT  INT32U      OSIdleCtrRun;
OS_EXT  INT32U      OSIdleCtrMax;
```

同时，uCOS_II.H 有中以下定义：

```
#define  OS_GLOBALS
#include “includes.h”
```

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

```
INT32U      OSIdleCtr;
INT32U      OSIdleCtrRun;
INT32U      OSIdleCtrMax;
```

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。

```
extern INT32U      OSIdleCtr;
extern INT32U      OSIdleCtrRun;
extern INT32U      OSIdleCtrMax;
```

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。这样的就只需在 .H 文件中定义一次就可以了。

1.04 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()

用户会看到，调用 OS_ENTER_CRITICAL()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段代码。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。9.03.02 节详细讨论定义这些宏的两种方法。

程序清单 L1.3 进入正确部分的宏。

```
#define  OS_CRITICAL_METHOD   2


#if     OS_CRITICAL_METHOD == 1
#define  OS_ENTER_CRITICAL()  asm  CLI
#define  OS_EXIT_CRITICAL()   asm  STI
#endif


#if     OS_CRITICAL_METHOD == 2
#define  OS_ENTER_CRITICAL()  asm {PUSHF; CLI}
#define  OS_EXIT_CRITICAL()   asm  POPF
#endif
```

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林阶段代码。

1.05 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。

1.05.01 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 2,000 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：

```
PC_DispClrScr()     Clear the screen
PC_DispClrLine()    Clear a single row (or line)
PC_DispChar()       Display a single ASCII character anywhere on the screen
PC_DispStr()        Display an ASCII string anywhere on the screen
```

1.05.02 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。表 1.4 说明了如何测量 PC_DisplayChar()的执行时间。注意，时间是以 uS 为单位的。

程序清单 L1.4 测量代码执行时间。

```
INT16U time;


PC_ElapsedInit();

.

.

PC_ElapsedStart();

PC_DispChar(40, 24, ‘A’, DISP_FGND_WHITE);

time = PC_ElapsedStop();
```

1.05.03 其他函数

μC/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μC/OS-II 是多任务，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μC/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，多数其它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 WINDOWS95 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 SACII 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，其它 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，其它 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 μC /OS-II 定义频率，以改变钟节拍的速率。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 65,535 的结果。如果初始化为 58,659，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(实际是上是 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 中最后两个函数是得到和设置中断向量，笔者是用 Borland C/C++中的库函数来完成的，但是 PC_VectGet()和 PC_VectSet()很容易改写，以适用于其它编译器。

1.06 应用 μC/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 Windows95 的 DOS 窗口下编译的。可执行代码可以在每个范例的 OBJ 子目录下找到。实际上这些代码是在 Borland IDE (Integrated Development Environment)下编译的，编译时的选项如表 1.1 所示：

表 T1.1　IDE 中编译选项。

Code generation	
Model	: Large
Options	: Treat enums as ints
Assume SS Equals DS	: Default for memory model
Advanced code generation	
Floating point	: Emulation
Instruction set	: 80186
Options	: Generate underbars
	Debug info in OBJs
	Fast floating point
Optimizations	
Optimizations	Global register allocation
	Invariant code motion

	Induction variables
	Loop optimization
	Suppress redundant loads
	Copy propagation
	Dead code elimination
	Jump optimization
	In-line intrinsic functions
Register variables	Automatic
Common subexpressions	Optimize globally
Optimize for	Speed

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 的提示下改变 IDE 的路径。

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#defines OS_???_EN 为 0 来实现。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。

1.07 例 1

第一个范例可以在\SOFTWARE\uCOS_II\EX1_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 个其它任务。TaskStart()任务是在函数 main()中建立的；它的功能是建立其它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；

TaskStart()还检查是否按下 ESC 键，以决定是否返回到 DOS。

其余 10 个任务基于相同的代码——Task();每个任务在屏幕上随机的位置显示一个 0 到 9 的数字。

1.07.01 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[L1.5(1)]。注意，笔者定义了白色的字符和黑色的背景色。既然要请屏幕，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() [L1.5(2)]。它会建立两个任务：

空闲任务和统计任务，前者在没有其它任务处于就绪态时运行；后者计算 CPU 的利用率。

程序清单 L1.5 main().

```
void main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);                (1)
    OSInit();                                                        (2)
    PC_DOSSaveReturn();                                              (3)
    PC_VectSet(uCOS, OSCtxSw);                                       (4)
    RandomSem = OSSemCreate(1);                                      (5)
    OSTaskCreate(TaskStart,                                          (6)
                 (void *)0,
                 (void *)&TaskStartStk[TASK_STK_SIZE-1],
                 0);
    OSStart();                                                       (7)
}
```

当前 DOS 环境是通过调用 PC_DOSSaveReturn()[L1.5(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 L1.6 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[L1.6(1)]，说明用户不是要返回 DOS，然后初始化 OSTickDOSCtr 为 1[L1.6(2)]，因为这个变量将在 OSTickISR()中递减，而 0 将使得这个变量在 OSTickISR()中减 1 后变为 255。然后，PC_DOSSaveReturn()将 DOS 的时钟节拍处理（tick handler）存入一个自由向量表入口中[L1.6(3)-(4)]，以便为μC/OS-II 的时钟节拍处理所调用。接着 PC_DOSSaveReturn()调用 jmp（）[L1.6(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使得程序返回到 PC_DOSSaveReturn()并且在调用 setjmp（）之后立即执行。因为 PC_ExitFlag 被初始化为 FALSE[L1.6(1)]。PC_DOSSaveReturn()跳过 if 状态语句 [L1.6(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 L 1.7)，它设置 PC_ExitFlag 为 TRUE，并且执行 longjmp（）语句[L1.7(2)],这时处理器将跳回 PC_DOSSaveReturn()[在调用 setjmp()之后] [L1.6(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[L1.6(6)]，恢复 PC 时钟节拍中断服务[L1.6(7)]，清屏幕[L1.6(8)]，通过 exit(0)返回 DOS [L1.6(9)]。

程序清单 L1.6 保存DOS环境。.

```
void PC_DOSSaveReturn (void)
{
    PC_ExitFlag  = FALSE;                                            (1)
```

```
    OSTickDOSCtr =    8;                                          (2)
    PC_TickISR   = PC_VectGet(VECT_TICK);                         (3)

    OS_ENTER_CRITICAL();
    PC_VectSet(VECT_DOS_CHAIN, PC_TickISR);                       (4)
    OS_EXIT_CRITICAL();

    Setjmp(PC_JumpBuf);                                           (5)
    if (PC_ExitFlag == TRUE) {
        OS_ENTER_CRITICAL();
        PC_SetTickRate(18);                                       (6)
        PC_VectSet(VECT_TICK, PC_TickISR);                        (7)
        OS_EXIT_CRITICAL();
        PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);         (8)
        exit(0);                                                  (9)
    }
}
```

程序清单 L1.7 设置返回DOS 。

```
void PC_DOSReturn (void)
{
    PC_ExitFlag = TRUE;                                           (1)
    longjmp(PC_JumpBuf, 1);                                       (2)
}
```

现在回到 main （） 这个函数，在程序清单 L 1.5 中，main （） 调用 PC_VectSet()来设置μCOS-II 中的 CPU 寄存器切换。任务级的 CPU 寄存器切换由 80x86 INT 指令来分配向量地址。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++库中的产生随机数的函数[L1.5(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[L1.5(6)]，在启动多任务 OSStart()之前用户至少要先建立一个任务,这一点非常重要[L1.5(7)]。不这样做用户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它任务运行。如果用户在启动多任务之前要建立其它任务，必须保证用户的任务代码监控全局变量 OSStatRdy 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。

1.07.02 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 L 1.8 所示。TaskStart()首先在屏幕顶端显示一个标识，说明这是例 1 [L1.8(1)]。然后关中断，以改变中断向量，让其指向μC/OS-II 的时钟节拍处理，而后，改变时钟节拍率，从 DOS 的 18.2Hz 变为 200Hz [L1.8(3)]。在处理器改变中断向量时以及系统没有完全初始化前，当然不希望有中断打入！注意 main () 这个函数(见程序清单 L 1.5)在系统初始化的时候并没有将中断向量设置成μC/OS-II 的时钟节拍处理程序，做嵌入式应用时，用户必须在第一个任务中打开时钟节拍中断。

程序清单 L 1.8 建立其它任务的任务。

```
void TaskStart (void *data)
{
    Prevent compiler warning by assigning ‘data’ to itself;
    Display banner identifying this as EXAMPLE #1;                    (1)

    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                                      (2)
    PC_SetTickRate(200);                                              (3)
    OS_EXIT_CRITICAL();

    Initialize the statistic task by calling ‘OSStatInit()’;          (4)

    Create 10 identical tasks;                                        (5)

    for (;;) {
        Display the number of tasks created;
        Display the % of CPU used;
        Display the number of task switches in 1 second;
        Display uC/OS-II’s version number
        If (key was pressed) {
            if (key pressed was the ESCAPE key) {
                PC_DOSReturn();
            }
        }
        Delay for 1 Second;
    }
```

```
}
```

在建立其他任务之前，必须调用 OSStatInit()[L1.8(4)]来确定用户的 PC 有多快，如程序清单 L1.9 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[L1.9(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 [L1.9(2)]，并产生另一个延时，这个延时使 OSStatInit()挂起。此时，uCOS-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）。空闲任务是一个无线的循环，它不断的递增 OSIdleCtr[L1.9(3)]。1 秒以后，uCOS-II 重新开始 OSStatInit()，并且将 OSIdleCtr 保存在 OSIdleMax 中[L1.9(4)。所以 OSIdleMax 是 OSIdleCtr 所能达到的最大值。而当用户再增加其他应用代码时，空闲任务就不会占用那样多的 CPU 时间。OSIdleCtr 不可能达到那样多的记数，（如果拥护程序每秒复位一次 OSIdleCtr）CPU 利用率的计算由μC/OS-II 中的 OSStatTask()函数来完成，这个任务每秒执行一次。而当 OSStatRdy 置为 TRUE[L1.9(5)],表示μC/OS-II 将统计 CPU 的利用率。

程序清单 L1.9 测试CPU速度。

```
void OSStatInit (void)
{
    OSTimeDly(2);                                                    (1)
    OS_ENTER_CRITICAL();
    OSIdleCtr    = 0L;                                               (2)
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC);                                     (3)
    OS_ENTER_CRITICAL();
    OSIdleCtrMax = OSIdleCtr;                                        (4)
    OSStatRdy    = TRUE;                                             (5)
    OS_EXIT_CRITICAL();
}
```

1.07.03 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 L1.10 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 L1.10(1)]（也就是禁止其他任务运行这段代码—译者注），然后调用

Borland C/C++的库函数 random（）来获得一个随机数[程序清单 L1.10(2)]，此处设 random（）函数是不可重入的，所以 10 个任务将轮流获得信号量，并调用该函数。当计算出 x 和 y 坐标后[程序清单 L1.10(3)]，任务释放信号量。随后任务在计算的坐标处显示其任务号（0-9，任务建立时的标识）[程序清单 L1.10(4)]。最后，任务延时一个时钟节拍[程序清单 L1.10(5)]，等待进入下一次循环。系统中每个任务每秒执行 200 次，10 个任务每秒钟将切换 2000 次。

程序清单 L1.10 在屏幕上显示随机位置显示数字的任务。

```
void Task (void *data)
{
    UBYTE x;
    UBYTE y;
    UBYTE err;



    for (;;) {
        OSSemPend(RandomSem, 0, &err);                                  (1)
        x = random(80);                                                 (2)
        y = random(16);
        OSSemPost(RandomSem);                                           (3)
        PC_DispChar(x, y + 5, *(char *)data, DISP_FGND_LIGHT_GRAY);     (4)
        OSTimeDly(1);                                                   (5)
    }
}
```

1.08 例 2

例 2 使用了带扩展功能的任务建立函数 OSTaskCreateExt（）和 uCOS-II 的堆栈检查操作（要使用堆栈检查操作必须用 OSTaskCreateExt（）建立任务—译者注）。当用户不知道应该给任务分配多少堆栈空间时，堆栈检查功能是很有用的。在这个例子里，先分配足够的堆栈空间给任务，然后用堆栈检查操作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。如果系统对稳定性要求高，则应该多一倍以上。

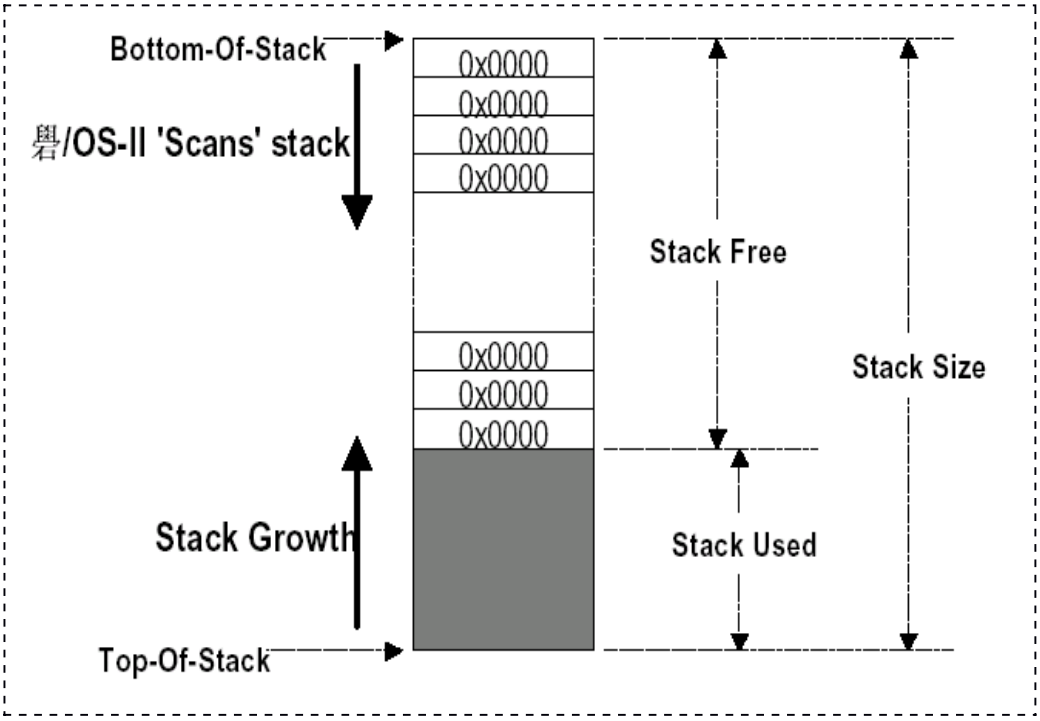
uCOS-II 的堆栈检查功能要求任务建立时堆栈清零。OSTaskCreateExt()可以执行此项操作（设置选项 OS_TASK_OPT_STK_CHK 和 OS_TASK_OPT_STK_CLR 打开此项操作）。如果任务运行过程中要进行建立、删除任务的操作，应该设置好上述的选项，确保任务建立后堆栈是清空的。同时要意识到 OSTaskCreateExt()进行堆栈清零操作是一项很费时的工作，而且取决于

堆栈的大小。执行堆栈检查操作的时候，uCOS-II 从栈底向栈顶搜索非 0 元素(参看图 F 1.1)，同时用一个计数器记录 0 元素的个数。

例 2 的磁盘文件为\SOFTWARE\uCOS-II\EX2_x86L，它包含 9 个任务。加上 uCOS-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：

- 每秒种任务切换的次数；
- CPU 利用率的百分比；
- 当前日期和时间；
- uCOS_II 的版本号；

图F 1.1　　μC/OS-II stack checking.



1.08.01 main()

例 2 的 main()函数和例 1 的看起来差不多（参看程序清单 L1.11），但是有两处不同。第一，main()函数调用 PC_ElapsedInit()[程序清单 L1.11(1)]来初始化定时器记录 OSTaskStkChk()的执行时间。第二，所有的任务都使用 OSTaskCreateExt()函数来建立任务[程序清单 L1.11(2)]（替代老版本的 OSTaskCreate（）），这使得每一个任务都可进行堆栈检查。

程序清单 L 1.11　例2中的Main（）函数.

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

```
    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);
    PC_ElapsedInit();                                                    (1)
    OSTaskCreateExt(TaskStart,                                           (2)
                    (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();
}
```

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,和一个用于指定对任务操作的变量。该变量的一个选项就是用来设定 uCOS-II 堆栈检查是否允许。例 2 中并没有用到 TCB 扩展数据结构指针。

1.08.02TaskStart()

程序清单 L1.12 列出了 TaskStart()的伪码。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 L1.12(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 L 1.12　TaskStart()的伪码。.

```
void TaskStart (void *data)
{
    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 task;
    Create 2 mailboxes which are used by Task #4 and #5;                 (1)
    Create a task that will display the date and time on the screen;     (2)
    Create 5 application tasks;
```

```
    for (;;) {
        Display #tasks running;
        Display CPU usage in %;
        Display #context switches per seconds;
        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 1 second;
    }
}
```

1.08.03 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 L1.13(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 L1.13(3)]（间隔 100ms）。

程序清单 L1.13 例2，任务1

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



    pdata = pdata;
    for (;;) {
        for (i = 0; i < 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 L1.14 所示的任务 2 在屏幕上显示一个顺时针旋转的指针（用横线，斜线等字符表示—译者注），每 200ms 旋转一格。

程序清单 L1.14 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);
        PC_DispChar(70, 15, '-',  DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
        PC_DispChar(70, 15, '\\', DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
    }
}
```

任务 3(程序清单 L1.15)也显示了与任务 2 相同的一个旋转指针，但是旋转的方向不同。任务 3 在堆栈中分配了一个很大的数组，将堆栈填充掉，使得 OSTaskStkChk()只需花费很少的时间来确定堆栈的利用率，尤其是当堆栈已经快满的时候。

程序清单 L1.15 任务3

```
void  Task3 (void *data)
{
   char   dummy[500];
   INT16U  i;
   data = data;
   for (I = 0; i < 499; i++) {
      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);
      OSTimeDly(20);
   }
}
```

任务 4(程序清单 L1.16)向任务 5 发送消息并等待确认[程序清单 L1.16(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 L1.16(2)],就将传递的 ASCII 码加 1 再发送[程序清单 L1.16(3)]，结果是不断的传送“ABCDEFG....”。

程序清单 L1.16 任务4

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



   data  = data;
   txmsg = 'A';
   for (;;) {
```

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

当任务 5 [程序清单 L1.17]接收消息后[程序清单 L1.17(1)]（发送的字符），就将消息显示到屏幕上[程序清单 L1.17(2)]，然后延时 1 秒[程序清单 L1.17(3)]，再向任务 4 发送确认信息。

程序清单 L1.17 任务5

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



    data = data;
    for (;;) {
        rxmsg = (char *)OSMboxPend(TxMbox, 0, &err);                     (1)
        PC_DispChar(70, 18, *rxmsg, DISP_FGND_YELLOW+DISP_BGND_RED);     (2)
        OSTimeDlyHMSM(0, 0, 1, 0);                                       (3)
        OSMboxPost(AckMbox, (void *)1);                                  (4)
    }
}
```

TaskClk()函数[程序清单 L1.18]显示当前日期和时间，每秒更新一次。

程序清单 L1.18 时钟显示任务

```
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);
    }
}
```

1.09 例 3

例 3 中使用了许多 uCOS-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（statistic task ），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。

1.09.01 main()

main()函数[程序清单 L1.19]和例 2 中的相不多，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 L1.19(1)]（扩展结构的声明在 INCLUDES.H 中定义，也可参看程序清单 L1.20)。笔者定义了 30 个字节来存放任务名（包括空格）[程序清单 L1.20(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 L1.19(2)]。

程序清单 L1.19 例3的main()函数

```
void main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);
    OSInit();
    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);
    PC_ElapsedInit();

    Strcpy(TaskUserData[TASK_START_ID].TaskName, "StartTask");        (1)
    OSTaskCreateExt(TaskStart,
                    (void *)0,
                    &TaskStartStk[TASK_STK_SIZE-1],
```

```
                TASK_START_PRIO,
                TASK_START_ID,
                &TaskStartStk[0],
                TASK_STK_SIZE,
                &TaskUserData[TASK_START_ID],
                0);                                                    (2)
    OSStart();
}
```

程序清单 L1.20 TCB扩展数据结构。

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

1.09.02 任务

TaskStart()的伪码如程序清单 L1.21 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 L1.21(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 L1.21(2)]；
- 禁止堆栈检查。

程序清单 L1.21 TaskStart()的伪码。

```
void TaskStart (void *data)
{
    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 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 TCB ext.;     (2)
    for (;;) {
```

```
        Display #tasks running;
        Display CPU usage in %;
        Display #context switches per seconds;
        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 1 second;
    }
}
```

任务 1 向消息队列发送一个消息[程序清单 L1.22(1)]，然后延时等待消息发送完成[程序清单 L1.22(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 L1.22 任务1。

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



    data = data;
    for (;;) {
        OSQPost(MsgQueue, (void *)&one);                                    (1)
        OSTimeDlyHMSM(0, 0, 1,   0);                                        (2)
        OSQPost(MsgQueue, (void *)&two);
        OSTimeDlyHMSM(0, 0, 0, 500);
        OSQPost(MsgQueue, (void *)&three);
        OSTimeDlyHMSM(0, 0, 1,   0);
    }
}
```

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 L1.23(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 L1.23(2)]，延时的时间可以使任务 3 检查消息队列。

程序清单 L1.23 任务2。

```
void  Task2 (void *data)
{
    INT8U *msg;
    INT8U  err;



    data = data;
    for (;;) {
        msg = (INT8U *)OSQPend(MsgQueue, 0, &err);                    (1)
        PC_DispChar(70, 14, *msg, DISP_FGND_YELLOW+DISP_BGND_BLUE);   (2)
        OSTimeDlyHMSM(0, 0, 0, 500);                                  (3)
    }
}
```

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 L1.24(1)]。如果有消息来到，任务 3 将显示消息号[程序清单 L1.24(3)]，如果超过了等待时间，任务 3 就显示“T”(意为 timeout) [程序清单 L1.24(2)]。

程序清单 L1.24任务3

```
void  Task3 (void *data)
{
    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 L1.25(1)]和接收[程序清单 L1.25(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 L1.25(3)]。

程序清单 L1.25 任务4。

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



    data = data;
    mbox = OSMboxCreate((void *)0);
    for (;;) {
        OSMboxPost(mbox, (void *)1);                                    (1)
        OSMboxPend(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 L1.26(1)]。注意所有的任务都应该调用 uCOS-II 的函数，等待延时结束或者事件的发生而让出 CPU。如果始终占用 CPU，这将使低优先级的任务无法得到 CPU。

程序清单 L1.26 任务5。

```
void  Task5 (void *data)
{
    data = data;
    for (;;) {
        OSTimeDly(1);                                                   (1)
    }
}
```

同样， TaskClk()函数[程序清单 L1.18]显示当前日期和时间。

1.09.03 注意

有些程序的细节只有请您仔细读一读 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 L1.27(1)] 来获取任务的运行时间[程序清单 L1.27(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 L1.27(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 L1.27(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 L1.27(4)],其中的计数器 TaskCtr 进行累加[程序清单 L1.27(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 L1.27(6)]用来记录函数从切入到切出的运行时间，TaskTotExecTime[程序清单 L1.27(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 L1.27 用户定义的OSTaskSwHook()

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



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

本例中的统计任务（statistic task）将调用对外接口函数 OSTaskStatHook()（设置 OS_CFG.H 文件中的 OS_TASK_STAT_EN 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 L1.28(1)]，DispTaskStat()用来将数字显示为 ASCII 字符[程序清单 L1.28(2)]。然后是计算每个任务运行时间的百分比[程序清单 L1.28(3)]，显示在合适的位置上 [程序清单 L1.28(4)]。

程序清单 L1.28 用户定义的OSTaskStatHook().

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



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

第 2 章 实时系统概念

实时系统的特点是，如果逻辑和时序出现偏差将会引起严重后果的系统。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广泛的领域，而多数实时系统又是嵌入式的。这意味着计算机建在系统内部，用户看不到有个计算机在系统里面。以下是一些嵌入式系统的例子：

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

通讯类
- Switch Hurb
- 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
- 洗依机
- 稳温调节器

实时应用软件的设计一般比非实时应用软件设计难一些。本章讲述实时系统概念。

2.0 前后台系统 （Foreground/Background System）

不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-Loops)。应用程序是一个无限的循环，循环中调用相应的函数完成相应的操作，这部分可以看成后台行为(background)。中断服务程序处理异步事件，这部分可以看成前台行为（foreground）。后台也可以叫做任务级。前台也叫中断级。时间相关性很强的关键操作(Critical operation)一定是靠中断服务来保证的。因为中断服务提供的信息一直要等到后台程序走到该处理这个信息这一步时才能得到处理，这种系统在处理信息的及时性上，比实际可以做到的要差。这个指标称作任务级响应时间。最坏情况下的任务级响应时间取决于整个循环的执行时间。因为循环的执行时间不是常数，程序经过某一特定部分的准确时间也是不能确定的。进而，如果程序修改了，循环的时序也会受到影响。

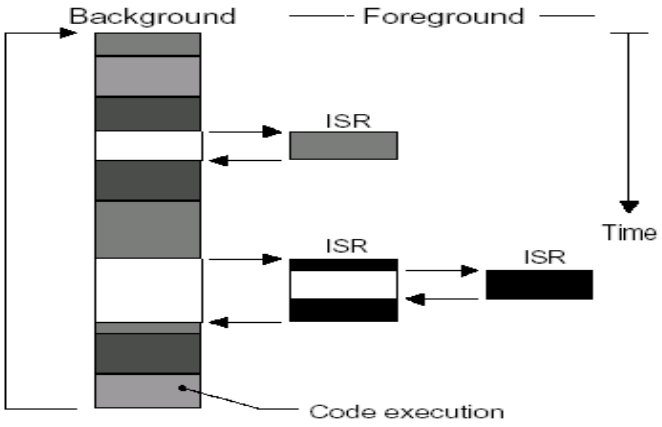


图 2-1 前后台系统

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 代码的临界段

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

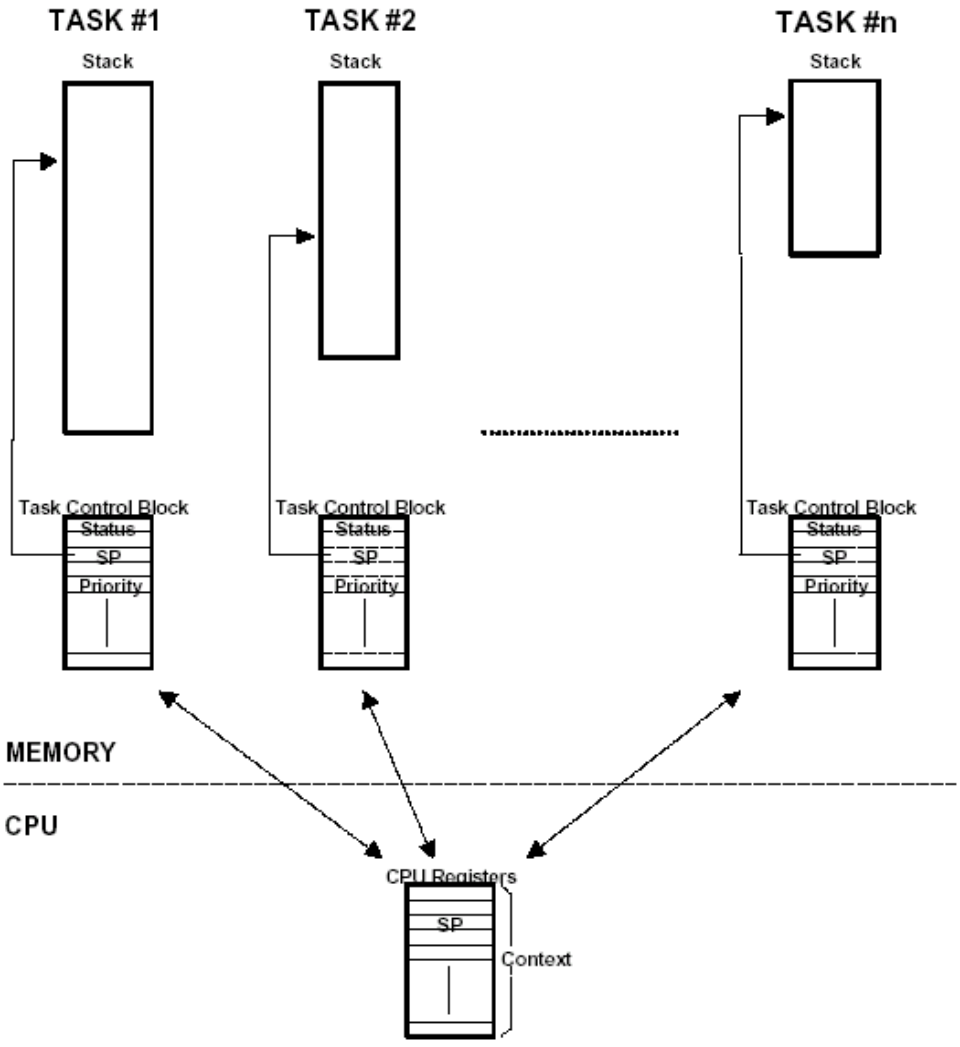


图 2.2 多任务。

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。

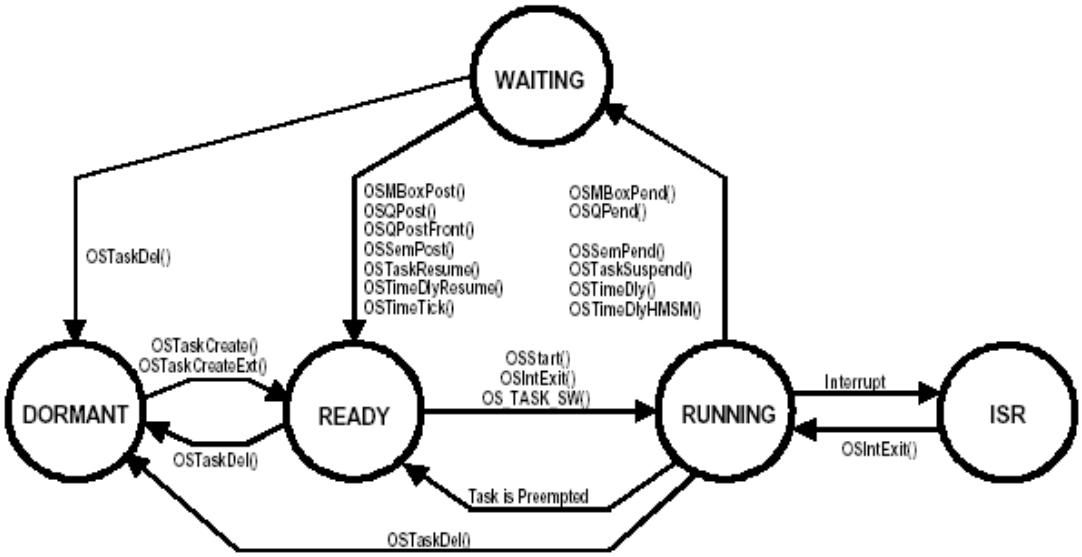


图 2.3 任务的状态

2.6 任务切换*(Context Switch or Task Switch)*

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

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讯。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荷，代码空间增加 ROM 的用量，内核本身的数据结构增加了 RAM 的用量。但更主要的是，每个任务要有自己的栈空间，这一块吃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。通过提供必不可缺少 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（Scheduler）,英文还有一词叫 dispatcher，也是调度的意思。这是内核的主要职责之一，就是要决定该轮到哪个任务运行了。多数实时内核是基于优先级调度法的。每个任务根据其重要程度的不同被赋予一定的优先级。基于优先级的调度法指，CPU 总是让处在就绪态的优先级最高的任务先运行。然而，究竟何时让高优先级任务掌握 CPU 的使用权，有两种不同的情况，这要看用的是什么类型的内核，是不可剥夺型的还是可剥夺型内核。

2.9 不可剥夺型内核 （Non-Preemptive Kernel）

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

不可剥夺型内核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，不可剥夺型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，而不必担心其它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。

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

不可剥夺型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 示意不可剥夺型内核的运行情况，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[F2.4(2)]进入中断服务子程序，中断服务子程序做事件处理[F2.4(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[F2.4(4)]，使 CPU 回到原来被中断的任务，接着执行该任务的代码[F2.4(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[F2.4(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[F2.4(7)]。

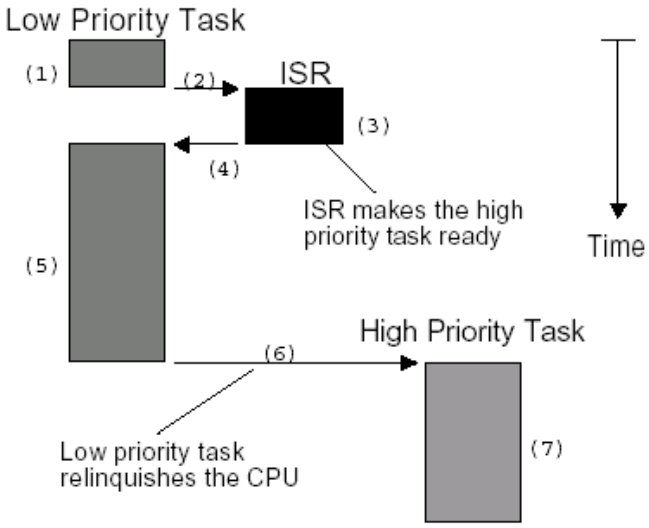


图 2.4 不可剥夺型内核

不可剥夺型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，要等，也许要等很长时间，直到当前运行着的任务释放 CPU。与前后系统一样，不可剥夺型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，不可剥夺型内核允许每个任务运行，直到该任务自愿放弃 CPU 的控制权。中断可以打入运行着的任务。中断服务完成以后将 CPU 控制权还给被中断了的任务。任务级响应时间要大大好于前后系统，但仍是不可知的，商业软件几乎没有不可剥夺型内核。

2.10 可剥夺型内核

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

那个任务开始运行。如图 2.5 所示。

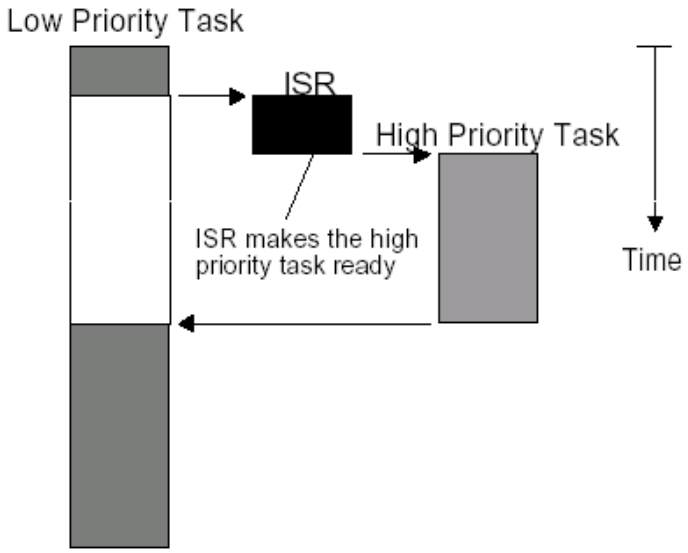


图 2.5 可剥夺型内核

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

使用可剥夺型内核时，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，低优先级的任务 CPU 的使用权被高优先级任务剥夺，不可重入型函数中的数据有可能被破坏。综上所述，可剥夺型内核总是让就绪态的高优先级的任务先运行，中断服务程序可以抢占 CPU，到中断服务完成时，内核让此时优先级最高的任务运行（不一定是那个被中断了的任务）。任务级系统响应时间得到了最优化，且是可知的。μC/OS-II 属于可剥夺型内核。

2.11 可重入性（*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

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

```
    }
    *dest = NUL;
}
```

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

不可重入型函数的例子如程序 2.2 所示。Swap()是一个简单函数，它使函数的两个形式变量的值互换。为便于讨论，假定使用的是可剥夺型内核，中断是开着的，Temp 定义为整数全程变量。

程序清单 2.2　　不可重入型函数

```
int Temp;

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

程序员打算让 Swap() 函数可以为任何任务所调用，如果一个低优先级的任务正在执行 Swap()函数，而此时中断发生了，于是可能发生的事情如图 2.6 所示。[F2.6(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[F2.6(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[F2.6(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[F2.6(4)],释放了 CPU 的使用权，低优先级任务得以继续运行[F2.6(5)]. 注意，此时 Temp 的值仍为 3！在低优先级任务接着运行时，Y 的值被错误地赋为 3，而不是正确值 1。

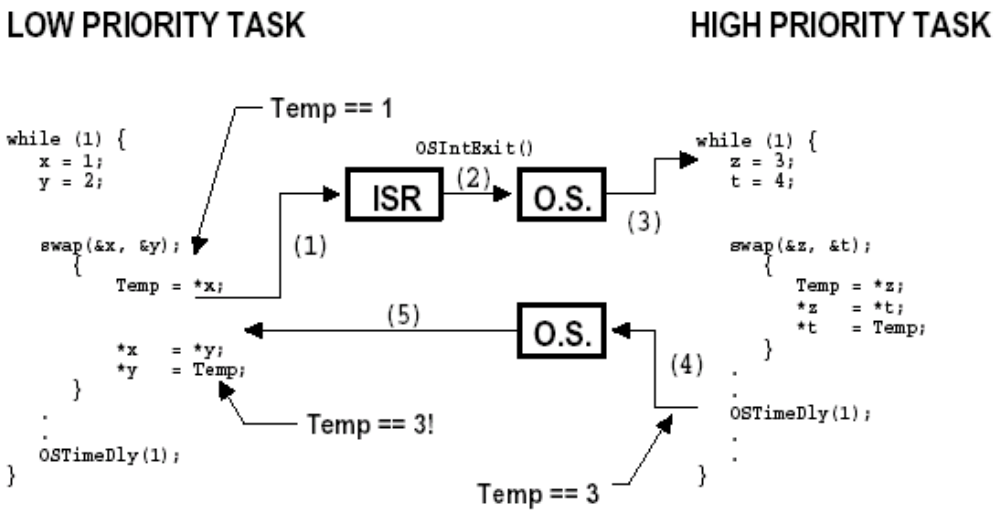


图 2.6 不可重入性函数

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,问题并非那么易解。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。如果在多任务上您还是把新手，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。

2.12 时间片轮番调度法

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。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。任务越重要，赋予的优先级应越高。

2.14 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)]。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 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 27(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，内核做任务切换，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,直等到任务 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。

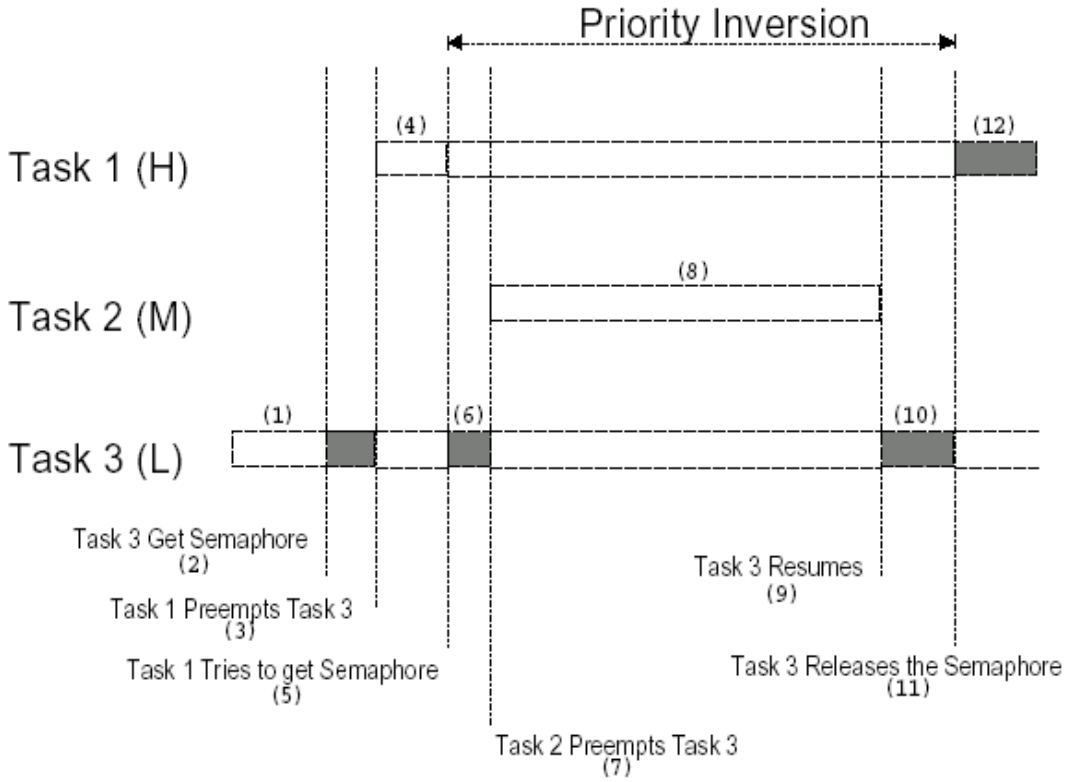


图 2.7 优先级反转问题

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 2.7(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。

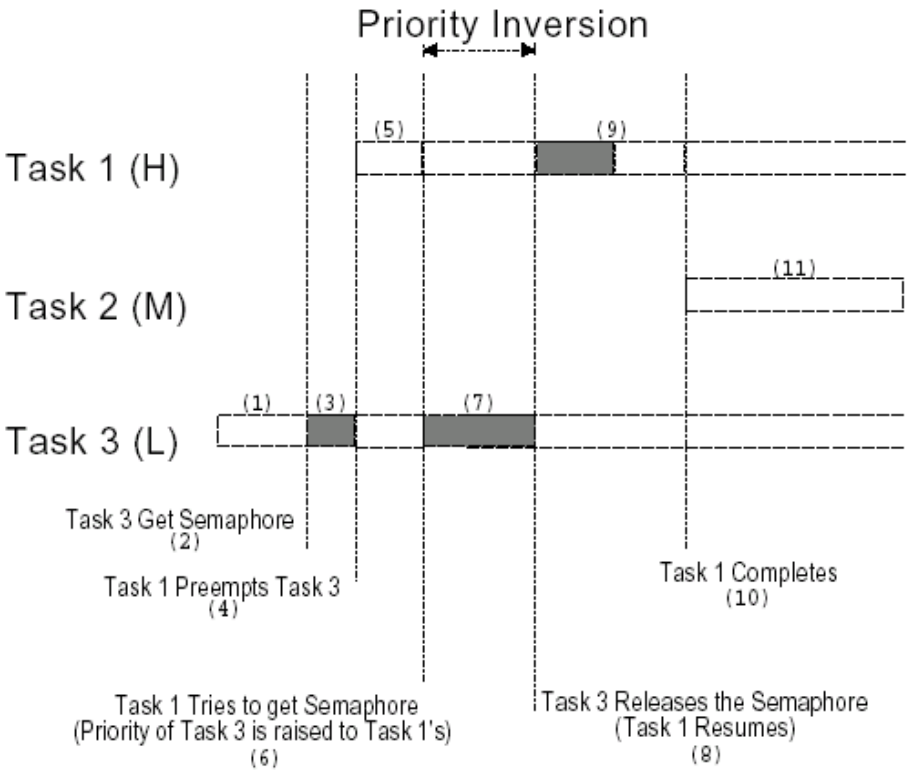


Figure 2-8, Kernel that supports priority inheritance.

图 2.8

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。

图 2.9 基于任务执行频繁度的优先级分配法

任务执行频繁度（Hz）
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，要使用可剥夺型调度法。

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

[2.1] $$\sum_i \frac{Ei}{Ti} \le n(2^{1/n} - 1)$$

这里 E_i 是任务 i 最长执行时间，T_i 是任务 i 的执行周期。换句话说，E_i/T_i 是任务 i 所需的 CPU 时间。表 2.1 给出 $n(2^{1/n} - 1)$ 的值，n 是系统中的任务数。对于无穷多个任务，极限值是 或 $\ell n(2)$ 0.693。这就意味着，基于 RMS，要任务都满足硬实时条件，所有有时间条件要求的任务 i 总的 CPU 利用时间应小于 70%！请注意，这是指有时间条件要求的任务，系统中当然还可以有对时间没有什么要求的任务，使得 CPU 的利用率达到 100%。使 CPU 利用率达到 100%并不好，因为那样的话程序就没有了修改的余地，也没法增加新功能了。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.

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

2.18 互斥条件

实现任务间通讯最简便到办法是使用共享数据结构。特别是当所有到任务都在一个单一

地址空间下，能使用全程变量、指针、缓冲区、链表、循环缓冲区等，使用共享数据结构通讯就更为容易。虽然共享数据区法简化了任务间的信息交换，但是必须保证每个任务在处理共享数据时的排它性，以避免竞争和数据的破坏。与共享资源打交道时，使之满足互斥条件最一般的方法有：

- 关中断
- 使用测试并置位指令
- 禁止做任务切换
- 利用信号量

2.18.1 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 2.3 所示：

程序清单 2.3　　关中断和开中断

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

μC/OS-Ⅱ在处理内部变量和数据结构时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 8.03.02 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICALL()],这两个宏调用的使用法见程序 2.4

程序清单 2.4 利用μC/OS_Ⅱ 宏调用关中断和开中断

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

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

如果不使用实时内核，当两个任务共享一个资源时，一定要约定好，先测试某一全程变量，如果该变量是 0，允许该任务与共享资源打交道。为防止另一任务也要使用该资源，前者只要简单地将全程变量置为 1，这通常称作测试并置位(Test-And-Set)，或称作 TAS。TAS 操作可能是微处理器的单独一条不会被中断的指令，或者是在程序中关中断做 TAS 操作再开中断，如程序清单 2.5 所示。

程序清单 2.5 利用测试并置位处理共享资源

```
Disable interrupts;                                关中断
if ('Access Variable' is 0) {                      如果资源不可用，标志为0
    Set variable to 1;                             置资源不可用，标志为1
    Reenable interrupts;                           重开中断
    Access the resource;                           处理该资源
    Disable interrupts;                            关中断
    Set the 'Access Variable' back to 0;           清资源不可使用，标志为0
    Reenable interrupts;                           重新开中断
} else {                                           否则
    Reenable interrupts;                           开中断
    /* You don't have access to the resource, try back later; */
    /* 资源不可使用，以后再试; */
}
```

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 3.06 给任务切换上锁和开锁)。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。

程序清单2.6　用给任务切换上锁，然后开锁的方法实现数据共享.

```
void Function (void)
{
    OSSchedLock();
    .
    .    /* You can access shared data in here (interrupts are recognized)
*/
    .    /*在这里处理共享数据(中断是开着的)*/
    OSSchedUnlock();
}
```

2.18.4 信号量(Semaphores)

信号量是 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

(译者注：信号与信号量在英文中都叫做 Semaphore，并不加以区分，而说它有两种类型，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))。

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)。

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，或者是
- 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)

有的内核有选择项，允许用户在信号量初始化时选定上述两种方法中的一种。但μC/OS-Ⅱ只支持优先级法。如果进入就绪态的任务比当前运行的任务优先级高(假设，是当前任务释放的信号量激活了比自己优先级高的任务)。则内核做任务切换(假设，使用的是可剥夺型内核)，高优先级的任务开始运行。当前任务被挂起。直到又变成就绪态中优先级最高任务。

程序清单 2.7 示意在μC/OS-Ⅱ中如何用信号量处理共享数据。要与同一共享数据打交道的任务调用等待信号量函数 OSSemPend()。处理完共享数据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

```
OS_EVENT *SharedDataSem;
void Function (void)
{
    INT8U err;
    OSSemPend(SharedDataSem, 0, &err);
    .
    .    /* You can access shared data in here (interrupts are recognized)
*/
    .    /*共享数据的处理在此进行，(中断是开着的)*/
    OSSemPost(SharedDataSem);
}
```

当诸任务共享输入输出设备时，信号量特别有用。可以想象，如果允许两个任务同时给

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。

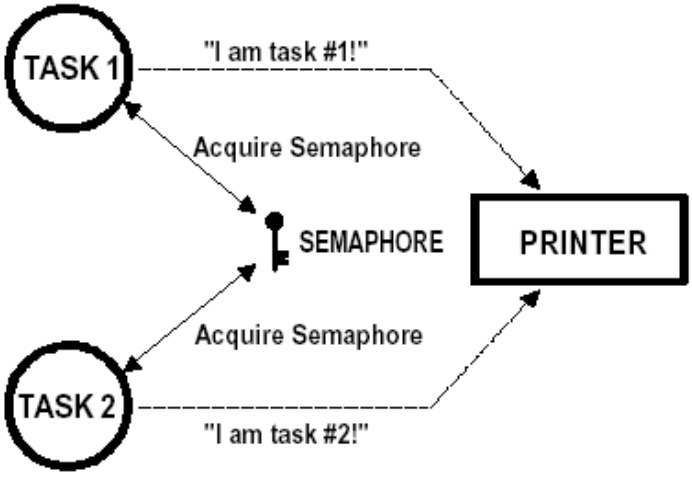


图 2.10 用获取信号量来得到打印机使用权

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。然而有些情况下，最好把信号量藏起来，各个任务在同某一资源打交道时，并不知道实际上是在申请得到一个信号量。例如，多任务共享一个 RS-232C 外设接口，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。

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。

```
INT8U CommSendCmd(char *cmd, char *response, INT16U timeout)
{
    Acquire port's semaphore;
    Send command to device;
    Wait for response (with timeout);
    if (timed out) {
        Release semaphore;
        return (error code);
```

```
    } else {
        Release semaphore;
        return (no error);
    }
}
```

要向外设发送命令的任务得调用上述函数。设信号量初值为 1，表示允许使用。初始化是在通讯口驱动程序的初始化部分完成的。第一个调用 CommSendCmd()函数的任务申请并得到了信号量，开始向外设发送命令并等待响应。而另一个任务也要送命令，此时外设正“ 忙 ”，则第二个任务被挂起，直到该信号量重新被释放。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。

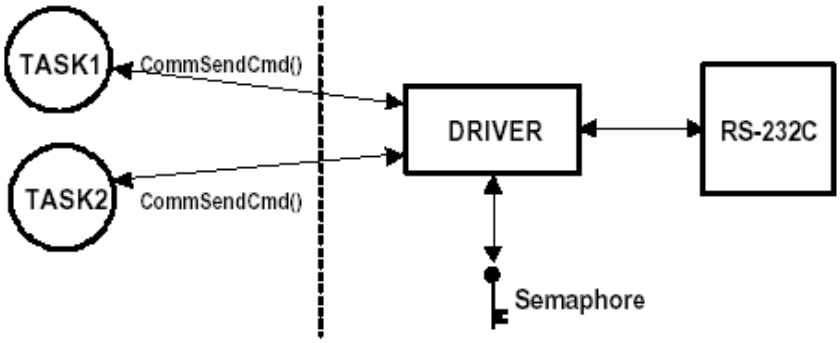


图 2.11 在任务级看不到隐含的信号量

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。

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

   Acquire a semaphore;
   Disable interrupts;
   ptr        = BufFreeList;
```

```
   BufFreeList = ptr->BufNext;
   Enable interrupts;
   return (ptr);
}



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

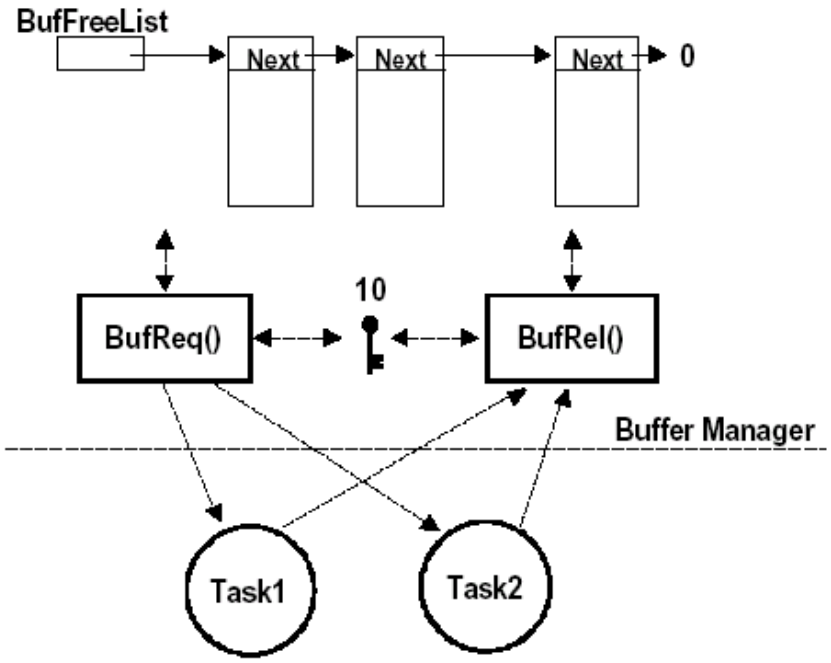


图 2.12 计数式信号量的用法

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中(Linked list)然后再给信号量加 1 或释放该信号量。这一过程隐含在缓冲区管理程序 BufReq()和 BufRel()之中，调用这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 2.18.0.1 关中断和开中断)。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。

2.19 死锁(或抱死) (Deadlock (or Deadly Embrace))

死锁也称作抱死，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 T2，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都：

- 先得到全部需要的资源再做下一步的工作
- 用同样的顺序去申请多个资源
- 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。

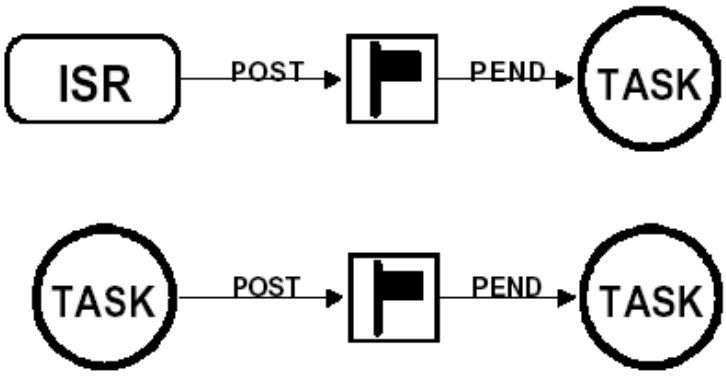


图 2.13 用信号量使任务与中断服务同步

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，或者
- 发信号给最先开始等待事件发生的那个任务

根据不同的应用，发信号以标识事件发生的中断服务或任务也可以是多个。

两个任务可以用两个信号量同步它们的行为。如图 2.14 所示。这叫做双向同步(bilateral rendezvous)。双向同步同单向同步类似，只是两个任务要相互同步。

例如则程序清单 2.10 中，运行到某一处的第一个任务发信号给第二个任务[L22.10(1)]，然后等待信号返回[L2.10(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[2.10(3)]等待返回信号[L2.10(4)]。至此，两个任务实现了互相同步。在任务与中断服务之间不能使用双向同步，因为在中断服务中不可能等一个信号量。

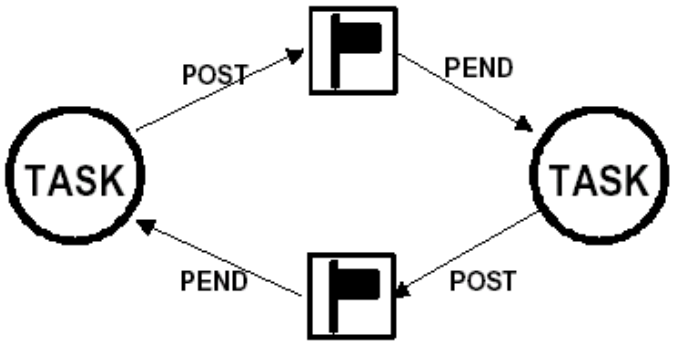


图 2.14 两个任务用信号量同步彼此的行为

程序清单2.10　双向同步

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

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

2.21 事件标志(Event Flags)

当某任务要与多个事件同步时,要使用事件标志。若任务需要与任何事件之一发生同步,可称为独立型同步(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(逻辑与关系)。独立型及关联型同步如图 2.15 所示。

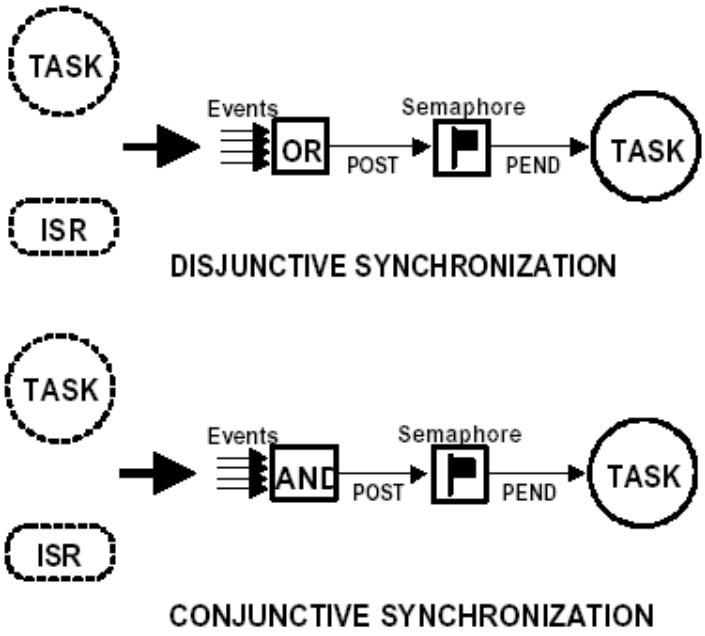


图 2.15 独立型及关联型同步

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。每个事件占一位(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辨定的。也就是在事件位置位时做辨断。

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。μC/OS-II 目前不支持事件标志.

2.22 任务间的通讯(Intertask Communication)

有时很需要任务间的或中断服务与任务间的通讯。这种信息传递称为任务间的通讯。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唯一办法是关中断。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讯，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。

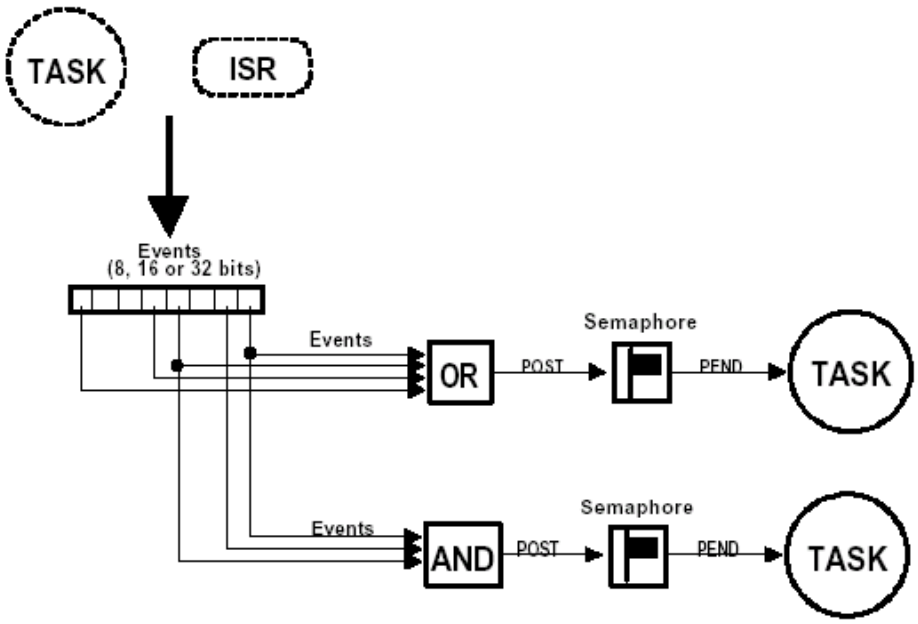


图 2.16 事件标志

2.23 消息邮箱(Message Mail boxes)

通过内核服务可以给任务发送消息。典型的消息邮箱也称作交换消息，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 Ticks)，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它任务占用。

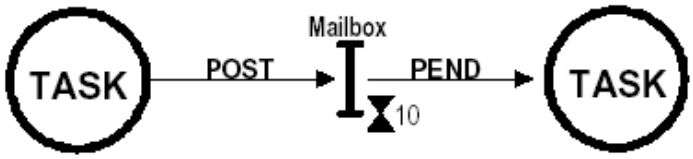


图 2.17 消息邮箱

2.24 消息队列(Message Queue)

消息队列用于给任务发消息。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。图中两个大写的 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。

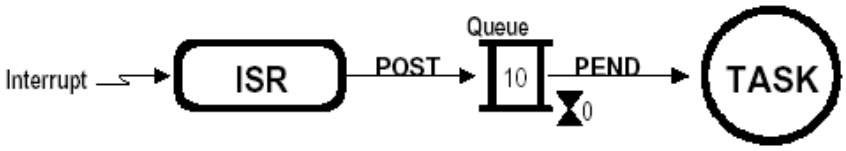


图 2.18 消息队列

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)现场(Context)即部分或全部寄存器的值，跳转到专门的子程序，称为中断服务子程序(ISR)。中断服务子程序做事件处理，处理完成后，程序回到：

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

中断使得 CPU 可以在事件发生时才予以处理，而不必让微处理器连续不断地查询(Polling)是否有事件发生。通过两条特殊指令：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 中断延迟)。关中断时间太长可能会引起中断丢失。微处理器一般允许中断嵌套，也就是说在中断服务期间，微处理器可以识别另一个更重要的中断，并服务于那个更重要的中断，如图 2.19 所示。

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。

[2.2] 中断延迟 = 关中断的最长时间 + 开始执行中断服务子程序的第一条指令的时间

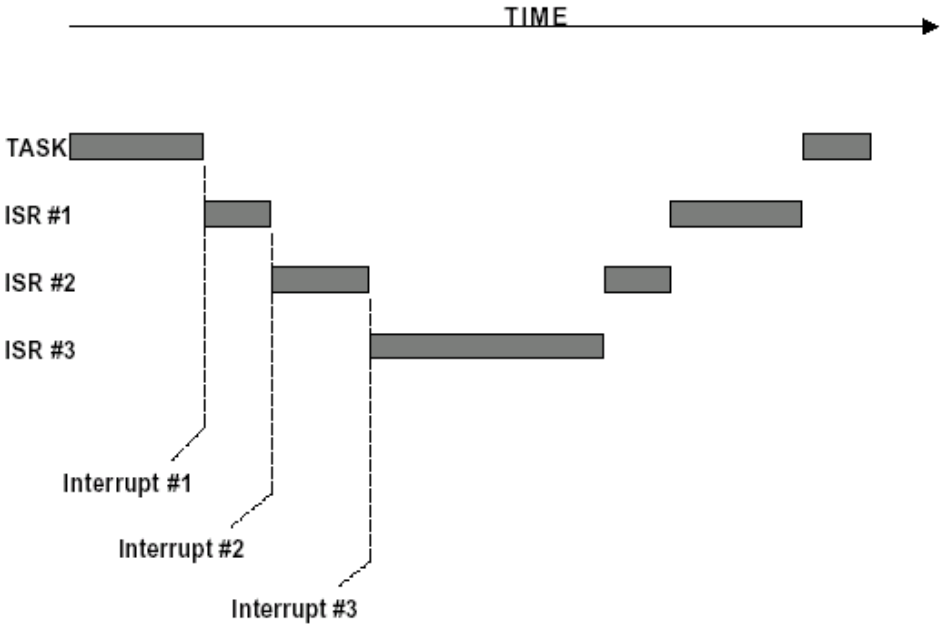


图 2.19 中断嵌套

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[2.3]给出。

[2.3] 中断响应时间 = 中断延迟 + 保存 CPU 内部寄存器的时间

对于不可剥夺型内核，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。不可剥夺型内核的中断响应时间由表达式[2.4]给出。

[2.4] 中断响应时间 = 中断延迟 + 保存 CPU 内部寄存器的时间

对于可剥夺型内核，则要先调用一个特定的函数，该函数通知内核即将进行中断服务，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()，可剥夺型内核的中断响应时间由表达式[2.5]给出：

[2.5] 中断响应 = 中断延迟 + 保存 CPU 内部寄存器的时间 + 内核的进入中断服务

函数的执行时间

中断响应是系统在最坏情况下的响应中断的时间，某系统 100 次中有 99 次在 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(Interrupt Recovery)

中断恢复时间定义为微处理器返回到被中断了的程序代码所需要的时间。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[2.6]式给出。

[2.6] 中断恢复时间 = 恢复 CPU 内部寄存器值的时间 + 执行中断返回指令的时间

和前后台系统一样，不可剥夺型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。

[2.7] 中断恢复时间 = 恢复 CPU 内部寄存器值的时间 + 执行中断返回指令的时间

对于可剥夺型内核，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。对于可剥夺型内核，中断恢复时间由表达式[2.8]给出。

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

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、不可剥夺性内核、可剥夺性内核相应的中断延迟、响应和恢复过程。

注意，对于可剥夺型实时内核，中断返回函数将决定是返回到被中断的任务[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。

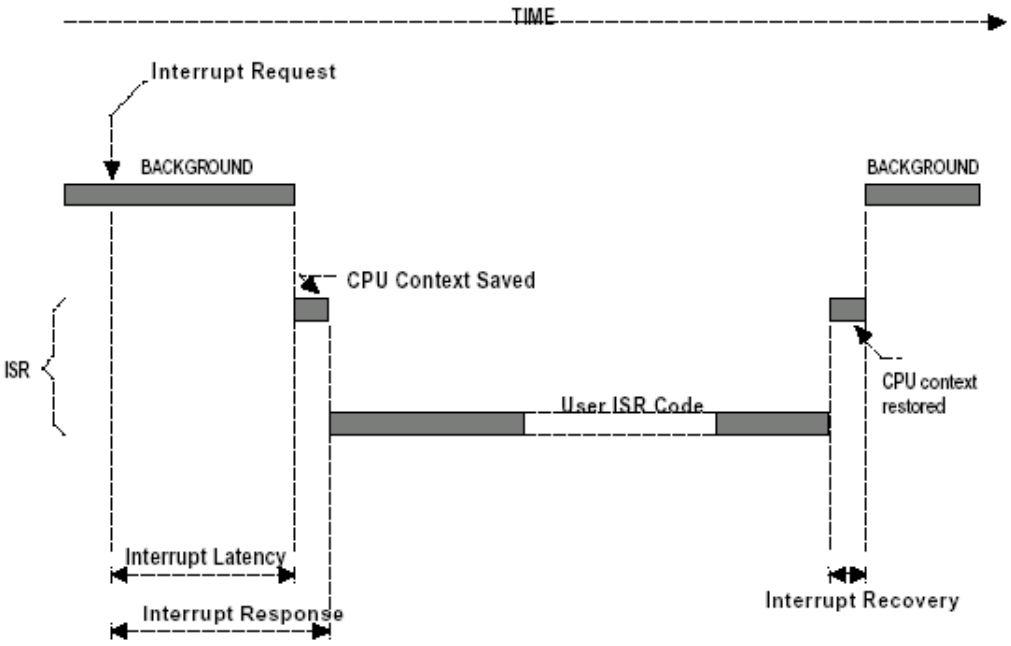


图 2.20 中断延迟、响应和恢复(前后台模式)

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，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。

[2.9] 中断延迟时间 = 指令执行时间中最长的那个时间 + 开始做非屏蔽中断服务的时间

[2.10] 中断响应时间 = 中断延迟时间 + 保存 CPU 寄存器花的时间

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

在一项应用中，我将非屏蔽中断用于可能每 150μS 发生一次的中断。中断处理时间在80至 125μS 之间。所使用的内核的关中断时间是 45μS。可以看出，如果使用可屏蔽中断的话，中断响应会推迟 20μS。

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。

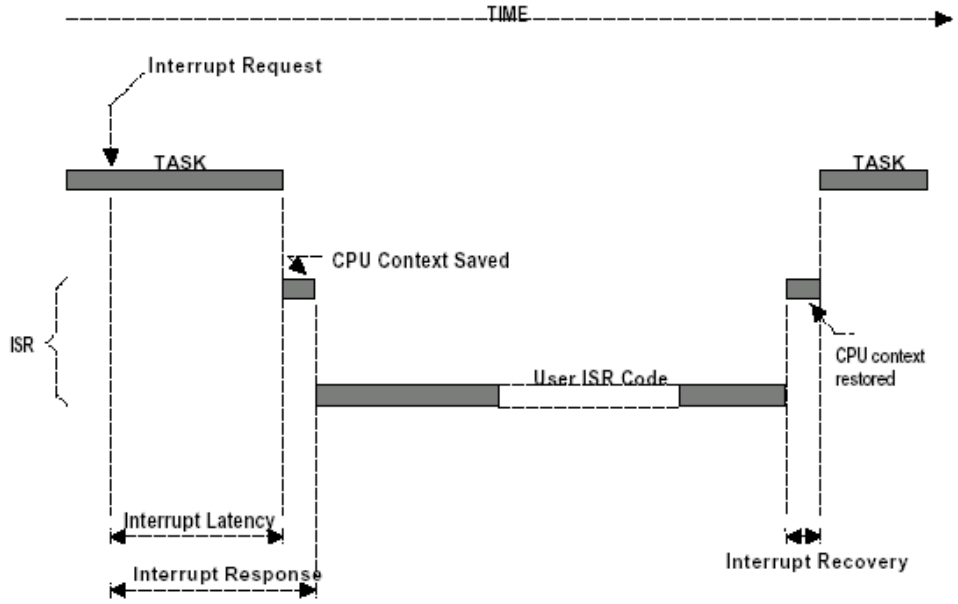


图 2.21 中断延迟、响应和恢复(不可剥夺型内核)

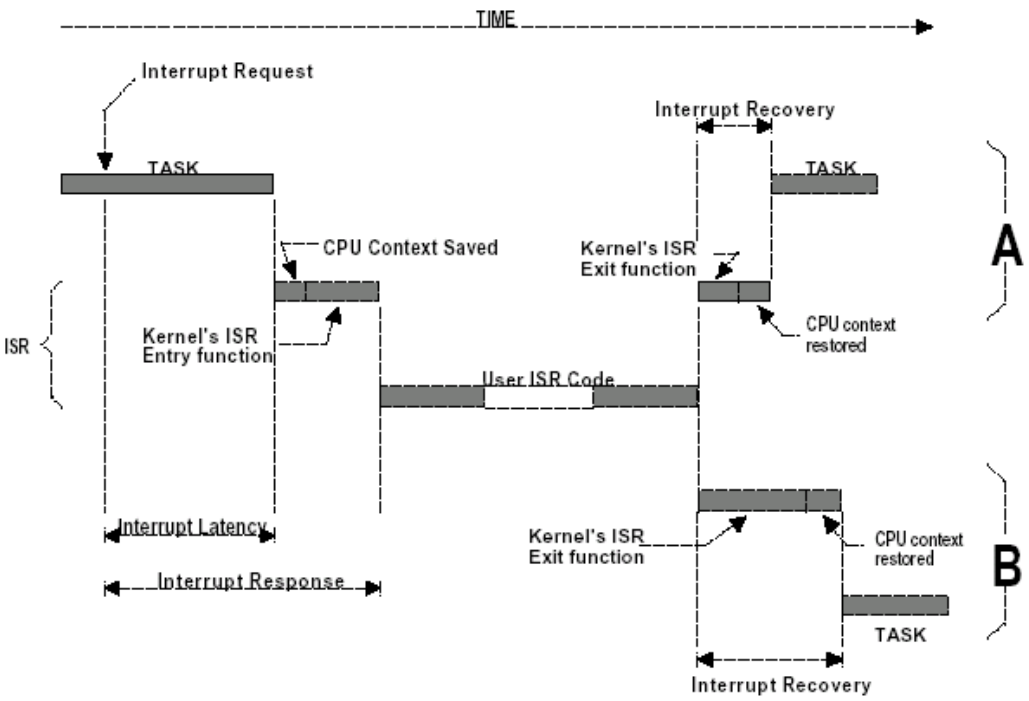


图 2.22 中断延迟、响应和恢复(可剥夺型内核)

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)。

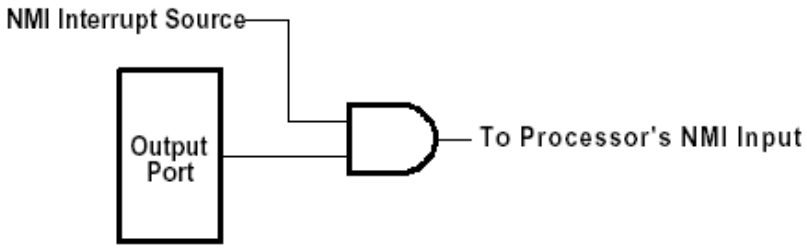


图 2.23 非屏蔽中断的禁止

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

μS 执行一次，则每 6mS(40*150μS)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。

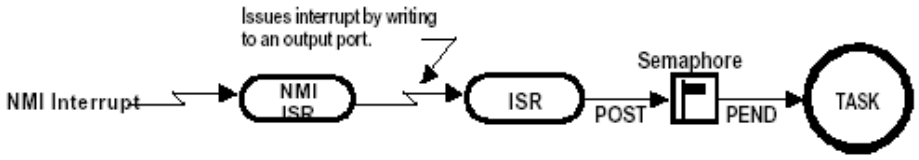


图 2.24 非屏蔽中断产生普通可屏蔽中断

2.32 时钟节拍(Clock Tick)

时钟节拍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。

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。

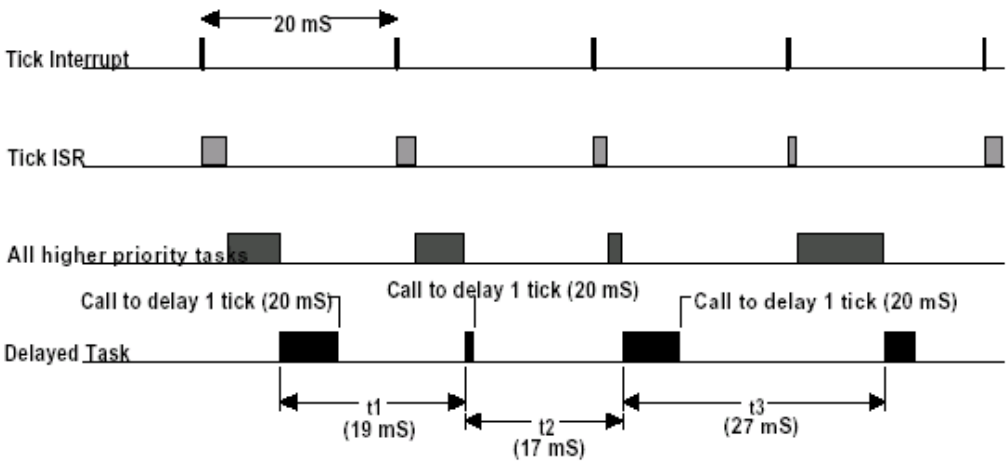


图 2.25 将任务延迟一个时钟节拍(第一种情况)

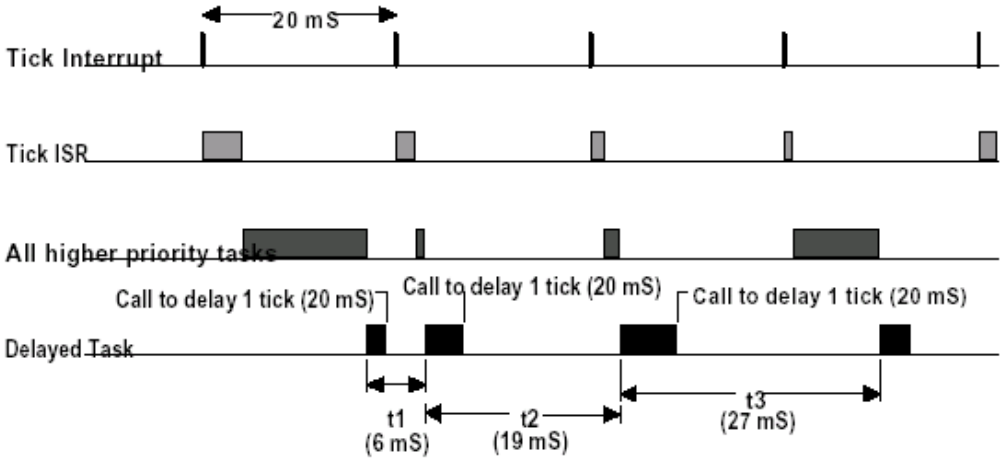


图 2.26 将任务延迟一个时钟节拍(第二种情况)

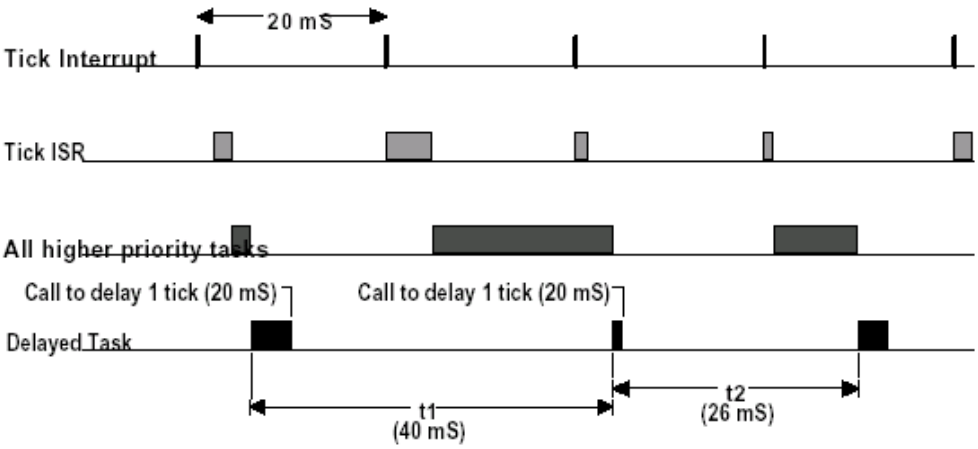


图 2.27 将任务延迟一个时钟节拍(第三种情况)

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。

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。

[2.12]　总代码量 = 应用程序代码 + 内核代码

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。

[2.13] RAM 总需求 = 应用程序的 RAM 需求 + (任务栈需求 + 最大中断嵌套栈需求) * 任务数

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

[2.14]=RAM 总需求 = 应用程序的 RAM 需求 + 内核数据区的 RAM 需求 + 各任务栈需求之总和 + 最多中断嵌套之栈需求

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减化。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 compilers)。RTOS 的价格从 70 美元到 30,000 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。同如今的其它软件包一样，还得考虑软件维护费，这部分开销为每年还得花 100 到 5,000 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，不可剥夺型内核和可剥夺型内核。

. 表 2.2　实时系统小结

	Foreground/ Background	***Non-Preemptive Kernel***	***Preemptive Kernel***
Interrupt latency (Time)	MAX(Longest instruction, User int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR
Interrupt response (Time)	Int. latency + Save CPU's context	Int. latency + Save CPU's context	Interrupt latency + Save CPU's context + Kernel ISR entry function
Interrupt recovery (Time)	Restore background's context + Return from int.	Restore task's context + Return from int.	Find highest priority task + Restore highest priority task's context + Return from interrupt
Task response (Time)	Background	Longest task + Find highest priority task	Find highest priority task + Context switch

		+ Context switch	
ROM size	Application code	Application code + Kernel code	Application code + Kernel code
RAM size	Application code	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))
Services available?	Application code must provide	Yes	Yes

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是怎样处理临界段代码的;
- 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，以及
- 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(), 以及
- OSVersion().

3.0 临界段(Critical Sections)

和其它内核一样，μC/OS-Ⅱ为了处理临界段代码需要关中断，处理完毕后再开中断。这使得μC/OS-Ⅱ能够避免同时有其它任务或中断服务进入临界段代码。关中断的时间是实时内核开发商应提供的最重要的指标之一，因为这个指标影响用户系统对实时事件的响应性。μC/OS-Ⅱ努力使关中断时间降至最短，但就使用μC/OS-Ⅱ而言，关中断的时间很大程度上取决于微处理器的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。这使得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。μC/OS-Ⅱ定义两个宏(macros)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。μC/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。每种微处理器都有自己的 OS_CPU.H 文件。

3.1 任务

一个任务通常是一个无限的循环[L3.1(2)]，如程序清单 3.1 所示。一个任务看起来像其它 C 的函数一样，有函数返回类型，有形式参数变量，但是任务是绝不会返回的。故返回参数必须定义成 void[L3.1(1)]。

程序清单 L3.1 任务是一个无限循环

```
void YourTask (void *pdata)                                    (1)
{
  for (;;) {                                                   (2)
    /* 用户代码 */
    调用uC/OS-II的某种系统服务:
    OSMboxPend();
    OSQPend();
    OSSemPend();
    OSTaskDel(OS_PRIO_SELF);
    OSTaskSuspend(OS_PRIO_SELF);
    OSTimeDly();
    OSTimeDlyHMSM();
    /* 用户代码 */
  }
}
```

不同的是，当任务完成以后，任务可以自我删除，如清单 L3.2 所示。注意任务代码并非真的删除了，μC/OS-Ⅱ只是简单地不再理会这个任务了，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。

程序清单 L 3.2 ． 任务完成后自我删除

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

形式参数变量[L3.1(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的任务，所有任务都使用同一个函数（或者说是同一个任务代码程序），见第一章的例 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）。

μC/OS-Ⅱ可以管理多达 64 个任务，但目前版本的μC/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 是作为定义的常数在 OS_CFG.H 文件中用定义常数语句#define constant 定义的。因此用户可以有多达 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。μC/OS-Ⅱ总是运行进入就绪态的

优先级最高的任务。目前版本的μC/OS-Ⅱ中，任务的优先级号就是任务编号（ID）。优先级号（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μC/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它参数一起传给下面两个函数中的一个：OSTastCreat 或 OSTaskCreatExt()。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四章，任务管理。

3.2 任务状态

图 3.1 是μC/OS-Ⅱ控制下的任务状态转换图。在任一给定的时刻，任务的状态一定是在这五种状态之一。

睡眠态（DORMANT）指任务驻留在程序空间之中，还没有交给μC/OS-Ⅱ管理，（见程序清单 L3.1 或 L3.2）。把任务交给μC/OS-Ⅱ是通过调用下述两个函数之一：OSTaskCreate()或 OSTaskCreateExt()。当任务一旦建立，这个任务就进入就绪态准备运行。任务的建立可以是在多任务运行开始之前，也可以是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()返回到睡眠态，或通过调用该函数让另一个任务进入睡眠态。

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。就绪的任务只有当所有优先级高于这个任务的任务转为等待状态，或者是被删除了，才能进入运行态。

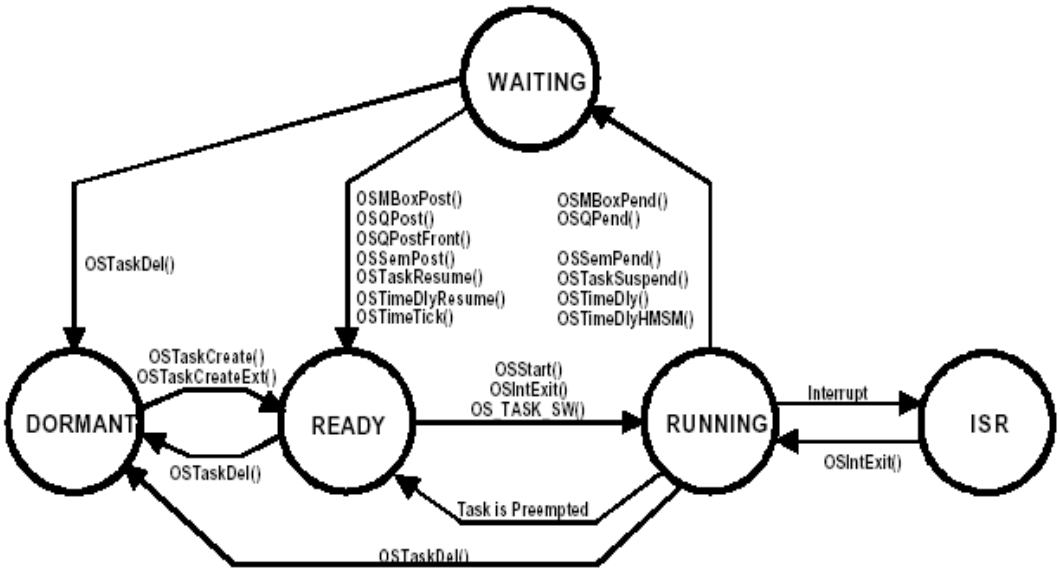


Figure 3-1, Task States

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之一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，时钟节拍）。

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，除非该任务将中断关了，或者μC/OS-Ⅱ将中断关了。被中断了的任务就进入了中断服务态（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才能继续运行。

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

3.3 任务控制块（Task Control Blocks，OS _TCBs）

一旦任务建立了，任务控制块 OS _TCBs 将被赋值（程序清单 3.3）。任务控制块是一个数据结构，当任务的 CPU 使用权被剥夺时，μC/OS-Ⅱ用它来保存该任务的状态。当任务重新得到 CPU 使用权时，任务控制块能确保任务从当时被中断的那一点丝毫不差地继续执行。OS _TCBs 全部驻留在 RAM 中。读者将会注意到笔者在组织这个数据结构时，考虑到了各成员的逻辑分组。任务建立的时候，OS _TCBs 就被初始化了（见第四章 任务管理）。

程序清单 L 3.3 μC/OS-II任务控制块

```
typedef struct os_tcb {
    OS_STK       *OSTCBStkPtr;


#if OS_TASK_CREATE_EXT_EN
    void         *OSTCBExtPtr;
    OS_STK       *OSTCBStkBottom;
    INT32U        OSTCBStkSize;
    INT16U        OSTCBOpt;
    INT16U        OSTCBId;
#endif


    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;


#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT     *OSTCBEventPtr;
#endif


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

```
    INT16U        OSTCBDly;
    INT8U         OSTCBStat;
    INT8U         OSTCBPrio;

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

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

.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之。这种限制浪费了 RAM，当各任务需要的栈空间不同时，也得按任务中预期栈容量需求最多的来分配栈空间。OSTCBStkPtr 是 OS_TCB 数据结构中唯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_CREAT_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，从数据结构的头上算偏移量比较方便。

.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 1,000 个入口地址，每个地址宽度是 32 位的，则实际栈容量是 4,000 字节。同样是 1,000 个入口地址，如果每个地址宽度是 16 位的，则总栈容量只有 2,000 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_CREAT_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_OTP_STK_CHK，OS_TASK_OPT_STK_CLR 和 OS_TASK_OPT_SAVE_FP 。OS_TASK_OTP_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。

.OSTCBEventPtr 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 任务间通讯与同步）。

.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 章任务间通讯与同步）。

.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允许等待事件发生的最多时钟节拍数。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 赋其它的值，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（避免在运行中去计算这些值）。这些值是在任务建立时算好的，或者是在改变任务优先级时算出的。这些值的算法见程序清单 L3.4。

程序清单 L 3.4 任务控制块OS_TCB中几个成员的算法

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

.OSTCBDelReq 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 任务管理）

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。这个最多任务数也是 μC/OS-Ⅱ分配给用户程序的最多任务控制块 OS_TCBs 的数目。将 OS_MAX_TASKS 的数目设置为用户应用程序实际需要的任务数可以减小 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 被链接成单向空任务链表。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 的值调整为指向下链表中下一个空的任务控制块。一旦任务被删除，任务控制块就还给空任务链表。

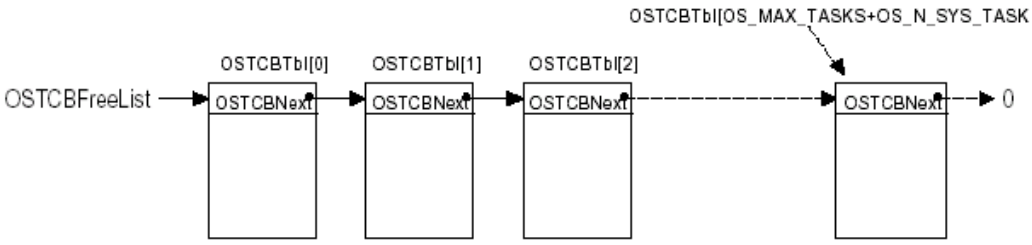


Figure 3-2, List of free OS_TCBs

图 3.2 空任务列表

3.4 就绪表（Ready List）

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中的，就绪表中有两个变量 OSRedyGrp 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 在就绪表中相应字节的相应位置 1。OSRdyGrp 和 OSRdyTbl[]之间的关系见图 3.3，是按以下规则给出的：

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

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 L3.5 使任务进入就绪态

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

表 T3.1 OSMapTbl[]的值

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

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）屏蔽字，用于限制 OSRdyTbl[]数组的元素下标在 0 到 7 之间，见表 3.1

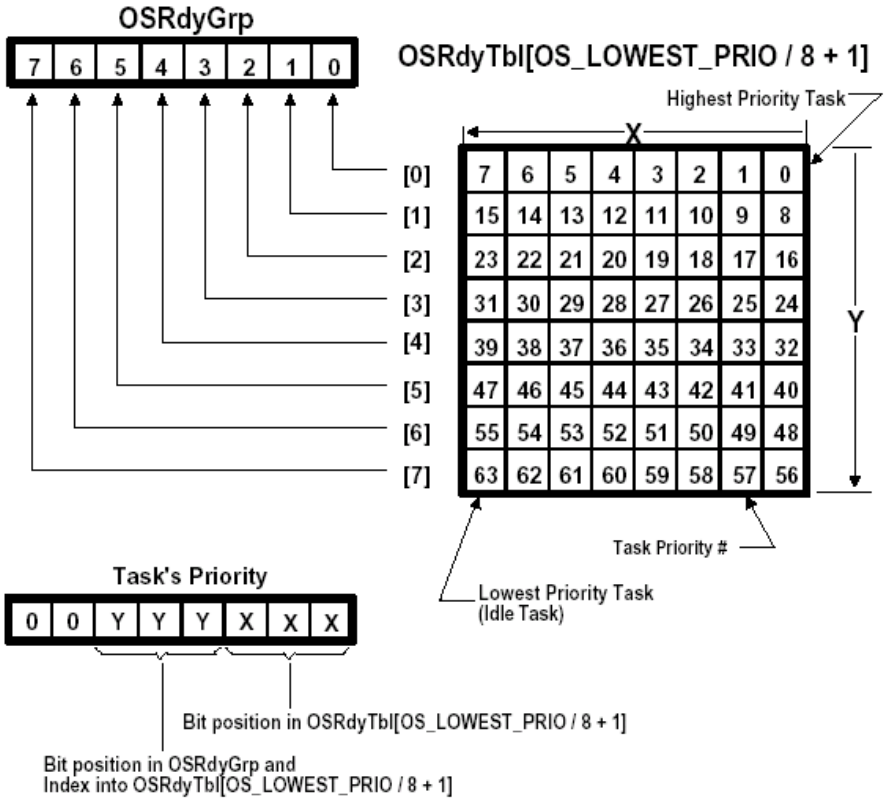


Figure 3-3, μC/OS-II's Ready List

图 3.3 μC/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 L3.6 从就绪表中删除一个任务

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

以上代码将就绪任务表数组 OSRdyTbl[]中相应元素的相应位清零，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 L3.7 所示。

程序清单 L3.7 找出进入就绪态的优先级最高的任务

```
y    = OSUnMapTbl[OSRdyGrp];
x    = OSUnMapTbl[OSRdyTbl[y]];
```

```
prio = (y << 3) + x;
```

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，如果 OSRdyTbl[3]的值是二进制 11100100,则 OSUnMapTbl[OSRdyTbc[3]]的值是 2，即第 2 位。于是任务的优先级 Prio 就等于 26（3*8+2）。利用这个优先级的值。查任务控制块优先级表 OSTCBPrioTbl[]，得到指向相应任务的任务控制块 OS_TCB 的工作就完成了。

3.5 任务调度（Task Scheduling）

μC/OS-Ⅱ总是运行进入就绪态任务中优先级最高的那一个。确定哪个任务优先级最高，下面该哪个任务运行了的工作是由调度器（Scheduler）完成的。任务级的调度是由函数 OSSched()完成的。中断级的调度是由另一个函数 OSIntExt()完成的，这个函数将在以后描述。OSSched()的代码如程序清单 L3.8 所示。

程序清单 L3.8 任务调度器（the Task Scheduler）

```
void OSSched (void)
{
    INT8U y;



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

μC/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[L3.8(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[L3.8(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[L3.8(3)]。注意，在μC/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μC/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μC/OS-Ⅱ比μC/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[L3.8(4)]。接着，统计计数器 OSCtxSwCtr 加 1，以跟踪任务切换次数[L3.8(5)]。最后宏调用 OS_TASK_SW()来完成实际上的任务切换[L3.8(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μC/OS-Ⅱ中，就绪任务的栈结构总是看起来跟刚刚发生过中断一样，所有微处理器的寄存器都保存在栈中。换句话说，μC/OS-Ⅱ运行就绪态的任务所要做的一切，只是恢复所有的 CPU 寄存器并运行中断返回指令。为了做任务切换，运行 OS_TASK_SW(),人为模仿了一次中断。多数微处理器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 hardler），也称作事故处理（exception handler），必须提供中断向量给汇编语言函数 OSCtxSw()。OSCtxSw()除了需要 OS_TCBHighRdy 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，移植μC/OS-Ⅱ，有关于 OSCtxSw()的更详尽的解释。

OSSched()的所有代码都属临界段代码。在寻找进入就绪态的优先级最高的任务过程中，为防止中断服务子程序把一个或几个任务的就绪位置位，中断是被关掉的。为缩短切换时间，OSSched()全部代码都可以用汇编语言写。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(Locking and UnLocking the Scheduler)

给调度器上锁函数 OSSchedlock()（程序清单 L3.9）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 L3.10）。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。然而，此时中断是可以被识别的，中断服务也能得到（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段代码，这段代码其它任务不得干预。μC/OS-Ⅱ允许嵌套深度达 255 层。当 OSLockNesting 等于零时，调度重新得到允许。函数 OSSchedLock()和 OSSchedUnlock()的使用要非常谨慎，因为它们影响μC/OS-Ⅱ对任务的正常管理。

当 OSLockNesting 减到零的时候，OSSchedUnlock()调用 OSSched[L3.10(2)]。OSSchedUnlock()是被某任务调用的，在调度器上锁的期间，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。

调用 OSSchedLock()以后，用户的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 回零为止。因为调度器上了锁，用户就锁住了系统，任何其它任务都不能运行。

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 任务间通讯和同步)，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，用户可以使用禁止调度器函数。

程序清单 L3.9 给调度器上锁

```
void OSSchedLock (void)
{
   if (OSRunning == TRUE) {
      OS_ENTER_CRITICAL();
      OSLockNesting++;
      OS_EXIT_CRITICAL();
   }
}
```

程序清单 L3.10 给调度器开锁.

```
void OSSchedUnlock (void)
{
   if (OSRunning == TRUE) {
      OS_ENTER_CRITICAL();
      if (OSLockNesting > 0) {
         OSLockNesting--;
         if ((OSLockNesting | OSIntNesting) == 0) {                (1)
            OS_EXIT_CRITICAL();
            OSSched();                                             (2)
         } else {
            OS_EXIT_CRITICAL();
         }
      } else {
         OS_EXIT_CRITICAL();
      }
   }
}
```

3.7 空闲任务(Idle Task)

μC/OS-Ⅱ总是建立一个空闲任务，这个任务在没有其它任务进入就绪态时投入运行。这个空闲任务[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 3.08 节，统计任务）使用这个计数器以确定现行应用软件实际消耗的 CPU 时间。程序清单 L3.11 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。空闲任务不可能被应用软件删除。

程序清单 L3.11 μC/OS-Ⅱ的空闲任务.

```
void OSTaskIdle (void *pdata)
{
   pdata = pdata;
   for (;;) {
```

```
        OS_ENTER_CRITICAL();
        OSIdleCtr++;
        OS_EXIT_CRITICAL();
    }
}
```

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(),如果用户将系统定义常数 OS_TASK_STAT_EN（见文件 OS_CFG.H）设为 1，这个任务就会建立。一旦得到了允许，OSTaskStat()每秒钟运行一次（见文件 OS_CORE.C），计算当前的 CPU 利用率。换句话说，OSTaskStat()告诉用户应用程序使用了多少 CPU 时间，用百分比表示，这个值放在一个有符号 8 位整数 OSCPUsage 中，精读度是 1 个百分点。

如果用户应用程序打算使用统计任务，用户必须在初始化时建立一个唯一的任务，在这个任务中调用 OSStatInit()（见文件 OS_CORE.C）。换句话说，在调用系统启动函数 OSStart()之前，用户初始代码必须先建立一个任务，在这个任务中调用系统统计初始化函数 OSStatInit()，然后再建立应用程序中的其它任务。程序清单 L3.12 是统计任务的示意性代码。

程序清单 L3.12 初始化统计任务.

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




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

因为用户的应用程序必须先建立一个起始任务[TaskStart（）]，当主程序 main()调用系统启动函数 OSStcnt()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()的名称是无所谓的，叫什么名

字都可以。因为μC/OS-II已经将空闲任务的优先级设为最低，即OS_LOWEST_PRIO，统计任务的优先级设为次低，OS_LOWEST_PRIO-1。启动任务TaskStart()总是优先级最高的任务。

图F3.4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F3.4(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F3.4(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F3.4(4)]]。

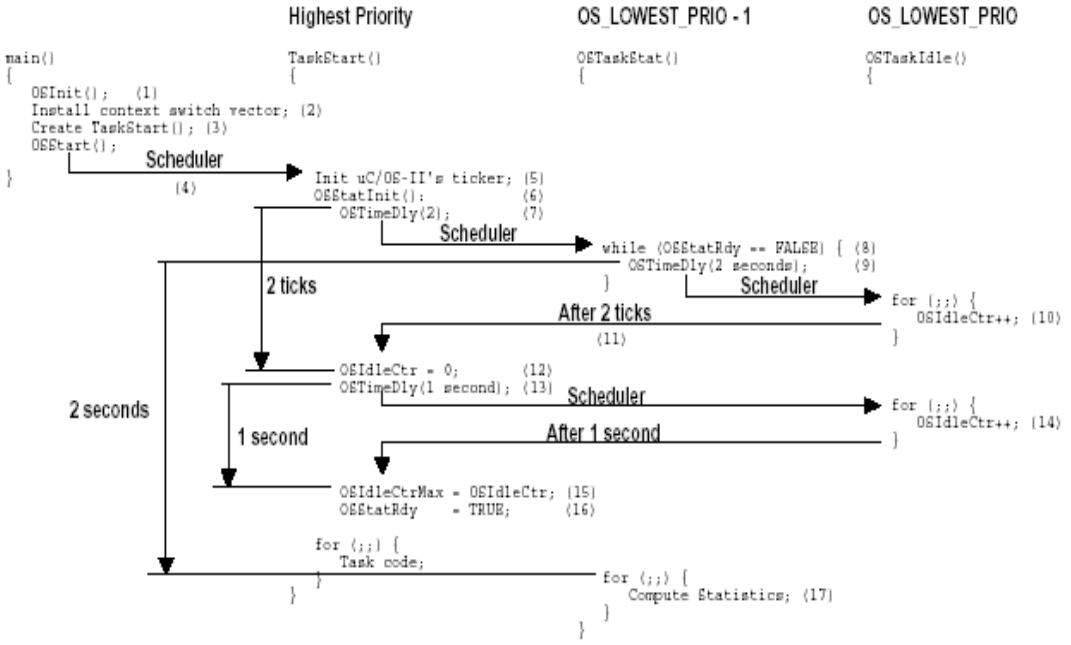


Figure 3-4, Statistic task initialization

图F3.4统计任务的初始化

TaskStart()负责初始化和启动时钟节拍[图F3.4(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F3.4（6）]。统计初始化函数OSStatInit()决定在没有其它应用任务运行时，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。OSIdleCtr的值离32位计数器的溢出极限值4,294,967,296还差得远。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()将自身延时2个时钟节拍以停止自身的运行[图F3.4(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

F3.4(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 DSTaskStat 也将自己推入休眠态（Sleep）两个时钟节拍[图 F3.4(9)]。于是任务切换到空闲任务，OSTaskIdle()开始运行，这是唯一一个就绪态任务了。CPU 处在空闲任务 OSTaskIdle 中，直到 TaskStart()的延迟两个时钟节拍完成[图 3.4(10)]。两个时钟节拍之后，TaskStart()恢复运行[图 F3.4(11)]。　　在执行 OSStartInit()时，空闲计数器 OSIdleCtr 被清零[图 F3.4(12)]。然后，OSStatInit()将自身延时整整一秒[图 F3.4(13)]。因为没有其它进入就绪态的任务，OSTaskIdle()又获得了 CPU 的控制权[图 F3.4(14)]。一秒钟以后，TaskStart()继续运行，还是在 OSStatInit()中，空闲计数器将 1 秒钟内计数的值存入空闲计数器最大值 OSIdleCtrMax 中[图 F3.4(15)]。
OSStarInit()将统计任务就绪标志 OSStatRdy 设为“真”[图 F3.4（16）]，以此来允许两个时钟节拍以后 OSTaskStat()开始计算 CPU 的利用率。
统计任务的初始化函数 OSStatInit()的代码如程序清单L3.13 所示。

程序清单 L3.13 统计任务的初始化

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

统计任务 OSStat()的代码程序清单 L3.14 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[L3.14(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。要知道，OSIdleCtr 的最大计数值是 OSStatInit()在初始化时保存在计数器最大值 OSIdleCtrMax 中的。CPU 利用率（表达式[3.1]）是保存在变量 OSCPUsage[L3.14(2)]中的：

[3.1]表达式　Need to typeset the equation.

一旦上述计算完成，OSTaskStat()调用任务统计外界接入函数 OSTaskStatHook()[L3.14(3)]，这是一个用户可定义的函数，这个函数能使统计任务得到扩展。这样，用户可以计算并显示所有任务总的执行时间，每个任务执行时间的百分比以及其它信息(参见 1.09 节例 3）。

程序清单 L3.14 统计任务

```
void OSTaskStat (void *pdata)
{
```

```
    INT32U run;
    INT8S usage;



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

3.9 μC/OS 中的中断处理

μC/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码如程序清单 L3.15 所示。

程序清单 L3.15 μC/OS-II中的中断服务子程序.

```
用户中断服务子程序:
    保存全部CPU寄存器;                                (1)
    调用OSIntEnter或OSIntNesting直接加1；             (2)
    执行用户代码做中断服务;                           (3)
    调用OSIntExit()；                                 (4)
    恢复所有CPU寄存器；                               (5)
```

执行中断返回指令；　　(6)

用户代码应该将全部 CPU 寄存器推入当前任务栈[L3.15(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。μC/OS-Ⅱ可以用在这类微处理器中，当任务切换时，寄存器是保存在被中断了的那个任务的栈中的。

μC/OS-Ⅱ需要知道用户在做中断服务，故用户应该调用 OSIntEnter()，或者将全程变量OSIntNesting[L3.15(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，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 OSIatNesting 中去，就不如调用 OSIatEnter()。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，中断将连续反复打入，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叫中断的设备了[L3.15(3)]。这一段完全取决于应用。μC/OS-Ⅱ允许中断嵌套，因为μC/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()[L3.15(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μC/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，μC/OS-Ⅱ就返回到那个高优先级的任务，OSIntExit()返回到调用点[L3.15(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[L3.16(6)]。注意，如果调度被禁止了（OSIntNesting>0），μC/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F3.5所示。中断来到了[F3.5(1)]但还不能被被CPU识别，也许是因为中断被μC/OS-Ⅱ或用户应用程序关了，或者是因为 CPU 还没执行完当前指令。一旦 CPU 响应了这个中断[F3.5(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[F3.5(3)]。如上所述，中断服务子程序保存 CPU 寄存器（也叫做 CPU context）[F3.5(4)]，一旦做完，用户中断服务子程序通知μC/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[F3.5(5)]。然后用户中断服务代码开始执行[F3.5(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()[F3.5(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[F3.5(8)]并执行中断返回指令[F3.5(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[F3.5(10)]。新任务的寄存器内容要恢复并执行中断返回指令[F3.5(12)]。

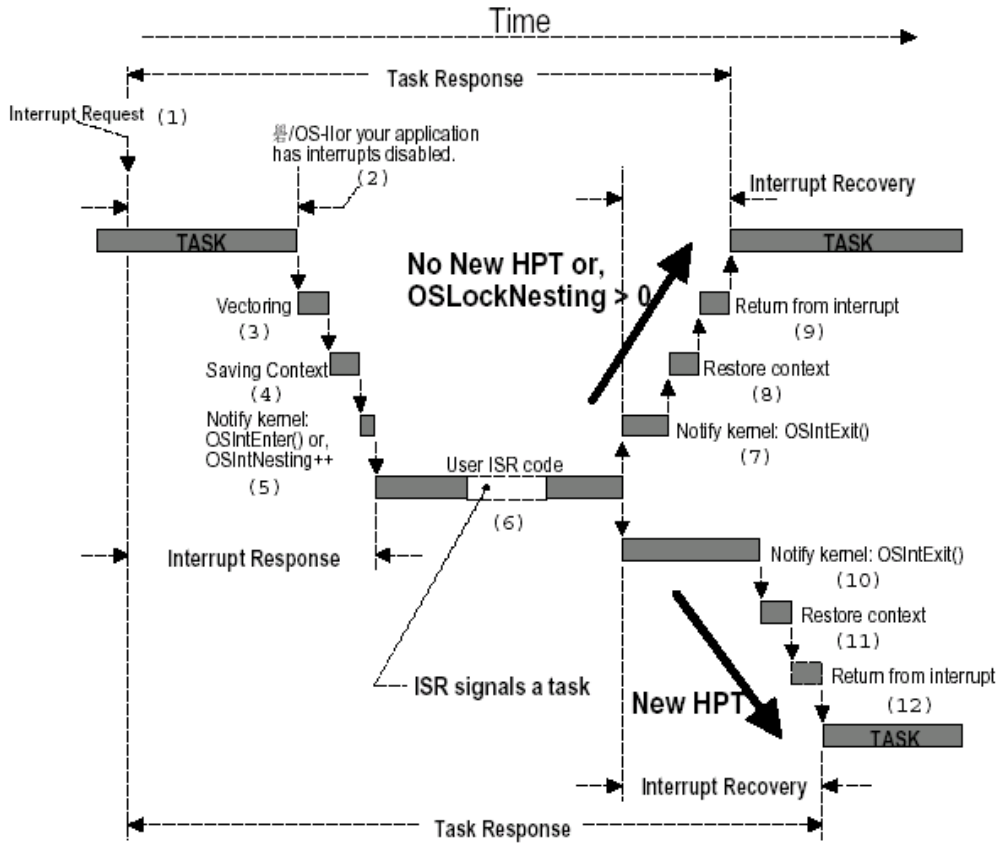


Figure 3-5, Servicing an interrupt

图 3.5 中断服务

进入中断函数 OSIntEnter()的代码如程序清单 L3.16 所示，从中断服务中退出函数 OSIntExit()的代码如程序清单 L3.17 所示。如前所述，OSIntEnter()所做的事是非常少的。

程序清单 L3.16 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 L3.17 通知μC/OS-II，脱离了中断服务

```
void OSIntExit (void)
{
    OS_ENTER_CRITICAL();                                          (1)
    if ((--OSIntNesting | OSLockNesting) == 0) {                  (2)
        OSIntExitY    = OSUnMapTbl[OSRdyGrp];                     (3)
```

```
        OSPrioHighRdy = (INT8U)((OSIntExitY << 3) +
                        OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OSIntCtxSw();                                                (4)
        }
    }
    OS_EXIT_CRITICAL();
}
```

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[L3.17(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 是保存在全程变量 OSIntExitY 中的[L3.17(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 9.04.03 节，中断任务切换函数）。最后一点，如果需要做任务切换，OSIntExit()将调用 OSIntCtxSw()[L3.17（4）]而不是调用 OS_TASK_SW(),正像在 OSSched()函数中那样。

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 L3.5(1)和图 F3.6(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[L3.15(4)和 F3.6(2)]。OSIntExit()中的进入临界段函数 OS_ENTER_CRITICAL()或许将 CPU 的状态字也推入了堆栈 L3.7(1)和 F3.6(3)。这取决于中断是怎么被关掉的（见第8章移植μC/OS-Ⅱ）。最后，调用 OSIntCtxSw()时的返回地址又被推入了堆栈[L3.17(4)和 F3.1(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 F3.6(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。

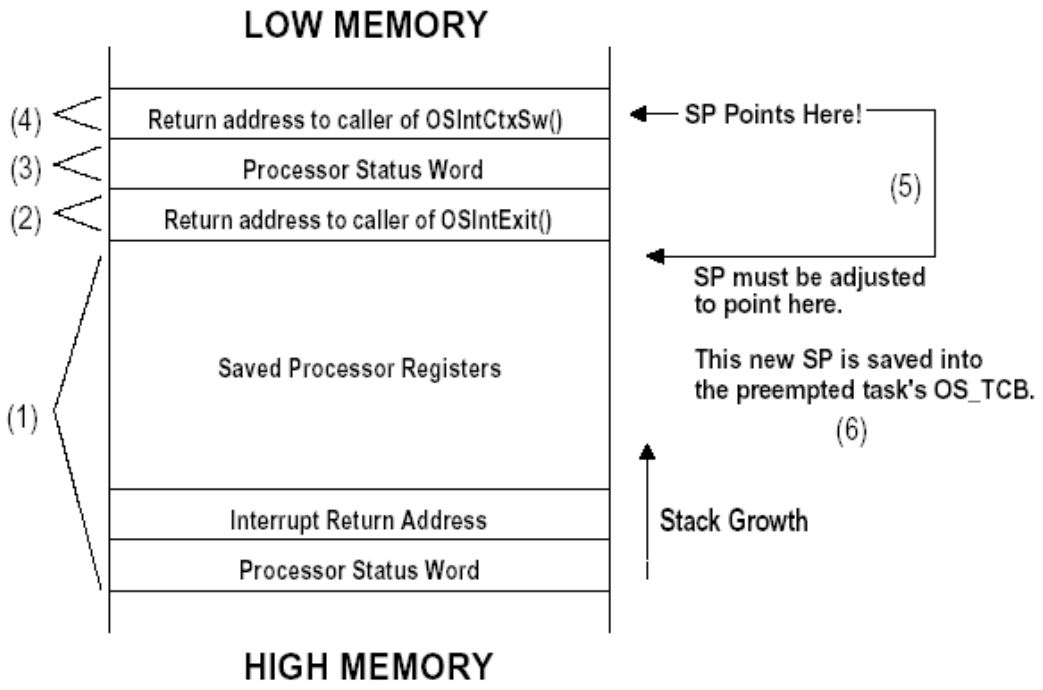


Figure 3-6, Cleanup by OSIntCtxSw().

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，如果用户中断服务子程序执行得非常快，用户不需要通知任务自身进入了中断服务，只要不在中断服务期间开中断，也不需要调用 OSIntEnter()或 OSIntNesting 加 1。程序清单 L3。18 中的示意代码表示这种情况。一个任务和这个中断服务子程序通讯的唯一方法是通过全程变量。

程序清单 L3.18 Motorola 68HC11中的中断服务子程序

```
M68HC11_ISR:                    /* 快中断服务程序，必须禁止中断*/
    所有寄存器被CPU自动保存;
    执行用户代码以响应中断;
    执行中断返回指令;
```

3.10 时钟节拍

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**再开启时钟节拍器，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 L3.19 所示。

程序清单 L3.19 启动时钟就节拍器的不正确做法.

```
void main(void)
{
    .
    .
    OSInit();                /* 初始化uC/OS-II                                  */
    .
    .
    /* 应用程序初始化代码 ...                                                    */
    /* ... 通过调用OSTaskCreate()创建至少一个任务                                */
    .
    .
    允许时钟节拍（TICKER）中断; /* 千万不要在这里允许时钟节拍中断!!!             */
    .
    .
    OSStart();               /* 开始多任务调度                                  */
}
```

这里潜在地危险是，时钟节拍中断有可能在μC/OS-Ⅱ启动第一个任务之前发生，此时μC/OS-Ⅱ是处在一种不确定的状态之中，用户应用程序有可能会崩溃。

μC/OS-Ⅱ中的时钟节拍服务是通过在中断服务子程序中调用 OSTimeTick()实现的。时钟节拍中断服从所有前面章节中描述的规则。时钟节拍中断服务子程序的示意代码如程序清单 L3.20 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 L3.20 时钟节拍中断服务子程序的示意代码

```
void OSTickISR(void)
{
    保存处理器寄存器的值；
    调用OSIntEnter()或是将OSIntNesting加1;
    调用OSTimeTick();

    调用OSIntExit();
    恢复处理器寄存器的值;
    执行中断返回指令;
}
```

时钟节拍函数 OSTimeTick()的代码如程序清单 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[L3.2(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 OSTCBList 开始，沿着 OS_TCB 链表做，一直做到空闲任务[L3.21(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[L3.21(5)]。而确切被任务挂起的函数 OSTaskSuspend()挂起的任务则不会进入就绪态[L3.21(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。

程序清单 L3.21 时钟节拍函数 OSTimtick() 的一个节拍服务

```
void OSTimeTick (void)
{
    OS_TCB *ptcb;

    OSTimeTickHook();                                                    (1)
    ptcb = OSTCBList;                                                    (2)
    while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {                            (3)
        OS_ENTER_CRITICAL();
        if (ptcb->OSTCBDly != 0) {
            if (--ptcb->OSTCBDly == 0) {
                if (!(ptcb->OSTCBStat & OS_STAT_SUSPEND)) {              (4)
                    OSRdyGrp             |= ptcb->OSTCBBitY;             (5)
                    OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                } else {
                    ptcb->OSTCBDly = 1;
                }
            }
        }
        ptcb = ptcb->OSTCBNext;
        OS_EXIT_CRITICAL();
    }
    OS_ENTER_CRITICAL();                                                 (6)
    OSTime++;                                                            (7)
    OS_EXIT_CRITICAL();
}
```

OSTimeTick()还通过调用 OSTime()[L3.21(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，可以从任务级调用 OSTimeTick()，如程序清单 L3.22 所示。要想这么做，得建立一个高于应用程序中所有其它任务优先级的任务。时钟节拍中断服务子程序利用信号量或邮箱发信号给这个高优先级的任务。

程序清单 L3.22 时钟节拍任务 TickTask() 作时钟节拍服务.

```
void TickTask (void *pdata)
{
    pdata = pdata;
    for (;;) {
        OSMboxPend(...);    /* 等待从时钟节拍中断服务程序发来的信号 */
        OSTimeTick();
    }
}
```

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 L3.23）。

程序清单L3.23时钟节拍中断服务函数OSTickISR()做节拍服务。

```
void OSTickISR(void)
{
    保存处理器寄存器的值；
    调用OSIntEnter()或是将OSIntNesting加1;

    发送一个‘空’消息(例如， (void *)1)到时钟节拍的邮箱;

    调用OSIntExit();
    恢复处理器寄存器的值;
    执行中断返回指令;
}
```

3.11 μC/OS-Ⅱ初始化

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

OSInit()建立空闲任务 idle task，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 F3.7 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_TCBs）是用双向链表链接在一起的。OSTCBList 指向这个链表的起始处。当建立一个任务时，这个任务总是被放在这个链表的起始处。换句话说，OSTCBList 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。

μC/OS-Ⅱ还初始化了 4 个空数据结构缓冲区，如图 F3.8 所示。每个缓冲区都是单向链表，允许 μC/OS-Ⅱ从缓冲区中迅速得到或释放一个缓冲区中的元素。注意，空任务控制块在空缓冲区中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。μC/OS-Ⅱ自动安排总的系统任务数 OS_N_SYS_TASKS(见文件 μC/OS-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，任务间通讯与同步中讨论。指向空存储区的指针表 OSMemFreeList 将在第 7 章存储管理中讨论。

3.12 μC/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 μC/OS-Ⅱ之前，用户至少要建立一个应用任务，如程序清单 L3.24 所示。

程序清单 L3.24 初始化和启动 μC/OS-Ⅱ

```
void main (void)
{
    OSInit();          /* 初始化uC/OS-II                          */
    .
    .
    通过调用OSTaskCreate()或OSTaskCreateExt()创建至少一个任务;
    .
    .
    OSStart();         /* 开始多任务调度!OSStart()永远不会返回 */
}
```

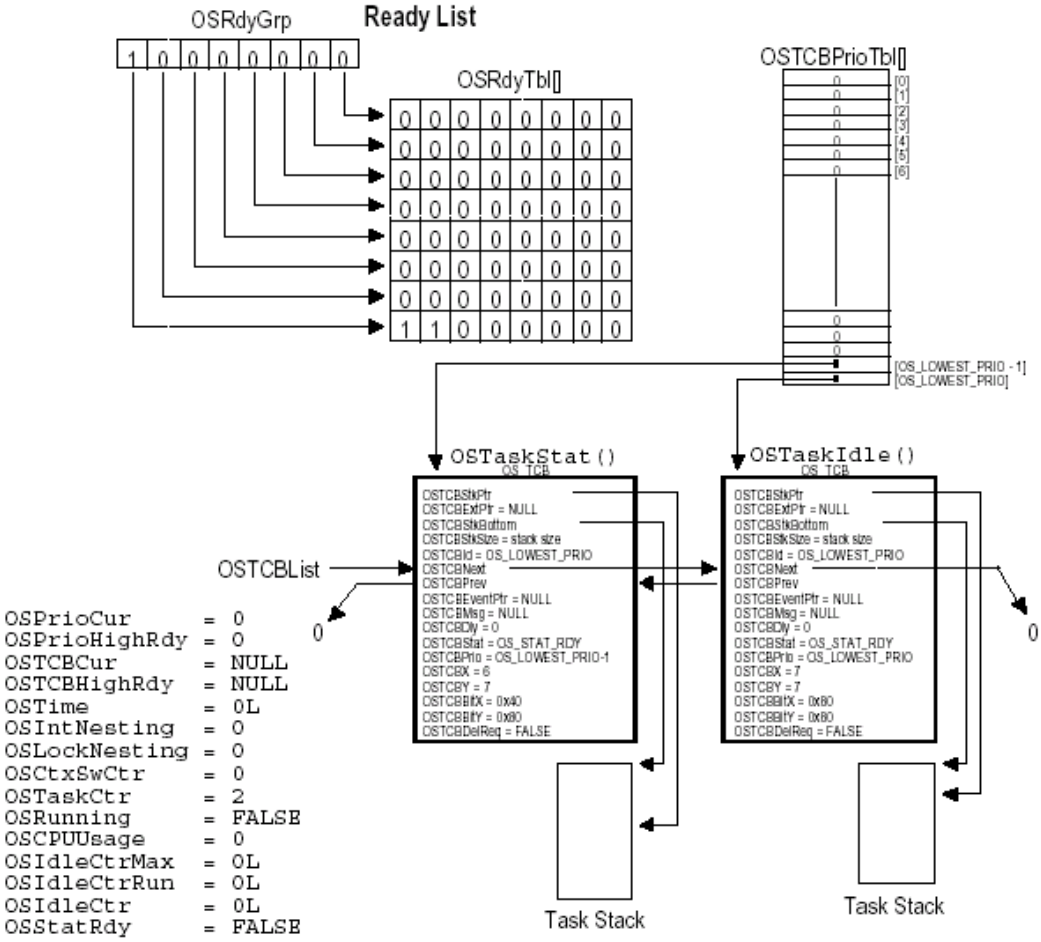


Figure 3-7, Data structures after calling OSInit()

图 3.7 调用 OSInit()之后的数据结构

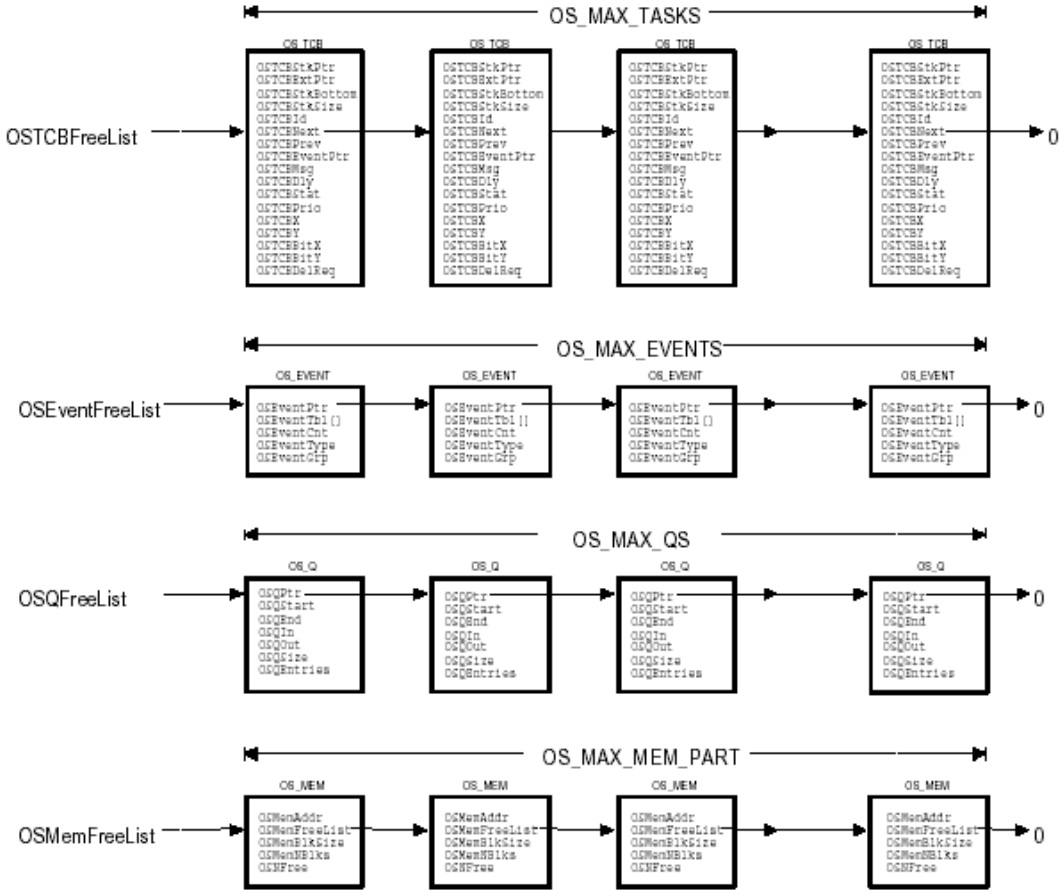


Figure 3-8, Free Pools

图 3.8 空缓冲区

OSStart()的代码如程序清单 L3.25 所示。当调用 OSStart()时，OSStart()从任务就绪表中找出那个用户建立的优先级最高任务的任务控制块[L3.25(1)]。然后，OSStart()调用高优先级就绪任务启动函数 OSStartHighRdy()[L3,25(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。见 9.04.01 节，高优先级就绪任务启动函数 OSStartHighRdy()。那一节详细介绍对于 80x86 微处理器是怎么做的。注意，OSStartHighRdy()将永远不返回到 OSStart()。

程序清单 L3.25 启动多任务.

```
void OSStart (void)
{
    INT8U y;
    INT8U x;


    if (OSRunning == FALSE) {
        y           = OSUnMapTbl[OSRdyGrp];
        x           = OSUnMapTbl[OSRdyTbl[y]];
```

```
        OSPrioHighRdy = (INT8U)((y << 3) + x);
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];                    (1)
        OSTCBCur      = OSTCBHighRdy;
        OSStartHighRdy();                                               (2)
    }
}
```

多任务启动以后变量与数据结构中的内容如图 F3.9 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。

3.13 获取当前μC/OS-II的版本号

应用程序调用 OSVersion()[程序清单 L3.26]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。

程序清单 L3.26 得到μC/OS-II当前版本号

```
INT16U OSVersion (void)
{
    return (OS_VERSION);
}
```

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

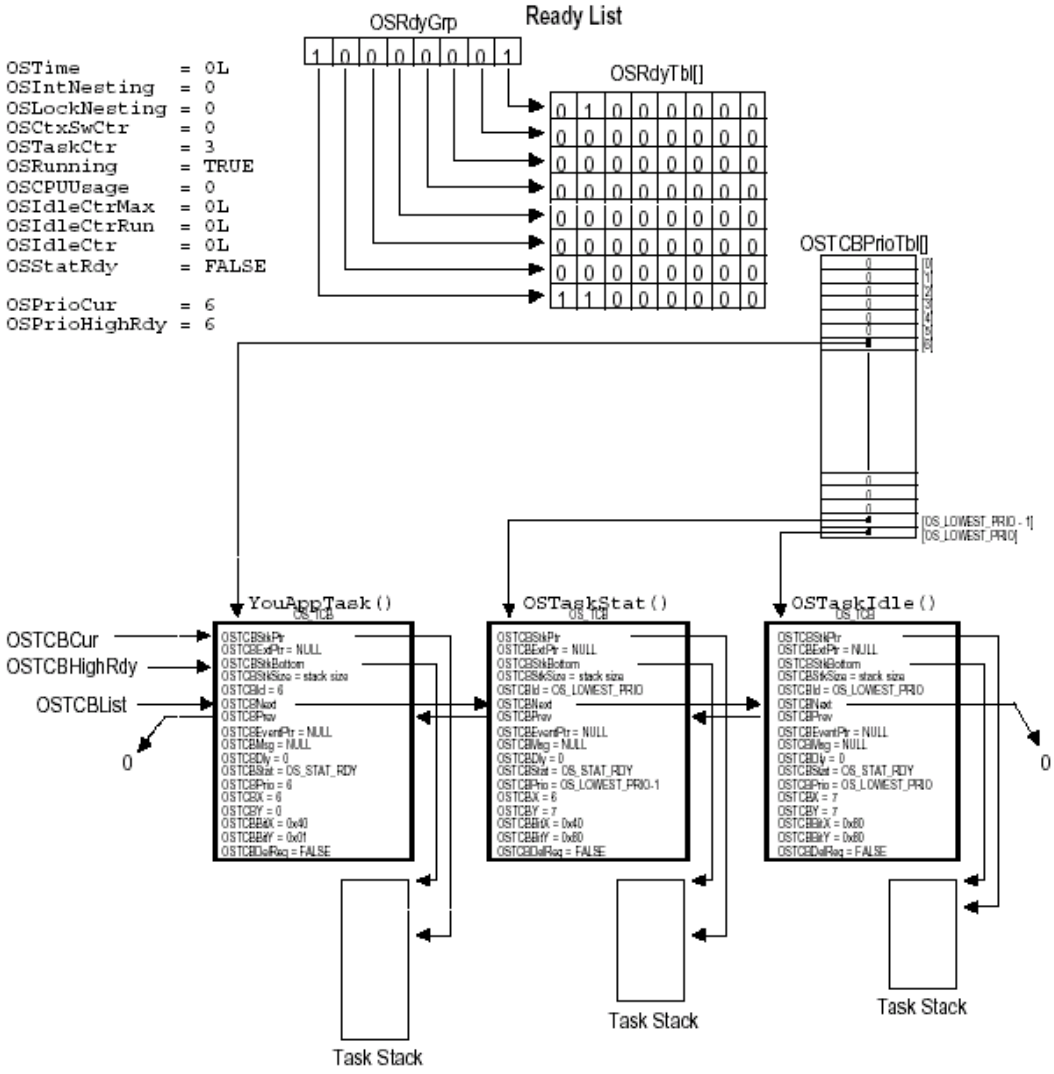


Figure 3-9, Variables and Data Structures after calling OSStart()

图 3.9 调用 OSStart()以后的变量与数据结构

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，任务间的通讯与同步。

第 4 章 任务管理

在前面的章节中，笔者曾说过任务可以是一个无限的循环，也可以是在一次执行完毕后被删除掉。这里要注意的是，任务代码并不是被真正的删除了，而只是μC/OS-Ⅱ不再理会该任务代码，所以该任务代码不会再运行。任务看起来与任何 C 函数一样，具有一个返回类型和一个参数，只是它从不返回。任务的返回类型必须被定义成 void 型。在本章中所提到的函数可以在 OS_TASK 文件中找到。如前所述，任务必须是以下两种结构之一：

```
void YourTask (void *pdata)
{
   for (;;) {
      /* 用户代码 */
      调用μC/OS-Ⅱ的服务例程之一:
         OSMboxPend();
         OSQPend();
         OSSemPend();
         OSTaskDel(OS_PRIO_SELF);
         OSTaskSuspend(OS_PRIO_SELF);
         OSTimeDly();
         OSTimeDlyHMSM();
      /* 用户代码 */
   }
}
```

或

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

本章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()

想让μC/OS-Ⅱ管理用户的任务，用户必须要先建立任务。用户可以通过传递任务地址和其它参数到以下两个函数之一来建立任务：OSTaskCreate() 或 OSTaskCreateExt()。OSTaskCreate()与μC/OS 是向下兼容的，OSTaskCreateExt()是 OSTaskCreate()的扩展版本，提供了一些附加的功能。用两个函数中的任何一个都可以建立任务。任务可以在多任务调度开始前建立，也可以在其它任务的执行过程中被建立。在开始多任务调度(即调用 OSStart())前，用户必须建立至少一个任务。任务不能由中断服务程序(ISR)来建立。

OSTaskCreate()的代码如程序清单 L4.1 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 4.02，任务堆栈)，prio 是分配给任务的优先级。

程序清单 L4.1 OSTaskCreate()

```
INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos,
INT8U prio)
{
   void   *psp;
   INT8U   err;



   if (prio > OS_LOWEST_PRIO) {                                        (1)
      return (OS_PRIO_INVALID);
   }
   OS_ENTER_CRITICAL();
   if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {                            (2)
      OSTCBPrioTbl[prio] = (OS_TCB *)1;                                (3)
      OS_EXIT_CRITICAL();                                              (4)
      psp = (void *)OSTaskStkInit(task, pdata, ptos, 0);               (5)
      err = OSTCBInit(prio, psp, (void *)0, 0, 0, (void *)0, 0);       (6)
      if (err == OS_NO_ERR) {                                          (7)
         OS_ENTER_CRITICAL();
         OSTaskCtr++;                                                  (8)
         OSTaskCreateHook(OSTCBPrioTbl[prio]);                         (9)
         OS_EXIT_CRITICAL();
         if (OSRunning) {                                             (10)
            OSSched();                                                (11)
         }
```

```
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;                          (12)
            OS_EXIT_CRITICAL();
        }
        return (err);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    }
}
```

OSTaskCreate()一开始先检测分配给任务的优先级是否有效[L4.1(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[L4.1(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[L4.1(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[L4.1(4)]。

然后，OSTaskCreate()调用 OSTaskStkInit()[L4.1(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——移植μC/OS-Ⅱ。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从上(高地址)往下(低地址)递减也可以从下往上递增。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(堆栈从下往上长)。

一旦 OSTaskStkInit()函数完成了建立堆栈的任务，OSTaskCreate()就调用 OSTCBInit()[L4.1(6)]，从空闲的 OS_TCB 池中获得并初始化一个 OS_TCB。OSTCBInit()的代码如程序清单 L4.2 所示，它存在于 OS_CORE.C 文件中而不是 OS_TASK.C 文件中。OSTCBInit()函数首先从 OS_TCB 缓冲池中获得一个 OS_TCB[L4.2(1)]，如果 OS_TCB 池中有空闲的 OS_TCB[L4.2(2)],它就被初始化[L4.2(3)]。注意一旦 OS_TCB 被分配，该任务的创建者就已经完全拥有它了，即使这时内核又创建了其它的任务，这些新任务也不可能对已分配的 OS_TCB 作任何操作，所以 OSTCBInit()在这时就可以允许中断，并继续初始化 OS_TCB 的数据单元。

程序清单 L 4.2 OSTCBInit()

```
INT8U OSTCBInit (INT8U  prio,    OS_STK *ptos,   OS_STK *pbos, INT16U id,
              INT16U stk_size, void   *pext,   INT16U  opt)
{
   OS_TCB *ptcb;



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

#if OS_TASK_DEL_EN
       ptcb->OSTCBDelReq   = 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_QS >= 2)) || OS_SEM_EN
        ptcb->OSTCBEventPtr  = (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     = OSTCBList;
        ptcb->OSTCBPrev     = (OS_TCB *)0;
        if (OSTCBList != (OS_TCB *)0) {
            OSTCBList->OSTCBPrev = ptcb;
        }
        OSTCBList            = 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[L4.2(5)]，它就禁止中断[L4.2(4)]。该双向链表开始于OSTCBList，而一个新任务的OS_TCB常常被插入到链表的表头。最后，该任务处于就绪状态[L4.2(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[L4.2(7)]。

现在，我可以继续讨论OSTaskCreate()(程序清单 L4.1)函数了。从OSTCBInit()返回后，OSTaskCreate()要检验返回代码[L4.1(7)]，如果成功，就增加OSTaskCtr[L4.1(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[L4.1(12)]以放弃该任务的优先级。然后，OSTaskCreate()调用

OSTaskCreateHook()[L4.1(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，或者其它与任务相关的内容。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)，也可以在其它地方定义。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[L4.1(10)])，则任务调度函数会被调用[L4.1(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 L4.3所示。

我们可以看到OSTaskCreateExt()需要九个参数！前四个参数(task,pdata,ptos和prio)与OSTaskCreate()的四个参数完全相同，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用于指定堆栈成员数目的容量。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，用户可以为每个任务增加一个名字(参看实例3)，或是在任务切换过程中将浮点寄存器的内容储存到这个附加数据域中，等等。

opt用于设定OSTaskCreateExt()的选项，指定是否允许堆栈检验，是否将堆栈清零，任务是否要进行浮点操作等等。μCOS_Ⅱ.H文件中有一个所有可能选项(OS_TASK_OPT_STK_CHK,OS_TASK_OPT_STK_CLR和OS_TASK_OPT_SAVE_FP)的常数表。每个选项占有opt的一位，并通过该位的置位来选定(用户在使用时只需要将以上OS_TASK_OPT_???选项常数进行位或(OR)操作就可以了)。

程序清单 L 4.3 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)
{
    void   *psp;
    INT8U   err;
    INT16U  i;
    OS_STK  *pfill;



    if (prio > OS_LOWEST_PRIO) {                                    (1)
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {                        (2)
        OSTCBPrioTbl[prio] = (OS_TCB *)1;                           (3)
        OS_EXIT_CRITICAL();                                         (4)

        if (opt & OS_TASK_OPT_STK_CHK) {                            (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++;                                                 (9)
            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[L4.3(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[L4.3(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[L4.3(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[L4.3(4)]。

为了对任务的堆栈进行检验[参看 4.03,堆栈检验，OSTaskStkChk()]，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[L4.3(5)]。

接着，OSTaskCreateExt()调用 OSTaskStkInit()[L4.3(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()的细节可参看第八章——移植μC/OS-Ⅱ。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 4.02,任务堆栈)。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() [L4.3(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 4.00 节)，从 OSTCBInit()返回后，OSTaskCreateExt()要检验返回代码[L4.3(8)]，如果成功，就增加 OSTaskCtr[L4.3(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[L4.3(13)]以放弃对该任务优先级的占用。然后，OSTaskCreateExt()调用 OSTaskCreateHook()[L4.3(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)，也可以在其它地方定义(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[L4.3(11)])，以任务调度函数会被调用[L4.3(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

每个任务都有自己的堆栈空间。堆栈必须声明为 OS_STK 类型，并且由连续的内存空间组成。用户可以静态分配堆栈空间(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 L4.4 和 4.5 所示，这两种声明应放置在函数的外面。

程序清单 L4.4 静态堆栈

```
static OS_STK  MyTaskStack[stack_size];
```

或

程序清单 L4.5 静态堆栈

```
OS_STK  MyTaskStack[stack_size];
```

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 L4.6 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 L L4.6 用malloc()为任务分配堆栈空间

```
OS_STK  *pstk;



pstk = (OS_STK *)malloc(stack_size);
if (pstk != (OS_STK *)0) {          /* 确认malloc()能得到足够地内存空间 */
    Create the task;
}
```

图 4.1 表示了一块能被 malloc()动态分配的 3K 字节的内存堆 [F4.1(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[F4.1(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[F4.1(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)。如果用户并不会去删除任务，使用 malloc()是非常可行的。

图 F4.1　　内存碎片

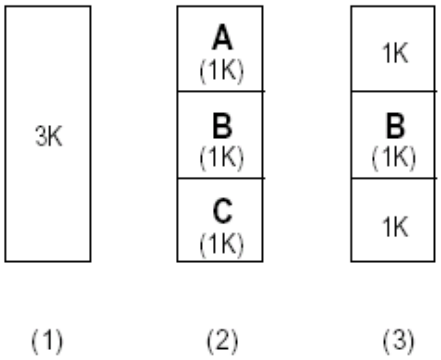


Figure 4-1, Fragmentation.

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)长也可以从下往上长(参看 4.02,任务堆栈)。用户在调用 OSTaskCreate()或 OSTaskCreateExt()的时候必须知道堆栈是怎样长的，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 L4.7 堆栈从下往上递增

```
OS_STK  TaskStack[TASK_STACK_SIZE];


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

当 OS_CPU.H 文件中的 OS_STK_GROWTH 置为 1 时，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 L4.8 堆栈从上往下递减

```
OS_STK  TaskStack[TASK_STACK_SIZE];


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

这个问题会影响代码的可移植性。如果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 L4.7 和 4.8 可重新写成如程序清单 L4.9 所示的形式。

程序清单 L 4.9 对两个方向增长的堆栈都提供支持

```
OS_STK  TaskStack[TASK_STACK_SIZE];



#if OS_STK_GROWTH == 0
    OSTaskCreate(task, pdata, &TaskStack[0], prio);
#else
    OSTaskCreate(task, pdata, &TaskStack[TASK_STACK_SIZE-1], prio);
#endif
```

任务所需的堆栈的容量是由应用程序指定的。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例程嵌套的堆栈需求。另外，用户的堆栈必须能储存所有的 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。μC/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长的堆栈[F4.2(1)]。μC/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，它不会自动地去不断检验任务的堆栈使用情况。在堆栈检验时，μC/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)[F4.2(2)]。另外，μC/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[F4.2(2)]。在任务建立的时候，BOS 的位置及堆栈的这两个值储存在任务的 OS_TCB 中。

为了使用μC/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

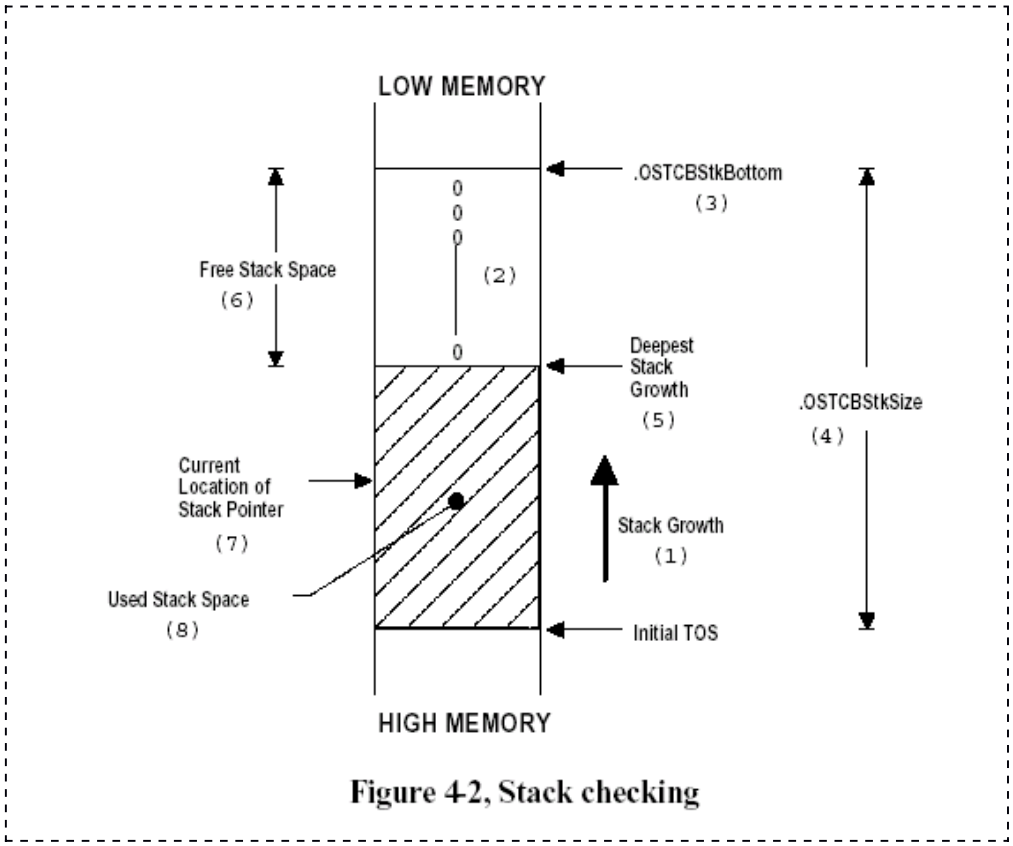


Figure 4-2, Stack checking

OSTaskStkChk()顺着堆栈的栈底开始计算空闲的堆栈空间大小，具体实现方法是统计储存值为 0 的连续堆栈入口的数目，直到发现储存值不为 0 的堆栈入口[F4.2(5)]。注意堆栈入口的储存值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，如果堆栈的入口有 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储存值为 0 的连续堆栈入口以后的大小。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，被检验的任务的堆栈指针可能会指向最初的堆栈栈顶(TOS)与堆栈最深处之间的任何位置[F4.2(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后容量了。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 L4.10 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它域，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[L4.10(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[L4.10(2)]。要执行堆栈检验，用户必须已用 OSTaskCreateExt()建立了任务并且已经传递了选项 OS_TASK_OPT_CHK[L4.10(3)]。如果所有的条件都满足了，OSTaskStkChk()就会象前面描述的那样从堆栈栈底开始统计堆栈的空闲空间[L4.10(4)]。最后，储存在 OS_STK_DATA 中的信息就被确定下来了[L4.10(5)]。注意函数所确定的是堆栈的实际空闲字节数和已被占用的字节数，而不是堆栈的总字节数。当然，堆栈的实际大小(用字节表示)就是该两项之和。

程序清单 L 4.10　堆栈检验函数

```
INT8U OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata)
{
    OS_TCB  *ptcb;
    OS_STK  *pchk;
    INT32U   free;
    INT32U   size;



    pdata->OSFree = 0;
    pdata->OSUsed = 0;
    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                                          (1)
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb == (OS_TCB *)0) {                                           (2)
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);
```

```
    }
    if ((ptcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) == 0) {                (3)
        OS_EXIT_CRITICAL();
        return (OS_TASK_OPT_ERR);
    }
    free = 0;                                                         (4)
    size = ptcb->OSTCBStkSize;
    pchk = ptcb->OSTCBStkBottom;
    OS_EXIT_CRITICAL();
#if OS_STK_GROWTH == 1
    while (*pchk++ == 0) {
        free++;
    }
#else
    while (*pchk-- == 0) {
        free++;
    }
#endif
    pdata->OSFree = free * sizeof(OS_STK);                            (5)
    pdata->OSUsed = (size - free) * sizeof(OS_STK);
    return (OS_NO_ERR);
}
```

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 3.02, 任务状态)，并不是说任务的代码被删除了，只是任务的代码不再被μC/OS-Ⅱ调用。通过调用 OSTaskDel()就可以完成删除任务的功能(如程序清单 L4.11 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[L4.11(1)]。不过，用户可以删除 statistic 任务[L4.11(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[L4.11(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[L4.11(4)]。接下来 OSTaskDel()会保证被删除的任务是确实存在的[L4.11(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 L 4.11　删除任务

```
INT8U OSTaskDel (INT8U prio)
{
   OS_TCB   *ptcb;
   OS_EVENT *pevent;

   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->OSTCBStat = OS_STAT_RDY;                                    (9)
      OSLockNesting++;                                                 (10)
      OS_EXIT_CRITICAL();                                              (11)
      OSDummy();                                                       (12)
```

```
        OS_ENTER_CRITICAL();
        OSLockNesting--;                                              (13)
        OSTaskDelHook(ptcb);                                          (14)
        OSTaskCtr--;
        OSTCBPrioTbl[prio] = (OS_TCB *)0;                             (15)
        if (ptcb->OSTCBPrev == (OS_TCB *)0) {                         (16)
            ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
            OSTCBList               = ptcb->OSTCBNext;
        } else {
            ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;
            ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;
        }
        ptcb->OSTCBNext = OSTCBFreeList;                              (17)
        OSTCBFreeList   = ptcb;
        OS_EXIT_CRITICAL();
        OSSched();                                                    (18)
        return (OS_NO_ERR);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_TASK_DEL_ERR);
    }
}
```

一旦所有条件都满足了，OS_TCB 就会从所有可能的μC/OS-Ⅱ的数据结构中移除。OSTaskDel()分两步完成该移除任务以减少中断响应时间。首先，如果任务处于就绪表中，它会直接被移除[L4.11(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除[L4.11(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[L4.11(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，而是阻止其它任务或 ISR 例程让该任务重新开始执行(即避免其它任务或 ISR 调用 OSTaskResume()[L4.11(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。

要被删除的任务不会被其它的任务或 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[L4.11(10)]在删除过程中切换到其它的任务中去，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[L4.11(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 链中解开 OS_TCB，并将 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()[L4.11(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（译者注：例如 MC68HC08 指令中的 NOP 指令），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μCOS-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[L4.11(13)]。接着，OSTaskDel()调用用户自定义的 OSTaskDelHook()函数[L4.11(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()简单地将指向被删除的任务的 OS_TCB 的指针指向 NULL[L4.11(15)]，从而达到将 OS_TCB 从优先级表中移除的目的。再接着，OSTaskDel()将被删除的任务的 OS_TCB 从 OS_TCB 双向链表中移除[L4.11(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，并允许其它任务的建立[L4.11(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[L4.11(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[L4.11(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 L4.12 所示。任务 B 需要决定在怎样的情况下请求删除任务[L4.12(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()[L4.12(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下面的工作，这时用户可以象笔者一样，通过让任务 B 延时一定时间来达到这个目的[L4.12(3)]。笔者延时了一个时钟节拍。如果需要，用户可以延时得更长一些。当任务 A 完全删除自己后，[L4.12(2)]中的返回值成为 OS_TASK_NOT_EXIST，此时循环结束[L4.12(4)]。

程序清单 L 4.12 请求删除其它任务的任务(任务B)

```
void RequestorTask (void *pdata)
```

```
{
    INT8U err;



    pdata = pdata;
    for (;;) {
        /* 应用程序代码 */
        if ('TaskToBeDeleted()' 需要被删除) {                          (1)
            while (OSTaskDelReq(TASK_TO_DEL_PRIO) != OS_TASK_NOT_EXIST) {(2)
                OSTimeDly(1);                                          (3)
            }
        }
        /*应用程序代码*/                                               (4)
    }
}
```

程序清单 L 4.13　需要删除自己的任务(任务A)

```
void TaskToBeDeleted (void *pdata)
{
    INT8U err;



    pdata = pdata;
    for (;;) {
        /*应用程序代码*/
        If (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ) {           (1)
            释放所有占用的资源;                                         (2)
            释放所有动态内存;
            OSTaskDel(OS_PRIO_SELF);                                   (3)
        } else {
            /*应用程序代码*/
        }
    }
}
```

需要删除自己的任务(任务 A)的代码如程序清单 L4.13 所示。在 OS_TAB 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 OSTaskDelReq(OS_PRIO_SELF) 而得到的。当 OSTaskDelReq() 返回给调用者 OS_TASK_DEL_REQ[L4.13(1)]时，则表明已经有另外的任务请求该任务被删除了。在这种情况下，被删除的任务会释放它所拥有的所用资源[L4.13(2)]，并且调用 OSTaskDel(OS_PRIO_SELF)来删除自己[L4.13(3)]。前面曾提到过，任务的代码没有被真正的删除，而只是μC/OS-Ⅱ不再理会该任务代码，换句话说，就是任务的代码不会再运行了。但是，用户可以通过调用 OSTaskCreate()或 OSTaskCreateExt()函数重新建立该任务。

OSTaskDelReq()的代码如程序清单 L4.14 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[L4.14(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[L4.14(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[L4.14(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[L4.14(4)]，OSTaskDelReq()就会设置任务的内部标志[L4.14(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[L4.14(6)]。

程序清单 L 4.14 OSTaskDelReq().

```
INT8U OSTaskDelReq (INT8U prio)
{
    BOOLEAN  stat;
    INT8U    err;
    OS_TCB  *ptcb;



    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);
    }
    if (prio == OS_PRIO_SELF) {                                          (3)
        OS_ENTER_CRITICAL();
        stat = OSTCBCur->OSTCBDelReq;
        OS_EXIT_CRITICAL();
        return (stat);
    } else {
```

```
        OS_ENTER_CRITICAL();
        if ((ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) {                (4)
            ptcb->OSTCBDelReq = OS_TASK_DEL_REQ;                         (5)
            err               = OS_NO_ERR;
        } else {
            err               = OS_TASK_NOT_EXIST;                       (6)
        }
        OS_EXIT_CRITICAL();
        return (err);
    }
}
```

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 L4.15 所示。用户不能改变空闲任务的优先级[L4.15(1)]，但用户可以改变调用本函数的任务或者其它任务的优先级。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是合法的（即不存在具有新优先级的任务）[L4.15(2)]。如果新优先级是合法的，μC/OS-Ⅱ通过将某些东西储存到 OSTCBPrioTbl[newprio]中保留这个优先级[L4.15(3)]。如此就使得 OSTaskChangePrio()可以重新允许中断，因为此时其它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[L4.15(4)]。而这些值用来将任务放入就绪表或从该表中移除(参看 3.04,就绪表)。

接着，OSTaskChangePrio()检验目前的任务是否想改变它的优先级[L4.15(5)]。然后，OSTaskChangePrio()检查想要改变优先级的任务是否存在[L4.15(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[L4.15(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[L4.15(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[L4.15(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[L4.15(9)]，然后在新的优先级下插入到就绪表中[L4.15(10)]。这儿需要注意的是，OSTaskChangePrio()所用的是重新计算的值[L4.15(4)]将任务插入就绪表中的。

如果任务已经就绪，它可能会正在等待一个信号量、一封邮件或是一个消息队列。如果OSTCBEventPtr 非空（不等于 NULL）[L4.15(8)]，OSTaskChangePrio()就会知道任务正在等待以上的某件事。如果任务在等待某一事件的发生，OSTaskChangePrio()必须将任务从事件控制块(参看 6.00,事件控制块)的等待队列(在旧的优先级下)中移除。并在新的优先级下将事件插入到等待队列中[L4.15(12)]。任务也有可能正在等待延时的期满(参看第五章－任务管理)或是被挂起(参看 4.07,挂起任务，OSTaskSuspend())。在这些情况下，从 L4.15(8)到 L4.15(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[L4.15(13)]。新的优先级被保存在OS_TCB中[L4.15(14)]，重新计算的值也被保存在OS_TCB中[L4.15(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[L4.15(16)]。

程序清单 L 4.15　OSTaskChangePrio().

```
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)  ||      (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;
            }
        }
        OSTCBPrioTbl[newprio] = ptcb;                               (13)
        ptcb->OSTCBPrio      = newprio;                             (14)
        ptcb->OSTCBY         = y;                                   (15)
        ptcb->OSTCBX         = x;
        ptcb->OSTCBBitY      = bity;
        ptcb->OSTCBBitX      = bitx;
        OS_EXIT_CRITICAL();
        OSSched();                                                  (16)
        return (OS_NO_ERR);
    } else {
        OSTCBPrioTbl[newprio] = (OS_TCB *)0;                        (17)
```

```
            OS_EXIT_CRITICAL();
            return (OS_PRIO_ERR);
        }
    }
}
```

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它任务。

OSTaskSuspend()函数的代码如程序清单 L4.16 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[L4.16(1)]，接着确认用户指定优先级是有效的[L4.16(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[L4.16(1)]在[L4.16(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[L4.16(3)]。用户也可以通过指定优先级来挂起调用本函数的任务[L4.16(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[L4.16(5)]。如果该任务存在的话，它就会从就绪表中被移除[L4.16(6)]。注意要被挂起的任务有可能没有在就绪表中，因为它有可能在等待事件的发生或延时的期满。在这种情况下，要被挂起的任务在 OSRdyTbl[]中对应的位已被清除了(即为 0)。再次清除该位，要比先检验该位是否被清除了再在它没被清除时清除它快得多，所以笔者没有检验该位而直接清除它。现在，OSTaskSuspend()就可以在任务的 OS_TCB 中设置 OS_STAT_SUSPEND 标志了，以表明任务正在被挂起[L4.16(7)]。最后，OSTaskSuspend()只有在被挂起的任务是调用本函数的任务本身的情况下才调用任务调度程序[L4.16(8)]。

程序清单 L 4.16 OSTaskSuspend().

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

```
   if (prio == OS_IDLE_PRIO) {                                          (1)
      return (OS_TASK_SUSPEND_IDLE);
   }
   if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {
(2)
      return (OS_PRIO_INVALID);
   }
   OS_ENTER_CRITICAL();
   if (prio == OS_PRIO_SELF) {                                          (3)
      prio = OSTCBCur->OSTCBPrio;
      self = TRUE;
   } else if (prio == OSTCBCur->OSTCBPrio) {                            (4)
      self = TRUE;
   } else {
      self = FALSE;
   }
   if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) {                     (5)
      OS_EXIT_CRITICAL();
      return (OS_TASK_SUSPEND_PRIO);
   } else {
      if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {          (6)
         OSRdyGrp &= ~ptcb->OSTCBBitY;
      }
      ptcb->OSTCBStat |= OS_STAT_SUSPEND;                               (7)
      OS_EXIT_CRITICAL();
      if (self == TRUE) {                                               (8)
         OSSched();
      }
      return (OS_NO_ERR);
   }
}
```

4.8 恢复任务，OSTaskResume()

在上一节中曾提到过，被挂起的任务只有通过调用 OSTaskResume()才能恢复。OSTaskResume()函数的代码如程序清单 L4.17 所示。因为 OSTaskSuspend()不能挂起空闲任务，所以必须得确认用户的应用程序不是在恢复空闲任务[L4.17(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，因为用户要需要操作它的任务控制块 OS_TCB[L4.17(2)]，并且该任务必须是被挂起的[L4.17(3)]。OSTaskResume()是通过清除 OSTCBStat 域中的 OS_STAT_SUSPEND 位来取消挂起的[L4.17(4)]。要使任务处于就绪状态，OS_TCBDly 域必须为 0[L4.17(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[L4.17(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[L4.17(7)]。

程序清单 L 4.17 OSTaskResume().

```
INT8U OSTaskResume (INT8U prio)
{
    OS_TCB   *ptcb;



    If (prio >= OS_LOWEST_PRIO) {                                          (1)
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    If ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) {                       (2)
        OS_EXIT_CRITICAL();
        return (OS_TASK_RESUME_PRIO);
    } else {
        if (ptcb->OSTCBStat & OS_STAT_SUSPEND) {                           (3)
            if (((ptcb->OSTCBStat &= ~OS_STAT_SUSPEND) == OS_STAT_RDY) &&(4)
                (ptcb->OSTCBDly  == 0)) {                                  (5)
                OSRdyGrp             |= ptcb->OSTCBBitY;                   (6)
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                OS_EXIT_CRITICAL();
                OSSched();                                                 (7)
            } else {
                OS_EXIT_CRITICAL();
```

```
            }
            return (OS_NO_ERR);
        } else {
            OS_EXIT_CRITICAL();
            return (OS_TASK_NOT_SUSPENDED);
        }
    }
}
```

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()来获得自身或其它应用任务的信息。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 L4.18 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 中指向其它 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 L 4.18 得到任务的信息

```
OS_TCB MyTaskData;


void MyTask (void *pdata)
{
    pdata = pdata;
    for (;;) {
        /*  用户代码                */
        err = OSTaskQuery(10, &MyTaskData);
        /* Examine error code ..       */
        /*  用户代码                 */
    }
}
```

OSTaskQuery()的代码如程序清单 L4.19 所示。注意，笔者允许用户查询所有的任务，包括空闲任务[L4.19(1)]。用户尤其需要注意的是不要改变OSTCBNext与OSTCBPrev的指向。

通常，OSTaskQuery()需要检验用户是否想知道当前任务的有关信息[L4.19(2)]以及该任务是否已经建立了[L4.19(3)]。所有的域是通过赋值语句一次性复制的而不是一个域一个域地复制的[L4.19(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 L 4.19　OSTaskQuery().

```
INT8U OSTaskQuery (INT8U prio, OS_TCB *pdata)
{
    OS_TCB *ptcb;


    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {              (1)
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                                       (2)
        prio = OSTCBCur->OSTCBPrio;
    }
    if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) {                 (3)
        OS_EXIT_CRITICAL();
        return (OS_PRIO_ERR);
    }
    *pdata = *ptcb;                                                   (4)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

第 5 章 时间管理

在 3.10 节时钟节拍中曾提到，μC/OS-Ⅱ(其它内核也一样)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节时钟节函数 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它的任务通过调用 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 L5.1 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[L5.1(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[L5.1(2)]。接着，这个延时节拍数会被保存在当前任务的 OS_TCB 中[L5.1(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，任务调度程序会执行下一个优先级最高的就绪任务。

程序清单 L 5.1 OSTimeDly().

```
void OSTimeDly (INT16U ticks)
{
   if (ticks > 0) {                                                    (1)
      OS_ENTER_CRITICAL();
      if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0)
{    (2)
          OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
```

```
        }
        OSTCBCur->OSTCBDly = ticks;                                    (3)
        OS_EXIT_CRITICAL();
        OSSched();                                                     (4)
    }
}
```

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，这种情况依然是存在的。图 F5.1 详细说明了整个过程。系统每隔 10ms 发生一次时钟节拍中断[F5.1(1)]。假如用户没有执行其它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[F5.1(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[F5.1(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[F5.1(4)]所示的 OSTimeDly(1)。μC/OS-Ⅱ会使该任务处于休眠状态直至下一个节拍的到来。当下一个节拍到来后，时钟节拍中断服务子程序会执行[F5.1(5)]，但是这一次由于没有高优先级的任务被执行，μC/OS-Ⅱ会立即执行申请延时一个时钟节拍的任务[F5.1(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，任务几乎就没有得到任何延时，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！

Figure 5.1　　Delay resolution.

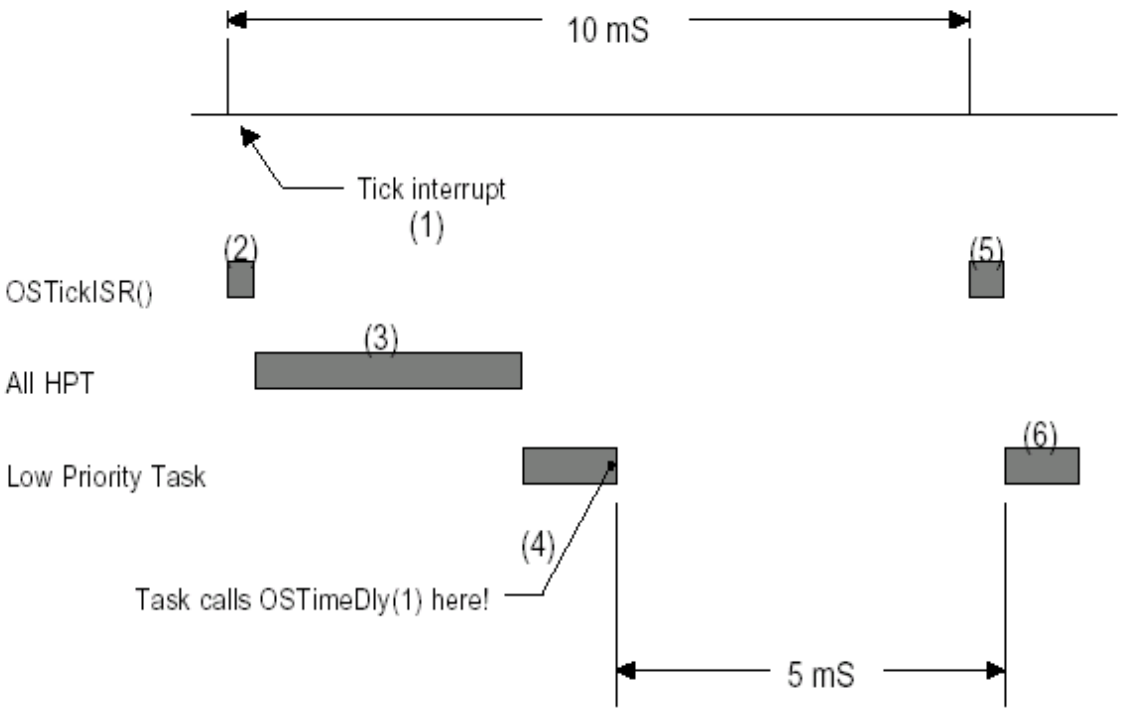


Figure 5-1, Delay resolution

5.1 按时分秒延时函数 OSTimeDlyHMSM()

OSTimeDly()虽然是一个非常有用的函数，但用户的应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(M)、秒(S)和毫秒(m)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它的任务通过调用 OSTimeDlyResume()取消了延时(参看 5.02,恢复延时的任务 OSTimeDlyResume())，它就会马上处于就绪态。同样，只有当该任务在所有就绪态任务中具有最高的优先级时，它才会立即运行。

程序清单 L5.2 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[L5.2(1)]。与 OSTimeDly()一样，即使用户没有定义延时，OSTimeDlyHMSM()也是存在的[L5.2(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[L5.2(3)]。很明显，程序清单 L5.2 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。[L5.2(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 65,535 个节拍。要想支持更长时间的延时，如 L5.2(2)所示，OSTimeDlyHMSM()确定了用户想延时多少次超过 65,535 个节拍的数目[L5.2(4)]和剩下的节拍数[L5.2(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 15x60x100=90,000 个时钟。这个延时会被分割成两次 32,768 个节拍的延时(因为用户只能延时 65,535 个节拍而不是 65536 个节拍)和一次 24,464 个节拍的延时。在这种情况下，OSTimeDlyHMSM()首先考虑剩下的节拍，然后是超过 65,535 的节拍数[L5.2(7)和(8)](即两个 32,768 个节拍延时)。

程序清单 L 5.2 OSTimeDlyHMSM().

```
INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U
milli)
{
    INT32U ticks;
    INT16U loops;



    if (hours > 0 || minutes > 0 || seconds > 0 || milli > 0) {       (1)
        if (minutes > 59) {
            return (OS_TIME_INVALID_MINUTES);
        }
        if (seconds > 59) {
            return (OS_TIME_INVALID_SECONDS);
        }
        If (milli > 999) {
            return (OS_TIME_INVALID_MILLI);
        }
        ticks = (INT32U)hours    * 3600L * OS_TICKS_PER_SEC              (2)
              + (INT32U)minutes  *   60L * OS_TICKS_PER_SEC
              + (INT32U)seconds  *        OS_TICKS_PER_SEC
              + OS_TICKS_PER_SEC * ((INT32U)milli
              + 500L/OS_TICKS_PER_SEC) / 1000L;                        (3)
        loops = ticks / 65536L;                                        (4)
        ticks = ticks % 65536L;                                        (5)
        OSTimeDly(ticks);                                              (6)
```

```
        while (loops > 0) {                                          (7)
            OSTimeDly(32768);                                        (8)
            OSTimeDly(32768);
            loops--;
        }
        return (OS_NO_ERR);
    } else {
        return (OS_TIME_ZERO_DLY);                                   (9)
    }
}
```

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()

μC/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，而是通过其它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件(参看第六章——任务间的通讯和同步)的任务，虽然这一点并没有提到过。在这种情况下，等待事件发生的任务会考虑是否终止等待事件。

OSTimeDlyResume()的代码如程序清单 L5.3 所示，它首先要确保指定的任务优先级有效 [L5.3(1)]。接着，OSTimeDlyResume()要确认要结束延时的任务是确实存在的[L5.3(2)]。如果任务存在，OSTimeDlyResume()会检验任务是否在等待延时期满[L5.3(3)]。只要 OS_TCB 域中的 OSTCBDly 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()或其它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[L5.3(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[L5.3(5)]。当上面的条件都满足后，任务就会被放在就绪表中[L5.3(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[L5.3(7)]。这会导致任务的切换。

程序清单 L 5.3 恢复正在延时的任务

```
INT8U OSTimeDlyResume (INT8U prio)
{
    OS_TCB *ptcb;

    if (prio >= OS_LOWEST_PRIO) {                                    (1)
```

```
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    ptcb = (OS_TCB *)OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {                                          (2)
        if (ptcb->OSTCBDly != 0) {                                      (3)
            ptcb->OSTCBDly  = 0;                                        (4)
            if (!(ptcb->OSTCBStat & OS_STAT_SUSPEND)) {                 (5)
                OSRdyGrp             |= ptcb->OSTCBBitY;                (6)
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                OS_EXIT_CRITICAL();
                OSSched();                                              (7)
            } else {
                OS_EXIT_CRITICAL();
            }
            return (OS_NO_ERR);
        } else {
            OS_EXIT_CRITICAL();
            return (OS_TIME_NOT_DLY);
        }
    } else {
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);
    }
}
```

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(参看第六章)。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。这种情况存在的唯一问题是它要求用户分配事件控制块(参看 6.00),因此用户的应用程序会多占用一些 RAM。

5.3 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 L5.4 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，而不能被中断等因素打断。

程序清单 L 5.4 得到和改变系统时间

```
INT32U OSTimeGet (void)
{
    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　任务之间的通讯与同步

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讯。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 3.00 节　临界段、8.03.02 节 OS_ENTER_CRITICAL()　和 OS_EXIT_CRITICAL()及 9.03.02 节 临界段，OS_CPU.H；

二是利用函数 OSSchedLock()和 OSSchekUnlock()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 3.06 节 给调度器上锁和开锁。

本章将介绍另外三种用于数据共享和任务通讯的方法：信号量、邮箱和消息队列。

图 F6.1 介绍了任务和中断服务子程序之间是如何进行通讯的。

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 Blocks）来向另外的任务发信号[F6.1A(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讯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[F6.1A(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[F6.1B]。在这种情况下，当该事件发生后，所有等待该事件的任务中，优先级最高的任务得到了该事件并进入就绪状态，准备执行。上面讲到的事件，可以是信号量、邮箱或者消息队列等。当事件控制块是一个信号量时，任务可以等待它，也可以给它发送消息。

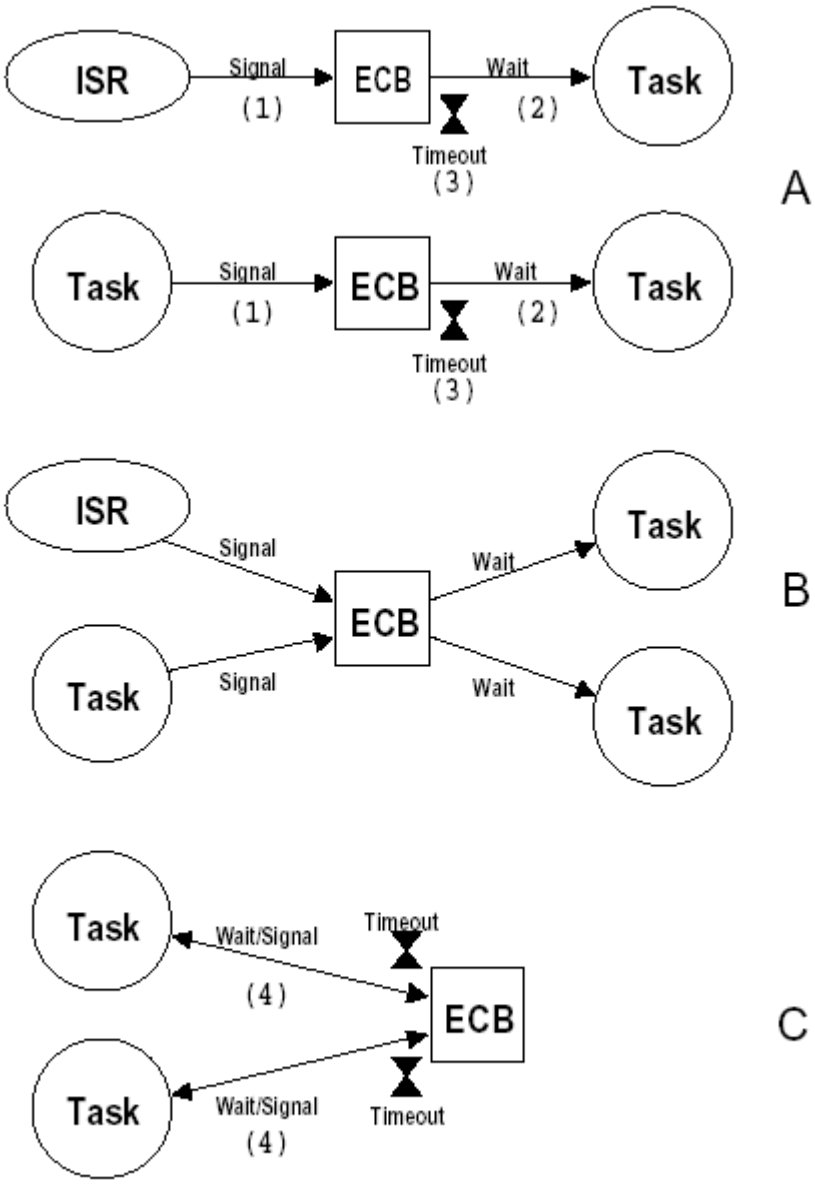


Figure 6-1, Use of Event Control Blocks.

图 6.1 事件控制块的使用

6.1 事件控制块 ECB

μC/OS-II 通过 uCOS_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 L6.1]，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 L6.1 是该数据结构的定义。

程序清单 L6.1 ECB数据结构

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

.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 6.06 节消息邮箱和 6.07 节消息队列。

.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 3.04 节 就绪表）

.OSEventCnt 当事件是一个信号量时，.OSEventCnt 是用于信号量的计数器，（见 6.05 节信号量）。

.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，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 F6.2 给出了.OSEventGrp 和.OSEventTbl[]之间的对应关系。该关系可以描述为：

当.OSEventTbl[0]中的任何一位为 1 时，.OSEventGrp 中的第 0 位为 1。

当.OSEventTbl[1]中的任何一位为 1 时，.OSEventGrp 中的第 1 位为 1。

当.OSEventTbl[2]中的任何一位为 1 时，.OSEventGrp 中的第 2 位为 1。

当.OSEventTbl[3]中的任何一位为 1 时，.OSEventGrp 中的第 3 位为 1。

当.OSEventTbl[4]中的任何一位为 1 时，.OSEventGrp 中的第 4 位为 1。

当.OSEventTbl[5]中的任何一位为 1 时，.OSEventGrp 中的第 5 位为 1。

当.OSEventTbl[6]中的任何一位为 1 时，.OSEventGrp 中的第 6 位为 1。

当.OSEventTbl[7]中的任何一位为 1 时，.OSEventGrp 中的第 7 位为 1。

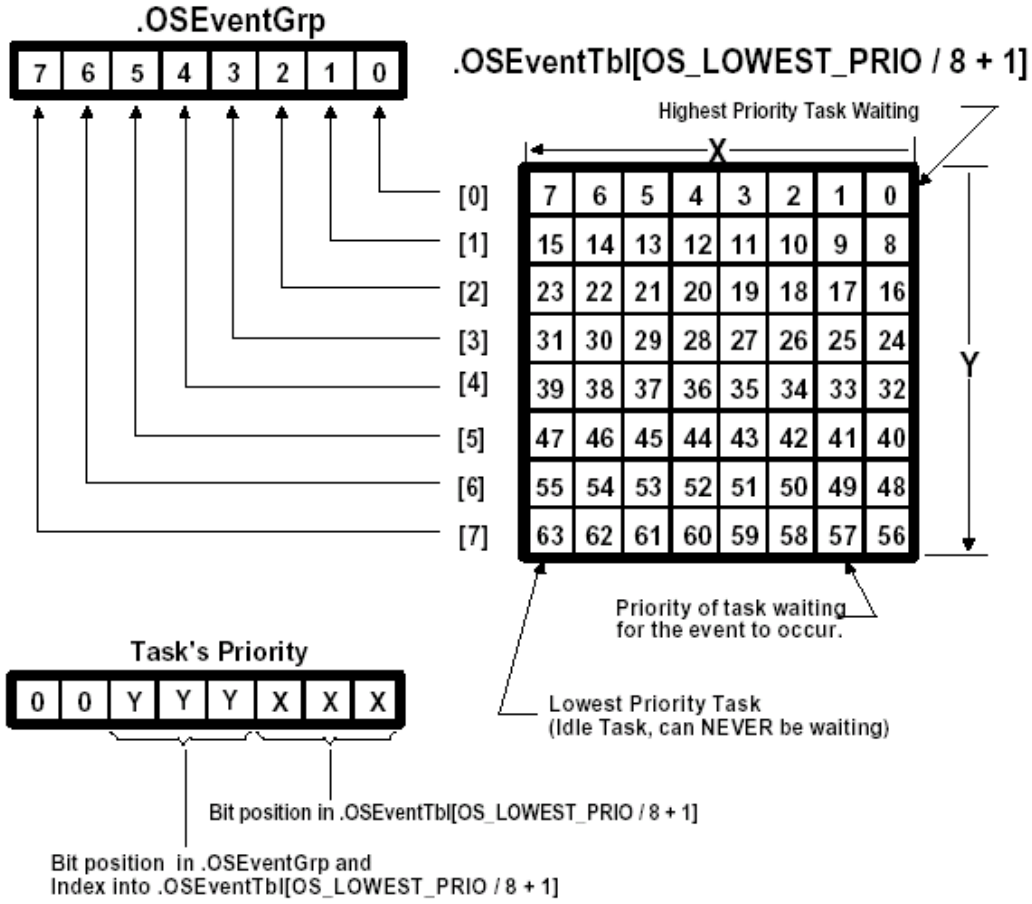


Figure 6-2, Wait list for task waiting for an event to occur.

图 F6.2 事件的等待任务列表

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 L6.2——将一个任务插入到事件的等待任务列表中

```
pevent->OSEventGrp               |= OSMapTbl[prio >> 3];
pevent->OSEventTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

其中，prio 是任务的优先级，pevent 是指向事件控制块的指针。

从程序清单 L6.2 可以看出，插入一个任务到等待任务列表中所花的时间是相同的，和表中现有多少个任务无关。从图 F6.2 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在.OSEventTbl[]中的字节索引。该算法中用到的查找表 OSMapTbl[]（定义在 OS_CORE.C 中）一般在 ROM 中实现。

表T6.1　　OSMapTbl[]

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

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 L6.3 所示。

程序清单 L6.3 从等待任务列表中删除一个任务

```
if ((pevent->OSEventTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0) {
    pevent->OSEventGrp &= ~OSMapTbl[prio >> 3];
}
```

该代码清除了任务在.OSEventTbl[]中的相应位，并且，如果其所在的组中不再有处于等待该事件的任务时（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，首先得到最高优先级任务所在的组的位置（0～7 之间的一个数）。然后利用.OSEventTbl[]中对应字节再在 OSUnMapTbl[]中查找，就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，最终就可以得到处于等待该事件状态的最高优先级任务了。程序清单 L6.4 是该算法的具体实现代码。

程序清单 L6.4 在等待任务列表中查找最高优先级的任务

```
y    = OSUnMapTbl[pevent->OSEventGrp];
x    = OSUnMapTbl[pevent->OSEventTbl[y]];
prio = (y << 3) + x;
```

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 3×8+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，μC/OS-II 的初始化），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 F6.3）。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。

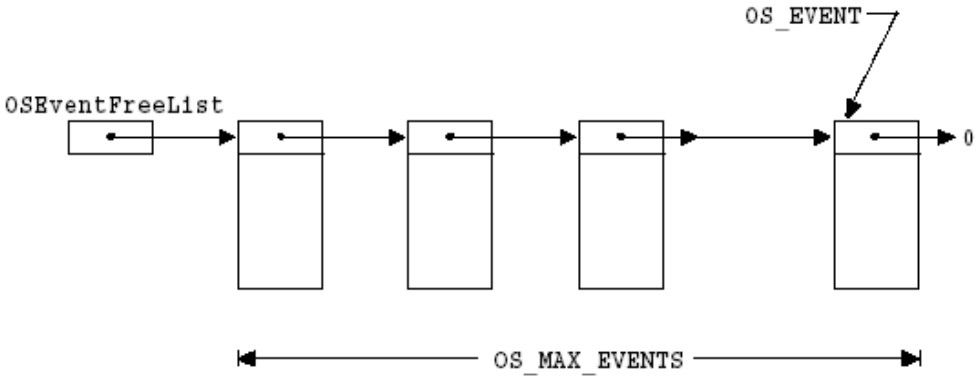


Figure 6-3, List of free ECBs.

图 F6.3 空闲事件控制块链表——Figure 6.3

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减短程代码长度，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()。

6.2 初始化一个事件控制块，OSEventWaitListInit()

程序清单 L6.5 是函数 OSEventWaitListInit()的源代码。当建立一个信号量、邮箱或者消息队列时，相应的建立函数 OSSemInit()，OSMboxCreate()，或者 OSQCreate()通过调用 OSEventWaitListInit()对事件控制块中的等待任务列表进行初始化。该函数初始化一个空的等待任务列表，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 L6.5 初始化ECB块的等待任务列表

```
void OSEventWaitListInit (OS_EVENT *pevent)
{
    INT8U i;


    pevent->OSEventGrp = 0x00;
```

```
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        pevent->OSEventTbl[i] = 0x00;
    }
}
```

6.3 使一个任务进入就绪态，OSEventTaskRdy()

程序清单 L6.6 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（Highest Priority Task），并把该任务置于就绪态。图 F6.4 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[L6.6/F6.4(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[L6.6/F6.4(2)]（从表 T6.1 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[L6.6/F6.4(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[L6.6/F6.4(4)]。根据以上结果，OSEventTaskRdy()函数计算出 HPT 任务的优先级[L6.6(5)]，然后就可以从等待任务列表中删除该任务了[L6.6(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[L6.6(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()直接将该域清澈 0[L6.6(8)]。因为该任务不再等待该事件的发生，所以 OSEventTaskRdy()函数将其任务控制块中指向事件控制块的指针指向 NULL[L6.6(9)]。如果 OSEventTaskRdy()是由 OSMboxPost()或者 OSQPost()调用的，该函数还要将相应的消息传递给 HPT，放在它的任务控制块中[L6.6(10)]。另外，当 OSEventTaskRdy()被调用时，位屏蔽码 msk 作为参数传递给它。该参数是用于对任务控制块中的位清零的位屏蔽码，和所发生事件的类型相对应[L6.6(11)]。最后，根据.OSTCBStat 判断该任务是否已处于就绪状态[L6.6(12)]。如果是，则将 HPT 插入到μC/OS-II 的就绪任务列表中[L6.6(13)]。注意，HPT 任务得到该事件后不一定进入就绪状态，也许该任务已经由于其它原因挂起了。[见 4.07 节，挂起一个任务，OSTaskSuspend()，和 4.08 节，恢复一个任务，OSTaskResume()]。

另外，.OSEventTaskRdy()函数要在中断禁止的情况下调用。

程序清单 L6.6 使一个任务进入就绪状态

```
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];                              (1)
    bity = OSMapTbl[y];                                                 (2)
    x    = OSUnMapTbl[pevent->OSEventTbl[y]];                           (3)
    bitx = OSMapTbl[x];                                                 (4)
    prio = (INT8U)((y << 3) + x);                                       (5)
    if ((pevent->OSEventTbl[y] &= ~bitx) == 0) {                        (6)
        pevent->OSEventGrp &= ~bity;
    }
    ptcb                 =  OSTCBPrioTbl[prio];                         (7)
    ptcb->OSTCBDly       =  0;                                          (8)
    ptcb->OSTCBEventPtr  = (OS_EVENT *)0;                               (9)
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    ptcb->OSTCBMsg       = msg;                                        (10)
#else
    msg                  = msg;
#endif
    ptcb->OSTCBStat     &= ~msk;                                       (11)
    if (ptcb->OSTCBStat == OS_STAT_RDY) {                              (12)
        OSRdyGrp        |=  bity;                                      (13)
        OSRdyTbl[y]     |=  bitx;
    }
}
```

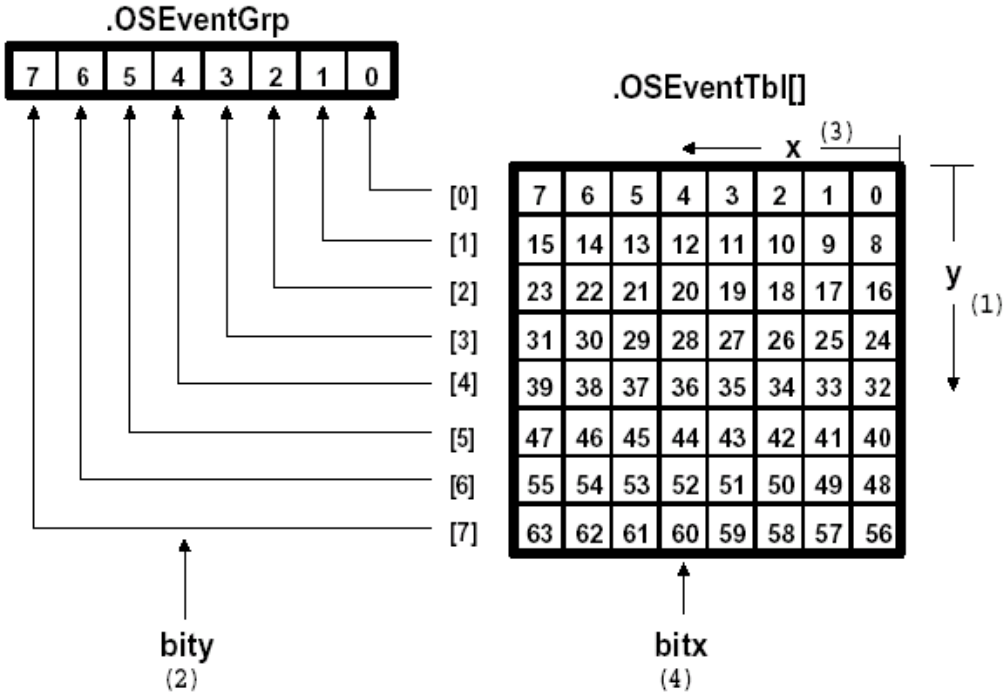


Figure 6-4, Making a task ready-to-run.

图 F6.4 使一个任务进入就绪状态——Figure 6.4

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

程序清单 L6.7 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。

程序清单 L6.7 使一个任务进入等待状态

```
void OSEventTaskWait (OS_EVENT *pevent)
{
    OSTCBCur->OSTCBEventPtr = pevent;                                    (1)
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {     (2)
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;        (3)
    pevent->OSEventGrp                   |= OSTCBCur->OSTCBBitY;
}
```

在该函数中，首先将指向事件控制块的指针放到任务的任务控制块中 [L6.7(1)]，接着将任务从就绪任务表中删除[L6.7(2)]，并把该任务放到事件控制块的等待任务表中[L6.7(3)]。

6.5 由于等待超时而将任务置为就绪态，OSEventTO()

程序清单 L6.8 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[L6.8(1)]，并把它置成就绪状态[L6.8(2)]。最后，从任务控制块中将指向事件控制块的指针删除[L6.8(3)]。用户应当注意，调用 OSEventTO()也应当先关中断。

程序清单 L6.8 因为等待超时将任务置为就绪状态

```
void  OSEventTO (OS_EVENT *pevent)
{
    if ((pevent->OSEventTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0)
{                                                                          (1)
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStat     = OS_STAT_RDY;                                 (2)
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                               (3)
}
```

6.6 信号量

μC/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 65,535 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μC/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（见下一节），对信号量的初始计数值赋值。该初始值为 0 到 65,535 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。

μC/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 F6.5 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 F6.5 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。

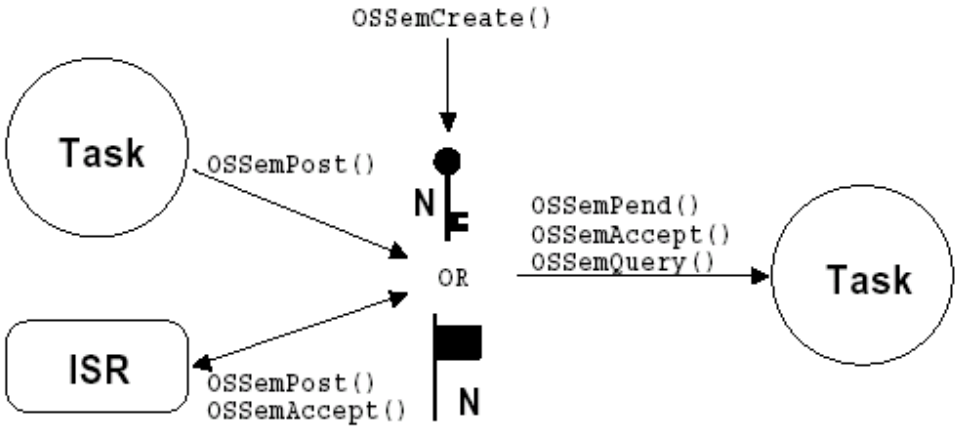


Figure 6-5, Relationship between tasks, ISRs and a semaphore.

图 F6.5 任务、中断服务子程序和信号量之间的关系——Figure 6.5

6.6.1 建立一个信号量，OSSemCreate()

程序清单 L6.9 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[L6.9(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[L6.9(2)]。如果这时有任务控制块可用[L6.9(3)]，就将该任务控制块的事件类型设置成信号量 OS_EVENT_TYPE_SEM[L6.9(4)]。其它的信号量操作函数 OSSem???()通过检查该域来保证所操作的任务控制块类型的正确。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[6.06 节，邮箱]。接着，用信号量的初始值对任务控制块进行初始化[L6.9(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 6.01 节，初始化一个任务控制块，OSEventWaitListInit()][L6.9(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，OSSemCreate()返回给调用函数一个指向任务控制块的指针。以后对信号量的所有操作，如 OSSemPend()，OSSemPost()，

OSSemAccept()和 OSSemQuery()都是通过该指针完成的。因此，这个指针实际上就是该信号量的句柄。如果系统中没有可用的任务控制块，OSSemCreate()将返回一个 NULL 指针。

值得注意的是，在μC/OS-II 中，信号量一旦建立就不能删除了，因此也就不可能将一个已分配的任务控制块再放回到空闲 ECB 链表中。如果有任务正在等待某个信号量，或者某任务的运行依赖于某信号量的出现时，删除该任务是很危险的。

程序清单 L6.9 建立一个信号量

```
OS_EVENT *OSSemCreate (INT16U cnt)
{
    OS_EVENT *pevent;



    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                          (1)
    if (OSEventFreeList != (OS_EVENT *)0) {                            (2)
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                                     (3)
        pevent->OSEventType = OS_EVENT_TYPE_SEM;                       (4)
        pevent->OSEventCnt  = cnt;                                     (5)
        OSEventWaitListInit(pevent);                                   (6)
    }
    return (pevent);                                                   (7)
}
```

6.6.2 等待一个信号量，OSSemPend()

程序清单 L6.10 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[L6.10(1)]。如果信号量当前是可用的（信号量的计数值大于 0）[L6.10(2)]，将信号量的计数值减 1[L6.10(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[L6.10(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（见下节）。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号有效或者等待超时。如果该参数值为 0，该任务将一直等待下去。OSSemPend()函数通过将任务控制块中的状态标志.OSTCBStat 置 1，把任务置于睡眠状态[L6.10(5)]，等待时间也同时置入任务控制块中[L6.10(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.OSTCBDly 域做递减操作（只要该域不为 0）[见 3.10 节，时钟节拍]。真正将任务置入睡眠状态的操作在 OSEventTaskWait()函数中执行 [见 6.03 节，让一个任务等待某个事件，OSEventTaskWait()][L6.10(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[L6.10(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[L6.10(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[L6.10(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[L6.10(11)]。

程序清单 L6.10 等待一个信号量

```
void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                    (1)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
    }
    if (pevent->OSEventCnt > 0) {                                      (2)
        pevent->OSEventCnt--;                                          (3)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) {                                     (4)
```

```
        OS_EXIT_CRITICAL();
        *err = OS_ERR_PEND_ISR;
    } else {
        OSTCBCur->OSTCBStat    |= OS_STAT_SEM;                          (5)
        OSTCBCur->OSTCBDly      = timeout;                              (6)
        OSEventTaskWait(pevent);                                        (7)
        OS_EXIT_CRITICAL();
        OSSched();                                                      (8)
        OS_ENTER_CRITICAL();
        if (OSTCBCur->OSTCBStat & OS_STAT_SEM) {                        (9)
            OSEventTO(pevent);                                         (10)
            OS_EXIT_CRITICAL();
            *err = OS_TIMEOUT;
        } else {
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                   (11)
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
        }
    }
}
```

6.6.3 发送一个信号量，OSSemPost()

程序清单 L6.11 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[L6.11(1)]，接着检查是否有任务在等待该信号量[L6.11(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 6.02 节，使一个任务进入就绪状态，OSEventTaskRdy()]，把其中的最高优先级任务从等待任务列表中删除[L6.11(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[L6.11(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的]，准备执行该就绪任务。如果不是，OSSched()直接返回，调用 OSSemPost()的任务得以继续执行。如果这时没有任务在等待该信号量，该信号量的计数值就简单地加 1[L6.11(5)]。

上面是由任务调用 OSSemPost()时的情况。当中断服务子程序调用该函数时，不会发生上面的任务切换。如果需要，任务切换要等到中断嵌套的最外层中断服务子程序调用 OSIntExit()函数后才能进行（见 3.09 节，μC/OS-II 中的中断）。

程序清单 L6.11 发出一个信号量

```
INT8U OSSemPost (OS_EVENT *pevent)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                                          (2)
        OSEventTaskRdy(pevent, (void *)0, OS_STAT_SEM);                (3)
        OS_EXIT_CRITICAL();
        OSSched();                                                     (4)
        return (OS_NO_ERR);
    } else {
        if (pevent->OSEventCnt < 65535) {
            pevent->OSEventCnt++;                                      (5)
            OS_EXIT_CRITICAL();
            return (OS_NO_ERR);
        } else {
            OS_EXIT_CRITICAL();
            return (OS_SEM_OVF);
        }
    }
}
```

6.6.4 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。这种情况下的操作是由OSSemAccept()函数完成的，其源代码见程序清单 L6.12。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OSSemCreate()函数建立的[L6.12(1)]，接着从该信号量的事件控制块中取出当前计数值[L6.12(2)]，并检查该信号量是否有效（计数值是否为非0值）[L6.12(3)]。如果有效，则将信号量的计数值减1[L6.12(4)]，然后将信号量的原有计数值返回给调用函数[L6.12(5)]。调用函数需要对该返回值进行检查。如果该值是0，说明该信号量无效。如果该值大于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资源数。应该注意的是，这些可用资源中，已经被该调用函数自身占用了一个（该计数值已经被减1）。中断服务子程序要请求信号量时，只能用OSSemAccept()而不能用OSSemPend()，因为中断服务子程序是不允许等待的。

程序清单 L6.12 无等待地请求一个信号量

```
INT16U OSSemAccept (OS_EVENT *pevent)
{
    INT16U cnt;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                    (1)
        OS_EXIT_CRITICAL();
        return (0);
    }
    cnt = pevent->OSEventCnt;                                          (2)
    if (cnt > 0) {                                                     (3)
        pevent->OSEventCnt--;                                          (4)
    }
    OS_EXIT_CRITICAL();
    return (cnt);                                                      (5)
}
```

6.6.5 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 L6.13]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。在这里，之所以使用一个新的数据结构的原因在于，调用函数应该只关心那些和特定信号量有关的信息,而不是象 OS_EVENT 数据结构包含的很全面的信息。该数据结构只包含信号量计数值.OSCnt 和等待任务列表.OSEventTbl[]、.OSEventGrp，而 OS_EVENT 中还包含了另外的两个域.OSEventType 和.OSEventPtr。

和其它与信号量有关的函数一样，OSSemQuery()也是先检查 pevent 指向的事件控制块是否是 OSSemCreate()产生的[L6.13(1)]，然后将等待任务列表[L6.13(2)]和计数值[L6.13(3)]从 OS_EVENT 结构拷贝到 OS_SEM_DATA 结构变量中去。

程序清单 L6.13 查询一个信号量的状态

```
INT8U OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata)
{
```

```
    INT8U  i;
    INT8U *psrc;
    INT8U *pdest;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;                            (2)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdest++ = *psrc++;
    }
    pdata->OSCnt      = pevent->OSEventCnt;                            (3)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

6.7 邮箱

邮箱是μC/OS-II 中另一种通讯机制，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（见下节），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 F6.6 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 F6.6 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 F6.6 任务、中断服务子程序和邮箱之间的关系

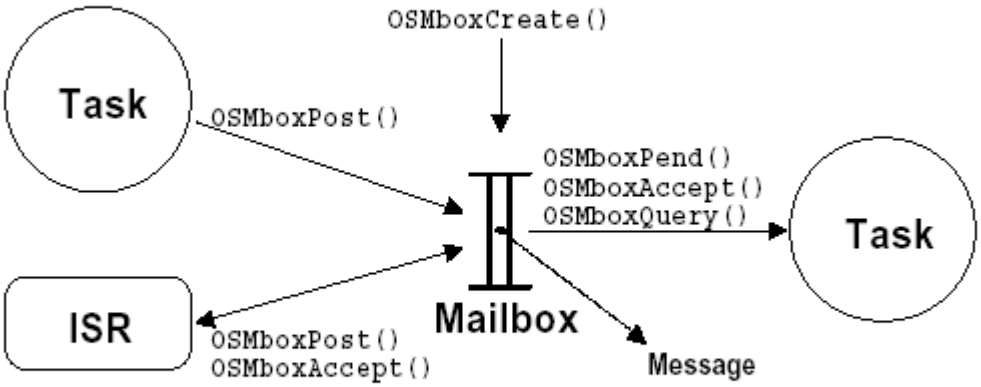


Figure 6-6, Relationship between tasks, ISRs and a message mailbox.

6.7.1 建立一个邮箱，OSMboxCreate()

程序清单 L6.14 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[L6.14(1)]，以及使用.OSEventPtr 域来容纳消息指针，而不是使用.OSEventCnt 域[L6.14(2)]。

OSMboxCreate()函数的返回值是一个指向事件控制块的指针[L6.14(3)]。这个指针在调用函数 OSMboxPend()，OSMboxPost()，OSMboxAccept()和 OSMboxQuery()时使用。因此，该指针可以看作是对应邮箱的句柄。值得注意的是，如果系统中已经没有事件控制块可用，函数 OSMboxCreate()将返回一个 NULL 指针。

邮箱一旦建立，是不能被删除的。比如，如果有任务正在等待一个邮箱的信息，这时删除该邮箱，将有可能产生灾难性的后果。

程序清单 L6.14 建立一个邮箱

```
OS_EVENT *OSMboxCreate (void *msg)
{
    OS_EVENT *pevent;
```

```
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;
    if (OSEventFreeList != (OS_EVENT *)0) {
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {
        pevent->OSEventType = OS_EVENT_TYPE_MBOX;                    (1)
        pevent->OSEventPtr  = msg;                                   (2)
        OSEventWaitListInit(pevent);
    }
    return (pevent);                                                 (3)
}
```

6.7.2 等待一个邮箱中的消息，OSMboxPend()

程序清单 L6.15 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[L6.15(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[L6.15(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[L6.15(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[L6.15(4)]。象 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()函数，也一样是成功的。

如果邮箱中没有可用的消息，OSMboxPend()的调用任务就被挂起，直到邮箱中有了消息或者等待超时[L6.15(5)]。当有其它的任务向该邮箱发送了消息后（或者等待时间超时），这时，该任务再一次成为最高优先级任务，OSSched()返回。这时，OSMboxPend()函数要检查是否有消息被放到该任务的任务控制块中[L6.15(6)]。如果有，那么该次函数调用成功，对应的消息被返回到调用函数。

程序清单 L6.15 等待一个邮箱中的消息

```
void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    void  *msg;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                    (1)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
    msg = pevent->OSEventPtr;
    if (msg != (void *)0) {                                             (2)
        pevent->OSEventPtr = (void *)0;                                 (3)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) {                                      (4)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_PEND_ISR;
    } else {
        OSTCBCur->OSTCBStat |= OS_STAT_MBOX;                            (5)
        OSTCBCur->OSTCBDly   = timeout;
        OSEventTaskWait(pevent);
        OS_EXIT_CRITICAL();
        OSSched();
        OS_ENTER_CRITICAL();
        if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) {                  (6)
            OSTCBCur->OSTCBMsg      = (void *)0;
            OSTCBCur->OSTCBStat     = OS_STAT_RDY;
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
            OS_EXIT_CRITICAL();
            *err                    = OS_NO_ERR;
        } else if (OSTCBCur->OSTCBStat & OS_STAT_MBOX) {                (7)
            OSEventTO(pevent);                                          (8)
            OS_EXIT_CRITICAL();
            msg                     = (void *)0;                        (9)
            *err                    = OS_TIMEOUT;
        } else {
            msg                     = pevent->OSEventPtr;              (10)
            pevent->OSEventPtr      = (void *)0;
```

```
(11)
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                    (12)
            OS_EXIT_CRITICAL();
            *err                    = OS_NO_ERR;
        }
    }
    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[L6.15(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[L6.15(8)]。因为此时邮箱中没有消息，所以返回的指针是 NULL[L6.15(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[L6.15(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，并且要将任务任务控制块中指向邮箱事件控制块的指针删除[L6.15(12)]。

6.7.3 发送一个消息到邮箱中，OSMboxPost()

程序清单 L6.16 是 OSMboxPost()函数的源代码。检查了事件控制块是否是一个邮箱后[L6.16(1)]，OSMboxPost()函数还要检查是否有任务在等待该邮箱中的消息[L6.16(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 6.02 节，使一个任务进入就绪状态，OSEventTaskRdy()][L6.16(3)]，加入系统的就绪任务列表中，准备运行。然后，调用 OSSched()函数[L6.16(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，指向消息的指针就被保存到邮箱中[L6.16(6)]（假设此时邮箱中的指针不是非 NULL 的[L6.16(5)]）。这样，下一个调用 OSMboxPend()函数的任务就可以立刻得到该消息了。

注意，如果 OSMboxPost()函数是从中断服务子程序中调用的，那么，这时并不发生上下文的切换。如果需要，中断服务子程序引起的上下文切换只发生在中断嵌套的最外层中断服务子程序对 OSIntExit()函数的调用时（见 3.09 节，μC/OS-II 中的中断）。

程序清单 L6.16 向邮箱中发送一条消息

```
INT8U OSMboxPost (OS_EVENT *pevent, void *msg)
{
```

```
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                                           (2)
        OSEventTaskRdy(pevent, msg, OS_STAT_MBOX);                      (3)
        OS_EXIT_CRITICAL();
        OSSched();                                                      (4)
        return (OS_NO_ERR);
    } else {
        if (pevent->OSEventPtr != (void *)0) {                          (5)
            OS_EXIT_CRITICAL();
            return (OS_MBOX_FULL);
        } else {
            pevent->OSEventPtr = msg;                                   (6)
            OS_EXIT_CRITICAL();
            return (OS_NO_ERR);
        }
    }
}
```

6.7.4 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 L6.17 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OSMboxCreate()函数建立的 [L6.17(1)]。接着，它得到邮箱中的当前内容[L6.17(2)]，并判断是否有消息是可用的[L6.17(3)]。如果邮箱中有消息，就把邮箱清空[L6.17(4)]，而邮箱中原来指向消息的指针被返回给 OSMboxAccept()的调用函数[L6.17(5)]。OSMboxAccept()函数的调用函数必须检查该返回值是否为 NULL。如果该值是 NULL，说明邮箱是空的，没有可用的消息。如果该值是非 NULL 值，说明邮箱中有消息可用，而且该调用函数已经得到了该消息。中断服务子程序在试图得到一个消息时，应该使用 OSMboxAccept()函数，而不能使用 OSMboxPend()函数。

OSMboxAccept()函数的另一个用途是，用户可以用它来清空一个邮箱中现有的内容。

程序清单 L6.17 无等待地从邮箱中得到消息

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

```
    void  *msg;


    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                    (1)
        OS_EXIT_CRITICAL();
        return ((void *)0);
    }
    msg = pevent->OSEventPtr;                                           (2)
    if (msg != (void *)0) {                                             (3)
        pevent->OSEventPtr = (void *)0;                                 (4)
    }
    OS_EXIT_CRITICAL();
    return (msg);                                                       (5)
}
```

6.7.5 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 L6.18 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，是因为这里关心的只是和特定邮箱有关的内容，而非整个 OS_EVENT 数据结构的内容。后者还包含了另外两个域（.OSEventCnt 和.OSEventType），而 OS_MBOX_DATA 只包含邮箱中的消息指针（.OSMsg）和该邮箱现有的等待任务列表（.OSEventTbl[]和.OSEventGrp）。

和前面的所以函数一样，该函数也是先检查事件控制是否是邮箱[L6.18(1)]。然后，将邮箱中的等待任务列表[L6.18(2)]和邮箱中的消息[L6.18(3)]从 OS_EVENT 数据结构复制到 OS_MBOX_DATA 数据结构。

程序清单 L6.18 查询邮箱的状态

```
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) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;                             (2)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdest++ = *psrc++;
    }
    pdata->OSMsg      = pevent->OSEventPtr;                             (3)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

6.7.6 用邮箱作二值信号量

一个邮箱可以被用作二值的信号量。首先，在初始化时，将邮箱设置为一个非零的指针（如 void *1）。这样，一个任务可以调用 OSMboxPend()函数来请求一个信号量，然后通过调用 OSMboxPost()函数来释放一个信号量。程序清单 L6.19 说明了这个过程是如何工作的。如果用户只需要二值信号量和邮箱，这样做可以节省代码空间。这时可以将 OS_SEM_EN 设置为 0，只使用邮箱就可以了。

程序清单 L6.19 使用邮箱作为二值信号量

```
OS_EVENT *MboxSem;



void Task1 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSMboxPend(MboxSem, 0, &err);   /* 获得对资源的访问权  */
        .
        .    /* 任务获得信号量,对资源进行访问  */
        .
        OSMboxPost(MboxSem, (void*)1);  /* 释放对资源的访问权 */
```

```
    }
}
```

6.7.7 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 L6.20 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“哑”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。

程序清单 L6.20 使用邮箱实现延时

```
OS_EVENT *MboxTimeDly;


void Task1 (void *pdata)
{
    INT8U err;


    for (;;) {
        OSMboxPend(MboxTimeDly, TIMEOUT, &err);   /* 延时该任务 */
        .
        .    /* 延时结束后执行的代码  */
        .
    }
}


void Task2 (void *pdata)
{
    INT8U err;


    for (;;) {
        OSMboxPost(MboxTimeDly, (void *)1);       /* 取消任务1的延时 */
        .
        .
```

```
    }
}
```

6.8 消息队列

消息队列是μC/OS-II 中另一种通讯机制，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 6.07.01 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 OSQQuery()函数。图 F6.7 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 F6.7 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()

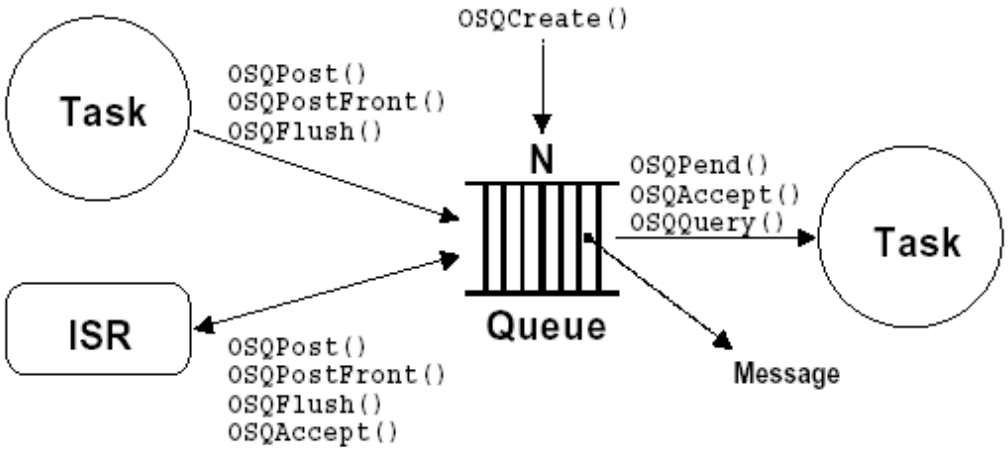


Figure 6-7, Relationship between tasks, ISRs and a message queue.

或者 OSQAccept()函数。但是，只有任务可以调用 OSQPend()和 OSQQuery()函数。

图 F6.7 任务、中断服务子程序和消息队列之间的关系——Figure 6.7

图 F6.8 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[F6.8(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[F6.8(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[F6.8(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 F6.9 所示。

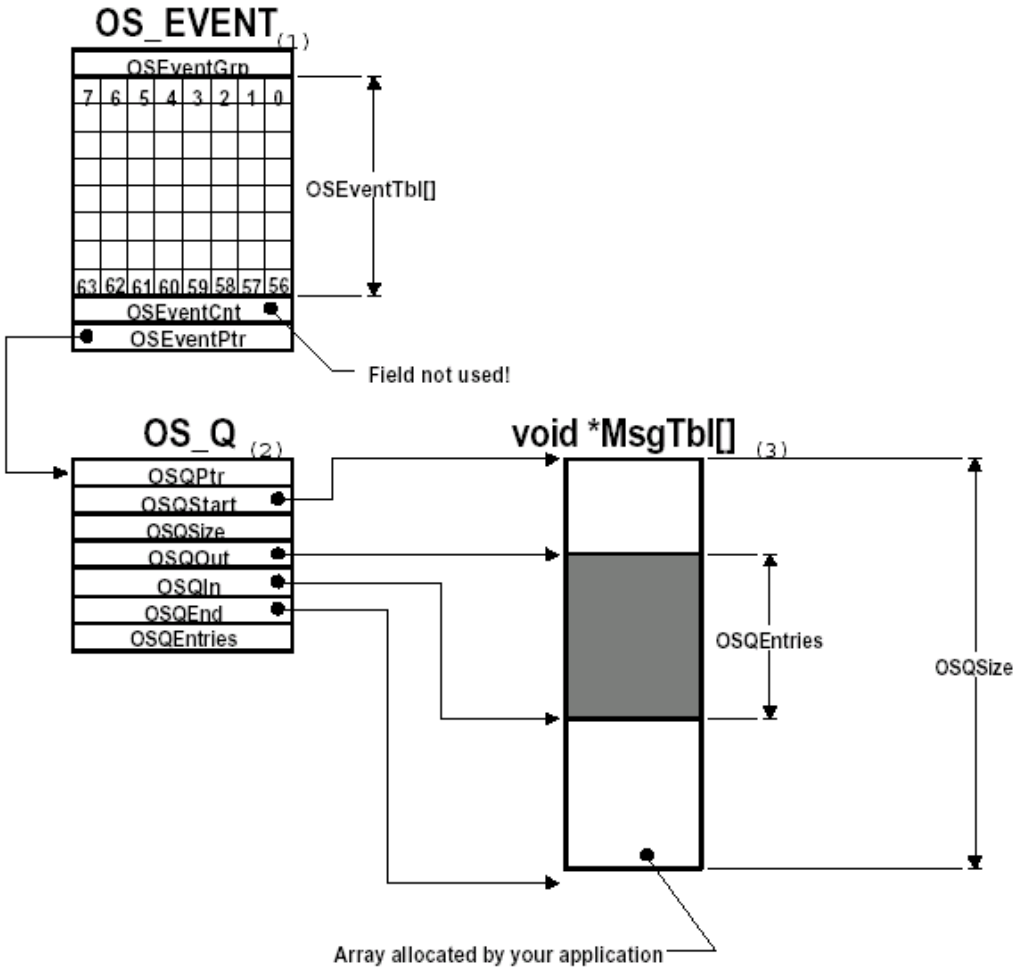


Figure 6-8, Data structures used in a message queue.

图 F6.8 用于消息队列的数据结构——Figure 6.8

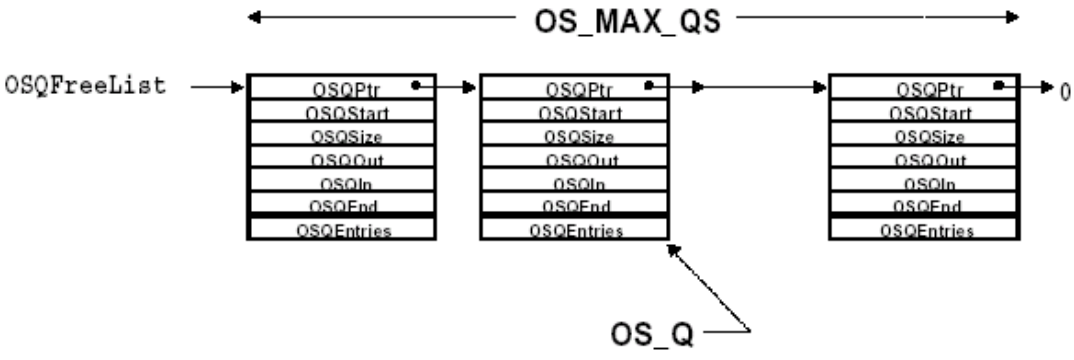


Figure 6-9, List of free queue control blocks.

图 F6.9 空闲队列控制块链表——Figure 6.9

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 65,535。

.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 F6.10。其中的每个单元包含一个指针。队列未满时，.OSQIn [F6.10(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 [F6.10(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）[F6.10(2)]。当.OSQEntries 和.OSQSize 相等时，说明队列已满。消息指针总是从.OSQOut [F6.10(4)]指向的单元取出。指针.OSQStart 和.OSQEnd [F6.10(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。

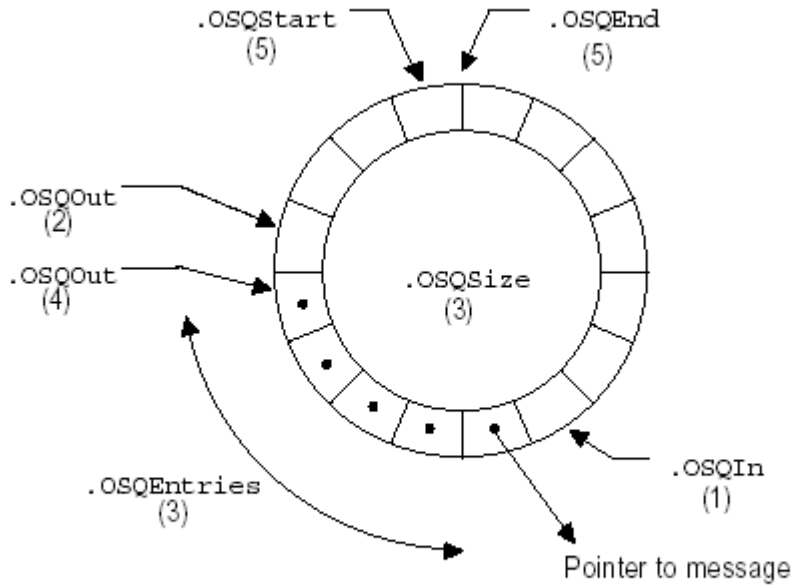


Figure 6-10, Message queue is a circular buffer of pointers.

图 F6.10 消息队列是一个由指针组成的循环缓冲区——Figure 6.10

6.8.1 建立一个消息队列，OSQCreate()

程序清单 L6.21 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。该指针数组必须声名为 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 F6.3）[L6.21(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[L6.21(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[L6.21(3)]。如果有空闲队列控制块是可以的，就对其进行初始化[L6.21(4)]。然后该函数将事件控制块的类型设置为 OS_EVENT_TYPE_Q [L6.21(5)]，并使其.OSEventPtr 指针指向队列控制块[L6.21(6)]。OSQCreate()还要调用 OSEventWaitListInit()函数对事件控制块的等待任务列表初始化[见 6.01 节，初始化一个事件控制块，OSEventWaitListInit()] [L6.21(7)]。因为此时消息队列正在初始化，显然它的等待任务列表是空的。最后，OSQCreate()向它的调用函数返回一个指向事件控制块的指针[L6.21(9)]。该指针将在调用 OSQPend()，OSQPost()，OSQPostFront()，OSQFlush()，OSQAccept()和 OSQQuery()等消息队列处理函数时使用。因此，该指针可以被看作是对应消息队列的句柄。值得注意的是，如果此时没有空闲的事件控制块，OSQCreate()函数将返回一个 NULL 指针。如果没有队列控制块可以使用，为了不浪费事件控制

块资源，OSQCreate()函数将把刚刚取得的事件控制块重新返还给空闲事件控制块列表[L6.21(8)]。

另外，消息队列一旦建立就不能再删除了。试想，如果有任务正在等待某个消息队列中的消息，而此时又删除该消息队列，将是很危险的。

程序清单 L6.21 建立一个消息队列

```
OS_EVENT *OSQCreate (void **start, INT16U size)
{
    OS_EVENT *pevent;
    OS_Q     *pq;



    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                          (1)
    if (OSEventFreeList != (OS_EVENT *)0) {
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;     (2)
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {
        OS_ENTER_CRITICAL();
        pq = OSQFreeList;                                              (3)
        if (OSQFreeList != (OS_Q *)0) {
            OSQFreeList = OSQFreeList->OSQPtr;
        }
        OS_EXIT_CRITICAL();
        if (pq != (OS_Q *)0) {
            pq->OSQStart        = start;                               (4)
            pq->OSQEnd          = &start[size];
            pq->OSQIn           = start;
            pq->OSQOut          = start;
            pq->OSQSize         = size;
            pq->OSQEntries      = 0;
            pevent->OSEventType = OS_EVENT_TYPE_Q;                     (5)
            pevent->OSEventPtr  = pq;                                  (6)
            OSEventWaitListInit(pevent);                               (7)
        } else {
            OS_ENTER_CRITICAL();
            pevent->OSEventPtr = (void *)OSEventFreeList;              (8)
```

```
            OSEventFreeList    = pevent;
            OS_EXIT_CRITICAL();
            pevent = (OS_EVENT *)0;
        }
    }
    return (pevent);                                                    (9)
}
```

6.8.2 等待一个消息队列中的消息，OSQPend()

程序清单 L6.22 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[L6.22(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)[L6.22(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[L6.22(3)],然后将队列中的有效消息数减1[L6.22(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 [L6.22(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 [L6.22(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 L6.22 在一个消息队列中等待一条消息

```
void *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *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++;                                            (3)
        pq->OSQEntries--;                                               (4)
        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;
        } else if (OSTCBCur->OSTCBStat & OS_STAT_Q) {                   (12)
            OSEventTO(pevent);                                          (13)
            OS_EXIT_CRITICAL();
            msg                     = (void *)0;                        (14)
            *err                    = OS_TIMEOUT;
        } else {
            msg = *pq->OSQOut++;                                        (15)
            pq->OSQEntries--;
            if (pq->OSQOut == pq->OSQEnd) {
                pq->OSQOut = pq->OSQStart;
            }
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                    (16)
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
        }
    }
    return (msg);                                                       (17)
}
```

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[L6.22(7)]。象 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[L6.22(8)]。当有其它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[L6.22(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[L6.22(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[L6.22(17)]，并将对应的消息被返回到调用函数[L6.22(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[L6.22(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[L6.22(13)]。这时，因为消息队列中没有消息，所以返回的指针是 NULL[L6.22(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[L6.22(15)]。然后，将任务的任务控制中指向事件控制块的指针删除[L6.22(16)]。

6.8.3 向消息队列发送一个消息（FIFO），OSQPost()

程序清单 L6.23 是 OSQPost()函数的源代码。在确认事件控制块是消息队列后[L6.23(1)]，OSQPost()函数检查是否有任务在等待该消息队列中的消息[L6.23(2)]。当事件控制块的.OSEventGrp 域为非 0 值时，说明该消息队列的等待任务列表中有任务。这时，调用 OSEventTaskRdy()函数 [见 6.02 节，使一个任务进入就绪状态，OSEventTaskRdy()]从列表中取出最高优先级的任务[L6.23(3)]，并将它置于就绪状态。然后调用函数 OSSched() [L6.23(4)]进行任务的调度。如果上面取出的任务的优先级在整个系统就绪的任务里也是最高的，而且 OSQPost()函数不是中断服务子程序调用的，就执行任务切换，该最高优先级任务被执行。否则的话，OSSched()函数直接返回，调用 OSQPost()函数的任务继续执行。

程序清单 L6.23 向消息队列发送一条消息

```
INT8U OSQPost (OS_EVENT *pevent, void *msg)
{
    OS_Q   *pq;



    OS_ENTER_CRITICAL();
```

```
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                                        (2)
        OSEventTaskRdy(pevent, msg, OS_STAT_Q);                      (3)
        OS_EXIT_CRITICAL();
        OSSched();
(4)
        return (OS_NO_ERR);
    } else {
        pq = pevent->OSEventPtr;
        if (pq->OSQEntries >= pq->OSQSize) {                         (5)
            OS_EXIT_CRITICAL();
            return (OS_Q_FULL);
        } else {
            *pq->OSQIn++ = msg;                                      (6)
            pq->OSQEntries++;
            if (pq->OSQIn == pq->OSQEnd) {
                pq->OSQIn = pq->OSQStart;
            }
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
    }
}
```

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[L6.23(5)]，指向该消息的指针被插入到消息队列中[L6.23(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能插入到消息队列中而丢失。

此外，如果 OSQPost()函数是由中断服务子程序调用的，那么即使产生了更高优先级的任务，也不会在调用 OSSched()函数时发生任务切换。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 3.09 节，μC/OS-II 中的中断）。

6.8.4 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 L6.24 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[L6.24(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[L6.24(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[L6.24(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 L6.24 向消息队列发送一条消息（LIFO）

```
INT8U OSQPostFront (OS_EVENT *pevent, void *msg)
{
    OS_Q   *pq;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {
        OSEventTaskRdy(pevent, msg, OS_STAT_Q);
        OS_EXIT_CRITICAL();
        OSSched();
        return (OS_NO_ERR);
    } else {
        pq = pevent->OSEventPtr;
        if (pq->OSQEntries >= pq->OSQSize) {
            OS_EXIT_CRITICAL();
            return (OS_Q_FULL);
        } else {
            if (pq->OSQOut == pq->OSQStart) {                    (1)
                pq->OSQOut = pq->OSQEnd;                         (2)
            }
            pq->OSQOut--;                                        (3)
            *pq->OSQOut = msg;
            pq->OSQEntries++;
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
```

```
    }
}
```

6.8.5 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 L6.25 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[L6.25(1)]，然后它检查当前消息队列中是否有消息[L6.25(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[L6.25(3)]。OSQAccept()函数的调用函数需要对 OSQAccept()返回的指针进行检查。如果该指针是 NULL 值，说明消息队列是空的，其中没有消息可以 [L6.25(4)]。否则的话，说明已经从消息队列中成功地取得了一条消息。当中断服务子程序要从消息队列中取消息时，必须使用 OSQAccept()函数，而不能使用 OSQPend()函数。

程序清单 L6.25 无等待地从消息队列中取一条消息

```
void *OSQAccept (OS_EVENT *pevent)
{
    void  *msg;
    OS_Q  *pq;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (1)
        OS_EXIT_CRITICAL();
        return ((void *)0);
    }
    pq = pevent->OSEventPtr;
    if (pq->OSQEntries != 0) {                                       (2)
        msg = *pq->OSQOut++;                                         (3)
        pq->OSQEntries--;
        if (pq->OSQOut == pq->OSQEnd) {
            pq->OSQOut = pq->OSQStart;
        }
    } else {
        msg = (void *)0;                                             (4)
    }
    OS_EXIT_CRITICAL();
```

```
    return (msg);
}
```

6.8.6 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 L6.26 是该函数的源代码。和前面的其它函数一样，该函数首先检查 pevent 指针是否是执行一个消息队列[L6.26(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 [L6.26(2)]。这里，没有检查该消息队列的等待任务列表是否为空，因为只要该等待任务列表不空，.OSQEntries 就一定是 0。唯一不同的是，指针.OSQIn 和.OSQOut 此时可以指向消息队列中的任何单元，不一定是起始单元。

程序清单 L6.26 清空消息队列

```
INT8U OSQFlush (OS_EVENT *pevent)
{
    OS_Q  *pq;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pq             = pevent->OSEventPtr;
    pq->OSQIn      = pq->OSQStart;                                   (2)
    pq->OSQOut     = pq->OSQStart;
    pq->OSQEntries = 0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

6.8.7 查询一个消息队列的状态，OSQQuery()

OSQQuery()函数使用户可以查询一个消息队列的当前状态。程序清单 L6.27 是该函数的源代码。OSQQuery()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 OSQQuery()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， OSQQuery()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。

OSQQuery()函数首先检查 pevent 指针指向的事件控制块是一个消息队列[L6.27(1)]，然后复制等待任务列表[L6.27(2)]。如果消息队列中有消息[L6.27(3)]，.OSQOut 指向的队列单元中的内容被复制到 OS_Q_DATA 结构中[L6.27(4)]，否则的话，就复制一个 NULL 指针[L6.27(5)]。最后，复制消息队列中的消息数和消息队列的容量大小[L6.27(6)]。

程序清单 L6.27 程序消息队列的状态

```
INT8U OSQQuery (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) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;                          (2)
    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) {                                        (3)
        pdata->OSMsg = pq->OSQOut;                                   (4)
```

```
    } else {
        pdata->OSMsg = (void *)0;                                    (5)
    }
    pdata->OSNMsgs = pq->OSQEntries;                                 (6)
    pdata->OSQSize = pq->OSQSize;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

6.8.8 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 5.00 节，延时一个任务，OSTimeDly()]，并且给出希望的抽样周期。然后，如图 F6.11 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。当没有其它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，我们可以让其它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。

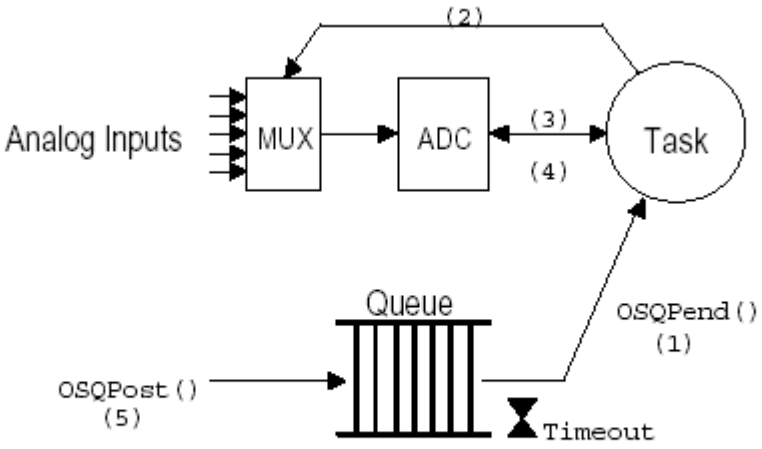


Figure 6-11, Reading analog inputs.

图 F6.11 读模拟量输入——Figure 6.11

6.8.9 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 L6.28。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 L6.28 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 */
    }
}
```

内存管理

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 4.02 节的任务堆栈中，我们讲到过用 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，如同图 F7.1。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 F7.2，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。

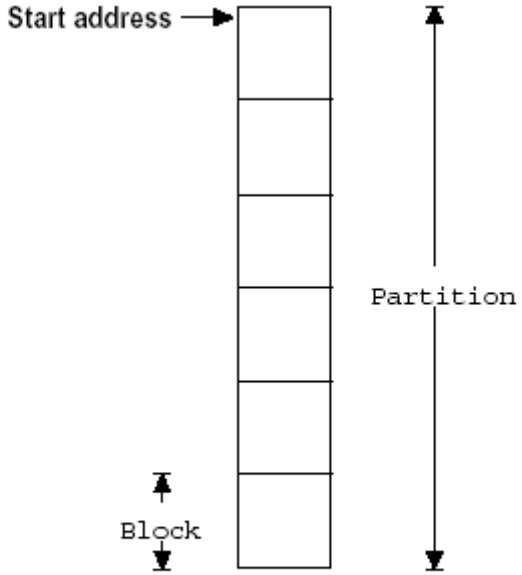


Figure 7-1, Memory partition

图 F7.1 内存分区——Figure 7.1

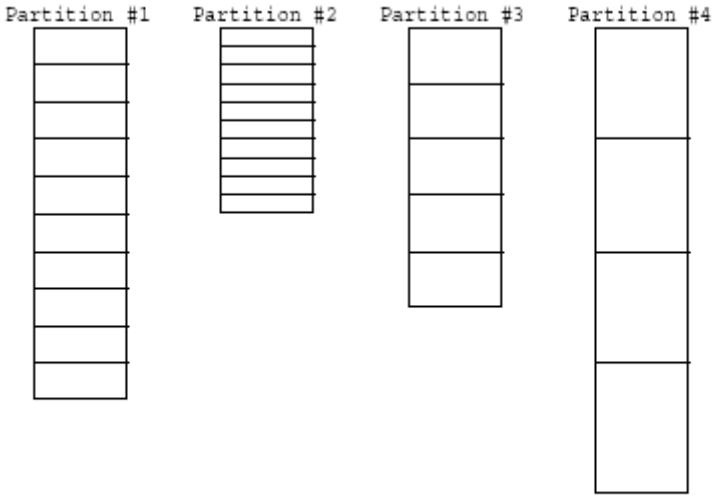


Figure 7-2, Multiple memory partitions.

图 F7.2 多个内存分区——Figure 7.2

内存控制块

为了便于内存的管理，在μC/OS-II 中使用内存控制块（memory control blocks）的数据结构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 L7.1 是内存控制块的定义。

程序清单 L7.1 内存控制块的数据结构

```
typedef struct {
    void   *OSMemAddr;
    void   *OSMemFreeList;
    INT32U  OSMemBlkSize;
    INT32U  OSMemNBlks;
    INT32U  OSMemNFree;
} OS_MEM;
```

.OSMemAddr 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，建立一个内存分区，OSMemCreate()]时被初始化，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]。
.OSMemNBlks 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 F7.3 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。

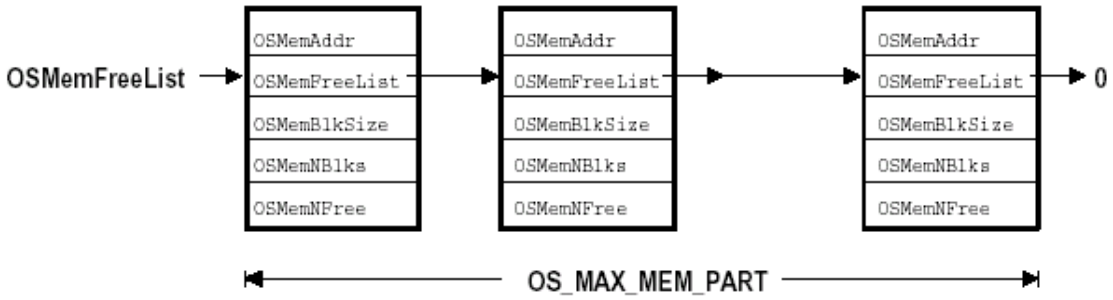


Figure 7-3, List of free memory control blocks.

图 F7.3 空闲内存控制块链表——Figure 7.3

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 L7.2 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 L7.2 建立一个内存分区

```
OS_MEM *CommTxBuf;
INT8U    CommTxPart[100][32];


void main (void)
{
    INT8U err;


    OSInit();
    .
    .
    CommTxBuf = OSMemCreate(CommTxPart, 100, 32, &err);
    .
    .
    OSStart();
}
```

程序清单 L7.3 是 OSMemCreate()函数的源代码。该函数共有 4 个参数：内存分区的起始地址、分区内的内存块总块数、每个内存块的字节数和一个指向错误信息代码的指针。如果 OSMemCreate()操作失败，它将返回一个 NULL 指针。否则，它将返回一个指向内存控制块的指针。对内存管理的其它操作，象 OSMemGet()，OSMemPut()，OSMemQuery()函数等，都要通过该指针进行。

每个内存分区必须含有至少两个内存块[L7.3(1)]，每个内存块至少为一个指针的大小，因为同一分区中的所有空闲内存块是由指针串联起来的[L7.3(2)]。接着，OSMemCreate()从系统中的空闲内存控制块中取得一个内存控制块[L7.3(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[L7.3(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[L7.3(5)]。然后，在对应的内存控制块中填写相应的信息[L7.3(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[L7.3(6)]。

程序清单 L7.3 OSMemCreate()

```
OS_MEM *OSMemCreate (void *addr, INT32U nblks, INT32U blksize, INT8U *err)
{
    OS_MEM  *pmem;
    INT8U   *pblk;
    void   **plink;
    INT32U   i;



    if (nblks < 2) {                                                    (1)
        *err = OS_MEM_INVALID_BLKS;
        return ((OS_MEM *)0);
    }
    if (blksize < sizeof(void *)) {                                     (2)
        *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 < (nblks - 1); i++) {
        *plink = (void *)pblk;
        plink  = (void **)pblk;
```

```
        pblk   = pblk + blksize;
    }
    *plink = (void *)0;
    OS_ENTER_CRITICAL();
    pmem->OSMemAddr     = addr;                                    (6)
    pmem->OSMemFreeList = addr;
    pmem->OSMemNFree    = nblks;
    pmem->OSMemNBlks    = nblks;
    pmem->OSMemBlkSize  = blksize;
    OS_EXIT_CRITICAL();
    *err   = OS_NO_ERR;
    return (pmem);                                                 (7)
}
```

图 F7.4 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。

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。该函数的唯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 7.03 节，释放一个内存块，OSMemPut()]。

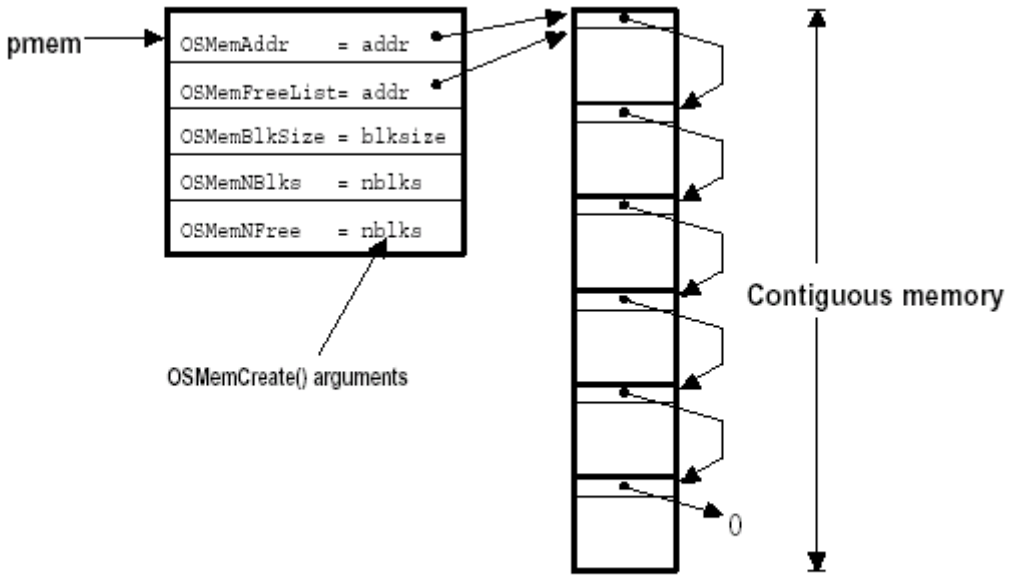


Figure 7-4, OSMemCreate()

图 F7.4 OSMemCreate()——Figure 7.4

程序清单 L7.4 是 OSMemGet()函数的源代码。参数中的指针 pmem 指向用户希望从其中分配内存块的内存分区[L7.4(1)]。OSMemGet()首先检查内存分区中是否有空闲的内存块[L7.4(2)]。如果有，从空闲内存块链表中删除第一个内存块[L7.4(3)]，并对空闲内存块链表作相应的修改 [L7.4(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[L7.4(5)]。最后，返回指向被分配内存块的指针[L7.4(6)]。

程序清单 L7.4 OSMemGet()

```
void *OSMemGet (OS_MEM *pmem, INT8U *err)                              (1)
{
    void    *pblk;



    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree > 0) {                                        (2)
        pblk                = pmem->OSMemFreeList;                     (3)
        pmem->OSMemFreeList = *(void **)pblk;                          (4)
        pmem->OSMemNFree--;                                            (5)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (pblk);                                                 (6)
    } else {
        OS_EXIT_CRITICAL();
        *err = OS_MEM_NO_FREE_BLKS;
        return ((void *)0);
    }
}
```

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

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，其它 88 字节属于其它的任务，这就有可能使系统崩溃。

程序清单 L7.5 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[L7.5(1)]。OSMemPut()首先检查内存分区是否已满[L7.5(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[L7.5(3)]。最后，将分区中空闲内存块总数加1[L7.5(4)]。

程序清单 L7.5 OSMemPut()

```
INT8U OSMemPut (OS_MEM  *pmem, void *pblk)                              (1)
{
    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree >= pmem->OSMemNBlks) {                         (2)
        OS_EXIT_CRITICAL();
        return (OS_MEM_FULL);
    }
    *(void **)pblk      = pmem->OSMemFreeList;                          (3)
    pmem->OSMemFreeList = pblk;
    pmem->OSMemNFree++;                                                 (4)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

查询一个内存分区的状态，OSMemQuery()

在μC/OS-II 中，可以使用 OSMemQuery()函数来查询一个特定内存分区的有关消息。通过该函数可以知道特定内存分区中内存块的大小、可用内存块数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 L7.6。

程序清单 L7.6 OS_MEM_DATA数据结构

```
typedef struct {
    void  *OSAddr;     /* 指向内存分区首地址的指针 */
    void  *OSFreeList; /* 指向空闲内存块链表首地址的指针 */
    INT32U OSBlkSize;  /* 每个内存块所含的字节数 */
    INT32U OSNBlks;    /* 内存分区总的内存块数 */
    INT32U OSNFree;    /* 空闲内存块总数 */
    INT32U OSNUsed;    /* 正在使用的内存块总数 */
} OS_MEM_DATA;
```

程序清单 L7.7 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[L7.7(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（critical section 中断屏蔽）的外面。

程序清单 L7.7 OSMemQuery()

```
INT8U OSMemQuery (OS_MEM *pmem, OS_MEM_DATA *pdata)
{
```

```
    OS_ENTER_CRITICAL();
    pdata->OSAddr     = pmem->OSMemAddr;                                    (1)
    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;                    (2)
    return (OS_NO_ERR);
}
```

Using Memory Partitions

图 F7.5 是一个演示如何使用μC/OS-II 中的动态分配内存功能，以及利用它进行消息传递[见第 6 章]的例子。程序清单 L7.8 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 F7.5 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象的任何可能的信息。

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。

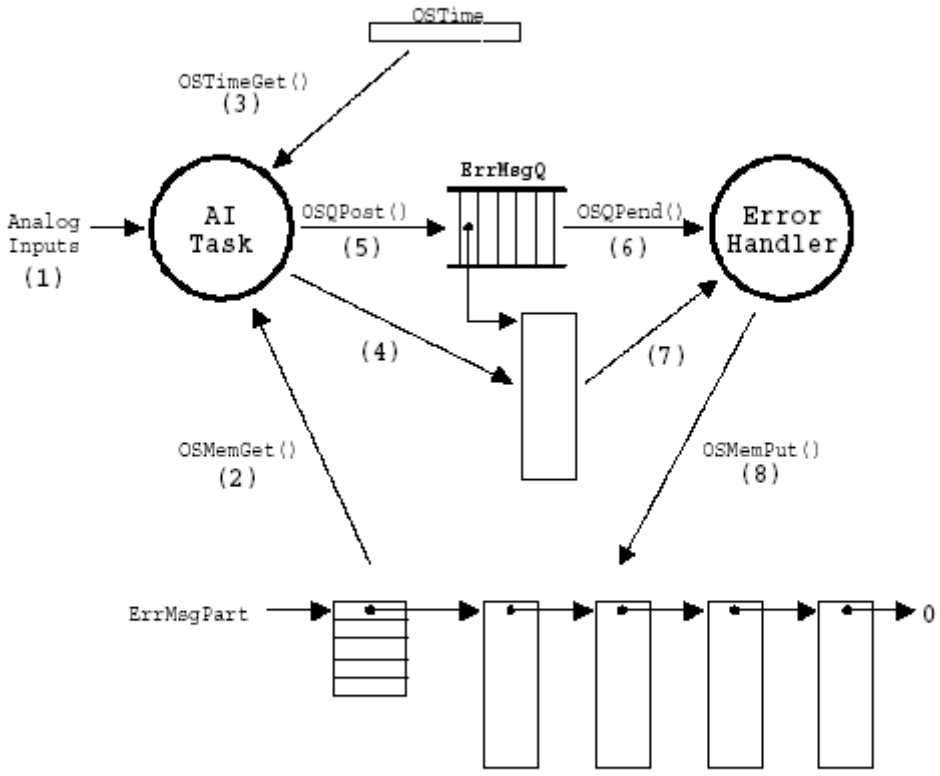


Figure 7-5, Using dynamic memory allocation.

图 F7.5 使用动态内存分配——Figure 7.5

程序清单 L7.8 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.
                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)
            (得到指向包含有关错误数据的内存块的指针)
        读入消息,并根据消息的内容执行相应的操作;                    (7)
        将内存块放回到相应的内存分区中;                              (8)
    }
}
```

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 6.05 节，信号量）。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 L7.9。

程序代码首先定义了程序中使用到的各个变量[L7.9(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，μC/OS-II 调用 OSInit()进行系统初始化[L7.9(2)]，然后用内存分区中总的内存块数来初始化一个信号量[L7.9(3)]，紧接着建立内存分区[L7.9(4)]和相应的要访问该分区的任务[L7.9(5)]。当然，到此为止，我们对如何增加其它的任务也已经很清楚了。显然，如果系统中只有一个任务使用动态内存块，就没有必要使用信号量了。这种情况不需要保证内存资源的互斥。事实上，除非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[L7.9(6)]。当一个任务运行时，只有在信号量有效时[L7.9(7)]，才有可能得到内存块[L7.9(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它任务释放并放回到内存分区后，μC/OS-II 才会返回到该任务去执行。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[L7.9(9)]，并发送该信号量[L7.9(10)]。

程序清单 L7.9 等待从一个内存分区中分配内存块

```
OS_EVENT  *SemaphorePtr;                                             (1)
OS_MEM    *PartitionPtr;
```

```
INT8U      Partition[100][32];
OS_STK     TaskStk[1000];


void main (void)
{
    INT8U err;

    OSInit();                                                          (2)
    .
    .
    SemaphorePtr = OSSemCreate(100);                                   (3)
    PartitionPtr = OSMemCreate(Partition, 100, 32, &err);              (4)
    .
    OSTaskCreate(Task, (void *)0, &TaskStk[999], &err);                (5)
    .
    OSStart();                                                         (6)
}
void Task (void *pdata)
{
    INT8U  err;
    INT8U *pblock;


    for (;;) {
        OSSemPend(SemaphorePtr, 0, &err);                              (7)
        pblock = OSMemGet(PartitionPtr, &err);                         (8)
        .
        .  /* 使用内存块 */
        .
        OSMemPut(PartitionPtr, pblock);                                (9)
        OSSemPost(SemaphorePtr);                                      (10)
    }
}
```

第八章　　移植μC/OS-Ⅱ

这一章介绍如何将μC/OS-Ⅱ移植到不同的处理器上。所谓移植，就是使一个实时内核能在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。如果已经有人在您使用的处理器上成功地移植了μC/OS-Ⅱ，您也得到了相关代码，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它 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。这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)。如果用户打算在其它处理器上使用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．μCOS-Ⅱ.com 中查找一些移植实例。

图 8.1　　μC/OS-II 硬件和软件体系结构

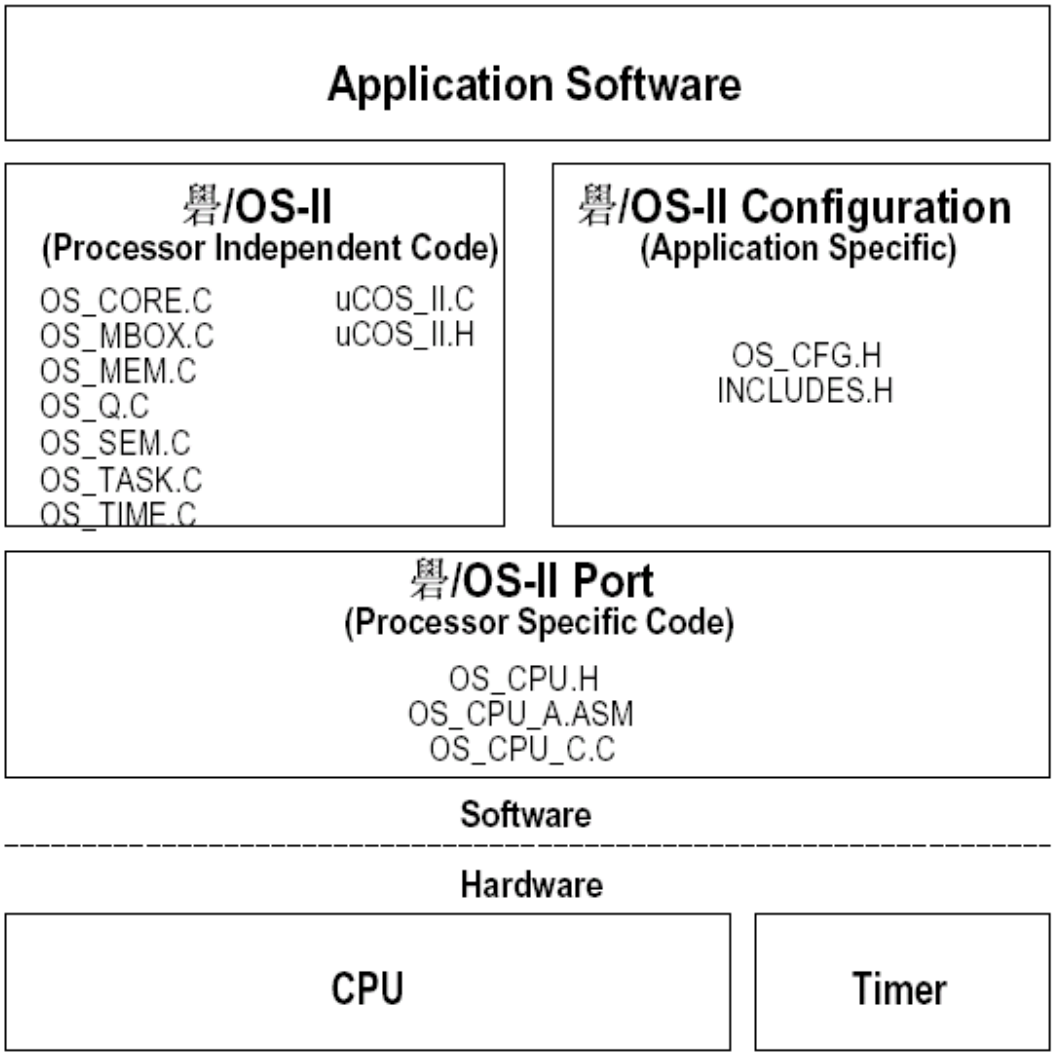


Figure 8-1, μ C/OS-II Hardware/Software Architecture

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。前提是您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 至 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。测试一个象μC/OS-II 一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。

8.00 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是一个可剥夺型内核，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。

8.01 目录和文件

本书所付的磁盘中提供了μC/OS-II的安装程序，可在硬盘上安装μC/OS-II和移植实例代码（Intel 80x86实模式，大模式编译)。我设计了一个连续的目录结构，使得用户更容易找到目标处理器的文件。如果想增加一个其它处理器的移植实例，您可以考虑采取同样的方法（包括目录的建立和文件的命名等等)。

所有的移植实例都应放在用户硬盘的\SOFTWARE\μCOS-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。

Intel/AMD 80186	\SOFTWARE\uCOS-II\Ix86S
	\OS_CPU.H
	\OS_CPU_A.ASM
	\OS_CPU_C.C
	\SOFTWARE\uCOS-II\Ix86L
	\OS_CPU.H
	\OS_CPU_A.ASM
	\OS_CPU_C.C
Motorola 68HC11	\SOFTWARE\uCOS-II\68HC11
	\OS_CPU.H
	\OS_CPU_A.ASM
	\OS_CPU_C.C

8.02 INCLUDES.H

在第一章中曾提到过，INCLUDES.H是一个头文件，它在所有.C文件的第一行被包含。

```
#include "includes.h"
```

INCLUDES.H 使得用户项目中的每个.C 文件不用分别去考虑它实际上需要哪些头文件。使用 INCLUDES.H 的唯一缺点是它可能会包含一些实际不相关的头文件。这意味着每个文件的编译时间可能会增加。但由于它增强了代码的可移植性，所以我们还是决定使用这一方法。用户可以通过编辑 INCLUDES.H 来增加自己的头文件，但是用户的头文件必须添加在头文件列表的最后。

8.03 OS_CPU.H

OS_CPU.H 包括了用#defines 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 L8.1 所示。

程序清单 L 8.1 OS_CPU.H.

```
#ifdef  OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT  extern
#endif

/*
*********************************************************************
*                              数据类型
*                         (与编译器相关)
*********************************************************************
*/

typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;     /* 无符号8位整数    */                (1)
typedef signed   char  INT8S;      /* 有符号8位整数    */
typedef unsigned int   INT16U;    /* 无符号16位整数   */
typedef signed   int   INT16S;     /* 有符号16位整数   */
typedef unsigned long  INT32U;    /* 无符号32位整数   */
typedef signed   long  INT32S;     /* 有符号32位整数   */
typedef float         FP32;       /* 单精度浮点数      */               (2)
typedef double        FP64;       /* 双精度浮点数     */

typedef unsigned int   OS_STK;     /* 堆栈入口宽度为16位 */

/*
*********************************************************************
*                          与处理器相关的代码
*********************************************************************
*/

#define  OS_ENTER_CRITICAL()  ???  /* 禁止中断           */           (3)
```

```
#define  OS_EXIT_CRITICAL()   ???  /* 允许中断            */

#define  OS_STK_GROWTH       1    /* 定义堆栈的增长方向： 1=向下, 0=向上 */ (4)

#define  OS_TASK_SW()        ???                                  (5)
```

8.03.01　与编译器相关的数据类型

因为不同的微处理器有不同的字长，所以μC/OS-Ⅱ的移植包括了一系列的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[L8.1(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[L8.1(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 声明位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。

8.03.02　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段，并且在访问完毕后重新允许中断。这就使得μC/OS-Ⅱ能够保护临界段代码免受多任务或中断服务例程(ISRs)的破坏。中断禁止时间是商业实时内核公司提供的重要指标之一，因为它将影响到用户的系统对实时事件的响应能力。虽然μC/OS-Ⅱ尽量使中断禁止时间达到最短，但是μC/OS-Ⅱ的中断禁止时间还主要依赖于处理器结构和编译器产生的代码的质量。通常每个处理器都会提供一定的指令来禁止/允许中断，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。其它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。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()[L8.1(3)]。

```
{
    OS_ENTER_CRITICAL();
    /* 才μC/OS-II 临界代码段 */
    OS_EXIT_CRITICAL();
}
```

方法 1

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。

方法 2

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()来保护代码的临界段。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。一个通用的办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！

问题是：哪种方法更好一点？这就得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中中断是否是允许的，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种方法。

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：

```
#define OS_ENTER_CRITICAL()  asm CLI
#define OS_EXIT_CRITICAL()   asm STI
```

CLI 和 SCI 指令都会在两个时钟周期内被马上执行(总共为四个周期)。为了保持中断状态，用户需要用下面的方法来执行宏：

```
#define OS_ENTER_CRITICAL()  asm PUSHF; CLI
#define OS_EXIT_CRITICAL()   asm POPF
```

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。

8.03.03 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 [L8.1(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。

8.03.04 OS_TASK_SW()

OS_TASK_SW()[L8.1(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 8.04.02)。

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。

8.04 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。

8.04.01 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 OSTaskCteateExt())。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，堆栈指针总是储存在任务控制块(即它的 OS_TCB)的开头。换句话说，也就是要想恢复的任务堆栈指针总是储存在 OS_TCB 的 0 偏址内存单元中。

```
void OSStartHighRdy (void)
{
    Call user definable OSTaskSwHook();
    Get the stack pointer of the task to resume:
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    OSRunning = TRUE;
    Restore all processor registers from the new task's stack;
    Execute a return from interrupt instruction;
}
```

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在调用它(OSRunning 为 TRUE).

OSStartHighRdy()还必须在最高优先级任务恢复之前和调用 OSTaskSwHook()之后设置 OSRunning 为 TRUE。

8.04.02 OSCtxSw()

如前面所述,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 L8.2 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 L 8.2 OSCtxSw()的原型

```
void OSCtxSw(void)
{
    保存处理器寄存器;
    将当前任务的堆栈指针保存到当前任务的OS_TCB中:
        OSTCBCur->OSTCBStkPtr = Stack pointer;
    调用用户定义的OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    得到需要恢复的任务的堆栈指针:
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    将所有处理器寄存器从新任务的堆栈中恢复出来;
    执行中断返回指令;
}
```

8.04.03 OSIntCtxSw()

OSIntExit()通过调用 OSIntCtxSw()来从 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，堆栈结构中还有其它的一些东西。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[F8.2(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.

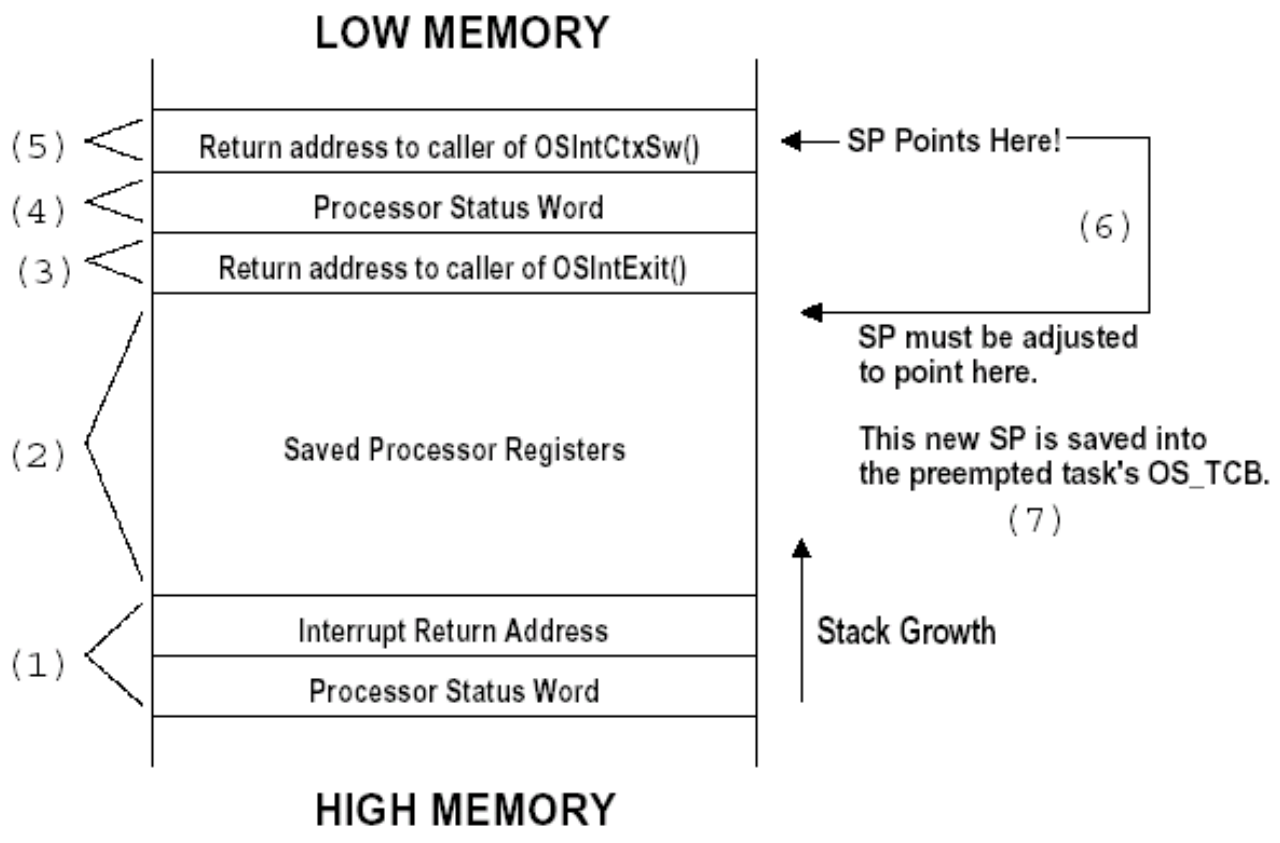


Figure 8-2, Stack contents during an ISR

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[F8.2(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[F8.2(3)]。

OSIntExit()刚开始时会禁止中断，因为它需要执行临界段的代码。根据 OS_ENTER_CRITICAL()的不同执行过程(参看 8.03.02),处理器的状态寄存器会被保存到被中断的任务的堆栈中[F8.2(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[F8.2(5)]。

在用户切换任务的时候，用户只想将某些项（[F8.2(1)]和[F8.2(2)]）保留在堆栈中，并忽略其它项（F8.2(3)，(4)和(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[F8.2(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[F8.2(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 其它方面的。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()中是否被正确地调整了。

OSIntCtxSw()的原型如程序清单 L8.3 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 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 L 8.3 OSIntCtxSw()的原型

```
void OSIntCtxSw(void)
{
    调整堆栈指针来去掉在调用：
        OSIntExit(),
        OSIntCtxSw()过程中压入堆栈的多余内容;
    将当前任务堆栈指针保存到当前任务的OS_TCB中:
        OSTCBCur->OSTCBStkPtr = 堆栈指针;
    调用用户定义的OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    得到需要恢复的任务的堆栈指针:
        堆栈指针 = OSTCBHighRdy->OSTCBStkPtr;
    将所有处理器寄存器从新任务的堆栈中恢复出来;
    执行中断返回指令;
}
```

8.04.04 OSTickISR()

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 L8.4 所示)。

程序清单 L 8.4 在不正确的位置启动时钟节拍中断

```
void main(void)
{
```

```
    .
    .
    OSInit();              /* 初始化 才μC/OS-II               */
    .
    .
    /* 应用程序初始化代码 ...                          */
    /* ... 调用OSTaskCreate()建立至少一个任务       */
    .
    .
    允许时钟节拍中断; /* 千万不要在这里允许!!!           */
    .
    .
    OSStart();             /* 开始多任务调度                */
}
```

有可能在μC/OS-Ⅱ开始执行第一个任务前时钟节拍中断就发生了。在这种情况下，μC/OS-Ⅱ的运行状态不确定，用户的应用程序也可能会崩溃。

时钟节拍 ISR 的原型如程序清单 L8.5 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 语言中访问 CPU 寄存器。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，并且作为临界段代码中受到保护。

程序清单 L 8.5 时钟节拍ISR的原型

```
void OSTickISR(void)
{
   保存处理器寄存器;
   调用OSIntEnter()或者直接将 OSIntNesting加1;

   调用OSTimeTick();

   调用OSIntExit();
   恢复处理器寄存器;
   执行中断返回指令;
}
```

8.05 OS_CPU_C.C

μC/OS-Ⅱ的移植实例要求用户编写六个简单的 C 函数：

OSTaskStkInit()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskSwHook()
OSTaskStatHook()
OSTimeTickHook()

唯一必要的函数是 OSTaskStkInit()，其它五个函数必须得声明但没必要包含代码。

8.05.01 OSTaskStkInt()

OSTaskCreate()和 OSTaskCreateExt()通过调用 OSTaskStkInt()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 OSTaskStkInt()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()还要求有其它的参数，但这些参数在讨论 OSTaskStkInt()的时候是无关紧要的。为了正确初始化堆栈结构，OSTaskStkInt()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()中得到。

图 8.3 堆栈初始化（pdata通过堆栈传递）

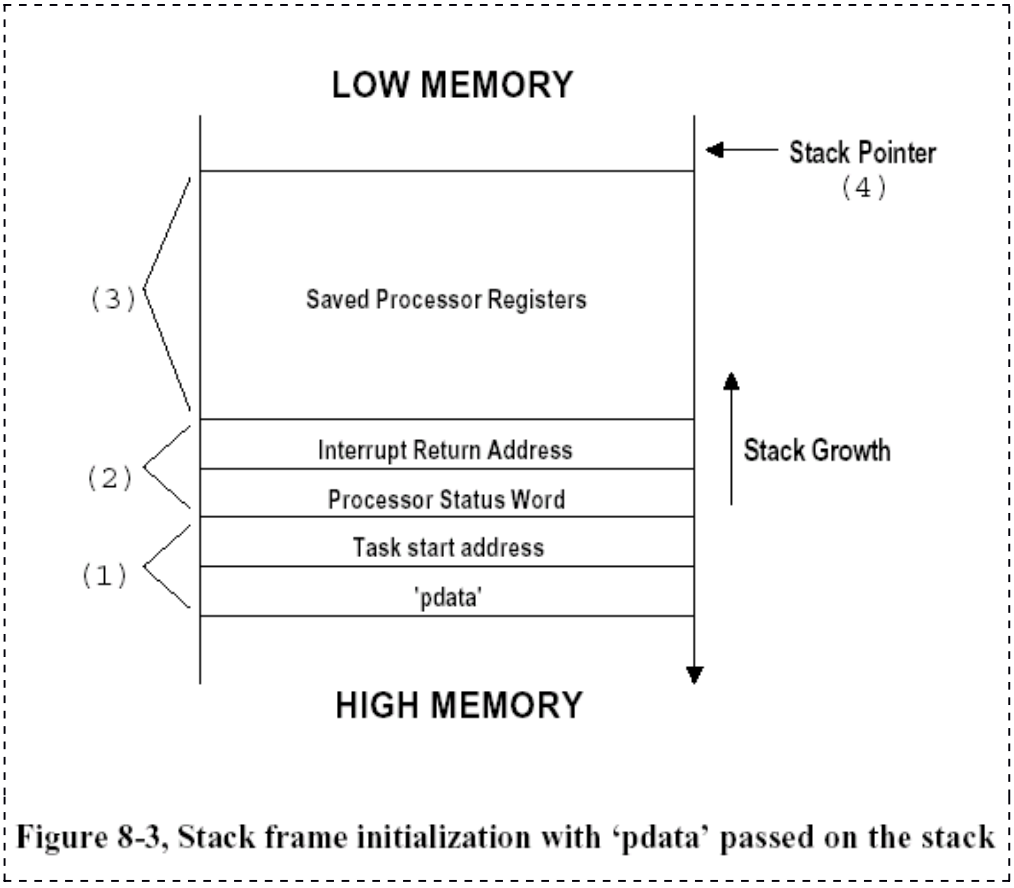


Figure 8-3, Stack frame initialization with ‘pdata’ passed on the stack

回顾一下，在μC/OS-Ⅱ中，无限循环的任务看起来就像其它的 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，好像它被其它的任务调用一样。

```
void MyTask (void *pdata)
{
    /* 对'pdata'做某些操作 */
    for (;;) {
        /* 任务代码                    */
    }
}
```

如果我想从其它的函数中调用 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[F8.3(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它函数中。

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，而其它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[F8.3(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[F8.3(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 储存在哪个寄存器中。pdata 的内容就会随着这个寄存器的储存被放置在堆栈中。

图 8.4　　堆栈初始化（pdata通过寄存器传递）

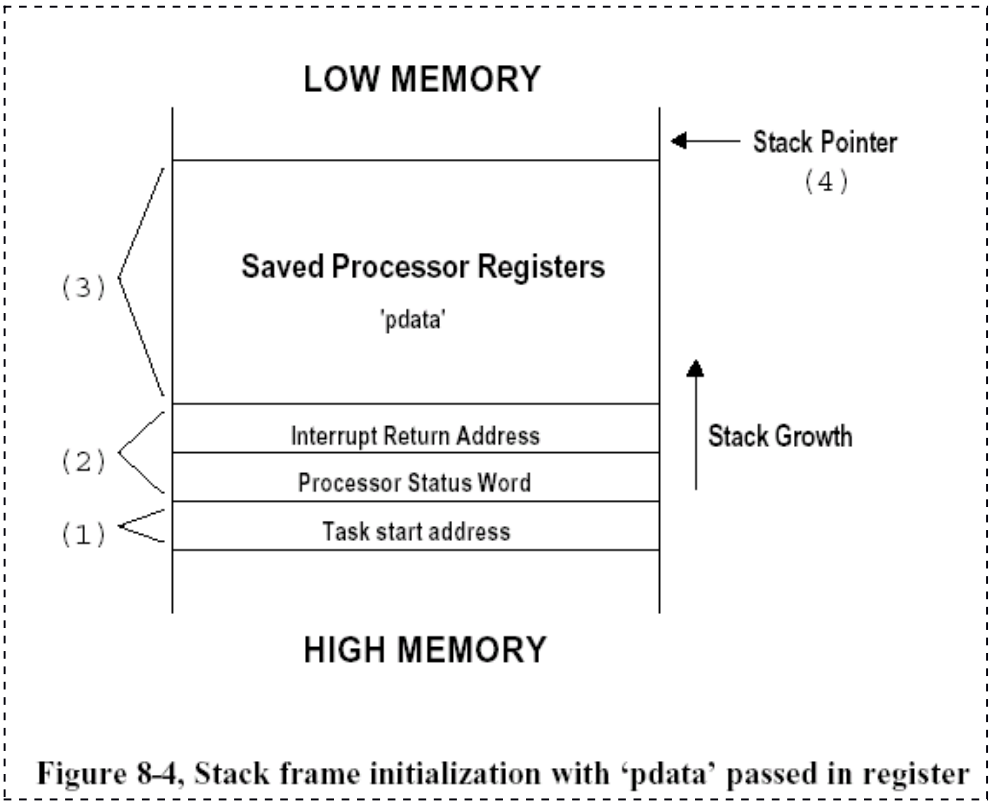


Figure 8-4, Stack frame initialization with 'pdata' passed in register

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[F8.3(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。

8.05.02 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它的文件中重新定义 hook 函数。

8.05.03 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 扩展是否被建立了(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。

8.05.04 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。

8.05.05 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。

8.05.06 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。

OSTaskCreateHook()

void OSTaskCreateHook(OS_TCB *ptcb)

File	*Called from*	*Code enabled by*
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 寄存器以及其它寄存器的内容。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()来触发示波器或逻辑分析仪，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

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 OSTaskCreateHook (OS_TCB *ptcb)
{
   if (ptcb->OSTCBExtPtr != (void *)0) {
      /* 储存浮点寄存器的内容到.. */
      /* ..TCB扩展域中                          */
    }
}
```

OSTaskDelHook()

void OSTaskDelHook(OS_TCB *ptcb)

File	*Called from*	*Code enabled by*
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()来触发示波器或逻辑分析仪，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

```
void OSTaskDelHook (OS_TCB *ptcb)
{
    /* 输出信号触发示波器          */
}
```

OSTaskSwHook()

void OSTaskSwHook(void)

File	*Called from*	*Code enabled by*
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 TCB，而 OSTCBCur 指向被切换出去的任务的 TCB。OSTaskSwHook()在保存好了任务的寄存器和保存好了指向当前任务 TCB 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

```
void OSTaskSwHook (void)
{
    /* 将浮点寄存器的内容储存在当前任务的TCB扩展域中。 */
    /* 用新任务的TCB扩展域中的值更新浮点寄存器的内容。   */
}
```

OSTaskStatHook()

void OSTaskStatHook(void)

File	*Called from*	*Code enabled by*
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

无

返回值

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

```
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      */
}
```

OSTimeTickHook()

void OSTimeTickHook(void)

File	*Called from*	*Code enabled by*
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示波器或逻辑分析仪来调试，或者为仿真器设置断点。

参数

无

返回值

无

注意事项

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的，这主要取决于该处理器上的移植是怎样进行的。如果中断是禁止的，该函数将会影响到中断响应时间。

范例

```
void OSTimeTickHook (void)
{
    /* 触发示波器                              */
}
```

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的为例只是一种更典型的情况。80x86 CPU每年的产量有数百万，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PUSHA指令，移植的时候要用几条PUSH指令来代替。

图F9.1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。

图F9.1　　80x86 实模式内部寄存器图.

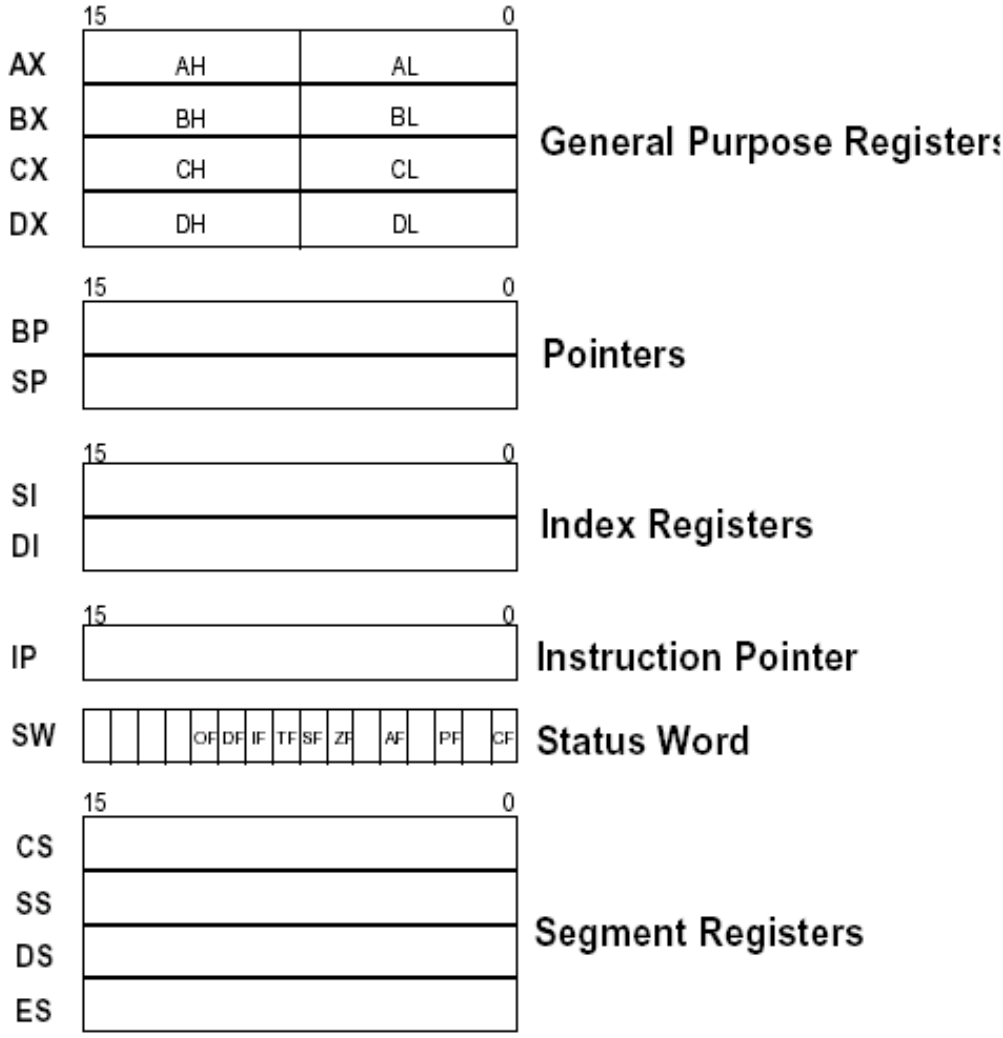


Figure 9-1, 80x86 Real-Mode Register Map

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F9.2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65,536个不同的段，因此可以寻址1,048,576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。

图F 9.2　　使用段寄存器和偏移量寄存器寻址.

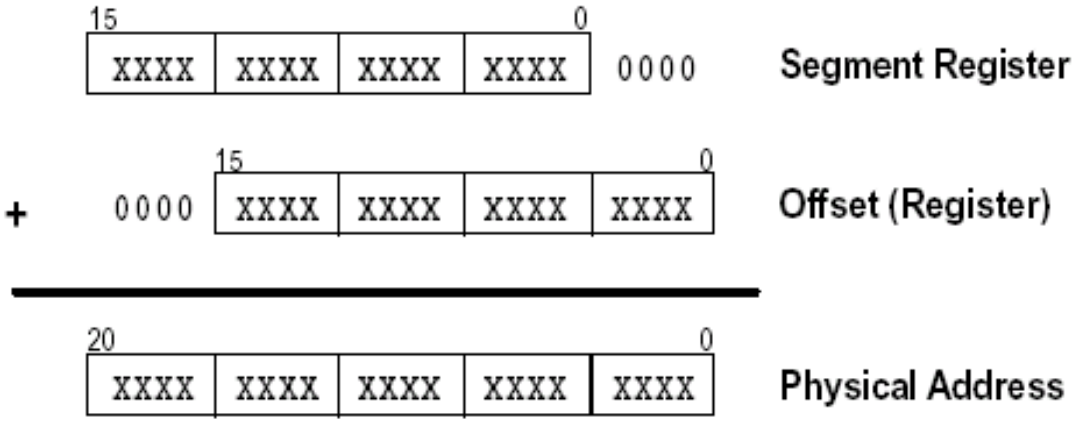


Figure 9-2, Addressing with a Segment and an Offset

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。

9.00 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。

只要您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，就只能麻烦您更改一下编译器和汇编器的设置了。

9.01 目录和文件

在安装μC/OS-II的时候，安装程序将把和硬件相关的，针对Intel 80x86的代码安装到\SOFTWARE\uCOS-II\Ix86L目录下。代码是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。

9.02 ***INCLUDES.H***文件

INCLUDES.H 是主头文件，在所有后缀名为.C的文件的开始都包含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，额外的增加编译时间。与优点相比，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L9.1是为80x86编写的INCLUDES.H文件的内容。

程序清单L *9.1* *INCLUDES.H.*

```
#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"
```

*9.03 OS_CPU.H*文件

OS_CPU.H 文件中包含与处理器相关的常量，宏和结构体的定义。程序清单L9.2是为80x86编写的OS_CPU.H文件的内容。

程序清单L *9.2 OS_CPU.H.*

```
#ifdef  OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT  extern
#endif
/*
*********************************************************************
******
*                              数据类型
*                         (与编译器相关的内容)
*********************************************************************
******
*/

typedef unsigned char BOOLEAN;
typedef unsigned char INT8U;          /* 无符号8位数                  (1)*/
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;   /* 堆栈入口宽度为16位                   */
```

```
#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

/*
*********************************************************************
******
*            Intel 80x86 (实模式, 大模式编译)
*********************************************************************
******
*/

#define  OS_STK_GROWTH   1 /* 堆栈由高地址向低地址增长                    (3)*/
```

```
#define  uCOS        0x80 /* 中断向量0x80用于任务切换                    (4)*/

#define  OS_TASK_SW() asm  INT   uCOS                                    (5)

/*
*********************************************************************************
******
*                              全局变量
*********************************************************************************
******
*/

OS_CPU_EXT  INT8U  OSTickDOSCtr;   /* 为调用DOS时钟中断而定义的计数器*/
                                                                        (6)*/
```

9.03.01 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，长整形（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。

9.03.02 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，您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段。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。

9.03.03 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L9.2(3)]。

9.03.04 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）（也称为例外处理过程）的入口点必须指向汇编函数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L9.2(4)]（PC和操作系统会占用一部分中断资源—译者注），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处理器，用户可能有更多的中断资源可供选用。

9.03.05 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（但不是必须的）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。如果您的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。

9.04 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()
```

9.04.01 OSStartHighRdy()

该函数由SStart()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F9.3给出了由函数OSTaskCreate()或 OSTaskCreateExt()创建的任务的堆栈结构。很明显，OSTCBHighRdy->OSTCBStkPtr指向的是任务堆栈的顶端。

函数OSStartHighRdy()的代码见程序清单L9.3。

图**F 9.3**　　任务创立时的**80x86**堆栈结构**.**

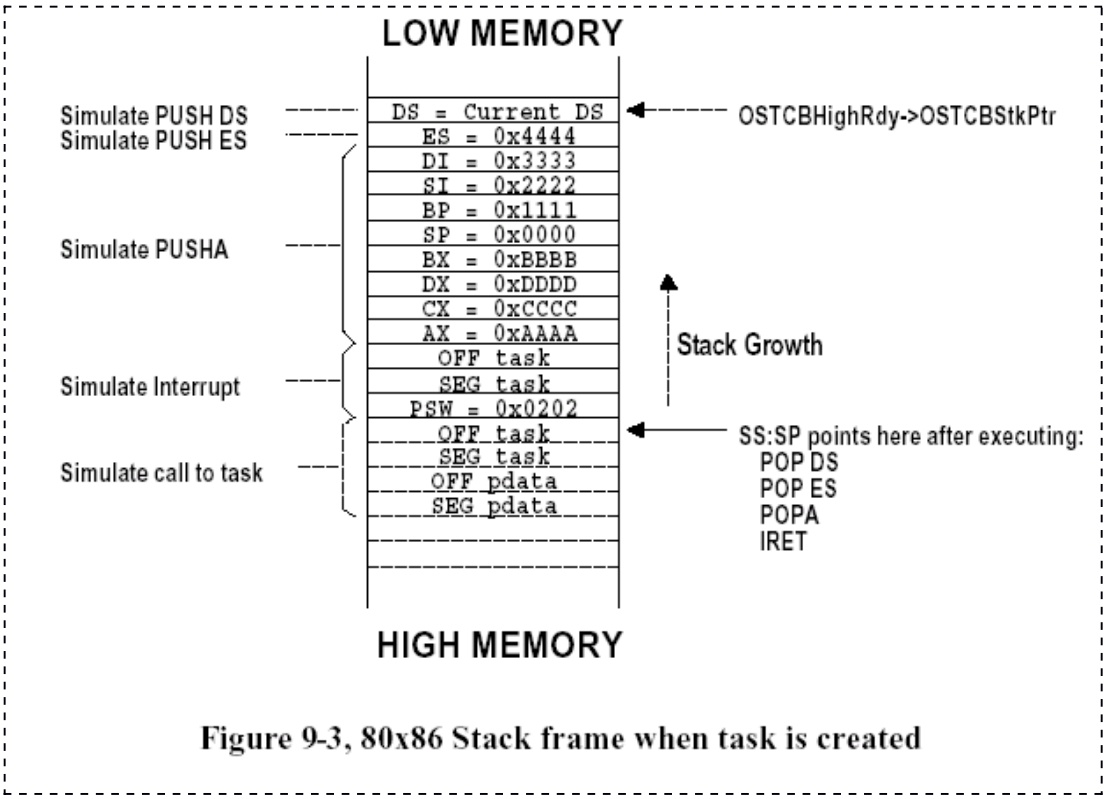


Figure 9-3, 80x86 Stack frame when task is created

为了启动任务，OSStartHighRdy()从任务控制块（OS_TCB）[程序清单L9.3(1)]中找到指向堆栈的指针，然后运行POP DS [程序清单L9.3(2)], POP ES [程序清单L9.3(3)], POPA [程序清单L9.3(4)], 和 IRET [程序清单L9.3(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。

程序清单L *9.3* ***OSStartHighRdy().***

```
_OSStartHighRdy  PROC FAR

   MOV    AX, SEG _OSTCBHighRdy         ; 载入 DS
   MOV    DS, AX                       ;

   LES    BX, DWORD PTR DS:_OSTCBHighRdy ; SS:SP = OSTCBHighRdy-
>OSTCBStkPtr                                                    (1)
   MOV    SS, ES:[BX+2]                 ;
   MOV    SP, ES:[BX+0]                 ;
;
   POP    DS                           ; 恢复任务环境           (2)
   POP    ES                           ;                        (3)
   POPA                                ;                        (4)
;
   IRET                                ; 运行任务               (5)

_OSStartHighRdy  ENDP
```

9.04.02 OSCtxSw()

OSCtxSw()是一个任务级的任务切换函数（在任务中调用，区别于在中断程序中调用的OSIntCtxSw()）。在80x86系统上，它通过执行一条软中断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L9.4为OSCtxSw()的代码。

图F9.4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F9.4/程序清单L9.4(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F9.4/程序清单L9.4(2)], PUSH ES [图F9.4/程序清单L9.4(3)],和 PUSH DS [图F9.4/程序清单L9.4(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。

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L9.4(6)]。请注意，此时OSTCBCur指向当前任务OS_TCB，OSTCBHighRdy指向新任务的OS_TCB。在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，请在头文件中把相应的开关选项关闭，加快任务切换的速度。

程序清单L 9.4 ***OSCtxSw().***

```
_OSCtxSw    PROC   FAR                                          (1)
```

```
;
   PUSHA                              ; 保存当前任务环境                    (2)
   PUSH ES                                                                 (3)
   PUSH DS                                                                 (4)
;
   MOV  AX, SEG _OSTCBCur             ; 载入DS
   MOV  DS, AX
;
   LES  BX, DWORD PTR DS:_OSTCBCur      ; OSTCBCur->OSTCBStkPtr = SS:S(5)
   MOV  ES:[BX+2], SS
   MOV  ES:[BX+0], SP
;
   CALL FAR PTR _OSTaskSwHook                                              (6)
;
   MOV  AX, WORD PTR DS:_OSTCBHighRdy+2  ; OSTCBCur = OSTCBHighRdy   (7)
   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  (8)
   MOV  BYTE PTR DS:_OSPrioCur, AL
;
   LES  BX, DWORD PTR DS:_OSTCBHighRdy   ; SS:SP = OSTCBHighRdy-
>OSTCBStkPtr                                                               (9)
   MOV  SS, ES:[BX+2]
   MOV  SP, ES:[BX]
;
   POP  DS                            ; 载入新任务的CPU环境              (10)
   POP  ES                                                                 (11)
   POPA                                                                    (12)
;
   IRET                               ; 返回新任务                       (13)
;
_OSCtxSw   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L9.4(7)]，同样，OSPrioHighRdy被拷贝到OSPrioCur中[程序清单L9.4(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F9.4(6)/程序清单L9.4(9)]，然后运行POP DS [图F9.4(7)/程序清单L9.4(10)]，POP ES [图F9.4(8)/程序清单L9.4(11)]，POPA [图F9.4(9)/程序清单L9.4(12)]取出其他寄存器的值，最后用中断返回指令IRET [图F9.4(10)/ L9.4(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。

9.04.03 OSIntCtxSw()

在μC/OS-II中，由于中断的产生可能会引起任务切换，在中断服务程序的最后会调用OSIntExit()函数检查任务就绪状态，如果需要进行任务切换，将调用OSIntCtxSw()。所以OSIntCtxSw()又称为中断级的任务切换函数。由于在调用OSIntCtxSw()之前已经发生了中断，OSIntCtxSw()将默认CPU寄存器已经保存在被中断任务的堆栈中了。

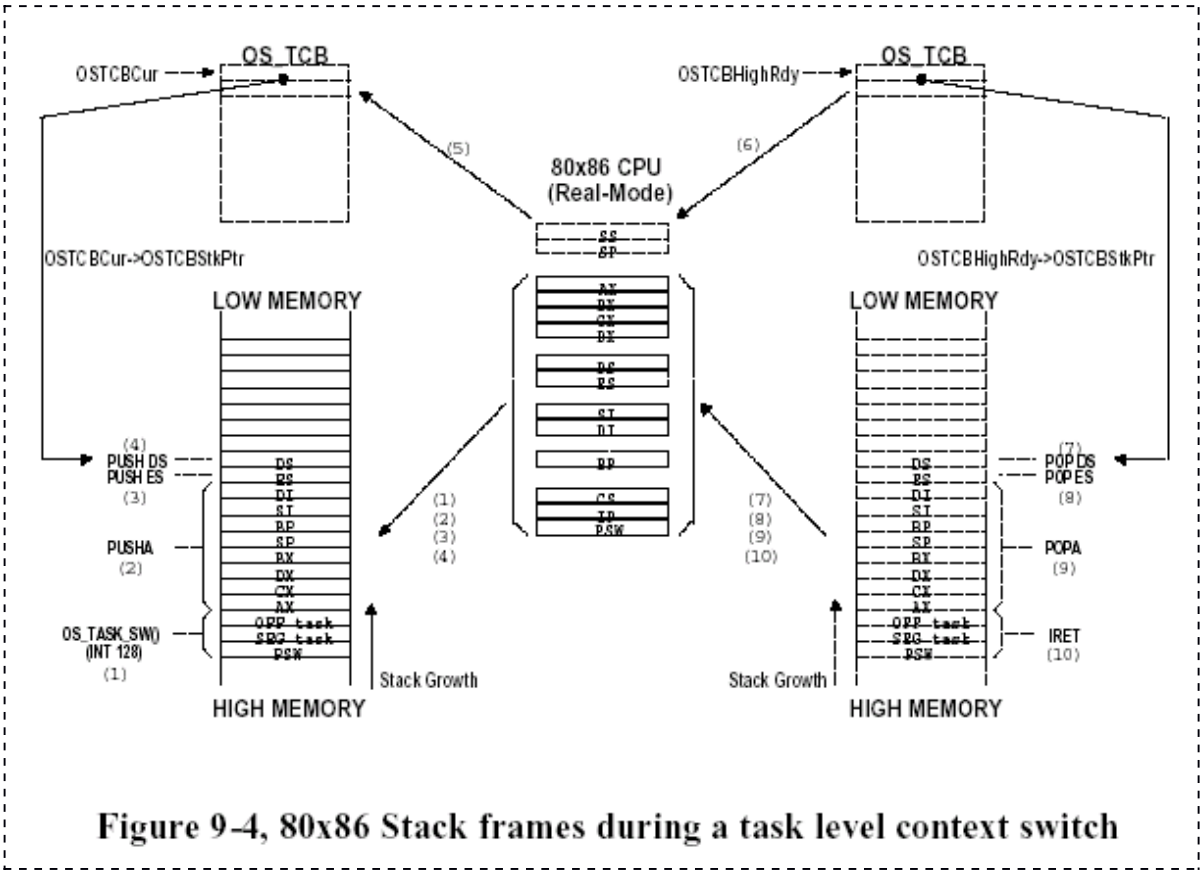


Figure 9-4, 80x86 Stack frames during a task level context switch

图F *9.4*　　任务级任务切换时的80x86堆栈结构.

程序清单L9.5给出的代码大部分与OSCtxSw()的代码相同，不同之处是，第一，由于中断已经发生，此处不需要再保存CPU寄存器(没有PUSHA, PUSH ES, 或PUSH DS)；第二，OSIntCtxSw()需要调整堆栈指针，去掉堆栈中一些不需要的内容，以使堆栈中只包含任务的运行环境。图F9.5可以帮助读者理解这一过程。

程序清单L *9.5*　*OSIntCtxSw()*.

```
_OSIntCtxSw PROC   FAR
;             ; Ignore calls to OSIntExit and OSIntCtxSw
;   ADD  SP,8    ; (Uncomment if OS_CRITICAL_METHOD is 1, see OS_CPU.H)(1)
    ADD  SP,10   ; (Uncomment if OS_CRITICAL_METHOD is 2, see OS_CPU.H)
;
    MOV  AX, SEG _OSTCBCur             ; 载入DS
    MOV  DS, AX
;
    LES  BX, DWORD PTR DS:_OSTCBCur     ; OSTCBCur->OSTCBStkPtr = SS:SP(2)
    MOV  ES:[BX+2], SS
    MOV  ES:[BX+0], SP
;
    CALL FAR PTR _OSTaskSwHook                                      (3)
```

```
;
   MOV  AX, WORD PTR DS:_OSTCBHighRdy+2 ; OSTCBCur = OSTCBHighRdy    (4)
   MOV  DX, WORD PTR DS:_OSTCBHighRdy
   MOV  WORD PTR DS:_OSTCBCur+2, AX
   MOV  WORD PTR DS:_OSTCBCur, DX
;
   MOV  AL, BYTE PTR DS:_OSPrioHighRdy  ; OSPrioCur = OSPrioHighRdy  (5)
   MOV  BYTE PTR DS:_OSPrioCur, AL
;
   LES  BX, DWORD PTR DS:_OSTCBHighRdy  ; SS:SP = OSTCBHighRdy-
>OSTCBStkPtr                                                         (6)
   MOV  SS, ES:[BX+2]
   MOV  SP, ES:[BX]
;
   POP  DS                           ; 载入新任务的CPU环境             (7)
   POP  ES                                                           (8)
   POPA                                                              (9)
;
   IRET                              ; 返回新任务                   (10)
;
_OSIntCtxSw ENDP
```

图F *9.5*　　中断级任务切换时的80x86堆栈结构

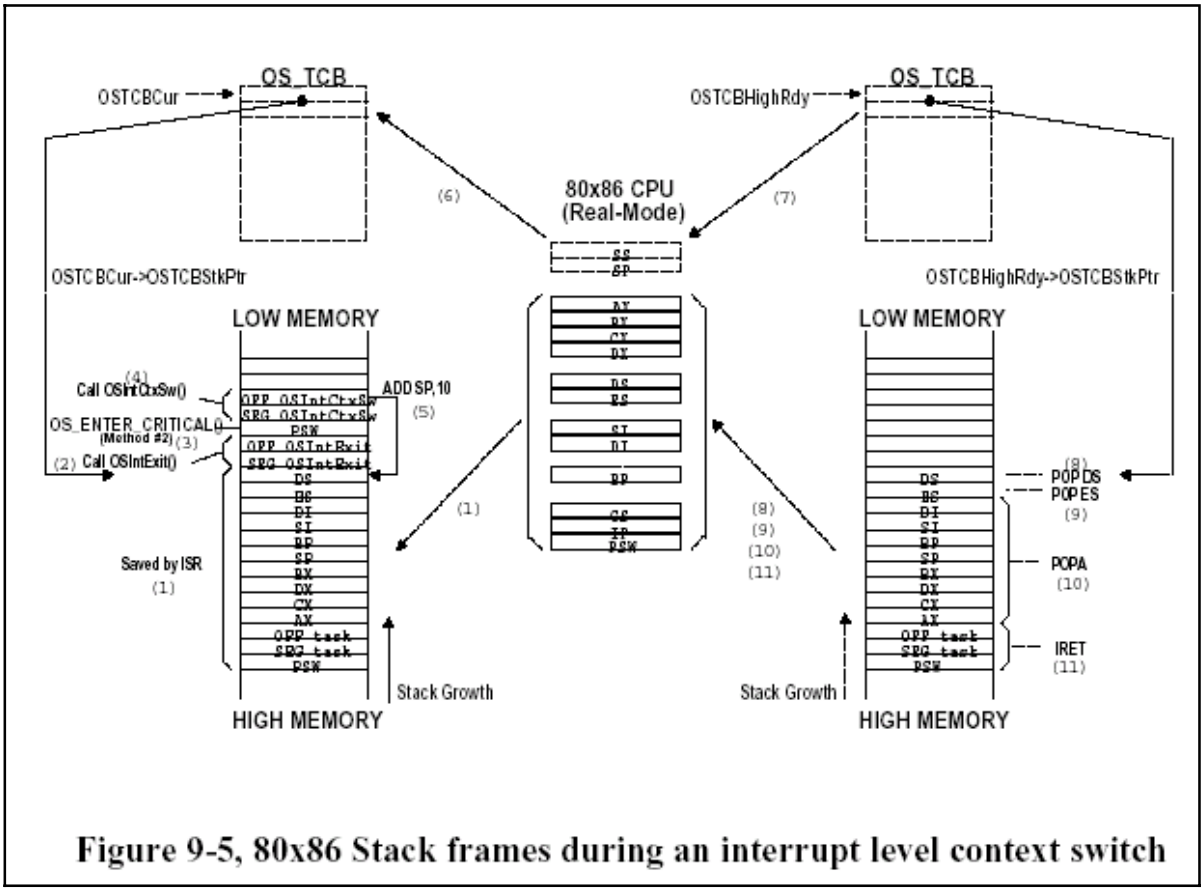


Figure 9-5, 80x86 Stack frames during an interrupt level context switch

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F9.5(1)]。此后，用户必须调用OSIntEnter()或着把全局变量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F9.5(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9.03.02节），状态寄存器SW的内容有可能被压入堆栈[图F9.5(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()会在再一次在堆栈中保存返回地址[图F9.5(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F9.5(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F9.5(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F9.5(5)/程序清单L9.5(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。

一但堆栈指针重新定位后，就被保存到将要被挂起的任务OS_TCB中[图F9.5(6)/程序清单L9.5(2)]。在μC/OS-II中（包括μC/OS），OSIntCtxSw()是唯一一个与编译器相关的函数，也是用户问的最多的。如果您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。

当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L9.5(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L9.5(4)]，同样，OSPrioHighRdy被拷贝到OSPrioCur中[程序清单L9.5(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F9.5(7)/程序清单L9.5(6)]，然后运行POP DS [图F9.5(8)/程序清单L9.5(7)], POP ES [图F9.5(9)/程序清单L9.5(8)], POPA[图F9.5(10)/程序清单L9.5(9)]取出其他寄存器的值,最后用中断返回指令IRET [图F9.5(11)/程序清单L9.5(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。

9.04.04 OSTickISR()

在9.03.05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F9.6为安装μC/OS-II前后的中断向量表。

在μC/OS-II中，当调用OSStart()启动多任务环境后，时钟中断的作用是非常重要的。但在PC环境下，启动μC/OS-II之前就已经有时钟中断发生了，实际上我们希望在μC/OS-II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

图F*9.6*　　PC 中断向量表(IVT).

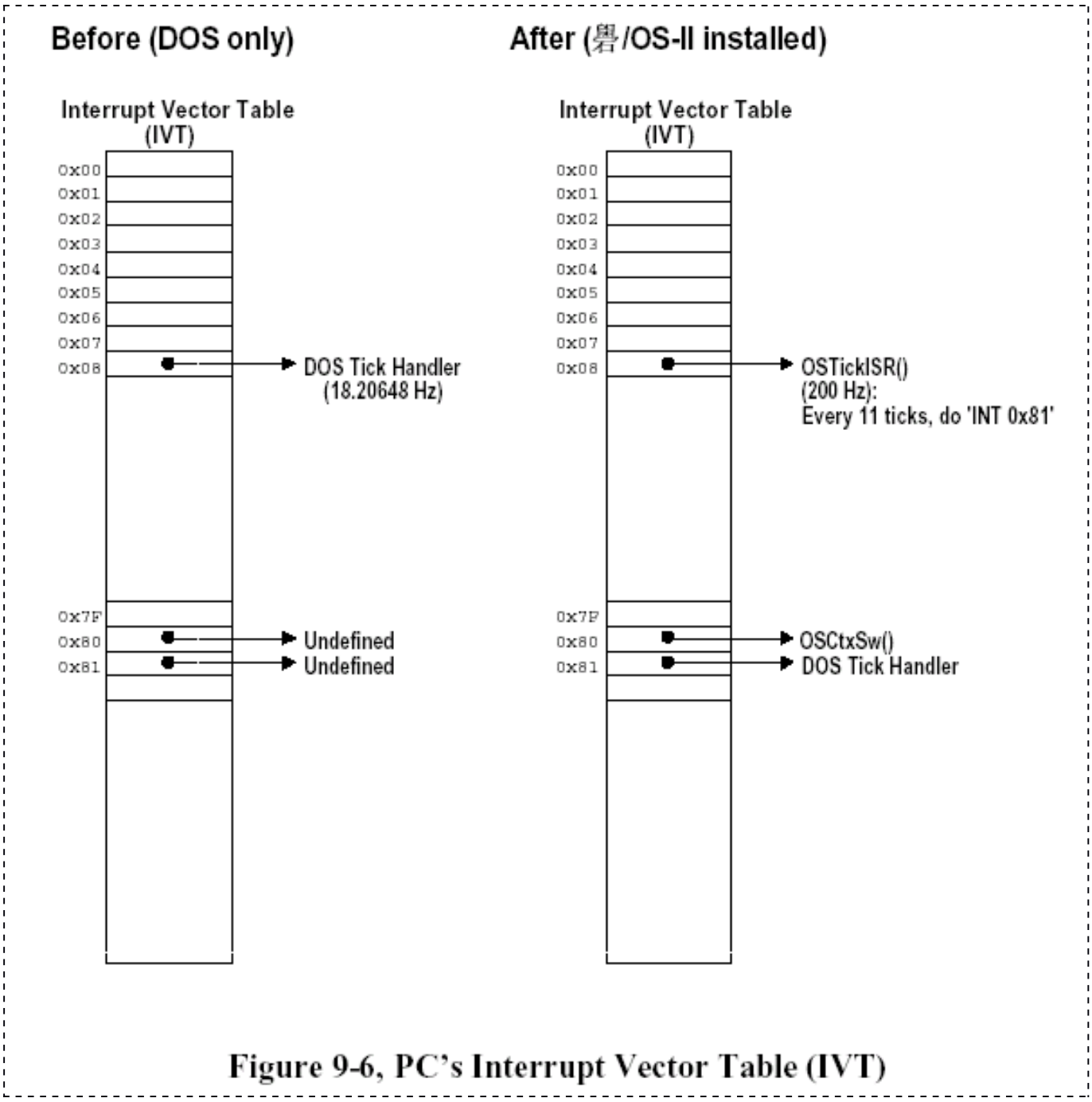


Figure 9-6, PC's Interrupt Vector Table (IVT)

在程序清单L9.6给出了函数OSTickISR()的伪码。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()。μC/OS-II要求在中断服务程序开头调用OSIntEnter()，其作用是将记录中断嵌套层数的全局变量OSIntNesting加1。如果不调用OSIntEnter()，直接将OSIntNesting加1也是允许的。接下来计数器OSTickDOSCtr减1[程序清单L9.6(3)]，每发生11次中断，OSTickDOSCtr减到0，则调用DOS的时钟中断处理函数[程序清单L9.6(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L9.6(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L9.7给出了OSTickISR()的完整代码。

程序清单L *9.6* **OSTickISR()伪码.**

```
void OSTickISR (void)
{
   Save processor registers;                                          (1)
   OSIntNesting++;                                                    (2)
   OSTickDOSCtr—-;                                                    (3)
   if (OSTickDOSCtr == 0) {
      Chain into DOS by executing an 'INT 81H' instruction;           (4)
   } else {
      Send EOI 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L9.7 *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   ; 标示 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       ; 标示uC/OS-II退出中断
;
    POP  DS                      ; 恢复被中断任务的CPU环境
    POP  ES
    POPA
;
    IRET                         ; 返回被中断任务
;
_OSTickISR  ENDP
```

如果不更改DOS下的时钟中断频率（保持18.20648 Hz），OSTickISR()函数还可以简化。程序清单L9.8为18.2 Hz的OSTickISR()函数的伪码。同样，函数开头要保存所有的CPU寄存器[程序清单L9.8(1)]，将OSIntNesting加1[程序清单L9.8(2)]。接下来调用DOS的时钟中断处理过程[程序清单L9.8(3)]，此处就不需要清除中断优先级控制器的操作了，因为DOS的时钟中断处理中包含了这一过程。然后调用OSTimeTick()检查任务的延时是否结束[程序清单L9.8(4)]，最后调用OSIntExit()[程序清单L9.8(5)]。结束部分是恢复CPU寄存器的内容[程序清单L9.8(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L9.9给出了18.2 Hz OSTickISR()的完整代码。

程序清单L 9.8　18.2Hz OSTickISR()伪码.

```
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)
}
```

9.05 OS_CPU_C.C

μC/OS-II 的移植需要用户改写OS_CPU_C.C中的六个函数：

OSTaskStkInit()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskSwHook()
OSTaskStatHook()
OSTimeTickHook()

实际需要修改的只有OSTaskStkInit()函数，其他五个函数需要声明，但不一定有实际内容。这五个函数都是用户定义的，所以OS_CPU_C.C中没有给出代码。如果用户需要使用这些函数，请将文件OS_CFG.H中的#define constant OS_CPU_HOOKS_EN设为1，设为0表示不使用这些函数。

程序清单L 9.9 18.2Hz 的OSTickISR()函数.

```
_OSTickISR  PROC   FAR
;
      PUSHA                          ; 保存被中断任务的CPU环境
      PUSH   ES
      PUSH   DS
;
      MOV    AX, SEG _OSIntNesting    ;载入 DS
      MOV    DS, AX
;
      INC    BYTE PTR _OSIntNesting    ;标示uC/OS-II进入中断
;
      INT    081H                    ; 调用DOS的时钟中断处理函数
;
      CALL   FAR PTR _OSTimeTick      ; 调用OSTimeTick（）函数
;
      CALL   FAR PTR _OSIntExit       ;标示uC/OS-II of中断结束
;
      POP    DS                      ; 恢复被中断任务的CPU环境
      POP    ES
      POPA
;
      IRET                           ; 返回被中断任务
;
```

```
_OSTickISR  ENDP
```

图F*9.7*　　传递参数 ***pdata***的堆栈初始化结构

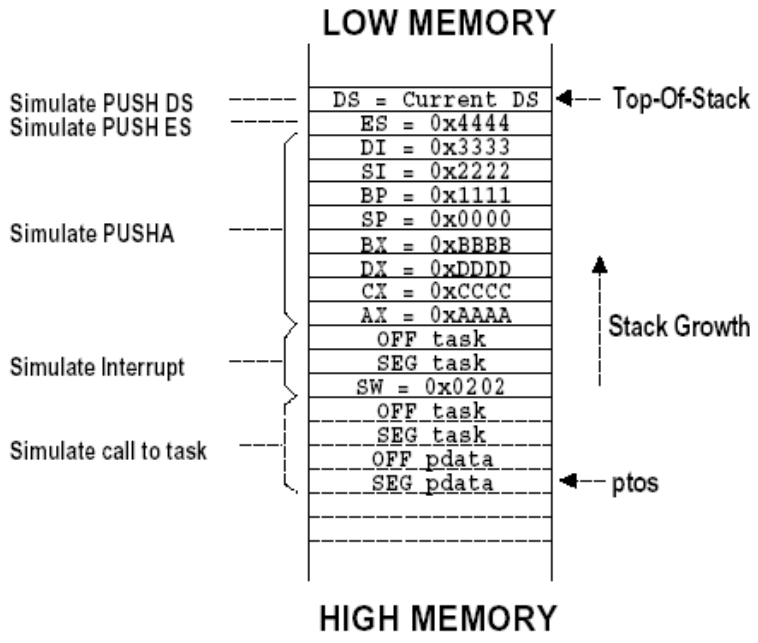


Figure 9-7, Stack frame initialization with 'pdata' passed on the stack

9.05.01 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F9.7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()任务的，而是新创建任务的。

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L9.10)只需要以上提到的3个参数（task, pdata,和ptos）。

程序清单L *9.10* ***OSTaskStkInit()***.

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



    opt     = opt;                  /* 'opt'未使用,此处可防止编译器的警告              */
    stk     = (INT16U *)ptos;         /* 载入堆栈指针                                  (1) */
    *stk--   = (INT16U)FP_SEG(pdata);   /* 放置向函数传递的参数                          (2) */
```

```
    *stk--   = (INT16U)FP_OFF(pdata);
    *stk--   = (INT16U)FP_SEG(task);    /* 函数返回地址(3) */
    *stk--   = (INT16U)FP_OFF(task);
    *stk--   = (INT16U)0x0202;         /* SW 设置为中断开启                          (4) */
    *stk--   = (INT16U)FP_SEG(task);    /* 堆栈顶端放置指向任务代码的指针*/
    *stk--   = (INT16U)FP_OFF(task);
    *stk--   = (INT16U)0xAAAA;         /* AX = 0xAAAA                            (5) */
    *stk--   = (INT16U)0xCCCC;         /* CX = 0xCCCC                                */
    *stk--   = (INT16U)0xDDDD;         /* DX = 0xDDDD                                */
    *stk--   = (INT16U)0xBBBB;         /* BX = 0xBBBB                                */
    *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 =当前CPU的 DS寄存器                     (6) */
    return ((void *)stk);
}
```

由于80x86 堆栈是16位宽的（以字为单位）[程序清单L9.10(1)]，OSTaskStkInit()将创立一个指向以字为单位内存区域的指针。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L9.10(2)]。

堆栈中紧接着是任务函数的起始地址[程序清单L9.10(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L9.10(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调用[程序清单L9.10(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L9.10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L9.10中（6）标记处用_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。

9.05.02 OSTaskCreateHook()

OS_CPU_C.C中未定义，此函数为用户定义。

9.05.03 OSTaskDelHook()

OS_CPU_C.C中未定义，此函数为用户定义。

9.05.04 OSTaskSwHook()

OS_CPU_C.C中未定义，此函数为用户定义。其用法请参考例程3。

9.05.05 OSTaskStatHook()

OS_CPU_C.C中未定义，此函数为用户定义。其用法请参考例程3。

9.05.06 OSTimeTickHook()

OS_CPU_C.C中未定义，此函数为用户定义。

9.06 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\uCOS-II\Ix836L\DOC\目录下的ROM-RAM.XLS文件。该文件为Microsoft Excel文件,需要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6,875字节，可减小大约1,475字节。

此外，空闲任务（idle）和统计任务（statistics）的堆栈都设为1,024字节(1Kb)。根据您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9.2说明了如何裁减μC/OS-II，应用在更小规模的系统上。此处的小系统有16个任务。并且不采用如下功能：

•邮箱功能(OS_MBOX_EN设为0)

•内存管理机制(OS_MEM_EN设为0)

•动态改变任务优先级(OS_TASK_CHANGE_PRIO_EN设为0)

•旧版本的任务创建函数OSTaskCreate()(OS_TASK_CREATE_EN设为0)

•任务删除(OS_TASK_DEL_EN设为0)

•挂起和唤醒任务(OS_TASK_SUSPEND_EN设为0)

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。

9.07 运行时间

表9.3到9.5列出了大部分μC/OS-II函数在80186处理器上的运行时间。统计的方法是将C

原程序编译为汇编代码，然后计算每条汇编指令所需的时钟周期，根据处理器的时钟频率，最后算出运行时间。表中的I 栏为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为运行所需的以微秒为单位的时间。表中有3类时间，分别是在函数中关闭中断的时间、函数运行的最小时间和最大时间。如果您不使用80186处理器，表中的数据就没有什么实际意义，但可以使您理解每个函数运行时间的相对大小。

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		2,925
OS_LOWEST_PRIO	63		264
OS_TASK_IDLE_STK_SIZE	512		1,024
OS_TASK_STAT_EN	1	325	10
OS_TASK_STAT_STK_SIZE	512		1,024
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	1,475	(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		2,700	35
应用程序堆栈	0		0
应用程序的RAM	0		0
总计		8,350	5,675

表 9.2 压缩后的μC/OS-II配置.

配置参数	值	代码 (字节)	数据 (字节)
OS_MAX_EVENTS	10		164
OS_MAX_MEM_PART	5		0
OS_MAX_QS	5		124
OS_MAX_TASKS	16		792
OS_LOWEST_PRIO	63		264
OS_TASK_IDLE_STK_SIZE	512		1,024
OS_TASK_STAT_EN	1	325	10
OS_TASK_STAT_STK_SIZE	512		1,024
OS_CPU_HOOKS_EN	1		0
OS_MBOX_EN	0	0	(参看OS_MAX_EVENTS)
OS_MEM_EN	0	0	(参看OS_MAX_MEM_PART)
OS_Q_EN	1	1,475	(参看OS_MAX_QS)
OS_SEM_EN	1	475	(参看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
µC/OS-II内核		2,700	35
应用程序堆栈	0		0
应用程序的RAM	0		0
总计		5,275	3,438

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1,100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初始化函数。

如果您用的是x86系列的其他CPU，您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。

表 9.3　　μC/OS-II函数在33MHz 80186上的执行时间.

	关闭中断时间			最小运行时间			最大运行时间		
函数	***I***	***C***	***μs***	***I***	***C***	***μs***	***I***	***C***	***μs***
杂项									
OSInit()	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OSSchedLock()	4	34	1.0	7	87	2.6	7	87	2.6
OSSchedUnlock()	57	567	17.2	13	130	3.9	73	782	23.7
OSStart()	0	0	0.0	35	278	8.4	35	278	8.4
OSStatInit()	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OSVersion()	0	0	0.0	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()	56	558	16.9	27	207	6.3	57	574	17.4
OSTickISR()	30	310	9.4	948	10,803	327.4	2,304	20,620	624.8
邮箱									
OSMboxAccept()	15	161	4.9	13	257	7.8	13	257	7.8
OSMboxCreate()	15	148	4.5	115	939	28.5	115	939	28.5
OSMboxPend()	68	567	17.2	28	317	9.6	184	1,912	57.9
OSMboxPost()	84	747	22.6	24	305	9.2	152	1,484	45.0
OSMboxQuery()	120	988	29.9	128	1,257	38.1	128	1,257	38.1

表9.3　　μC/OS-II函数在33MHz 80186上的执行时间.*（续表）*

内存管理									
OSMemCreate()	21	181	5.5	72	766	23.2	72	766	23.2
OSMemGet()	19	247	7.5	18	172	5.2	33	350	10.6
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	1,291	39.1	154	1,291	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	1,938	58.7
OSQPost()	98	873	26.5	51	547	16.6	155	1,493	45.2
OSQPostFront()	87	788	23.9	44	412	12.5	153	1,483	44.9
OSQQuery()	128	1,100	33.3	137	1,171	35.5	137	1,171	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	1,690	51.2
OSSemPost()	87	776	23.5	18	198	6.0	151	1,469	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	1,532	46.4
OSTaskCreate()	57	567	17.2	217	2,388	72.4	266	2,939	89.1
OSTaskCreateExt()	57	567	17.2	235	2,606	79.0	284	3,157	95.7
OSTaskDel()	62	620	18.8	116	1,206	36.5	165	1,757	53.2
OSTaskDelReq()	23	199	6.0	39	330	10.0	39	330	10.0
OSTaskQuery()	84	1,025	31.1	95	1,122	34.0	95	1,122	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	1,130	34.2

表9.3　　μC/OS-II函数在33MHz 80186上的执行时间.*（续表）*

时钟管理									
OSTimeDly()	57	567	17.2	81	844	25.6	85	871	26.4
OSTimeDlyHMSM()	57	567	17.2	216	2,184	66.2	220	2,211	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	10,257	310.8	1,908	19,707	597.2
用户定义函数									
OSTaskCreateHook()	0	0	0.0	4	38	1.2	4	38	1.2
OSTaskDelHook()	0	0	0.0	4	38	1.2	4	38	1.2
OSTaskStatHook()	0	0	0.0	1	16	0.5	1	16	0.5
OSTaskSwHook()	0	0	0.0	1	16	0.5	1	16	0.5
OSTimeTickHook()	0	0	0.0	1	16	0.5	1	16	0.5

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，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（处理延时结束情况的代码最长—译者注），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（处理延时结束情况的代码最长—译者注），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。

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（处理延时结束情况的代码最长—译者注），最后返回调用任务。OSQ`Pend()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（处理延时结束情况的代码最长—译者注），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OSQPend()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（处理延时结束情况的代码最长—译者注），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（处理延时结束情况的代码最长—译者注），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（堆栈操作以字为单位—译者注）需要100个时钟周期(3μs)，1000字节的堆栈将需要1,500μs（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。

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（堆栈操作以字为单位—译者注）需要80钟周期(2.4μs)，1000字节的堆栈将需要1,200μs（1000字节除以2再乘以2.4μs/每字）。再加上其他的一些操作，总共需要大约1,218μs。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65,536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)。

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OSTimeTick()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OSTimeTick()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。例如在最坏的情况，设时钟节拍间隔10ms，OSTimeTick()需要约600μs，占了6%的CPU利用率

表 9.4　　各函数的执行时间（按关闭中断时间排序）.

	关闭中断时间			最小运行时间			最大运行时间		
函数	***I***	***C***	***μs***	***I***	***C***	***μs***	***I***	***C***	***μs***
OSVersion()	0	0	0.0	2	19	0.6	2	19	0.6
OSStart()	0	0	0.0	35	278	8.4	35	278	8.4
OSSchedLock()	4	34	1.0	7	87	2.6	7	87	2.6
OSIntEnter()	4	42	1.3	4	42	1.3	4	42	1.3
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
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
OSMboxCreate()	15	148	4.5	115	939	28.5	115	939	28.5
OSQCreate()	14	150	4.5	154	1,291	39.1	154	1,291	39.1
OSMboxAccept()	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
OSTaskDelReq()	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
OSTaskResume()	27	242	7.3	48	430	13.0	97	981	29.7
OSMemGet()	19	247	7.5	18	172	5.2	33	350	10.6
OSMemPut()	23	282	8.5	12	161	4.9	29	321	9.7
OSTimeTick()	30	310	9.4	900	10,257	310.8	1,908	19,707	597.2
OSTickISR()	30	310	9.4	948	10,803	327.4	2,304	20,620	624.8
OSTaskStkChk()	31	316	9.6	62	599	18.2	62	599	18.2
OSTaskSuspend()	37	352	10.7	63	579	17.5	112	1,130	34.2
OSQAccept()	34	387	11.7	25	269	8.2	44	479	14.5
OSMemQuery()	40	400	12.1	45	450	13.6	45	450	13.6
OSIntExit()	56	558	16.9	27	207	6.3	57	574	17.4
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	1,532	46.4
OSSemPend()	58	567	17.2	17	184	5.6	164	1,690	51.2
OSMboxPend()	68	567	17.2	28	317	9.6	184	1,912	57.9
OSTimeDlyHMSM()	57	567	17.2	216	2,184	66.2	220	2,211	67.0
OSTaskCreate()	57	567	17.2	217	2,388	72.4	266	2,939	89.1
OSTaskCreateExt()	57	567	17.2	235	2,606	79.0	284	3,157	95.7
OSTaskDel()	62	620	18.8	116	1,206	36.5	165	1,757	53.2
OSQPend()	64	620	18.8	45	495	15.0	186	1,938	58.7
OSMboxPost()	84	747	22.6	24	305	9.2	152	1,484	45.0
OSSemPost()	87	776	23.5	18	198	6.0	151	1,469	44.5
OSQPostFront()	87	788	23.9	44	412	12.5	153	1,483	44.9
OSQPost()	98	873	26.5	51	547	16.6	155	1,493	45.2
OSSemQuery()	110	882	26.7	116	931	28.2	116	931	28.2
OSMboxQuery()	120	988	29.9	128	1,257	38.1	128	1,257	38.1
OSTaskQuery()	84	1,025	31.1	95	1,122	34.0	95	1,122	34.0
OSQQuery()	128	1,100	33.3	137	1,171	35.5	137	1,171	35.5
OSStatInit()	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OSInit()	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

表9.5　　各函数的执行时间（按最大运行时间排序）.

	关闭中断时间			最大运行时间			最小运行时间		
Service	***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
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	1,025	31.1	95	1,122	34.0	95	1,122	34.0
OSTaskSuspend()	37	352	10.7	63	579	17.5	112	1,130	34.2
OSQQuery()	128	1,100	33.3	137	1,171	35.5	137	1,171	35.5
OSMboxQuery()	120	988	29.9	128	1,257	38.1	128	1,257	38.1
OSQCreate()	14	150	4.5	154	1,291	39.1	154	1,291	39.1
OSSemPost()	87	776	23.5	18	198	6.0	151	1,469	44.5
OSQPostFront()	87	788	23.9	44	412	12.5	153	1,483	44.9
OSMboxPost()	84	747	22.6	24	305	9.2	152	1,484	45.0
OSQPost()	98	873	26.5	51	547	16.6	155	1,493	45.2
OSTaskChangePrio()	63	567	17.2	178	981	29.7	166	1,532	46.4
OSSemPend()	58	567	17.2	17	184	5.6	164	1,690	51.2
OSTaskDel()	62	620	18.8	116	1,206	36.5	165	1,757	53.2
OSMboxPend()	68	567	17.2	28	317	9.6	184	1,912	57.9
OSQPend()	64	620	18.8	45	495	15.0	186	1,938	58.7
OSTimeDlyHMSM()	57	567	17.2	216	2,184	66.2	220	2,211	67.0
OSTaskCreate()	57	567	17.2	217	2,388	72.4	266	2,939	89.1
OSTaskCreateExt()	57	567	17.2	235	2,606	79.0	284	3,157	95.7
OSTimeTicK()	30	310	9.4	900	10,257	310.8	1,908	19,707	597.2
OSTickISR()	30	310	9.4	948	10,803	327.4	2,304	20,620	624.8
OSInit()	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OSStatInit()	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

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上，移植µC/OS-II所要做的工作应当非常有限。在多数情况下，用户能够在1个小时之内完成这项工作。如果用户熟悉µC/OS的移植，可隔过本章前一部分直接参阅10.05节。

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 Modle）时，文件名为Ix86L.H， Ix86L_C.C， 和Ix86L_A.ASM.。

表 L10.1在µC/OS-II中重新命名的文件.

\SOFTWARE\uCOS\Ix86L	\SOFTWARE\uCOS-II\Ix86L
Ix86L.H	OS_CPU.H
Ix86L_A.ASM	OS_CPU_A.ASM
Ix86L_C.C	OS_CPU_C.C

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。

表 L10.2对不同微处理器从µC/OS到µC/OS-II，要重新命名的文件.

\SOFTWARE\uCOS\I80251	\SOFTWARE\uCOS-II\I80251
I80251.H	OS_CPU.H
I80251.C	OS_CPU_C.C
\SOFTWARE\uCOS\M680x0	\SOFTWARE\uCOS-II\M680x0
M680x0.H	OS_CPU.H
M680x0.C	OS_CPU_C.C
\SOFTWARE\uCOS\M68HC11	\SOFTWARE\uCOS-II\M68HC11
M68HC11.H	OS_CPU.H
M68HC11.C	OS_CPU_C.C

\SOFTWARE\uCOS\Z80	\SOFTWARE\uCOS-II\Z80
Z80.H	OS_CPU.H
Z80_A.ASM	OS_CPU_A.ASM
Z80_C.C	OS_CPU_C.C

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 IX86L.H 为 OS_CPU.H
- 变文件名UCOS.H 为 uCOS_II.H

新旧文件如程序清单 L10.1和 L10.2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 L10.3所示。

程序清单 L10.1 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 L10.2 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"
```

程序清单 L10.3μC/OS到μC/OS-II 数据类型的修改.

```
/* uC/OS data types:                                                  */
typedef unsigned char  UBYTE;    /* Unsigned  8 bit quantity          */
typedef signed   char  BYTE;     /* Signed    8 bit quantity          */
typedef unsigned int   UWORD;    /* Unsigned 16 bit quantity          */
typedef signed   int   WORD;     /* Signed   16 bit quantity          */
typedef unsigned long  ULONG;    /* Unsigned 32 bit quantity          */
typedef signed   long  LONG;     /* Signed   32 bit quantity          */

/* uC/OS-II data types                                                */
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          */
```

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 L10.4所示。

程序清单 L10.4 μC/OS 和 μC/OS-II任务栈的数据类型

```
#define OS_STK_TYPE  UWORD       /* 在 uC/OS 中                  */
#define OS_STK      INT16U      /*  在 uC/OS-II 中              */
```

10.2.2OS_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。.

10.2.3OS_STK_GROWTH

大多数微处理器和微处理器的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，故要加到OS_CPU.H中去。

10.2.4OS_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()的调用总是出现在任务级代码中。

10.2.5OS_FAR

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。结果是对于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L10.5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 L10.5 在 μC/OS 中任务函数的定义

```
void OS_FAR task (void *pdata)
{
    pdata = pdata;
    while (1) {
        .
        .
    }
}
```

10.3OS_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()
```

10.3.1OSStartHighRdy()

在μC/OS-II中，OSStartHighRdy ()要调用OSSTaskSwHook()。OSTaskSwHook()这个函数在μC/OS中没有。用户将最高优先级任务的栈指针装入CPU之前要先调用OSTaskSwHook()。还有， OSStartHighRdy要在调用OSTaskSwHook()之后立即将OSRunning设为1。程序清

单L10.6 给出OSStartHighRdy()的示意代码。.μC/OS只有其中最后三步。

程序清单 L10.6 OSStartHighRdy()的示意代码

```
OSStartHighRdy:
    Call OSTaskSwHook();                                    调用OSTaskSwHook();
    Set OSRunning to 1;                                     置 OSRunning 为
1;
    Load the processor stack pointer with OSTCBHighRdy->OSTCBStkPtr;
                                将 OSTCBHighRdy->OSTCBStkPtr 装入处理器的栈指
针;
    POP all the processor registers from the stack;  从栈中弹出所有寄存器的值;
    Execute a Return from Interrupt instruction;            执行中断返回指令;
```

10.3.2OSCtxSw()

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OSPrioCur设为OSPrioHighRdy 。OSCtxSw()的示意代码如程序清单L10.7所示。μC/OS-II加上了步骤L10.7(1)和(2)。

程序清单 L10.7 OSCtxSw()的示意代码

```
OSCtxSw:
    PUSH processor registers onto the current task’s stack;
                                                所有处理器寄存器的值推入当前任务栈;
    Save the stack pointer at OSTCBCur->OSTCBStkPtr;
    Call OSTaskSwHook();                                1)
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;                                              (2)
    Load the processor stack pointer with OSTCBHighRdy->OSTCBStkPtr;
                                将 OSTCBHighRdy->OSTCBStkPtr 装入处理器的栈指
针;
    POP all the processor registers from the stack;  从栈中弹出所有寄存器的值;
    Execute a Return from Interrupt instruction;
```

10.3.3OSIntCtxSw()

如同上述函数一样，在μC/OS-II.中，OSCtxSw()也增加了两件事。首先，将当前任务的栈指

针保存到当前任务的控制块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OSPrioCur 设为OSPrioHighRdy。程序清单L10.8给出OSIntCtxSw()的示意代码。μC/OS-II.中增加了L10.8（1）和 (2)。

程序清单 L10.8 OSIntCtxSw()的示意代码

```
OSIntCtxSw():
   Adjust the stack pointer to remove call to OSIntExit(), locals in
OSIntExit()
     and the call to OSIntCtxSw();
                                        调整由于调用上述子程序引起的栈指针值的变化；
   Save the stack pointer at OSTCBCur->OSTCBStkPtr;
                                              保存栈指针到OSTCBCur->OSTCBStkPtr;
   Call OSTaskSwHook();                                调用OSTaskSwHook();(1)
   OSTCBCur  = OSTCBHighRdy;
   OSPrioCur = OSPrioHighRdy;                                                (2)
   Load the processor stack pointer with OSTCBHighRdy->OSTCBStkPtr;
                               将 OSTCBHighRdy->OSTCBStkPtr 装入处理器的栈指针；
   POP all the processor registers from the stack; 从栈中弹出所有寄存器的值；
   Execute a Return from Interrupt instruction;           执行中断返回指令；
```

10.3.4OSTickISR()

在μC/OS-II和μC/OS 中，这个函数的代码是一样，无须改变。

10.4OS_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。其它5个只需定义，而不包括任何代码。

10.4.1OSTaskStkInit()

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()只负责设定任务的栈，使之看起来好像中断刚刚发生过，所有的CPU寄存器都被推入堆栈。作为提供给用户的例子，程序清单L10.9给出Intel 80x86实模式，在大模式下编译的 μC/OS的OSTaskCreate()函数的代码。程序清单L10.10是同类微微处理器的μC/OS-II的OSTaskStkInit()函数的代码。比较这两段代码，可以看出：从 [L10.9(1)] OS_EXIT_CRIITICAL() 到 [L10.9(2)] 调用 OSTaskStkInit() 都抽出来并移到了OSTaskStkInit()中。

程序清单L10.9 μC/OS 中的 OSTaskCreate()

```
UBYTE OSTaskCreate(void (*task)(void *pd), void *pdata, void *pstk, UBYTE
p)
{
    UWORD OS_FAR *stk;
    UBYTE        err;



    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[p] == (OS_TCB *)0) {
        OSTCBPrioTbl[p] = (OS_TCB *)1;
        OS_EXIT_CRITICAL();                                          (1)
        stk    = (UWORD OS_FAR *)pstk;
        *--stk = (UWORD)FP_OFF(pdata);
        *--stk = (UWORD)FP_SEG(task);
        *--stk = (UWORD)FP_OFF(task);
        *--stk = (UWORD)0x0202;
        *--stk = (UWORD)FP_SEG(task);
        *--stk = (UWORD)FP_OFF(task);
        *--stk = (UWORD)0x0000;
        *--stk = (UWORD)0x0000;
        *--stk = (UWORD)0x0000;
        *--stk = (UWORD)0x0000;
        *--stk = (UWORD)0x0000;
        *--stk = (UWORD)0x0000;
        *--stk = (UWORD)0x0000;
        *--stk = (UWORD)0x0000;
        *--stk = (UWORD)0x0000;
        *--stk = (UWORD)0x0000;
        *--stk = _DS;
        err    = OSTCBInit(p, (void far *)stk);                      (2)
        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 L10.10 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;
    *stk-- = (INT16U)0x3333;
    *stk-- = (INT16U)0x4444;
    *stk   = _DS;
    return ((void *)stk);
}
```

10.4.2OSTaskCreateHook()

OSTaskCreateHook()在μC/OS中没有，如程序清单L10.11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
OSTaskCreateHook(OS_TCB *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。这样做的好处是允许用户移植时可在不同文件中定义钩子函数。

10.4.3OSTaskDelHook()

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 L10.12 μC/OS-II中的OSTaskDelHook().

```
#if OS_CPU_HOOKS_EN
OSTaskDelHook(OS_TCB *ptcb)
{
    ptcb = ptcb;
}
#endif
```

也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不同的文件中定义钩子函数。

10.4.4OSTaskSwHook()

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L10.13所示。

程序清单 L10.13 μC/OS-II中的OSTaskSwHook()函数

```
#if OS_CPU_HOOKS_EN
OSTaskSwHook(void)
{
}
#endif
```

也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。.

10.4.5OSTaskStatHook()

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L10.14所示。

也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 L10.14 μC/OS-II中的OSTaskStatHook()函数

```
#if OS_CPU_HOOKS_EN
OSTaskStatHook(void)
{
}
#endif
```

10.4.6OSTimeTickHook()

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L10.15所示。

也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，OSTimeTickHook()的代码才能生成。

.

程序清单 L10.15 μC/OS-II中的OSTimeTickHook()

```
#if OS_CPU_HOOKS_EN
OSTimeTickHook(void)
{
}
#endif
```

10.5总结

表T10.3总结了从μC/OS向μC/OS-II.升级需要改变得地方。其中*processor_name.?是*μC/OS中移植范例程序的文件名。

表 T10.3 升级 μC/OS到 μC/OS-I要修改的地方

µC/OS	*µC/OS-II*
Processor_name.H	OS_CPU.H
数据类型: UBYTE BYTE UWORD WORD ULONG LONG	数据类型: INT8U INT8S INT16U INT16S INT32U 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_FAR	定义OS_FAR 为空，或删除所有的 OS_FAR
Processor_name.ASM	OS_CPU_A.ASM
OSStartHighRdy()	增加了调用 OSTaskSwHook(); 置 OSRunning 为 1 (8 bits)
OSCtxSw()	增加了调用 OSTaskSwHook(); 拷贝OSPrioHighRdy 到 OSPrioCur (8 bits)
OSIntCtxSw()	增加了调用OSTaskSwHook(); 拷贝 OSPrioHighRdy 到 OSPrioCur (8 bits)
OSTickISR()	不变
Processor_name.C	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 11 章

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
- 函数使用到的常量
- 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：

```
void main (void)
{
     .
     .
     OSInit();     /* 初始化 uC/OS-II */
     .
     .
     OSStart();    /*启动多任务内核  */
}
```

OSIntEnter( )

Void OSIntEnter（void）;

所属文件	调用者	开关量
OS_CORE.C	中断	无

OSIntEnter（）通知μC/OS-Ⅱ一个中断处理函数正在执行，这有助于μC/OS-Ⅱ掌握中断嵌套的情况。OSIntEnter（）函数通常和 OSIntExit（）函数联合使用。

参数

无

返回值

无

注意/警告

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，，real mode，large model）

```
    ISRx PROC   FAR
         PUSHA                      ; 保存中断现场
         PUSH   ES
         PUSH   DS
;
         MOV    AX, DGROUP          ; 读入数据段
         MOV    DS, AX
;
         CALL   FAR PTR _OSIntEnter  ; 通知内核进入中断
         .
         .
         POP    DS                  ; 恢复中断现场
```

```
        POP    ES
        POPA
        IRET                    ; 中断返回
    ISRx ENDP
```

范例二：

（Intel 80x86 的实模式，在大模式下编译，, real mode , large model）

```
    ISRx    PROC   FAR
            PUSHA                       ; 保存中断现场
            PUSH   ES
            PUSH   DS
;
            MOV    AX, DGROUP            ; 读入数据段
            MOV    DS, AX
;
            INC    BYTE PTR _OSIntNesting  ; 通知内核进入中断
            .
            .
            .
            POP    DS                   ; 恢复中断现场
            POP    ES
            POPA
            IRET                        ; 中断返回
    ISRx    ENDP
```

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

无

返回值

无

注意/警告

在任务级不能调用该函数。并且即使没有调用 OSIntEnter（）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， real mode , large model）

```
ISRx    PROC    FAR
        PUSHA                   ; 保存中断现场
        PUSH    ES
        PUSH    DS
        .
        .
        CALL    FAR PTR _OSIntExit ; 通知内核进入中断
        POP     DS              ; 恢复中断现场
        POP     ES
        POPA
        IRET                    ; 中断返回
ISRx    ENDP
```

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序。(参考 OSMboxCreate（）函数)。

返回值

如果消息已经到达，返回指向该消息的指针；如果消息邮箱没有消息，返回空指针。

注意/警告

必须先建立消息邮箱，然后使用。

范例：

```
OS_EVENT *CommMbox;



void Task (void *pdata)
{
   void *msg;



   pdata = pdata;
   for (;;) {
      msg = OSMboxAccept(CommMbox); /* 检查消息邮箱是否有消息             */
      if (msg != (void *)0) {
        .                        /* 处理消息                           */
        .
      } else {
        .                        /*没有消息                            */
        .
      }
     .
     .
   }
}
```

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

msg 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息。

返回值

指向分配给所建立的消息邮箱的事件控制块的指针。如果没有可用的事件控制块，返回空指针。

注意/警告

必须先建立消息邮箱，然后使用。

范例：

```
OS_EVENT *CommMbox;



   void main(void)
   {
       .
       .
       OSInit();                          /* 初始化μC/OS-Ⅱ */
       .
       .
       CommMbox = OSMboxCreate((void *)0);    /* 建立消息邮箱  */
       OSStart();                         /* 启动多任务内核  */
}
```

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 65,535 个时钟节拍。这个时间长度并不是非常严格的，可能存在一个时钟节拍的误差，因为只有在一个时钟节拍结束后才会减少定义的等待超时时钟节拍。

Err 是指向包含错误码的变量的指针。OSMboxPend（）函数返回的错误码可能为下述几种：

- 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：

```
OS_EVENT *CommMbox;



void CommTask(void *pdata)
{
     INT8U  err;
     void  *msg;



     pdata = pdata;
     for (;;) {
        .
        .
        msg = OSMboxPend(CommMbox, 10, &err);
        if (err == OS_NO_ERR) {
            .
            .  /* 消息正确的接受              */
            .
        } else {
            .
            .  /* 在指定时间内没有接受到消息*/
            .
        }
        .
        .
     }
}
```

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，发生了一次任务切换。

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：

- OS_NO_ERR ：消息成功的放到消息邮箱中。
- OS_MBOX_FULL ：消息邮箱已经包含了其他消息，不空。
- 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。

不允许传递一个空指针，因为这意味着消息邮箱为空。

范例：

```
OS_EVENT *CommMbox;
INT8U    CommRxBuf[100];



void CommTaskRx(void *pdata)
{
     INT8U  err;



     pdata = pdata;
     for (;;) {
        .
        .
         err = OSMboxPost(CommMbox, (void *)&CommRxBuf[0]);
        .
        .
     }
}
```

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（）函数可以知道任务是否在等待消息以及有多少个任务在等待消息，还可以检查消息邮箱现在的消息。

参数

pevent　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：

```
Void　*OSMsg;　　/*  消息邮箱中消息的复制  */
INT8U　OSEventTbl[OS_EVENT_TBL_SIZE];　　/*消息邮箱等待队列的复制*/
INT8U OSEventGrp；
```

返回值

OSMboxQuery（）函数的返回值为下述之一：

- OS_NO_ERR　：调用成功
- OS_ERR_EVENT_TYPE　：pevent 不是指向消息邮箱的指针。

注意/警告

必须先建立消息邮箱，然后使用。

范例：

```
OS_EVENT *CommMbox;



void Task (void *pdata)
{
     OS_MBOXDATA mbox_data;
     INT8U      err;



     pdata = pdata;
     for (;;) {
        .
        .
        err = OSMboxQuery(CommMbox, &mbox_data);
        if (err == OS_NO_ERR) {
          .   /* 如果mbox_data.OSMsg为非空指针，说明消息邮箱非空*/
        }
        .
        .
     }
}
```

OSMemCreate( )

OS_MEM *OSMemCreate(void *addr, INT32U nblks ,INT32U blksize, INT8U *err);

所属文件	调用者	开关量
OS_MEM.C	任务或初始代码	OS_/MEM_EN

OSMemCreate（）函数建立并初始化一块内存区。一块内存区包含指定数目的大小确定的内存块。程序可以包含这些内存块并在用完后释放回内存区。

参数

addr 建立的内存区的起始地址。内存区可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_MEM_INVALID_PART ：没有空闲的内存区。

OS_MEM_INVALID_BLKS ：没有为每一个内存区建立至少两个内存块。

OS_MEM_INVALID_SIZE ：内存块大小不足以容纳一个指针变量。

返回值

OSMemCreate（）函数返回指向内存区控制块的指针。如果没有剩余内存区，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：

```
OS_MEM *CommMem;
INT8U   CommBuf[16][128];



void main(void)
{
     INT8U err;



     OSInit();                    /* 初始化μC/OS-Ⅱ       */
     .
     .
     CommMem = OSMemCreate(&CommBuf[0][0], 16, 128, &err);
     .
     .
     OSStart();                   /* 启动多任务内核        */
}
```

OSMemGet( )

Void *OSMemGet(OS_MEM *pmem, INT8U *err);

所属文件	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。

范例：

```
OS_MEM *CommMem;



void Task (void *pdata)
{
     INT8U *msg;



     pdata = pdata;
     for (;;) {
        msg = OSMemGet(CommMem, &err);
        if (msg != (INT8U *)0) {
           .                        /* 内存块已经分配 */
           .
        }
        .
        .
     }
}
```

OSMemPut( )

INT8U OSMemPut(OS_MEM *pmem, void *pblk);

所属文件	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。

Pblk 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，释放了多于用 OSMemGet（）函数得到的内存块。

注意/警告

必须首先建立内存区，然后使用。

内存块必须释放回原先申请的内存区。

范例：

```
OS_MEM *CommMem;
INT8U  *CommMsg;



void Task (void *pdata)
{
     INT8U err;



     pdata = pdata;
     for (;;) {
        err = OSMemPut(CommMem, (void *)CommMsg);
        if (err == OS_NO_ERR) {
           .                            /* 释放内存块     */
           .
        }
        .
        .
     }
}
```

OSMemQuery( )

INT8U OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata);

所属文件	调用者	开关量
OS_MEM.C	任务或中断	OS_MEM_EN

OSMemQuery（）函数得到内存区的信息。该函数返回 OS_MEM 结构包含的信息，但使用了一个新的 OS_MEM_DATA 的数据结构。OS_MEM_DATA 数据结构还包含了正被使用的内存块数目的域。

参数

pmem 是指向内存区控制块的指针，可以从 OSMemCreate（）函数 返回得到。

Pdata 是指向 OS_MEM_DATA 数据结构的指针，该数据结构包含了以下的域：

```
Void     OSAddr;      /*指向内存区起始地址的指针                */
Void     OSFreeList;  /*指向空闲内存块列表起始地址的指针        */
INT32U   OSBlkSize;   /*每个内存块的大小                        */
INT32U   OSNBlks;     /*该内存区的内存块总数                    */
INT32U   OSNFree;     /*空闲的内存块数目                        */
INT32U   OSNUsed;     /*使用的内存块数目                        */
```

返回值

OSMemQuery（）函数返回值总是 OS_NO_ERR。

注意/警告

必须首先建立内存区，然后使用。

范例：

```
OS_MEM     *CommMem;



void Task (void *pdata)
{
     INT8U      err;
     OS_MEM_DATA  mem_data;



     pdata = pdata;
     for (;;) {
        .
        .
        err = OSMemQuery(CommMem, &mem_data);
        .
        .
     }
}
```

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

pevent 是指向需要查看的消息队列的指针。当建立消息队列时，该指针返回到用户程序。(参考 OSMboxCreate（）函数)。

返回值

如果消息已经到达，返回指向该消息的指针；如果消息队列没有消息，返回空指针。

注意/警告

必须先建立消息队列，然后使用。

范例：

```
OS_EVENT *CommQ;



void Task (void *pdata)
{
     void *msg;



     pdata = pdata;
     for (;;) {
       msg = OSQAccept(CommQ);     /* 检查消息队列  */
       if (msg != (void *)0) {
          .                      /* 处理接受的消息  */
          .
       } else {
          .                      /* 没有消息 */
          .
       }
       .
       .
     }
}
```

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。

参数

start 是消息内存区的基地址，消息内存区是一个指针数组。

Size 是消息内存区的大小。

返回值

OSQCreate（）函数返回一个指向消息队列事件控制块的指针。如果没有空余的事件空闲块，OSQCreate（）函数返回空指针。

注意/警告

必须先建立消息队列，然后使用。

范例：

```
OS_EVENT *CommQ;
void    *CommMsg[10];



void main(void)
{
      OSInit();                                /* 初始化μC/OS-Ⅱ   */
      .
      .
      CommQ = OSQCreate(&CommMsg[0], 10);       /*建立消息队列      */
      .
      .
      OSStart();                               /* 启动多任务内核   */
}
```

OSQFlush( )

INT8U *SOQFlush（OS_EVENT *pevent）；

所属文件	调用者	开关量
OS_Q.C	任务或中断	OS_Q_EN

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。

参数

pevent 是指向消息队列的指针。该指针的值在建立该队列时可以得到。(参考 OSQCreate（）函数)。

返回值

OSQFlush（）函数的返回值为下述之一：

- OS_NO_ERR ：消息队列被成功清空
- OS_ERR_EVENT_TYPE ：试图清除不是消息队列的对象

注意/警告

必须先建立消息队列，然后使用。

范例：

```
OS_EVENT *CommQ;



void main(void)
{
      INT8U err;

      OSInit();                        /* 初始化μC/OS-Ⅱ */
      .
      .
      err = OSQFlush(CommQ);
      .
      .
      OSStart();                       /* 启动多任务内核   */
}
```

OSQPend( )

Void *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

所属文件	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。

范例：

```
OS_EVENT *CommQ;



void CommTask(void *data)
{
     INT8U  err;
     void  *msg;



     pdata = pdata;
     for (;;) {
        .
        .
        msg = OSQPend(CommQ, 100, &err);
        if (err == OS_NO_ERR) {
           .
           .                /* 在指定时间内接受到消息      */
           .
        } else {
           .
           .               /* 在指定的时间内没有接受到指定的消息  */
           .
        }
        .
        .
     }
}
```

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。

注意/警告

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：

```
OS_EVENT *CommQ;
INT8U    CommRxBuf[100];



void CommTaskRx(void *pdata)
{
     INT8U  err;



     pdata = pdata;
     for (;;) {
        .
        .
        err = OSQPost(CommQ, (void *)&CommRxBuf[0]);
        if (err == OS_NO_ERR) {
           .                    /* 将消息放入消息队列   */
           .
        } else {
           .                    /* 消息队列已满               */
           .
        }
        .
        .
     }
}
```

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。

注意/警告

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：

```
OS_EVENT *CommQ;
INT8U    CommRxBuf[100];



void CommTaskRx(void *pdata)
{
     INT8U  err;



     pdata = pdata;
     for (;;) {
        .
        .
        err = OSQPostFront(CommQ, (void *)&CommRxBuf[0]);
        if (err == OS_NO_ERR) {
           .                    /* 将消息放入消息队列  */
           .
        } else {
           .                    /* 消息队列已满 */
           .
        }
        .
        .
     }
}
```

OSQQuery( )

INT8U OSQQuery(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

OSQQuery（）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 OSQQuery（）函数可以知道任务是否在等待消息、有多少个任务在等待消息、队列中有多少消息以及消息队列可以容纳的消息数。OSQQuery（）函数还可以得到即将被传递给任务的消息的信息。

参数

pevent 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：

```
Void        *OSMsg;                          /* 下一个可用的消息*/
INT16U      OSNMsgs；                        /*  队列中的消息数目*/
INT16U      OSQSize；                        /*  消息队列的大小  */
INT8U       OSEventTbl[OS_EVENT_TBL_SIZE];   /*  消息队列的等待队列*/
INT8U       OSEventGrp；
```

返回值

OSQQuery（）函数的返回值为下述之一：

- OS_NO_ERR ：调用成功
- OS_ERR_EVENT_TYPE ：pevent 不是指向消息队列的指针。

注意/警告

必须先建立消息队列，然后使用。

范例：

```
OS_EVENT *CommQ;



void Task (void *pdata)
{
     OS_Q_DATA qdata;
     INT8U    err;



     pdata = pdata;
     for (;;) {
        .
        .
        err = OSQQuery(CommQ, &qdata);
        if (err == OS_NO_ERR) {
          .   /* 取得消息队列的信息 */
        }
        .
        .
     }
}
```

OSSchedLock( )

Void OSSchedLock(void);

所属文件	调用者	开关量
OS_CORE.C	任务或中断	N/A

OSSchedLock（）函数停止任务调度，只有使用配对的函数 OSSchedUnlock（）才能重新开始内核的任务调度。调用 OSSchedLock（）函数的任务独占 CPU，不管有没有其他高优先级的就绪任务。在这种情况下，中断仍然可以被接受和执行（中断必须允许）。OSSchedLock（）函数和 OSSchedUnlock（）函数必须配对使用。μC/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：

```
void TaskX(void *pdata)
{
     pdata = pdata;
     for (;;) {
        .
        OSSchedLock();        /* 停止任务调度      */
        .
        .                   /* 不允许被打断的执行代码 */
        .
        OSSchedUnlock();      /* 恢复任务调度   */
        .
     }
}
```

OSSchedUnlock( )

Void OSSchedUnlock(void);

所属文件	调用者	开关量
OS_CORE.C	任务或中断	N/A

在调用了 OSSchedLock（）函数后，OSSchedUnlock（）函数恢复任务调度。

参数

无

返回值

无

注意/警告

任务调用了 OSSchedLock（）函数后，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：

```
void TaskX(void *pdata)
{
     pdata = pdata;
     for (;;) {
        .
        OSSchedLock();        /* 停止任务调度       */
        .
        .                   /* 不允许被打断的执行代码 */
        .
        OSSchedUnlock();      /* 恢复任务调度   */
        .
     }
}
```

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。不同于 OSSemPend（）函数，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。

参数

pevent 是指向需要查询的设备的信号量。当建立信号量时，该指针返回到用户程序。（参考 OSSemCreate（）函数）。

返回值

当调用 OSSemAccept（）函数时，设备信号量的值大于零，说明设备就绪，这个值被返回调用者，设备信号量的值减一。如果调用 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：

```
OS_EVENT *DispSem;



void Task (void *pdata)
{
     INT16U value;



     pdata = pdata;
     for (;;) {
        value = OSSemAccept(DispSem);  /*查看设备是否就绪或事件是否发生 */
        if (value > 0) {
           .                              /* 就绪，执行处理代码 */
           .
        }
        .
        .
     }
}
```

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

OSSemCreate（）函数建立并初始化一个信号量。信号量的作用如下：

- 允许一个任务和其他任务或者中断同步。
- 取得设备的使用权
- 标志事件的发生

参数

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：

```
OS_EVENT *DispSem;



void main(void)
{
     .
     .
     OSInit();                          /* 初始化μC/OS-Ⅱ        */
     .
     .
     DispSem = OSSemCreate(1);            /* 建立显示设备的信号量      */
     .
     .
     OSStart();                         /* 启动多任务内核 */
}
```

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的场合。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

注意/警告

必须先建立信号量，然后使用。
不允许从中断调用该函数。

范例：

```
OS_EVENT *DispSem;



void DispTask(void *pdata)
{
     INT8U  err;



     pdata = pdata;
     for (;;) {
        .
        .
        OSSemPend(DispSem, 0, &err);
        .                    /* 只有信号量置起，该任务才能执行 */
        .
     }
}
```

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_SEM_OVF ：信号量的值溢出
- OS_ERR_EVENT_TYPE ：pevent 不是指向信号量的指针。

注意/警告

必须先建立信号量，然后使用。

范例：

```
OS_EVENT *DispSem;



void TaskX(void *pdata)
{
     INT8U  err;



     pdata = pdata;
     for (;;) {
        .
        .
        err = OSSemPost(DispSem);
        if (err == OS_NO_ERR) {
           .                        /* 信号量置起  */
           .
        } else {
           .                        /* 信号量溢出 */
           .
        }
        .
        .
     }
}
```

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 OSSemQuery（）可以得知是否有，以及有多少任务位于信号量的任务等待队列中（通过查询.OSEventTbl [ ]域），还可以获取信号量的标识号码。OSEventTbl [ ]域的大小由语句：#define constant OS_ENENT_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 OSSemCreat（）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：

```
INT16U OSCnt;                                /* 当前信号量标识号码 */
INT8U  OSEventTbl[OS_EVENT_TBL_SIZE];        /*信号量等待队列*/
INT8U  OSEventGrp;
```

返回值

OSSemQuery（）函数有下述两个返回值：

- 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。

```
OS_EVENT *DispSem;



void Task (void *pdata)
{
     OS_SEM_DATA sem_data;
     INT8U      err;
     INT8U      highest;  /* 在信号量中等待的优先级最高的任务 */
     INT8U      x;
     INT8U      y;



     pdata = pdata;
     for (;;) {
        .
        .
        err = OSSemQuery(DispSem, &sem_data);
        if (err == OS_NO_ERR) {
           if (sem_data.OSEventGrp != 0x00) {
              y      = OSUnMapTbl[sem_data.OSEventGrp];
              x      = OSUnMapTbl[sem_data.OSEventTbl[y]];
              highest = (y << 3) + x;
              .
              .
           }
        }
        .
        .
     }
}
```

OSStart ( )

void OSStart(void);

所属文件	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 才C/OS-II  */
   .                              /* 用户代码           */
   .
   OSStart();                      /* 启动多任务环境   */
}
```

OSStatInit ( )

void OSStatInit (void);

所属文件	调用者	开关量
OS_CORE.C	只能是初始化代码	OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_EN

OSStatInit（）获取当系统中没有其他任务运行时，32 位计数器所能达到的最大值。OSStatInit（）的调用时机是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：

```
void FirstAndOnlyTask (void *pdata)
{
     .
     .
     OSStatInit();          /* 计算CPU使用率 */
     .
     OSTaskCreate(/ );       /* 建立其他任务 */
     OSTaskCreate(/ );
     .
     for (;;) {
        .
        .
     }
}
```

OSTaskChangPrio( )

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。

参数

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_PRO_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（）中还会先判断要改变优先级的任务是否存在。

范例：

```
void TaskX(void *data)
{
     INT8U  err;



     for (;;) {
        .
        .
        err = OSTaskChangePrio(10, 15);
        .
        .
     }
}
```

OSTaskCreate( )

INT8U OSTaskCreate(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio);

所属文件	调用者	开关量
OS_TASK.C	任务或初始化代码	无

OSTaskCreate（）建立一个新任务。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。中断处理程序中不能建立任务。一个任务必须为无限循环结构（如下所示），且不能有返回点。

OSTaskCreate（）是为与先前的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：

```
void Task (void *pdata)
{
     .                 /* Do something with 'pdata'                  */
     for (;;) {          /* 任务函数体.         */
        .
        .
        /* 在任务体中必须调用如下函数之一:                      */
        /*    OSMboxPend()                                          */
        /*    OSQPend()                                            */
        /*    OSSemPend()                                           */
        /*    OSTimeDly()                                           */
        /*    OSTimeDlyHMSM()                                        */
        /*    OSTaskSuspend()    (挂起任务本身)                      */
        /*    OSTaskDel()       (删除任务本身)                       */
        .
        .
     }
```

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。

返回值

OSTaskCreate（）的返回值为下述之一：

- OS_NO_ERR：函数调用成功。
- OS_PRIO_EXIST：具有该优先级的任务已经存在。
- OS_PRIO_INVALID：参数指定的优先级大于 OS_LOWEST_PRIO。
- OS_NO_MORE_TCB：系统中没有 OS_TCB 可以分配给任务了。

注意/警告

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]。如果在您的程序中设

定堆栈向高地址增长，则传递的栈顶指针应该为&Task1Stk [0]。

```
OS_STK  Task1Stk[1024];

void main(void)
{
     INT8U err;


     .
     OSInit();              /* 初始化 μC/OS-II           */
     .
     OSTaskCreate(Task1,
                (void *)0,
                &Task1Stk[1023],
                25);
     .
     OSStart();             /* 启动多任务环境            */
}



void Task1(void *pdata)
{
     pdata = pdata;
     for (;;) {
        .                  /* 任务代码                  */
        .
     }
}
```

范例 2：

您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讯口的函数。传递不同的初始化数据（端口地址、波特率）和指定不同的通讯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();                  /* 初始化才 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)     /* 通讯任务             */
{
     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，但中断处理程序中不能建立新任务。一个任务必须为无限循环结构（如下所示），且不能有返回点。

参数

task 是指向任务代码的指针。

Pdata 指针指向一个数据结构，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 中的任务代码结构以及如何传递参数 pdata：(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；----译者注)

```
void Task (void *pdata)
{
     .                  /* 对参数pdata进行操作，例如pdata= pdata   */
     for (;;) {          /* 任务函数体.总是为无限循环结构         */
        .
        .
        /* 任务中必须调用如下的函数:                        */
        /*    OSMboxPend()                                    */
        /*    OSQPend()                                      */
        /*    OSSemPend()                                     */
        /*    OSTimeDly()                                     */
        /*    OSTimeDlyHMSM()                                  */
        /*    OSTaskSuspend()    (挂起任务自身)                 */
        /*    OSTaskDel()       (删除任务自身)                  */
        .
        .
     }
}
```

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，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唯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** 的单位为分别为字节（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：

本例中使用了一个用户自定义的数据结构 TASK_USER_DATA [标识（1）]，在其中保存了任务名称和其他一些数据。任务名称可以用标准库函数 strcpy（）初始化 [标识（2）]。在本例中，允许堆栈检查操作 [标识（4）]，程序可以调用 OSTaskStkChk（）函数。本例中设定堆栈向低地址方向增长 [标识（3）]。本例中 OS_STK_GROWTH 设为 1。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）。

```
typedef struct {                  /*  用户定义的数据结构                (1)*/
    char    TaskName[20];
    INT16U  TaskCtr;
    INT16U  TaskExecTime;
    INT32U  TaskTotExecTime;
} TASK_USER_DATA;



OS_STK          TaskStk[1024];
TASK_USER_DATA   TaskUserData;



void main(void)
{
     INT8U err;


     .
     OSInit();                          /* 初始化ォC/OS-II                */
     .
     strcpy(TaskUserData.TaskName, "MyTaskName");  /*  任务名         (2)*/
     err = OSTaskCreateExt(Task,
            (void *)0,
            &TaskStk[1023],             /*  堆栈向低地址增长 (TOS)     (3)*/

            10,
            &TaskStk[0],                /*  堆栈向低地址增长 (BOS)     (3)*/
            1024,
```

```
            (void *)&TaskUserData,        /* TCB 的扩展                    */
            OS_TASK_OPT_STK_CHK);         /*  允许堆栈检查              (4)*/
    .
    OSStart();                           /* 启动多任务环境                 */
}



void Task(void *pdata)
{
    pdata = pdata;              /* 此句可避免编译中的警告信息              */
    for (;;) {
      .                        /* 任务代码                                */
      .
    }
}
```

范例 2：

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[标识（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 [标识（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 ォC/OS-II              */
     .
     err = OSTaskCreateExt(Task,
            (void *)0,
            &TaskStk[0],                /*  堆栈向高地址增长 (TOS)     (1)*/
            10,
            10,
            &TaskStk[1023],             /*  堆栈向高地址增长 (BOS)    (1)*/
            1024,
            (void *)0,
            OS_TASK_OPT_STK_CHK);       /*  允许堆栈检查              (2)*/
     .
     OSStart();                         /* 启动多任务环境                */
}



void Task(void *pdata)
{
     pdata = pdata;            /* 此句可避免编译中出现警告信息             */
     for (;;) {
        .                     /* 任务代码                                 */
        .
     }
}
```

OSTaskDel( )

INT8U OSTaskDel (INT8U prio);

所属文件	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（Idle task）。
- 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。

注意/警告

OSTaskDel（）将判断用户是否试图删除μC/OS 中的空闲任务（Idle task）。

在删除占用系统资源的任务时要小心，此时，为安全起见可以用另一个函数 OSTaskDelReq（）。

范例：

```
void TaskX(void *pdata)
{
```

```
    INT8U err;



    for (;;) {
       .
       .
       err = OSTaskDel(10);     /* 删除优先级为10的任务                    */
       if (err == OS_NO_ERR) {
          .                     /* 任务被删除                              */
          .
       }
       .
       .
    }
}
```

OSTaskDelReq( )

INT8U 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

OSTaskDelReq（）的返回值为下述之一：

- OS_NO_ERR：删除请求已经被任务记录。
- OS_TASK_NOT_EXIST：指定的任务不存。发送删除请求的任务可以等待此返回值，看删除是否成功。
- OS_TASK_DEL_IDLE：错误操作，试图删除空闲任务（Idle task）。
- OS_PRIO_INVALID ：参数指定的优先级大于 OS_LOWEST_PRIO 或没有设定 OS_PRIO_SELF 的值。
- OS_TASK_DEL_REQ：当前任务收到来自其他任务的删除请求。

注意/警告

OSTaskDelReq（）将判断用户是否试图删除μC/OS 中的空闲任务（Idle task）。

范例：

```
void TaskThatDeletes(void *pdata)       /* 任务优先级 5                  */
{
```

```
    INT8U err;



    for (;;) {
       .
       .
       err = OSTaskDelReq(10);    /* 请求任务#10删除自身                  */
       if (err == OS_NO_ERR) {
          err = OSTaskDelReq(10);
          while (err != OS_TASK_NOT_EXIST) {
             OSTimeDly(1);        /* 等待任务删除                        */
          }
          .                     /* 任务#10已被删除                      */
       }
       .
       .
    }
}



void TaskToBeDeleted(void *pdata)       /* 任务优先级 10                 */
{
   .
   .
   pdata = pdata;
   for (;;) {
      OSTimeDly(1);
      if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ) {
         /* 释放任务占用的系统资源                                        */
         /* 释放动态分配的内存                                            */
         OSTaskDel(OS_PRIO_SELF);
      }
   }
}
```

OSTaskQuery( )

INT8U OSTaskQuery (INT8U prio, OS_TCB *pdata);

所属文件	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。

注意/警告

任务控制块（TCB）中所包含的数据成员取决于下述开关量在初始化时的设定（参见 OS_CFG.H）

- OS_TASK_CREATE_EN
- OS_Q_EN
- OS_MBOX_EN
- OS_SEM_EN
- OS_TASK_DEL_EN

范例：

```
void Task (void *pdata)
```

```
{
     OS_TCB  task_data;
     INT8U   err;
     void   *pext;
     INT8U   status;



     pdata = pdata;
     for (;;) {
        .
        .
        err = OSTaskQuery(OS_PRIO_SELF, &task_data);
        if (err == OS_NO_ERR) {
           pext   = task_data.OSTCBExtPtr; /* 获取TCB扩展数据结构的指针   */
           status = task_data.OSTCBStat;   /* 获取任务状态           */
           .
           .
        }
        .
        .
     }
  }
```

OSTaskResume( )

INT8U OSTaskResume (INT8U prio);

所属文件	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（）也是唯一能“ 解挂 ”挂起任务的函数。

参数

prio 指定要唤醒任务的优先级。

返回值

OSTaskResume （）的返回值为下述之一：

- OS_NO_ERR：函数调用成功。
- OS_TASK_RESUME_PRIO：要唤醒的任务不存在。
- OS_TASK_NOT_SUSPENDED：要唤醒的任务不在挂起状态。
- OS_PRIO_INVALID：参数指定的优先级大于或等于 OS_LOWEST_PRIO。

注意/警告

无

范例：

```
void TaskX(void *pdata)
```

```
{
     INT8U err;



     for (;;) {
        .
        .
        err = OSTaskResume(10);       /* 唤醒优先级为10的任务    */
        if (err == OS_NO_ERR) {
           .                         /* 任务被唤醒              */
           .
        }
        .
        .
     }
}
```

OSTaskStkChk( )

INT8U OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata);

所属文件	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** 参数中OS_TASK_OPT_STK_CHK 操作项打开。

计算堆栈未用空间的方法是从堆栈底端向顶端逐个字节比较，检查堆栈中 0 的个数，直到一个非 0 的数值出现。这种方法的前提是堆栈建立时已经全部清零。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：

```
INT32U OSFree;       /* 堆栈中未使用的字节数          */
INT32U OSUsed;       /* 堆栈中已使用的字节数          */
```

返回值

OSTaskStkChk（）的返回值为下述之一：

- OS_NO_ERR：函数调用成功。
- OS_PRIO_INVALID ：参数指定的优先级大于 OS_LOWEST_PRIO，或未指定OS_PRIO_SELF。
- OS_TASK_NOT_EXIST：指定的任务不存在。
- OS_TASK_OPT_ERR ：任务用 OSTaskCreateExt （） 函数建立的时候没有指定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。

范例：

```
void Task (void *pdata)
{
     OS_STK_DATA stk_data;
     INT32U     stk_size;



     for (;;) {
        .
        .
        err = OSTaskStkChk(10, &stk_data);
        if (err == OS_NO_ERR) {
           stk_size = stk_data.OSFree + stk_data.OSUsed;
        }
        .
        .
     }
}
```

OSTaskSuspend( )

INT8U OSTaskSuspend (INT8U prio);

所属文件	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。
- OS_PRIO_INVALID ：参数指定的优先级大于 OS_LOWEST_PRIO 或没有设定 OS_PRIO_SELF 的值。
- OS_TASK_ SUSPEND _PRIO：要挂起的任务不存在。

注意/警告

在程序中 OSTaskSuspend（）和 OSTaskResume （）应该成对使用。

用 OSTaskSuspend（）挂起的任务只能用 OSTaskResume （）唤醒。

范例：

```
void TaskX(void *pdata)
{
     INT8U err;



     for (;;) {
        .
        .
        err = OSTaskSuspend(OS_PRIO_SELF);     /* 挂起当前任务     */
        .                  /* 当其他任务唤醒被挂起任务时，任务可继续运行 */
        .
        .
     }
}
```

OSTimeDly( )

void OSTimeDly (INT16U ticks);

所属文件	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

ticks 为要延时的时钟节拍数。

返回值

无

注意/警告

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：

```
void TaskX(void *pdata)
{
     for (;;) {
        .
```

```
        .
        OSTimeDly(10);              /* 任务延时10个时钟节拍 */
        .
        .
    }
}
```

OSTimeDlyHMSM( )

void OSTimeDlyHMSM(INT8U hours，INT8U minutes，INT8U seconds，INT8U milli);

所属文件	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

OSTimeDlyHMSM（）的返回值为下述之一：

- OS_NO_ERR：函数调用成功。
- OS_TIME_INVALID_MINUTES：参数错误，分钟数大于 59。
- OS_TIME_INVALID_SECONDS：参数错误，秒数大于 59。
- OS_TIME_INVALID_MILLI：参数错误，毫秒数大于 999。
- OS_TIME_ZERO_DLY：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：

```
void TaskX(void *pdata)
{
     for (;;) {
        .
```

```
        .
        OSTimeDlyHMSM(0, 0, 1, 0);  /* 任务延时 1 秒 */
        .
        .
    }
}
```

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：

- OS_NO_ERR：函数调用成功。
- OS_PRIO_INVALID：参数指定的优先级大于 OS_LOWEST_PRIO。
- OS_TIME_NOT_DLY：要唤醒的任务不在延时状态。
- OS_TASK_NOT_EXIST：指定的任务不存在。

注意/警告

用户不应该用 OSTimeDlyResume（）去唤醒一个设置了等待超时操作，并且正在等待事件发生的任务。操作的结果是使该任务结束等待，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。
（OSTimeDlyHMSM（0，10，55，350）共延时　[10 minutes *60 + （55+0.35）seconds] *100 =65,535 次时钟节拍------译者注）

范例：

```
void TaskX(void *pdata)
{
   INT8U err;
```

```
   pdata = pdata;
   for (;;) {
      .
      err = OSTimeDlyResume(10);        /* 唤醒优先级为10的任务     */
      if (err == OS_NO_ERR) {
         .                             /* 任务被唤醒               */
         .
      }
      .
   }
}
```

OSTimeGet( )

INT32U OSTimeGet (void);

所属文件	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，记录系统上电后或时钟重新设置后的时钟计数。

参数

无。

返回值

当前时钟计数（时钟节拍数）。

注意/警告

无

范例：

```
void TaskX(void *pdata)
{
     INT32U clk;



     for (;;) {
        .
        .
        clk = OSTimeGet();  /* 获取当前系统时钟的值 */
        .
        .
     }
}
```

OSTimeSet( )

void OSTimeSet (INT32U ticks);

所属文件	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

ticks 要设置的时钟数，单位是时钟节拍数。

返回值

无。

注意/警告

无

范例：

```
void TaskX(void *pdata)
{
     for (;;) {
        .
        .
        OSTimeSet(0L);    /* 复位系统时钟  */
        .
        .
     }
}
```

OSTimeTick( )

void OSTimeTick (void);

所属文件	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

OSTimeTick（）的运行时间和系统中的任务数直接相关，在任务或中断中都可以调用。如果在任务中调用，任务的优先级应该很高（优先级数字很小），这是因为 OSTimeTick（）负责所有任务的延时操作。

范例：

（Intel　８０ｘ８６，实模式）

```
TickISRPROC   FAR
              PUSHA                        ; 保存CPU寄存器内容
              PUSH ES
              PUSH DS
;
```

```
                INC  BYTE PTR _OSIntNesting   ; 标识C/OS-II进入中断处理程序
                CALL FAR PTR _OSTimeTick      ; 调用时钟节拍处理函数
                .                           ; 用户代码清除中断标志
                .
                CALL FAR PTR _OSIntExit       ; 标识C/OS-II退出中断处理程序
                POP  DS                     ; 恢复CPU寄存器内容
                POP  ES
                POPA
;
                IRET                        ; 中断返回
TickISRENDP
```

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。

注意/警告

无

范例：

```
void TaskX(void *pdata)
{
     INT16U os_version;



     for (;;) {
        .
        .
        os_version = OSVersion();  /* 获取 uC/OS-II's 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 (;;) {
        .
        .
        OS_ENTER_CRITICAL();    /* 关闭中断     */
        .
        .                     /* 进入核心代码   */
        .
        OS_EXIT_CRITICAL();     /* 打开中断     */
        .
        .
     }
}
```

第 12 章

配置手册

本章将介绍µC/OS-II 中的初始化配置项。由于µC/OS-II 向用户提供源代码，初始化配置项由一系列#define constant 语句构成，都在文件 OS_CFG.H 中。用户的工程文件组中都应该包含这个文件。

本节介绍每个用#define constant 定义的常量，介绍的顺序和它们在 OS_CFG.H 中出现的顺序是相同的。表 12.1 列出了常量控制的µC/OS-II 函数。“ 类型 ”为函数所属的类型，“ 置 1 ”表示当定义常量为 1 时可以打开相应的函数，“ 其他常量 ”为与这个函数有关的其他控制常量。

注意编译工程文件时要包含 OS_CFG.H，使定义的常量生效。

表 T12.1 µC/OS-II 函数和相关的常量（#define constant 定义）

表 T12.1 µC/OS-II 函数和相关常量

类型	*置1*	*其他常量*
杂相		
OSInit()	无	OS_MAX_EVENTS OS_Q_EN and OS_MAX_QS OS_MEM_EN OS_TASK_IDLE_STK_SIZ E OS_TASK_STAT_EN OS_TASK_STAT_STK_SIZ E
OSSchedLock()	无	无
OSSchedUnlock()	无	无
OSStart()	无	无
OSStatInit()	OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_E N	OS_TICKS_PER_SEC
OSVersion()	无	无

中断处理		
OSIntEnter()	无	无
OSIntExit()	无	无
消息邮箱		
OSMboxAccept()	OS_MBOX_EN	无
OSMboxCreate()	OS_MBOX_EN	OS_MAX_EVENTS
OSMboxPend()	OS_MBOX_EN	无
OSMboxPost()	OS_MBOX_EN	无
OSMboxQuery()	OS_MBOX_EN	无
内存块管理		
OSMemCreate()	OS_MEM_EN	OS_MAX_MEM_PART
OSMemGet()	OS_MEM_EN	无
OSMemPut()	OS_MEM_EN	无
OSMemQuery()	OS_MEM_EN	无
消息队列		
OSQAccept()	OS_Q_EN	无
OSQCreate()	OS_Q_EN	OS_MAX_EVENTS OS_MAX_QS
OSQFlush()	OS_Q_EN	无
OSQPend()	OS_Q_EN	无
OSQPost()	OS_Q_EN	无
OSQPostFront()	OS_Q_EN	无
OSQQuery()	OS_Q_EN	无
信号量管理		
OSSemAccept()	OS_SEM_EN	无
OSSemCreate()	OS_SEM_EN	OS_MAX_EVENTS
OSSemPend()	OS_SEM_EN	无
OSSemPost()	OS_SEM_EN	无
OSSemQuery()	OS_SEM_EN	无
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()	无	无
OSTimeDlyHMSM()	无	OS_TICKS_PER_SEC
OSTimeDlyResume()	无	OS_LOWEST_PRIO
OSTimeGet()	无	无
OSTimeSet()	无	无
OSTimeTick()	无	无
用户定义函数		
OSTaskCreateHook()	OS_CPU_HOOKS_EN	无
OSTaskDelHook()	OS_CPU_HOOKS_EN	无
OSTaskStatHook()	OS_CPU_HOOKS_EN	无
OSTaskSwHook()	OS_CPU_HOOKS_EN	无
OSTimeTickHook()	OS_CPU_HOOKS_EN	无

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（Idle task）(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（Idle task）堆栈的容量。注意堆栈容量的单位不是字节，而是 OS_STK（µC/OS-II 中堆栈统一用 OS_STK 声明，根据不同的硬件环境，OS_STK 可为不同的长度----译者注）。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。

OS_TASK_STAT_STK_SIZE

OS_TASK_STAT_STK_SIZE 设置µC/OS-II 中统计任务（statistic task）堆栈的容量。注意单位不是字节，而是 OS_STK（µC/OS-II 中堆栈统一用 OS_STK 声明，根据不同的硬件环境，OS_STK 可为不同的长度----译者注）。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。

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（即系统硬件定时器中断发生的频率----译者注）。在函数 OSStatInit（），OSTaskStat（）和 OSTimeDlyHMSM（）中都会用到 OS_TICKS_PER_SEC。