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 L1.2 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();//保存DOS环境                                      (3)
    PC_VectSet(uCOS, OSCtxSw);//cpu寄存器切换                             (4)
    RandomSem = OSSemCreate(1);//生成信号量避免重入                       (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);//将调用PC_DOSSaveReturn
(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 L1.8 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();关中断
    PC_VectSet(0x08, OSTickISR); 让其指向μC/OS-II的时钟节拍处理          (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);                将自身延时两个时钟节拍  与时钟节拍同步          (1)
    OS_ENTER_CRITICAL();
    OSIdleCtr    = 0L;            将 一个32位的计数器清零                      (2)
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC); 产生另一个延时                               (3)
    OS_ENTER_CRITICAL();           使OSStatInit()挂起，不断的递增OSIdleCtr
    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。

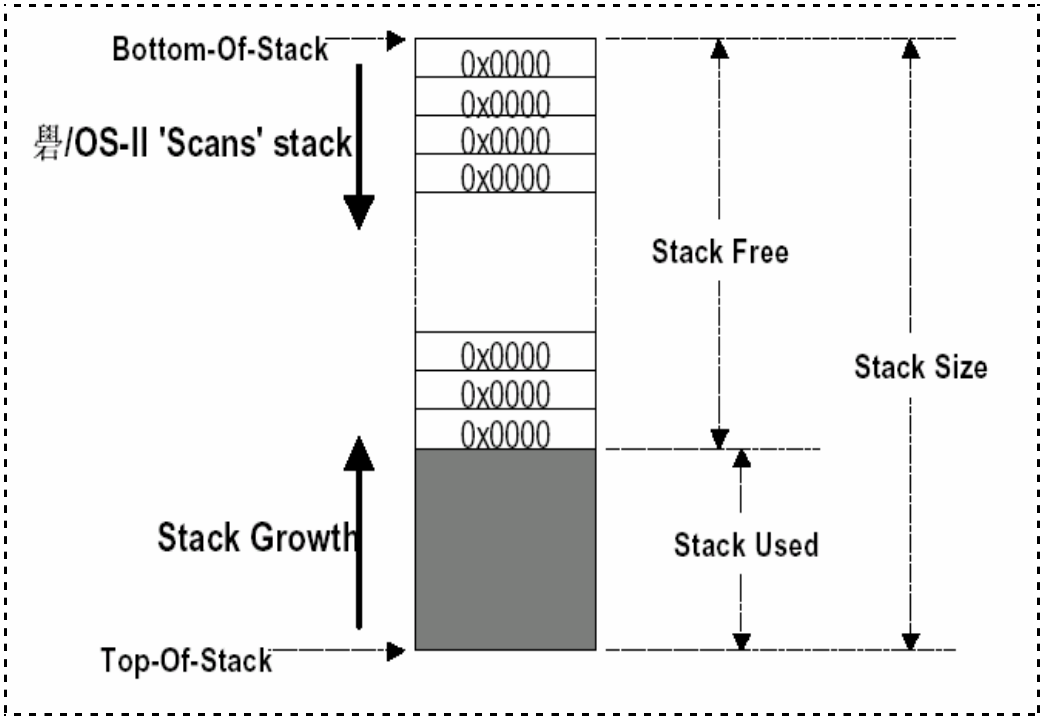
uCOS-II 的堆栈检查功能要求任务建立时堆栈清零。OSTaskCreateExt()可以执行此项操作（设置选项 OS_TASK_OPT_STK_CHK 和 OS_TASK_OPT_STK_CLR 打开此项操作）。如果任务运行过程中要进行建立、删除任务的操作，应该设置好上述的选项，确保任务建立后堆栈是清空的。同时要意识到 OSTaskCreateExt()进行堆栈清零操作是一项很费时的工作，而且取决于

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 L1.12 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;
      }
   }
}
```