# 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.**

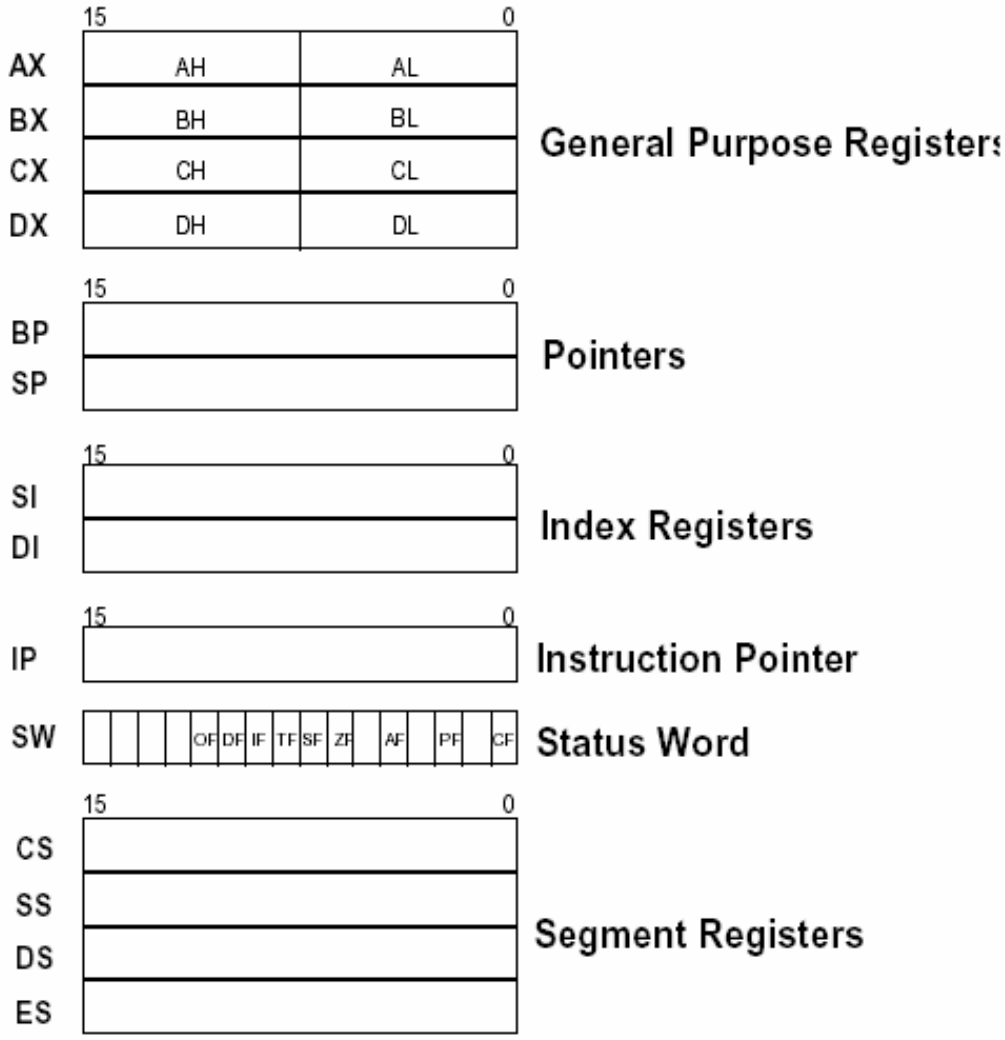


**Figure 9-1, 80x86 Real-Mode Register Map**

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.

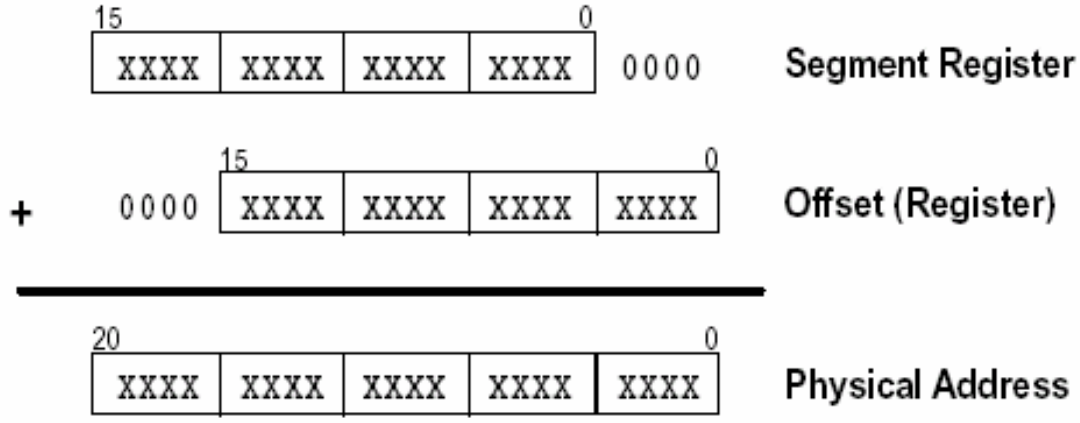


**Figure 9-2, Addressing with a Segment and an Offset**

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.**

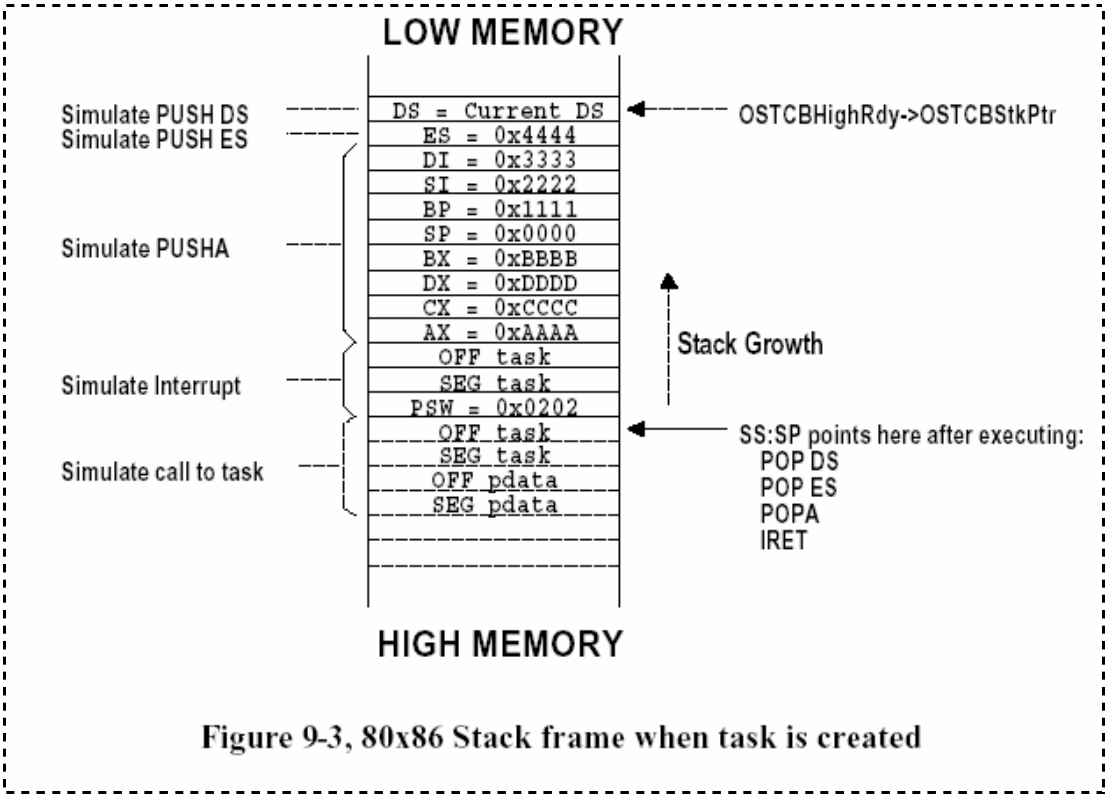


Figure 9-3, 80x86 Stack frame when task is created

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。

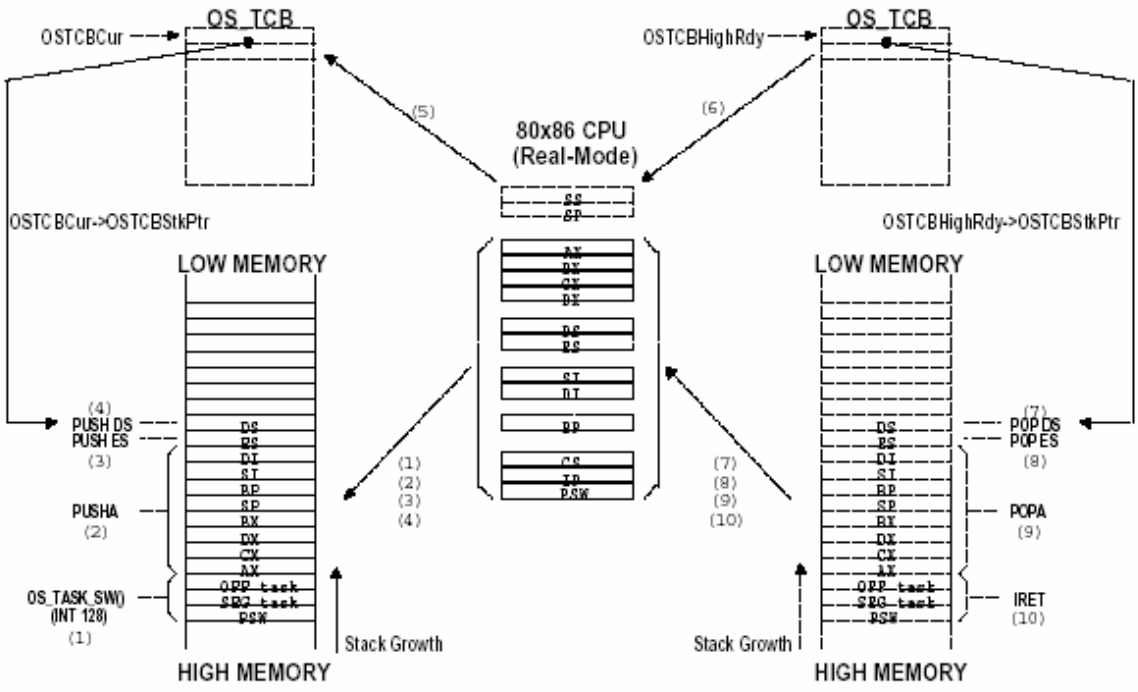


**Figure 9-4, 80x86 Stack frames during a task level context switch**

**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**

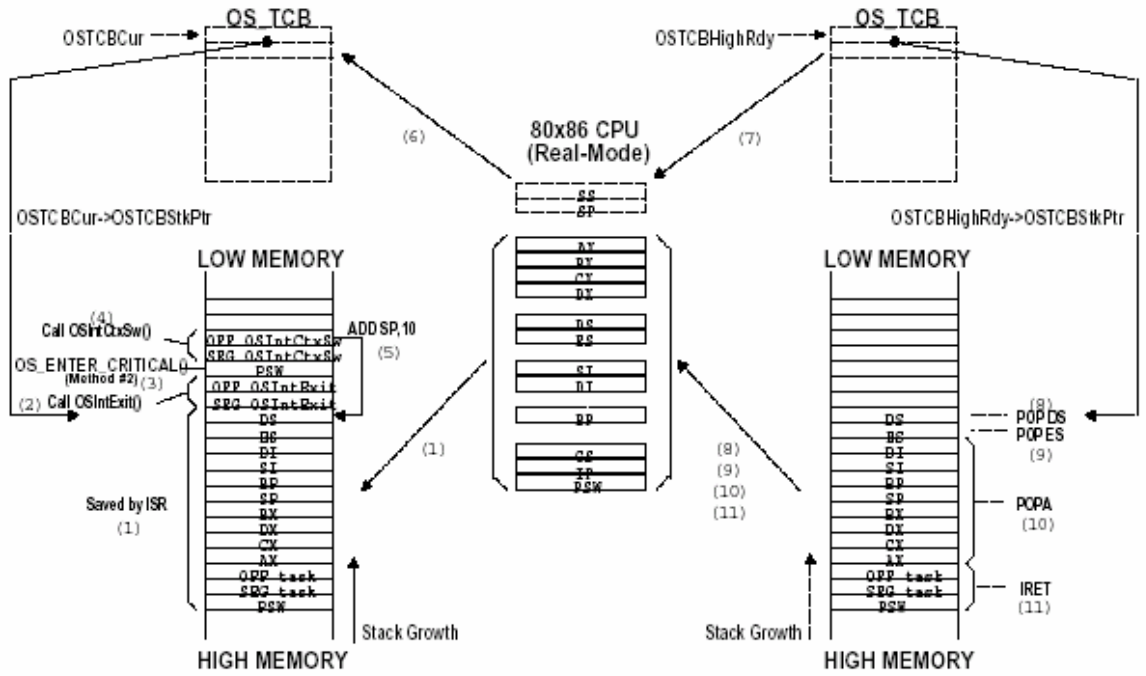


**Figure 9-5, 80x86 Stack frames during an interrupt level context switch**

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).

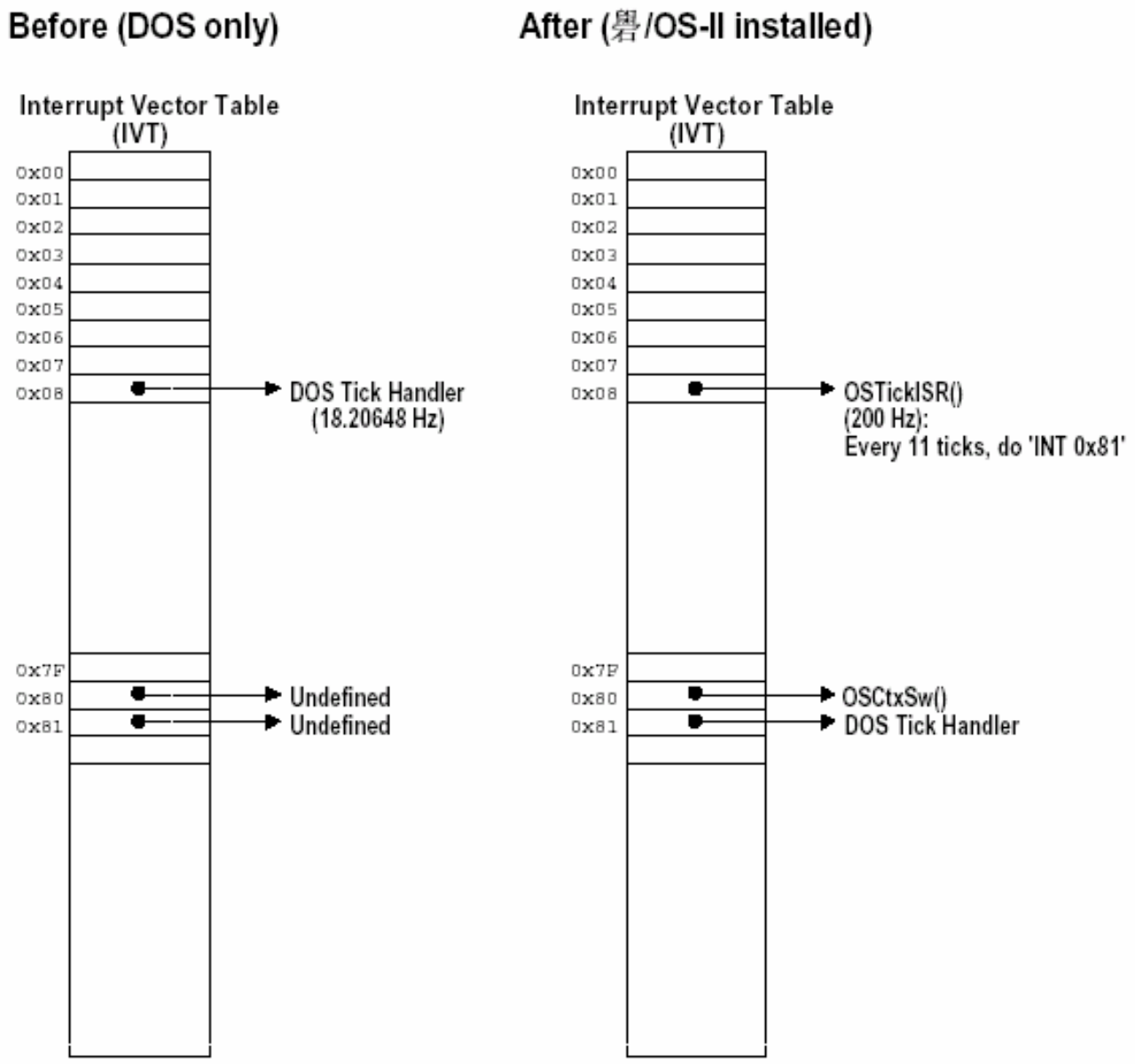


Figure 9-6, PC's Interrupt Vector Table (IVT)

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F9.7　　传递参数 pdata的堆栈初始化结构

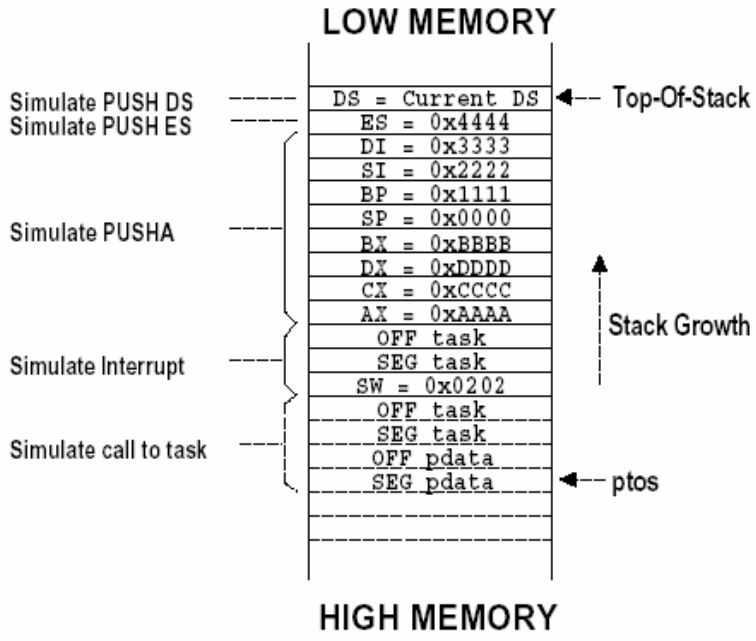


Figure 9-7, Stack frame initialization with 'pdata' passed on the stack

### 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 |