

# 第 11 章

## 参考手册

本章提供了  $\mu$ C/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  $\mu$ C/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  $\mu$  C/OS-II 一个中断处理函数正在执行，这有助于  $\mu$  C/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  $\mu$  C/OS-II 一个中断服务已执行完毕，这有助于  $\mu$  C/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， $\mu$  C/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 = OSMsgboxAccept (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  $\mu$ C/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， $\mu$ C/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  $\mu$ C/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收到需要的消息，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时释放回内存区。

### **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  $\mu$ C/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	<b>OSAddr;</b>	/*指向内存区起始地址的指针	*/
Void	<b>OSFreeList;</b>	/*指向空闲内存块列表起始地址的指针	*/
INT32U	<b>OSBlkSize;</b>	/*每个内存块的大小	*/
INT32U	<b>OSNBlks;</b>	/*该内存区的内存块总数	*/
INT32U	<b>OSNFree;</b>	/*空闲的内存块数目	*/
INT32U	<b>OSNUsed;</b>	/*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  $\mu$ C/OS-II    */
    .
    .
    CommQ = OSQCreate(&CommMsg[0], 10);     /* 建立消息队列      */
    .
    .
    OSStart();                               /* 启动多任务内核    */
}
```

## ***OSQFlush( )***

**INT8U \*OSQFlush (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， $\mu$ C/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  $\mu$ C/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。 $\mu$ C/OS-II 可以支持多达 254 层的 OSSchedLock ( ) 函数嵌套，必须调用同样次数的 OSSchedUnlock ( ) 函数才能恢复任务调度。

### 参数

无

### 返回值

无

### 注意/警告

任务调用了 OSSchedLock ( ) 函数后，决不能再调用可能导致当前任务挂起的系统函数：OSTimeDly ( )，OSTimeDlyHMSM ( )，OSSemPend ( )，OSMboxPend ( )，OSQPend ( )。因为任务调度已经被禁止，其他任务不能运行，这会导致系统死锁。

### 范例：

```
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  $\mu$ C/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， $\mu$ C/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  $\mu$ C/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  $\mu$  C/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  $\mu$ 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( )***



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  $\mu$ C/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  $\mu$ C/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  $\mu$ C/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uC/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  $\mu$  C/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，任务切入次数等信息。

`opt` 存放与任务相关的操作信息。`opt` 的低 8 位由  $\mu$ C/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  $\mu$ C/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  $\mu$ C/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  $\mu$ C/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  $\mu$ 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  $\mu$ 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  $\mu$ C/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  $\mu$ C/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  $\mu$  C/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** 为延时分分钟数，范围从 0-59。  
**seconds** 为延时秒数，范围从 0-59  
**milli** 为延时毫秒数，范围从 0-999。需要说明的是，延时操作函数都是以时钟节拍为单位的。实际的延时时间是时钟节拍的整数倍。例如系统每次时钟节拍间隔是 10ms，如果设定延时为 5ms，将不产生任何延时操作，而设定延时 15ms，实际的延时是两个时钟节拍，也就是 20ms。

返回值

- 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  $\mu$  C/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    */
        .
        .
    }
}
```