μC/OS-II 2.92 API 函数详解

修正记录

版本	日期	内容
V1.0	2015.8.25	初版
V1.1	2015.8.30	NO.1 更正软件定时器部分说明文字。 NO.2 修改部分 API 函数错误代码排版。 NO.3 增加 API 的简介说明
V1.2	2015.9.3	NO.1 修改任务通讯部分 API 使用说明文字，使之更加通俗易懂。
V2.0	2016.12.04	NO.1 重新调整了内容，更符合学习的顺序。 NO.2 修正了部分 BUG。 MODIFY BY 陈志发 2016/01/04

深圳信盈达陈志发编制

目录

深圳信盈达技术有限公司 未经允许不得复制

深圳信盈达技术总监专用

深圳信盈达陈志发编制

μC/*OS-II* 2.92 API 函数详解

1.1 任务管理

uCOSII 的任务函数看起来与任何普通 C 函数一样，具有一个返回类型和一个参数，只是它从不返回。任务的返回类型必须被定义成 void 型。

- **相关的 API 函数：**
- OSStatInit() 统计任务初始化。
- OSTaskChangePrio () 改变一个任务的优先级。
- OSTaskCreate () 创建任务。
- OSTaskCreateExt () 创建扩展任务。
- OSTaskDel () 删除任务。
- OSTaskDelReq () 请求一个任务删除其它任务或自身。
- OSTaskNameGet() 获取任务名称。
- OSTaskNameSet() 设置任务名称。
- OSTaskResume () 唤醒一个用 OSTaskSuspend()函数挂起的任务。
- OSTaskStkChk () 检查任务堆栈状态。
- OSTaskSuspend () 无条件挂起一个任务。
- OSTaskQuery () 获取任务信息。

1.1.1 OSStatInit()

作用：统计任务初始化。

函数原型：

void OSStatInit (void);

所属文件	调用者	开关量
OS_CORE.C	只能是启动代码	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。

在使用本函数时，OS_TASK_STAT_EN 必须置为 1。这时候 uCOS-II 会自动创建一个统计任务 OSTaskStat()。这个统计任务每秒计算一次 CPU 在单位时间内被使用的时间，并把计算结果以百分比的形式存放在变量 **OSCPUUsage** 中，以便应用程序通过访问它来了解 CPU 的利用率。

参数： 无。

返回值：无。

注意：

如果用户应用程序要使用统计任务，则必须把定义在系统头文件 OS_CFG.H 中的系统配置常量 OS_TASK_STAT_EN 设置为 1，并且必须在创建统计任务之前调用函数 OSStatInit()对统计任务进行初始化。

范例1.1.1

```
/*启动任务*/
void FirstAndOnlyTask (void *pdata)
{
        /* 初始化内核时钟,使用统计任务，调用 OSStatInit 前必须先初始化滴答时钟 */
         OS_CPU_SysTickInit(72000000L / OS_TICKS_PER_SEC);
```

深圳信盈达技术有限公司 专业提供单片机、嵌入式、M3、A9、LINUX 技术服务、技术培训。

```
    OSStatInit();                    /*统计任务初始化，用于计算 CPU 使用率 */
    .
    OSTaskCreate(?);                 /* 建立其他任务 */
    OSTaskCreate(?);
    .
    for (;;) {
        .
    }
}

/*display 任务函数*/
void display_task(void *pdata)
{
    u8 str[32];
    while(1)
    {
        sprintf((char *)str,"CPU 使用率:%d",OSCPUUsage);
        lcd_show_str(35,80,240,16,str,16,RED,WHITE,0);/*显示 CPU 利用率*/
        OSTimeDly(100);
    }
}
```

1.1.2 OSTaskChangePrio ()

作用：改变一个任务的优先级。

函数原型：

```
INT8U OSTaskChangePrio (INT8U oldprio,
                        INT8U newprio);
```

所属文件	调用者	开关量
OS_TASK.C	只能是任务	OS_TASK_CHANGE_PRIO_EN

StaskChangePrio()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()返回下列错误代码之一：

OS_ERR_NONE: 任务的优先级更改成功。

OS_ERR_PRIO_INVALID: 旧优先或新的优先权是否等于或超过 OS_LOWEST_PRIO。

OS_ERR_PRIO_EXIST: newprio 已经存在。

OS_ERR_PRIO: oldprio 不存在。

OS_ERR_TASK_NOT_EXITS: newprio 为互斥 PIP,不允许。

注意：

所需的优先级一定不能已经存在;否则将返回错误代码。

范例1.1.2

```
void TaskX (void *p_arg)
{
    INT8U err;
    while(1)
    {
    ...
    err = OSTaskChangePrio(10, 15);
    ...
    }
}
```

1.1.3 OSTaskCreate()

作用：创建一个新任务。

函数原型：

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

所属文件	调用者	开关量
OS_TASK.C	只能是任务	OS_TASK_CHANGE_PRIO_EN

OSTaskCreate（）建立一个新任务。任务的建立可以在多任务环境启动之前，也可以在正在运行的任务中建立。中断处理程序中不能建立任务。**一个任务必须为无限循环结构，且不能有返回点。**

OSTaskCreate（）是为与先前的 uC/OS 版本保持兼容，新增的特性在 OSTaskCreateExt（）函数中。无论用户程序中是否产生中断，在初始化任务堆栈时，堆栈的结构必须与 CPU 中断后寄存器入栈的顺序结构相同。详细说明请参考所用处理器的手册。

参数：

task：是指向任务代码的指针。

pdata：指向一个数据结构，该结构用来在建立任务时向任务传递参数。

ptos：为指向任务堆栈栈顶的指针。任务堆栈用来保存局部变量，函数参数，返回地址以及任务被中断时的 CPU 寄存器内容。任务堆栈的大小决定于任务的需要及预计的中断嵌套层数。计算堆栈的大小，需要知道任务的局部变量所占的空间，可能产生嵌套调用的函数，及中断嵌套所需空间。如果初始化常量 OS_STK_GROWTH 设为 1，堆栈被设为从内存高地址向低地址增长，此时 ptos 应该指向任务堆栈空间的最高地址。反之，如果 OS_STK_GROWTH 设为 0，堆栈将从内存的低地址向高地址增长。

prio：为任务的优先级。每个任务必须有一个唯一的优先级作为标识。数字越小，优先级越高。

返回值：

OSTaskCreate（）返回下列错误代码之一：

OS_ERR_NONE： 创建成功。

OS_ERR_PRIO_EXIST： 所请求的优先级已经存在。

OS_ERR_PRIO_INVALID： prio 比 OS_LOWEST_PRIO 高。

OS_ERR_NO_MORE_TCB： μC/ OS-II 并没有任何更多的 OS_TCBs 分配。

OS_ERR_TASK_CREATE_ISR：试图从 ISR 创建任务，不允许。

注意：

任务堆栈必须声明为 OS_STK 类型。

在任务中必须调用 uC/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。

用户程序中不建议使用优先级 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 uC/OS 系统保留，其余的 56 个优先级提供给应用程序。

范例1.1.3

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  UC/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1.1.4

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*/
        }
}
```

1.1.4 OSTaskCreateExt ()

作用：建立一个新任务。

函数原型：

```
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	任务或者初始化代码	OS_TASK_CREATE_EXT_EN

OSTaskCreateExt（）建立一个新任务。任务的建立可以在多任务环境启动之前，也可以在正在运行的任务中建立，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。与 OSTaskCreate（）不同的是，OSTaskCreateExt（）允许用户设置更多的细节内容。

参数：

task：是指向任务代码的指针。

pdata：指向一个数据结构，该结构用来在建立任务时向任务传递参数。

ptos：为指向任务堆栈栈顶的指针。任务堆栈用来保存局部变量，函数参数，返回地址以及任务被中断时的 CPU 寄存器内容。任务堆栈的大小决定于任务的需要及预计的中断嵌套层数。计算堆栈的大小，需要知道任务的局部变量所占的空间，可能产生嵌套调用的函数，及中断嵌套所需空间。如果初始化常量 OS_STK_GROWTH 设为 1，堆栈被设为从内存高地址向低地址增长，此时 ptos 应该指向任务堆栈空间的最高地址。反之，如果 OS_STK_GROWTH 设为 0，堆栈将从内存的低地址向高地址增长。

prio：为任务的优先级。每个任务必须有一个唯一的优先级作为标识。数字越小，优先级越高。

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(就是任务栈元素数量)。其单位由 OS_STK 定义：当 OS_STK 的类型定义为 INT8U、INT16U、INT32U 的时候， stk_size 的单位为分别为字节（8 位）、字（16 位）和双字（32 位）。

pext：是一个用户定义数据结构的指针，可作为 TCB 的扩展。例如，当任务切换时，用户定义的数据结构中可存放浮点寄存器的数值，任务运行时间，任务切入次数等等信息。一般情况下使用 NULL

opt：存放与任务相关的操作信息。opt 的低 8 位由 uC/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 uC/OS 版本支持下列操作：

~~**OS_TASK_OPT_NONE**：~~ ~~任务创建成功。~~ ← 不存在

OS_TASK_OPT_STK_CHK： 决定是否进行任务堆栈检查(已经使用了多少空间)。

OS_TASK_OPT_STK_CLR： 决定是否清空堆栈。（如果栈空间数组定时已经初始化为 0，则不需要设置这一项）

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。

返回值：

OSTaskCreateExt（）返回下列错误代码之一：

OS_ERR_NONE：	创建成功。
OS_ERR_PRIO_EXIST：	所请求的优先级已经存在。
OS_ERR_PRIO_INVALID：	PRIO 比 OS_LOWEST_PRIO 高。
OS_ERR_NO_MORE_TCB：	μC/ OS-II 并没有任何更多的 OS_TCBs 分配。
OS_ERR_TASK_CREATE_ISR：	试图从 ISR 创建任务，不允许。

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 uC/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 uC/OS 系统保留，其余 56 个优先级提供给应用程序。

范例1.1.5

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 uC/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1.1.6

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 uC/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      */
    }
}
```

1.1.5 OSTaskDel ()

作用：删除任务，不让再有机会调度，除非重新创建。

函数原型：

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（）返回下列错误代码之一：

OS_ERR_NONE：删除成功(在任务不删除自己的情况下)。
OS_ERR_TASK_IDLE：试图删除空闲任务，不允许。
OS_ERR_TASK_DEL：要删除的任务不存在。
OS_ERR_PRIO_INVALID：指定任务优先级高于 OS_LOWEST_PRIO。
OS_ERR_TASK_DEL_ISR：尝试从 ISR 删除任务，不允许。
OS_ERR_TASK_DEL：任务被分配到一互斥。
OS_ERR_TASK_NOT_EXIST：如果任务被分配到一互斥优先级继承。

注意：

OSTaskDel（）将判断用户是否试图删除 uC/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1.1.7

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

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

1.1.6 OSTaskDelReq()

作用：请求一个任务删除其它任务或自身。

函数原型：

INT8U OSTaskDelReq(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_ERR_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_ERR_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（）返回下列错误代码之一：

OS_ERR_NONE：删除请求已任务记录。

OS_ERR_TASK_NOT_EXIST：指定的任务不存在。发送删除请求的任务可以等待此返回值看是否成功。

OS_ERR_TASK_IDLE：尝试删除空闲任务，不允许。

OS_ERR_PRIO_INVALID：指定任务的优先级比 OS_LOWEST_PRIO 高于或不指定 OS_PRIO_SELF。

OS_ERR_TASK_DEL：任务被分配到一互斥。

OS_ERR_TASK_DEL_REQ：当前任务收到来自其他任务的删除请求。

注意：

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

范例1.1.8

```
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_ERR_NONE)    //首次请求
       {
            /* 等待任务删除      */
            while (err != OS_ERR_TASK_NOT_EXIST)
            {

                OSTimeDly(1);
                err = OSTaskDelReq(10);
            }
            /* 任务#LED4_TASK_PRO 已被删除 */
       }
    }
}


void TaskToBeDeleted(void *pdata)          /* 任务优先级 10     */
{
    pdata = pdata;
    for (;;) {

       if (OSTaskDelReq(OS_PRIO_SELF) == OS_ERR_TASK_DEL_REQ)
       {
            /* 释放任务占用的系统资源   */

            /* 释放动态分配的内存   */

            /* 删除自己   */
            OSTaskDel(OS_PRIO_SELF);
       }
      OSTimeDly(1);

    }
}
```

1.1.7 OSTaskNameGet()

作用：获取指定优先级任务的名称。

函数原型：

```
INT8U OSTaskNameGet(INT8U prio,
                    INT8U **pname,
                    INT8U *perr);
```

所属文件	调用者	开关量
OS_TASK.C	任务	OS_TASK_NAME_EN

OSTaskNameGet（）获取任务名称。

参数:

prio: 是你将要从中获取从名字的任务的优先级。如果指定 OS_PRIO_SELF，你会获得当前任务的名称。

pname: 指向包含该任务的名称的 ASCII 字符串。

perr: 指向错误代码变量的指针，其返回值如下:

OS_ERR_NONE: 获取成功。
OS_ERR_TASK_NOT_EXIST: 指定的任务没有创建或已被删除。
OS_ERR_PRIO_INVALID: 指定了一个无效的优先级。
OS_ERR_PNAME_NULL: pname 你传递一个 NULL 指针。
OS_ERR_NAME_GET_ISR: 尝试从 ISR 此功能，不允许。

返回值: 指向 pname 的 ASCII 字符串的大小；如果为 0，说明遇到错误。

范例1.1.9

```
INT8U *EngineTaskName;          //存放名字字符串指针
void Task (void *p_arg)
{
    INT8U err;
    INT8U size;   //存放任务名的长度
    (void)p_arg;
    while(1)
    {
        size = OSTaskNameGet(OS_PRIO_SELF, &EngineTaskName, &err);
        ...
    }
}
```

1.1.8 OSTaskNameSet()

作用: 设置任务名称。

函数原型:

```
void OSTaskNameSet(INT8U prio,
                   INT8U *pname,
                   INT8U *perr);
```

所属文件	调用者	开关量
OS_TASK.C	任务	OS_TASK_NAME_EN

OSTaskNameSet（）设置任务名称。

参数:

prio: 将要设置名字的任务的优先级。如果指定 OS_PRIO_SELF，将设置当前任务的名称。

pname: 指向包含任务名称的 ASCII 字符串。

perr: 指向错误代码变量的指针，其返回值如下:

OS_ERR_NONE: 设置成功。
OS_ERR_PNAME_NULL: pname 指向 NULL。
OS_ERR_NAME_SET_ISR: 在 ISR 使用本函数，不允许。
OS_ERR_PRIO_INVALID: 指定了无效的优先级。

OS_ERR_TASK_NOT_EXIST：指定的任务没有创建或已被删除。

返回值：无

范例1.1.10

```
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while(1)
    {
        OSTaskNameSet(OS_PRIO_SELF,"Engine Task", &err);//表示设置任务本身
        ...
    }
}
```

1.1.9 OSTaskResume()

作用：无条件唤醒一个任务。

函数原型：

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。

pname: 指向包含任务名称的 ASCII 字符串。

返回值：

OS_ERR_NONE: 解挂成功。

OS_ERR_TASK_RESUME_PRIO: 试图解挂不存在的任务。

OS_ERR_TASK_NOT_SUSPENDED: 要解挂的任务尚未挂起。

OS_ERR_PRIO_INVALID: prio 高于或等于 OS_LOWEST_PRIO。

OS_ERR_TASK_NOT_EXIST: 任务被分配到一互斥 PIP。

范例1.1.11

```
void TaskX(void *pdata)
{
    INT8U err;
    for (;;) {
        err = OSTaskResume(10);          /*  唤醒优先级为 10 的任务      */
        if (err == OS_ERR_NONE) {
            .                            /*  任务被唤醒                  */
        }
    }
```

```
}
```

1.1.10 OSTaskSuspend()

作用：无条件挂起一个任务。(放弃 CPU，休眠了，不再参与任务调度)

函数原型：

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()返回的这些错误代码之一：

OS_ERR_NONE： 任务挂起成功。

OS_ERR_TASK_SUSPEND_IDLE：试图挂起空闲任务，不允许。

OS_ERR_PRIO_INVALID： 指定了无效的任务优先级。

OS_ERR_TASK_SUSPEND_PRIO：要挂起的任务不存在。

OS_ERR_TASK_NOT_EXITS： 任务被分配到一互斥 PIP。

注意：

1) 在程序中 OSTaskSuspend()和 OSTaskResume ()应该成对使用。
2) 用 OSTaskSuspend()挂起的任务只能用 OSTaskResume ()唤醒。

范例1.1.12

```
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 */
    }
}
```

1.1.11 OSTaskStkChk()

作用：检查任务堆栈状态。任务栈很重要，如果设置小了，可能造成程跑飞。

函数原型：

INT8U OSTaskStkChk(INT8U prio, OS_STK_DATA *p_stk_data);

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（）返回的这些错误代码之一：

OS_ERR_NONE：　　获取成功。

OS_ERR_PRIO_INVALID：　指定了无效的任务优先级（任务不存在）。

OS_ERR_TASK_NOT_EXIST：指定的任务不存在

OS_ERR_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。

OS_ERR_PDATA_NULL：　p_stk_data 是一个空指针。

注意：

1) 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。
2) 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1.1.13

```
void Task (void *pdata)
{
    OS_STK_DATA stk_data;
    INT32U      stk_size;
    for (;;) {
        err = OSTaskStkChk(10, &stk_data);
        if (err == OS_ERR_NONE) {
            stk_size = stk_data.OSFree + stk_data.OSUsed;
            //stk_data.OSUsed *100 /stk_size；使用的百分比，如果比例接近 100，则要考虑增加栈
        }
    }
}
```

1.1.12 OSTaskQuery ()

作用：获取任务信息。

函数原型：

```
INT8U OSTaskQuery(INT8U prio,
                  OS_TCB *p_task_data);
```

所属文件	调用者	开关量
OS_TASK.C	任务或中断	OS_TASK_QUERY_EN

OSTaskQuery()获取任务的相关信息。应用程序必须分配的 OS_TCB 数据结构来获得所需的任务控制块的信息。您的副本包含了 OS_TCB 结构体的各个成员。

在访问 OS_TCB 结构的时候，特别是 OSTCBNext 和 OSTCBPrev 的内容，它们指向下一个和以前 OS_TCB 的任务链，所以必须小心使用。

参数：

prio：要获取数据的任务优先级。可以通过指定 OS_PRIO_SELF 调用任务本身的信息。

p_task_data：是指向 OS_TCB，其包含任务控制块的拷贝信息。

返回值：

OSTaskQuery（）返回的这些错误代码之一：

OS_ERR_NONE： 获取成功。
OS_ERR_PRIO_INVALID： 指定的优先级比 OS_LOWEST_PRIO 高。
OS_ERR_PRIO： 试图获得一个无效的任务信息。
OS_ERR_TASK_NOT_EXIST： 任务被分配到一互斥 PIP。
OS_ERR_PDATA_NULL： p_task_data 是一个空指针。

注意：

1.在任务控制块的字段取决于以下配置选项（参见 OS_CFG.H）：

- OS_TASK_CREATE_EN
- OS_Q_EN
- OS_FLAG_EN
- OS_MBOX_EN
- OS_SEM_EN
- OS_TASK_DEL_EN

范例1.1.14

```
void Task (void *p_arg)
{
    OS_TCB task_data;
    INT8U err;
    void *pext;
    INT8U status;
    (void)p_arg;
    for (;;)
    {
        ...
```

```
                err = OSTaskQuery(OS_PRIO_SELF, &task_data);
                if (err == OS_ERR_NONE)
                {
                    pext = task_data.OSTCBExtPtr; /* Get TCB extension pointer */
                    status = task_data.OSTCBStat; /* Get task status */
                    ...
                }
            ...
        }
    }
```

1.2 时间管理

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

- **相关的 API 函数:**
- OSTimeDly () 任务延时函数(时钟节拍数)。
- OSTimeDlyHMSM () 将一个任务延时若干时间(设定时、分、秒、毫秒)。
- OSTimeDlyResume () 唤醒一个用 OSTimeDly()或 OSTimeDlyHMSM()函数的任务(优先级)。
- OSTimeGet () 获取当前系统时钟数值。
- OSTimeSet () 设置当前系统时钟数值。

在 os_cfg.h 中有定义:
#define OS_TICKS_PER_SEC 1000
意思是把 1 秒分成 1000 份，则每份是 1ms，则一个时钟节拍就是 1MS

1.2.1 OSTimeDly()

作用: 任务延时函数(时钟节拍数)。实际是上放弃 CPU，休眠去了。
函数原型:
void OSTimeDly(INT32U ticks);

所属文件	调用者	开关量
OS_TIME.C	只能是任务	N/A

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 2^32 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:
因为延时时间 0 表示不进行延时操作，而立即返回调用者。所以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1.2.1

```
void TaskX(void *pdata)
{
    for (;;) {
        OSTimeDly(10);                    /* 任务延时 10 个时钟节拍 */
    }
}
```

1.2.2 OSTimeDlyHMSM()

作用：任务延时函数(时钟节拍数)。

函数原型：

```
INT8U OSTimeDlyHMSM (INT8U hours,
                     INT8U minutes,
                     INT8U seconds,
                     INT16U ms);
```

所属文件	调用者	开关量
OS_TIME.C	只能是任务	N/A

OSTimeDlyHMSM()将一个任务延时若干时间。延时的单位是小时、分、秒、毫秒。所以使用 OSTimeDlyHMSM()比 OSTimeDly()更方便。调用 OSTimeDlyHMSM()后，如果延时时间不为 0，系统将立即进行任务调度。

参数：

hours： 为延时小时数，范围从 0-255。

minutes：为延时分钟数，范围从 0-59。

seconds：为延时秒数，范围从 0-59。

ms： 为延时毫秒数，范围从 0-999。需要说明的是，延时操作函数都是以时钟节拍为为单位的。实际的延时时间是时钟节拍的整数倍。例如系统每次时钟节拍间隔是 10ms，如果设定延时为 5ms，将不产生任何延时操作，而设定延时 15ms，实际的延时是两个时钟节拍，也就是 20ms。

返回值：

OSTimeDlyHMSM()返回的这些错误代码之一：

OS_ERR_NONE: 指定有效的参数并且调用成功。

OS_ERR_TIME_INVALID_MINUTES：分钟参数大于 59。

OS_ERR_TIME_INVALID_SECONDS：秒参数大于 59。

OS_ERR_TIME_INVALID_MS： 毫秒参数大于 999。

OS_ERR_TIME_ZERO_DLY： 所有四个参数都为 0。

OS_ERR_TIME_DLY_ISR： 从 ISR 使用此功能，不允许。

注意：

OSTimeDlyHMSM(0，0，0，0)表示不进行延时操作，而立即返回调用者。

范例 1.2.2

```
void TaskX (void *p_arg)
{
```

```
    for (;;)
    {
        ...
        OSTimeDlyHMSM(0, 0, 1, 0); /* Delay task for 1 second */
        ...
    }
}
```

1.2.3 OSTimeDlyResume()

作用：唤醒一个用 OSTimeDly()或 OSTimeDlyHMSM()函数的任务。

函数原型：

```
INT8U OSTimeDlyResume(INT8U prio);
```

所属文件	调用者	开关量
OS_TIME.C	只能是任务	N/A

OSTimeDlyResume()用于提前唤醒一个用 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()返回的这些错误代码之一：

OS_ERR_NONE： 唤醒任务成功。

OS_ERR_PRIO_INVALID： 指定任务优先级大于 OS_LOWEST_PRIO 更大。

OS_ERR_TIME_NOT_DLY： 任务没有在延时。

OS_ERR_TASK_NOT_EXIST:任务尚未创建或已被分配给一个互斥 PIP。

范例1.2.3

```
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_ERR_NONE) {
            .                               /*  任务被唤醒                  */
        }
    }
}
```

1.2.4 OSTimeGet()

作用：获取当前系统时钟节拍数值。

函数原型：

```
INT32U OSTimeGet(void);
```

所属文件	调用者	开关量

OS_TIME.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：无。

范例1.2.4

```
void TaskX(void *pdata)
{
    INT32U clk;
    for (;;) {
        . ..
        clk = OSTimeGet();   /*  获取当前系统时钟的值  */
        …
    }
}
```

这个值是一直变化的 可以用来做随便函数的随机种子，或做文件名等。

1.2.5 OSTimeSet()

作用：设置当前系统时钟数值。

函数原型：

```
void OSTimeSet(INT32U ticks);
```

所属文件	调用者	开关量
OS_TIME.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：无。

范例1.2.5

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

```
            OSTimeSet(0L);      /*  复位系统时钟    */
            .
        }
    }
```

1.3 信号量

信号量用于对共享资源的访问。信号量标识了共享资源的有效可被访问数量，要获得共享资源的访问权，首先必须得到信号量这把钥匙。使用信号量管理共享资源，请求访问资源就演变为请求信号量了。资源是具体的现实的东西，把它数字化后，操作系统就便于管理这些资源，这就是信号量的理论意义。

在μC/OS-II 信号量中，信号量的取值范围是 16 位的二进制整数，范围是十进制的 0 ~ 65535。或是其他长度，如 8 位、32 位。

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。

- **相关的 API 函数:**
- **OSSemAccept()　无等待请求一个信号量。**
- **OSSemCreate()　建立并初始化一个信号量。　第 1 个使用函数**
- OSSemDel()　删除一个信号量。
- **OSSemPend ()　挂起任务等待一个信号量。　第 2 个使用函数**
- OSSemPendAbort()　放弃任务中信号量的等待。
- **OSSemPost ()　发出一个信号量函数。　第 3 个使用函数**
- OSSemQuery ()　查询一个信号量的相关信息。
- OSSemSet()　用于改变当前信号量的计数值。

信号使用注意做到生产和消费均衡
信号量可以是其他任务或中断程序中释放
信号量可以是本任务申请，本任务释放—这种情况用于保护共享资源。
示例说明 ：
psem = OSSemCreate(1);//初始值为 1 的信号量

A 任务:

```
。。。。。。
while(1)
{
  OSSemPend(psem, 0, &err);//timeout 为 0 表示无限等待，不存在超时情况
    往串口 DR 寄存器写数据
    USART1->DR = 0x30;
  OSSemPost(psem);
}
```

B 任务:

```
。。。。。。
while(1)
{
    OSSemPend(psem, 0, &err);//timeout 为 0 表示无限等待，不存在超时情况
```

深圳信盈达技术有限公司 未经许可 禁止发布

```
    往串口 DR 寄存器写数据
    USART1->DR = 0x60;
     OSSemPost(psem);
}
```

1.3.1 OSSemAccept()

作用：无等待查看消息邮箱是否收到消息。

函数原型：

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&& OS_SEM_ACCEPT_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：

必须先建立信号量，然后使用。

范例 1.3.1：

```
OS_EVENT *DispSem;
void Task (void *p_arg)
{
    INT16U value;
    (void)p_arg;
    while(1)
    {
        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 */
        }
        ...
    }
}
```

1.3.2 OSSemCreate ()

作用：建立并初始化一个信号量。

函数原型：

OS_EVENT ***OSSemCreate(INT16U value);**

所属文件	调用者	开关量
OS_SEM.C	任务或初始化代码	OS_SEM_EN

OSSemCreate（）函数建立并初始化一个信号量。

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。不允许在中断中调用此函数

注意:

必须在使用信量量前创建。

范例1.3.2

```
OS_EVENT *DispSem;
void main(void)
{
        OSInit();                          /* 初始化 uC/OS-Ⅱ          */

        DispSem = OSSemCreate(1);          /* 建立显示设备的信号量      */
        ...
        OSStart();                         /* 启动多任务内核 */
}
```

1.3.3 OSSemDel ()

作用: 删除一个信号量。

函数原型:

```
OS_EVENT *OSSemDel(OS_EVENT *pevent,
                   INT8U opt,
                   INT8U *perr);
```

所属文件	调用者	开关量
OS_SEM.C	任务	OS_SEM_EN && OS_SEM_DEL_EN

OSSemDel ()用于删除一个信号量。使用这个函数必须小心，要检测 OSSemPend 函数的错误码情况。

参数:

pevent: 指向信号量的指针。

opt: 选择指定要删除信号量类型:

OS_DEL_NO_PEND: 只有当不存在任务删除挂起时才删除，否则不删除。

OS_DEL_ALWAYS: 直接删除。

perr: 指向错误代码变量的指针，其返回值如下:

OS_ERR_NONE 信号量已经成功删除。

OS_ERR_DEL_ISR 尝试从一个 ISR 删除一个信号量。

OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。
OS_ERR_EVENT_TYPE	pevent 不是指向一个信号量。
OS_ERR_INVALID_OPT	传入选项参数(opt)有误。
OS_ERR_TASK_WAITING	一个或多个任务正在等待信号量。

返回值:

如果信号量被成功删除，则返回空指针；否则需要根据错误代码进行检查。

注意:

必须先建立信号量，然后使用。

范例1.3.3

```
OS_EVENT *DispSem;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while(1)
    {
        ...
        DispSem = OSSemDel(DispSem, OS_DEL_ALWAYS, &err);
        if (DispSem == (OS_EVENT *)0)
        {
            /* Semaphore has been deleted */
        }
        ...
    }
}
```

1.3.4 OSSemPend()

作用：申请一个信号量（挂起任务等待信号量）。

函数原型:

```
void   OSSemPend  (OS_EVENT *pevent,
                   INT32U    timeout,
                   INT8U     *perr);
```

所属文件	调用者	开关量
OS_SEM.C	只能是任务	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。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（把信号值设置为>0 值）或 超出等待的预期时间 timeout。如果在预期的时钟节拍内信号量被置起，uC/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。

这个函数返回情况:

- 正确得到信号量

- 函数执行出错
- 等待超时

参数：

pevent: 指向消息邮箱的指针。

timeout：任务等待超时周期，为 0 时表示无限等待；其它，递减到 0 时任务恢复执行。

perr: 做为输出参数使用，指向错误代码变量的指针，***perr** 其返回值如下：

OS_ERR_NONE	**信号量不为 0，任务得到信号量，可以运行。**
OS_ERR_TIMEOUT	信号量未在指定的超时周期内置起。
OS_ERR_PEND_ABORT	等待中止。是由于另一个任务或 ISR 通过调用 OSSemPendAbort ()。
OS_ERR_EVENT_TYPE	pevent 不是指向一个信号量。
OS_ERR_PEND_LOCKED	在调用本函数时，调度器被锁定。
OS_ERR_PEND_ISR	尝试从一个 ISR（中断程序）调用 OSSemPend ()，不允许。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针 NULL。这个选项依赖于 OS_ARG_CHK_EN 这个宏，在 os_cfg.h 中有定义，要检测参数是否空，则需要打开 OS_ARG_CHK_EN 这个宏。

返回值：无

注意：

必须先建立信号量，然后使用。

范例1.3.4

```
OS_EVENT *DispSem;
void DispTask(void *pdata)
{
        INT8U    err;
        pdata = pdata;
        while(1)
        {
            ...
            OSSemPend(DispSem, 0, &err);
            /*只有信号量置起，该任务才能执行*/
            ...
        }
}
```

1.3.5 OSSemPendAbort()

作用：使正在等待该信号量的任务取消挂起。不再等待。

函数原型：

```
INT8U OSSemPendAbort( OS_EVENT *pevent,
                      INT8U opt,
                      INT8U *perr);
```

所属文件	调用者	开关量
OS_SEM.C	只能是任务	OS_SEM_EN &&

		OS_SEM_PEND_ABORT_EN

OSSemPendAbort ( ) 用来使正在等待该信号量的任务取消挂起。取消任务挂起后，uC/OS-II 会自动执行调度。若应用中不需其在取消任务后实现调度，可在上述两种方式后或上操作宏 OS_OPT_POST_NO_SCHED。

参数:

pevent: 指向信号量的指针。

opt: 指定要中止信号量等待的类型。

OS_PEND_OPT_NONE: 只使正在等待该信号量的最高优先级任务取消挂起。

OS_PEND_OPT_BROADCAST: 使所有正在等待该信号量的任务取消挂起。

perr: 指向错误代码变量的指针，其返回值如下:

OS_ERR_NONE	没有任务在等待该信号量。
OS_ERR_EVENT_TYPE	pevent 不是指向一个信号量。
OS_ERR_PEND_ABORT	至少一个任务等待的信号量已经退出等待。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。

返回值:

>0 返回被本函数取消挂起的正在等待信号量的任务数。

0 表示没有任务在等待该信号量，因此该功能没作用。

注意:

必须先建立信号量，然后使用。

范例1.3.5

```
OS_EVENT * psem;
void CommTaskRx (void *p_arg)
{
    INT8U err,ret;
    (void)p_arg;
    while(1)
    {
        ...
        ret = OSSemPendAbort (psem, OS_PEND_OPT_NONE, &err);
        if(ret == 0)
        {
            printf("file:%s\r\nline:%d\r\nno tasks were waiting on the semaphore, or upon error\r\n ",
                __FILE__, __LINE__);
        }
        else
        {
            printf("file:%s\r\nline:%d\r\none or more tasks waiting on the semaphore are now readied and informed\r\n ",
    __FILE__, __LINE__);
            }        ...
    }
}
```

陈志发编制

1.3.6 OSSemPost()

作用：发出一个信号量，把信号值加 1。

函数原型：

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(把信号值加 1)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。

参数：

pevent： 指向信号量的指针。

返回值：

OS_ERR_NONE 信号量成功置起。
OS_ERR_SEM_OVF 信号计数溢出。
OS_ERR_EVENT_TYPE pevent 不是指向一个消息邮箱。
OS_ERR_PEVENT_NULL pevent 传入的是一个空指针。

注意：

必须先建立信号量，然后使用。

范例1.3.6

```
OS_EVENT *DispSem;
void TaskX(void *pdata)
{
    INT8U   err;
    pdata = pdata;
    while(1)
    {
        err = OSSemPost(DispSem);
        if (err == OS_ERR_NONE)
        {
            ./*  信号量置起   */
        }
        else
        {
            ./*  信号量溢出  */
        }
    }
}
```

1.3.7 OSSemQuery()

作用：获取一个信号量的相关信息。

函数原型：

```
INT8U OSSemQuery(OS_EVENT *pevent,
                 OS_SEM_DATA *p_sem_data);;
```

所属文件	调用者	开关量
OS_SEM.C	任务或中断	OS_SEM_EN && OS_SEM_QUERY_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。

参数:

pevent: 指向信号量的指针。

p_sem_data: 是指向 OS_SEM_DATA 数据结构的指针，该数据结构定义如下:

```
typedef struct os_sem_data {
    INT16U    OSCnt;                                    /* Semaphore count    */
    OS_PRIO OSEventTbl[OS_EVENT_TBL_SIZE];   /* List of tasks waiting for event to occur    */
    OS_PRIO OSEventGrp                        /* Group corresponding to tasks waiting for event to occur */
} OS_SEM_DATA;
```

返回值:

OS_ERR_NONE	获取成功。
OS_ERR_EVENT_TYPE	pevent 不是指向一个信号量。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。
OS_ERR_PNAME_NULL	p_mbox_data 传入的是一个空指针。

注意:

必须先建立信号量，然后使用。

范例1.3.7

```
OS_EVENT *DispSem;
void Task (void *p_arg)
{
    OS_SEM_DATA sem_data;
    INT8U err;

    while(1)
    {
        ...
        err = OSSemQuery(DispSem, &sem_data);
        if (err == OS_ERR_NONE)
        {
          /* Examine sem_data */
        ...
        }
        ...
    }
```

```
}
```

1.3.8 OSSemSet()

作用：改变当前信号量的计数值。当信号量被用于表示某事件发生了多少次的情况下会使用。

函数原型：

```
void OSSemSet(OS_EVENT *pevent,
              INT16U cnt,
              INT8U *perr);
```

所属文件	调用者	开关量
OS_SEM.C	任务或中断	OS_SEM_EN && OS_SEM_SET_EN

OSSemSet()被用于改变当前信号量的计数值。

参数：

pevent： 指向信号量的指针。

cnt： 期望的数值。

perr: 指向错误代码变量的指针，其返回值如下：

OS_ERR_NONE 计数值改变成功。

OS_ERR_EVENT_TYPE pevent 不是一个指向信号量的指针。

OS_ERR_PEVENT_NULL pevent 传入的是一个空指针。

OS_ERR_TASK_WAITING 存放任务在等待信号量。

返回值：无。

注意：

1. 必须先建立信号量，然后使用。
2. 这个函数只能修改当信号量>0 的情况，也就是没有任务等待信号量的情况。

范例1.3.8

```
OS_EVENT *SignalSem;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    for (;;)
    {
        OSSemSet(SignalSem, 0, &err); /* Reset the semaphore count */
        ...
    }
}
```

1.4 消息邮箱

邮箱顾名思义就是用于通信的，日常生活中邮箱中的内容一般是信件。在操作系统中也是通过邮箱来管理任务间的通讯与同步，但是必须注意的是，系统中的邮箱中的内容并不是信件本身，而是指向消息内容的地址！这个指针是 void 类型的，可以指向任何类型的数据结构。因此，邮箱所发送的信息范围更宽，可以容纳下任何长度的数据。

盈达际志发编制

■ **相关的 API 函数：**

- ◆ OSMboxAccept () 无等待查看消息邮箱是否收到消息。
- ◆ OSMboxCreate () 建立并初始化一个消息邮箱。
- ◆ OSMboxDel () 删除消息邮箱。
- ◆ OSMboxPendAbort() 中止任务中消息邮箱的等待。
- ◆ OSMboxPend () 挂起任务等待一个消息。
- ◆ OSMboxPost () 向邮箱发送一则消息。
- ◆ OSMboxPostOpt () 按照规则向邮箱发送一则消息。
- ◆ OSMboxQuery () 获取一个消息邮箱的相关信息。

1.4.1 OSMboxAccept ()

作用：无等待查看消息邮箱是否收到消息。

函数原型：

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 && OS_MBOX_ACCEPT_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或 pevent 传入为 NULL,或其他不是指向消息邮箱的指针，返回空指针。

注意：

必须先建立消息邮箱，然后使用。

范例1.4.1

```
OS_EVENT *CommMbox;
void Task (void *pdata)
{
    void *msg;
    pdata = pdata;
    while(1)
    {
        msg = OSMboxAccept(CommMbox); /*  检查消息邮箱是否有消息 */
        if (msg != (void *)0)                /*  如果 msg 不为空则有消息*/
        {
          ...                                /*  处理消息      */
        }
        else
        {
          ...                                /*没有消息      */
```

信盈达陈志兵编制

```
            }
        }
    }
```

1.4.3 OSMboxCreate ()

作用：建立并初始化一个消息邮箱。

函数原型：

OS_EVENT *OSMboxCreate (void *pmsg);

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，则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。

范例1.4.2

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

1.4.4 OSMboxDel ()

作用：删除一个消息邮箱

函数原型：

OS_EVENT *OSMboxDel (OS_EVENT *pevent, INT8U opt, INT8U *perr);

所属文件	调用者	开关量
OS_MBOX.C	任务	OS_FLAG_EN && OS_FLAG_DEL_EN

OSMboxDel ()用于删除一个消息邮箱。使用这个函数是很危险的，因为多个任务可能是依赖于消息邮箱而运行的。所以在删除消息邮箱之前，必须先删除所有访问消息邮箱的任务。

使用这个函数，要特别小心，删除后，OSMboxAccept()函数并不知道预期的邮箱已被删除。

OSMboxPend()函数申请消息也会失败，所以必须要检查它的返回值。

参数:

pevent: 指向消息邮箱的指针。

opt: 选择指定要删除消息邮箱类型:

OS_DEL_NO_PEND:只有当不存任务等待该消息邮箱时才删除,否则不删除。

OS_DEL_ALWAYS:不管有多少任务在等待该邮箱,都直接删除。当使用这一项删除消息邮箱,则所有等待该消息邮箱的任务都会被唤醒,并且返回 OS_ERR_PEND_ABORT 错误码

perr: 指向错误代码变量的指针,其返回值如下:

OS_ERR_NONE	消息邮箱已经成功删除。
OS_ERR_DEL_ISR	尝试从一个 ISR 删除一个消息邮箱。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。
OS_ERR_EVENT_TYPE	pevent 不是指向一个消息邮箱。
OS_ERR_INVALID_OPT	传入选项参数(opt)有误。
OS_ERR_TASK_WAITING	一个或多个任务正在等待消息。

返回值:

如果消息邮箱被成功删除,则返回空指针;否则需要根据错误代码进行检查。

范例 1.4.3

```
OS_EVENT *DispMbox;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while (1)
  {
        ...
        DispMbox = OSMboxDel(DispMbox, OS_DEL_ALWAYS, &err);
        if (DispMbox == (OS_EVENT *)0)
        {
          /* 消息邮箱被成功删除*/
        }
        ...
    }
}
```

1.4.5 OSMboxPendAbort()

作用: 使在等待该邮箱消息的任务取消挂起。

函数原型:

```
INT8U    OSMboxPendAbort (OS_EVENT *pevent,
                          INT8U     opt,
                          INT8U     *perr);
```

所属文件	调用者	开关量

未经授权禁止复制

OS_MBOX.C	只能是任务	OS_MBOX_EN && OS_MBOX_PEND_ABORT_EN

OSMboxPendAbort ( ) 用来使在等待该邮箱消息的任务取消挂起。取消任务挂起后，uC/OS-II 会自动执行调度。若应用中不需其在取消任务后实现调度，可在上述两种方式后或上操作宏 OS_OPT_POST_NO_SCHED。

参数：

pevent: 指向消息邮箱的指针。

opt: 指定要中止消息邮箱等待的类型。

OS_PEND_OPT_NONE： 只中止最高优先级任务中在等待消息的邮箱。

OS_PEND_OPT_BROADCAST： 中止所有任务中在等待消息的邮箱。

perr: 指向错误代码变量的指针，其返回值如下：

OS_ERR_NONE 没有任务在等待该邮箱。

OS_ERR_EVENT_TYPE pevent 不是指向一个消息邮箱。

OS_ERR_PEND_ABORT 至少一个任务等待的消息邮箱已经退出等待。

OS_ERR_PEVENT_NULL pevent 传入的是一个空指针。

返回值：

返回被本函数中止的正在等待该邮箱消息的任务数。0 表示没有任务在等待邮箱消息，因此该功能没作用。

范例1.4.4

```
OS_EVENT *CommMbox;
void CommTask(void *p_arg)
{
    INT8U err;
    INT8U nbr_tasks;
    (void)p_arg;
    while(1)
    {
        ...
        nbr_tasks = OSMboxPendAbort(CommMbox, OS_PEND_OPT_BROADCAST, &err);
        if (err == OS_ERR_NONE)
        {
          /* 没有任务在等待该邮箱 */
        }
        else if(err == OS_ERR_PEND_ABORT)
        {
            /*所有在等待该消息邮箱的任务退出等待 */
        }
        ...
    }
}
```

1.4.6 OSMboxPend ()

作用： 挂起任务等待一个消息

修志发编制

函数原型：

```
void    *OSMboxPend   (OS_EVENT *pevent,
                       INT32U    timeout,
                       INT8U     *perr);
```

所属文件	调用者	开关量
OS_MBOX.C	只能是任务	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，uC/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()函数后恢复任务的运行。

参数：

pevent: 指向消息邮箱的指针。

timeout: 任务等待超时周期，为 0 时表示无限等待；其它，递减到 0 时任务恢复执行。

perr: 指向错误代码变量的指针，其返回值如下：

OS_ERR_NONE	接收到消息。
OS_ERR_TIMEOUT	未在指定的超时周期内收到的消息。
OS_ERR_PEND_ABORT	等待中止。是由于另一个任务或通过 ISR 调用 OSMboxPendAbort()。
OS_ERR_EVENT_TYPE	pevent 不是指向一个消息邮箱。
OS_ERR_PEND_LOCKED	在调用本函数时，调度器被锁定。
OS_ERR_PEND_ISR	尝试从一个 ISR 调用 OSMboxPend ()，不允许。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。

返回值：

OSMboxPend()返回由任何一个任务或中断服务程序发送的消息，以及*perr 值为 OS_ERR_NONE。如果未在指定的超时期限内收到消息，返回的消息是一个 NULL 指针，* perr 值为 OS_ERR_TIMEOUT。

注意：

1) 必须先创建邮箱在使用本函数。
2) 不允许在中断函数中调用本函数。

范例 1.4.5

```
OS_EVENT *CommMbox;
void CommTask(void *p_arg)
{
    INT8U err;
    void *pmsg;
    (void)p_arg;
    while(1)
    {
        ...
        pmsg = OSMboxPend(CommMbox,0,&err);
        if (err == OS_ERR_NONE)
        {
```

深圳信盈达技术有限公司 专业编写

```
            /*处理接收到的消息*/
        }
        else
        {
            /*接收消息失败，检查相关错误代码以查明原因*/
        }
        ...
    }
}
```

1.4.7 OSMboxPost ()

作用：向邮箱发送一则消息。

函数原型：

```
INT8U   OSMboxPost   (OS_EVENT   *pevent,
                      void       *pmsg);
```

所属文件	调用者	开关量
OS_MBOX.C	任务或中断	OS_MBOX_EN && OS_MBOX_POST_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： 指向消息邮箱的指针。

pmsg： 是即将发送给任务的消息。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：

OS_ERR_NONE	发送成功。
OS_ERR_MBOX_FULL	邮箱已满。
OS_ERR_EVENT_TYPE	pevent 不是指向一个消息邮箱。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。
OS_ERR_POST_NULL_PTR	发送的内容为 NULL(空)，不允许。

注意：

1) 必须先建立消息邮箱，然后使用。
2) 不允许传递一个空指针，因为这意味着消息邮箱为空。

范例1.4.6

```
OS_EVENT *CommMbox;
INT8U CommRxBuf[100];
void CommTaskRx (void *p_arg)
{
    INT8U err;
    (void)p_arg;
```

深圳信盈达技术有限公司 专业编制

```
    while(1)
    {
        ...
        err = OSMboxPost(CommMbox, (void *)CommRxBuf);
        ...
    }
}
```

1.4.8 OSMboxPostOpt()

作用：按照规则向邮箱发送一则消息。

函数原型：

```
INT8U    OSMboxPostOpt (OS_EVENT    *pevent,
                        void        *pmsg,
                        INT8U       opt);
```

所属文件	调用者	开关量
OS_MBOX.C	任务或中断	OS_MBOX_EN && OS_MBOX_POST_OPT_EN

OSMboxPostOpt()的功能跟 OSMboxPost()类似，不同的是有多个发送选项可选。

参数：

pevent： 指向消息邮箱的指针。

pmsg： 是即将发送给任务的消息。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。

opt：发送选项，有以下几种：

OS_POST_OPT_NONE： 消息只发给等待邮箱消息的任务中优先级最高的任务。

OS_POST_OPT_BROADCAST： 消息发给所有等待邮箱消息的任务,所有等待该消息邮箱的任务都会获得消息，进入就绪状态。

OS_POST_OPT_NO_SCHED： 消息被提交到邮箱，但不马上调用调度器进行任务切换。选项是可以叠加的，例如：OS_POST_OPT_BROADCAST | OS_POST_OPT_NO_SCHED。

返回值：

OS_ERR_NONE	发送成功。
OS_ERR_MBOX_FULL	邮箱已满。
OS_ERR_EVENT_TYPE	pevent 不是指向一个消息邮箱。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。
OS_ERR_POST_NULL_PTR	发送的内容为 NULL(空)，不允许。

注意：

1) 必须先建立消息邮箱，然后使用。
2) 不允许传递一个空指针，因为这意味着消息邮箱为空。
3) 如果需要使用此功能，并希望减少代码空间，可以禁用代码生成 OSMboxPost()，因为 OSMboxPostOpt()可以模拟 OSMboxPost()。

范例 1.4.7

```
OS_EVENT *CommMbox;
```

深圳信盈达陈志发编制

```
INT8U CommRxBuf[100];
void CommTaskRx (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while(1)
    {
        ...
        err = OSMboxPostOpt(CommMbox,
                            (void *)CommRxBuf,
                            OS_POST_OPT_BROADCAST);
        ...
    }
}
```

1.4.9 OSMboxQuery ()

作用：获取一个消息邮箱的相关信息。

函数原型：

```
INT8U   OSMboxQuery   (OS_EVENT          *pevent,
                       OS_MBOX_DATA   *p_mbox_data);
```

所属文件	调用者	开关量
OS_MBOX.C	任务或中断	OS_MBOX_EN && OS_MBOX_QUERY_EN

OSMboxQuery()函数用来取得消息邮箱的相关信息。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收数据。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： 指向消息邮箱的指针。

p_mbox_data：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*/
INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];      /*消息邮箱等待队列的任务链表*/
INT8U OSEventGrp;
```

返回值：

OS_ERR_NONE	调用成功。
OS_ERR_EVENT_TYPE	pevent 不是指向一个消息邮箱。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。
OS_ERR_PNAME_NULL	p_mbox_data 传入的是一个空指针。

范例1.4.8

```
OS_EVENT *CommMbox;
void Task (void *p_arg)
{
    OS_MBOX_DATA mbox_data;
```

```
    INT8U err;
    (void)p_arg;
    while(1)
    {
        ...
        err = OSMboxQuery(CommMbox, &mbox_data);
        if (err == OS_ERR_NONE)
        {
            /*邮箱包含的相关信息获取成功*/
        }
    }
}
```

1.5 消息队列

消息队列也是用于给任务发消息，但是它是有多个消息邮箱组合而成的，是消息邮箱的集合，实质上是消息邮箱的队列。一个消息邮箱只能容纳一条消息，采用消息队列，一是可以容纳多条消息，二是消息是排列有序的。

- **相关的 API 函数：**
- OSQAccept () 检查消息队列中是否已经有需要的消息。
- OSQCreate () 建立一个消息队列。
- OSQDel () 删除一个消息队列。
- OSQFlush () 清空消息队列。
- OSQPend () 挂起任务等待消息队列中的消息。
- OSQPendAbort() 放弃任务中消息队列的等待。
- OSQPost () 向消息队列发送一则消息 FIFO。
- OSQPostFront () 向消息队列发送一则消息 LIFO。
- OSQPostOpt () 向消息队列发送一则消息 LIFO。
- OSQQuery () 获取一个消息队列的相关信息。

1.5.1 OSQAccept()

作用：无等待查看消息队列是否收到消息。

函数原型：

```
void   *OSQAccept (OS_EVENT   *pevent,
                   INT8U      *perr);
```

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。当建立消息邮箱时，该指针返回到用户程序。(参考 OSQCreate () 函数)。

perr： 指向一个用于保存错误代码的变量。可能为以下几种：

OS_ERR_NONE： 请求成功。

OS_ERR_EVENT_TYPE： pevent 不是指向一个消息队列。

OS_ERR_PEVENT_NULL：pevent 是一个空指针。
OS_ERR_PEND_ISR： 从 ISR 调用 OSMutexAccept(),不允许。
OS_ERR_PIP_LOWER： 拥有互斥任务的优先级比设置 PIP 优先级高。
OS_ERR_Q_EMPTY：队列不包含任何信息。

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：

必须先建立消息队列，然后使用。

范例1.5.1

```
OS_EVENT *CommQ;
void Task (void *p_arg)
{
    void *pmsg;
    (void)p_arg;
    while(1)
    {
        pmsg = OSQAccept(CommQ); /* Check queue for a message */
        if (pmsg != (void *)0)
        {
        . /* Message received, process */
        }
        else
        {
        . /* Message not received, do .. */
        . /* .. something else */
        }
        ...
    }
}
```

1.5.2 OSQCreate()

作用：建立并初始化一个消息队列。

函数原型：

```
OS_EVENT  *OSQCreate (void      **start,
                      INT16U     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。

范例1.5.2

```
OS_EVENT *CommQ;
void *CommMsg[10];
void main (void)
{
    OSInit(); /* Initialize μC/OS-II */
    ...
    CommQ = OSQCreate(&CommMsg[0], 10); /* Create COMM Q */
    ...
    OSStart(); /* Start Multitasking */
}
```

1.5.3 OSQDel ()

作用： 删除一个消息队列。

函数原型：

```
OS_EVENT   *OSQDel (OS_EVENT   *pevent,
                    INT8U       opt,
                    INT8U       *perr);
```

所属文件	调用者	开关量
OS_Q.C	任务	OS_Q_EN && OS_Q_DEL_EN

OSQDel()用于删除不再使用的消息队列，使用该函数时需非常谨慎，最好是先删除与该消息队列相关的所有任务，或者事先解除那些任务与该消息队列间的联系，不然会导致这些任务的失效。如果失效的任务是系统或应用中的关键任务，后果将不堪设想，因此，使用该函数时应当非常慎重。

如果删除该队列，所有的任务挂起的队列将准备就绪，因此你必须小心应用程序中队列用于互斥的情况，因为资源将不再由队列看守。

此调用可能会禁止中断很长一段时间。中断禁止时间成正比任务等待队列的数目。

使用这个函数，如果选项是 OS_DEL_ALWAYS，OSQAccept()函数并不知道预期的邮箱已被删除。

OSQPend()函数申请消息也会失败返回 OS_ERR_PEND_ABORT，所以必须要检查它的返回值,。

参数：

pevent： 指向消息队列的指针。当建立消息队列时，该指针返回到用户程序。（参考 OSQCreate ()函数）。

opt： 选择指定要删除消息邮箱类型：

OS_DEL_NO_PEND：只有当不存在悬而未决的任务删除时才删除，否则不删除。

OS_DEL_ALWAYS： 不管是否有任务等待消息都直接删除。

perr： 指向一个用于保存错误代码的变量。可能为以下几种：

OS_ERR_NONE： 消息队列删除成功。

OS_ERR_DEL_ISR： 试图从 ISR 删除一个消息队列。

OS_ERR_INVALID_OPT： 指定操作参数有误。

OS_ERR_TASK_WAITING：一个或多个任务正在等待消息队列同时指定了 OS_DEL_NO_PEND。

OS_ERR_EVENT_TYPE： pevent 不是指向一个消息队列。

深圳信盈达技术有限公司 未经允许禁止发布

OS_ERR_PEVENT_NULL：pevent 是一个空指针。

返回值：

如果消息队列被成功删除返回 NULL 指针，否则删除失败。在后一种情况下，需要检查错误代码以确定原因。

范例1.5.3

```
OS_EVENT *DispQ;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while (1)
    {
        ...
        DispQ = OSQDel(DispQ, OS_DEL_ALWAYS, &err);
        if (DispQ == (OS_EVENT *)0)
        {
          /* 消息队列删除成功 */
        }
        ...
    }
}
```

1.5.4 OSQFlush()

作用：清空消息队列。

函数原型：

INT8U *OSQFlush(OS_EVENT *pevent);

所属文件	调用者	开关量
OS_Q.C	任务	OS_Q_EN && OS_Q_FLUSH_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： 指向消息队列的指针。当建立消息队列时，该指针返回到用户程序。（参考 OSQCreate ()函数）。

返回值：

OS_ERR_NONE： 清空成功。

OS_ERR_EVENT_TYPE： pevent 不是指向一个消息队列。

OS_ERR_PEVENT_NULL：pevent 是一个空指针。

注意：

必须先创建消息队列后，才能使用。

范例1.5.4

```
OS_EVENT *CommQ;
void main (void)
```

```
{
    INT8U err;
    OSInit(); /* Initialize μC/OS-II */
    ...
    err = OSQFlush(CommQ);
    ...
    OSStart(); /* Start Multitasking */
}
```

1.5.5 OSQPend()

作用：挂起任务等待消息队列。

函数原型：

```
void *OSQPend( OS_EVENT *pevent,
               INT32U timeout,
               INT8U *perr);
```

所属文件	调用者	开关量
OS_Q.C	任务	OS_Q_EN

OSQPend()函数用于任务等待队列中的消息。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，uC/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： 指向消息队列的指针。当建立消息队列时，该指针返回到用户程序。(参考 OSQCreate ()函数)。

timeout：任务等待超时周期，为 0 时表示无限等待；其它，递减到 0 时任务恢复执行。

perr：指向错误代码变量的指针，其返回值如下：

OS_ERR_NONE	接收到消息。
OS_ERR_TIMEOUT	未在指定的超时周期内收到的消息。
OS_ERR_PEND_ABORT	等待中止。是由于另一个任务或通过 ISR 调用 OSQPendAbort()。
OS_ERR_EVENT_TYPE	pevent 不是指向一个消息邮箱。
OS_ERR_PEND_LOCKED	在调用本函数时，调度器被锁定。
OS_ERR_PEND_ISR	尝试从一个 ISR 调用 OSMboxPend ()，不允许。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。

返回值：

无。

注意：

必须先创建消息队列后，才能使用。

范例1.5.5

```
OS_EVENT *CommQ;
void CommTask(void *p_arg)
{
    INT8U err;
```

深圳信盈达技术有限公司 版权所有 未经许可 不得复制

```
    void *pmsg;
    (void)p_arg;
    while(1)
    {
        ...
        pmsg = OSQPend(CommQ, 100, &err);
        if (err == OS_ERR_NONE)
        {
          /* Message received within 100 ticks! */
        }
        else
        {
          /* Message not received, must have timed out */
        }
        ...
    }
}
```

1.5.6 OSQPendAbort ()

作用： 使正在等待该消息队列的任务取消挂起。

函数原型：

```
INT8U    OSQPendAbort (OS_EVENT    *pevent,
                       INT8U       opt,
                       INT8U       *perr);
```

所属文件	调用者	开关量
OS_Q.C	任务	OS_Q_EN && OS_Q_PEND_ABORT_EN

OSQPendAbort （）用来使正在等待该消息队列的任务取消挂起。取消任务挂起后，uC/OS-II 会自动执行调度。若应用中不需其在取消任务后实现调度，可在上述两种方式后或上操作宏 OS_OPT_POST_NO_SCHED。

参数：

pevent: 指向消息队列的指针。

opt: 指定要中止消息邮箱等待的类型。

OS_PEND_OPT_NONE： 只使正在等待该队列中消息的最高优先级任务取消挂起。

OS_PEND_OPT_BROADCAST： 使所有在等待该队列消息的任务取消挂起。

perr: 指向错误代码变量的指针，其返回值如下：

OS_ERR_NONE	没有任务在等待该消息队列。
OS_ERR_EVENT_TYPE	pevent 不是指向一个消息队列。
OS_ERR_PEND_ABORT	至少一个任务等待的消息队列已经退出等待。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。

返回值：

返回被本函数取消挂起的正在等待队列中消息的任务数。0 表示没有任务在等待队列中的消息，因此该功能没作用。

注意：

必须先创建消息队列后，才能使用。

范例 1.5.6

```
OS_EVENT *CommQ;
void CommTask(void *p_arg)
{
    INT8U err;
    INT8U nbr_tasks;
    (void)p_arg;
    while(1)
    {
        ...
        nbr_tasks = OSQPendAbort(CommQ, OS_PEND_OPT_BROADCAST, &err);
        if (err == OS_ERR_NONE)
        {
          /* 没有任务在等待该消息队列 */
        }
        else if(err == OS_ERR_PEND_ABORT)
        {
            /*所有在等待该消息队列的任务退出等待 */
        }
        ...
    }
}
```

1.5.7 OSQPost ()

作用：向邮箱发送一则消息(FIFO)。

函数原型：

```
INT8U   OSQPost (OS_EVENT   *pevent,
                 void       *pmsg);
```

所属文件	调用者	开关量
OS_Q.C	任务或中断	OS_Q_EN && OS_Q_POST_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： 指向消息队列的指针。

pmsg： 是即将发送给任务的消息。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：

OS_ERR_NONE	发送成功。
OS_ERR_Q_FULL	消息队列已满。
OS_ERR_EVENT_TYPE	pevent 不是指向消息队列的指针。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。
OS_ERR_POST_NULL_PTR	发送的内容为 NULL(空)，不允许。

注意：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1.5.7

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

void CommTaskRx(void *pdata)
{
        INT8U    err;
        pdata = pdata;
        for (;;) {
            err = OSQPost(CommQ, (void *)&CommRxBuf[0]);
            if (err == OS_ERR_NONE) {
                .                                   /*  将消息放入消息队列     */
            } else {
                .                                   /*  消息队列已满                    */
            }
        }
}
```

1.5.8 OSQPostFront()

作用：向邮箱发送一则消息(LIFO)。

函数原型：

```
INT8U    OSQPostFront (OS_EVENT    *pevent,
                       void            *pmsg);
```

所属文件	调用者	开关量
OS_Q.C	任务或中断	OS_Q_EN && OS_Q_POST_FRONT_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）。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: 指向消息队列的指针。

pmsg: 是即将发送给任务的消息。不允许传递一个空指针。

返回值:

OS_ERR_NONE	发送成功。
OS_ERR_Q_FULL	消息队列已满。
OS_ERR_EVENT_TYPE	pevent 不是指向消息队列的指针。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。
OS_ERR_POST_NULL_PTR	发送的内容为 NULL(空)，不允许。

注意: 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1.5.8

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

void CommTaskRx(void *pdata)
{
        INT8U    err;
        pdata = pdata;
        for (;;) {
            err = OSQPostFront (CommQ, (void *)&CommRxBuf[0]);
            if (err == OS_ERR_NONE) {
                .                                   /*  将消息放入消息队列     */
            } else {
                .                                   /*  消息队列已满                    */
            }
        }
}
```

1.5.9 OSMboxPostOpt()

作用: 按照规则向邮箱发送一则消息。

函数原型:

```
INT8U    OSQPostOpt (OS_EVENT    *pevent,
                     void            *pmsg,
                     INT8U           opt);
```

所属文件	调用者	开关量
OS_MBOX.C	任务或中断	OS_MBOX_EN && OS_MBOX_POST_OPT_EN

OSQPostOpt()用于通过一个队列发送一个消息给一个任务。消息是一个指针大小可变，并且其用途是应用特定的。如果消息队列已满，错误代码返回，表示队列已满。 OSQPostOpt()然后立即返回到其调用者，并且消息不放置在队列中。如果有任何任务是在队列中等待消息，OSQPostOpt()中既可以发布消息的优先级最高的任务队列(OPT 设置为 OS_POST_OPT_NONE)或所有任务在队列中等待(OPT 设置为 OS_POST_OPT_BROADCAST)等。OSQPostOpt()实际上可以代替 OSQPost()和 OSQPostFront()，因为你通过选项参数指定的操作模式，选择。这样做可以让您减少所需的μC/ OS-II 的代码空间。

参数：

pevent： 指向消息邮箱的指针。

pmsg： 是即将发送给任务的消息。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。

opt： 发送选项参数，有以下几种：

OS_POST_OPT_NONE POST： 到单个等待任务（跟 OSQPost()完全相同）。
OS_POST_OPT_BROADCAST：等待队列中的所有任务。
OS_POST_OPT_FRONT： POST 作为后进先出（模拟 OSQPostFront()）。
OS_POST_OPT_NO_SCHED： 调度程序将不会被调用。

下面是一些这些标志的可能的组合的列表：
OS_POST_OPT_NONE： 功能相同于 OSQPost()
OS_POST_OPT_FRONT： 功能相同于 OSQPostFront()
OS_POST_OPT_BROADCAST：功能相同于 OSQPost()，但 pmsg 广播给所有等待任务。
OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST：功能相同于 OSQPostFront()，除了广播 pmsg 给所有的等待任务。
OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST + OS_POST_OPT_NO_SCHED：功能相同于 OSQPostFront()，除了广播 pmsg 给所有的等待任务，调度程序将不会马上被调用。

返回值：

OS_ERR_NONE	发送成功。
OS_ERR_Q_FULL	消息队列已满。
OS_ERR_EVENT_TYPE	pevent 不是指向消息队列的指针。
OS_ERR_PEVENT_NULL	pevent 传入的是一个空指针。
OS_ERR_POST_NULL_PTR	发送的内容为 NULL(空)，不允许。

注意：

1) 在使用之前必须先创建一队列。
2) 如果需要使用此功能，并希望减少代码空间，可以禁止代码生成 OSQPost()和 OSQPostFront()。在 OS_CFG . H 设置 OS_Q_POST_EN 为 0、OS_Q_POST_FRONT_EN 为 0），因为 OSQPostOpt()可以模拟这两种功能。
3) OSQPostOpt() 的执行时间取决于任务在等待队列中的数量，如果你发送选项选择 OS_POST_OPT_BROADCAST。
4) 当同时提交多个消息时，可先失能提交后的调度，待到最后一个消息被提交时，再使能调度。从而避免了需提交几个消息就得执行几次调度的尴尬局面，提高了系统的实时性。

范例 1.5.9

```
OS_EVENT *CommQ;
INT8U CommRxBuf[100];
void CommRxTask (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while(1)
    {
        ...
        ...
        err = OSQPostOpt( CommQ,
```

深圳信盈达技术有限公司 版权所有

```
                                (void *)&CommRxBuf[0],
                                OS_POST_OPT_BROADCAST);
            ...
            ...
        }
    }
```

1.5.10 OSQQuery ()

作用：获取消息队列的相关信息。

函数原型：

```
INT8U    OSQQuery (OS_EVENT    *pevent,
                   OS_Q_DATA *p_q_data);
```

所属文件	调用者	开关量
OS_Q.C	任务或中断	OS_Q_EN && OS_Q_POST_EN

OSQQuery（）函数用来取得消息队列的相关信息。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： 指向消息队列的指针。

p_q_data：指向 OS_Q_DATA 的一个数据结构，它包含下列字段:

```
    void          *OSMsg;          /* Pointer to next message to be extracted from queue              */
    INT16U        OSNMsgs;    /* Number of messages in message queue        */
    INT16U        OSQSize;    /* Size of message queue        */
    OS_PRIO       OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur              */
    OS_PRIO       OSEventGrp;      /* Group corresponding to tasks waiting for event to occur        */
```

返回值：

OSQQuery()返回的这些错误代码之一:

OS_ERR_NONE： 获取成功。

OS_ERR_EVENT_TYPE：pevent 不是指向一个消息队列。

OS_ERR_PEVENT_NULL：pevent 是一个空指针。

OS_ERR_PDATA_NULL：p_q_data 是一个空指针。

注意：必须先创建消息队列，才能使用本函数。

范例1.5.10

```
OS_EVENT *CommQ;
void Task (void *p_arg)
{
    OS_Q_DATA qdata;
    INT8U err;
    (void)p_arg;
    while(1)
```

```
    {
        ...
        err = OSQQuery(CommQ, &qdata);
        if (err == OS_ERR_NONE)
        {
        . /* 'qdata' can be examined! */
        }
        ...
    }
}
```

1.6 互斥型信号量

互斥型信号量用于处理共享资源；由于终端硬件平台的某些实现特性，例如单片机管脚的复用，多个任务需要对硬件资源进行独占式访问。所谓独占式访问，指在任意时刻只能有一个任务访问和控制某个资源，而且必须等到该任务访问完成后释放该资源，其他任务才能对此资源进行访问。

操作系统进行任务切换时，可能被切换的低优先级任务正在对某个共享资源进行独占式访问，而任务切换后运行的高优先级任务需要使用此共享资源，此时会出现优先级反转的问题。即，高优先级的任务需要等待低优先级的任务继续运行直到释放该共享资源，高优先级的任务才可以获得共享资源继续运行。

可以在应用程序中利用互斥型信号量（mutex）解决优先级反转问题。互斥型信号量是二值信号量。由于 uC/OS-II 不支持多任务处于同一优先级，可以把占有 mutex 的低优先级任务的优先级提高到略高于等待 mutex 的高优先级任务的优先级。等到低优先级任务使用完共享资源后，调用 OSMutexPost() ，将低优先级任务的优先级恢复到原来的水平。

优先级反转问题发生于高优先级的任务需要使用某共享资源，而该资源已被一个低优先级的任务占用的情况。为了降解优先级反转，内核可以将低优先级任务的优先级提升到高于高优先级任务的优先级，直到低优先级的任务使用完占用的共享资源。**优先级继承优先级 PIP，设置为略高于最高优先级任务的优先级。**

- **相关的 API 函数：**
- OSMutexAccept () 无等待地获取互斥型信号量。
- OSMutexCreate () 建立并初始化一个互斥型信号量。
- OSMutexDel () 删除一个互斥型信号量。
- OSMutexPend () 阻塞任务等待一个互斥型信号量。
- OSMutexPost () 释放一个互斥型信号量。
- OSMutexQuery () 获取一个互斥型信号量的相关信息。

注意：所有服务只能用于任务与任务之间，不能用于任务与中断服务子程序之间。

1.6.1 OSMutexAccept ()

作用：无等待请求互斥型信号量。

函数原型：

```
BOOLEAN    OSMutexAccept (OS_EVENT    *pevent,
                          INT8U        *perr);
```

所属文件	调用者	开关量
OS_MUTEX.C	任务或中断	OS_MUTEX_EN

OSMutexAccept() 无等待请求互斥型信号量。

参数：

pevent： 指向要请求的互斥信号量的指针。当建立互斥信号量时，该指针返回到用户程序。（参考

OSMutexCreate()函数）。

perr： 指向一个用于保存错误代码的变量。可能为以下几种：

OS_ERR_NONE： 请求成功。

OS_ERR_EVENT_TYPE： pevent 不是指向一个互斥信号量。

OS_ERR_PEVENT_NULL：pevent 是一个空指针。

OS_ERR_PEND_ISR： 从 ISR 调用 OSMutexAccept(),不允许。

OS_ERR_PIP_LOWER： 拥有互斥任务的优先级比设置 PIP 优先级高。

返回值：

OS_TRUE：互斥信号量可用。

OS_FALSE：互斥信号量被另一个任务拥有，不可用。

注意：

1) 在使用之前必须先创建一互斥信号量。
2) 此功能不能被 ISR 调用。
3) 通过 OSMutexAccept()成功请求互斥量后，当共享资源使用完成时必须调用 OSMutexPost()来释放互斥锁。

范例1.6.1

```
OS_EVENT *DispMutex;
void Task (void *p_arg)
{
        INT8U err;
        BOOLEAN test;
        (void)p_arg;
        while(1)
        {
                test = OSMutexAccept(DispMutex, &err);
                if(test == OS_TRUE)
                {
                  /*  请求互斥信号量成功  */
                }
                else
                {
                   /*请求互斥信号量失败*/
                }
                ...
         }
}
```

安编制

1.6.2 OSMutexCreate()

作用：建立并初始化一个互斥型信号量。

函数原型：

```
OS_EVENT   *OSMutexCreate (INT8U     prio,
                           INT8U    *perr);
```

所属文件	调用者	开关量

OS_MUTEX.C	任务	OS_MUTEX_EN

OSMutexCreate () 建立并初始化一个互斥型信号量。互斥型信号量的使用与信号量的使用差不多，但还需要一个较高的空闲优先级，这个级别要比使用这个互斥型信号量的所有任务优先级都高（数字更小）。

参数：

prio： 较高的空闲优先级，用于任务提权。

perr： 指向一个用于保存错误代码的变量。可能为以下几种：

OS_ERR_NONE： 创建成功。

OS_ERR_PEVENT_NULL： pevent 是一个空指针。

OS_ERR_PRIO_EXIST： 指定的优先级已有任务存在。

OS_ERR_PEND_ISR： 从 ISR 调用 OSMutexCreate (),不允许。

OS_ERR_PRIO_INVALID： 指定的优先级比 OS_LOWEST_PRIO(系统最低优先级)高。

返回值：

一个指向分配给互斥事件控制块。如果没有事件控制块可用，OSMutexCreate（）返回 NULL 指针。

注意：

1) 在使用之前必须先创建一互斥。
2) 您必须确保 PRIO 具有比任何使用互斥访问资源的任务更高的优先级。例如，如果优先级 20，25 的三个任务，和 30 将要使用互斥，那么 PRIO 的值必须比 20 低，同时指定的值必须没有任务占用。

范例1.6.2

```
OS_EVENT *DispMutex;
void main (void)
{
    INT8U err;
    ...
    OSInit(); /* Initialize μC/OS-II */
    ...
    DispMutex = OSMutexCreate(18, &err); /* Create Display Mutex */
    ...
    OSStart(); /* Start Multitasking */
}
```

1.6.3 OSMutexDel ()

作用：删除一个互斥信号量。

函数原型：

```
OS_EVENT   *OSMutexDel (OS_EVENT    *pevent,
                        INT8U        opt,
                        INT8U       *perr);
```

所属文件	调用者	开关量
OS_MUTEX.C	任务	OS_MUTEX_EN && OS_MUTEX_DEL_EN

OSMutexDel()用来删除一个互斥信号量。使用这个函数是很危险的，因为多个任务可能是依赖于互斥信号量而运行的。所以在删除互斥信号量之前，必须先删除所有访问互斥信号量的任务。

参数:

pevent: 指向要请求的互斥信号量的指针。当建立互斥信号量时，该指针返回到用户程序。(参考OSMutexCreate()函数)。

opt: 选择指定要删除消息邮箱类型:

OS_DEL_NO_PEND: 只有当不存在悬而未决的任务删除时才删除，否则不删除。

OS_DEL_ALWAYS: 直接删除。

perr: 指向一个用于保存错误代码的变量。可能为以下几种:

OS_ERR_NONE: 互斥信号量删除成功。

OS_ERR_DEL_ISR: 试图从 ISR 删除一个互斥信号量。

OS_ERR_INVALID_OPT: 指定操作参数有误。

OS_ERR_TASK_WAITING: 一个或多个任务正在等待互斥信号量同时指定了 OS_DEL_NO_PEND。

OS_ERR_EVENT_TYPE: pevent 不是指向一个互斥信号量。

OS_ERR_PEVENT_NULL: pevent 是一个空指针。

返回值:

如果互斥锁被成功删除返回 NULL 指针，否则删除失败。在后一种情况下，需要检查错误代码以确定原因。

范例1.6.3

```
OS_EVENT *DispMutex;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while (1)
    {
        ...
        DispMutex = OSMutexDel(DispMutex, OS_DEL_ALWAYS, &err);
        if (DispMutex == (OS_EVENT *)0)
        {
          /* 互斥信号量删除成功*/
        }
        ...
  }
}
```

1.6.4 OSMutexPend()

作用: 阻塞任务等待互斥型信号量。

函数原型:

```
void   OSMutexPend (OS_EVENT   *pevent,
                    INT32U      timeout,
                    INT8U      *perr);
```

所属文件	调用者	开关量
OS_MUTEX.C	只能是任务	OS_MUTEX_EN

OSMutexPend()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 OSMutexPend()函数时，如果互斥信号量被其他任务持有，OSMutexPend()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。并且，该互斥信号号的任务的持有者会把自己的优先级提升到一个比较高的优先级（在创建互斥量时指定），来保证他不被其他任务中止，来确保在等待该信号的高优先级任务能尽快得到互斥信号。如果在预期的时钟节拍内信号量被置起，uC/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互斥信号量，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： 指向要请求的互斥信号量的指针。当建立互斥信号量时，该指针返回到用户程序。（参考 OSMutexCreate()函数）。

perr： 指向一个用于保存错误代码的变量。可能为以下几种：

OS_ERR_NONE： 请求成功。
OS_ERR_TIMEOUT： 互斥量不在指定的超时时间内可用。
OS_ERR_PEND_ABORT： OSMutexPend()被另一个任务中止。
OS_ERR_PEND_LOCKED：在调用本函数时，调度器被锁定。
OS_ERR_EVENT_TYPE： pevent 不是指向一个互斥信号量。
OS_ERR_PEVENT_NULL：pevent 是一个空指针。
OS_ERR_PEND_ISR： 从 ISR 调用 OSMutexPend ()，不允许。
OS_ERR_PIP_LOWER： 拥有互斥任务的优先级比设置 PIP 优先级高。

返回值：无。

注意：必须先建立互斥信号量，然后再使用。不允许从中断调用该函数。

范例1.6.4

```
OS_EVENT *DispMutex;
void DispTask (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while(1)
    {
        ...
        OSMutexPend(DispMutex, 0, &err);/*申请互斥信号量*/
        ...
        /*共享设备（资源）代码；*/
        ...
        OSMutexPost(DispMutex);/*释放互斥信号量*/
        ...
    }
}
```

1.6.5 OSMutexPost ()

作用：释放一个互斥型信号量。

函数原型：

```
INT8U    OSMutexPost (OS_EVENT *pevent);
```

陈志发编制

所属文件	调用者	开关量
OS_MUTEX.C	任务或中断	OS_MUTEX_EN

OSMutexPost ()释放一个互斥型信号量。

参数:

pevent: 指向一个互斥型信号量的指针。

返回值:

OS_ERR_NONE: 指定互斥信号量被释放。
OS_ERR_EVENT_TYPE: pevent 不是指向一个互斥信号量。
OS_ERR_PEVENT_NULL: pevent 传入的是一个空指针。
OS_ERR_PEND_ISR: 从 ISR 调用 OSMutexPend ()，不允许。
OS_ERR_PIP_LOWER: 拥有互斥任务的优先级比设置 PIP 优先级高。
OS_ERR_NOT_MUTEX_OWNER: 互斥信号量并没有被占用。

注意: 必须先建立互斥信号量，然后再使用。不允许从中断调用该函数。

范例1.6.5

```
OS_EVENT *DispMutex;
void TaskX (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while(1)
    {
        ...
        err = OSMutexPost(DispMutex);
        if(err == OS_ERR_NONE)
        {
            /*释放互斥信号量成功*/
        }
        ...
    }
}
```

1.6.6 OSMutexQuery ()

作用: 获取一个互斥信号量的相关信息。

函数原型:

```
INT8U   OSMutexQuery (OS_EVENT          *pevent,
                      OS_MUTEX_DATA     *p_mutex_data);
```

所属文件	调用者	开关量
OS_MUTEX.C	任务	OS_MUTEX_EN

OSMutexQuery (获取一个互斥信号量的相关信息。

参数:

pevent： 指向一个互斥型信号量的指针。

p_mutex_data： 存放查询到的状态信息。

```
typedef struct os_mutex_data {
        /* List of tasks waiting for event to occur                    */
        OS_PRIO OSEventTbl[OS_EVENT_TBL_SIZE];

        /* Group corresponding to tasks waiting for event to occur */
        OS_PRIO OSEventGrp;

        /* Mutex value (OS_FALSE = used, OS_TRUE = available)          */
        BOOLEAN OSValue;
        INT8U      OSOwnerPrio;

        /* Mutex owner's task priority or 0xFF if no owner              */
        INT8U      OSMutexPCP;

        /* Priority Ceiling Priority or 0xFF if PCP disabled            */
} OS_MUTEX_DATA;
```

返回值：

OS_ERR_NONE：	获取成功。
OS_ERR_EVENT_TYPE：	pevent 不是指向一个互斥信号量。
OS_ERR_PEVENT_NULL：	pevent 传入的是一个空指针。
OS_ERR_QUERY_ISR：	从 ISR 调用 OSMutexQuery ()，不允许。
OS_ERR_PDATA_NULL：	p_mutex_data 传入的是一个空指针。

注意： 必须先建立互斥信号量，然后再使用。不允许从中断调用该函数。

范例1.6.6

```
OS_EVENT *DispMutex;
void Task (void *p_arg)
{
        OS_MUTEX_DATA mutex_data;
        INT8U err;
        INT8U highest; /* Highest priority task waiting on mutex */
        INT8U x;
        INT8U y;
        (void)p_arg;
        while(1)
        {
                ...
                err = OSMutexQuery(DispMutex, &mutex_data);
                if (err == OS_ERR_NONE)
                {
                        /* Examine Mutex data */
                }
                ...
        }
}
```

陈志发编制

1.7 事件标志组

在信号量和互斥信号量的管理中，任务请求资源，如果资源未被占用就继续需运行，否则只能阻塞，等待资源释放事件的发生。这种事件是单一的事件，如果任务要等待多个事件的发生，或者多个事件中某一个事件的发生就可以继续运行，那么就应该采用事件标志组管理。事件标志组管理的条件组合可以是多个事件都发生(与关系)，也可以是多个事件中有任何一个事件发生（或关系）。

事件标志组由 2 部分组成：一是保存各事件状态的标志位；二是等待这些标志位置位或清除的任务列表。可以用 8 位、16 位或 32 位的序列表示事件标志组，每一位表示一个事件的发生。要使系统支持事件标志组功能，需要在 OS_CFG.H 文件中打开 OS_FLAG_EN 选项。

- **相关的 API 函数：**
- OSFlagAccept() 无等待检查事件标志组的状态。
- OSFlagCreate() 建立一个事件标志组。
- OSFlagDel() 删除一个事件标志组。
- OSFlagPend() 挂起任务等待事件标志组的事件标志位。
- OSFlagPost() 置位或清 0 事件标志组中的标志位。
- OSFlagQuery() 获取一组事件标志的当前值。
- OSFlagNameGet() 获取事件标志组名称。
- OSFlagNameSet() 设置事件标志组名称。
- OSFlagPendGetFlagsRdy() 获取使任务就绪的标志。

1.7.1 OSFlagAccept()

作用：无等待检查事件标志组的状态。

函数原型：

```
OS_FLAGS OSFlagAccept(OS_FLAG_GRP   *pgrp,
                      OS_FLAGS      flags,
                      INT8U         wait_type,
                      INT8U         *perr);
```

所属文件	调用者	开关量
OS_FLAG.C	任务或中断	OS_FLAG_EN && OS_FLAG_ACCEPT_EN

OSFlagAccept（）函数查看指定的事件标志组的组合状态。OSFlagAccept（）函数并不挂起任务。

参数：

pgrp: 指向事件标志组的指针。

flags: 期望的标志位组合值，如：0x80,0x11 等等。

wait_type： 指定的检查类型，有以下几种选项：

OS_FLAG_WAIT_CLR_ALL(AND)：期望的所有的事件标志位都清零时（与关系）有效。

OS_FLAG_WAIT_CLR_ANY(OR)： 期望的所有的事件标志位有一个清零时（或关系）有效。

OS_FLAG_WAIT_SET_ALL(AND)：期望的所有的事件标志位都置位时（与关系）有效。

OS_FLAG_WAIT_SET_ANY(OR)： 期望的所有的事件标志位有一个置位时（或关系）有效。

perr: 指向错误代码变量的指针，其返回值如下：

OS_ERR_NONE 成功获取到指定标志。

OS_ERR_EVENT_TYPE pgrp 不是指向一个事件标志组。

OS_ERR_FLAG_WAIT_TYPE 没有指定一个合适的 wait_type 类型。

OS_ERR_FLAG_INVALID_PGRP pgrp 传入的是一个空指针，可能事件标志组没有被创建。

深圳信盈达陈志发编制

OS_ERR_FLAG_NOT_RDY 等待所需的标志不可用。

返回值:

在 OSFlagAccept（）调用成功时返回事件标志组当前的标志。

注意:

本函数在成功获取到期望的标志位后，并不会清掉或设置相关标志位。如果想在成功获取到期望的标志位后清除相关标志位，则可以在原有 wait_type 类型的基础上加上 OS_FLAG_CONSUME 项。如：(OS_FLAG_WAIT_SET_ANY | OS_FLAG_CONSUME)。

范例 1.7.7

```
/*LED2 任务*/
void led2_task(void *pdata)
{
    u8 err;
    OS_FLAGS value;
    pdata=pdata;/*防止编译器优化警告*/
    while(1)
    {
        /*查询标志*/
        value = OSFlagAccept(FlagStat,(1<<2) | (1<<1), OS_FLAG_WAIT_SET_AND,&err);
        if(err == OS_ERR_NONE)/*成功获取标志*/
        {
            LED2=0;
            OSTimeDly(60);/*延时 60 个时钟节拍*/
            LED2=1;
            OSTimeDly(100);/*延时 100 个时钟节拍*/
        }
        sprintf((char *)str,"value: 0x%x",value);
        lcd_show_str(35,60,240,16,str,16,RED,WHITE,0);/*实时显示 value 值*/
        OSTimeDly(20);/*延时 20 个时钟节拍*/
    }
}
```

1.7.2 OSFlagCreate()

作用: 建立一个事件标志组。

函数原型:

```
OS_FLAG_GRP  *OSFlagCreate  (OS_FLAGS flags,
                             INT8U     *perr);
```

所属文件	调用者	开关量
OS_FLAG.C	任务或启动代码	OS_FLAG_EN

OSFlagCreate ()用于创建和初始化一个事件标志组。

参数:

flags: 包含在事件标志组中存储的初始值。

perr: 指向错误代码变量的指针，其返回值如下:

OS_ERR_NONE 事件标志组创建成功。

OS_ERR_CREATE_ISR 程序尝试从一个 ISR 创建事件标志组。

OS_ERR_FLAG_GRP_DEPLETED 没有更多可用的事件标志组，此时需要在 os_cfg.H 增加 os_max_flags 数量以解决本错误。

返回值：

指向分配给所建立的事件标志组的事件标志控制块的指针。如果没有可用的事件标志控制块，返回空指针。

范例1.7.2

```
OS_FLAG_GRP    *EngineStatus;/*创建事件标志组指针*/
void main (void)
{
    INT8U err;/*存放错误代码变量*/
    ...
    OSInit(); /*  初始化   μC/OS-II */
    ...
    ...
    /*  创建一个事件标志组并初始化  */
    EngineStatus = OSFlagCreate(0x00, &err);
    ...
    ...
    OSStart(); /*启动多任务*/
}
```

1.7.3 OSFlagDel()

作用： 删除一个事件标志组

函数原型：

```
OS_FLAG_GRP    *OSFlagDel    (OS_FLAG_GRP    *pgrp,
                              INT8U           opt,
                              INT8U           *perr);
```

所属文件	调用者	开关量
OS_FLAG.C	任务	OS_FLAG_EN && OS_FLAG_DEL_EN

OSFlagDel ()用于删除一个事件标志组。使用这个函数是很危险的，因为多个任务可能是依赖于事件标志组而运行的。所以在删除事件标志组之前，必须先删除所有访问事件标志组的任务。

参数：

pgrp: 指向事件标志组的指针。

opt: 选择指定要删除事件标志组类型：

OS_DEL_NO_PEND：只有当不存在因等待事件标志组而挂起的任务删除时才删除，否则不删除。

OS_DEL_ALWAYS： 直接删除。

perr: 指向错误代码变量的指针，其返回值如下：

OS_ERR_NONE 事件标志组已经删除。

OS_ERR_DEL_ISR	尝试从一个 ISR 删除一个事件标志组。
OS_ERR_FLAG_INVALID_PGRP	PGRP 传入的是一个空指针。
OS_ERR_EVENT_TYPE	PGRP 不是指向一个事件标志组
OS_ERR_INVALID_OPT	传入选项参数(opt)有误。
OS_ERR_TASK_WAITING	一个或多个任务正在等待事件标志组量。

返回值:

如果事件标志组被成功删除，则返回空指针；否则需要根据错误代码进行检查。

注意:

1) 必须先创建邮箱在使用本函数。
2) 不允许在中断函数中调用本函数。

范例1.7.3

```
OS_FLAG_GRP *EngineStatusFlags;
void Task (void *p_arg)
{
    INT8U err;
    OS_FLAG_GRP *pgrp;
    (void)p_arg;
    while (1)
    {
        ...
        ...
        pgrp = OSFlagDel(EngineStatusFlags, OS_DEL_ALWAYS, &err);
        if (pgrp == (OS_FLAG_GRP *)0)
        {
            /* 事件标志组已成功删除*/
        }
        ...
        ...
    }
}
```

1.7.4 OSFlagPend()

作用：挂起任务获取指定组合的事件标志位。

函数原型：

```
OS_FLAGS    OSFlagPend (OS_FLAG_GRP *pgrp,
                        OS_FLAGS      flags,
                        INT8U         wait_type,
                        INT32U        timeout,
                        INT8U        *perr);
```

所属文件	调用者	开关量
OS_FLAG.C	只能是任务	OS_FLAG_EN

OSFlagPend（）用来在任务中等待事件标志组中条件的组合（置位或清零）。如果调用任务的条件不可用，

那么调用本函数的任务将被挂起，直到所需的条件满足或指定的时间超时任务才恢复执行。

参数：

pgrp： 指向事件标志组的指针。

flags： 包含在事件标志组中存储的初始值。

wait_type： 指定的检查类型，有以下几种选项：

OS_FLAG_WAIT_CLR_ALL(AND)：期望的所有的事件标志位都清零时（与关系）有效。

OS_FLAG_WAIT_CLR_ANY(OR)： 期望的所有的事件标志位有一个清零时（或关系）有效。

OS_FLAG_WAIT_SET_ALL(AND)：期望的所有的事件标志位都置位时（与关系）有效。

OS_FLAG_WAIT_SET_ANY(OR)： 期望的所有的事件标志位有一个置位时（或关系）有效。

timeout： 任务等待超时值，为 0 时表示无限等待；其它，递减到 0 时任务恢复执行。

perr： 指向错误代码变量的指针，其返回值如下：

OS_ERR_NONE	成功获取到指定标志。
OS_ERR_EVENT_TYPE	pgrp 不是指向一个事件标志组。
OS_ERR_FLAG_WAIT_TYPE	没有指定一个合适的 wait_type 类型。
OS_ERR_FLAG_INVALID_PGRP	pgrp 传入的是一个空指针，可能事件标志组没有被创建。
OS_ERR_TIMEOUT	未在指定的超时周期内获取到期望标志位。
OS_ERR_PEND_ISR	尝试从一个 ISR 调用 OSFlagPend()，不允许。

返回值：

使任务准备就绪的标志（或 0），如果没有一个标志是准备好的，或是出现错误。

注意：

1) 必须先创建邮箱在使用本函数。
2) 不允许在中断函数中调用本函数。
3) 本函数在成功获取到期望的标志位后，并不会清掉或设置相关标志位。如果想在成功获取到期望的标志位后清除相关标志位，则可以在原有 wait_type 类型的基础上加上 OS_FLAG_CONSUME 项。如：(OS_FLAG_WAIT_CLR_AND | OS_FLAG_CONSUME)。

范例 1.7.4

```
/*LED3 任务*/
void led3_task(void *pdata)
{
    u8 err;
    pdata=pdata;/*防止编译器优化警告*/
    while(1)
    {
        OSFlagPend(FlagStat,0x10 | 0x02, OS_FLAG_WAIT_SET_AND,0,&err);
        LED3=0;
        OSTimeDly(50);/*延时 50 个时钟节拍*/
        LED3=1;
        OSTimeDly(120);/*延时 120 个时钟节拍*/
    }
}
```

1.7.5 OSFlagPost()

作用： 置位或清 0 事件标志组中的标志位

函数原型：

```
OS_FLAGS  OSFlagPost (OS_FLAG_GRP  *pgrp,
                      OS_FLAGS      flags,
                      INT8U         opt,
                      INT8U         *perr);
```

所属文件	调用者	开关量
OS_FLAG.C	任务或中断	OS_FLAG_EN

OSFlagPost（）根据位掩码（flags）置位或者清零事件标志组中的对应位。

参数:

pgrp: 指向事件标志组的指针。

flags: 包含在事件标志组中存储的初始值。

opt: 指定的操作类型，有以下 2 种:

OS_FLAG_SET: 置位。

OS_FLAG_CLR: 清零。

perr: 指向错误代码变量的指针，其返回值如下:

OS_ERR_NONE	设置成功
OS_ERR_FLAG_INVALID_PGRP	pgrp 传入的是一个空指针，可能事件标志组没有被创建。
OS_ERR_EVENT_TYPE	pgrp 不是指向一个事件标志组。
OS_ERR_FLAG_INVALID_OPT	opt 指定了一个无效的选项。

返回值:

事件标志组中新的标志状态。

范例 1.7.5

```
/*按键任务*/
void key_task(void *pada)
{
u8 kdat,err;
    (void)pada;/*防止编译器警告*/
     while(1)
     {
              kdat =   KEY_Scan(0);
              switch(kdat)
              {
                   case Up_KEY:      OSFlagPost(FlagStat, 0xffff, OS_FLAG_CLR, &err);/*清零所有位*/
                                     break;
                   case Down_KEY:    OSFlagPost(FlagStat, (1<<1), OS_FLAG_SET, &err);/*第 1 位置位*/
                                     break;
                   case Left_KEY:    OSFlagPost(FlagStat, (1<<2), OS_FLAG_SET, &err); /*第 2 位置位*/
                                     break;
                   case Right_KEY:   OSFlagPost(FlagStat, (1<<3), OS_FLAG_SET, &err); /*第 3 位置位*/
                                     break;
                   default:
                                     break;
              }
          OSTimeDly(20);/*延时 20 个时钟节拍*/
        }
}
```

修志发编制

1.7.6 OSFlagQuery()

作用：获取一组事件标志的当前值。

函数原型：

```
OS_FLAGS    OSFlagQuery (OS_FLAG_GRP    *pgrp,
                         INT8U          *perr);
```

所属文件	调用者	开关量
OS_FLAG.C	任务或中断	OS_FLAG_EN && OS_FLAG_QUERY_EN

OSFlagQuery（）被用于获得一组事件标志的当前值。此时，这个函数不返回任务等待事件标志组的列表。

参数：

pgrp: 指向事件标志组的指针。

perr: 指向错误代码变量的指针，其返回值如下：

OS_ERR_NONE 获取成功。

OS_ERR_FLAG_INVALID_PGRP pgrp 传入的是一个空指针，可能事件标志组没有被创建。

OS_ERR_EVENT_TYPE pgrp 不是指向一个事件标志组。

返回值：

事件标志组中的标志状态。

范例1.7.6

```
OS_FLAG_GRP *EngineStatusFlags;
void Task (void *p_arg)
{
    OS_FLAGS flags;
    INT8U err;
    (void)p_arg;
    while(1)
    {
        ...
        ...
        flags = OSFlagQuery(EngineStatusFlags, &err);/*获取当前事件标志组标志位*/
        ...
        ...
    }
}
```

1.7.7 OSFlagNameGet()

作用：获取事件标志组名称。

函数原型：

```
INT8U    OSFlagNameGet    (OS_FLAG_GRP    *pgrp,
                           INT8U          **pname,
                           INT8U          *perr);
```

所属文件	调用者	开关量

OS_FLAG.C	任务或中断	OS_FLAG_NAME_EN

OSFlagNameGet（）可以获取你分配给事件标志组的名称。

参数:

pgrp: 指向事件标志组的指针。

pname: 是一个指向指针的指针的事件标志组的名称。

perr: 指向错误代码变量的指针，其返回值如下:

OS_ERR_NONE	获取成功
OS_ERR_FLAG_INVALID_PGRP	pgrp 传入的是一个空指针，可能事件标志组没有被创建。
OS_ERR_PNAME_NULL	pname 指向 NULL。
OS_ERR_EVENT_TYPE	pgrp 不是指向一个事件标志组。

返回值:

指向 PNAME 的 ASCII 字符串的大小；如果为 0，说明遇到错误。

范例1.7.7

```
INT8U *EngineStatusName;              /*事件标志组名称指针*/
OS_FLAG_GRP *EngineStatusFlags;   /*事件标志组指针*/
void Task (void *p_arg)
{
    INT8U err;
    INT8U size;
    (void)p_arg;
    while(1)
    {
        size = OSFlagNameGet(EngineStatusFlags,
                             &EngineStatusName,
                             &err); /*获取事件标志组名称*/
        ...
        ...
    }
}
```

1.1.8 OSFlagNameSet()

作用：设置事件标志组名称。

函数原型：

```
void  OSFlagNameSet   (OS_FLAG_GRP   *pgrp,
                       INT8U         *pname,
                       INT8U         *perr);
```

所属文件	调用者	开关量
OS_FLAG.C	任务	OS_FLAG_NAME_EN

OSFlagNameSet（）可以设置事件标志组的名称。

参数:

pgrp: 指向事件标志组的指针。

pname: 指向包含该事件标志组的名称的 ASCII 字符串。

perr: 指向错误代码变量的指针，其返回值如下:

OS_ERR_NONE	设置成功
OS_ERR_FLAG_INVALID_PGRP	pgrp 传入的是一个空指针，可能事件标志组没有被创建。
OS_ERR_PNAME_NULL	pname 指向 NULL。
OS_ERR_EVENT_TYPE	pgrp 不是指向一个事件标志组。
OS_ERR_NAME_SET_ISR	在 ISR 使用本函数，不允许。

返回值: 无

范例1.7.8

```
OS_FLAG_GRP *EngineStatus; /*事件标志组指针*/
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while(1)
    {
        /*设置事件标志组名称*/
        OSFlagNameSet(EngineStatus, “Engine Status Flags”, &err);
        ...
        ...
    }
}
```

1.1.9 OSFlagPendGetFlagsRdy()

作用: 获取当前使任务就绪的标志。

函数原型:

```
OS_FLAGS    OSFlagPendGetFlagsRdy (void);
```

所属文件	调用者	开关量
OS_FLAG.C	只能是任务	OS_FLAG_EN

OSFlagPendGetFlagsRdy（）用于获取当前使任务就绪的标志。

参数: 无。

返回值: 使任务就绪的标志。

范例1.7.9

```
/*LED3 任务*/
void led3_task(void *pdata)
{
    u8 err;
    OS_FLAGS flags;
    (void)pdata;/*防止编译器优化警告*/
    while(1)
    {
      OSFlagPend(FlagStat,(1<<4) | (1<<1), OS_FLAG_WAIT_SET_AND,100,&err);
```

```
        if(err == OS_ERR_NONE)/*成功获取到期望的标志*/
        {
                flags = OSFlagPendGetFlagsRdy(); /*获取使任务就绪的标志*/
                printf("FlagsRdy: 0x%x",flags);
                LED3=0;
                OSTimeDly(50);/*延时 50 个时钟节拍*/
                LED3=1;
                OSTimeDly(120);/*延时 120 个时钟节拍*/
        }
    }
}
```

1.8 软件定时器

uCOSII 软件定时器定义了一个单独的计数器 OSTmrTime，用于软件定时器的计时，UCOSII 并不在 OSTimTick 中进行软件定时器的到时判断与处理， 而是创建了一个高于应用程序中所有其他任务优先级的定时器管理任务 OSTmr_Task，在这个任务中进行定时器的到时判断和处理。时钟节拍函数通过信号量给这个高优先级任务发信号。这种方法缩短了中断服务程序的执行时间，但也使得定时器到时处理函数的响应受到中断退出时恢复现场和任务切换的影响。软件定时器功能实现代码存放在 tmr．c 文件中，移植时需只需在 os_cfg.h 文件中使能定时器和设定定时器的相关参数。

UCOSII 中软件定时器的实现方法是，将定时器按定时时间分组，使得每次时钟节拍到来时只对部分定时器进行比较操作，缩短了每次处理的时间。但这就需要动态地维护一个定时器组。定时器组的维护只是在每次定时器到时时才发生，而且定时器从组中移除和再插入操作不需要排序。这是一种比较高效的算法，减少了维护所需的操作时间。

- **相关的 API 函数:**
  - OSTmrCreate() 创建软件定时器。
  - OSTmrDel() 删除软件定时器。
  - OSTmrNameGet() 获取软件定时器名称。
  - OSTmrRemainGet() 获取剩余时间。
  - OSTmrSignal() 更新计时器。
  - OSTmrStart() 开始计时。
  - OSTmrStateGet() 获取定时器当前状态。
  - OSTmrStop() 停止计时。

1.8.1 OSTmrCreate ()

作用: 创建软件定时器。

函数原型:

```
OS_TMR *OSTmrCreate(INT32U dly,
                    INT32U period,
                    INT8U opt,
                    OS_TMR_CALLBACK callback,
                    void *callback_arg,
                    INT8U *pname,
                    INT8U *perr);
```

所属文件	调用者	开关量
OS_TMR.C	任务或初始化代码	OS_TMR_EN

OSTmrCreate()用于创建一个定时器。定时器可以被配置为**周期性**运行或**单次**运行。当倒计时（设定的定时值）到 0，可选的“回调”函数会被执行。回调函数用于发信号告诉任务定时器时间到或者执行任何其他功能。但是建议保持回调函数可能短。使用时必须调用 OSTmrStart()来启动计时器。如果配置了定时器单次计时模式，定时时间到，需要调用 OSTmrStart()来重新触发定时器。如果不打算重新触发它，或者不使用定时器了，可以调用 OSTmrDel（）来删除的定时器。如果使用单次计时模式，可以在回调函数里调用 OSTmrDel（）删除定时器。使用软件定时器时，必须指定本任务的优先级（**OS_TASK_TMR_PRIO**），一般设置为最高优先级。

参数：

dly：指定使用的计时器初始延迟时间。

- 在单次触发模式时，这是单次定时的时间。
- 在周期模式下，这是最初定时器进入周期模式之前的延时。
- 延时单位取决于你多久调用一次 OSTmrSignal（）。如果 OS_TMR_CFG_TICKS_PER_SEC 设置为 10，则 DLY 指定延迟时间为 1/10 秒（也就是 100ms）。OS_TMR_CFG_TICKS_PER_SEC 设置为多少，就是 **1/OS_TMR_CFG_TICKS_PER_SEC** 秒。需要注意的是在创建时的定时器不能启动。

period：指定计时器周期性定时的时间量。如果设置为 0，那么将使用单次模式。

opt：

OS_TMR_OPT_PERIODIC：指定周期性定时模式。

OS_TMR_OPT_ONE_SHOT：指定单次定时模式。

请注意，必须选择其中一个选项。

callback：

- 指定一个回调函数的地址（可选），这个函数将被调用，处理定时结束的相关事项，回调函数必须声明如下：

  void MyCallback (void *ptmr, void *callback_arg);
- 如果不需要回调函数，可以传入一个空指针。

callback_arg：是传递给回调函数参数，如果回调函数不需要参数，可以是一个空指针。

pname：指向一个 ASCII 字符串，用于设置定时器名称。通过调用 ostmrnameget()获取这个名称。

perr： 指向存放错误代码的指针，并且可以是以下之一：

- OS_ERR_NONE： 定时器已成功创建。
- OS_ERR_TMR_INVALID_DLY： 在单次延时模式下指定了 0 延时。
- OS_ERR_TMR_INVALID_PERIOD： 在周期性延时模式下指定了 0 延时。。
- OS_ERR_TMR_INVALID_OPT： 指定的延时模式有误。
- OS_ERR_TMR_ISR： 在中断函数中调用本函数，不允许。
- OS_ERR_TMR_NON_AVAIL： 当你不能启动一个计时器时，你会得到这个错误，因为所有的定时器元素（即对象）已经被分配。
- OS_ERR_TMR_NAME_TOO_LONG： 设置的定时器的名字太长，必须比 OS_TMR_CFG_NAME_SIZE 小。

返回值：

返回指向分配给所建立的定时器的时间控制块的指针。如果没有可用的时间控制块，返回空指针。

注意：

1) 建议检查返回值，以确保计时器成功创建。
2) 不能从一个中断服务程序调用这个函数。
3) 请注意，在创建时不启动计时器。开始计时必须调用 ostmrstart()启动计时器。

深圳信盈达技术有限公司 未经允许不得复制

范例1.8.1

```
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    for (;;)
    {
        CloseDoorTmr = OSTmrCreate( 10,
                                    100,
                                    OS_TMR_OPT_PERIODIC,
                                    DoorCloseFnct,
                                    (void *)0,
                                    “Door Close”,
                                    &err);
        if (err == OS_ERR_NONE)
        {
            /* 定时器创建成功，但还没启动*/
        }
    }
}
```

1.8.2 OSTmrDel()

作用：删除软件定时器。

函数原型：

BOOLEAN OSTmrDel(OS_TMR *ptmr,
INT8U *perr);

所属文件	调用者	开关量
OS_TMR.C	任务	OS_TMR_EN

OSTmrDel()允许你删除一个定时器。如果一个计时器正在运行，它先会被停止，然后被删除。如果定时器已经到了，并且停止不再计时，那么将直接被删除。它是由你来删除未使用的计时器。如果删除一个计时器，你必须不能再使用它。

参数：

ptmr：指向想要删除的定时器。该指针的值为在创建定时器时返回的值（见 OSTmrCreate（））。

perr： 指向存放错误代码的变量。其值为以下几种:

OS_ERR_NONE: 计时器被成功删除。
OS_ERR_TMR_INVALID: ptmr 传入一个空指针。
OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向一个计时器。
OS_ERR_TMR_ISR: 在 ISR 中调用本函数，不允许。
OS_ERR_TMR_INACTIVE: ptmr 指向一个已被删除或没有创建一个定时器。

返回值：

OS_TRUE： 定时器已删除。
OS_FALSE：出现错误。

注意：

1) 建议检查返回值，以确本函数的执行结构是有效的。
2) 不能调用从 ISR 本函数。
3) 如果你删除了一个定时器，必须不能再引用它。

范例1.8.2

```
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    for (;;)
    {
        CloseDoorTmr = OSTmrDel(CloseDoorTmr,
                                &err);
        if (err == OS_ERR_NONE)
        {
            /* Timer was deleted ... DO NOT reference it anymore! */
        }
    }
}
```

1.8.3 OSTmrNameGet()

作用：获取软件定时器名称。
函数原型：

```
INT8U OSTmrNameGet(OS_TMR *ptmr,
                   INT8U **pname,
                   INT8U *perr);
```

所属文件	调用者	开关量
OS_TMR.C	任务	OS_TMR_EN && OS_TMR_CFG_NAME_EN

OSTmrNameGet()用于获取定时器的名称。

参数：

ptmr：指向想要获取名称的定时器。该指针的值为在创建定时器时返回的值（见 OSTmrCreate（））。
pname：指向一个要存放 ASCII 字符串内存区的首地址。
perr： 指向存放错误代码的变量。其值为以下几种：

OS_ERR_NONE： 获取成功。
OS_ERR_TMR_INVALID_DEST： pname 传入一个空指针。
OS_ERR_TMR_INVALID： ptmr 传入一个空指针。
OS_ERR_TMR_INVALID_TYPE： ptmr 不是指向一个计时器。

OS_ERR_NAME_GET_ISR: 在 ISR 中调用本函数，不允许。
OS_ERR_TMR_INACTIVE: ptmr 指向一个已被删除或没有创建一个定时器。

返回值: 计数器名称的字符长度。

注意:

1) 建议检查函数的返回值。
2) 不能在中断函数调用本函数。

范例1.8.3

```
INT8U *CloseDoorTmrName;
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    for (;;)
    {
        OSTmrNameGet(CloseDoorTmr, &CloseDoorTmrName, &err);
        if (err == OS_ERR_NONE)
        {
          /* CloseDoorTmrName points to the name of the timer */
        }
    }
}
```

1.8.4 OSTmrRemainGet()

作用: 获取定时器的剩余时间。

函数原型:

```
INT32U OSTmrRemainGet(OS_TMR *ptmr,
                      INT8U *perr);
```

所属文件	调用者	开关量
OS_TMR.C	任务	OS_TMR_EN

OSTmrRemainGet（）可以获取指定定时器的剩余（定时）的时间。该返回值表示的时间取决于 OS_TMR_CFG_TICKS_PER_SEC 的配置。如果 OS_TMR_CFG_TICKS_PER_SEC 设置为 10，那么返回的剩下的真实时间为：**返回时间*100ms**。如果定时器超时，返回值为 0。

参数:

ptmr: 指向想要获取剩余时间的定时器。该指针的值为在创建定时器时返回的值（见 OSTmrCreate（））。

perr: 指向存放错误代码的变量。其值为以下几种:

OS_ERR_NONE: 获取成功。
OS_ERR_TMR_INVALID: ptmr 传入一个空指针。
OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向一个计时器。
OS_ERR_NAME_GET_ISR: 在 ISR 中调用本函数，不允许。
OS_ERR_TMR_INACTIVE: ptmr 指向一个已被删除或没有创建一个定时器。

返回值：返回指定的计时器的剩余时间。如果指定了无效计时器或者定时器时间已到，返回的值将是 0。

注意：

1) 建议检查函数的返回值。
2) 不能在中断函数调用本函数。

范例 2.9.4

```
INT32U TimeRemainToCloseDoor;
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    for (;;)
    {
        TimeRemainToCloseDoor = OSTmrRemainGet(CloseDoorTmr, &err);
        if (err == OS_ERR_NONE)
        {
            /* Call was successful */
        }
    }
}
```

1.8.5 OSTmrSignal()

作用：更新计时器。

函数原型：

INT8U OSTmrSignal(void);

所属文件	调用者	开关量
OS_TMR.C	任务或中断	OS_TMR_EN

OSTmrSignal()通过任务或中断服务程序被调用，用以更新计时器。通常情况下，OSTmrSignal()将由 OSTimeTickHook()按照 **OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC** 的时间来调用。换句话说，如果 OS_CFG.H 中的 OS_TICKS_PER_SEC 设置为 1000，那么你应该每 10 或 100 个滴答中断（100 赫兹或 10 赫兹）调用 OSTmrSignal()。一般情况下，建议 100ms 调用一次 OSTmrSignal()更新计时器，因为更高的计时器速率，会使您的系统开销增大。

参数：无。

返回值：

返回 OS_ERR_NONE 表明调用成功。

注意：

本函数一般由系统中断调用，用户代码尽量不要调用。

范例1.8.5

```
#if OS_TMR_EN > 0
static INT16U OSTmrTickCtr = 0;
#endif
```

```
void OSTimeTickHook (void)
{
    #if OS_TMR_EN > 0
    OSTmrTickCtr++;
    if (OSTmrTickCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC))
    {
        OSTmrTickCtr = 0;
        OSTmrSignal();
    }
    #endif
}
```

1.8.6 OSTmrStart()

作用：启动计时器。

函数原型：

```
BOOLEAN OSTmrStart(OS_TMR *ptmr,
                   INT8U  *perr);
```

所属文件	调用者	开关量
OS_TMR.C	任务	OS_TMR_EN

OSTmrStart()，启动（或重新启动）计时器的倒计时过程。要启动的计时器必须是先前已经建立的。

参数：

ptmr：指向想要启动的定时器。该指针的值为在创建定时器时返回的值（见 OSTmrCreate（））。

perr： 指向存放错误代码的变量。其值为以下几种：

OS_ERR_NONE：启动成功。

OS_ERR_TMR_INVALID：ptmr 传入一个空指针。

OS_ERR_TMR_INVALID_TYPE：ptmr 不是指向一个计时器。

OS_ERR_NAME_GET_ISR：在 ISR 中调用本函数，不允许。

OS_ERR_TMR_INACTIVE：ptmr 指向一个已被删除或没有创建一个定时器。

返回值：

OS_TRUE: 定时器启动成功。

OS_FALSE: 操作有误。

注意：

1) 建议检查函数的返回值。
2) 不能在中断函数调用本函数。
3) 要启动的计时器必须是先前已经建立的。

范例1.8.6

```
OS_TMR *CloseDoorTmr;
BOOLEAN status;
```

```
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    for (;;)
    {
        status = OSTmrStart(CloseDoorTmr,&err);
        if (err == OS_ERR_NONE)
        {
            /* Timer was started */
        }
    }
}
```

1.8.7 OSTmrStateGet()

作用：获取计时器的当前状态。

函数原型：

```
INT8U OSTmrStateGet(OS_TMR *ptmr,
                    INT8U *perr);
```

所属文件	调用者	开关量
OS_TMR.C	任务	OS_TMR_EN

OSTmrStateGet()用来获取计时器的当前状态。

参数：

ptmr：指向想要获取信息的定时器。该指针的值为在创建定时器时返回的值（见 OSTmrCreate（））。

perr： 指向存放错误代码的变量。其值为以下几种：

OS_ERR_NONE： 启动成功。

OS_ERR_TMR_INVALID： ptmr 传入一个空指针。

OS_ERR_TMR_INVALID_TYPE： ptmr 不是指向一个计时器。

OS_ERR_NAME_GET_ISR： 在 ISR 中调用本函数，不允许。

OS_ERR_TMR_INACTIVE： ptmr 指向一个已被删除或没有创建一个定时器。

返回值：

定时器的状态，有以下几种：

OS_TMR_STATE_UNUSED： 计时器尚未创建。

OS_TMR_STATE_STOPPED： 定时器已创建，但尚未开始或已经停止计时。

OS_TMR_STATE_COMPLETED： 定时器在单次模式，已经完成了延时。

OS_TMR_STATE_RUNNING： 当前定时器正在运行。

注意：

1) 建议检查函数的返回值。
2) 不能在中断函数调用本函数。

范例1.8.7

深圳信盈达技术支持专用

```
INT8U CloseDoorTmrState;
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    for (;;)
    {
        CloseDoorTmrState = OSTmrStateGet(CloseDoorTmr, &err);
        if (err == OS_ERR_NONE)
        {
            /* Call was successful */
        }
    }
}
```

1.8.8 OSTmrStop()

作用：停止计时器。

函数原型：

```
BOOLEAN OSTmrStop(OS_TMR *ptmr,
                  INT8U opt,
                  void *callback_arg,
                  INT8U *perr);
```

所属文件	调用者	开关量
OS_TMR.C	任务	OS_TMR_EN

OSTmrStop()用于停止（即中止）定时器。

参数：

ptmr：指向想要停止的定时器。该指针的值为在创建定时器时返回的值（见 OSTmrCreate（））。

opt： 停止后的要做什么事情：

1) OS_TMR_OPT_NONE： 直接停止，不做任何其他处理。
2) OS_TMR_OPT_CALLBACK： 停止，用初始化的参数执行一次回调函数。
3) OS_TMR_OPT_CALLBACK_ARG：停止，用新的参数执行一次回调函数。

callback_arg：新的回调函数参数，如果回调函数不需要参数，可以是一个空指针。

perr： 指向存放错误代码的变量。其值为以下几种：

OS_ERR_NONE： 计时器已启动。
OS_ERR_TMR_INVALID： ptmr 传入一个空指针。
OS_ERR_TMR_INVALID_TYPE：ptmr 不是指向一个计时器。
OS_ERR_NAME_GET_ISR： 在 ISR 中调用本函数，不允许。
OS_ERR_TMR_INVALID_OPT： 指定了无效选项。
OS_ERR_TMR_STOPPED： 试图停止一个已停止计时器。
OS_ERR_TMR_INACTIVE： ptmr 指向一个已被删除或没有创建一个定时器。
OS_ERR_TMR_NO_CALLBACK：选项中指定了调用回调函数，但没有传入回调函数相关参数。

深圳信盈达技术支持

返回值：

OS_TRUE: 定时器停止成功。

OS_FALSE: 操作有误。

注意：

1) 建议检查函数的返回值。
2) 不能在中断函数调用本函数。
3) 要停止的计时器必须是先前已经建立的。

范例1.8.8

```
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    for (;;)
    {
        OSTmrStop(CloseDoorTmr,
        OS_TMR_OPT_CALLBACK,
        (void *)0,
        &err);
        if (err == OS_ERR_NONE || err == OS_ERR_TMR_STOPPED)
        {
            /* Timer was stopped ... */
            /* ... callback was called only if timer was running */
        }
    }
}
```

1.9 内存管理

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

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

■ **相关的 API 函数：**

- OSMemCreate () 建立并初始化一块内存区。
- OSMemGet () 从内存区分配一个内存块。
- OSMemNameGet() 获取存储分区名称。
- OSMemNameSet() 设置存储分区名称。
- OSMemPut () 释放一个内存块，内存块必须释放回原先申请的内存区。
- OSMemQuery () 得到内存区的信息。

1.9.1 OSMemCreate ()

作用： 建立并初始化一块内存区。

函数原型：

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

所属文件	调用者	开关量
OS_MEM.C	任务或初始化代码	OS_MEM_EN

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

参数：

addr： 建立的内存区的起始地址。内存区可以使用静态数组或在初始化时使用 malloc()函数建立。

nblks： 需要的内存块的数目。每一个内存区最少需要定义两个内存块。

blksize： 每个内存块的大小，最少应该能够容纳一个指针。

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

OS_ERR_NONE ： 成功建立内存区。
OS_ERR_MEM_INVALID_ADDR：指定一个无效的地址（空指针）或分区没有正确对齐。
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。如果没有空闲内存区，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1.9.1

```
OS_MEM *CommMem;
INT32U CommBuf[16][32];
void main (void)
{
    INT8U err;
    OSInit(); /* Initialize μC/OS-II */
    ...
    CommMem = OSMemCreate(&CommBuf[0][0], 16, 32 * sizeof(INT32U), &err);
    ...
    OSStart(); /* Start Multitasking */
}
```

深圳信盈达技术总发编制

1.9.2 OSMemGet ()

作用： 从内存区分配一个内存块。

函数原型：

```
void    *OSMemGet (OS_MEM    *pmem,
                   INT8U     *perr);
```

所属文件	调用者	开关量
OS_MEM.C	任务或中断代码	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_ERR_NONE：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：

OSMemCreate()函数返回指向内存区控制块的指针。如果没有空闲的内存区，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1.9.2

```
OS_MEM *CommMem;
void Task (void *p_arg)
{
    INT8U *pmsg;
    (void)p_arg;
    while(1)
    {
        pmsg = OSMemGet(CommMem, &err);
        if (pmsg != (INT8U *)0)
        {
         /*内存申请成功*/
        }
        ....
    }
}
```

1.9.3 OSMemNameGet()

作用： 获取存储分区名称。

函数原型：

```
INT8U    OSMemNameGet (OS_MEM      *pmem,
```

```
                    INT8U     **pname,
                    INT8U      *perr);
```

所属文件	调用者	开关量
OS_MEM.C	任务	OS_MEM_NAME_EN

OSMemNameGet（）获取存储分区的名称。

参数:

pmem: 指向存储器分区。

pname: 指向一个要存放 ASCII 字符串内存区的首地址。

perr: 指向包含错误码的变量的指针，并且可以是任何以下的:

OS_ERR_NONE: 获取成功。

OS_ERR_INVALID_PMEM: pmem 传递一个 NULL 指针。

OS_ERR_PNAME_NULL: pname 传递一个 NULL 指针。

OS_ERR_NAME_GET_ISR: 尝试从 ISR 使用本功能，不允许。

返回值:

获取到的 pname 字符串的长度或者 0。

注意/警告:

必须首先建立内存区，然后使用。

范例1.9.3

```
OS_MEM *CommMem;
INT8U *CommMemName;
void Task (void *pdata)
{
    INT8U err;
    INT8U size;
    pdata = pdata;
    while (;;)
    {
      /*获取内存分区名称*/
        size = OSMemNameGet(CommMem, &CommMemName, &err);
        ...
    }
}
```

1.9.4 OSMemNameSet()

作用: 设置存储分区名称。

函数原型:

```
void    OSMemNameSet (OS_MEM     *pmem,
                      INT8U     **pname,
                      INT8U      *perr);
```

深圳信盈达技术有限公司 版权所有 未经许可 不得复制

所属文件	调用者	开关量
OS_MEM.C	任务	OS_MEM_NAME_EN

OSMemNameSet（）设置存储分区名称。

参数:

pmem: 指向存储器分区。

pname: 指向一个 ASCII 字符串的首地址。

perr: 指向包含错误码的变量的指针，并且可以是任何以下的:

OS_ERR_NONE: 设置成功。

OS_ERR_INVALID_PMEM: pmem 传递一个 NULL 指针。

OS_ERR_PNAME_NULL: pname 传递一个 NULL 指针。

OS_ERR_MEM_NAME_TOO_LONG：设置的内存分区名称长度太长。

OS_ERR_NAME_GET_ISR: 尝试从 ISR 使用本功能，不允许。

返回值:

无。

注意/警告:

必须首先建立内存区，然后使用。

范例1.9.4

```
OS_MEM *CommMem;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while(1)
    {
        OSMemNameSet(CommMem, “Comm. Buffer”, &err);
        ...
    }
}
```

深圳信盈达 专编制

1.9.5 OSMemPut ()

作用： 设置存储分区名称。
函数原型： 释放一个内存块，内存块必须释放回原先申请的内存区。
INT8U OSMemPut(OS_MEM *pmem,
void *pblk);

所属文件	调用者	开关量
OS_MEM.C	任务	OS_MEM_NAME_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：

OS_ERR_NONE: 释放成功。
OS_ERR_MEM_FULL: 释放的空间比申请的多。
OS_ERR_MEM_INVALID_PMEM: pmem 传递了一个空指针。
OS_ERR_MEM_INVALID_PBLK: pblk 传递了一个空指针。

注意/警告：

必须首先建立内存区，然后使用。

范例1.9.4

```
OS_MEM *CommMem;
INT8U *CommMsg;
void Task (void *p_arg)
{
    INT8U err;
    (void)p_arg;
    while(1)
    {
        err = OSMemPut(CommMem, (void *)CommMsg);/*释放内存块*/
        if (err == OS_ERR_NONE)
        {
          /* 内存块释放成功 */
        }
        ...
    }
}
```

1.9.6 OSMemQuery ()

作用： 获取内存区的信息。
函数原型：

```
INT8U    OSMemQuery (OS_MEM          *pmem,
                     OS_MEM_DATA    *p_mem_data);
```

所属文件	调用者	开关量
OS_MEM.C	任务或中断函数	OS_MEM_EN&& OS_MEM_QUERY_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。

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

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

返回值:

OS_ERR_NONE: 获取成功。

OS_ERR_MEM_INVALID_PMEM: pmem 传递了一个空指针。

OS_ERR_MEM_INVALID_PBLK: pblk 传递了一个空指针。

注意/警告:

必须首先建立内存区，然后使用。

范例1.9.6

```
OS_MEM *CommMem;
void Task (void *p_arg)
{
    INT8U err;
    OS_MEM_DATA mem_data;
    (void)p_arg;
    while(1)
    {
    ...
    err = OSMemQuery(CommMem, &mem_data); /*获取内存分区信息*/
    ...
    }
}
```

1.10 其他函数

OSInit() 初始化 UCOS-II 内核相关函数。

OSStart() 启动多个任务并开始调度。

OSIntEnter() 中断函数正在执行。

OSIntExit() 中断函数已经完成(脱离中断)。

OSSchedLock() 给调度器上锁，禁止调度。
OSSchedUnlock() 给调度器解锁，允许调度。
OSVersion() 获得内核版本号。
OSSafetyCriticalStart() 表示所有的初始化工作已经完成，并且内核对象不再允许被创建(如：任务、邮箱、信号量等)。

1.10.1 OSStart()

作用： OSStart( )启动 uC/OS-II 的多任务环境。
函数原型：
void OSStart(void);

所属文件	调用者	开关量
OS_CORE.C	只能是启动代码	无

OSStart( )启动 uC/OS-II 的多任务环境。
参数： 无。
返回值： 无。
注意：

1) 在调用 OSStart( )之前必须先调用 OSInit ( )。
2) 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1.10.1

```
void main (void)
{
    ……
    OSInit();          // 初始化 uC/OS-II
    …….               // 最少创建一个任务
    OSStart();         //启动多任务内核
}
```

1.10.2 OSIntEnter()

作用： 通知内核中断函数正在执行。
函数原型：
void OSIntEnter(void);

所属文件	调用者	开关量
OS_CORE.C	中断	无

OSIntEnter()通知 uC/OS-ii 一个中断处理函数正在执行,这有助于 uC/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：

1) 在任务级不能调用该函数。
2) 如果用户使用的微处理器有存储器直接加 1 的单条指令的话,将全程变量 OSIntNesting 直接加 1,可以避免调用函数所带来的额外的开销!

3) 如果用户使用的微处理器没有这样的指令，必须先将 OSIntNesting 读入寄存器，再将寄存器加 1，然后再写回到变量 OSIatNesting 中去，就不如调用 OSIntEnter()。OSIntNesting 是共享资源。OSIntEnter()把上述三条指令用开中断、关中断保护起来，以保证处理 OSIntNesting 时的排它性。直接给 OSIntNesting 加 1 比调用 OSIntEnter()快得多，可能时，直接加 1 更好。当心的是，在有些情况下，从 OSIntEnter()返回时，会把中断开了。这种情况，在调用 OSIntEnter()之前要先清中断源，否则中断将连续反复打入，用户应用程序就会崩溃！

1.10.3 OSIntExit()

作用：通知内核一个中断服务已执行完毕。

函数原型：

void OSIntExit (void);

所属文件	调用者	开关量
OS_CORE.C	中断	无

OSIntExit()通知 uC/OS-ii 一个中断服务已执行完毕,这有助于 uC/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，uC/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：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。

1.10.4 OSSchedLock()

作用：给调度器上锁，禁止调度。

函数原型：

void OSSchedLock(void);

所属文件	调用者	开关量
OS_CORE.C	任务或中断	OS_SCHED_LOCK_EN

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。uC/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：

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1.10.2

```
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          */
    }
}
```

1.10.5 OSSchedUnlock()

作用：给调度器解锁，允许调度。

函数原型：

void OSSchedUnlock(void);

所属文件	调用者	开关量
OS_CORE.C	任务或中断	OS_SCHED_LOCK_EN

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：

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1.10.3

```
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          */
    }
}
```

1.10.6 OSVersion()

作用：获取当前 uC/OS-II 的版本号。

函数原型：

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 uC/OS-II 的版本号。

参数：无。

返回值：

当前版本，格式为 x.yy，返回值为乘以 10000 后的数值。例如当前版本 2.92，则返回 29200。

注意： 无。

范例1.10.4

```
void TaskX(void *pdata)
{
        INT16U os_version;

        for (;;) {
            os_version = OSVersion();    /*  获取  uC/OS-II's  的版本     */
        }
}
```

1.10.7 OSSafetyCriticalStart()

作用： 表示所有的初始化工作已经完成，并且内核对象不再允许被创建(如：任务、邮箱、信号量等)。

函数原型：

void OSSafetyCriticalStart(void);

所属文件	调用者	开关量
OS_CORE.C	任务或中断	无

OSSafetyCriticalStart（）表示所有的初始化已经完成，内核对象不再允许被创建。

参数： 无。

返回值： 无

注意： 无。

范例1.10.5

```
OS_STK Task1Stk[1024];
void main (void)
{
      INT8U err;
      ...
      OSInit(); /* Initialize μC/OS-II */
      ...
      OSTaskCreate(Task1,
                          (void *)0,
                          &Task1Stk[1023],
                          25);
      ...
      OSStart(); /* Start Multitasking */
}

void Task1 (void *p_arg)
{
      (void)p_arg; /* Prevent compiler warning */
      OSTaskCreate(_); /* Create the other tasks */
      OSSemCreate(_); /* Create semaphores */
      /* Create other kernel objects */
```

法陈志发编带

```
    OSQCreate(_); /* Create queues */
    OSSafetyCriticalStart() /* Prevent kernel objects from... */
    /* ... being created. */
    for (;;) {
        . /* Task code */
        .
    }
}
```

1.10.8 OSInit()

作用：初始化 UCOS-II 内核相关函数。
函数原型：
void OSInit(void);

所属文件	调用者	开关量
OS_CORE.C	只能是启动代码	无

OSinit()用于初始化 uC/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：必须先于 OSStart（）函数的调用。

范例1.10.6

```
void main (void)
{
    ……
    OSInit();          // 初始化 uC/OS-II
    …….              // 最少创建一个任务
    OSStart();         //启动多任务内核
}
```

1.11 临界区处理宏：

代码的临界段也称为临界区，指处理时不可分割的代码。一旦这部分代码开始执行，则不允许任何中断打入。为确保临界段代码的执行，在进入临界段之前要关中断，而临界段代码执行完以后要立即开中断。从代码的角度上来看，处在关中断和开中断之间的代码段就是临界段。

OS_ENTER_CRITICAL()　进入临界区，禁止被中断打断。
OS_EXIT_CRITICAL()　　退出临界区，允许被中断打断。

1.11.1 OS_ENTER_CRITICAL() ,OS_EXIT_CRITICAL()

作用：关闭、打开 CPU 的中断。
函数原型：无

所属文件	调用者	开关量
OS_CPU.H	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:

OS_ENTER_CRITICAL() 进入临界区，禁止被中断打断。

OS_EXIT_CRITICAL() 退出临界区，允许被中断打断。

返回值: 无

注意: OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()必须成对使用。

范例1.11.1

```
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        */

    }
}
```

1.12 内部函数原型

警告：内核专用函数，在应用程序中不能使用它们,否则内核会崩溃。

OS_Dummy() 建立一个虚拟函数。

OS_EventTaskRdy() 使一个任务进入就绪态(OS_EVENT *pevent, void *msg, INT8U msk)。

OS_EventTaskWait() 使一个任务进入等待某事件发生状态(ECB 指针)。

OS_EventTO() 由于超时而将任务置为就绪态(ECB 指针)。

OS_EventWaitListInit() 事件控制块列表初始化(事件控制块指针)。

OS_FlagInit() 初始化事件标志结构。

OS_FlagUnlink() 把这个 OS_FLAG_NODE 从事件标志组的等待任务链表中删除。

OS_MemInit() 初始化内存分区。

OS_QInit() 初始化事件队列结构。

OS_Sched() 任务调度函数。

OS_TaskIdle() 空闲任务函数(指向一个数据结构)。

OS_TaskStat() 统计任务(指向一个数据结构)。

OS_TCBInit() 初始化任务控制块 TCB(优先级指针、栈顶指针、栈底指针、任务标志符、 堆栈容量、扩展指针、选择项)。