第4章　任务管理

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

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

或

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

本章所讲的内容包括如何在用户的应用程序中建立任务、删除任务、改变任务的优先级、挂起和恢复任务，以及获得有关任务的信息。

μC/OS-Ⅱ可以管理多达 64 个任务，并从中保留了四个最高优先级和四个最低优先级的任务供自己使用，所以用户可以使用的只有 56 个任务。任务的优先级越高，反映优先级的值则越低。在最新的 μC/OS-Ⅱ版本中，任务的优先级数也可作为任务的标识符使用。

4.0 建立任务，OSTaskCreate()

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

OSTaskCreate()的代码如程序清单 L4.1 所述。从中可以知道，OSTaskCreate()需要四个参数:task 是任务代码的指针,pdata 是当任务开始执行时传递给任务的参数的指针,ptos 是分配给任务的堆栈的栈顶指针(参看 4.02，任务堆栈)，prio 是分配给任务的优先级。

程序清单 L4.1 OSTaskCreate()

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



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

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

OSTaskCreate()一开始先检测分配给任务的优先级是否有效[L4.1(1)]。任务的优先级必须在 0 到 OS_LOWEST_PRIO 之间。接着，OSTaskCreate()要确保在规定的优先级上还没有建立任务[L4.1(2)]。在使用 μC/OS-Ⅱ时，每个任务都有特定的优先级。如果某个优先级是空闲的，μC/OS-Ⅱ通过放置一个非空指针在 OSTCBPrioTbl[]中来保留该优先级[L4.1(3)]。这就使得 OSTaskCreate()在设置任务数据结构的其他部分时能重新允许中断[L4.1(4)]。

然后，OSTaskCreate()调用 OSTaskStkInit()[L4.1(5)],它负责建立任务的堆栈。该函数是与处理器的硬件体系相关的函数，可以在 OS_CPU_C.C 文件中找到。有关实现 OSTaskStkInit()的细节可参看第 8 章——移植 μC/OS-Ⅱ。如果已经有人在你用的处理器上成功地移植了 μC/OS-Ⅱ，而你又得到了他的代码，就不必考虑该函数的实现细节了。OSTaskStkInit()函数返回新的堆栈栈顶(psp),并被保存在任务的 OS_TCB 中。注意用户得将传递给 OSTaskStkInit()函数的第四个参数 opt 置 0，因为 OSTaskCreate()与 OSTaskCreateExt()不同，它不支持用户为任务的创建过程设置不同的选项，所以没有任何选项可以通过 opt 参数传递给 OSTaskStkInit()。

μC/OS-Ⅱ支持的处理器的堆栈既可以从上(高地址)往下(低地址)递减也可以从下往上递增。用户在调用 OSTaskCreate()的时候必须知道堆栈是递增的还是递减的(参看所用处理器的 OS_CPU.H 中的 OS_STACK_GROWTH)，因为用户必须得把堆栈的栈顶传递给 OSTaskCreate(),而栈顶可能是堆栈的最高地址(堆栈从上往下递减),也可能是最低地址(堆栈从下往上长)。

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

程序清单 L 4.2 OSTCBInit()

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



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

#if OS_TASK_DEL_EN
        ptcb->OSTCBDelReq    = OS_NO_ERR;
#endif

        ptcb->OSTCBY        = prio >> 3;
```

```
        ptcb->OSTCBBitY     = OSMapTbl[ptcb->OSTCBY];
        ptcb->OSTCBX        = prio & 0x07;
        ptcb->OSTCBBitX     = OSMapTbl[ptcb->OSTCBX];

#if     OS_MBOX_EN || (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_SEM_EN
        ptcb->OSTCBEventPtr  = (OS_EVENT *)0;
#endif

#if     OS_MBOX_EN || (OS_Q_EN && (OS_MAX_QS >= 2))
        ptcb->OSTCBMsg      = (void *)0;
#endif

        OS_ENTER_CRITICAL();                                        (4)
        OSTCBPrioTbl[prio]   = ptcb;                                (5)
        ptcb->OSTCBNext     = OSTCBList;
        ptcb->OSTCBPrev     = (OS_TCB *)0;
        if (OSTCBList != (OS_TCB *)0) {
            OSTCBList->OSTCBPrev = ptcb;
        }
        OSTCBList            = ptcb;
        OSRdyGrp            |= ptcb->OSTCBBitY;                     (6)
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        OS_EXIT_CRITICAL();
        return (OS_NO_ERR);                                         (7)
    } else {
        OS_EXIT_CRITICAL();
        return (OS_NO_MORE_TCB);
    }
}
```

当OSTCBInit()需要将OS_TCB插入到已建立任务的OS_TCB的双向链表中时[L4.2(5)]，它就禁止中断[L4.2(4)]。该双向链表开始于OSTCBList，而一个新任务的OS_TCB常常被插入到链表的表头。最后，该任务处于就绪状态[L4.2(6)]，并且 OSTCBInit()向它的调用者[OSTaskCreate()]返回一个代码表明OS_TCB已经被分配和初始化了[L4.2(7)]。

现在，我可以继续讨论OSTaskCreate()(程序清单 L4.1)函数了。从OSTCBInit()返回后，OSTaskCreate()要检验返回代码[L4.1(7)]，如果成功，就增加OSTaskCtr[L4.1(8)]，OSTaskCtr用于保存产生的任务数目。如果OSTCBInit()返回失败，就置OSTCBPrioTbl[prio]

的入口为 0[L4.1(12)]以放弃该任务的优先级。然后，OSTaskCreate()调用 OSTaskCreateHook()[L4.1(9)]，OSTaskCreateHook()是用户自己定义的函数，用来扩展 OSTaskCreate()的功能。例如，用户可以通过 OSTaskCreateHook()函数来初始化和存储浮点寄存器、MMU 寄存器的内容，或者其它与任务相关的内容。一般情况下，用户可以在内存中存储一些针对用户的应用程序的附加信息。OSTaskCreateHook()既可以在 OS_CPU_C.C 中定义(如果 OS_CPU_HOOKS_EN 置 1)，也可以在其它地方定义。注意，OSTaskCreate()在调用 OSTaskCreateHook()时，中断是关掉的，所以用户应该使 OSTaskCreateHook()函数中的代码尽量简化，因为这将直接影响中断的响应时间。OSTaskCreateHook()在被调用时会收到指向任务被建立时的OS_TCB的指针。这意味着该函数可以访问OS_TCB数据结构中的所有成员。

如果 OSTaskCreate()函数是在某个任务的执行过程中被调用(即 OSRunning 置为 True[L4.1(10)])，则任务调度函数会被调用[L4.1(11)]来判断是否新建立的任务比原来的任务有更高的优先级。如果新任务的优先级更高，内核会进行一次从旧任务到新任务的任务切换。如果在多任务调度开始之前(即用户还没有调用 OSStart())，新任务就已经建立了，则任务调度函数不会被调用。

4.1 建立任务，OSTaskCreateExt()

用 OSTaskCreateExt()函数来建立任务会更加灵活，但会增加一些额外的开销。OSTaskCreateExt()函数的代码如程序清单 L4.3 所示。

我们可以看到OSTaskCreateExt()需要九个参数!前四个参数(task,pdata,ptos 和 prio)与 OSTaskCreate()的四个参数完全相同，连先后顺序都一样。这样做的目的是为了使用户能够更容易地将用户的程序从 OSTaskCreate()移植到 OSTaskCreateExt()上去。

id 参数为要建立的任务创建一个特殊的标识符。该参数在 μC/OS 以后的升级版本中可能会用到，但在 μC/OS-Ⅱ中还未使用。这个标识符可以扩展 μC/OS-Ⅱ功能，使它可以执行的任务数超过目前的 64 个。但在这里，用户只要简单地将任务的 id 设置成与任务的优先级一样的值就可以了。

pbos 是指向任务的堆栈栈底的指针，用于堆栈的检验。

stk_size 用于指定堆栈成员数目的容量。也就是说，如果堆栈的入口宽度为 4 字节宽，那么 stk_size 为 10000 是指堆栈有 40000 个字节。该参数与 pbos 一样，也用于堆栈的检验。

pext 是指向用户附加的数据域的指针，用来扩展任务的 OS_TCB。例如，用户可以为每个任务增加一个名字(参看实例 3)，或是在任务切换过程中将浮点寄存器的内容储存到这个附加数据域中，等等。

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

程序清单 L 4.3 OSTaskCreateExt()

```
INT8U OSTaskCreateExt (void   (*task)(void *pd),
                       void   *pdata,
                       OS_STK  *ptos,
```

```
                    INT8U   prio,
                    INT16U  id,
                    OS_STK  *pbos,
                    INT32U  stk_size,
                    void   *pext,
                    INT16U  opt)
{
    void   *psp;
    INT8U   err;
    INT16U  i;
    OS_STK  *pfill;



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

        if (opt & OS_TASK_OPT_STK_CHK) {                                (5)
            if (opt & OS_TASK_OPT_STK_CLR) {
                Pfill = pbos;
                for (i = 0; i < stk_size; i++) {
                    #if OS_STK_GROWTH == 1
                    *pfill++ = (OS_STK)0;
                    #else
                    *pfill-- = (OS_STK)0;
                    #endif
                }
            }
        }
        psp = (void *)OSTaskStkInit(task, pdata, ptos, opt);            (6)
        err = OSTCBInit(prio, psp, pbos, id, stk_size, pext, opt);      (7)
```

```
        if (err == OS_NO_ERR) {                                          (8)
            OS_ENTER_CRITICAL;
            OSTaskCtr++;                                                 (9)
            OSTaskCreateHook(OSTCBPrioTbl[prio]);                       (10)
            OS_EXIT_CRITICAL();
            if (OSRunning) {                                            (11)
                OSSched();                                              (12)
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;                           (13)
            OS_EXIT_CRITICAL();
        }
        return (err);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    }
}
```

OSTaskCreateExt()一开始先检测分配给任务的优先级是否有效[L4.3(1)]。任务的优先级必须在 0 到 OS_LOWEST_PRIO 之间。接着，OSTaskCreateExt()要确保在规定的优先级上还没有建立任务[L4.3(2)]。在使用 μC/OS-Ⅱ时，每个任务都有特定的优先级。如果某个优先级是空闲的，μC/OS-Ⅱ通过放置一个非空指针在 OSTCBPrioTbl[]中来保留该优先级[L4.3(3)]。这就使得 OSTaskCreateExt()在设置任务数据结构的其他部分时能重新允许中断[L4.3(4)]。

为了对任务的堆栈进行检验[参看 4.03,堆栈检验，OSTaskStkChk()]，用户必须在 opt 参数中设置 OS_TASK_OPT_STK_CHK 标志。堆栈检验还要求在任务建立时堆栈的存储内容都是 0(即堆栈已被清零)。为了在任务建立的时候将堆栈清零，需要在 opt 参数中设置 OS_TASK_OPT_STK_CLR。当以上两个标志都被设置好后，OSTaskCreateExt()才能将堆栈清零[L4.3(5)]。

接着，OSTaskCreateExt()调用 OSTaskStkInit()[L4.3(6)],它负责建立任务的堆栈。该函数是与处理器的硬件体系相关的函数，可以在 OS_CPU_C.C 文件中找到。有关实现 OSTaskStkInit()的细节可参看第八章——移植 μC/OS-Ⅱ。如果已经有人在你用的处理器上成功地移植了 μC/OS-Ⅱ，而你又得到了他的代码，就不必考虑该函数的实现细节了。OSTaskStkInit()函数返回新的堆栈栈顶(psp),并被保存在任务的 OS_TCB 中。

μC/OS-Ⅱ支持的处理器的堆栈既可以从上(高地址)往下(低地址)递减也可以从下往上递增(参看 4.02,任务堆栈)。用户在调用 OSTaskCreateExt()的时候必须知道堆栈是递增的还是递减的(参看用户所用处理器的 OS_CPU.H 中的 OS_STACK_GROWTH)，因为用户必须得把

堆栈的栈顶传递给 OSTaskCreateExt()，而栈顶可能是堆栈的最低地址(当 OS_STK_GROWTH 为 0 时)，也可能是最高地址(当 OS_STK_GROWTH 为 1 时)。

一旦 OSTaskStkInit()函数完成了建立堆栈的任务，OSTaskCreateExt()就调用 OSTCBInit() [L4.3(7)],从空闲的 OS_TCB 缓冲池中获得并初始化一个 OS_TCB。OSTCBInit()的代码在 OSTaskCreate()中曾描述过(参看 4.00 节)，从 OSTCBInit()返回后，OSTaskCreateExt()要检验返回代码[L4.3(8)]，如果成功，就增加 OSTaskCtr[L4.3(9)]，OSTaskCtr 用于保存产生的任务数目。如果 OSTCBInit()返回失败,就置 OSTCBPrioTbl[prio]的入口为 0[L4.3(13)]以放弃对该任务优先级的占用。然后，OSTaskCreateExt()调用 OSTaskCreateHook()[L4.3(10)]，OSTaskCreateHook()是用户自己定义的函数，用来扩展 OSTaskCreateExt()的功能。OSTaskCreateHook()可以在 OS_CPU_C.C 中定义(如果 OS_CPU_HOOKS_EN 置 1)，也可以在其它地方定义(如果 OS_CPU_HOOKS_EN 置 0)。注意，OSTaskCreateExt()在调用 OSTaskCreateHook()时，中断是关掉的，所以用户应该使 OSTaskCreateHook()函数中的代码尽量简化，因为这将直接影响中断的响应时间。OSTaskCreateHook()被调用时会收到指向任务被建立时的 OS_TCB 的指针。这意味着该函数可以访问 OS_TCB 数据结构中的所有成员。

如果 OSTaskCreateExt()函数是在某个任务的执行过程中被调用的(即 OSRunning 置为 True[L4.3(11)])，以任务调度函数会被调用[L4.3(12)]来判断是否新建立的任务比原来的任务有更高的优先级。如果新任务的优先级更高，内核会进行一次从旧任务到新任务的任务切换。如果在多任务调度开始之前(即用户还没有调用 OSStart())，新任务就已经建立了，则任务调度函数不会被调用。

4.2 任务堆栈

每个任务都有自己的堆栈空间。堆栈必须声明为 OS_STK 类型，并且由连续的内存空间组成。用户可以静态分配堆栈空间(在编译的时候分配)也可以动态地分配堆栈空间(在运行的时候分配)。静态堆栈声明如程序清单 L4.4 和 4.5 所示，这两种声明应放置在函数的外面。

程序清单 L4.4 静态堆栈

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

或

程序清单 L4.5 静态堆栈

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

用户可以用 C 编译器提供的 malloc()函数来动态地分配堆栈空间，如程序清单 L4.6 所示。在动态分配中，用户要时刻注意内存碎片问题。特别是当用户反复地建立和删除任务时，内存堆中可能会出现大量的内存碎片，导致没有足够大的一块连续内存区域可用作任务堆栈，这时 malloc()便无法成功地为任务分配堆栈空间。

程序清单 L L4.6　用malloc()为任务分配堆栈空间

```
OS_STK  *pstk;



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

图 4.1 表示了一块能被 malloc()动态分配的 3K 字节的内存堆 [F4.1(1)]。为了讨论问题方便，假定用户要建立三个任务(任务 A,B 和 C)，每个任务需要 1K 字节的空间。设第一个 1K 字节给任务 A, 第二个 1K 字节给任务 B, 第三个 1K 字节给任务 C[F4.1(2)]。然后，用户的应用程序删除任务 A 和任务 C，用 free()函数释放内存到内存堆中[F4.1(3)]。现在，用户的内存堆虽有 2K 字节的自由内存空间，但它是不连续的，所以用户不能建立另一个需要 2K 字节内存的任务(即任务 D)。如果用户并不会去删除任务，使用 malloc()是非常可行的。

图 F4.1　　内存碎片

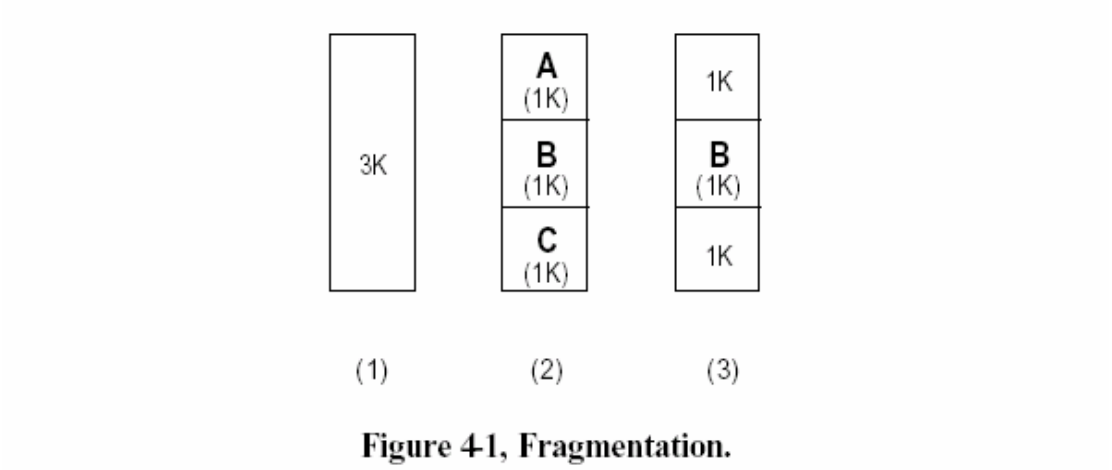

Figure 4-1, Fragmentation.

μC/OS-Ⅱ支持的处理器的堆栈既可以从上(高地址)往下(低地址)长也可以从下往上长(参看 4.02,任务堆栈)。用户在调用 OSTaskCreate()或 OSTaskCreateExt()的时候必须知道堆栈是怎样长的，因为用户必须得把堆栈的栈顶传递给以上两个函数，当 OS_CPU.H 文件中的 OS_STK_GROWTH 置为 0 时，用户需要将堆栈的最低内存地址传递给任务创建函数，如程序清单 4.7 所示。

程序清单 L4.7 堆栈从下往上递增

```
OS_STK  TaskStack[TASK_STACK_SIZE];


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

当 OS_CPU.H 文件中的 OS_STK_GROWTH 置为 1 时，用户需要将堆栈的最高内存地址传递给任务创建函数，如程序清单 4.8 所示。

程序清单 L4.8 堆栈从上往下递减

```
OS_STK  TaskStack[TASK_STACK_SIZE];


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

这个问题会影响代码的可移植性。如果用户想将代码从支持往下递减堆栈的处理器中移植到支持往上递增堆栈的处理器中的话，用户得使代码同时适应以上两种情况。在这种特殊情况下，程序清单 L4.7 和 4.8 可重新写成如程序清单 L4.9 所示的形式。

程序清单 L 4.9 对两个方向增长的堆栈都提供支持

```
OS_STK  TaskStack[TASK_STACK_SIZE];



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

任务所需的堆栈的容量是由应用程序指定的。用户在指定堆栈大小的时候必须考虑用户的任务所调用的所有函数的嵌套情况，任务所调用的所有函数会分配的局部变量的数目，以及所有可能的中断服务例程嵌套的堆栈需求。另外，用户的堆栈必须能储存所有的 CPU 寄存器。

4.3 堆栈检验，OSTaskStkChk()

有时候决定任务实际所需的堆栈空间大小是很有必要的。因为这样用户就可以避免为任务分配过多的堆栈空间，从而减少自己的应用程序代码所需的 RAM(内存)数量。μC/OS-Ⅱ提供的 OSTaskStkChk()函数可以为用户提供这种有价值的信息。

在图 4.2 中，笔者假定堆栈是从上往下递减的(即 OS_STK_GROWTH 被置为 1)，但以下的讨论也同样适用于从下往上长的堆栈[F4.2(1)]。μC/OS-Ⅱ是通过查看堆栈本身的内容来决定堆栈的方向的。只有内核或是任务发出堆栈检验的命令时，堆栈检验才会被执行，它不会自动地去不断检验任务的堆栈使用情况。在堆栈检验时，μC/OS-Ⅱ要求在任务建立的时候堆栈中存储的必须是 0 值(即堆栈被清零)[F4.2(2)]。另外，μC/OS-Ⅱ还需要知道堆栈栈底(BOS)的位置和分配给任务的堆栈的大小[F4.2(2)]。在任务建立的时候，BOS 的位置及堆栈的这两个值储存在任务的 OS_TCB 中。

为了使用 μC/OS-Ⅱ的堆栈检验功能，用户必须要做以下几件事情：

- 在 OS_CFG.H 文件中设 OS_TASK_CREATE_EXT 为 1。
- 用 OSTaskCreateExt()建立任务，并给予任务比实际需要更多的内存空间。
- 在 OSTaskCreateExt()中，将参数 opt 设置为 OS_TASK_OPT_STK_CHK+OS_TASK_OPT_STK_CLR。注意如果用户的程序启动代码清除了所有的 RAM，并且从未删除过已建立了的任务，那么用户就不必设置选项 OS_TASK_OPT_STK_CLR 了。这样就会减少 OSTaskCreateExt()的执行时间。
- 将用户想检验的任务的优先级作为 OSTaskStkChk()的参数并调用之。

图 4.2　　堆栈检验

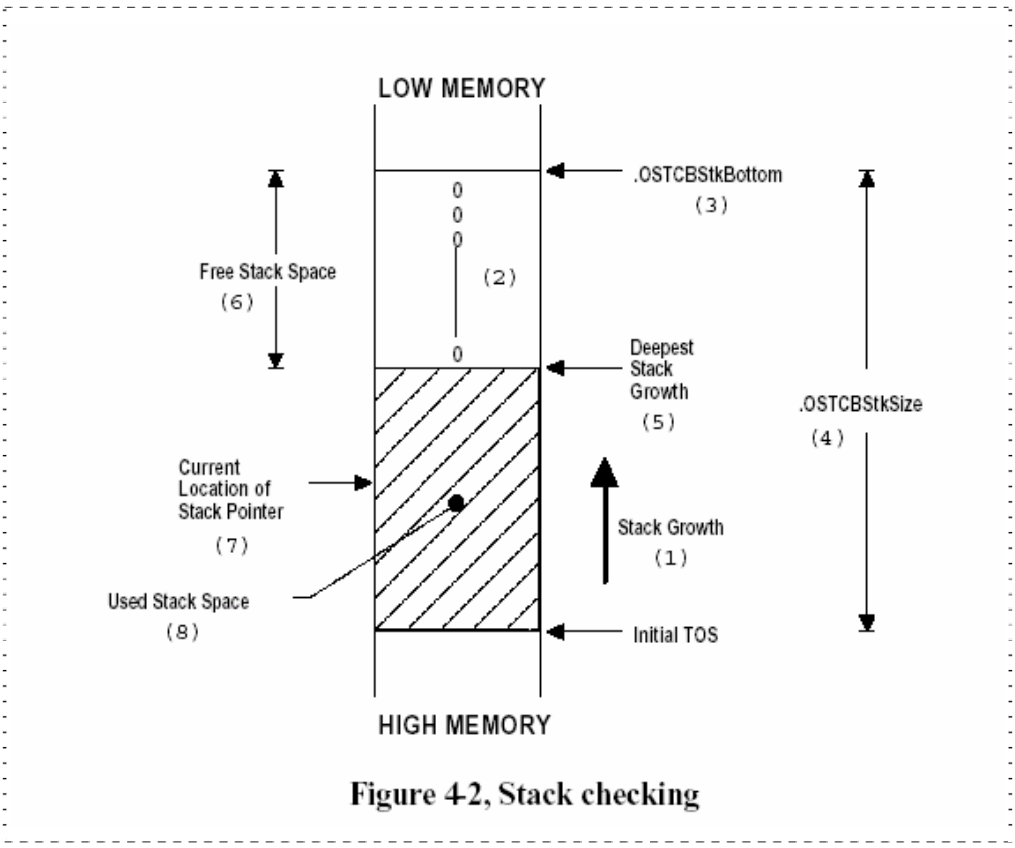

Figure 4-2, Stack checking

OSTaskStkChk()顺着堆栈的栈底开始计算空闲的堆栈空间大小，具体实现方法是统计储存值为 0 的连续堆栈入口的数目，直到发现储存值不为 0 的堆栈入口[F4.2(5)]。注意堆栈入口的储存值在进行检验时使用的是堆栈的数据类型(参看 OS_CPU.H 中的 OS_STK)。换句话说，如果堆栈的入口有 32 位宽，对 0 值的比较也是按 32 位完成的。所用的堆栈的空间大小是指从用户在 OSTaskCreateExt()中定义的堆栈大小中减去了储存值为 0 的连续堆栈入口以后的大小。OSTaskStkChk()实际上把空闲堆栈的字节数和已用堆栈的字节数放置在 OS_STK_DATA 数据结构中(参看 μCOS_Ⅱ.H)。注意在某个给定的时间，被检验的任务的堆栈指针可能会指向最初的堆栈栈顶(TOS)与堆栈最深处之间的任何位置[F4.2(7)]。每次在调用 OSTaskStkChk()的时候，用户也可能会因为任务还没触及堆栈的最深处而得到不同的堆栈的空闲空间数。

用户应该使自己的应用程序运行足够长的时间，并且经历最坏的堆栈使用情况，这样才能得到正确的数。一旦 OSTaskStkChk()提供给用户最坏情况下堆栈的需求，用户就可以重新设置堆栈的最后容量了。为了适应系统以后的升级和扩展，用户应该多分配 10%－100%

的堆栈空间。在堆栈检验中，用户所得到的只是一个大致的堆栈使用情况，并不能说明堆栈使用的全部实际情况。

OSTaskStkChk()函数的代码如程序清单 L4.10 所示。OS_STK_DATA(参看 μCOS_Ⅱ.H)数据结构用来保存有关任务堆栈的信息。笔者打算用一个数据结构来达到两个目的。第一，把OSTaskStkChk()当作是查询类型的函数，并且使所有的查询函数用同样的方法返回，即返回查询数据到某个数据结构中。第二，在数据结构中传递数据使得笔者可以在不改变OSTaskStkChk()的 API(应用程序编程接口)的条件下为该数据结构增加其它域，从而扩展OSTaskStkChk()的功能。现在，OS_STK_DATA 只包含两个域：OSFree 和 OSUsed。从代码中用户可看到，通过指定执行堆栈检验的任务的优先级可以调用 OSTaskStkChk()。如果用户指定 OS_PRIO_SELF[L4.10(1)]，那么就表明用户想知道当前任务的堆栈信息。当然，前提是任务已经存在[L4.10(2)]。要执行堆栈检验，用户必须已用 OSTaskCreateExt()建立了任务并且已经传递了选项 OS_TASK_OPT_CHK[L4.10(3)]。如果所有的条件都满足了，OSTaskStkChk()就会象前面描述的那样从堆栈栈底开始统计堆栈的空闲空间[L4.10(4)]。最后，储存在 OS_STK_DATA 中的信息就被确定下来了[L4.10(5)]。注意函数所确定的是堆栈的实际空闲字节数和已被占用的字节数，而不是堆栈的总字节数。当然，堆栈的实际大小(用字节表示)就是该两项之和。

程序清单 L 4.10　堆栈检验函数

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



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

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

4.4 删除任务，OSTaskDel()

有时候删除任务是很有必要的。删除任务,是说任务将返回并处于休眠状态(参看 3.02,任务状态)，并不是说任务的代码被删除了，只是任务的代码不再被 μC/OS-Ⅱ调用。通过调用 OSTaskDel()就可以完成删除任务的功能(如程序清单 L4.11 所示)。OSTaskDel()一开始应确保用户所要删除的任务并非是空闲任务，因为删除空闲任务是不允许的[L4.11(1)]。不过，用户可以删除 statistic 任务[L4.11(2)]。接着，OSTaskDel()还应确保用户不是在 ISR 例程中去试图删除一个任务，因为这也是不被允许的[L4.11(3)]。调用此函数的任务可以通过指定 OS_PRIO_SELF 参数来删除自己[L4.11(4)]。接下来 OSTaskDel()会保证被删除的任务是确实存在的[L4.11(3)]。如果指定的参数是 OS_PRIO_SELF 的话，这一判断过程(任务是否存在)自然是可以通过的，但笔者不准备为这种情况单独写一段代码，因为这样只会增加代码并延长程序的执行时间。

程序清单 L 4.11　删除任务

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

   if (prio == OS_IDLE_PRIO) {                                          (1)
      return (OS_TASK_DEL_IDLE);
   }
   if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {                (2)
      return (OS_PRIO_INVALID);
   }
   OS_ENTER_CRITICAL();
   if (OSIntNesting > 0) {                                              (3)
      OS_EXIT_CRITICAL();
      return (OS_TASK_DEL_ISR);
   }
   if (prio == OS_PRIO_SELF) {                                          (4)
      Prio = OSTCBCur->OSTCBPrio;
   }
   if ((ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) {                    (5)
      if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {          (6)
         OSRdyGrp &= ~ptcb->OSTCBBitY;
      }
      if ((pevent = ptcb->OSTCBEventPtr) != (OS_EVENT *)0) {            (7)
         if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0)
{
            pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
         }
      }
      Ptcb->OSTCBDly  = 0;                                              (8)
      Ptcb->OSTCBStat = OS_STAT_RDY;                                    (9)
      OSLockNesting++;                                                 (10)
      OS_EXIT_CRITICAL();                                              (11)
      OSDummy();                                                       (12)
```

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

一旦所有条件都满足了，OS_TCB 就会从所有可能的 μC/OS-Ⅱ的数据结构中移除。OSTaskDel()分两步完成该移除任务以减少中断响应时间。首先，如果任务处于就绪表中，它会直接被移除[L4.11(6)]。如果任务处于邮箱、消息队列或信号量的等待表中，它就从自己所处的表中被移除[L4.11(7)]。接着，OSTaskDel()将任务的时钟延迟数清零，以确保自己重新允许中断的时候，ISR 例程不会使该任务就绪[L4.11(8)]。最后，OSTaskDel()置任务的.OSTCBStat 标志为 OS_STAT_RDY。注意，OSTaskDel()并不是试图使任务处于就绪状态，而是阻止其它任务或 ISR 例程让该任务重新开始执行(即避免其它任务或 ISR 调用 OSTaskResume()[L4.11(9)])。这种情况是有可能发生的，因为 OSTaskDel()会重新打开中断，而 ISR 可以让更高优先级的任务处于就绪状态，这就可能会使用户想删除的任务重新开始执行。如果不想置任务的.OSTCBStat 标志为 OS_STAT_RDY，就只能清除 OS_STAT_SUSPEND 位了(这样代码可能显得更清楚，更容易理解一些)，但这样会使得处理时间稍长一些。

要被删除的任务不会被其它的任务或 ISR 置于就绪状态，因为该任务已从就绪任务表中删除了，它不是在等待事件的发生，也不是在等待延时期满，不能重新被执行。为了达到删除任务的目的，任务被置于休眠状态。正因为这样，OSTaskDel()必须得阻止任务调度程序[L4.11(10)]在删除过程中切换到其它的任务中去，因为如果当前的任务正在被删除，它不可能被再次调度！接下来，OSTaskDel()重新允许中断以减少中断的响应时间[L4.11(11)]。

这样，OSTaskDel()就能处理中断服务了，但由于它增加了 OSLockNesting，ISR 执行完后会返回到被中断任务，从而继续任务的删除工作。注意 OSTaskDel()此时还没有完全完成删除任务的工作，因为它还需要从 TCB 链中解开 OS_TCB，并将 OS_TCB 返回到空闲 OS_TCB 表中。

另外需要注意的是，笔者在调用 OS_EXIT_CRITICAL()函数后，马上调用了 OSDummy()[L4.11(12)]，该函数并不会进行任何实质性的工作。这样做只是因为想确保处理器在中断允许的情况下至少执行一个指令。对于许多处理器来说，执行中断允许指令会强制 CPU 禁止中断直到下个指令结束！Intel 80x86 和 Zilog Z-80 处理器就是如此工作的。开中断后马上关中断就等于从来没开过中断，当然这会增加中断的响应时间。因此调用 OSDummy()确保在再次禁止中断之前至少执行了一个调用指令和一个返回指令。当然，用户可以用宏定义将 OSDummy()定义为一个空操作指令（译者注：例如 MC68HC08 指令中的 NOP 指令），这样调用 OSDummy()就等于执行了一个空操作指令，会使 OSTaskDel()的执行时间稍微缩短一点。但笔者认为这种宏定义是没价值的，因为它会增加移植 μCOS-Ⅱ的工作量。

现在，OSTaskDel()可以继续执行删除任务的操作了。在 OSTaskDel()重新关中断后，它通过锁定嵌套计数器(OSLockNesting)减一以重新允许任务调度[L4.11(13)]。接着，OSTaskDel()调用用户自定义的 OSTaskDelHook()函数[L4.11(14)]，用户可以在这里删除或释放自定义的 TCB 附加数据域。然后，OSTaskDel()减少 μCOS-Ⅱ的任务计数器。OSTaskDel()简单地将指向被删除的任务的 OS_TCB 的指针指向 NULL[L4.11(15)]，从而达到将 OS_TCB 从优先级表中移除的目的。再接着，OSTaskDel()将被删除的任务的 OS_TCB 从 OS_TCB 双向链表中移除[L4.11(16)]。注意,没有必要检验 ptcb->OSTCBNext==0 的情况，因为 OSTaskDel()不能删除空闲任务，而空闲任务就处于链表的末端(ptcb->OSTCBNext==0)。接下来，OS_TCB 返回到空闲 OS_TCB 表中，并允许其它任务的建立[L4.11(17)]。最后，调用任务调度程序来查看在 OSTaskDel()重新允许中断的时候[L4.11(11)]，中断服务子程序是否曾使更高优先级的任务处于就绪状态[L4.11(18)]。

4.5 请求删除任务，OSTaskDelReq()

有时候，如果任务 A 拥有内存缓冲区或信号量之类的资源，而任务 B 想删除该任务，这些资源就可能由于没被释放而丢失。在这种情况下，用户可以想法子让拥有这些资源的任务在使用完资源后，先释放资源，再删除自己。用户可以通过 OSTaskDelReq()函数来完成该功能。

发出删除任务请求的任务(任务 B)和要删除的任务(任务 A)都需要调用 OSTaskDelReq()函数。任务 B 的代码如程序清单 L4.12 所示。任务 B 需要决定在怎样的情况下请求删除任务[L4.12(1)]。换句话说，用户的应用程序需要决定在什么样的情况下删除任务。如果任务需要被删除，可以通过传递被删除任务的优先级来调用 OSTaskDelReq()[L4.12(2)]。如果要被删除的任务不存在(即任务已被删除或是还没被建立)，OSTaskDelReq()返回 OS_TASK_NOT_EXIST。如果 OSTaskDelReq()的返回值为 OS_NO_ERR，则表明请求已被接受但任务还没被删除。用户可能希望任务 B 等到任务 A 删除了自己以后才继续进行下面的工作，这时用户可以象笔者一样，通过让任务 B 延时一定时间来达到这个目的[L4.12(3)]。笔者延时了一个时钟节拍。如果需要，用户可以延时得更长一些。当任务 A 完全删除自己后，[L4.12(2)]中的返回值成为 OS_TASK_NOT_EXIST，此时循环结束[L4.12(4)]。

程序清单 L 4.12 请求删除其它任务的任务(任务B)

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

```
{
    INT8U err;



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

程序清单 L 4.13　需要删除自己的任务(任务A)

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



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

```
}
```

需要删除自己的任务(任务 A)的代码如程序清单 L4.13 所示。在 OS_TAB 中存有一个标志，任务通过查询这个标志的值来确认自己是否需要被删除。这个标志的值是通过调用 OSTaskDelReq(OS_PRIO_SELF) 而得到的。当 OSTaskDelReq() 返回给调用者 OS_TASK_DEL_REQ[L4.13(1)]时，则表明已经有另外的任务请求该任务被删除了。在这种情况下，被删除的任务会释放它所拥有的所用资源[L4.13(2)]，并且调用 OSTaskDel(OS_PRIO_SELF)来删除自己[L4.13(3)]。前面曾提到过，任务的代码没有被真正的删除，而只是 μC/OS-Ⅱ不再理会该任务代码，换句话说，就是任务的代码不会再运行了。但是，用户可以通过调用 OSTaskCreate()或 OSTaskCreateExt()函数重新建立该任务。

OSTaskDelReq()的代码如程序清单 L4.14 所示。通常 OSTaskDelReq()需要检查临界条件。首先，如果正在删除的任务是空闲任务，OSTaskDelReq()会报错并返回[L4.14(1)]。接着，它要保证调用者请求删除的任务的优先级是有效的[L4.14(2)]。如果调用者就是被删除任务本身，存储在 OS_TCB 中的标志将会作为返回值[L4.14(3)]。如果用户用优先级而不是 OS_PRIO_SELF 指定任务，并且任务是存在的[L4.14(4)]，OSTaskDelReq()就会设置任务的内部标志[L4.14(5)]。如果任务不存在，OSTaskDelReq()则会返回 OS_TASK_NOT_EXIST，表明任务可能已经删除自己了[L4.14(6)]。

程序清单 L 4.14 OSTaskDelReq().

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



    if (prio == OS_IDLE_PRIO) {                                          (1)
        return (OS_TASK_DEL_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {
(2)
        return (OS_PRIO_INVALID);
    }
    if (prio == OS_PRIO_SELF) {                                          (3)
        OS_ENTER_CRITICAL();
        stat = OSTCBCur->OSTCBDelReq;
        OS_EXIT_CRITICAL();
        return (stat);
```

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

4.6 改变任务的优先级，OSTaskChangePrio()

在用户建立任务的时候会分配给任务一个优先级。在程序运行期间，用户可以通过调用OSTaskChangePrio()来改变任务的优先级。换句话说，就是 μC/OS-Ⅱ允许用户动态的改变任务的优先级。

OSTaskChangePrio()的代码如程序清单 L4.15 所示。用户不能改变空闲任务的优先级[L4.15(1)]，但用户可以改变调用本函数的任务或者其它任务的优先级。为了改变调用本函数的任务的优先级，用户可以指定该任务当前的优先级或 OS_PRIO_SELF，OSTaskChangePrio()会决定该任务的优先级。用户还必须指定任务的新(即想要的)优先级。因为 μC/OS-Ⅱ不允许多个任务具有相同的优先级，所以 OSTaskChangePrio()需要检验新优先级是否是合法的（即不存在具有新优先级的任务）[L4.15(2)]。如果新优先级是合法的，μC/OS-Ⅱ通过将某些东西储存到 OSTCBPrioTbl[newprio]中保留这个优先级[L4.15(3)]。如此就使得 OSTaskChangePrio()可以重新允许中断，因为此时其它任务已经不可能建立拥有该优先级的任务，也不能通过指定相同的新优先级来调用 OSTaskChangePrio()。接下来 OSTaskChangePrio()可以预先计算新优先级任务的 OS_TCB 中的某些值[L4.15(4)]。而这些值用来将任务放入就绪表或从该表中移除(参看 3.04,就绪表)。

接着，OSTaskChangePrio()检验目前的任务是否想改变它的优先级[L4.15(5)]。然后，OSTaskChangePrio()检查想要改变优先级的任务是否存在[L4.15(6)]。很明显，如果要改变优先级的任务就是当前任务，这个测试就会成功。但是，如果 OSTaskChangePrio()想要改变优先级的任务不存在，它必须将保留的新优先级放回到优先级表 OSTCBPrioTbl[]中[L4.15(17)]，并返回给调用者一个错误码。

现在，OSTaskChangePrio()可以通过插入 NULL 指针将指向当前任务 OS_TCB 的指针从优先级表中移除了[L4.15(7)]。这就使得当前任务的旧的优先级可以重新使用了。接着，我们检验一下 OSTaskChangePrio()想要改变优先级的任务是否就绪[L4.15(8)]。如果该任务处于就绪状态，它必须在当前的优先级下从就绪表中移除[L4.15(9)]，然后在新的优先级下插

入到就绪表中[L4.15(10)]。这儿需要注意的是，OSTaskChangePrio()所用的是重新计算的值[L4.15(4)]将任务插入就绪表中的。

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

接着，OSTaskChangePrio()将指向任务 OS_TCB 的指针存到 OSTCBPrioTbl[]中[L4.15(13)]。新的优先级被保存在OS_TCB中[L4.15(14)],重新计算的值也被保存在OS_TCB中[L4.15(15)]。OSTaskChangePrio()完成了关键性的步骤后，在新的优先级高于旧的优先级或新的优先级高于调用本函数的任务的优先级情况下，任务调度程序就会被调用[L4.15(16)]。

程序清单 L 4.15　OSTaskChangePrio().

```
INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio)
{
    OS_TCB   *ptcb;
    OS_EVENT *pevent;
    INT8U    x;
    INT8U    y;
    INT8U    bitx;
    INT8U    bity;



    if ((oldprio >= OS_LOWEST_PRIO && oldprio != OS_PRIO_SELF)  ||      (1)
        newprio >= OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[newprio] != (OS_TCB *)0) {                         (2)
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    } else {
        OSTCBPrioTbl[newprio] = (OS_TCB *)1;                            (3)
        OS_EXIT_CRITICAL();
        y    = newprio >> 3;                                            (4)
```

```
        bity = OSMapTbl[y];
        x    = newprio & 0x07;
        bitx = OSMapTbl[x];
        OS_ENTER_CRITICAL();
        if (oldprio == OS_PRIO_SELF) {                                  (5)
            oldprio = OSTCBCur->OSTCBPrio;
        }
        if ((ptcb = OSTCBPrioTbl[oldprio]) != (OS_TCB *)0) {            (6)
            OSTCBPrioTbl[oldprio] = (OS_TCB *)0;                        (7)
            if (OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) {             (8)
                if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {(9)
                    OSRdyGrp &= ~ptcb->OSTCBBitY;
                }
                OSRdyGrp    |= bity;                                   (10)
                OSRdyTbl[y] |= bitx;
            } else {
                if ((pevent = ptcb->OSTCBEventPtr) != (OS_EVENT *)0) { (11)
                    if ((pevent->OSEventTbl[ptcb->OSTCBY] &=
                         ~ptcb->OSTCBBitX) == 0) {
                        pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
                    }
                    pevent->OSEventGrp    |= bity;                      (12)
                    pevent->OSEventTbl[y] |= bitx;
                }
            }
            OSTCBPrioTbl[newprio] = ptcb;                               (13)
            ptcb->OSTCBPrio      = newprio;                             (14)
            ptcb->OSTCBY         = y;                                   (15)
            ptcb->OSTCBX         = x;
            ptcb->OSTCBBitY      = bity;
            ptcb->OSTCBBitX      = bitx;
            OS_EXIT_CRITICAL();
            OSSched();                                                  (16)
            return (OS_NO_ERR);
        } else {
```

```
            OSTCBPrioTbl[newprio] = (OS_TCB *)0;                          (17)
            OS_EXIT_CRITICAL();
            return (OS_PRIO_ERR);
        }
    }
}
```

4.7 挂起任务，OSTaskSuspend()

有时候将任务挂起是很有用的。挂起任务可通过调用 OSTaskSuspend()函数来完成。被挂起的任务只能通过调用 OSTaskResume()函数来恢复。任务挂起是一个附加功能。也就是说，如果任务在被挂起的同时也在等待延时的期满，那么，挂起操作需要被取消，而任务继续等待延时期满，并转入就绪状态。任务可以挂起自己或者其它任务。

OSTaskSuspend()函数的代码如程序清单 L4.16 所示。通常 OSTaskSuspend()需要检验临界条件。首先，OSTaskSuspend()要确保用户的应用程序不是在挂起空闲任务[L4.16(1)]，接着确认用户指定优先级是有效的[L4.16(2)]。记住最大的有效的优先级数(即最低的优先级)是 OS_LOWEST_PRIO。注意，用户可以挂起统计任务（statistic)。可能用户已经注意到了，第一个测试[L4.16(1)]在[L4.16(2)]中被重复了。笔者这样做是为了能与 μC/OS 兼容。第一个测试能够被移除并可以节省一点程序处理的时间，但是，这样做的意义不大，所以笔者决定留下它。

接着，OSTaskSuspend()检验用户是否通过指定 OS_PRIO_SELF 来挂起调用本函数的任务本身[L4.16(3)]。用户也可以通过指定优先级来挂起调用本函数的任务[L4.16(4)]。在这两种情况下，任务调度程序都需要被调用。这就是笔者为什么要定义局部变量 self 的原因，该变量在适当的情况下会被测试。如果用户没有挂起调用本函数的任务，OSTaskSuspend()就没有必要运行任务调度程序，因为正在挂起的是较低优先级的任务。

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

程序清单 L 4.16 OSTaskSuspend().

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

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

4.8 恢复任务，OSTaskResume()

在上一节中曾提到过，被挂起的任务只有通过调用 OSTaskResume()才能恢复。OSTaskResume()函数的代码如程序清单 L4.17 所示。因为 OSTaskSuspend()不能挂起空闲任务，所以必须得确认用户的应用程序不是在恢复空闲任务[L4.17(1)]。注意，这个测试也可以确保用户不是在恢复优先级为 OS_PRIO_SELF 的任务(OS_PRIO_SELF 被定义为 0xFF，它总是比 OS_LOWEST_PRIO 大)。

要恢复的任务必须是存在的，因为用户要需要操作它的任务控制块 OS_TCB[L4.17(2)]，并且该任务必须是被挂起的[L4.17(3)]。OSTaskResume()是通过清除 OSTCBStat 域中的 OS_STAT_SUSPEND 位来取消挂起的[L4.17(4)]。要使任务处于就绪状态，OS_TCBDly 域必须为 0[L4.17(5)]，这是因为在 OSTCBStat 中没有任何标志表明任务正在等待延时的期满。只有当以上两个条件都满足的时候，任务才处于就绪状态[L4.17(6)]。最后，任务调度程序会检查被恢复的任务拥有的优先级是否比调用本函数的任务的优先级高[L4.17(7)]。

程序清单 L 4.17　OSTaskResume().

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



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

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

4.9 获得有关任务的信息，OSTaskQuery()

用户的应用程序可以通过调用 OSTaskQuery()来获得自身或其它应用任务的信息。实际上，OSTaskQuery()获得的是对应任务的 OS_TCB 中内容的拷贝。用户能访问的 OS_TCB 的数据域的多少决定于用户的应用程序的配置(参看 OS_CFG.H)。由于 μC/OS-Ⅱ是可裁剪的，它只包括那些用户的应用程序所要求的属性和功能。

要调用 OSTaskQuery()，如程序清单 L4.18 中所示的那样，用户的应用程序必须要为 OS_TCB 分配存储空间。这个 OS_TCB 与 μC/OS-Ⅱ分配的 OS_TCB 是完全不同的数据空间。在调用了 OSTaskQuery()后，这个 OS_TCB 包含了对应任务的 OS_TCB 的副本。用户必须十分小心地处理 OS_TCB 中指向其它 OS_TCB 的指针(即 OSTCBNext 与 OSTCBPrev)；用户不要试图去改变这些指针!一般来说,本函数只用来了解任务正在干什么——本函数是有用的调试工具。

程序清单 L 4.18 得到任务的信息

```
OS_TCB MyTaskData;


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

OSTaskQuery()的代码如程序清单 L4.19 所示。注意，笔者允许用户查询所有的任务，包括空闲任务[L4.19(1)]。用户尤其需要注意的是不要改变OSTCBNext与OSTCBPrev的指向。通常，OSTaskQuery()需要检验用户是否想知道当前任务的有关信息[L4.19(2)]以及该任务是否已经建立了[L4.19(3)]。所有的域是通过赋值语句一次性复制的而不是一个域一个域地复制的[L4.19(4)]。这样复制会比较快一点，因为编译器大多都能够产生内存拷贝指令。

程序清单 L 4.19　OSTaskQuery().

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


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