《嵌入式实时操作系统 UCOSII》

读书笔记

By Jamza

目录

2 实时系统概念

一般具有两种类型的实时系统，即软实时系统，与硬实时系统，软实时系统要求任务能够尽快的运行，但是任务运行是否及时并不是非常的重要，而硬实时系统则需要任务严格的按照时序执行，如果出现超时等情况，则结果会是灾难性的。

在实际应用中，大部分的嵌入式系统是软实时系统与硬实时系统的结合。

2.00 前台系统与后台系统

前台系统与后台系统（foreground system / background system）是嵌入式系统设计的一种方式，即应用程序有主循环程序与中断服务程序组成，主循环程序由 N 个时间片组成，每个时间片内安排的任务，按照时间片前后顺序循环执行。当外部中断打断主循环程序后，在中断服务程序中完成关键信息的处理或者缓存，然后退出中断后，在主循环程序中的某个时间片内处理中断信息任务，当然前提是主循环程序按照时间片顺序执行到包含该中断信息任务的时间片时，方可执行中断信息任务，因此对于中断任务处理的及时性较差。

主循环程序叫后台行为，中断服务程序叫前台行为，后台也叫任务级，前台也叫中断级。

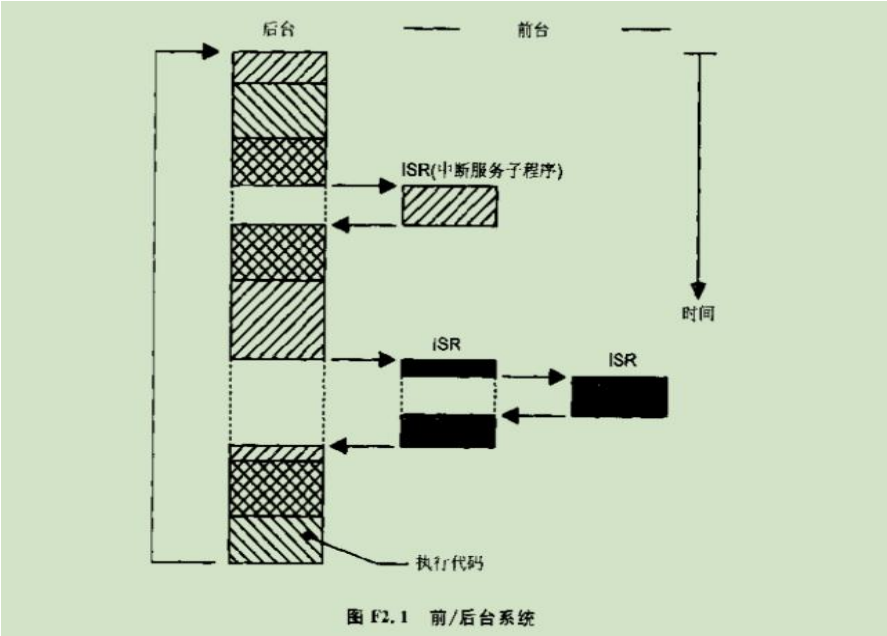


图 F2.1 前/后台系统

2.01 代码的临界段

不希望被中断打断的代码，需要通过关闭 CPU 的中断来实现代码完整性的保护。关闭中断的方式，对于不同的 CPU，具有不同的实现方法，UCOSII 通过封装两个宏，保证了对于不同 CPU 关闭中断方式的可移植性。

通过特定的 CPU 的汇编程序，实现关闭中断，比较高效率。

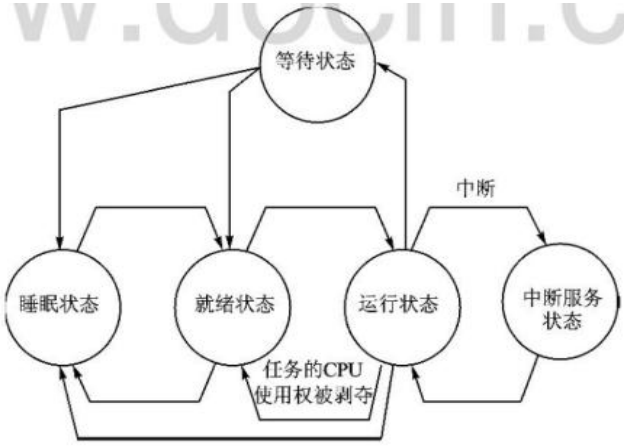
2.03 共享资源

共享资源是多个任务共同使用的实体资源，比如打印机，当一个任务占用打印机时候，另一个任务不能打断其占用，也即任务必须独占共享资源，术语为互斥，mutual exclusion。在 UCOSII 中，使用互斥信号量来实现任务对共享资源的独占。

2.05 任务

一个实时系统应用程序由多个任务组成，每个任务各自有独立的堆栈空间与运行 CPU 寄存器状态，即运行环境。实时操作系统按照规定的调度策略，轮转运行应用程序的多个任务。

每个任务均为无限循环的程序结构，在 UCOSII 中，定义了任务的 5 种状态，分别为休眠态、运行态、就绪态、挂起态、中断态。



休眠态：也即睡眠状态，表示任务以代码的形式驻留在内存中，但是不会被实时操作系统调用，也即实时操作系统中没有该任务的的控制块；

运行态：也即运行状态，表示任务掌握了 CPU 的主动权，CPU 在执行任务的代码；

就绪态：也即就绪状态，表示任务正在排队等待掌握 CPU，但是由于有更高优先级的任务正在运行，当前任务只能排队等待，即就绪态；

挂起态：也即等待状态，表示任务在掌握了 CPU 以后，代码运行到需要等待一个事件的发生，或者等待一个延时时间的到来的地方时，任务让出 CPU 的控制权，以让优先级低的任务得到 CPU，此时任务让出 CPU 后，即处于挂起态。

中断态：也即中断服务状态，表示当 CPU 接收到一个中断，进入了中断服务程序，此时

被中断打断的任务失去 CPU 的控制权，即该任务处于中断态。

2.06 任务切换

假设任务 A 为正在运行的任务，任务 B 为即将运行的任务，当从任务 A 切换到任务 B 时，需完成的工作为：

1) 将正在运行的任务 A 的断点处的 PC 寄存器，任务 A 的运行环境，即此时 CPU 各个寄存器的数值，压入任务 A 的堆栈中；
2) 从任务 B 的堆栈中，弹出任务 B 上次运行结束时的运行环境，即弹出 CPU 的各个寄存器数值；
3) 将任务 B 上次运行结束时的代码指针从任务 B 的堆栈中恢复到 PC 寄存器中，CPU 根据新的 PC 寄存器值，继续执行任务 B 的代码，即完成从任务 A 到任务 B 的切换。

其中对于各个 CPU 寄存器的入栈出栈操作，通过 CPU 的指令集可实现，但是对于任务 A 的运行指针 PC 压入任务 A 的堆栈，将任务 B 的断点指针从任务 B 的堆栈出栈至 PC 寄存器，这两个步骤的操作，对于一般的 CPU，没有直接的指令集实现，因此 UCOSII 一般是通过软件设定出一个中断，在进入中断与退出中断时，CPU 自动处理机制可实现对断点指针的压栈出栈操作。

2.09 不可剥夺型内核

基于优先级的实时内核可以分为 2 种，不可剥夺型内核，与可剥夺型内核。

不可剥夺型内核，指的是允许每个任务运行，直至该任务主动放弃 CPU 使用权，然后由内核调度当前就绪状态的最高优先级任务获得 CPU 使用权。若任务不主动放弃 CPU 使用权，则就绪状态的高优先级任务将一直得不得运行。

如果中断打断当前正在运行的任务，且中断让更高优先级任务就绪，中断返回后，继续运行被中断打断的低优先级任务，而在中断中就绪的高优先级任务，必须在当前运行任务主动放弃了 CPU 使用权后，才能得到 CPU 的使用权。

因此不可剥夺型内核的任务级响应时间是不确定的。

2.10 可剥夺型内核

可剥夺型内核，指的是最高优先级的任务一旦就绪，总能立即得到 CPU 的使用权，即高

优先级的任务可以剥夺低优先级任务的 CPU 使用权。如当中断服务程序中使得一个高优先级任务就绪，在退出中断服务程序后，立即运行这个就绪的高优先级任务，而不是去运行被中断的低优先级任务。

对于使用可剥夺型内核的应用程序，注意使用不可重入函数时需注意，即需满足互斥条件。否则函数中的数据可能被破坏，因此在应用程序编写时，尽量少用全局变量，以减小不可重入函数的个数，减少程序的缺陷。

UCOSII 属于可剥夺型内核。

2.11 可重入函数

可重入函数，指得是对于该函数，多个任务可调用，而不必担心某些数据被改写。而不可重入函数指的是如果多个任务调用该函数，则某些数据可能会被破坏，而使得调用的任务发生错误。

不可重入函数产生的原因是使用了全局变量。该全局变量可能在别的任务中被修改，因此函数尽量不使用全局变量，以避免出现不可重入函数，而如果必须使用全局变量，则可考虑使用互斥信号量、关闭中断等方式，来保护全局变量。

由于不可重入函数而导致的问题比较隐蔽，测试时难以发现，因此在编程时候需加以小心。

2.16 优先级反转

在实时系统中，会出现优先级反转的情况，具体示例如下：

1) 假设有三个任务，最高优先级的任务 1，次高优先级的任务 2，以及最低优先级的任务 3，还有一个共享资源 R；
2) 任务 1 与任务 2 开始时处于挂起状态，任务 3 正在运行，且任务 3 通过使用互斥信号量占用共享资源 R；
3) 任务 1 满足运行条件，剥夺了任务 3 的 CPU 使用权，开始运行；
4) 任务 1 在运行过程中，也需要使用共享资源 R，但是由于任务 3 正在占用，因此任务 1 挂起，等待任务 3 释放共享资源 R；
5) 任务 3 继续运行，若此时任务 2 满足运行条件，剥夺了任务 3 的 CPU 使用权，开始运行；

6) 任务 2 运行完成后，将 CPU 的使用权交回任务 3；

7) 任务 3 运行完成后，释放共享资源 R，将 CPU 使用权交给正在挂起的高优先级任务 1；

8) 任务 1 此时得到共享资源 R，继续其运行。

从以上的示例看出，由于共享资源 R 没有得到释放，高优先级任务 1 一直得不得运行，且任务 2 还剥夺了 CPU 的使用权，使得高优先级任务 1 的响应时间进一步恶化，即任务 1 与任务 2 的优先级发生了反转，低优先级的任务 2 反而得到的优先运行，高优先级的任务 1 被剥夺了运行权。

在实时操作系统中，需要避免出现优先级反转的情况，避免的办法是提高任务 3 的优先级，高于允许使用共享资源 R 的任何任务，这样当任务 3 运行完成，释放了共享资源 R 后，任务 1 能够立马得到运行。任务 3 运行完成后，再将其优先级降回原来。

2.18 互斥条件

多个任务在使用共享资源时候，需要满足互斥条件，即一个任务在占用共享资源时，另一个任务不可同时也使用该资源，否则系统会出现错误，影响系统的正确运行。

在实时操作系统中，使用以下方式来实现互斥条件：

1) 关中断；

2) 使用测试并置位指令；

3) 禁止任务切换；

4) 信号量。

1 关中断

处理共享资源的互斥性问题时，最简单快捷的办法就是关中断和开中断。一般任务会在处理完共享资源后，将自身挂起，进行任务切换，例外的就是当中断发生时，在中断服务函数中可能会使得更高优先级的任务就绪，中断返回后任务切换直接运行高优先级任务，被中断任务的共享资源未得到保护，系统容易出现错误。

但是需要注意关中断的时间不可太长，因为对于实时系统，关闭中断时间太长，会影响中断响应，关闭中断的时间越短越好。

```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
    .    /* 在这里处理共享数据 */
    .
    OS_EXIT_CRITICAL();
}
```

2 使用测试并置位指令

使用测试并置位指令，即是设置一个全局变量，对应某个共享资源，通过该全局变量来标示该共享资源是否可用。即 TAS（test and set）操作。

```
Disable interrupts;                          关中断
if ('Access Variable' is 0) {                如果资源未占用,标志为 0
    Set variable to 1;                       置资源已占用,标志为 1
    Reenable interrupts;                     重开中断
    Access the resource;                     处理该资源
    Disable interrupts;                      关中断
    Set the 'Access Variable' back to 0;     恢复资源为未占用,标志为 0
    Reenable interrupts;                     重新开中断
} else {                                     否则
    Reenable interrupts;                     开中断
    /* 资源不可使用,以后再试 */
}
```

3 禁止任务切换

当任务与中断服务程序没有共享数据或者其他共享资源时，可用通过禁止任务切换的方式实现互斥性。当在中断服务程序中使得某个高优先级任务就绪时，若之前禁止了任务切换，则返回中断后，不会立即运行高优先级任务。

释放掉共享资源后，再使能任务切换，在禁止任务切换期间实现了对共享资源的互斥性。

但是为了内核的正常运行，应该尽量避免使用该方法。

```
void Function (void)
{
    OSSchedLock();
    .
    .    /* 在这里处理共享数据(中断是开着的) */
    .
    OSSchedUnlock();
}
```

4 信号量

信号量是一种约定机制，可用于控制共享资源的使用权，标志某个事件的发生，或者同步 2 个任务。

信号量包括 3 种操作，信号量的建立（creat），等待信号量，也叫挂起（pend），给信号量，也叫发信号量（post）。

想得到信号量的任务，必须执行 pend 操作，如果该信号量有效，即该信号量值大于 0，则信号量值减去 1，任务得到了信号量，继续运行。如果该信号量值为 0，则等待信号量的任务被列入等待信号量任务表，继续等待该信号量。

当任务完成对信号量对应的共享任务的处理，或者任务需标志信号量对应的某个事件的发生，则必须执行 post 操作，以释放信号量。如果没有任务等待该信号量，则信号量的值加 1，如果有任务等待该信号量，那么就让该任务进入就绪状态，信号量的值不变。

收到信号量的任务可能是等待该信号量的任务中优先级最高的，或者是最早开始等待信号量的任务。UCOSII 使用前一个原则，即优先级最高的任务优先得到信号量。

```
OS_EVENT *SharedDataSem;
void Function (void)
{
    INT8U err;
    OSSemPend(SharedDataSem, 0, &err);
    .
    .    /*共享数据的处理在此进行(中断是开着的)*/
    .
    OSSemPost(SharedDataSem);
}
```

2.19 死锁

死锁指的是 2 个任务都互相等待对方占用的共享资源，使得 2 个任务均无法进行下去。比如任务 1 占用共享资源 R1，等待共享资源 R2，而任务 2 占用共享资源 R2，却等待着共享资源 R1，这样任务 1 与任务 2 均无法继续运行下去，系统出现了死锁的情况。

避免出现死锁的措施主要包括：对于每个任务，在等待到所有需要的共享资源后再开始运行；所有任务使用同样的顺序申请多个共享资源，且按照相反的顺序释放各个共享资源。

2.22 任务间通信

在任务与任务之间通信，或者中断服务程序与任务之间通信，可考虑使用全局变量的方式，或者使用发送消息的方式。

在使用全局变量的方式时，需要保证任务，或者中断服务程序独享该全局变量。不推荐使用全局变量，尽管其是比较方便快捷的通信方式。

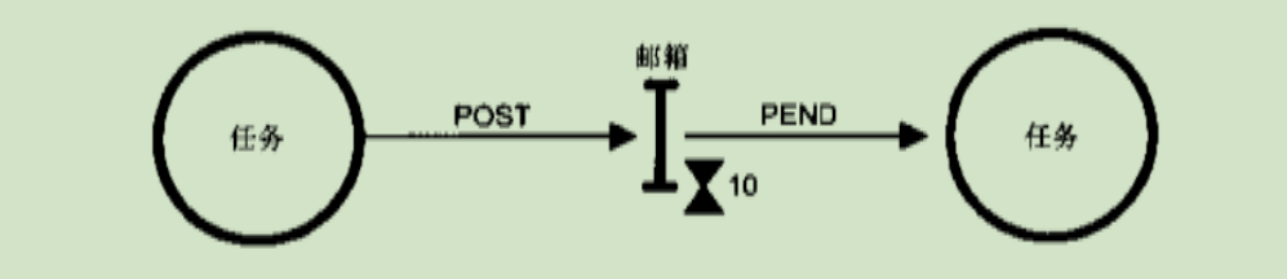
优先的方式是使用邮箱，或者消息队列实现任务之间的通信。

2.23 消息邮箱

消息邮箱是任务之间通信的一种方式，其实任务之间，或者任务与中断服务程序之间发送消息邮箱，就是传递了一个指向消息存放地址的指针变量。当任务或者中断服务程序收到了该消息邮箱，即是收到了指针变量，根据指针变量指向的地址位置，可读取传递的消息内容数据。

所谓的邮箱，是虚拟的存放指针变量的“容器”，在初始化时，可在该“容器”中存放一个有效的指针变量，或者存放空指针。将指针放入“容器”中即是 POST，等待指针进入“容器”中即是 PEND。

消息放入邮箱后，UCOSII 将把消息传送给等待该消息的任务列表中优先级最高的任务。



2.24 消息队列

消息队列也是任务之间通信的一种方式，消息队列即是由多个消息邮箱，按照顺序组成的队列，任务或者中断服务程序可从消息队列中取走一则消息，即指向消息存放地址的指针变量，或者将一则消息放入消息队列中。

对于消息队列中的多个消息，可以按照先进先出的原则（FIFO），即先进入消息队列的消息，先传递给任务，也可以按照后进先出的原则（LIFO），则最后进入消息队列的消息，先传递给任务，可将消息队列类比为堆栈，消息类比为存放堆栈的数据。消息放入消息队列类比为将数据压入堆栈，而从消息队列取出消息类比为将数据弹出堆栈。

每个消息队列都要一张等待任务列表，当消息队列中有消息时，按照 FIFO 原则，或者

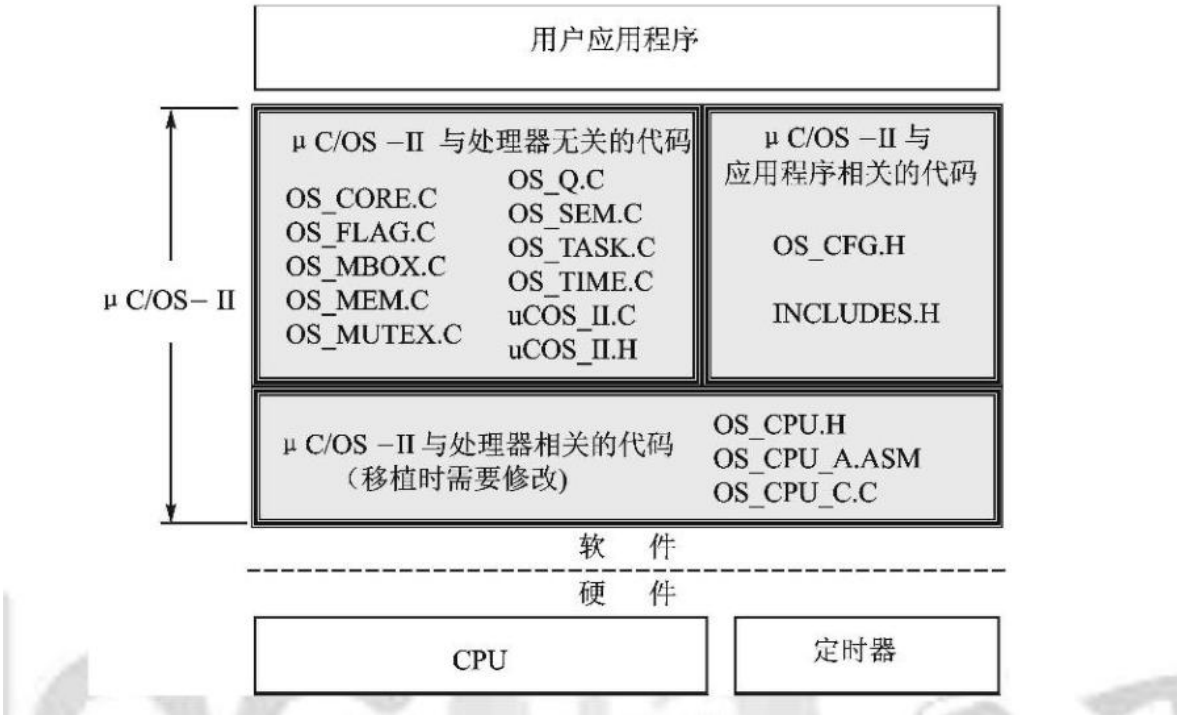
LIFO 原则将消息传递给等待任务列表中优先级最高的任务。

消息队列初始化时候为空，将一则消息放入消息队列中为 POST，而等待消息队列中的消息则为 PEND。

一个消息队列具有有限的空间存放消息，即消息队列长度是有限长。

3 内核结构

UCOSII 的内核结构如下图所示。



3.00 临界段 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()

在 UCOSII 中定义了两个宏来实现关中断和开中断操作，以保护临界代码，关中断的宏为 OS_ENTER_CRITICAL()，开中断的宏为 OS_EXIT_CRITICAL()。对于不同的 CPU，不同的指令集，不同的 C 编译器，这两个宏的实现方式均不相同，因此在移植 UCOSII 到不同的 CPU 时，这两个宏是需要改写的。

OS_ENTER_CRITICAL()与 OS_EXIT_CRITICAL()总是成对出现的，因为在实时操作系统中，长时间的关闭中断是不可靠的，更何况关掉中断而不再打开，特别是对于中断具有重要作用的嵌入式系统中，其后果可能是灾难性的。

```
{
    .
    .
    OS_ENTER_CRITICAL();
    /* μC/OS-II 临界段代码 */
    OS_EXIT_CRITICAL();
    .
    .
}
```

另外在使用这两个宏实现关闭与打开中断时，需要注意对系统时钟的影响，特别是对于UCOSII 使用了时钟中断来实现系统时钟时，关闭中断后，系统时钟将不再运行，此时若在调用系统函数，即延时等待函数之前关闭中断，则延时等待函数永远也等不得其延时时间的到来，系统将崩溃。因此需要遵循的原则是，在调用 UCOSII 系统函数时，保证中断是打开的状态。

OS_ENTER_CRITICAL()与 OS_EXIT_CRITICAL()可在 os_cpu.h 文件中找到。另外UCOSII 中定义了常数 OS_CRITICAL_METHOD 选择对于 OS_ENTER_CRITICAL()与OS_EXIT_CRITICAL()的实现方式，共有三种方式来实现开关中断。分别为直接使用 CPU 的开关中断指令实现，使用堆栈保存与恢复 CPU 的中断状态实现，或者是利用编译器扩展功能获得 CPU 中断状态，保存在局部变量 cpu_sr 中来实现宏定义。下面分别介绍其三种方式。

1 OS_CRITICAL_METHOD=1

当 OS_CRITICAL_METHOD=1 时，则使用的是直接使用 CPU 的开关中断指令实现宏OS_ENTER_CRITICAL()与 OS_EXIT_CRITICAL()的实现。

该方式的缺陷是，当调用宏 OS_EXIT_CRITICAL()以后，CPU 的中断肯定是打开的，而不管在关闭中断之前，CPU 的中断是打开或者关闭的状态。

2 OS_CRITICAL_METHOD=2

当 OS_CRITICAL_METHOD=2 时，则使用的是使用堆栈保存与恢复 CPU 的中断状态实现宏 OS_ENTER_CRITICAL()与 OS_EXIT_CRITICAL()的实现。

即在调用宏 OS_ENTER_CRITICAL()时，先在堆栈中保存 CPU 的中断开关状态，然后再关闭中断，在调用宏 OS_EXIT_CRITICAL()时，从堆栈中弹出 CPU 的中断开关状态。

示意性代码如下：

```
#define OS_ENTER_CRITICAL()          \
        asm(" PUSH     PSW")         \
        asm(" DI")

#define OS_EXIT_CRITICAL()           \
        asm(" POP      PSW")
```

3 OS_CRITICAL_METHOD=3

当 OS_CRITICAL_METHOD=3 时，利用编译器扩展功能获得 CPU 中断状态，保存在局

部变量 cpu_sr 中来实现宏 OS_ENTER_CRITICAL()与 OS_EXIT_CRITICAL()的实现。

保护与处理 CPU 的中断状态示意代码如下：

```
void Some_uCOS_II_Service (arguments)
{
    OS_CPU_SR  cpu_sr;                                    (1)

    .
    cpu_sr = get_processor_psw();                         (2)
    disable_interrupts();                                 (3)
    .
    /* 临界段代码 */                                      (4)
    .
    set_processor_psw(cpu_sr);                            (5)
    .
}
```

在应用程序中定义一个局部变量 cpu_sr，将 CPU 的中断状态 PSW 保存在局部变量 cpu_sr 中，关闭中断，开始执行临界代码，执行完成后，将局部变量 cpu_sr 中的缓存 CPU 的中断状态 PSW 恢复。

在将 UCOSII 移植到 STM32 处理器中时，使用的是第三种方式，具体代码如下：

```
//OS_CRITICAL_METHOD = 1 :直接使用处理器的开关中断指令来实现宏
//OS_CRITICAL_METHOD = 2 :利用堆栈保存和恢复CPU的状态
//OS_CRITICAL_METHOD = 3 :利用编译器扩展功能获得程序状态字，保存在局部变量cpu_sr

#define  OS_CRITICAL_METHOD   3      //进入临界段的方法

#if OS_CRITICAL_METHOD == 3
#define  OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();}
#define  OS_EXIT_CRITICAL()   {OS_CPU_SR_Restore(cpu_sr);}
#endif
```

```
OS_CPU_SR_Save
    MRS     R0, PRIMASK      ;读取PRIMASK到R0,R0为返回值
    CPSID   I                ;PRIMASK=1,关中断(NMI和硬件FAULT可以响应)
    BX      LR               ;返回

OS_CPU_SR_Restore
    MSR     PRIMASK, R0      ;读取R0到PRIMASK中,R0为参数
    BX      LR               ;返回
```

3.01 任务

在 UCOSII 中，一个任务一般为一个无限循环程序结构，任务循环执行任务代码，不同的任务通过实时操作系统 UCOSII 内核进行调度。

```
void YourTask (void *pdata)                                    (1)
{
   for (;;) {                                                  (2)
      /* 用户代码 */
      调用 μC/OS-II 的某些功能函数:
      OSFlagPend();
      OSMboxPend();
      OSMutexPend();
      OSQPend();
      OSSemPend();
      OSTaskDel(OS_PRIO_SELF);
      OSTaskSuspend(OS_PRIO_SELF);
      OSTimeDly();
      OSTimeDlyHMSM();
      /* 用户代码 */
   }
}
```

以上为任务的示例代码，由于任务为无限循环，没有更高级别的任务调用该任务，因此对于每个应用程序中的各个任务，其返回必须为 void。输入参数 pdata 为一个 void 类型的指针，通过该指针可传递任何类型的输入参数，如变量，结构体等。

任务完成后，可自我删除，示例代码如下：

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

UCOSII 最多可以管理 64 个任务，其中空闲任务与统计任务占用了最低的 2 个优先级任务，因此用户可使用最多 62 个任务，根据 UCOSII 作者建议，用户不要使用优先级最高 4 个与最低 4 个任务，即不要使用优先级为 0、1、2、3 与 OS_LOWEST_PRIO-3、OS_LOWEST_PRIO-2、OS_LOWEST_PRIO-1、OS_LOWEST_PRIO 的任务。

OS_LOWEST_PRIO 为最低优先级，其为常数，在 OS_CFG.H 文件中。

3.02 任务状态

上文已经介绍了任务的状态分为 5 种，分别为睡眠态、就绪态、运行态、等待态、与中断服务态。

任务还以代码形式缓存在程序空间中，没有交由 UCOSII 内核管理时，为睡眠态，当调用

函数 OSTaskCreate()或者 OSTaskCreateExt()来创建任务后，任务才可由 UCOSII 内核管理。而 OSTaskCreate()或者 OSTaskCreateExt()函数告诉内核任务的优先级，任务的堆栈空间，任务的代码地址位置等信息。

任务具备了被内核调用，可随时取得 CPU 运作权的状态，为任务的就绪态。任务可通过调用 OSTaskDel()返回到睡眠态。

任务取得了 CPU 的运行权，开始运行时，任务处于运行态，调用 OSStart()可以启动内核的任务调度，优先级最高的任务最先得到运行，即进入运行态。

当任务需要等待一个事件的发生，或者需要等待一段延时时间时，任务即进入等待态。正在运行的任务，可通过调用 OSTimeDly()或者 OSTimeDlyHMSM()，将自身延时等待一段时间，在延时期间，内核将任务挂起，进入等待态，通过调度运行其他任务。任务可通过 OSFlagPend()、OSSemPend()、OSMutexPend()、OSMboxPend()、OSQPend()来等待某一事件的发生，在等待事件发生的期间，任务处于等待态，即挂起。

正在运行的任务被中断打断了，该任务即进入中断服务态。

3.03 任务控制块

任务控制块 TCB 是一个结构体数据，表征任务的属性，结构体中含有多个数据，具体代码如下：

```
529 /*
530 *********************************************************************************************************
531 *                                          TASK CONTROL BLOCK
532 *********************************************************************************************************
533 */
534
535 typedef struct os_tcb {
536     OS_STK          *OSTCBStkPtr;           /* Pointer to current top of stack                        */
537
538 #if OS_TASK_CREATE_EXT_EN > 0u
539     void            *OSTCBExtPtr;           /* Pointer to user definable data for TCB extension       */
540     OS_STK          *OSTCBStkBottom;        /* Pointer to bottom of stack                             */
541     INT32U           OSTCBStkSize;          /* Size of task stack (in number of stack elements)       */
542     INT16U           OSTCBOpt;              /* Task options as passed by OSTaskCreateExt()            */
543     INT16U           OSTCBId;               /* Task ID (0..65535)                                     */
544 #endif
545
546     struct os_tcb   *OSTCBNext;             /* Pointer to next     TCB in the TCB list                */
547     struct os_tcb   *OSTCBPrev;             /* Pointer to previous TCB in the TCB list                */
548
549 #if (OS_EVENT_EN)
550     OS_EVENT        *OSTCBEventPtr;         /* Pointer to          event control block                */
551 #endif
552
553 #if (OS_EVENT_EN) && (OS_EVENT_MULTI_EN > 0u)
554     OS_EVENT       **OSTCBEventMultiPtr;    /* Pointer to multiple event control blocks               */
555 #endif
```

```
556
557 #if ((OS_Q_EN > 0u) && (OS_MAX_QS > 0u)) || (OS_MBOX_EN > 0u)
558     void            *OSTCBMsg;              /* Message received from OSMboxPost() or OSQPost()          */
559 #endif
560
561 #if (OS_FLAG_EN > 0u) && (OS_MAX_FLAGS > 0u)
562 #if OS_TASK_DEL_EN > 0u
563     OS_FLAG_NODE    *OSTCBFlagNode;         /* Pointer to event flag node                               */
564 #endif
565     OS_FLAGS         OSTCBFlagsRdy;         /* Event flags that made task ready to run                  */
566 #endif
567
568     INT32U           OSTCBDly;              /* Nbr ticks to delay task or, timeout waiting for event    */
569     INT8U            OSTCBStat;             /* Task      status                                         */
570     INT8U            OSTCBStatPend;         /* Task PEND status                                         */
571     INT8U            OSTCBPrio;             /* Task priority (0 == highest)                             */
572
573     INT8U            OSTCBX;                /* Bit position in group  corresponding to task priority    */
574     INT8U            OSTCBY;                /* Index into ready table corresponding to task priority    */
575     OS_PRIO          OSTCBBitX;             /* Bit mask to access bit position in ready table           */
576     OS_PRIO          OSTCBBitY;             /* Bit mask to access bit position in ready group           */
577
578 #if OS_TASK_DEL_EN > 0u
579     INT8U            OSTCBDelReq;           /* Indicates whether a task needs to delete itself          */
580 #endif
581
582 #if OS_TASK_PROFILE_EN > 0u
```

```
583     INT32U           OSTCBCtxSwCtr;         /* Number of time the task was switched in                  */
584     INT32U           OSTCBCyclesTot;        /* Total number of clock cycles the task has been running   */
585     INT32U           OSTCBCyclesStart;      /* Snapshot of cycle counter at start of task resumption    */
586     OS_STK          *OSTCBStkBase;          /* Pointer to the beginning of the task stack               */
587     INT32U           OSTCBStkUsed;          /* Number of bytes used from the stack                      */
588 #endif
589
590 #if OS_TASK_NAME_EN > 0u
591     INT8U           *OSTCBTaskName;
592 #endif
593
594 #if OS_TASK_REG_TBL_SIZE > 0u
595     INT32U           OSTCBRegTbl[OS_TASK_REG_TBL_SIZE];
596 #endif
597 } OS_TCB;
```

各个数据的含义如下：

1 OSTCBStkPtr

指向当前任务堆栈栈顶的指针。

每个任务在运行时候，其堆栈内容是变化的，其栈顶指针随着也变化。当进行任务切换时，需要使用该指针，使用任务堆栈来保存当前任务的运行环境等数据。

2 OSTCBExtPtr

指向用户定义的任务控制块扩展。

该指针只能在函数 OsTaskCreatExt()中使用，需要将宏定义 OS_TASK_CREATE_EXT_EN 设置为 1。

当用户需要对任务控制功能进行扩展时，可使用该指针指向一个数据结构，通过该数据结构可跟踪任务的执行时间，任务的切换次数等，实现任务功能的扩展。

3 OSTCBStkBottom

指向任务堆栈栈底的指针。

各个 CPU 所使用的堆栈递增方向是不相同的，如果 CPU 的堆栈指针是递减的，即堆栈空间是从高地址向低地址方向分配，则 OSTCBStkBottom 指向任务堆栈空间的最低地址处。反之，若 CPU 的堆栈指针是递减增的，即堆栈空间是从低地址向高地址方向分配，则 OSTCBStkBottom 指向任务堆栈空间的最高地址处。

该参数只能在函数 OsTaskCreatExt()中使用，需要将宏定义 OS_TASK_CREATE_EXT_EN 设置为 1。

4 OSTCBStkSize

任务堆栈可容纳的指针元数目。

若堆栈可以容纳的指针元数目是 1000，若每个地址宽度是 32 位，则实际堆栈容量是 4000 字节。若每个地址宽度是 16 位，则实际堆栈容量是 2000 字节，若每个地址宽度是 8 位，则实际堆栈容量是 1000 字节。

该参数只能在函数 OsTaskCreatExt()中使用，需要将宏定义 OS_TASK_CREATE_EXT_EN 设置为 1。

5 OSTCBOpt

使用函数 OsTaskCreatExt()创建任务时的选择项。

UCOSII 使用 3 个选项，分别为 OS_TASK_OPT_STK_CHK，OS_TASK_OPT_STK_CLR，OS_TASK_OPT_SAVE_FP。

OS_TASK_OPT_STK_CHK 用于告之 OsTaskCreatExt()，在任务创建时，检测任务堆栈。

OS_TASK_OPT_STK_CLR 表示任务创建时，对任务堆栈所有空间清零。

OS_TASK_OPT_SAVE_FP 用于告之 OsTaskCreatExt()，任务将要做浮点运算，在建立的任务做任务切换调度时，浮点寄存器的内容需要保存。

该参数只能在函数 OsTaskCreatExt()中使用，需要将宏定义 OS_TASK_CREATE_EXT_EN 设置为 1。

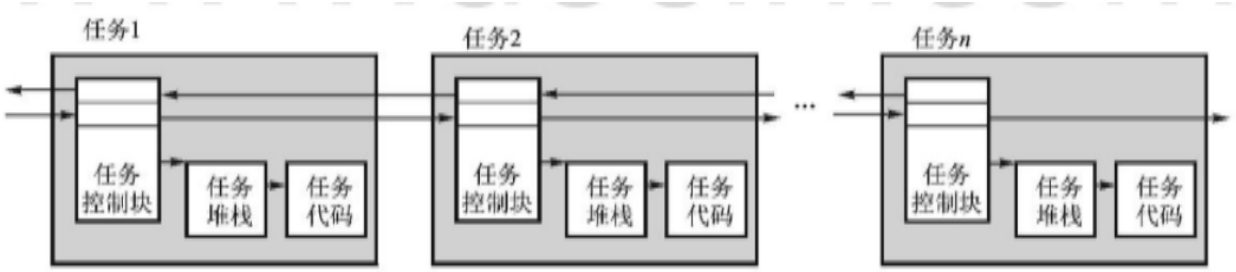
```
/*
*********************************************************************************************************
*                                       TASK OPTIONS (see OSTaskCreateExt())
*********************************************************************************************************
*/
#define  OS_TASK_OPT_NONE          0x0000u  /* NO option selected                                    */
#define  OS_TASK_OPT_STK_CHK       0x0001u  /* Enable stack checking for the task                    */
#define  OS_TASK_OPT_STK_CLR       0x0002u  /* Clear the stack when the task is create               */
#define  OS_TASK_OPT_SAVE_FP       0x0004u  /* Save the contents of any floating-point registers     */
```

6 OSTCBId

任务的 ID 编号，该参数在 UCOSII 暂未使用。

7 OSTCBNext 与 OSTCBPrev

用于任务控制块 TCB 双向链表的前向链接与后向链接。



每个任务的任务控制块 OS_TCB 在任务创建时，链接到任务链表中，在任务删除时，从链表中删除。双重链表可以使得任一成员能被快速的插入或者删除。

8 OSTCBEventPtr

指向事件控制块的指针。

9 OSTCBEventMultiPtr

指向多个事件控制块的指针。

只有在 OS_EVENT_EN 与 OS_EVENT_MULTI_EN 均为 1 时，该参数才有效使用，其中宏 OS_EVENT_MULTI_EN 表示 UCOSII 包含函数 OSEventPendMulti()。该函数为新版本中的新增函数。

10 OSTCBMsg

指向传递给任务的消息的指针。

11 OSTCBFlagNode

指向事件标志节点的指针。

12 OSTCBFlagsRdy

当任务等等事件标志组时，OSTCBFlagsRdy 是使得任务进入就绪状态的事件标志。

13 OSTCBDly

表示任务允许等待事件发生的最多时钟节拍数，或者任务延时的最多时钟节拍数。

当 OSTCBDly 为 0 时，表示任务不延时，或者等待事情发生的时间没有限制。

14 OSTCBStat

任务的状态字。

具体的任务状态字表示如下代码所示。

```
/*
*********************************************************************************************************
*                                TASK STATUS (Bit definition for OSTCBStat)
*********************************************************************************************************
*/
#define  OS_STAT_RDY                0x00u  /* Ready to run                                             */
#define  OS_STAT_SEM                0x01u  /* Pending on semaphore                                     */
#define  OS_STAT_MBOX               0x02u  /* Pending on mailbox                                       */
#define  OS_STAT_Q                  0x04u  /* Pending on queue                                         */
#define  OS_STAT_SUSPEND            0x08u  /* Task is suspended                                        */
#define  OS_STAT_MUTEX              0x10u  /* Pending on mutual exclusion semaphore                    */
#define  OS_STAT_FLAG               0x20u  /* Pending on event flag group                              */
#define  OS_STAT_MULTI              0x80u  /* Pending on multiple events                               */

#define  OS_STAT_PEND_ANY         (OS_STAT_SEM | OS_STAT_MBOX | OS_STAT_Q | OS_STAT_MUTEX | OS_STAT_FLAG)

```

OS_STAT_RDY 表示任务处于就绪态；

OS_STAT_SEM 表示任务在等待信号量；

OS_STAT_MBOX 表示任务在等待消息邮箱；

OS_STAT_Q 表示任务在等待消息队列；

OS_STAT_SUSPEND 表示任务处于挂起态；

OS_STAT_MUTEX 表示任务在等待互斥型信号量；

OS_STAT_FLAG 表示任务在等待事件标志组；

OS_STAT_MULTI 表示任务在等待多个事件；

OS_STAT_PEND_ANY 表示任务在等待信号量，或者消息邮箱，或者消息队列，或者互斥型信号量，或者事件标志组。

15 OSTCBStatPend

任务等待状态字。

具体的任务等待状态字如下代码：

```
/*
*********************************************************************************************************
*                                TASK PEND STATUS (Status codes for OSTCBStatPend)
*********************************************************************************************************
*/
#define  OS_STAT_PEND_OK              0u  /* Pending status OK, not pending, or pending complete     */
#define  OS_STAT_PEND_TO              1u  /* Pending timed out                                       */
#define  OS_STAT_PEND_ABORT           2u  /* Pending aborted                                         */

```

OS_STAT_PEND_OK 表示任务等待状态好，即任务未等待任何事件，或者已经等待完成；

OS_STAT_PEND_TO 表示任务等待超时；

OS_STAT_PEND_ABORT 表示任务等待中止。

16 OSTCBPrio

任务的优先级。

数值越小，则任务的优先级越高。

17 OSTCBX、OSTCBY、OSTCBBitX 与 OSTCBBitY

用于加速任务进入就绪态的过程，或者进入等待事件发生状态过程的计算变量。

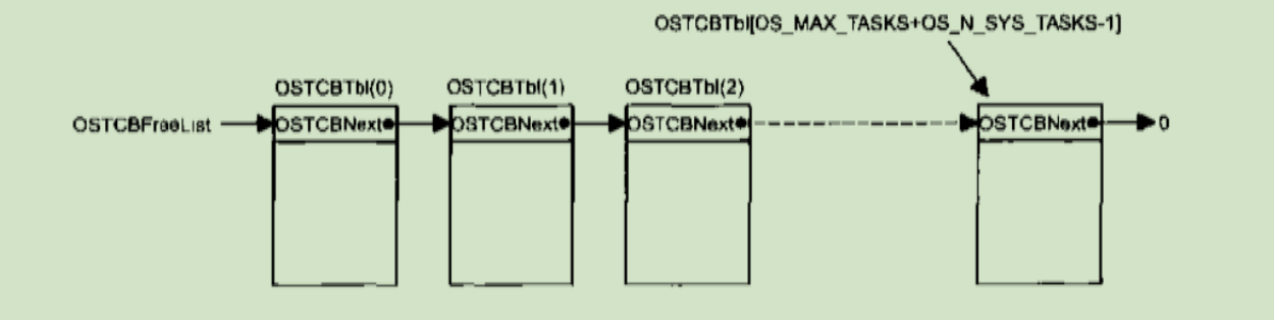
18 OSTCBDelReq

表示该任务是否需要被删除。

宏定义 OS_MAX_TASKS 定义了应用程序中最多的任务数量，即确定了该应用程序最多的任务控制块 TCB 的数目。在 UCOSII 中，所有的任务控制块 TCB 均是放在任务控制块列表数组 OSTCBTbl[]中。

宏定义 OS_N_SYS_TASKS 表示分配的系统任务数量，系统任务一般包括空闲任务与统计任务。

在 UCOSII 初始化时，所有的任务控制块 TCB 被链接成单向的空任务链表，任务一旦建立，空任务控制块指针 OSTCBFreeList 指向的任务控制块 TCB 被赋予给该新建立的任务，然后 OSTCBFreeList 指向下一个空的任务控制块。任务被删除后，任务控制块返回给空任务链表。



函数 OS_TCBInit()用来初始化任务控制块，而函数 OSTaskCreat()或者函数 OSTaskCreatExt()调用 OS_TCBInit()初始化任务控制块。

函数 OS_TCBInit()的代码如下：

```
1874 /*
1875 *********************************************************************************************************
1876 *                                          INITIALIZE TCB
1877 *
1878 * Description: This function is internal to uC/OS-II and is used to initialize a Task Control Block when
1879 *              a task is created (see OSTaskCreate() and OSTaskCreateExt()).
1880 *
1881 * Arguments  : prio          is the priority of the task being created
1882 *
1883 *              ptos          is a pointer to the task's top-of-stack assuming that the CPU registers
1884 *                            have been placed on the stack.  Note that the top-of-stack corresponds to a
1885 *                            'high' memory location is OS_STK_GROWTH is set to 1 and a 'low' memory
1886 *                            location if OS_STK_GROWTH is set to 0.  Note that stack growth is CPU
1887 *                            specific.
1888 *
1889 *              pbos          is a pointer to the bottom of stack.  A NULL pointer is passed if called by
1890 *                            'OSTaskCreate()'.
1891 *
1892 *              id            is the task's ID (0..65535)
1893 *
1894 *              stk_size      is the size of the stack (in 'stack units').  If the stack units are INT8Us
1895 *                            then, 'stk_size' contains the number of bytes for the stack.  If the stack
1896 *                            units are INT32Us then, the stack contains '4 * stk_size' bytes.  The stack
1897 *                            units are established by the #define constant OS_STK which is CPU
1898 *                            specific.  'stk size' is 0 if called by 'OSTaskCreate()'.
```

```
1900 *              pext          is a pointer to a user supplied memory area that is used to extend the task
1901 *                            control block.  This allows you to store the contents of floating-point
1902 *                            registers, MMU registers or anything else you could find useful during a
1903 *                            context switch.  You can even assign a name to each task and store this name
1904 *                            in this TCB extension.  A NULL pointer is passed if called by OSTaskCreate().
1905 *
1906 *              opt           options as passed to 'OSTaskCreateExt()' or,
1907 *                            0 if called from 'OSTaskCreate()'.
1908 *
1909 * Returns    : OS_ERR_NONE         if the call was successful
1910 *              OS_ERR_TASK_NO_MORE_TCB  if there are no more free TCBs to be allocated and thus, the task cannot
1911 *                                    be created.
1912 *
1913 * Note       : This function is INTERNAL to uC/OS-II and your application should not call it.
1914 *********************************************************************************************************
1915 */
```

```
1917 INT8U  OS_TCBInit (INT8U    prio,
1918                    OS_STK  *ptos,
1919                    OS_STK  *pbos,
1920                    INT16U   id,
1921                    INT32U   stk_size,
1922                    void    *pext,
1923                    INT16U   opt)
1924 {
1925     OS_TCB    *ptcb;
1926 #if OS_CRITICAL_METHOD == 3u                          /* Allocate storage for CPU status register */
1927     OS_CPU_SR  cpu_sr = 0u;
1928 #endif
1929 #if OS_TASK_REG_TBL_SIZE > 0u
1930     INT8U      i;
1931 #endif
1932
1933
1934     OS_ENTER_CRITICAL();
1935     ptcb = OSTCBFreeList;                            /* Get a free TCB from the free TCB list    */
1936     if (ptcb != (OS_TCB *)0) {
1937         OSTCBFreeList            = ptcb->OSTCBNext;  /* Update pointer to free TCB list          */
1938         OS_EXIT_CRITICAL();
1939         ptcb->OSTCBStkPtr        = ptos;             /* Load Stack pointer in TCB                */
1940         ptcb->OSTCBPrio          = prio;             /* Load task priority into TCB              */
1941         ptcb->OSTCBStat          = OS_STAT_RDY;      /* Task is ready to run                     */
1942         ptcb->OSTCBStatPend      = OS_STAT_PEND_OK;  /* Clear pend status                        */
1943         ptcb->OSTCBDly           = 0u;               /* Task is not delayed                      */
```

```
1945 #if OS_TASK_CREATE_EXT_EN > 0u
1946         ptcb->OSTCBExtPtr        = pext;                /* Store pointer to TCB extension           */
1947         ptcb->OSTCBStkSize       = stk_size;            /* Store stack size                         */
1948         ptcb->OSTCBStkBottom     = pbos;                /* Store pointer to bottom of stack         */
1949         ptcb->OSTCBOpt           = opt;                 /* Store task options                       */
1950         ptcb->OSTCBId            = id;                  /* Store task ID                            */
1951 #else
1952         pext                     = pext;                /* Prevent compiler warning if not used     */
1953         stk_size                 = stk_size;
1954         pbos                     = pbos;
1955         opt                      = opt;
1956         id                       = id;
1957 #endif
1958
1959 #if OS_TASK_DEL_EN > 0u
1960         ptcb->OSTCBDelReq        = OS_ERR_NONE;
1961 #endif
1962
1963 #if OS_LOWEST_PRIO <= 63u                                   /* Pre-compute X, Y                 */
1964         ptcb->OSTCBY             = (INT8U)(prio >> 3u);
1965         ptcb->OSTCBX             = (INT8U)(prio & 0x07u);
1966 #else                                                       /* Pre-compute X, Y                 */
1967         ptcb->OSTCBY             = (INT8U)((INT8U)(prio >> 4u) & 0xFFu);
1968         ptcb->OSTCBX             = (INT8U) (prio & 0x0Fu);
1969 #endif
1970                                                            /* Pre-compute BitX and BitY        */
1971         ptcb->OSTCBBitY          = (OS_PRIO)(1uL << ptcb->OSTCBY);
1972         ptcb->OSTCBBitX          = (OS_PRIO)(1uL << ptcb->OSTCBX);
1973
1974 #if (OS_EVENT_EN)
1975         ptcb->OSTCBEventPtr      = (OS_EVENT  *)0;          /* Task is not pending on an  event         */
1976 #if (OS_EVENT_MULTI_EN > 0u)
1977         ptcb->OSTCBEventMultiPtr = (OS_EVENT **)0;          /* Task is not pending on any events        */
1978 #endif
1979 #endif
1980
1981 #if (OS_FLAG_EN > 0u) && (OS_MAX_FLAGS > 0u) && (OS_TASK_DEL_EN > 0u)
1982         ptcb->OSTCBFlagNode  = (OS_FLAG_NODE *)0;           /* Task is not pending on an event flag     */
1983 #endif
1984
1985 #if (OS_MBOX_EN > 0u) || ((OS_Q_EN > 0u) && (OS_MAX_QS > 0u))
1986         ptcb->OSTCBMsg       = (void *)0;                   /* No message received                      */
1987 #endif
1988
1989 #if OS_TASK_PROFILE_EN > 0u
1990         ptcb->OSTCBCtxSwCtr    = 0uL;                       /* Initialize profiling variables           */
1991         ptcb->OSTCBCyclesStart = 0uL;
1992         ptcb->OSTCBCyclesTot   = 0uL;
1993         ptcb->OSTCBStkBase     = (OS_STK *)0;
1994         ptcb->OSTCBStkUsed     = 0uL;
1995 #endif
1996
1997 #if OS_TASK_NAME_EN > 0u
1998         ptcb->OSTCBTaskName    = (INT8U *)(void *)"?";
1999 #endif
2000
2001 #if OS_TASK_REG_TBL_SIZE > 0u                               /* Initialize the task variables            */
2002     for (i = 0u; i < OS_TASK_REG_TBL_SIZE; i++) {
2003         ptcb->OSTCBRegTbl[i] = 0u;
2004     }
2005 #endif
2006
```

```
2007            OSTCBInitHook(ptcb);
2008
2009            OSTaskCreateHook(ptcb);                             /* Call user defined hook                   */
2010
2011            OS_ENTER_CRITICAL();
2012            OSTCBPrioTbl[prio] = ptcb;
2013            ptcb->OSTCBNext    = OSTCBList;                     /* Link into TCB chain                      */
2014            ptcb->OSTCBPrev    = (OS_TCB *)0;
2015            if (OSTCBList != (OS_TCB *)0) {
2016                OSTCBList->OSTCBPrev = ptcb;
2017            }
2018            OSTCBList               = ptcb;
2019            OSRdyGrp               |= ptcb->OSTCBBitY;          /* Make task ready to run                   */
2020            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
2021            OSTaskCtr++;                                        /* Increment the #tasks counter             */
2022            OS_EXIT_CRITICAL();
2023            return (OS_ERR_NONE);
2024        }
2025        OS_EXIT_CRITICAL();
2026        return (OS_ERR_TASK_NO_MORE_TCB);
2027    }
```

从以上代码可知，函数 OS_TCBInit()的输入参数如下：

输入参数 prio 是创建任务的优先级；

输入参数 ptos 是建立任务堆栈后，指向堆栈栈顶的指针。如果宏定义 OS_STK_GROWTH 为 1，则 CPU 的堆栈指针是递减的，即堆栈空间是从高地址向低地址方向分配，栈顶的指针是高地址。反之，如果宏定义 OS_STK_GROWTH 为 0，则 CPU 的堆栈指针是递增的，即堆栈空间是从低地址向高地址方向分配，栈顶的指针是低地址；

输入参数 pbos 是建立任务堆栈后，指向堆栈栈底的指针。

输入参数 id 是任务的 ID，范围是 0~65535，即 0~FFFFH；

输入参数 stk_size 是任务堆栈的容量，其单位是由数据类型 OS_STK 决定的，可为 INT8U，或者 INT16U，或者 INT32U。OS_STK 在文件 os_cpu.h 中定义。

输入参数 pext 是任务功能扩展指针，指向用户提供的扩展内存区域，当用户需要对任务控制功能进行扩展时，可使用该指针指向一个数据结构，通过该数据结构可跟踪任务的执行时间，任务的切换次数等，实现任务功能的扩展。

输入参数 opt 是传递给函数 OSTaskCreatExt()的选项参数。

返回参数为 OS_ERR_NONE 表示任务控制块初始化成功，如果函数的返回值为 OS_ERR_TASK_NO_MORE_TCB 表示没有空任务链表中没有多余的任务控制块，任务创建失败。

函数 OS_TCBInit()的主要工作为从空任务链表中取得一个任务控制块 TCB，根据函数输入参数对任务控制块 TCB 中各个参数进行初始化设置，调用用户定义的钩子函数，将初始化好的 TCB 链接入 TCB 链表中，将任务设置为就绪态，等待调度器进行调度，返回任务控制块 TCB 初始化成功参数。

3.04 就绪表

UCOSII 根据任务的优先级，通过查表方式，快速的置任务为就绪状态，或者清除任务的就绪态，或者查找就绪态优先级最高的任务。内核涉及的主要参数变量如下：

OSRdyGrp：任务就绪组；

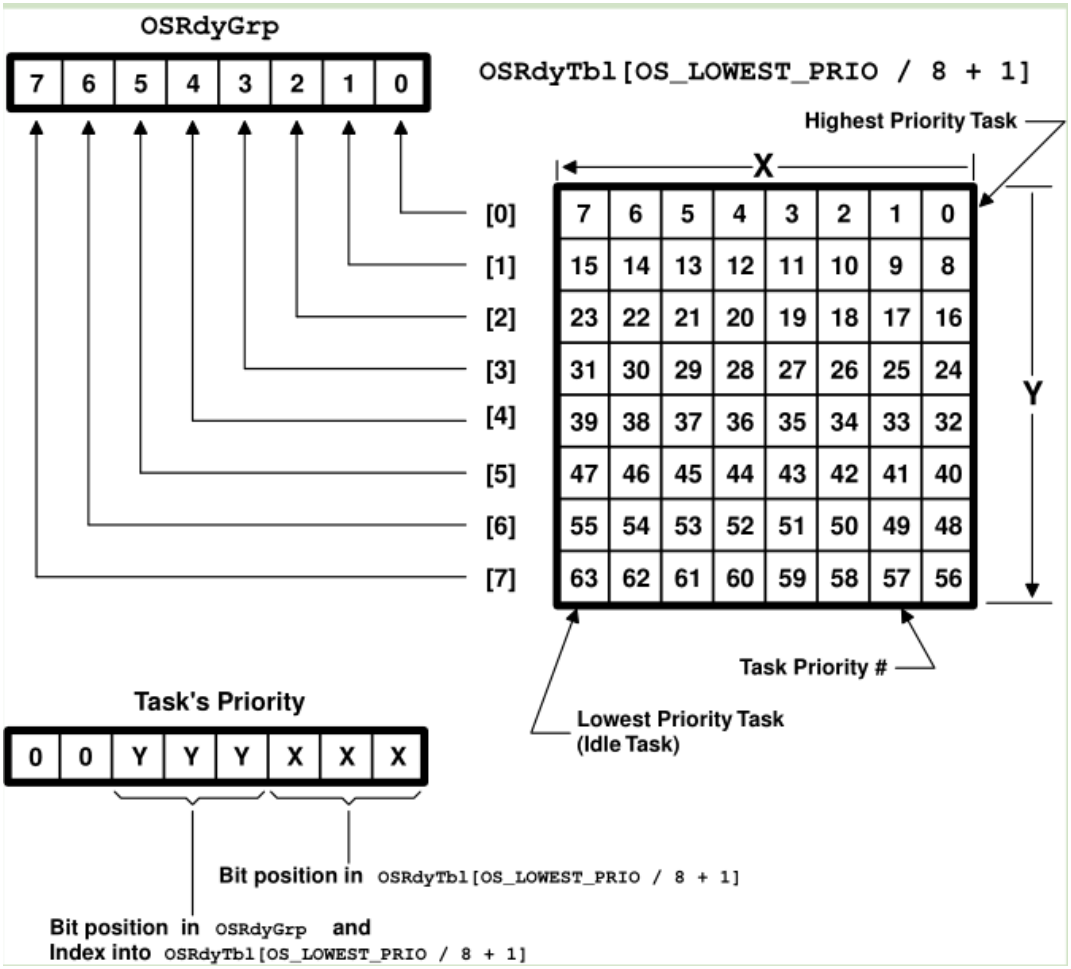
OSRdyTbl[]：任务就绪表；

OSMapTbl[]：掩码查询表 1；

OSUnMapTbl[]：掩码查询表 2；

prio：任务优先级；

任务就绪组 OSRdyGrp 与任务就绪表 OSRdyTbl[]的关系如下：



由上图关系可知，任务就绪表 OSRdyTbl[]中每个元素表示对应优先级任务的就绪状态，若任务就绪则置 1，否则为 0。OSRdyTbl[]的元素个数由宏 OS_LOWEST_PRIO 决定，UCOSII 最多有 64 个任务，即任务优先级最大范围为 0~63。

OSRdyGrp 为任务就绪组，若 OSRdyTbl[0]中任何一位为 1，即任务优先级 0~7 的任务至

少一个任务就绪，则 OSRdyGrp 的 bit0 为 1。若 OSRdyTbl[1]中任何一位为 1，即任务优先级 8~15 的任务至少一个任务就绪，则 OSRdyGrp 的 bit1 为 1，依次类推。

另外从图中可知，任务的优先级 prio 的 bit0~bit2 表示 OSRdyTbl[]中每个元素的 bit 位置，而任务的优先级 prio 的 bit3~bit5 表示 OSRdyGrpd 的 bit 位置，并且还表示 OSRdyTbl[]的下标。由于 UCOSII 中最大的任务优先级为 63，则任务的优先级 prio 的 bit6~bit7 恒为 0。

使得任务进入就绪态，则需要将任务就绪表 OSRdyTbl[]中对应任务的位置为 1，且需将任务就绪组 OSRdyGrp 对应的位置 1 即可，在内核中，是通过查表的方式进行快速置位操作，使用的掩码查询表为 OSMapTbl[]。其内容如下：

Table 3.2 Contents of OSMapTbl[].

Index	*Bit Mask (Binary)*
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

因此使得一个任务优先级为 prio 的任务就绪的代码如下：

Listing 3.7 Making a task ready to run.

```
OSRdyGrp               |= OSMapTbl[prio >> 3];
OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

同样的，使得任务脱离就绪态的代码如下：

Listing 3.8 Removing a task from the ready list.

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

以上代码将 OSRdyTbl[]中相应的元素置 0，只有当 OSRdyTbl[]对应任务组中所有的任务均不是就绪态，则才将 OSRdyGrp 对应位置 0。

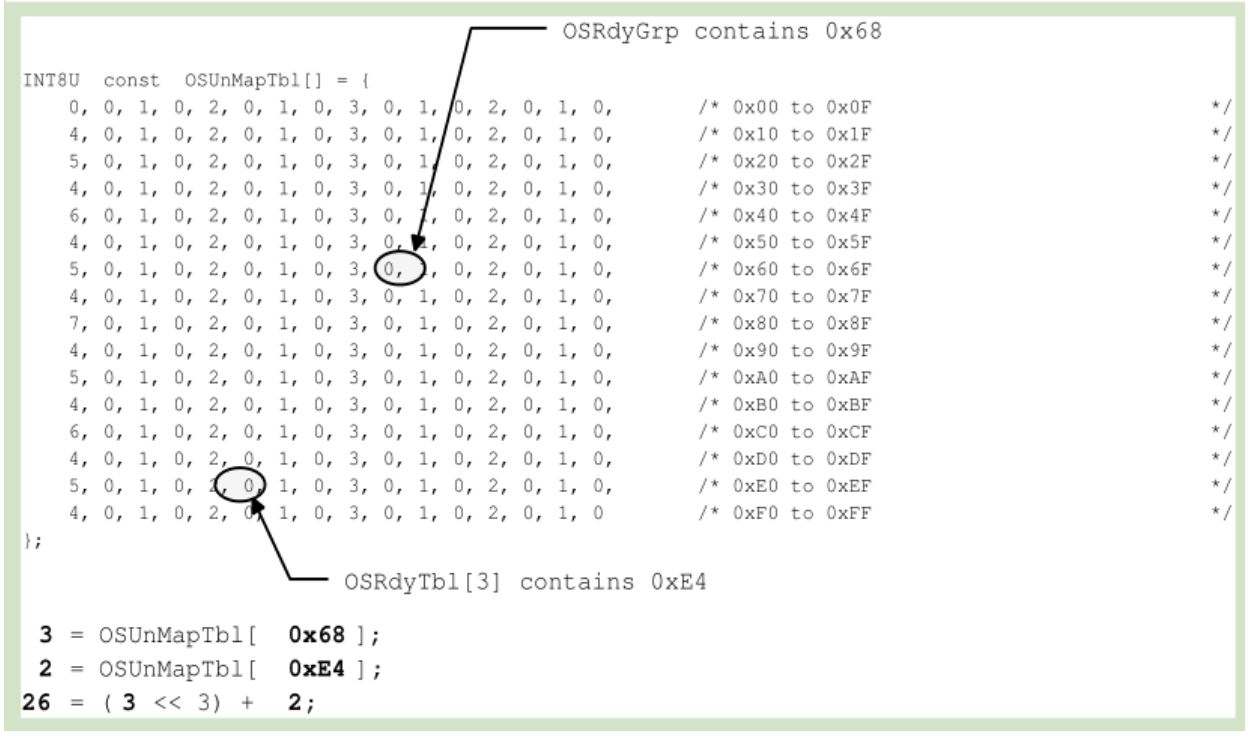
为了快速的查找出就绪态任务中优先级最高的任务，需要借组掩码查询表 OSUnMapTbl[]，该查询表返回值为 0~7，通过返回值可确定就绪态任务中优先级最高的任务的优先级号。查找就绪态任务中优先级最高的任务的代码如下：

Listing 3.9 Finding the highest priority task ready to run.

```
y    = OSUnMapTbl[OSRdyGrp];    /* Determine Y position in OSRdyTbl[]  */
x    = OSUnMapTbl[OSRdyTbl[y]]; /* Determine X position in OSRdyTbl[Y] */
prio = (y << 3) + x;
```

查找出就绪态任务中优先级最高的任务的范例如下：

Figure 3.5 Finding the highest priority task ready to run.



掩码查询表 OSUnMapTbl[]的编码规则是列出 0~255 每个数值的最低位为 1 的 bit 位数，例如 01H 的最低位为 1 的是 bit0，则返回为 0，如 10H 的最低位为 1 的是 bit4，则返回为 4。利用此查询表的规则，则查询数值为 OSRdyGrp 则返回的是其最低位为 1 的 bit 数，即优先级最高的就绪态任务所在的任务组号 Y，在查询数值为 OSRdyTbl[Y]则返回该任务组中最低位为 1 的 bit 号 X，则 X 为就绪态任务中优先级最高任务的优先级号的 bit0~bit2，Y 为就绪态任务中优先级最高任务的优先级号的 bit3~bit5。从而得到就绪态任务中优先级最高任务的优先级号 prio。

根据 prio 查询任务控制块优先级表 OSTCBPrioTbl[]即可得到指向该任务控制块 TCB 的指针。

3.05 任务调度

任务调度主要完成查找并运行就绪态任务中优先级最高的任务。任务级的调度函数为OSSched()，中断级的调度函数为 OSIntExt()。

调度函数 OSSched()的代码如下：

Listing 3.10　Task scheduler.

```
void  OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      y;


    OS_ENTER_CRITICAL();
    if ((OSIntNesting == 0) && (OSLockNesting == 0)) {                          (1)
        y             = OSUnMapTbl[OSRdyGrp];                                   (2)
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {                                       (3)
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];                         (4)
            OSCtxSwCtr++;                                                       (5)
            OS_TASK_SW();                                                       (6)
        }
    }
    OS_EXIT_CRITICAL();
}
```

变量 OSTCBHighRdy 指向就绪态任务中优先级最高的任务控制块 TCB。统计计数器 OSCtxSwCtr 跟踪任务切换的次数，只是让用户知道进行了多少次的任务切换。

宏 OS_TASK_SW()完成实际上的任务切换。任务切换的过程实际就是将挂起的任务的运行环境压入任务堆栈中，然后将就绪态任务中优先级最高的任务的堆栈中的 CPU 寄存器等数据恢复。

在 STM32 的移植中，宏 OS_TASK_SW()定义为宏 OSCtxSw()，而宏 OSCtxSw()的任务主要是简单的通过设定中断控制和状态寄存器，实现触发 PendSV 中断，然后返回，在 PendSV 中断函数 PendSV_Handler 中，实现任务运行环境的压栈与出栈操作，即实现任务切换。

函数 PendSV_Handler 的伪代码如下：

```
OS_CPU_PendSVHandler:
        if (PSP != NULL) {                              (1)
                Save R4-R11 onto task stack;            (2)
                OSTCBCur->OSTCBStkPtr = SP;             (3)
        }
        OSTaskSwHook();                                 (4)
        OSPrioCur = OSPrioHighRdy;                      (5)
        OSTCBCur  = OSTCBHighRdy;                       (6)
        PSP       = OSTCBHighRdy->OSTCBStkPtr;          (7)
        Restore R4-R11 from new task stack;             (8)
        Return from exception;                          (9)
```

3.06 任务级的任务切换

任务切换为 context switch，而 context 就是 CPU 中的各个寄存器内容，任务切换即是恢复任务在 CPU 使用权被剥夺时保存下来的全部寄存器的数值。

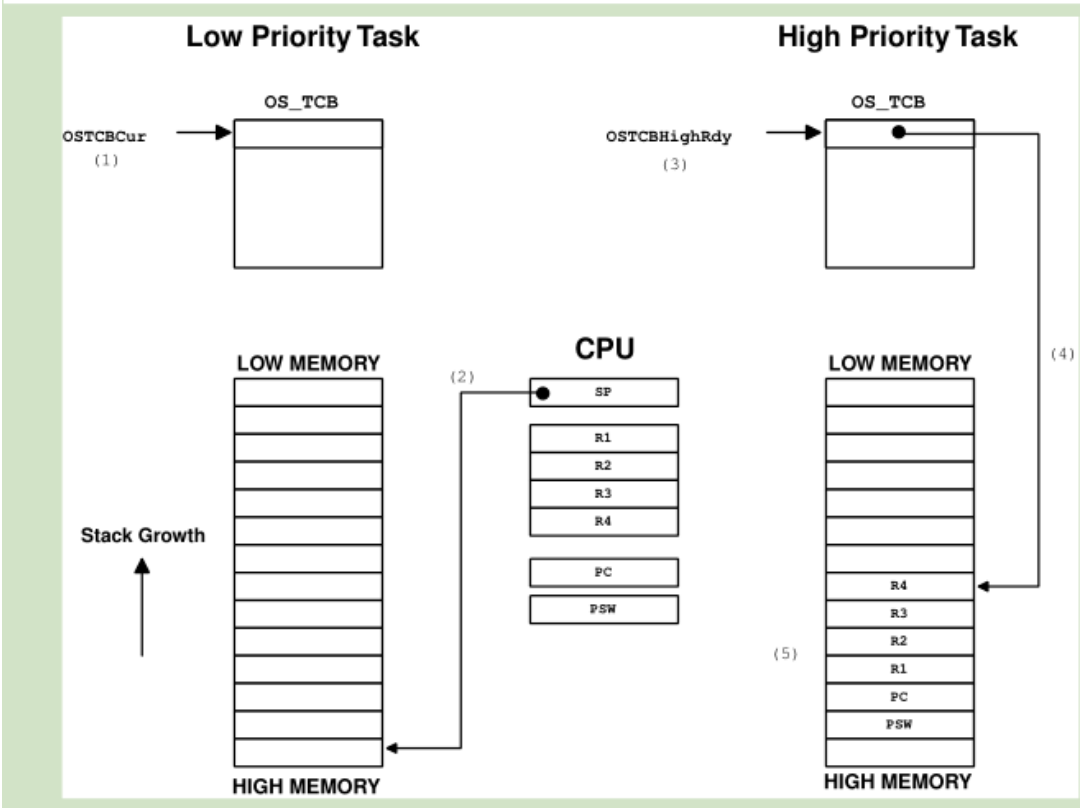
UCOSII 使用中断方式，对 CPU 相关寄存器进行堆栈缓存与恢复操作。

以下为任务级的任务切换时的寄存器变化情况的范例，假设一个 CPU 具有 7 个寄存器，分别为状态寄存器 PSW，堆栈指针 SP，程序计数器 PC，以及 4 个通用寄存器 R1、R2、R3、R4。

发送任务切换之前，低优先级的任务具有 CPU 的使用权，全局变量 OSTCBCur 指向低优先级任务的任务控制块 TCB，CPU 的堆栈指针 SP 指向低优先级任务的堆栈，PC、R1~R4 为低优先级任务运行过程中的运行数值。

而高优先级任务将要开始剥夺低优先级任务的 CPU 使用权，在剥夺之前，通过任务调度，查找到当前就绪态任务中优先级最高的任务，全局变量 OSTCBHighRdy 指向高优先级任务的任务控制块 TCB，而 TCB 中的第一项 OSTCBStkPtr 指向高优先级任务的堆栈，堆栈中缓存上次高优先级任务被剥夺 CPU 使用权时的运行环境，即保存 R1~R4，PC，PSW。如下图所示。

Figure 3.6 µC/OS-II structures when OS_TASK_SW() is called.

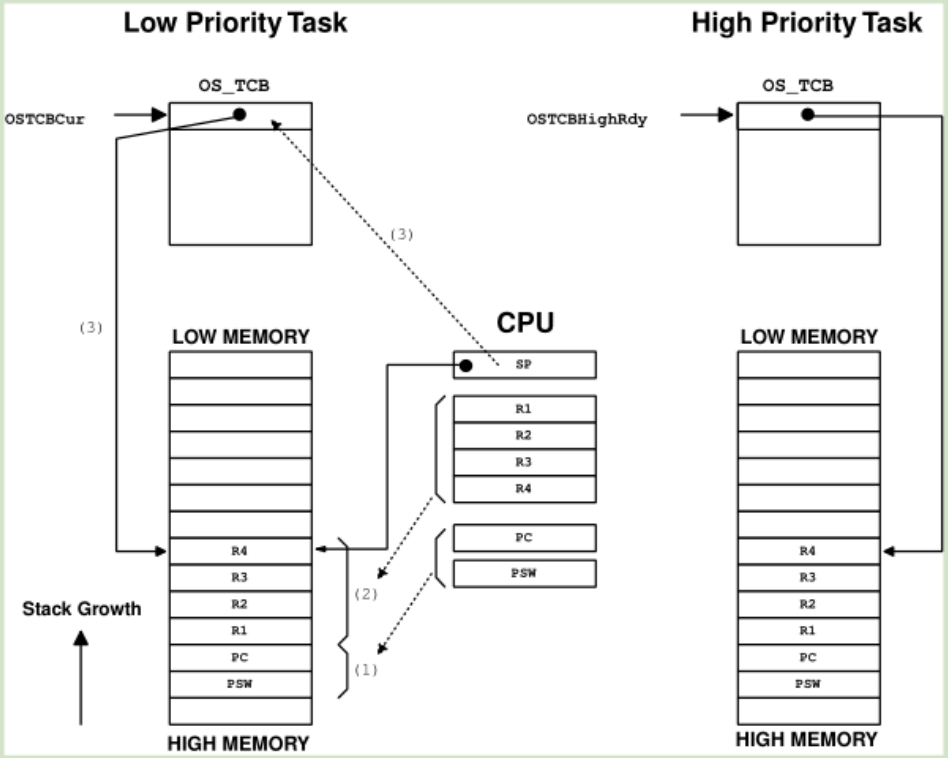


就绪任务切换的第一步是保存当前任务的运行环境，即保存低优先级任务的各个 CPU 寄存器值至任务堆栈中。

由于一般的 CPU 没有直接的将 PC 压入堆栈的指令，假设 CPU 也没有直接将 PSW 压入堆栈的指令，则开始时，调用含有软中断指令的宏 OS_TASK_SW()，通过中断机制，将 PC 与 PSW 压入 SP 指向的堆栈中，即低优先级任务的任务堆栈中，随后可在中断服务程序中将通用寄存器依次压入任务堆栈中。

然后将堆栈指针 SP 保存到低优先级任务的任务控制块 TCB 的第一项 OSTCBStkPtr 中，此时 SP 与 TCB. OSTCBStkPtr 均指向低优先级的任务堆栈栈顶处。如下图所示。

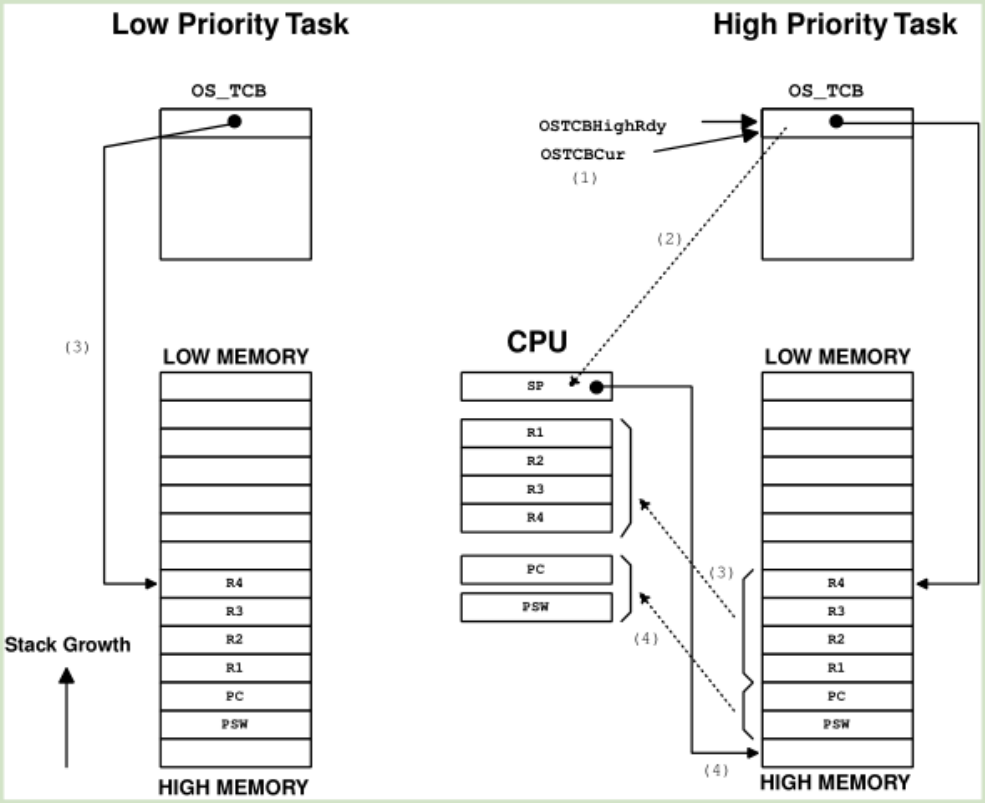
Figure 3.7 Saving the current task's context.



即将要运行的任务的控制块通过全局变量 OSTCBHighRdy 可获得，将 OSTCBHighRdy 指向的任务控制块 TCB 的 OSTCBStkPtr 装入 CPU 的 SP 寄存器中，然后按照一定的次序将新任务的任务堆栈中缓存的运行环境参数恢复到 CPU 的各个寄存器中。

此时退出软指令中断服务函数，在执行中断返回指令时，CPU 内在处理机制自动将 PC 与 PSW 恢复到 CPU 寄存器中，CPU 则根据新 PC 寄存器的值，开始运行高优先级任务代码。

Figure 3.8 Resuming the current task.



任务切换的伪代码如下：

Listing 3.11 Context-switch pseudocode.

```
void  OSCtxSw (void)
{
    PUSH R1, R2, R3 and R4 onto the current stack;              See F3.6(2)
    OSTCBCur->OSTCBStkPtr = SP;                                 See F3.6(3)
    OSTCBCur              = OSTCBHighRdy;                       See F3.7(1)
    SP                    = OSTCBHighRdy->OSTCBStkPtr;          See F3.7(2)
    POP R4, R3, R2 and R1 from the new stack;                   See F3.7(3)
    Execute a return from interrupt instruction;                See F3.7(4)
}
```

3.07 给调度器上锁与开锁

在 UCOSII 中可通过函数 OSSchedlock()给调度器上锁，即禁止任务调度，通过函数 OSSchedUnlock()给调度器开锁，即允许任务调度。函数 OSSchedlock()与 OSSchedUnlock()需要成对使用。

当调用了函数 OSSchedlock()以后，不得调用可能会使得当前任务挂起的系统功能函数，

例如 OSTimeDly()、OSFlagPend()、OSMboxPend()、OSQPend()等函数。

Listing 3.12 Locking the scheduler.

```
void  OSSchedLock (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


    if (OSRunning == TRUE) {                                          (1)
        OS_ENTER_CRITICAL();
        if (OSLockNesting < 255) {                                    (2)
            OSLockNesting++;
        }
        OS_EXIT_CRITICAL();
    }
}
```

任务上锁的嵌套深度为 255 层。

Listing 3.13 Unlocking the scheduler.

```
void  OSSchedUnlock (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


    if (OSRunning == TRUE) {                                          (1)
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {                                      (2)
            OSLockNesting--;                                          (3)
            if ((OSLockNesting == 0) && (OSIntNesting == 0)) {        (4)
                OS_EXIT_CRITICAL();
                OS_Sched();                                           (5)
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}
```

3.08 空闲任务

空闲任务是优先级最低的任务，即 OS_LOWEST_PRIO，在空闲任务中，可通过 OSTaskIdleHook()函数写入用户代码，由于空闲任务永远处于就绪态，则禁止在 OSTaskIdleHook()函数中调用任何使得任务被挂起的系统函数。

空闲任务代码如下。

Listing 3.14 The µC/OS-II idle task.

```
void  OS_TaskIdle (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


    pdata = pdata;
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtr++;                                    (1)
        OS_EXIT_CRITICAL();
        OSTaskIdleHook();                               (2)
    }
}
```

3.09 统计任务

宏定义 OS_TASK_STAT_EN 为 1 时，内核将建立统计任务 OSTaskStat()，该任务计算当前 CPU 的利用率。

在使用统计任务时，用户需在启动 OSStart()之前，建立一个任务，在该任务中调用统计任务初始化函数 OSStatInit()，然后在建立程序中的其他任务。

Listing 3.15 Initializing the statistic task.

```
void main (void)
{
    OSInit();                  /* Initialize uC/OS-II                      (1)*/
    /* Install uC/OS-II's context switch vector */
    /* Create your startup task (for sake of discussion, TaskStart())      (2)*/
    OSStart();                 /* Start multitasking                       (3)*/
}


void TaskStart (void *pdata)
{
    /* Install and initialize µC/OS-II's ticker                            (4)*/
    OSStatInit();              /* Initialize statistics task               (5)*/
    /* Create your application task(s) */
    for (;;) {
        /* Code for TaskStart() goes here! */
    }
}
```

如上的示例代码，在 OSStart()时，UCOSII 中只有三个任务，即空闲任务，统计任务，与用户创建的 TaskStart 任务，TaskStart 任务总是优先级最高的任务，空闲任务是优先级最低的任务，统计任务优先级次低。

在 TaskStart 任务中，首先初始化和启动时钟节拍，然后调用统计任务初始化函数 OSStatInit()，然后再创建用户程序的各个任务。

3.10 中断服务程序

在 UCOSII 中的中断服务子程序的伪代码如下所示。

Listing 3.18 ISRs under µC/OS-II.

```
YourISR:
    Save all CPU registers;                                          (1)
    Call OSIntEnter() or, increment OSIntNesting directly;           (2)
    if (OSIntNesting == 1) {                                         (3)
        OSTCBCur->OSTCBStkPtr = SP;                                  (4)
    }
    Clear interrupting device;                                       (5)
    Re-enable interrupts (optional)                                  (6)
    Execute user code to service ISR;                                (7)
    Call OSIntExit();                                                (8)
    Restore all CPU registers;                                       (9)
    Execute a return from interrupt instruction;                    (10)
```

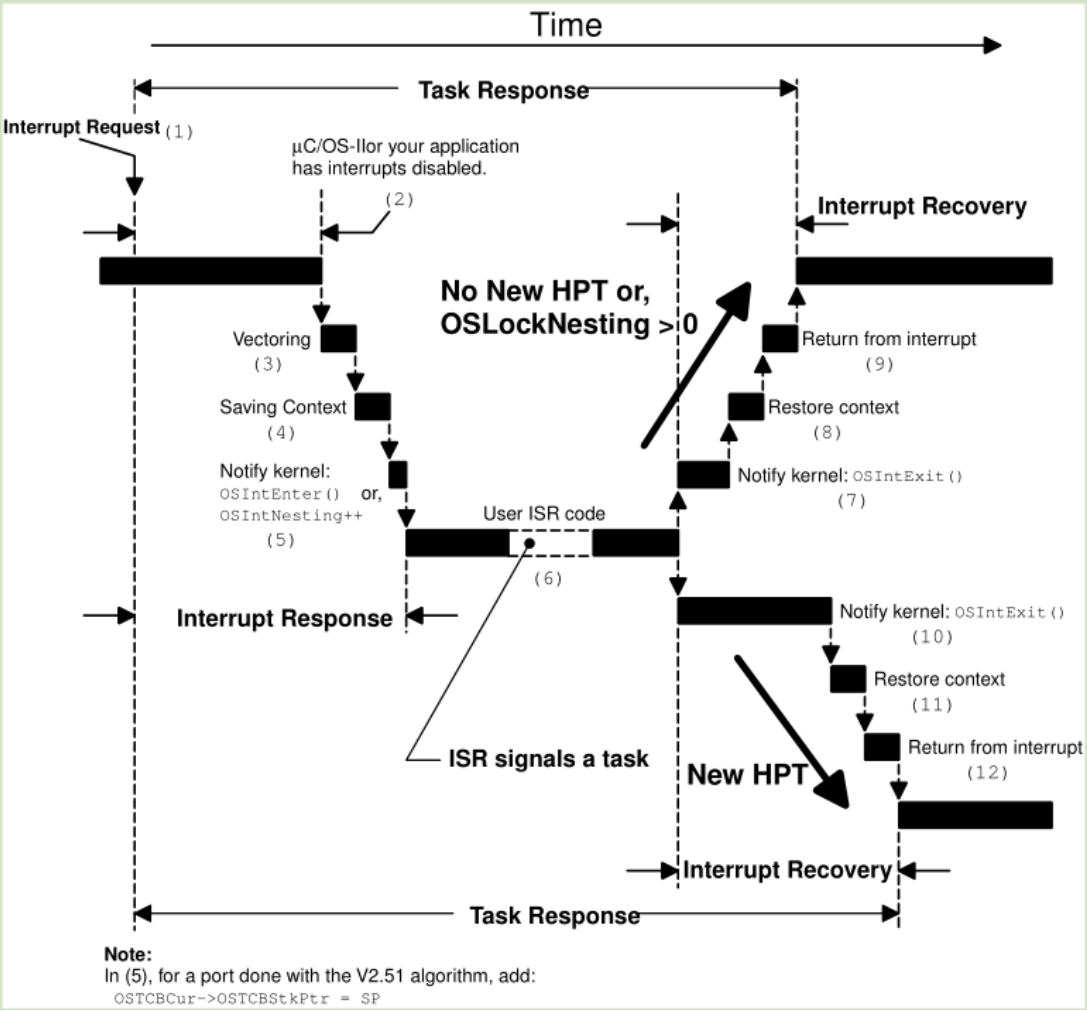
在中断服务程序开始时，首先需要将所有 CPU 的寄存器压入当前被中断任务的任务堆栈中，然后调用系统函数 OSIntEnter()，以告知内核正在做中断服务，

然后查看是否只有一层中断，若是则将当前堆栈指针保存到当前任务的任务控制块中，因为当只有一层中断时，若该中断服务程序退出，则可能会有更高优先级的任务被中断服务程序唤醒，则在中断结束时会进行任务调度，保存当前任务的 SP 是为了后续任务调度作准备。然后清中断源，重新打开中断，以允许中断嵌套。

然后开始允许用户的中断服务代码，然后调用系统函数 OSIntExit()，即将中断嵌套层数减去 1，当中断嵌套层数减为 0 时，则说明所有嵌套的中断均已经完成，此时内核开始判断是否有更高优先级任务就绪，进行任务调度。

随后恢复 CPU 寄存器值，执行中断返回指令。

Figure 3.10 Servicing an interrupt.



以上表描述了中断服务的过程，从 1 到 2 是中断的响应过程，响应时间根据 CPU 的不同而定，或者此时中断被程序关闭，直到中断打开，此时到第 3 步骤，跳转到中断向量，根据向量程序开始运行中断服务程序。

第 4 步骤保存 CPU 的各个寄存器值，即保存当前被中断任务的运行环境。

第 5 步骤是中断服务程序通知内核当前已经进入中断服务程序，通知的办法是调用系统函数 OSIntEnter()，即将中断嵌套层数变量 OSIntNesting 加 1。同时若中断层数为 1，则将堆栈指针 SP 保存到当前任务的 TCB 中。

第 6 步骤开始执行用户的中断服务代码，一般中断服务代码中所做的任务尽可能的少，大部分的任务将由外部任务完成，中断服务程序只是通过信号量，邮箱等手段去通知相关的任务，即调用 OSFlagPost()，OSMboxPost()，OSQPost()等系统函数，以通知任务。

第 7 步骤调用系统函数 OSIntExit()通知内核退出中断服务程序，如果没有更高优先级的

任务被中断服务程序激活而进入就绪态，则退出中断程序后将返回被中断的任务，此时只需用恢复 CPU 寄存器数值，执行中断返回指令，如步骤 8、9 所示。

如果有高优先级的任务被中断服务程序激活而进入就绪态，则在系统函数 OSIntExit()中将进行任务切换，优先级最高的就绪态任务的堆栈中的 CPU 寄存器值将恢复，并执行中断返回指令，如步骤 10、11、12。

系统函数 OSIntEnter()的代码如下：

Listing 3.19 Notify µC/OS-II about beginning an ISR.

```
void  OSIntEnter (void)
{
    if (OSRunning == TRUE) {
        if (OSIntNesting < 255) {
            OSIntNesting++;
        }
    }
}
```

系统函数 OSIntExit()的代码如下：

Listing 3.20 Notify µC/OS-II about leaving an ISR.

```
void  OSIntExit (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
```

Listing 3.20 Notify µC/OS-II about leaving an ISR. (Continued)

```
    OS_ENTER_CRITICAL();
    if (OSRunning == TRUE) {
        if (OSIntNesting > 0) {                                          (1)
            OSIntNesting--;
        }
        if ((OSIntNesting == 0) && (OSLockNesting == 0)) {
            OSIntExitY    = OSUnMapTbl[OSRdyGrp];                        (2)
            OSPrioHighRdy = (INT8U)((OSIntExitY << 3)
                          + OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
            if (OSPrioHighRdy != OSPrioCur) {
                OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
                OSCtxSwCtr++;
                OSIntCtxSw();                                            (3)
            }
        }
    }
    OS_EXIT_CRITICAL();
}
```

需用注意的是在中断级，当进行任务切换时，调用的是 OSIntCtxSw()，而在任务级进行任务切换时，调用的是 OS_TASK_SW()，这是因为当发生了中断，进入中断服务程序时，已经将 CPU 的寄存器存入被中断任务的堆栈中，因此无需再次执行保存运行环境的操作。

3.11 时钟节拍

UCOSII 内核需用心跳信号，以实现系统运行，一般节拍率为 10~100 次/秒。需用注意的是必须在多任务系统启动后，即调用 OSStart()后，再开启时钟节拍器。即在调用 OSStart()后的第一项任务就是初始化与开启时钟节拍器。

另外如果使用了统计任务，在统计任务初始化函数 OSStatInit()中需用延时等待，因此若先调用 OSStatInit()，然后再初始化时钟节拍器，则系统将死在空闲任务中。

UCOSII 中的时钟节拍服务是在时钟中断服务程序中调用系统函数 OSTimeTick()实现的，系统函数 OSTimeTick()跟踪所有任务的定时器与超时时限。

时钟节拍中断服务函数的伪代码如下：

Listing 3.23 Pseudocode for tick ISR.

```
void OSTickISR(void)
{
    Save processor registers;
    Call OSIntEnter() or increment OSIntNesting;
    if (OSIntNesting == 1) {
        OSTCBCur->OSTCBStkPtr = SP;
    }
    Call OSTimeTick();
    Clear interrupting device;
    Re-enable interrupts (optional);
    Call OSIntExit();
    Restore processor registers;
    Execute a return from interrupt instruction;
}
```

在中断函数中调用系统函数 OSTimeTick()，而 OSTimeTick()函数的代码如下：

Listing 3.24 Service a tick, OSTimeTick().

```
void  OSTimeTick (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_TCB    *ptcb;


    OSTimeTickHook();                                                          (1)
#if OS_TIME_GET_SET_EN > 0
    OS_ENTER_CRITICAL();
    OSTime++;                                                                  (2)
    OS_EXIT_CRITICAL();
#endif
    if (OSRunning == TRUE) {
        ptcb = OSTCBList;                                                      (3)
        while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {                              (4)
            OS_ENTER_CRITICAL();
            if (ptcb->OSTCBDly != 0) {
                if (--ptcb->OSTCBDly == 0) {
                    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == 0x00) {         (5)
                        OSRdyGrp               |= ptcb->OSTCBBitY;             (6)
                        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                    } else {
                        ptcb->OSTCBDly = 1;
```

Listing 3.24 Service a tick, OSTimeTick(). (Continued)

```
                    }
                }
            }
            ptcb = ptcb->OSTCBNext;
            OS_EXIT_CRITICAL();
        }
    }
}
```

系统函数 OSTimeTick()沿着 OS_TCB 链表，从优先级最高任务一直到空闲任务，计算各个任务的时间延时项，当时间延时项减为 0，且任务处于等待态，则设置该任务进入就绪态。

3.12 UCOSII 的初始化

系统初始化函数 OSInit()对 UCOSII 中的所有变量和数据结构初始化，其任务主要包括：

1) 建立空闲任务，并设该任务一直处于就绪态；

2) 如果允许统计任务，则建立统计任务；

3) 设置就绪表；

4) 设置任务控制块 TCB 及其双向链表，表中只有空闲任务与统计任务；

5) 初始化 5 个空的数据结构缓存区，每个缓存区均为单向链表，其缓存区包括 OS_MAX_TASKS 个任务控制块 TCB，OS_MAX_EVENTS 个事件控制块，OS_MAX_QS 个消息队列，OS_MAX_FLAGS 个标志控制块，以及 OS_MAX_MEM_PART 个存储控制块。

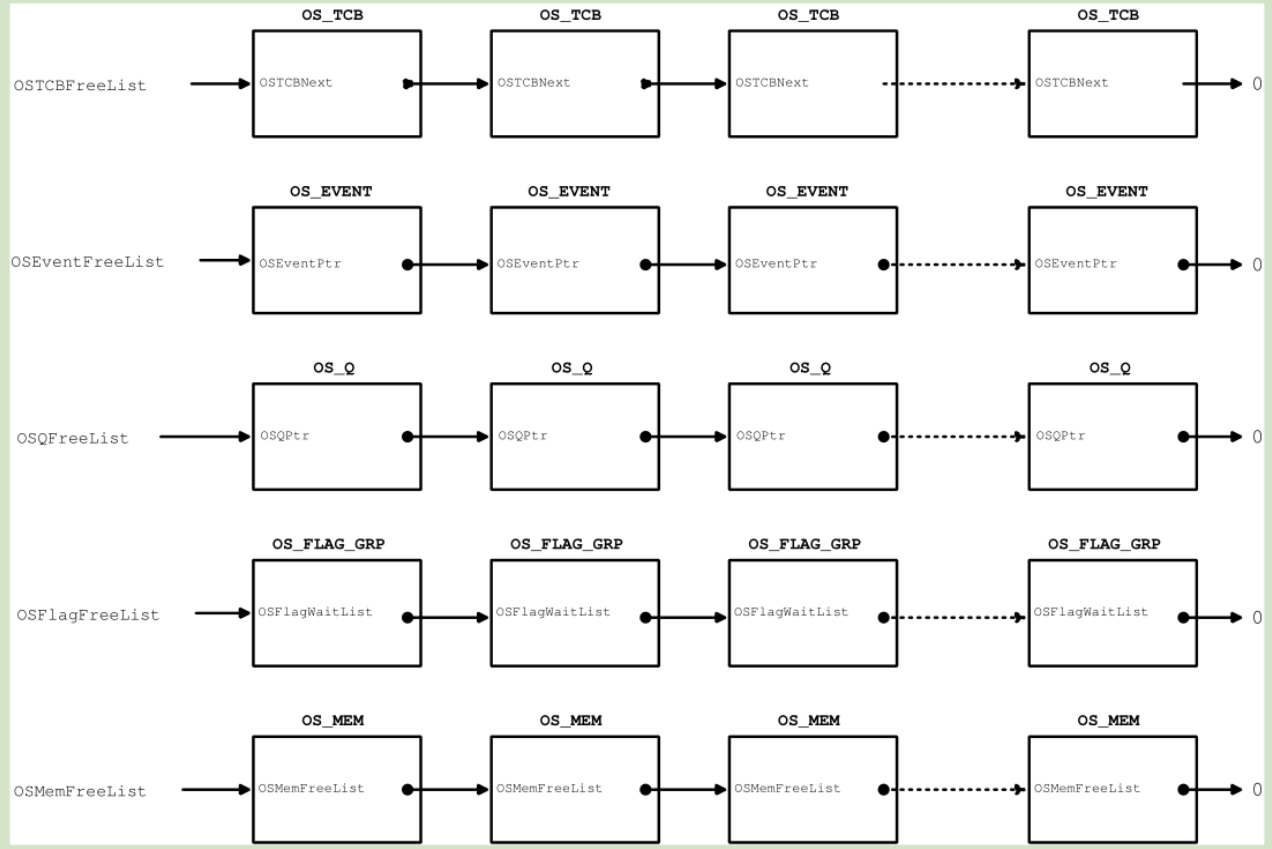
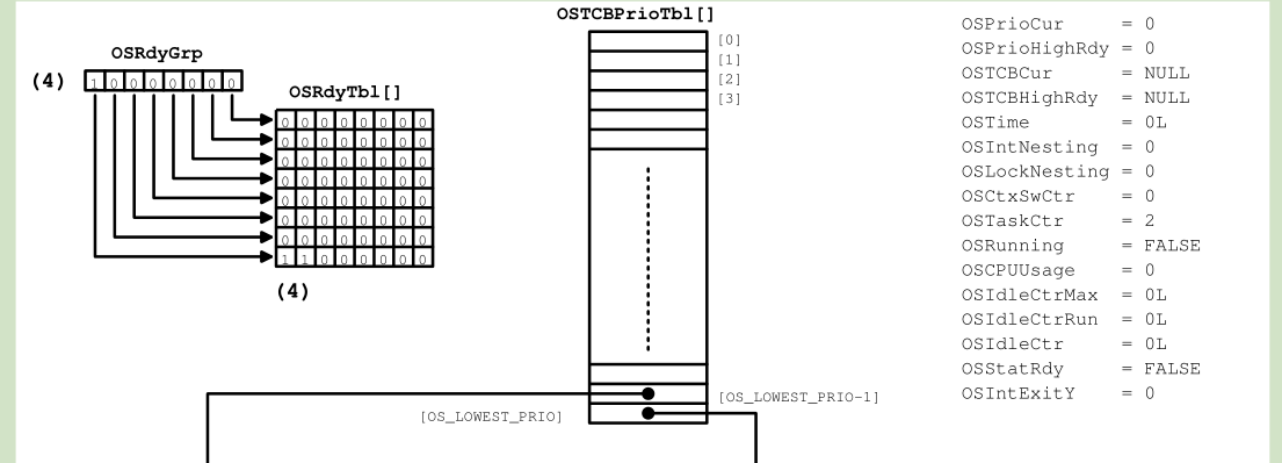
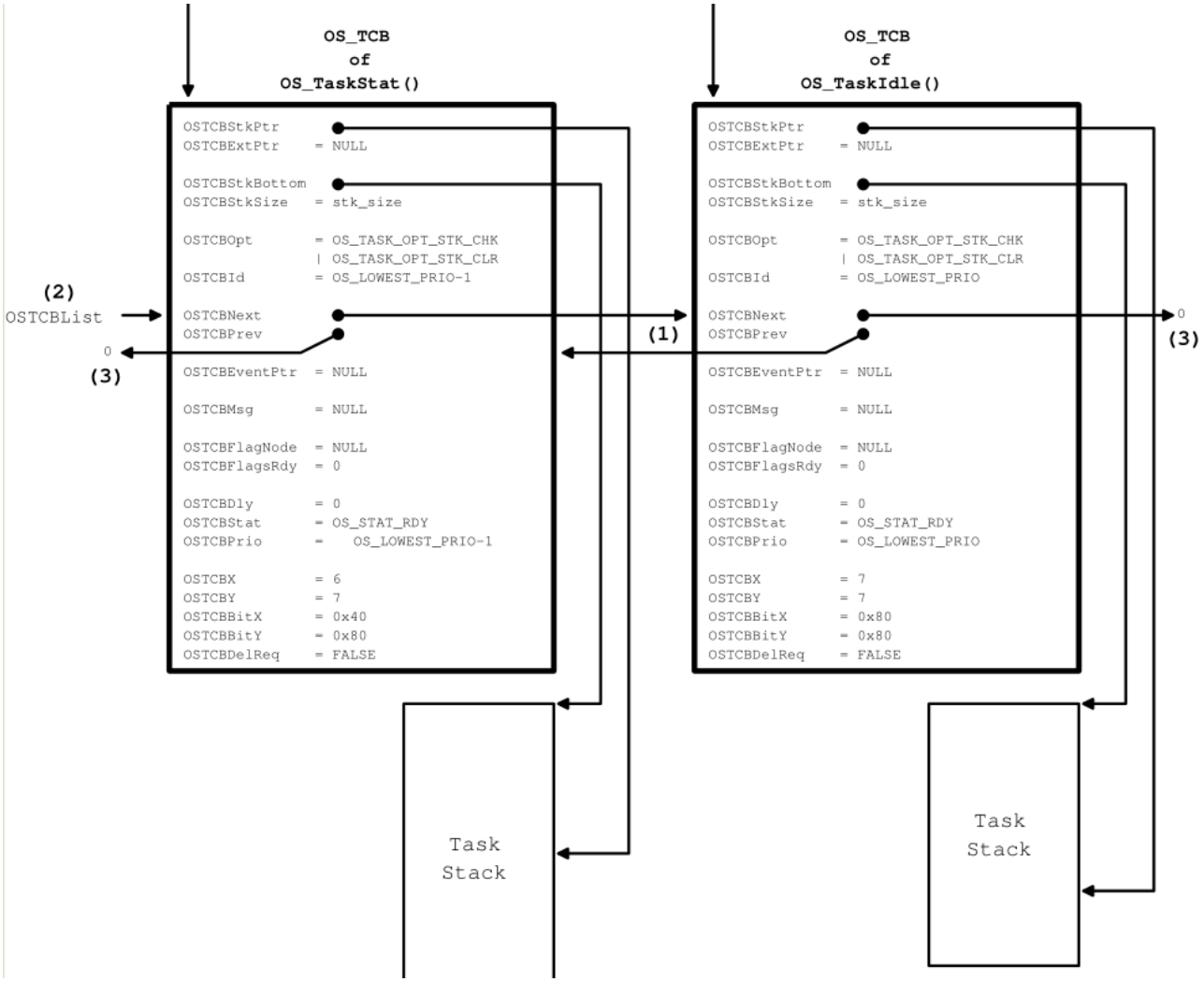


Figure 3.11 Variables and data structures after calling OSInit().





3.13 UCOSII 的启动

内核的启动是通过调用系统函数 OSStart()实现的，但在调用之前，需要至少创建一个应用任务。

Listing 3.27 Initializing and starting µC/OS-II.

```
void main (void)
{
    OSInit();           /* Initialize uC/OS-II                              */
    .
    .
    Create at least 1 task using either OSTaskCreate() or OSTaskCreateExt();
    .
    .
    OSStart();          /* Start multitasking!  OSStart() will not return */
}
```

系统函数 OSStart()的代码如下：

Listing 3.28 Starting multitasking.

```
void OSStart (void)
{
    INT8U y;
    INT8U x;

    if (OSRunning == FALSE) {
        y             = OSUnMapTbl[OSRdyGrp];
        x             = OSUnMapTbl[OSRdyTbl[y]];
        OSPrioHighRdy = (INT8U)((y << 3) + x);
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];                    (1)
        OSTCBCur      = OSTCBHighRdy;
        OSStartHighRdy();                                               (2)
    }
}
```

当调用系统函数 OSStart()时，函数将从任务就绪表中找出已经建立的优先级最高的任务的任务控制块，然后调用 OSStartHighRdy()启动该优先级最高的任务，函数 OSStartHighRdy()是将任务堆栈中保存的 CPU 寄存器的值恢复，执行一条中断返回指令，开始执行该任务的代码，注意 OSStartHighRdy()将永远不会返回到 OSStart()中。

函数 OSStartHighRdy()与具体的 CPU 有关，一般函数处于文件 os_cpu_a.asm 中，在进行 UCOSII 移植时需要更改。对于 STM32 的移植，其 OSStartHighRdy()的代码如下：

```
81  ;/*********************************************************************************************
82  ;* 函数名称: OSStartHighRdy
83  ;*
84  ;* 功能描述: 使用调度器运行第一个任务
85  ;*
86  ;* 参    数: None
87  ;*
88  ;* 返 回 值: None
89  ;*********************************************************************************************/
90  OSStartHighRdy
91          LDR     R4, =NVIC_SYSPRI2       ; set the PendSV exception priority
92          LDR     R5, =NVIC_PENDSV_PRI
93          STR     R5, [R4]
94
95          MOV     R4, #0                  ; set the PSP to 0 for initial context switch call
96          MSR     PSP, R4
97
98          LDR     R4, =OSRunning          ; OSRunning = TRUE
99          MOV     R5, #1
100         STRB    R5, [R4]
101
102                                         ;切换到最高优先级的任务
103         LDR     R4, =NVIC_INT_CTRL      ;rigger the PendSV exception (causes context switch)
104         LDR     R5, =NVIC_PENDSVSET
105         STR     R5, [R4]
106
107         CPSIE   I                       ;enable interrupts at processor level
108 OSStartHang
109         B       OSStartHang             ;should never get here
110
```

程序实现开始运行第一个优先级最高的任务。程序首先设定 PendSV 的优先级，设定 PSP 堆栈指针，设定系统运行标志 OSRunning，然后通过软指令触发 PendSV 中断，在 PendSV 中断服务程序中实现任务现场保护与切换，即完成任务调度，开始运行第一个程序。

4 任务管理

在 UCOSII 中任务结构要不是一个无限循环，要不是执行完成后任务进入休眠态。任务的返回类型必须是 void。以下是两种任务结构的范例。

```
void YourTask (void *pdata)
{
    for (;;) {
        /* USER CODE */
        Call one of uC/OS-II's services:
            OSFlagPend();
            OSMboxPend();
            OSMutexPend();
            OSQPend();
            OSSemPend();
            OSTaskSuspend(OS_PRIO_SELF);
            OSTimeDly();
            OSTimeDlyHMSM();
        /* USER CODE */
    }
}
```

```
void YourTask (void *pdata)
{
    /* USER CODE */
    OSTaskDel(OS_PRIO_SELF);
}
```

4.00 建立任务 OSTaskCreat()

在应用程序中建立任务，调用系统函数 OSTaskCreat()或者 OSTaskCreatExt()，其中函数 OSTaskCreatExt()是函数 OSTaskCreat()的扩展，提供附加的功能。

任务不能在中断服务程序中建立。

系统函数 OSTaskCreat()的输入参数为 4 个，分别为 task 是指向任务代码的指针，pdata 是任务开始执行时，传递给任务的参数指针，ptos 是分配给任务的堆栈的栈顶指针，prio 是分配给任务的优先级。

系统函数 OSTaskCreat()的代码如下：

Listing 4.1 OSTaskCreate().

```
INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    void      *psp;
    INT8U      err;

#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO) {                                          (1)
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {                              (2)
```

Listing 4.1 OSTaskCreate(). (Continued)

```
        OSTCBPrioTbl[prio] = (OS_TCB *)1;                                 (3)
        OS_EXIT_CRITICAL();                                               (4)
        psp = (void *)OSTaskStkInit(task, pdata, ptos, 0);                (5)
        err = OS_TCBInit(prio, psp, (void *)0, 0, 0, (void *)0, 0);       (6)
        if (err == OS_NO_ERR) {                                           (7)
            OS_ENTER_CRITICAL();
            OSTaskCtr++;                                                  (8)
            OS_EXIT_CRITICAL();
            if (OSRunning == TRUE) {                                      (9)
                OS_Sched();                                              (10)
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;                            (11)
            OS_EXIT_CRITICAL();
        }
        return (err);
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
```

如果宏 OS_ARG_CHK_EN 为 1，则函数 OSTaskCreat()会检测分配给任务的优先级是否有效。

如果分配的优先级是空闲的，则内核在 OSTCBPrioTbl[]中放置一个非空指针，以保留该优先级。

随后函数 OSTaskCreat()调用 OSTaskStkInit()初始化任务堆栈，该函数是与具体的 CPU 有

关，函数代码在文件 os_cpu_c.c 中，在进行 UCOSII 移植时，需要改写该函数。对于 STM32 处理器，该函数的源码如下，函数返回任务堆栈的栈顶指针 psp。

```
192  OS_STK *OSTaskStkInit (void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT16U opt)
193  {
194      OS_STK *stk;
195
196
197      (void)opt;                                   /* 'opt' is not used, prevent warning                  */
198      stk       = ptos;                            /* Load stack pointer                                  */
199
200                                                   /* Registers stacked as if auto-saved on exception     */
201      *(stk)    = (INT32U)0x01000000L;             /* xPSR                                                */
202      *(--stk)  = (INT32U)task;                    /* Entry Point                                         */
203      *(--stk)  = (INT32U)0xFFFFFFFEL;             /* R14 (LR) (init value will cause fault if ever used)*/
204      *(--stk)  = (INT32U)0x12121212L;             /* R12                                                 */
205      *(--stk)  = (INT32U)0x03030303L;             /* R3                                                  */
206      *(--stk)  = (INT32U)0x02020202L;             /* R2                                                  */
207      *(--stk)  = (INT32U)0x01010101L;             /* R1                                                  */
208      *(--stk)  = (INT32U)p_arg;                   /* R0 : argument                                       */
209
210                                                   /* Remaining registers saved on process stack          */
211      *(--stk)  = (INT32U)0x11111111L;             /* R11                                                 */
212      *(--stk)  = (INT32U)0x10101010L;             /* R10                                                 */
213      *(--stk)  = (INT32U)0x09090909L;             /* R9                                                  */
214      *(--stk)  = (INT32U)0x08080808L;             /* R8                                                  */
215      *(--stk)  = (INT32U)0x07070707L;             /* R7                                                  */
216      *(--stk)  = (INT32U)0x06060606L;             /* R6                                                  */
217      *(--stk)  = (INT32U)0x05050505L;             /* R5                                                  */
218      *(--stk)  = (INT32U)0x04040404L;             /* R4                                                  */
219
220      return (stk);
221  }
```

不同的 CPU，其堆栈可能是地址递减的，也可能是地址递增的，因此在调用函数 OSTaskCreat()时，需要知道传递给函数的 ptos 是堆栈的最高地址或者是最低地址。

随后函数 OSTaskCreat()调用 OS_TCBInit()对任务控制块初始化，即从空闲的 OS_TCB 缓冲区中获得并初始化一个任务控制块。

变量 OSTaskCtr 用于跟踪已经建立的任务的数目。如果任务控制块初始化成功，则变量 OSTaskCtr 加 1，如果任务控制块初始化失败，则设置 OSTCBPrioTbl[]的入口为 0，以放弃该任务优先级。

如果任务是在某个任务的执行过程中被建立，则内核将会进行任务调度，以确定新建立的任务是否满足执行的条件，如果任务是在多任务调度之前建立，即在 OSStart()调用之前建立，则不会进行任务调度。

4.01 建立任务 OSTaskCreatExt()

函数 OSTaskCreatExt()是带有扩展功能的任务建立函数。函数的代码如下：

Listing 4.2 *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)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_STK    *psp;
    INT8U      err;


#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO) {                                          (1)
        return (OS_PRIO_INVALID);
    }
#endif
    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) != 0x0000) ||                    (5)
            ((opt & OS_TASK_OPT_STK_CLR) != 0x0000)) {
            #if OS_STK_GROWTH == 1
            (void)memset(pbos, 0, stk_size * sizeof(OS_STK));
            #else
            (void)memset(ptos, 0, stk_size * sizeof(OS_STK));
            #endif
        }
```

Listing 4.2 OSTaskCreateExt(). (Continued)

```
        psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, opt);          (6)
        err = OS_TCBInit(prio, psp, pbos, id, stk_size, pext, opt);     (7)
        if (err == OS_NO_ERR) {                                         (8)
            OS_ENTER_CRITICAL();                                        (9)
            OSTaskCtr++;
            OS_EXIT_CRITICAL();
            if (OSRunning == TRUE) {                                    (10)
                OS_Sched();                                             (11)
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;
            OS_EXIT_CRITICAL();
        }
        return (err);
    }
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
```

函数 OSTaskCreatExt()具有 9 个输入函数，介绍如下。

输入参数 pd：执行任务代码的指针；

输入参数 pdata：任务开始执行时，传递给任务的参数指针；

输入参数 ptos：任务堆栈的栈顶指针；

输入参数 prio：任务优先级；

输入参数 id：任务的 id，在 UCOSII 中暂未使用；

输入参数 pbos：执行任务堆栈的栈底的指针；

输入参数 stk_size：指定堆栈的容量，如果堆栈空间数据为 16 位，则堆栈容量为 2*stk_size 字节，如果堆栈空间数据为 32 位，则堆栈容量为 4*stk_size 字节；

输入参数 pext：指向用户附加的数据域的指针，用来扩展任务控制块；

输入参数 opt：控制选项，分别为 OS_TASK_OPT_STK_CHK，OS_TASK_OPT_STK_CLR，OS_TASK_OPT_SAVE_FP，具体见 3.03 节相关内容。

在函数 OSTaskCreatExt()的代码中，使用了 1 个 C 语言库函数 memset，以及一个操作符 sizeof。操作符 sizeof 返回一个对象或者类型所占的内存字节数。下面详细讲解 memset 函数。

函数 memset 的原型为：void *memset(void *s，int ch，size_t n)。函数的作用是将 s 所指向的某一块内存中的前 n 个字节的内容全部设置为 ch 指定的 ASCII 值。这个函数通常为内存做初始化工作，其返回值为指向 s 的指针。该函数对较大的结构体或者数组进行清零操作的

一种最快的方法。

其范例如下：

```
int array[5]={1,4,3,5,2};
for(int i=0;i<5;i++)
cout<<array[i]<<"";
cout<<endl;

memset(array,0,5*sizeof(int));
for(int k=0;k<5;k++)
cout<<array[k]<<"";
cout<<endl;
```

其输出结果为：

```
14352
00000
```

注意函数 memset 是对字节的操作，因此最好使用 sizeof 操作符。

memset 可以方便的清空一个结构类型的变量或数组。

比如结构体：

```
struct sample_struct
{
   char csName[16];
   int iSeq;
   int iType;
};
```

对于变量 struct sample_struct stTest;用 memset 函数清空的方法是：

```
memset(&stTest,0,sizeof(structsample_struct));
```

函数 OSTaskCreatExt()的其他代码与函数 OSTaskCreat ()类似，不再详细讲解。

4.02 任务堆栈

每个任务都有其堆栈空间，可以在编译时分配堆栈空间，即静态分配堆栈空间。如下所示。

Listing 4.3 Static stack.

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

or

Listing 4.4 Static stack.

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

任务堆栈也可以在运行时分配，即动态分配堆栈空间，可以使用 C 编译器提供的 malloc() 函数动态分配内存。

Listing 4.5 Using malloc() to allocate stack space for a task.

```
OS_STK  *pstk;


pstk = (OS_STK *)malloc(stack_size);
if (pstk != (OS_STK *)0) {      /* Make sure malloc() has enough space */
    Create the task;
}
```

处理器支持的堆栈可以是递减的，也可以是递增的，在建立任务时，需要将堆栈的栈顶地址传递给函数 OSTaskCreatExt()，因此必须知道 CPU 支持的堆栈是递减或者递增。宏 OS_STK_GROWTH 来表征堆栈的递增或者递减。

当宏 OS_STK_GROWTH 为 0 时，堆栈空间是递增的，即堆栈栈顶是低地址，因此需要将堆栈的最低地址传递给任务创建函数。

Listing 4.6 Stack grows from low to high memory.

```
OS_STK  TaskStk[TASK_STK_SIZE];

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

当宏 OS_STK_GROWTH 为 1 时，堆栈空间是递减的，即堆栈栈顶是高地址，因此需要将堆栈的最高地址传递给任务创建函数。

Listing 4.7 Stack grows from high to low memory.

```
OS_STK  TaskStk[TASK_STK_SIZE];

OSTaskCreate(task, pdata, &TaskStk[TASK_STK_SIZE-1], prio);
```

引入宏 OS_STK_GROWTH 是为了方便程序的移植，当程序运行在堆栈增长方向不同的处理上时，只需要更改宏 OS_STK_GROWTH 的值即可。

Listing 4.8 Supporting stacks that grow in either direction.

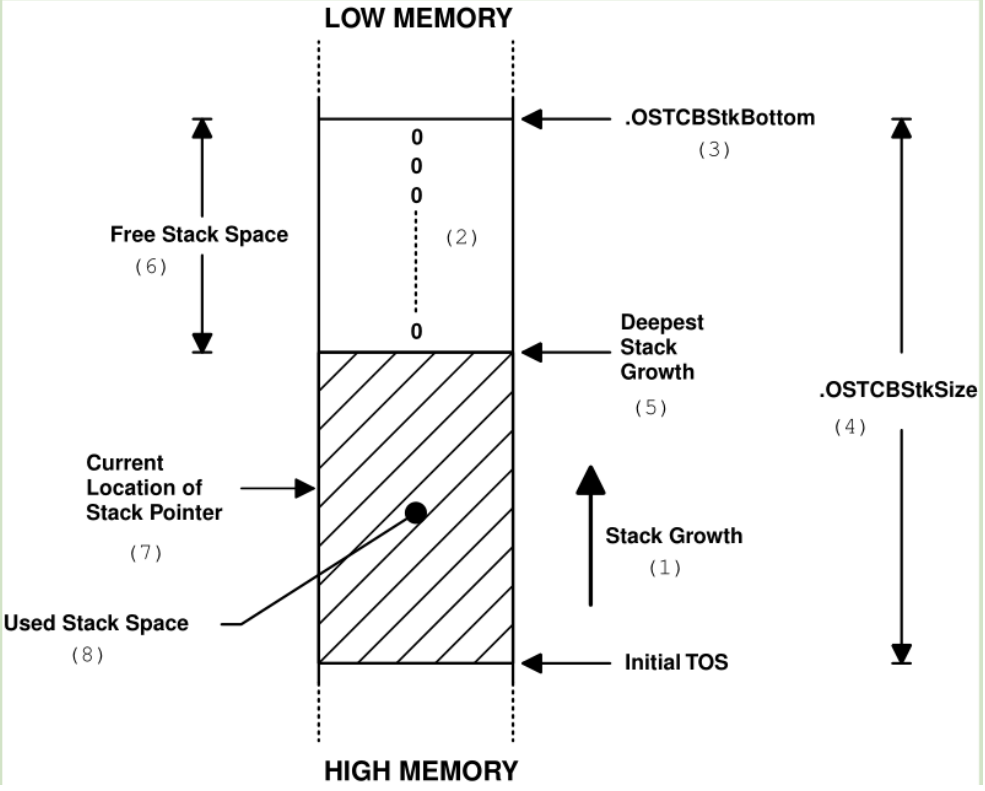
```
OS_STK  TaskStk[TASK_STK_SIZE];


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

4.03 堆栈检测

系统函数 OSTaskStkChk()用来做任务堆栈的检测，使用该系统功能，需要的提前是：

1) 将宏 OS_TASK_CREATE_EXT 设置为 1；
2) 使用函数 OSTaskCreatExt()建立任务；
3) 在系统函数 OSTaskCreatExt()中，将函数的输入参数 opt 设置为 OS_TASK_OPT_STK_CHK+OS_TASK_OPT_STK_CLR，
4) 将待检测任务的优先级传递给系统函数 OSTaskStkChk()。

Figure 4.2 Stack checking.



系统函数 OSTaskStkChk()从栈底开始计算空闲的堆栈空间的大小，具体实现方式是统计存储值为 0 的连续堆栈入口的数目，直到发现存储值不为 0 的堆栈入口，因此已经使用了的堆栈数目指从在函数 OSTaskCreatExt()中定义的堆栈容量中减去存储值为 0 的连续堆栈入口的数目以后的大小。

系统函数 OSTaskStkChk()将空闲任务堆栈的字节数和已用堆栈的字节数放在数据结构 OS_STK_DATA 中，其数据结构的定义在文件 ucos_ii.h 中，具体代码如下：

```
515 /*
516 *********************************************************************************************************
517 *                                          TASK STACK DATA
518 *********************************************************************************************************
519 */
520
521 #if OS_TASK_CREATE_EXT_EN > 0u
522 typedef struct os_stk_data {
523     INT32U  OSFree;                    /* Number of free bytes on the stack                          */
524     INT32U  OSUsed;                    /* Number of bytes used on the stack                          */
525 } OS_STK_DATA;
526 #endif
```

系统函数 OSTaskStkChk()的代码如下：

Listing 4.9　　Stack-checking function.

```
INT8U  OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_TCB    *ptcb;
    OS_STK    *pchk;
    INT32U     free;
    INT32U     size;


#if OS_ARG_CHK_EN > 0
    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {                    (1)
        return (OS_PRIO_INVALID);
    }
#endif
    pdata->OSFree = 0;
    pdata->OSUsed = 0;
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                                             (2)
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb == (OS_TCB *)0) {                                              (3)
```

Listing 4.9 Stack-checking function. (Continued)

```
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);
    }
    if ((ptcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) == 0) {                    (4)
        OS_EXIT_CRITICAL();
        return (OS_TASK_OPT_ERR);
    }
    free = 0;                                                             (5)
    size = ptcb->OSTCBStkSize;
    pchk = ptcb->OSTCBStkBottom;
    OS_EXIT_CRITICAL();
#if OS_STK_GROWTH == 1
    while (*pchk++ == (OS_STK)0) {
        free++;
    }
#else
    while (*pchk-- == (OS_STK)0) {
        free++;
    }
#endif
    pdata->OSFree = free * sizeof(OS_STK);                                (6)
    pdata->OSUsed = (size - free) * sizeof(OS_STK);
    return (OS_NO_ERR);
}
```

在代码中，宏 OS_PRIO_SELF 表示当前任务的优先级，当 OSTCBPrioTbl[]存在一个非空指针，则说明该优先级任务是存在的。

当堆栈检测的条件满足了，则从堆栈栈底开始统计堆栈的空闲空间，直到发现一个存储值为非 0 的堆栈入口。检测结果存储在结构体 OS_STK_DATA 中。检测结果是以字节为单位的。

4.04 删除任务

删除任务即是将任务返回并处于休眠态，通过调用系统函数 OSTaskDel()可实现对任务的删除，具体代码如下：

Listing 4.10 Task delete.

```
INT8U  OSTaskDel (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR     cpu_sr;
#endif

#if OS_EVENT_EN > 0
    OS_EVENT     *pevent;
#endif
#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
    OS_FLAG_NODE *pnode;
#endif
    OS_TCB       *ptcb;
    BOOLEAN       self;




    if (OSIntNesting > 0) {                                          (1)
        return (OS_TASK_DEL_ISR);
    }
#if OS_ARG_CHK_EN > 0
    if (prio == OS_IDLE_PRIO) {                                      (2)
        return (OS_TASK_DEL_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {            (3)
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                                      (4)
```

Listing 4.10 Task delete. (Continued)

```
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {                                          (5)
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {     (6)
            OSRdyGrp &= ~ptcb->OSTCBBitY;
        }
#if OS_EVENT_EN > 0
        pevent = ptcb->OSTCBEventPtr;                                   (7)
        if (pevent != (OS_EVENT *)0) {
            if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
            }
        }
#endif
#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
        pnode = ptcb->OSTCBFlagNode;                                    (8)
        if (pnode != (OS_FLAG_NODE *)0) {
            OS_FlagUnlink(pnode);
        }
#endif
        ptcb->OSTCBDly  = 0;                                            (9)
        ptcb->OSTCBStat = OS_STAT_RDY;                                 (10)
        if (OSLockNesting < 255) {                                     (11)
            OSLockNesting++;
        }
        OS_EXIT_CRITICAL();                                            (12)
        OS_Dummy();                                                    (13)
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {                                       (14)
            OSLockNesting--;
        }
        OSTaskDelHook(ptcb);                                           (15)
        OSTaskCtr--;                                                   (16)
        OSTCBPrioTbl[prio] = (OS_TCB *)0;                              (17)
        if (ptcb->OSTCBPrev == (OS_TCB *)0) {                          (18)
            ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
            OSTCBList                  = ptcb->OSTCBNext;
        } else {
```

Listing 4.10 Task delete. (Continued)

```
            ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;
            ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;
        }
        ptcb->OSTCBNext = OSTCBFreeList;                              (19)
        OSTCBFreeList   = ptcb;
        OS_EXIT_CRITICAL();
        OS_Sched();                                                   (20)
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_TASK_DEL_ERR);
}
```

在 UCOSII 中，不允许在中断服务程序中删除任务，因此序号 1 处是为了防止中断中删除任务。序号 2 处是不允许删除空闲任务。

OSTaskDel()需要确保要删除的 TCB 是存在的，然后从所有可能的数据结构中删除任务的控制块 TCB，序号 6 删除就绪表与就绪组中的任务，序号 7 删除处于互斥型信号量，邮箱，消息队列或者信号量的事件等待表中的任务，序号 8 删除事件标志的等待表中的任务。

序号 13 的函数 OS_Dummy()没有实质的操作，只是保证处理器在中断开着的情况下至少执行一条指令。

序号 17 从优先级表中将任务的 TCB 删除，即设置指针为 NULL。

序号 18 与 19 是从 OSTCBList 开头的 TCB 双向链表中把被删除任务的任务控制块去掉，并将被删除任务的控制块 TCB 退回到空闲 OS_TCB 链表中，以供其他任务使用。

随后调用任务调度，查看开中断后，是否有更高优先级的任务进入就绪状态。

4.05 请求删除任务

当某个任务正在占用一些共享资源，这时候如果其他任务视图删除该任务，则该任务占用的共享资源会因为没有被释放而丢失，使得其他视图使用共享资源的任务得不得运行。解决办法就是使得占用共享资源的任务在释放完资源后再删除自己。内核提供了系统函数 OSTaskDelReq()完成该功能。

在用户程序中，发出删除任务请求的任务代码范例如下：

Listing 4.11 Requester code requesting a task to delete itself.

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


    pdata = pdata;
    for (;;) {
        /* Application code */
        if ('TaskToBeDeleted()' needs to be deleted) {                          (1)
            while (OSTaskDelReq(TASK_TO_DEL_PRIO) != OS_TASK_NOT_EXIST) {       (2)
                OSTimeDly(1);                                                   (3)
            }
        }
        /* Application code */                                                  (4)
    }
}
```

需要删除自己的任务，可通过任务控制块中的参数.OSTCBDelReq 来确认自己是否需要被删除，通过调用函数 OSTaskDelReq(OS_PRIO_SELF)可以得到这个参数标志的值。需要删除自己的任务的代码范例如下：

Listing 4.12 Task requesting to delete itself.

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

Listing 4.12 Task requesting to delete itself. (Continued)

```
    pdata = pdata;
    for (;;) {
        /* Application code */
        if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ) {                    (1)
            Release any owned resources;                                        (2)
            De-allocate any dynamic memory;
            OSTaskDel(OS_PRIO_SELF);                                            (3)
        } else {
            /* Application code */
        }
    }
}
```

系统函数 OSTaskDelReq()的代码如下：

Listing 4.13 *OSTaskDelReq().*

```
INT8U  OSTaskDelReq (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    stat;
    INT8U      err;
    OS_TCB    *ptcb;


#if OS_ARG_CHK_EN > 0
    if (prio == OS_IDLE_PRIO) {                                          (1)
        return (OS_TASK_DEL_IDLE);
    }
```

Listing 4.13 *OSTaskDelReq(). (Continued)*

```
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {                (2)
        return (OS_PRIO_INVALID);
    }
#endif
    if (prio == OS_PRIO_SELF) {                                          (3)
        OS_ENTER_CRITICAL();
        stat = OSTCBCur->OSTCBDelReq;
        OS_EXIT_CRITICAL();
        return (stat);
    }
    OS_ENTER_CRITICAL();
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb != (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);
}
```

如果需要被删除的任务是空闲任务，则函数报错并返回，如果需要被删除的任务是任务本身，则将任务 TCB 中的参数.OSTCBDelReq 返回，如果需要被删除的任务是存在，且不是任务本身，则会设置任务 TCB 中的参数.OSTCBDelReq，返回 OS_NO_ERR，如果任务不存在，则返回 OS_TASK_NOT_EXIST。

4.06 改变任务的优先级

内核允许动态地改变任务的优先级，内核提供系统函数 OSTaskChangePrio()实现该功能，其函数的具体代码如下：

Listing 4.14 *OSTaskChangePrio().*

```
INT8U  OSTaskChangePrio (INT8U oldprio, INT8U newprio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif

#if OS_EVENT_EN > 0
    OS_EVENT    *pevent;
#endif

    OS_TCB      *ptcb;
    INT8U        x;
    INT8U        y;
    INT8U        bitx;
    INT8U        bity;


#if OS_ARG_CHK_EN > 0
    if ((oldprio >= OS_LOWEST_PRIO && oldprio != OS_PRIO_SELF)  ||                    (1)
         newprio >= OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }
#endif
    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;
```

Listing 4.14 *OSTaskChangePrio(). (Continued)*

```
        }
        ptcb = OSTCBPrioTbl[oldprio];
        if (ptcb != (OS_TCB *)0) {                                          (6)
            OSTCBPrioTbl[oldprio] = (OS_TCB *)0;                            (7)
            if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {       (8)
                if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) { (9)
                    OSRdyGrp &= ~ptcb->OSTCBBitY;
                }
                OSRdyGrp    |= bity;                                        (10)
                OSRdyTbl[y] |= bitx;
#if OS_EVENT_EN > 0
            } else {
                pevent = ptcb->OSTCBEventPtr;
                if (pevent != (OS_EVENT *)0) {                              (11)
                    if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                        pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
                    }
                    pevent->OSEventGrp    |= bity;                          (12)
                    pevent->OSEventTbl[y] |= bitx;
                }
#endif
            }
            OSTCBPrioTbl[newprio] = ptcb;                                   (13)
            ptcb->OSTCBPrio       = newprio;                                (14)
            ptcb->OSTCBY          = y;                                      (15)
            ptcb->OSTCBX          = x;
            ptcb->OSTCBBitY       = bity;
            ptcb->OSTCBBitX       = bitx;
            OS_EXIT_CRITICAL();
            OS_Sched();                                                     (16)
            return (OS_NO_ERR);
        } else {
            OSTCBPrioTbl[newprio] = (OS_TCB *)0;                            (17)
            OS_EXIT_CRITICAL();
            return (OS_PRIO_ERR);
        }
    }
}
```

序号 1 判断输入参数，即旧优先级与新优先级的参数合法性；

序号 2 判断新优先级是否为空闲，否则改变失败，内核不运行多个任务具有相同的优先级；

序号 4 当检测到新优先级为空闲，则在优先级表 OSTCBPrioTbl[]中设置非 0 指针，以保留这个优先级，此时即可打开中断。

然后判断旧优先级任务是否已经就绪，如果就绪，则在旧优先级下，从就绪表与就绪组中移除该任务，然后在新优先级下，将该任务插入到就绪表中，如序号 8、9、10 所示。

如果旧优先级任务未就绪，则其可能正在等待一个信号量，或者一个消息队列等，若的确在等待事件，则在旧优先级下，从事件等待列表中移除该任务，然后在新优先级下，将该任务插入到事件等待列表中，如序号 11、12 所示。

随后更新任务控制块 TCB 中的数值，即序号 13、14、15 所示。最后进行一次任务调度。

4.07 挂起任务

内核提供任务挂起与任务恢复操作，任务挂起调用系统函数 OSTaskSuspend()，挂起的任务只能通过调用系统函数 OSTaskResume()来恢复。任务可以挂起自己或者其他任务。

系统函数 OSTaskSuspend()的代码如下：

Listing 4.15 *OSTaskSuspend().*

```
INT8U  OSTaskSuspend (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    self;
    OS_TCB    *ptcb;


#if OS_ARG_CHK_EN > 0
    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);
    }
#endif
    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;
    }
    ptcb = OSTCBPrioTbl[prio];                                           (5)
    if (ptcb == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_TASK_SUSPEND_PRIO);
    }
```

Listing 4.15 *OSTaskSuspend( ). (Continued)*

```
    if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {          (6)
        OSRdyGrp &= ~ptcb->OSTCBBitY;
    }
    ptcb->OSTCBStat |= OS_STAT_SUSPEND;                                  (7)
    OS_EXIT_CRITICAL();
    if (self == TRUE) {
        OS_Sched();                                                      (8)
    }
    return (OS_NO_ERR);
}
```

当任务挂起的是自己时，需要调用任务调度函数，否则不需要，因此函数中定义了变量 self 来实现。将任务挂起，其核心操作即是将被挂起的任务从就绪表与就绪组中去掉。

4.08 恢复任务

挂起的任务只能通过调用系统函数 OSTaskResume()来恢复，系统函数 OSTaskResume() 的代码如下：

因为内核不能挂起空闲任务，因此首先必须确认程序不是在恢复空闲任务。

在任务的最后，进行任务调度，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。

Listing 4.16 *OSTaskResume().*

```
INT8U  OSTaskResume (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_TCB    *ptcb;


#if OS_ARG_CHK_EN > 0
    if (prio >= OS_LOWEST_PRIO) {                                          (1)
        return (OS_PRIO_INVALID);
    }
#endif
    OS_ENTER_CRITICAL();
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb == (OS_TCB *)0) {                                             (2)
        OS_EXIT_CRITICAL();
        return (OS_TASK_RESUME_PRIO);
    }
    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) != 0x00) {                     (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();
            OS_Sched();                                                    (7)
        } else {
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_TASK_NOT_SUSPENDED);
}
```

4.09 获得任务信息

内核提供系统函数 OSTaskQuery()来获取自身或者其他任务的信息，其实函数获得的是指定的任务控制块 OS_TCB 中的内容。获取任务信息的用户程序范例如下：

Listing 4.17 Obtaining information about a task.

```
void MyTask (void *pdata)
{
    OS_TCB  MyTaskData;

    pdata = pdata;
    for (;;) {
        /* User code                    */
        err = OSTaskQuery(10, &MyTaskData);
        /* Examine error code ..        */
        /* User code                    */
    }
}
```

系统函数 OSTaskQuery()的代码如下：

Listing 4.18 OSTaskQuery().

```
INT8U  OSTaskQuery (INT8U prio, OS_TCB *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_TCB    *ptcb;


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

系统函数 OSTaskQuery()代码中调用了 C 库函数 memcpy()，其函数原型为：

void *memcpy(void *dest，const void *src，size_t n)，函数的功能是从源 src 所指的内存地址的

起始位置开始拷贝 n 个字节到目标 dest 所指的内存地址的起始位置中。C 语言所需的头文件是 string.h。函数返回指向 dest 的指针。

5 时间管理

UCOSII 内核使用定时器中断，来实现系统定时与运行。定时中断即为时钟节拍。内核提供了系统函数用来处理时间，分别为 OSTimeDly()，OSTimeDlyHMSM()，OSTimeDlyResume()，OSTimeGet()与 OSTimeSet()。

5.00 任务延时函数

内核提供系统函数 OSTimeDly()将任务延时一段时间，时间的长度由系统时钟节拍数确定，调用该函数可以使得内核进行一次任务调度，去执行下一个就绪态的优先级最高任务。当任务调用系统函数 OSTimeDly()后，如果延时时间到，或者有其他任务调用 OSTimeDlyResume()取消了延时，则任务重新进入就绪态。

系统函数 OSTimeDly()的代码如下：

Listing 5.1　　OSTimeDly().

```
void OSTimeDly (INT16U ticks)
{
    if (ticks > 0) {                                                    (1)
        OS_ENTER_CRITICAL();
        if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {    (2)
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBDly = ticks;                                     (3)
        OS_EXIT_CRITICAL();
        OSSched();                                                      (4)
    }
}
```

函数的输入参数 ticks 是时钟节拍数，如果输入的时钟节拍数是 0，则函数不延时，立即返回任务，如果输入的时钟节拍数为非 0，则函数将任务从就绪表与就绪组中移除，同时时钟节拍数保存在任务控制块 TCB 中，每间隔一个时钟节拍，系统函数 OSTimeTick()将该时钟节拍数减去 1，随后函数进行一次任务调度。

另外在进行较长时间的延时时，可以使用宏 OS_TICKS_PER_SEC 来定义一些定义，将延时时间对应的时钟节拍数字化，然后系统函数 OSTimeDly()的输入参数直接为这样数字化的宏即可，比如如下定义：

```
#define  OS_TIME_100mS  (INT16U)((INT32U)OS_TICKS_PER_SEC * 100L / 1000L)
#define  OS_TIME_500mS  (INT16U)((INT32U)OS_TICKS_PER_SEC * 500L / 1000L)
#define  OS_TIME_2S     (INT16U)(OS_TICKS_PER_SEC * 2)
```

5.01 按时分秒延时函数

内核提供系统函数 OSTimeDlyHMSM()，可以按照时、分、秒、毫秒来定义时间，调用函数 OSTimeDlyHMSM()也会使得内核进行一次任务调度，如果延时时间到，或者有其他任务调用 OSTimeDlyResume()取消了延时，则任务重新进入就绪态。

系统函数 OSTimeDlyHMSM()的代码如下：

Listing 5.2 OSTimeDlyHMSM().

```
INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U milli)
{
    INT32U ticks;
    INT16U loops;


    if (hours > 0 || minutes > 0 || seconds > 0 || milli > 0) {              (1)
        if (minutes > 59) {
            return (OS_TIME_INVALID_MINUTES);
        }
        if (seconds > 59) {
            return (OS_TIME_INVALID_SECONDS);
        }
        if (milli > 999) {
            return (OS_TIME_INVALID_MILLI);
        }
```

Listing 5.2 OSTimeDlyHMSM(). (Continued)

```
        ticks = (INT32U)hours    * 3600L * OS_TICKS_PER_SEC                  (2)
              + (INT32U)minutes  *   60L * OS_TICKS_PER_SEC
              + (INT32U)seconds  *         OS_TICKS_PER_SEC
              + OS_TICKS_PER_SEC * ((INT32U)milli
              + 500L / OS_TICKS_PER_SEC) / 1000L;                           (3)
        loops = ticks / 65536L;                                             (4)
        ticks = ticks % 65536L;                                             (5)
        OSTimeDly(ticks);                                                   (6)
        while (loops > 0) {                                                 (7)
            OSTimeDly(32768);                                               (8)
            OSTimeDly(32768);
            loops--;
        }
        return (OS_NO_ERR);
    }
    return (OS_TIME_ZERO_DLY);                                              (9)
}
```

函数首先检验输入的参数是否有效。输入参数 hours 为时间小时，输入参数 minutes 为时间分钟，输入参数 seconds 为时间秒，输入参数 milli 为时间毫秒。如果输入参数均为 0，则

不延时，立即返回任务。

代码中的时间公式计算了与指定的毫秒数最接近的时钟节拍数 ticks，内核支持最长的时钟节拍为 65535，即 FFFFH，为了获得支持更长的时间延时，函数 OSTimeDlyHMSM()确定指定的延时对应的时钟节拍总数中含有多少个完整的 65536 个时钟节拍，以及余下的时钟节拍数为多少。由于内核支持最长的时钟节拍为 65535，而不是 65536，因此对于 65536 个时钟，分为 2 个 32768 个节拍，即代码中调用了 2 次 OSTimeDly (32768)。

其中需要注意，对于函数 OSTimeDlyHMSM()延时超过 65535 个时钟节拍的任务，不能通过调用函数 OSTimeDlyResume()使之恢复。

5.02 恢复延时的任务

可以通过指定要恢复的任务的优先级来调用系统函数 OSTimeDlyResume()，使得延时的任务不等待延时时间到而处于就绪态。

函数 OSTimeDlyResume()的代码如下：

Listing 5.3　　Resuming a delayed task.

```
INT8U  OSTimeDlyResume (INT8U prio)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_TCB    *ptcb;

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

函数首先确定输入参数任务优先级是否有效，然后确认任务是否存在，然后取消任务控制块 TCB 中的延时参数，若任务没有被挂起，则将任务放入就绪表与就绪组中，然后进行一次任务调度。

5.03 系统时间的获取与设置

在内核中设置了一个 32 位的计数器 OSTime，每当时钟中断到了，则该计数器加 1，在内核初始化时该计数器置 0，时钟节拍频率为 100Hz 时，每间隔 497 天寄存器溢出重新开始计数。

内核提供 2 个系统函数设置与获取这个计数器的值，系统函数 OSTimeGet()获取计数器的值，系统函数 OSTimeSet()改变该计数器的值。其代码如下：

Listing 5.4　　Obtaining and setting the system time.

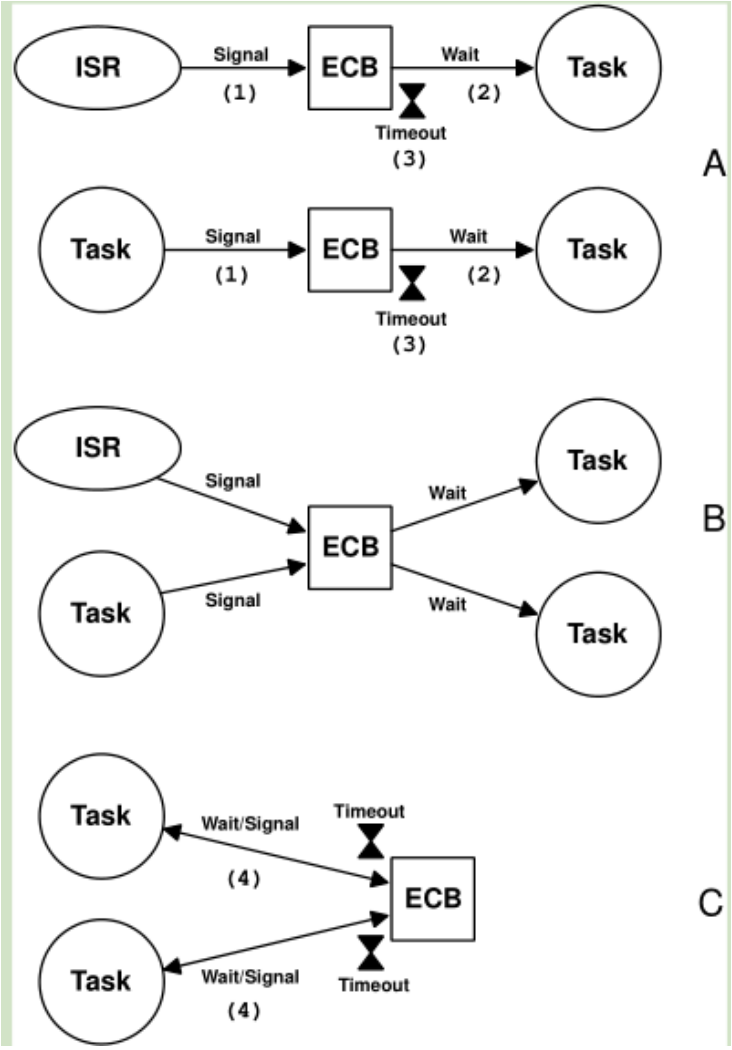
```
INT32U  OSTimeGet (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT32U     ticks;

    OS_ENTER_CRITICAL();
    ticks = OSTime;
    OS_EXIT_CRITICAL();
    return (ticks);
}
```

Listing 5.4　　Obtaining and setting the system time. (Continued)

```
void  OSTimeSet (INT32U ticks)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif

    OS_ENTER_CRITICAL();
    OSTime = ticks;
    OS_EXIT_CRITICAL();
}
```

6 事件控制块

任务之间以及任务与中断服务程序之间通过事件进行通信，管理事件的数据结构为事件控制块 ECB（Event Control Blocks）。



在 UCOSII 系统中，任务或者中断服务程序均可以给事件控制块 ECB 发送信号，而只有任务可以等待事件控制块 ECB 发送过来的信号，中断服务程序是不能等待事件控制块 ECB 给它发送信号的。等待事件的任务可以设置一个最长的等待时间，防止任务无限期的等待下去。

而多个任务可以等待同一事件的发送，而只有优先级最高的任务可以得到该事件而进入就绪态。

事件控制块 ECB 用于管理信号量、互斥型信号量、消息邮箱或者消息队列。

事件控制块 ECB 的数据结构定义如下：

Listing 6.1　　Event control block data structure.

```
typedef struct {
    INT8U   OSEventType;                    /* Event type                          */
    INT8U   OSEventGrp;                     /* Group for wait list                 */
    INT16U  OSEventCnt;                     /* Count (when event is a semaphore) */
    void   *OSEventPtr;                     /* Ptr to message or queue structure */
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; /* Wait list for event to occur      */
} OS_EVENT;
```

参数.OSEventType 定义了事件的具体类型。其值可以包括信号量（OS_EVENT_TYPE_SEM）、互斥型信号量（OS_EVENT_TYPE_MUTEX）、邮箱（OS_EVENT_TYPE_MBOX)、或者消息队列（OS_EVENT_TYPE_Q)。

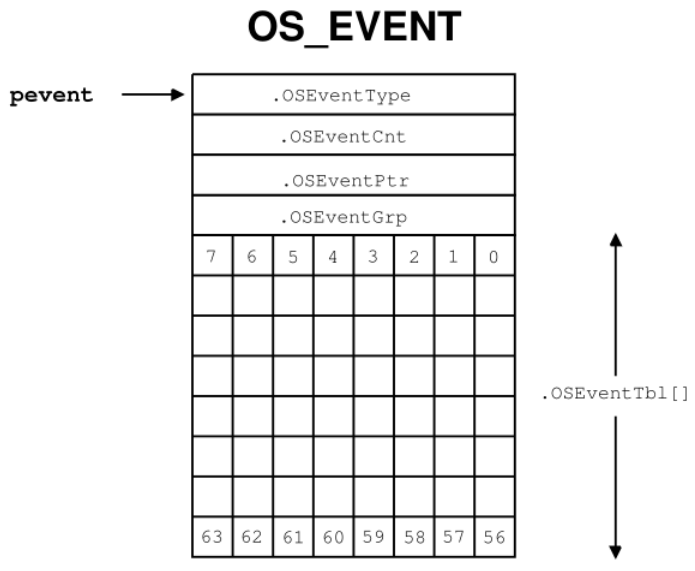
参数.OSEventPtr 定义了指向消息队列或者邮箱的指针，当定义的事件为邮箱时，其指向一个消息，当定义的事件为消息队列时，其指向一个数据结构。

参数.OSEventTbl[]与.OSEventGrp 分别为等待事件表与等待事件组，其定义与就绪任务表和就绪任务组类似，包含的是等待某事件的任务。

参数.OSEventCnt 当用于信号量时，该参数用于信号量的计数器，当用于互斥型信号量时，该参数用于优先级继承优先级计数器。

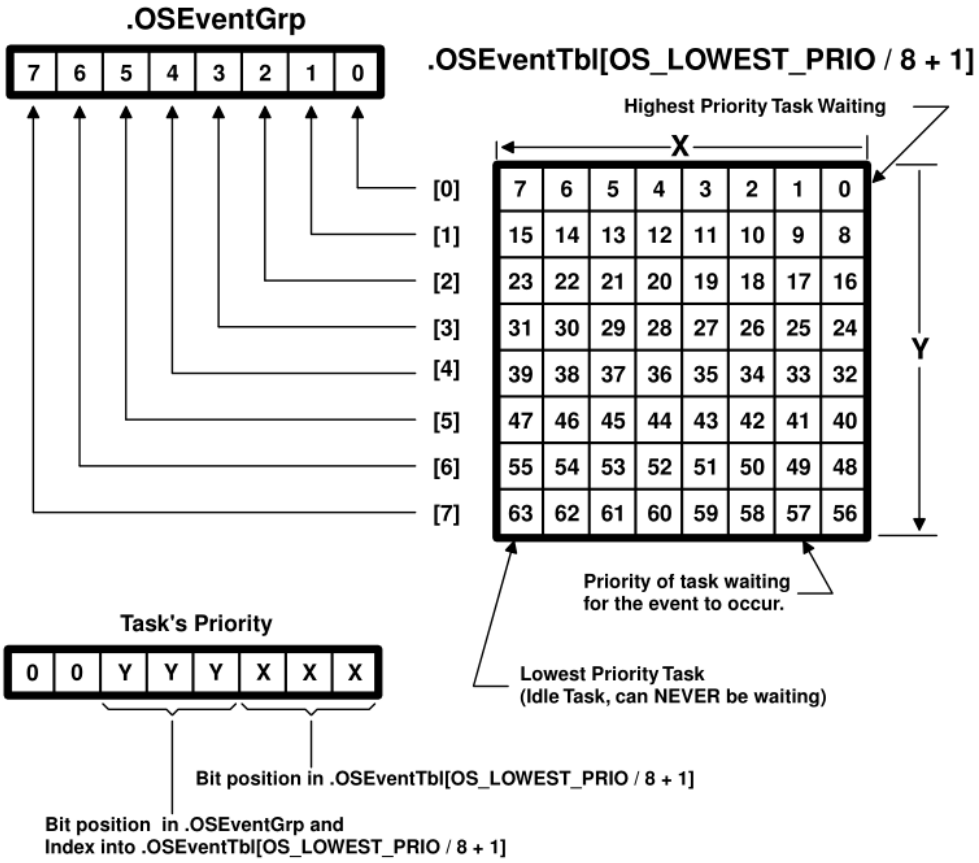
事件控制块 ECB 的结构示意如下所示：

Figure 6.2　　Event Control Block (ECB).



参数 OSEventTbl[]与 OSEventGrp 分别为等待事件表与等待事件组，其定义如下，基本的规则与任务就绪表和任务就绪组类似。

Figure 6.3 Wait list for task waiting for an event to occur.



6.00 将任务置于等待事件的任务列表

将一个任务置于等待事件的任务列表中的代码如下：

Listing 6.2 Making a task wait for an event.

```
    pevent->OSEventGrp              |= OSMapTbl[prio >> 3];
    pevent->OSEventTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

代码中使用了掩码列表 OSMapTbl[]，具体方式与任务就绪表和任务就绪组类似，代码中 prio 是任务的优先级，pevent 是指向任务控制块的指针。掩码列表 OSMapTbl[]内容如下：

Table 6.1 Content of OSMapTbl[].

Index	Bit Mask (Binary)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

6.01 从等待事件的任务列表中使任务脱离等待状态

从等待任务列表中使得任务脱离等待状态的算法正好相反，其代码如下：

Listing 6.3 Removing a task from a wait list.

```
if ((pevent->OSEventTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0) {
    pevent->OSEventGrp &= ~OSMapTbl[prio >> 3];
}
```

6.02 在等待事件的任务列表中查找优先级最高的任务

内核使用了查表的方法在等待事件的任务列表中查找优先级最高的任务，具体代码如下：

Listing 6.4 Finding the highest priority task waiting for the event.

```
    y    = OSUnMapTbl[pevent->OSEventGrp];                    (1)
    x    = OSUnMapTbl[pevent->OSEventTbl[y]];                 (2)
    prio = (y << 3) + x;                                      (3)
```

代码首先确定 OSEventTbl[]中优先级最高的任务所在的组，掩码查询表 OSUnMapTbl[]返回优先级最高的任务所在的组的位置，返回数值 0~7。知道了优先级最高的任务所在的组后，再次通过查表 OSUnMapTbl[]，确认优先级最高的任务在 OSEventTbl[]该组中的具体位置。其查询方式与查询就绪表中最高优先级任务的方式相同。掩码查询表 OSUnMapTbl[]如下：

Listing 6.5 OSUnMapTbl[].

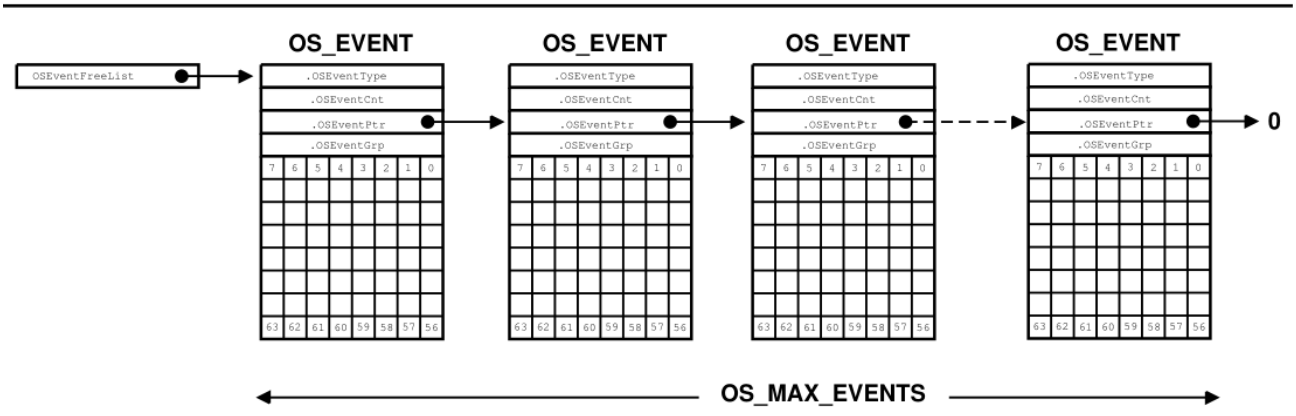
```
INT8U  const  OSUnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x00 to 0x0F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x10 to 0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x20 to 0x2F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x30 to 0x3F */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x40 to 0x4F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x50 to 0x5F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x60 to 0x6F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x70 to 0x7F */
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x80 to 0x8F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0x90 to 0x9F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0xA0 to 0xAF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0xB0 to 0xBF */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0xC0 to 0xCF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0xD0 to 0xDF */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,     /* 0xE0 to 0xEF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0      /* 0xF0 to 0xFF */
};
```

6.03 空余事件控制块链表

在内核进行初始化时，首先根据应用程序所需要的信号量、互斥型信号量、邮箱和消息队列的总数，即内核中的宏定义 OS_MAX_EVENTS 的值，建立 OS_MAX_EVENTS 个空的事件控制块，并将所有空的事件控制块 ECB 链接成一个单向的链表。

每当建立一个信号量、互斥型信号量、邮箱或者消息队列时，就从空闲的事件控制块链表中取出一个空的事件控制块，并对其进行初始化设置。当删除一个信号量、互斥型信号量、邮箱或者消息队列时，就将事件控制块放回空闲的事件控制块单向链表中。

Figure 6.5 List of free ECBs.



6.04 初始化一个事件控制块

内核提供系统函数 OS_EventWaitListInit()对事件控制块中的等待任务列表进行初始化设置，该函数初始化一个空的等待任务列表，当初始化完成后，表中没有任何等待事件的任务，函数的输入参数是创建信号量、互斥型信号量、邮箱或者消息队列时分配的事件控制块指针pevent。函数的代码通过条件编译的方式，减少了系统的开销，具体代码如下：

Listing 6.6　　Initializing the wait list.

```
void  OS_EventWaitListInit (OS_EVENT *pevent)
{
    INT8U  *ptbl;


    pevent->OSEventGrp = 0x00;
    ptbl               = &pevent->OSEventTbl[0];
```

```
#if OS_EVENT_TBL_SIZE > 0
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 1
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 4
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *ptbl              = 0x00;
#endif
}
```

6.05 使一个等待事件的任务进入就绪态

内核提供系统函数 OS_EventTaskRdy()，从等待事件队列中最高优先级任务脱离等待状态，并把该任务置于就绪态。信号量、互斥型信号量、邮箱或者消息队列的 POST 函数会调用该函数。函数的具体代码如下：

Listing 6.7　　Making a task ready to run.

```
INT8U  OS_EventTaskRdy (OS_EVENT *pevent, void *msg, INT8U msk)
{
    OS_TCB *ptcb;
    INT8U   x;
    INT8U   y;
    INT8U   bitx;
    INT8U   bity;
    INT8U   prio;



    y    = OSUnMapTbl[pevent->OSEventGrp];                          (1)
    bity = OSMapTbl[y];                                             (2)
    x    = OSUnMapTbl[pevent->OSEventTbl[y]];                       (3)
    bitx = OSMapTbl[x];                                             (4)
    prio = (INT8U)((y << 3) + x);                                   (5)
    if ((pevent->OSEventTbl[y] &= ~bitx) == 0x00) {                 (6)
        pevent->OSEventGrp &= ~bity;
    }
```

```
    ptcb                =  OSTCBPrioTbl[prio];                      (7)
    ptcb->OSTCBDly      =  0;                                       (8)
    ptcb->OSTCBEventPtr  = (OS_EVENT *)0;                           (9)
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    ptcb->OSTCBMsg      = msg;                                     (10)
#else
    msg                 = msg;
#endif
    ptcb->OSTCBStat     &= ~msk;                                   (11)
    if (ptcb->OSTCBStat == OS_STAT_RDY) {                          (12)
        OSRdyGrp        |=  bity;                                  (13)
        OSRdyTbl[y]     |=  bitx;
    }
    return (prio);                                                 (14)
}
```

6.06 使一个任务进入等待某事件发生状态

内核提供系统函数 OS_EventTaskWait()，使当前任务从就绪表中脱离就绪态，并放到相对应的事件控制块 ECB 的等待任务表中。函数的源码如下：

Listing 6.8　　Making a task wait on an ECB.

```
void  OS_EventTaskWait (OS_EVENT *pevent)
{
    OSTCBCur->OSTCBEventPtr = pevent;                                    (1)
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0x00) {   (2)
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;         (3)
    pevent->OSEventGrp                   |= OSTCBCur->OSTCBBitY;
}
```

函数首先将事件控制块 ECB 的指针放到任务的任务控制块 TCB 中，建立事件与任务的联系，然后将任务从就绪任务表与就绪任务组中删除，并将该任务放到事件控制块 ECB 的等待事件任务列表中。

6.07 由于等待超时而将任务置为就绪态

内核提供系统函数 OS_EventTO()，如果预先在指定的等待时限内任务等待的事件没有发生，那么内核会因为等待超时而将任务置为就绪态。函数的代码如下：

Listing 6.9　　Making a task ready because of a timeout.

```
void  OS_EventTO (OS_EVENT *pevent)
{
    if ((pevent->OSEventTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0x00) {   (1)
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStat     = OS_STAT_RDY;                                  (2)
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                                (3)
}
```

7 信号量管理

内核提供了 6 个系统函数，以对信号量进行管理，函数分别为 OSSemAccept()、OSSemCreate()、OSSemDel()、OSSemPend()、OSSemPost()、OSSemQuery()。函数的编译与否是通过相关的宏定义来配置的。

7.00 建立一个信号量

内核提供系统函数 OSSemCreate()来建立信号量，如果信号量表示一个或者多个事件的发生，这信号量的初始值通常为 0，如果信号量用于对共享资源的访问，则该信号量的初始值为 1，如果信号量用来表示允许任务访问 n 个相同的资源，则该信号量的初始值应该为 n。

系统函数 OSSemCreate()的代码如下：

Listing 7.1　　Creating a semaphore.

```
OS_EVENT  *OSSemCreate (INT16U cnt)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;                                          (1)
#endif
    OS_EVENT  *pevent;



    if (OSIntNesting > 0) {                                     (2)
        return ((OS_EVENT *)0);
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                   (3)
    if (OSEventFreeList != (OS_EVENT *)0) {                     (4)
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;  (5)
    }
```

Listing 7.1　　Creating a semaphore. (Continued)

```
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                              (6)
        pevent->OSEventType = OS_EVENT_TYPE_SEM;                (7)
        pevent->OSEventCnt  = cnt;                              (8)
        pevent->OSEventPtr  = (void *)0;                        (9)
        OS_EventWaitListInit(pevent);                          (10)
    }
    return (pevent);                                           (11)
}
```

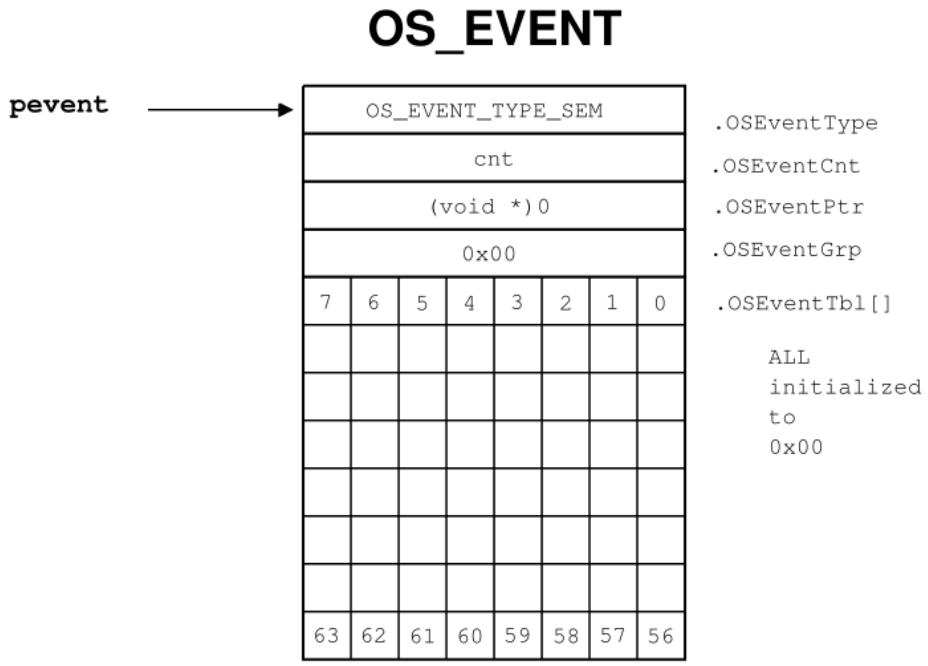
中断服务程序不能调用系统函数 OSSemCreate()，所有的信号量的建立必须在任务级代码中完成。

系统函数 OSSemCreate()从空闲的事件控制块链表中获取一个事件控制块 ECB，调整空闲事件控制块链表的指针指向下一个空闲的事件控制块，将获取的事件控制块 ECB 的事件类型设置为信号量 OS_EVENT_TYPE_SEM，其他信号量的操作函数可以通过检验这个事件类型，以保证所操作的事件类型的正确性。

系统函数 OSSemCreate()的输入参数 cnt 是信号量的初始值，将 OSEventPtr 初始化为 NULL，调用 OS_EventWaitListInit()对事件控制块的等待任务列表与等待任务组进行初始化设置，即将 OSEventGrp 与 OSEventTbl[]清为 0。

系统函数 OSSemCreate()返回一个指向创建的信号量的事件控制块 ECB 的指针。函数返回前的事件控制块内容如下：

Figure 7.2 ECB just before OSSemCreate() returns.



7.01 删除一个信号量

内核提供系统函数 OSSemDel()来删除信号量，当宏定义 OS_SEM_DEL_EN 为 1 时该函数才编译。当调用函数删除信号量时，需要保证此时没有试图操作该信号量的任务。

系统函数 OSSemDel()的代码如下：

Listing 7.2 Deleting a semaphore.

```
OS_EVENT  *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    tasks_waiting;
```

Listing 7.2 Deleting a semaphore. (Continued)

```
    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_DEL_ISR;
        return (pevent);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                   (2)
        *err = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                  (3)
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:
             if (tasks_waiting == FALSE) {                           (5)
                 pevent->OSEventType = OS_EVENT_TYPE_UNUSED;         (6)
                 pevent->OSEventPtr  = OSEventFreeList;              (7)
                 OSEventFreeList     = pevent;
                 OS_EXIT_CRITICAL();
                 *err = OS_NO_ERR;
                 return ((OS_EVENT *)0);                             (8)
             } else {
                 OS_EXIT_CRITICAL();
                 *err = OS_ERR_TASK_WAITING;
                 return (pevent);
             }
```

Listing 7.2 Deleting a semaphore. (Continued)

```
        case OS_DEL_ALWAYS:                                          (9)
             while (pevent->OSEventGrp != 0x00) {                    (10)
                 OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM);
             }
             pevent->OSEventType = OS_EVENT_TYPE_UNUSED;             (11)
             pevent->OSEventPtr  = OSEventFreeList;                  (12)
             OSEventFreeList     = pevent;
             OS_EXIT_CRITICAL();
             if (tasks_waiting == TRUE) {
                 OS_Sched();                                         (13)
             }
             *err = OS_NO_ERR;
             return ((OS_EVENT *)0);                                 (14)

        default:
             OS_EXIT_CRITICAL();
             *err = OS_ERR_INVALID_OPT;
             return (pevent);
    }
}
```

在中断服务程序中不能调用系统函数 OSSemDel()；

函数的输入参数pevent为指向要删除的信号量的事件控制块，opt为删除的选项控制参数，err 为指向返回错误数据的指针。根据输入 opt 的不同，函数执行相应的删除操作，当 opt 为 OS_DEL_NO_PEND 时，则函数只有在没有任务等待信号量时才删除该信号量，当 opt 为 OS_DEL_ALWAYS 时，函数不管有没有任务在等待该信号量，均删除信号量。

当 opt 为 OS_DEL_NO_PEND，且没有任务等待该信号量时，函数将事件控制块的类型设置为未使用，并将其退回到空闲的事件控制块链表中，返回一个 NULL 指针。若有任务在等待该信号量，则不删除信号量，返回错误代码。

当 opt 为 OS_DEL_ALWAYS，函数将所有等待该信号量的任务均设置为进入就绪态，即使得每个任务均得到该信号量，随后函数将事件控制块的类型设置为未使用，并将其退回到空闲的事件控制块链表中，返回一个 NULL 指针。当有任务等待信号量时，进行任务调度。在此时，用户程序需分析将所有等待该信号量的任务均设置为进入就绪态的影响与后果。

7.02 等待一个信号量

内核提供系统函数 OSSemPend()来使得任务等待一个信号量，函数的代码如下：

Listing 7.3 Waiting on a semaphore.

```
void  OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_PEND_ISR;
        return;
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                   (2)
        *err = OS_ERR_PEVENT_NULL;
        return;
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                  (3)
        *err = OS_ERR_EVENT_TYPE;
        return;
    }
#endif
    OS_ENTER_CRITICAL();
```

Listing 7.3 Waiting on a semaphore. (Continued)

```
    if (pevent->OSEventCnt > 0) {                                    (4)
        pevent->OSEventCnt--;                                        (5)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;                                            (6)
        return;
    }
    OSTCBCur->OSTCBStat |= OS_STAT_SEM;                              (7)
    OSTCBCur->OSTCBDly   = timeout;                                  (8)
    OS_EventTaskWait(pevent);                                        (9)
    OS_EXIT_CRITICAL();
    OS_Sched();                                                     (10)
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStat & OS_STAT_SEM) {                        (11)
        OS_EventTO(pevent);                                         (12)
        OS_EXIT_CRITICAL();
        *err = OS_TIMEOUT;                                          (13)
        return;
    }
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                        (14)
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
```

函数首先检查是否是在中断服务程序中调用了系统函数 OSSemPend()，在中断中调用等

待信号量的函数没有意义，因此不可在中断服务程序中调用 OSSemPend()。

当信号量的计数值为非 0，则说明信号量是有效的，而当信号量的计数值为 0，折说明信号量是无效的。代码序号 4、5、6，当信号量的计数值非 0，说明信号量本来是有效的，则将信号量的计数值减去 1，函数返回 OS_NO_ERR，外部调用的函数如果得到该返回值，则可以判断是否可以进一步访问信号量标示的共享资源。

当信号量的计数值为 0，则说明信号量是无效的，则调用 OSSemPend()的任务需要进入休眠态，等待另一个任务发出该信号量。

序号 7 将任务控制块 TCB 中的状态标志设置为等待信号量，使得任务进入休眠态；

序号 8 设置最长等待时限值，该值在 OSTimeTick()函数中每个时钟节拍递减；

真正将任务进入休眠态的是在函数 OS_EventTaskWait()中，函数首先将事件控制块 ECB 的指针放到任务的任务控制块 TCB 中，建立事件与任务的联系，然后将任务从就绪任务表与就绪任务组中删除，并将该任务放到事件控制块 ECB 的等待事件任务列表中。

序号 10 由于得不到信号量，当前任务不再处于就绪态，于是调用任务调度函数，使得下一优先级最高的就绪态任务得到运行。这样调用 OSSemPend()函数的任务将被挂起，直到信号量有效才继续运行。

序号 11、12、13 当信号量有效，或者等待超时时，调用 OSSemPend()函数的任务得到继续运行，函数首先判断任务控制块中的状态标志是否仍然处于等待信号量的状态，如果是，则说明任务并没有被 OSSemPost()发出的信号量唤醒，而是由于等待超时被 OSTimeTick()函数就绪的，此时函数调用 OS_EventTO()将任务从等待任务列表中删除，并返回超时的错误代码。

如果函数判断任务控制块中的状态标志不是处于等待信号量的状态，则说明 OSSemPost()已经发出了信号量，此时将指向信号量 ECB 的指针从任务控制块 TCB 中删除，返回无错误的代码，调用 OSSemPend()函数的任务继续向下运行。

7.03 发出一个信号量

内核提供系统函数 OSSemPost()来使得任务发出一个信号量，函数的代码如下：

Listing 7.4 Signaling a semaphore.

```
INT8U  OSSemPost (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
```

Listing 7.4 Signaling a semaphore. (Continued)

```
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                    (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                   (2)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                 (3)
        OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM);              (4)
        OS_EXIT_CRITICAL();
        OS_Sched();                                                   (5)
        return (OS_NO_ERR);
    }
    if (pevent->OSEventCnt < 65535) {
        pevent->OSEventCnt++;                                         (6)
        OS_EXIT_CRITICAL();
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_SEM_OVF);                                              (7)
}
```

函数调用系统函数 OS_EventTaskRdy()把优先级最高的任务从等待任务列表中删除，并使其进入就绪态，然后调用 OSSched()任务调度，如果是优先级最高的就绪态任务，则进行任务切换，此时调用 OSSemPost()的任务就不再继续运行了。即因为 OSSemPost()使得更重要的任务得到了运行。

注意在系统函数 OS_EventTaskRdy()，会将任务控制块的状态标志置为无效状态，即下面的语句，所以可在 OSSemPend()函数中通过任务控制块的状态标志判断等待信号量的任务被唤醒是因为另一个任务发出了信号量，或者是由于等待时间超时。

```
    ptcb->OSTCBStat     &= ~msk;                                      (11)
```

如果没有任务在等待该信号量，则该信号量的计数值就简单的加 1。

7.04 无等待地请求一个信号量

当任务请求一个信号量时，如果信号量暂时无效，可以让任务继续运行下去，而不是进入休眠态，内核提供了系统函数 OSSemAccept()实现这样的功能，具体代码如下：

Listing 7.5　　Getting a semaphore without waiting.

```
INT16U  OSSemAccept (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT16U     cnt;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                  (1)
        return (0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                 (2)
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
    cnt = pevent->OSEventCnt;                                       (3)
    if (cnt > 0) {                                                  (4)
        pevent->OSEventCnt--;                                       (5)
    }
    OS_EXIT_CRITICAL();
    return (cnt);                                                   (6)
}
```

如果信号量有效，即信号量的计数器非 0，则将信号量计数器减去 1，函数返回信号量的计数器的值。

调用函数对返回值进行检查，如果返回值为 0，则说明信号量是无效的，而如果返回值为非 0，则说明该信号量是有效的，该返回值也说明了信号量当前可用的资源数。

7.05 查询一个信号量的当前状态

内核提供系统函数 OSSemQuery()来查询一个信号量的当前状态。函数的具体代码如下:

Listing 7.6 Obtaining the status of a semaphore.

```
INT8U  OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U     *psrc;
    INT8U     *pdest;
```

Listing 7.6 Obtaining the status of a semaphore. (Continued)

```
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                          (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                         (2)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    pdata->OSEventGrp = pevent->OSEventGrp;                                 (3)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
#if OS_EVENT_TBL_SIZE > 0
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 1
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 4
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *pdest++          = *psrc++;
#endif
```

Listing 7.6 Obtaining the status of a semaphore. (Continued)

```
#if OS_EVENT_TBL_SIZE > 7
    *pdest            = *psrc;
#endif
    pdata->OSCnt      = pevent->OSEventCnt;                    (4)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

函数 OSSemQuery()有 2 个输入参数，输入参数 pevent 是指向信号量对应事件控制块 ECB 的指针，输入参数 pdata 是指向用于记录信号量信息的数据结构 OS_SEM_DATA 的指针。因此在调用函数 OSSemQuery()之前，需要定义结构体 OS_SEM_DATA 的变量，用于存储信号量的相关信息数据。

结构体 OS_SEM_DATA 中只包括 OSCnt、OSEventTbl[]、OSEventGrp。

8 互斥型信号量管理

UCOSII 系统内核提供了互斥型信号量对共享资源进行独占式管理，互斥型信号量（mutual exclusion semaphores）是二值信号量，其可以用于解决优先级反转的问题。当高优先级的任务需要使用某一共享资源，但是此时共享资源却被一个低优先级的任务占用，则会发送优先级反转。为了解决此问题，内核可以将低优先级的任务的优先级提高到高于高优先级任务的优先级，使得低优先级任务尽快的释放共享资源。

以下代码为范例：

Listing 8.1 Mutex use example.

```
OS_EVENT *ResourceMutex;
OS_STK    TaskPrio10Stk[1000];
OS_STK    TaskPrio15Stk[1000];
OS_STK    TaskPrio20Stk[1000];
```

Listing 8.1 Mutex use example. (Continued)

```
void main (void)
{
    INT8U err;


    OSInit();                                                          (1)
     ---------- Application Initialization ----------
    OSMutexCreate(9, &err);                                            (2)
    OSTaskCreate(TaskPrio10, (void *)0, &TaskPrio10Stk[999], 10);      (3)
    OSTaskCreate(TaskPrio15, (void *)0, &TaskPrio15Stk[999], 15);
    OSTaskCreate(TaskPrio20, (void *)0, &TaskPrio20Stk[999], 20);
     ---------- Application Initialization ----------
    OSStart();                                                         (4)
}


void TaskPrio10 (void *pdata)                                  /* Task #1 */
{
    INT8U err;


    pdata = pdata;
    while (1) {
         --------- Application Code ----------
        OSMutexPend(ResourceMutex, 0, &err);
         ------- Access common resource ------
        OSMutexPost(ResourceMutex);
         --------- Application Code ----------
    }
}
```

Listing 8.1 Mutex use example. (Continued)

```
void TaskPrio15 (void *pdata)                                    /* Task #2 */
{
    INT8U err;


    pdata = pdata;
    while (1) {
         --------- Application Code ----------
        OSMutexPend(ResourceMutex, 0, &err);
         ------- Access common resource ------
        OSMutexPost(ResourceMutex);
         --------- Application Code ----------
    }
}


void TaskPrio20 (void *pdata)                                    /* Task #3 */
{
    INT8U err;


    pdata = pdata;
    while (1) {
         --------- Application Code ----------
        OSMutexPend(ResourceMutex, 0, &err);
         ------- Access common resource ------
        OSMutexPost(ResourceMutex);
         --------- Application Code ----------
    }
}
```

假设有 3 个任务需要使用共享资源。任务 1 的优先级最高，为 10，任务 2 的优先级为 15，任务 3 的优先级最低，为 20。优先级 9 保留作为优先级反转时的优先级继承优先级 PIP。

假设程序运行的某一时刻，优先级最低的任务 3 首先申请到了共享资源并使用，一段时间后优先级最高的任务 1 就绪，剥夺了 CPU 的使用权，当任务 1 需要使用共享资源时，通过调用系统函数 OSMutexPend()申请共享资源的使用权，系统函数 OSMutexPend()注意到此时有低优先级任务占用共享资源，则将低优先级任务 3 的优先级升至优先级 9，并强制任务调度回任务 3，任务 3 继续运行，试图尽快释放共享资源。

当共享资源使用结束，任务 3 调用 OSMutexPost()释放资源，系统函数 OSMutexPost()将任务 3 的优先级恢复到原来水平，然后将 CPU 的使用权交由需要共享资源的任务 1。

在 UCOSII 系统中，互斥型信号量由 3 个元素组成，即指示 Mutex 是否可以使用的标志，优先级继承的优先级，以及等待 Mutex 的任务列表。

内核提供 6 中系统函数用来实现互斥型信号量，分别为 OSMutexCreate()、OSMutexDel()、

OSMutexPend()、OSMutexPost()、OSMutexAccept()、OSMutexQuery()。

互斥型信号量只能由任务调用使用。

8.00 建立一个互斥型信号量

内核提供系统函数 OSMutexCreate()来建立一个互斥型信号量，互斥型信号量的初始值为1，表示共享资源初始时是可以使用的。系统函数 OSMutexCreat()的代码如下：

Listing 8.2 Creating a mutex.

```
OS_EVENT *OSMutexCreate (INT8U prio, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_EVENT *pevent;


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_CREATE_ISR;
        return ((OS_EVENT *)0);
    }
#if OS_ARG_CHK_EN
    if (prio >= OS_LOWEST_PRIO) {                                    (2)
        *err = OS_PRIO_INVALID;
        return ((OS_EVENT *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] != (OS_TCB *)0) {                         (3)
        *err = OS_PRIO_EXIST;
        OS_EXIT_CRITICAL();
        return ((OS_EVENT *)0);
    }
    OSTCBPrioTbl[prio] = (OS_TCB *)1;                                (4)
    pevent             = OSEventFreeList;                            (5)
    if (pevent == (OS_EVENT *)0) {
        OSTCBPrioTbl[prio] = (OS_TCB *)0;
        OS_EXIT_CRITICAL();
        *err               = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    OSEventFreeList     = (OS_EVENT *)OSEventFreeList->OSEventPtr;   (6)
    OS_EXIT_CRITICAL();
```

Listing 8.2 Creating a mutex. (Continued)

```
    pevent->OSEventType = OS_EVENT_TYPE_MUTEX;                    (7)
    pevent->OSEventCnt  = (prio << 8) | OS_MUTEX_AVAILABLE;       (8)
    pevent->OSEventPtr  = (void *)0;                              (9)
    OSEventWaitListInit(pevent);                                  (10)
    *err                = OS_NO_ERR;
    return (pevent);                                              (11)

    }
}
```

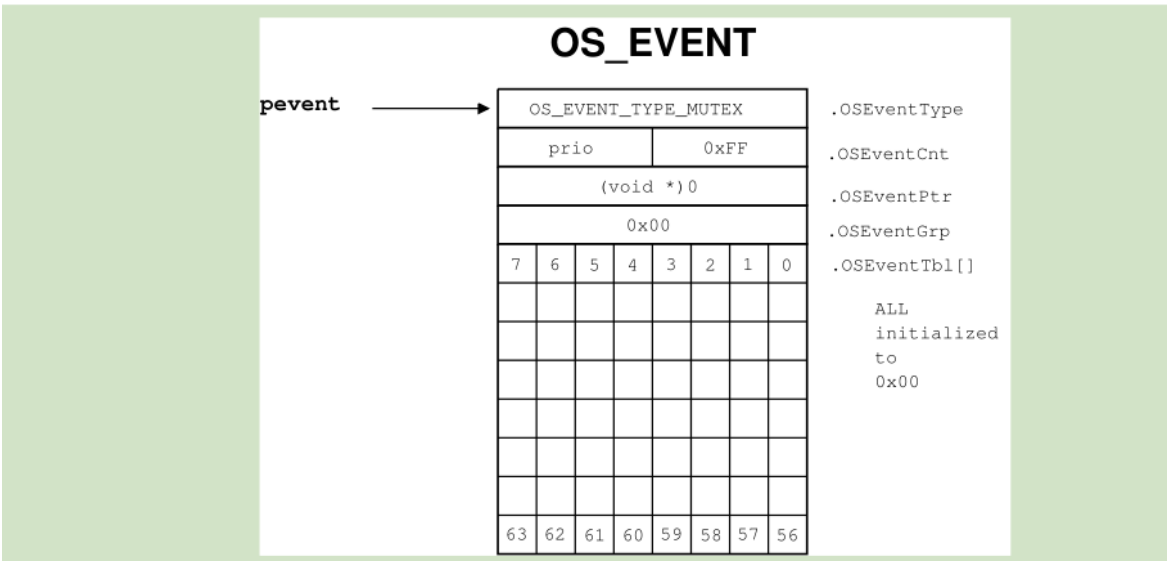
系统函数 OSMutexCreat()的输入参数为两个，prio 为优先级继承的优先级，err 为指向错误代码的指针。

不允许在中断服务函数中调用系统函数 OSMutexCreat()。

在代码序号 8 处，事件控制块 ECB 的 OSEventCnt 的高 8 位保存优先级继承的优先级，低 8 位在资源无任务占用时为 FFH，有任务占用时为占用 Mutex 的任务的优先级。宏 OS_MUTEX_AVAILABLE 定义为 00FFH。

在系统函数 OSMutexCreat()返回前，互斥型信号量的事件控制块 ECB 的结构如下：

Figure 8.2 ECB just before OSMutexCreate() returns.



8.01 删除一个互斥型信号量

内核提供系统函数 OSMutexDel()来删除一个互斥型信号量，一般在删除一个互斥型信号量之前，需要删除可能用到该互斥型信号量的所有任务，以防止某一任务试图使用已经删除掉的互斥型信号量。系统函数 OSMutexDel()的代码如下：

Listing 8.3 Deleting a mutex.

```
OS_EVENT  *OSMutexDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    tasks_waiting;


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_DEL_ISR;
        return (pevent);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                                   (2)
        *err = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
```

Listing 8.3 Deleting a mutex. (Continued)

```
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {                (3)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:
             if (tasks_waiting == FALSE) {                           (5)
                 pevent->OSEventType = OS_EVENT_TYPE_UNUSED;         (6)
                 pevent->OSEventPtr  = OSEventFreeList;              (7)
                 OSEventFreeList     = pevent;
                 OS_EXIT_CRITICAL();
                 *err = OS_NO_ERR;
                 return ((OS_EVENT *)0);                             (8)
             } else {
                 OS_EXIT_CRITICAL();
                 *err = OS_ERR_TASK_WAITING;
                 return (pevent);
             }

        case OS_DEL_ALWAYS:                                          (9)
             while (pevent->OSEventGrp != 0x00) {                   (10)
                 OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
             }
             pevent->OSEventType = OS_EVENT_TYPE_UNUSED;            (11)
             pevent->OSEventPtr  = OSEventFreeList;                 (12)
             OSEventFreeList     = pevent;
             OS_EXIT_CRITICAL();
             if (tasks_waiting == TRUE) {                           (13)
                 OS_Sched();
             }
             *err = OS_NO_ERR;
             return ((OS_EVENT *)0);                                (14)
```

Listing 8.3 Deleting a mutex. (Continued)

```
        default:
             OS_EXIT_CRITICAL();
             *err = OS_ERR_INVALID_OPT;
             return (pevent);
    }
}
```

不允许在中断服务程序中调用系统函数 OSMutexDel()。

系统函数 OSMutexDel()允许有 2 中删除方式，即当 opt 为 OS_DEL_NO_PEND 时，在没有任务因这个 Mutex 而挂起的情况下删除，当 opt 为 OS_DEL_ALWAYS 时，不管是否有任务在等待这个 Mutex，强制删除。

代码结构方式同 7.01 节相关内容，不详细解读。

8.02 等待一个互斥型信号量

内核提供系统函数 OSMutexPend()来等待一个互斥型信号量，函数的代码如下：

Listing 8.4 Waiting for a mutex.

```
void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      pip;
    INT8U      mprio;
    BOOLEAN    rdy;
    OS_TCB    *ptcb;


    if (OSIntNesting > 0) {                                            (1)
        *err = OS_ERR_PEND_ISR;
        return;
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                                     (2)
        *err = OS_ERR_PEVENT_NULL;
        return;
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {                  (3)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return;
    }
#endif
                                                                       (4)
    if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;                   (5)
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;                     (6)
        pevent->OSEventPtr  = (void *)OSTCBCur;                        (7)
        OS_EXIT_CRITICAL();
        *err  = OS_NO_ERR;
        return;
    }
```

Listing 8.4 Waiting for a mutex. (Continued)

```
    pip   = (INT8U)(pevent->OSEventCnt >> 8);                                   (8)
    mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);                (9)
    ptcb  = (OS_TCB *)(pevent->OSEventPtr);                                     (10)

    if (ptcb->OSTCBPrio != pip && mprio > OSTCBCur->OSTCBPrio) {                (11)
        if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {               (12)
                                                                                (13)

            if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
                OSRdyGrp &= ~ptcb->OSTCBBitY;
            }
            rdy = TRUE;                                                         (14)
        } else {
            rdy = FALSE;                                                        (15)
        }
        ptcb->OSTCBPrio         = pip;                                          (16)
        ptcb->OSTCBY            = ptcb->OSTCBPrio >> 3;
        ptcb->OSTCBBitY         = OSMapTbl[ptcb->OSTCBY];
        ptcb->OSTCBX            = ptcb->OSTCBPrio & 0x07;
        ptcb->OSTCBBitX         = OSMapTbl[ptcb->OSTCBX];
        if (rdy == TRUE) {                                                      (17)
            OSRdyGrp               |= ptcb->OSTCBBitY;
            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        }
        OSTCBPrioTbl[pip]       = (OS_TCB *)ptcb;
    }
    OSTCBCur->OSTCBStat |= OS_STAT_MUTEX;                                       (18)
    OSTCBCur->OSTCBDly   = timeout;                                             (19)
    OS_EventTaskWait(pevent);                                                   (20)
    OS_EXIT_CRITICAL();
    OS_Sched();                                                                 (21)
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStat & OS_STAT_MUTEX) {                                  (22)
        OS_EventTO(pevent);                                                     (23)
        OS_EXIT_CRITICAL();
        *err = OS_TIMEOUT;                                                      (24)
        return;
    }
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                                    (25)
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
```

不能在中断服务程序中调用系统函数 OSMutexPend()。

如果事件控制块 ECB 的 OSEventCnt 的低 8 位为 FFH，则此互斥型信号量有效，且没有任务占用该互斥型信号量。则系统函数 OSMutexPend()将事件控制块 ECB 的 OSEventCnt 的低 8 位设置为调用该函数的任务的优先级，事件控制块 ECB 的 OSEventPtr 指向当前任务的任务控制块 TCB。函数运行结束。

如果事件控制块 ECB 的 OSEventCnt 的低 8 位非 FFH，则此互斥型信号量被其他任务占用，则当前任务应当进入休眠态，直到占用互斥型信号量的任务释放了该信号量。代码序号 8、9、10 处提取了优先级继承的优先级 pip，占用信号量的任务的优先级 mprio，以及指向占用信号量的任务控制块的指针 ptcb。

如果占用互斥型信号量的任务优先级比调用系统函数 OSMutexPend()的任务优先级低，则

占用互斥型信号量的任务优先级将被提升到 pip，以尽快释放互斥型信号量。

代码序号 12，函数确认占用互斥型信号量的任务是否进入就绪态，如果任务处于就绪态，则取消占用互斥型信号量的任务在原来优先级下的就绪态，置 rdy 为 TRUE。如果任务未处于就绪态，则置 rdy 为 FALSE。

随后函数设置占用互斥型信号量的任务优先级为 pip，若 rdy 为 TRUE，在就绪表与就绪组中设置 pip 优先级下任务的就绪态。

为了让调用系统函数OSMutexPend()的任务进入休眠态，设置任务控制块TCB中的标志，标明任务因为等待 Mutex 而挂起。缓存等待超时参数。调用系统函数 OS_EventTaskWait()使得调用系统函数 OSMutexPend()的任务进入休眠态。

当前任务已经不处于就绪态，代码序号 21 处调用任务调度函数，运行下一个优先级最高的就绪态任务。

当 Mutex 被释放，则调用系统函数 OSMutexPend()的任务又将变成优先级最高的任务，OS_Sched()函数返回，OSMutexPend()代码继续运行。

函数检查任务控制块TCB的状态标志，如果还是等待Mutex，则说明没有从OSMutexPost()函数调用中得到 Mutex，而是由于等待超时使得当前任务进入就绪态。在这种情况下，调用OS_EventTO()函数将任务从等待 Mutex 的任务列表中删除，返回超时错误代码。

函数检查任务控制块 TCB 的状态标志，如果不再是等待 Mutex，说明 Mutex 已经被释放，则释放事件控制块指针。

8.03 释放一个互斥型信号量

内核提供系统函数 OSMutexPost()来释放一个互斥型信号量，函数的代码如下：

Listing 8.5　　Signaling a mutex.

```
INT8U OSMutexPost (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      pip;
    INT8U      prio;


    if (OSIntNesting > 0) {                                          (1)
        return (OS_ERR_POST_ISR);
    }
```

Listing 8.5 Signaling a mutex. (Continued)

```
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                                          (2)
        return (OS_ERR_PEVENT_NULL);
    }
#endif
    OS_ENTER_CRITICAL();
    pip  = (INT8U)(pevent->OSEventCnt >> 8);
    prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {                       (3)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (OSTCBCur->OSTCBPrio != pip ||
        OSTCBCur->OSTCBPrio != prio) {                                      (4)
        OS_EXIT_CRITICAL();
        return (OS_ERR_NOT_MUTEX_OWNER);
    }
#endif
    if (OSTCBCur->OSTCBPrio == pip) {                                       (5)

                                                                            (6)
        if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBPrio         = prio;
        OSTCBCur->OSTCBY            = prio >> 3;
        OSTCBCur->OSTCBBitY         = OSMapTbl[OSTCBCur->OSTCBY];
        OSTCBCur->OSTCBX            = prio & 0x07;
        OSTCBCur->OSTCBBitX         = OSMapTbl[OSTCBCur->OSTCBX];
        OSRdyGrp                   |= OSTCBCur->OSTCBBitY;
        OSRdyTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
        OSTCBPrioTbl[prio]          = (OS_TCB *)OSTCBCur;
    }
    OSTCBPrioTbl[pip] = (OS_TCB *)1;
    if (pevent->OSEventGrp != 0x00) {                                       (7)
                                                                            (8)
        prio               = OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
        pevent->OSEventCnt &= 0xFF00;                                       (9)
        pevent->OSEventCnt |= prio;
```

Listing 8.5 Signaling a mutex. (Continued)

```
        pevent->OSEventPtr  = OSTCBPrioTbl[prio];
        OS_EXIT_CRITICAL();
        OS_Sched();                                                         (10)
        return (OS_NO_ERR);
    }
    pevent->OSEventCnt |= 0x00FF;                                           (11)
    pevent->OSEventPtr  = (void *)0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

在中断服务程序中不能调用系统函数 OSMutexPost()。

其他代码不详细叙述。

8.04 无等待地获取互斥型信号量

内核提供系统函数OSMutexAccept()来实现当前Mutex无效时，不让当前任务进入休眠态。具体代码如下：

Listing 8.6　　Getting a mutex without waiting.

```
INT8U OSMutexAccept (OS_EVENT *pevent, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_PEND_ISR;
        return (0);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {
        *err = OS_ERR_PEVENT_NULL;
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
                                                                     (2)
```

Listing 8.6　　Getting a mutex without waiting. (Continued)

```
    if ((pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;                 (3)
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
        pevent->OSEventPtr  = (void *)OSTCBCur;                      (4)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (1);
    }
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (0);
}
```

8.05 获取互斥型信号量的当前状态

内核提供系统函数 OSMutexQuery()来获取互斥型信号量的当前状态。代码如下：

Listing 8.7 Obtaining the status of a mutex.

```
INT8U OSMutexQuery (OS_EVENT *pevent, OS_MUTEX_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U     *psrc;
    INT8U     *pdest;
```

Listing 8.7 Obtaining the status of a mutex. (Continued)

```
    if (OSIntNesting > 0) {                                          (1)
        return (OS_ERR_QUERY_ISR);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                                   (2)
        return (OS_ERR_PEVENT_NULL);
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {                (3)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    pdata->OSMutexPIP  = (INT8U)(pevent->OSEventCnt >> 8);           (4)
    pdata->OSOwnerPrio = (INT8U)(pevent->OSEventCnt & 0x00FF);
    if (pdata->OSOwnerPrio == 0xFF) {
        pdata->OSValue = 1;                                          (5)
    } else {
        pdata->OSValue = 0;                                          (6)
    }
    pdata->OSEventGrp  = pevent->OSEventGrp;                         (7)
    psrc               = &pevent->OSEventTbl[0];
    pdest              = &pdata->OSEventTbl[0];
#if OS_EVENT_TBL_SIZE > 0
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 1
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *pdest++           = *psrc++;
#endif
```

Listing 8.7 Obtaining the status of a mutex. (Continued)

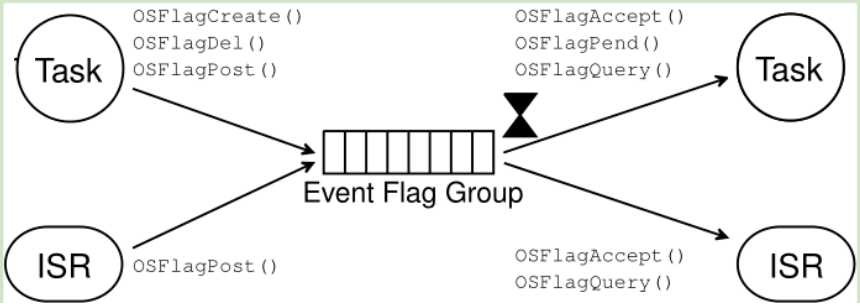
```
#if OS_EVENT_TBL_SIZE > 4
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *pdest             = *psrc;
#endif
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

数据结构体 OS_MUTEX_DATA 包括了 Mutex 的优先级继承优先级 PIP（OSMutexPIP），占用 Mutex 的任务的优先级（OSMutexPrio），以及 Mutex 的值（OSMutexValue），该值为 1 表示 Mutex 可以使用，该值为 0 表示 Mutex 已经被其他任务占用。

9 事件标志组管理

UCOSII 内核提供了 6 个系统函数来完成对事件标志组的管理，具体为 OSFlagCreate()、OSFlagPend()、OSFlagPost()、OSFlagDel()、OSFlagAccept()、OSFlagQuery()。

Figure 9.1 µC/OS-II event flag services.



9.00 深入事件标志组

内核提供的事件标志组的数据结构体 OS_FLAG_GRP 如下：

Listing 9.1 Event flag group data structure.

```
typedef struct {
    INT8U     OSFlagType;                                    (1)
    void     *OSFlagWaitList;                                (2)
    OS_FLAGS  OSFlagFlags;                                   (3)
} OS_FLAG_GRP;
```

OSFlagType 变量用来检验指针的类型是否是指向事件标志组的指针；

OSFlagWaitList 变量包含一个等待事件标志组的任务列表；

OSFlagFlags 变量包含了一系列表明当前事件标志状态的位，位的数目又数据类型 OS_FLAGS 定义，可以为 8 位，16 位或者 32 位。

内核提供的另一种跟事件标志组相关的数据结构 OS_FLAG_NODE 如下：

Listing 9.2 Event flag group node data structure.

```
typedef struct {
    void     *OSFlagNodeNext;                                (1)
    void     *OSFlagNodePrev;
    void     *OSFlagNodeTCB;                                 (2)
    void     *OSFlagNodeFlagGrp;                             (3)
    OS_FLAGS  OSFlagNodeFlags;                               (4)
    INT8U     OSFlagNodeWaitType;                            (5)
} OS_FLAG_NODE;
```

OSFlagNodeNext 与 OSFlagNodePrev 变量用来构建双向的 OS_FLAG_NODE 数据结构链表，以方便地向链表中插入或者删除一个 OS_FLAG_NODE 结构；

OSFlagNodeTCB 变量指针指向某个等待事件标志的任务控制块 TCB；

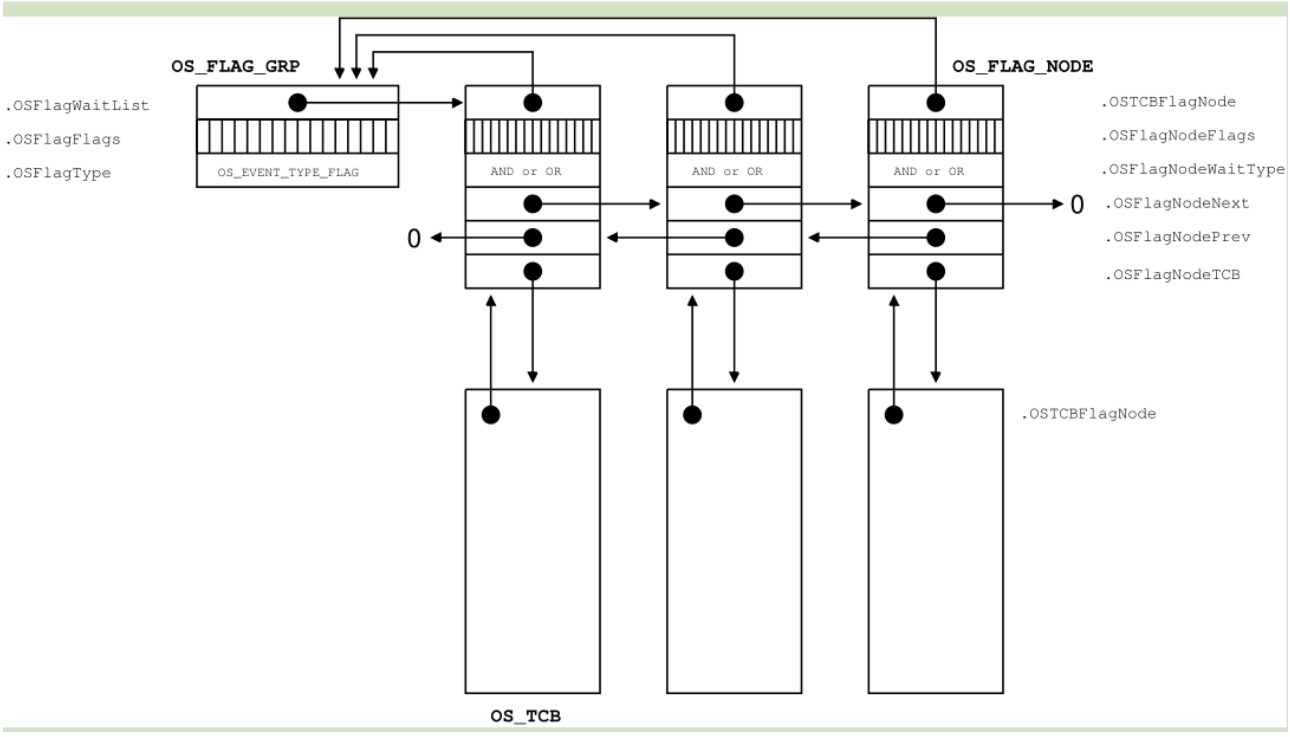
OSFlagNodeFlagGrp 变量是反向指向事件标志组的指针；

OSFlagNodeFlags 变量用来指明任务等待事件标志组中的哪些事件标志。位的数目又数据类型 OS_FLAGS 定义，可以为 8 位，16 位或者 32 位。例如如果任务等待事件标志组中的事件标志 0、4、6、7，则 OSFlagNodeFlags 的值为 D1H。

OSFlagNodeWaitType 变量指明任务是等待事件标志组中所有的事件标志都发生(AND)，还是任何一个事件标志发生（OR）。OSFlagNodeWaitType 的值如下：

```
/*
*********************************************************************************************************
*                                          EVENT FLAGS
*********************************************************************************************************
*/
#define  OS_FLAG_WAIT_CLR_ALL           0u  /* Wait for ALL    the bits specified to be CLR (i.e. 0)   */
#define  OS_FLAG_WAIT_CLR_AND           0u

#define  OS_FLAG_WAIT_CLR_ANY           1u  /* Wait for ANY of the bits specified to be CLR (i.e. 0)   */
#define  OS_FLAG_WAIT_CLR_OR            1u

#define  OS_FLAG_WAIT_SET_ALL           2u  /* Wait for ALL    the bits specified to be SET (i.e. 1)   */
#define  OS_FLAG_WAIT_SET_AND           2u

#define  OS_FLAG_WAIT_SET_ANY           3u  /* Wait for ANY of the bits specified to be SET (i.e. 1)   */
#define  OS_FLAG_WAIT_SET_OR            3u

```

数据结构 OS_FLAG_GRP、OS_FLAG_NODE 以及 TCB 的关系如下图：



9.01 建立一个事件标志组

内核提供系统函数 OSFlagCreate()来建立一个事件标志组。具体代码如下：

Listing 9.3　　Creating an event flag group.

```
OS_FLAG_GRP  *OSFlagCreate (OS_FLAGS flags, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR    cpu_sr;
#endif
    OS_FLAG_GRP *pgrp;


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_CREATE_ISR;
        return ((OS_FLAG_GRP *)0);
    }
    OS_ENTER_CRITICAL();
    pgrp = OSFlagFreeList;                                           (2)
    if (pgrp != (OS_FLAG_GRP *)0) {                                  (3)
                                                                     (4)
        OSFlagFreeList       = (OS_FLAG_GRP *)OSFlagFreeList->OSFlagWaitList;
        pgrp->OSFlagType     = OS_EVENT_TYPE_FLAG;                   (5)
        pgrp->OSFlagFlags    = flags;                                (6)
        pgrp->OSFlagWaitList = (void *)0;                            (7)
        OS_EXIT_CRITICAL();
        *err                 = OS_NO_ERR;
    } else {
        OS_EXIT_CRITICAL();
        *err                 = OS_FLAG_GRP_DEPLETED;
    }
    return (pgrp);                                                   (8)
}
```

在中断服务程序中不允许建立一个事件标志组；

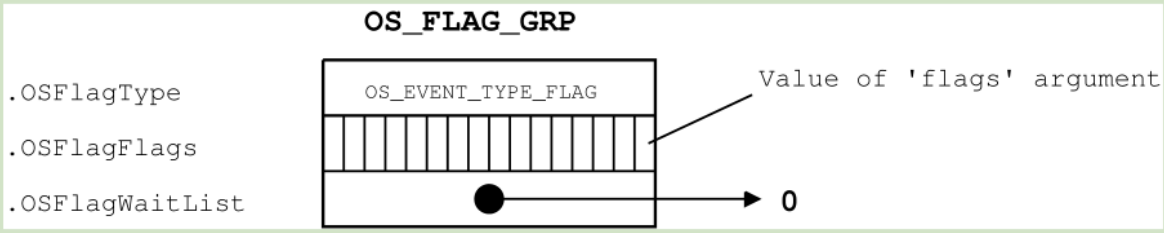
系统函数 OSFlagCreat()从空闲的事件标志组链表中取得一个空闲的事件标志组，在 UCOSII 初始化时，内核会建立空闲的事件标志组链表，OSFlagFreeList 指向表头，除此之外，还会建立空闲的任务控制块链表（OSTCBFreeList 指向表头）、建立空闲的事件控制块链表（OSEventFreeList 指向表头）、建立空闲的消息队列链表（OSQFreeList 指向表头）、建立空闲的内存管理链表（OSMemFreeList 指向表头）。

随后系统函数 OSFlagCreat()初始化事件标志组数据结构中的各个变量。使用函数的输入参数 flags 初始化 OSFlagFlags，通常设置为全 1 或者全 0。刚建立事件标志组时，没有任务等待该事件标志组，则等待链表 OSFlagWaitList 初始化为 NULL。

系统函数 OSFlagCreat()返回指向刚建立的事件标志组的指针。

事件标志组刚建立后，数据结构体如下：

Figure 9.3 Event flag group just before OSFlagCreate() returns.



9.02 删除一个事件标志组

内核提供系统函数 OSFlagDel()来删除一个事件标志组。具体代码如下：

Listing 9.4 Deleting an event flag group.

```
OS_FLAG_GRP  *OSFlagDel (OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR     cpu_sr;
#endif
    BOOLEAN       tasks_waiting;
    OS_FLAG_NODE *pnode;
```

Listing 9.4 Deleting an event flag group. (Continued)

```
    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_DEL_ISR;
        return (pgrp);
    }
#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                                  (2)
        *err = OS_FLAG_INVALID_PGRP;
        return (pgrp);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {                    (3)
        *err = OS_ERR_EVENT_TYPE;
        return (pgrp);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pgrp->OSFlagWaitList != (void *)0) {                         (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:                                         (5)
             if (tasks_waiting == FALSE) {
                 pgrp->OSFlagType     = OS_EVENT_TYPE_UNUSED;
                 pgrp->OSFlagWaitList = (void *)OSFlagFreeList;      (6)
                 OSFlagFreeList       = pgrp;
                 OS_EXIT_CRITICAL();
                 *err                 = OS_NO_ERR;
                 return ((OS_FLAG_GRP *)0);                          (7)
             } else {
                 OS_EXIT_CRITICAL();
                 *err                 = OS_ERR_TASK_WAITING;
                 return (pgrp);
             }
```

Listing 9.4 Deleting an event flag group. (Continued)

```
        case OS_DEL_ALWAYS:                                          (8)
             pnode = pgrp->OSFlagWaitList;
             while (pnode != (OS_FLAG_NODE *)0) {                    (9)
                 OS_FlagTaskRdy(pnode, (OS_FLAGS)0);
                 pnode = pnode->OSFlagNodeNext;
             }
             pgrp->OSFlagType     = OS_EVENT_TYPE_UNUSED;
             pgrp->OSFlagWaitList = (void *)OSFlagFreeList;          (10)
             OSFlagFreeList       = pgrp;
             OS_EXIT_CRITICAL();
             if (tasks_waiting == TRUE) {                            (11)
                 OS_Sched();
             }
             *err = OS_NO_ERR;
             return ((OS_FLAG_GRP *)0);                              (12)

        default:
             OS_EXIT_CRITICAL();
             *err = OS_ERR_INVALID_OPT;
             return (pgrp);
    }
}
```

系统函数 OSFlagDel()提供不同的删除选型，当输入参数 opt 为 OS_DEL_NO_PEND 时，则在没有任务等待事件标志组的情况下才删除事件标志组。当输入参数 opt 为 OS_DEL_ALWAYS 时，则不管是否有等待任务，立即删除事件标志组。

代码不再详细论述。

9.03 等待事件标志组的事件标志位

内核提供系统函数 OSFlagPend()来实现等待事件标志组中的事件标志位。代码如下：

Listing 9.5 Waiting for event(s) of an event flag group.

```
OS_FLAGS  OSFlagPend (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR     cpu_sr;
#endif
    OS_FLAG_NODE  node;
    OS_FLAGS      flags_cur;
    OS_FLAGS      flags_rdy;
    BOOLEAN       consume;


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_PEND_ISR;
        return ((OS_FLAGS)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                                  (2)
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
```

Listing 9.5 Waiting for event(s) of an event flag group. (Continued)

```
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {                          (3)
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    if (wait_type & OS_FLAG_CONSUME) {                                     (4)
        wait_type &= ~OS_FLAG_CONSUME;
        consume    = TRUE;
    } else {
        consume    = FALSE;
    }
    OS_ENTER_CRITICAL();
    switch (wait_type) {                                                   (5)
        case OS_FLAG_WAIT_SET_ALL:
             flags_rdy = pgrp->OSFlagFlags & flags;                        (6)
             if (flags_rdy == flags) {                                     (7)
                 if (consume == TRUE) {                                    (8)
                     pgrp->OSFlagFlags &= ~flags_rdy;                      (9)
                 }
                 flags_cur = pgrp->OSFlagFlags;                            (10)
                 OS_EXIT_CRITICAL();
                 *err      = OS_NO_ERR;
                 return (flags_cur);                                       (11)
             } else {                                                      (12)
                 OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                 OS_EXIT_CRITICAL();
             }
             break;

        case OS_FLAG_WAIT_SET_ANY:
             flags_rdy = pgrp->OSFlagFlags & flags;                        (13)
             if (flags_rdy != (OS_FLAGS)0) {                               (14)
                 if (consume == TRUE) {                                    (15)
                     pgrp->OSFlagFlags &= ~flags_rdy;                      (16)
                 }
                 flags_cur = pgrp->OSFlagFlags;                            (17)
                 OS_EXIT_CRITICAL();
                 *err      = OS_NO_ERR;
                 return (flags_cur);                                       (18)
             } else {                                                      (19)
                 OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                 OS_EXIT_CRITICAL();
             }
             break;
```

Listing 9.5 Waiting for event(s) of an event flag group. (Continued)

```
#if OS_FLAG_WAIT_CLR_EN > 0
        case OS_FLAG_WAIT_CLR_ALL:
             flags_rdy = ~pgrp->OSFlagFlags & flags;
             if (flags_rdy == flags) {
                 if (consume == TRUE) {
                     pgrp->OSFlagFlags |= flags_rdy;
                 }
                 flags_cur = pgrp->OSFlagFlags;
                 OS_EXIT_CRITICAL();
                 *err      = OS_NO_ERR;
                 return (flags_cur);
             } else {
                 OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                 OS_EXIT_CRITICAL();
             }
             break;

        case OS_FLAG_WAIT_CLR_ANY:
             flags_rdy = ~pgrp->OSFlagFlags & flags;
             if (flags_rdy != (OS_FLAGS)0) {
                 if (consume == TRUE) {
                     pgrp->OSFlagFlags |= flags_rdy;
                 }
                 flags_cur = pgrp->OSFlagFlags;
                 OS_EXIT_CRITICAL();
                 *err      = OS_NO_ERR;
                 return (flags_cur);
             } else {
                 OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                 OS_EXIT_CRITICAL();
             }
             break;
#endif

        default:
             OS_EXIT_CRITICAL();
             flags_cur = (OS_FLAGS)0;
             *err      = OS_FLAG_ERR_WAIT_TYPE;
             return (flags_cur);
    }
    OS_Sched();                                                          (20)
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStat & OS_STAT_FLAG) {                            (21)
```

Listing 9.5 Waiting for event(s) of an event flag group. (Continued)

```
        OS_FlagUnlink(&node);                                                    (22)
        OSTCBCur->OSTCBStat = OS_STAT_RDY;
        OS_EXIT_CRITICAL();
        flags_cur           = (OS_FLAGS)0;
        *err                = OS_TIMEOUT;
    } else {
        if (consume == TRUE) {                                                   (23)
            switch (wait_type) {
                case OS_FLAG_WAIT_SET_ALL:
                case OS_FLAG_WAIT_SET_ANY:                                       (24)
                     pgrp->OSFlagFlags &= ~OSTCBCur->OSTCBFlagsRdy;
                     break;

                case OS_FLAG_WAIT_CLR_ALL:
                case OS_FLAG_WAIT_CLR_ANY:
                     pgrp->OSFlagFlags |= OSTCBCur->OSTCBFlagsRdy;
                     break;
            }
        }
        flags_cur = pgrp->OSFlagFlags;                                           (25)
        OS_EXIT_CRITICAL();
        *err      = OS_NO_ERR;
    }
    return (flags_cur);
}
```

在中断服务程序中不允许调用系统函数 OSFlagPend()来等待事件标志组中的事件；

函数具有 4 个输入参数，具体如下：

pgrp 为指向事件标志组的指针；

flag 为任务等待的事件标志位，比如 flag 为 03H，则任务等待事件标志的 bit0 与 bit1 位；

wait_type 为等待类型，可以取值 OS_FLAG_WAIT_SET_ALL（等待所有指定的事件标志组中的事件标志位置 1）、OS_FLAG_WAIT_SET_ANY（等待任意一个指定的事件标志组中的事件标志位置 1）、OS_FLAG_WAIT_CLR_ALL（等待所有指定的事件标志组中的事件标志位置 0）、OS_FLAG_WAIT_CLR_ANY（等待任意一个指定的事件标志组中的事件标志位置 0）。wait_type 还可与一个参数 OS_FLAG_CONSUME 相或，以表征在任务等待的事件发生后，重新置起或者清除对应的事件标志位；

timeout 为等待事件标志组的超时时间；

err 为错误代码缓存区指针；

当 wait_type 为 OS_FLAG_WAIT_SET_ALL 或者 OS_FLAG_WAIT_SET_ALL+OS_FLAG_CONSUME 时，则等待所有指定的事件标志组中的事件标志位置 1。在代码序号 6 处取出事件标志组中由 flag 参数指定的事件标志位，存在 flag_rdy 中，如果恰好 flag_rdy 中的事件标志位符合 flag 参数的要求，说明任务等待的事件标志已经均置 1，如果 wait_type 为 OS_FLAG_WAIT_SET_ALL+OS_FLAG_CONSUME，即要

求清除对应的事件标志位，即序号 9 处的代码。函数返回事件标志组的新的事件标志状态值。

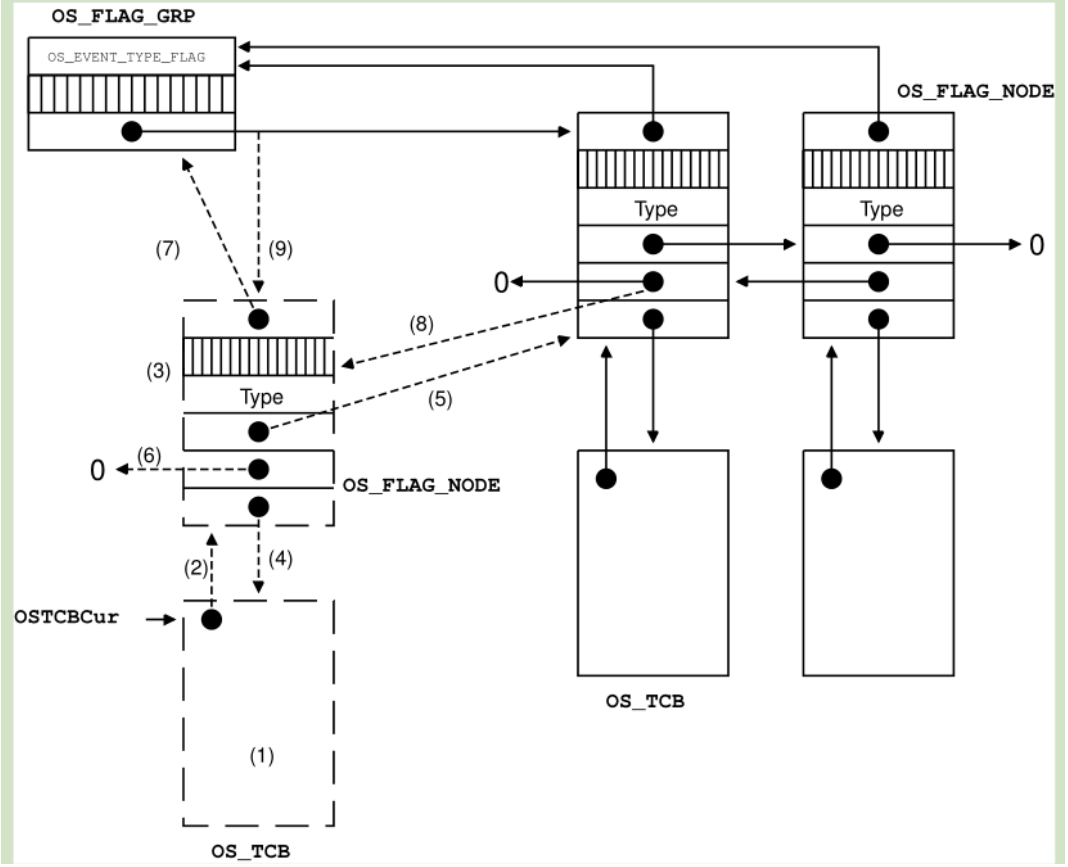
当 wait_type 为 OS_FLAG_WAIT_SET_ANY 或者 OS_FLAG_WAIT_SET_ANY +OS_FLAG_CONSUME 时，则等待任意一个指定的事件标志组中的事件标志位置 1。在代码序号 13 处取出事件标志组中由 flag 参数指定的事件标志位，存在 flag_rdy 中，如果 flag_rdy 不等于 0，则说明任务等待的事件标志组中至少有个事件标志位置 1，如果 wait_type 为 OS_FLAG_WAIT_SET_ANY+OS_FLAG_CONSUME，即要求清除对应的事件标志位，即序号 16 处代码。函数返回事件标志组的新的事件标志状态值。

如果指定的事件标志位没有置位，则调用 OSFlagPend()的任务将被挂起，直到指定的事件标志位置位，或者指定的等待超时时间到。让任务挂起由函数 OS_FlagBlock()完成。函数 OS_FlagBlock()代码如下：

Listing 9.6 Adding a task to the event flag group wait list.

```
static void OS_FlagBlock (OS_FLAG_GRP  *pgrp,
                          OS_FLAG_NODE *pnode,
                          OS_FLAGS      flags,
                          INT8U         wait_type,
                          INT16U        timeout)
{
    OS_FLAG_NODE  *pnode_next;

    OSTCBCur->OSTCBStat      |= OS_STAT_FLAG;                                  (1)
    OSTCBCur->OSTCBDly        = timeout;
#if OS_TASK_DEL_EN > 0
    OSTCBCur->OSTCBFlagNode   = pnode;                                         (2)
#endif
    pnode->OSFlagNodeFlags    = flags;                                         (3)
    pnode->OSFlagNodeWaitType = wait_type;
    pnode->OSFlagNodeTCB      = (void *)OSTCBCur;                              (4)
    pnode->OSFlagNodeNext     = pgrp->OSFlagWaitList;                          (5)
    pnode->OSFlagNodePrev     = (void *)0;                                     (6)
    pnode->OSFlagNodeFlagGrp  = (void *)pgrp;                                  (7)
    pnode_next                = pgrp->OSFlagWaitList;
    if (pnode_next != (void *)0) {
        pnode_next->OSFlagNodePrev = pnode;                                    (8)
    }
    pgrp->OSFlagWaitList = (void *)pnode;                                      (9)
                                                                              (10)
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
}
```

Figure 9.4 Adding the current task to the wait list of the event flag group.



函数 OS_FlagBlock()返回时，进行任务调度，因为调用该函数的任务由于等待的事件标志没有发生而不能继续运行。

当任务恢复时，首先检查任务恢复是由于等待的事件标志有效，还是由于等待超时。当由于等待超时使得任务恢复，调用 OS_FlagUnlink()函数，将 OS_FLAG_NODE 从事件标志组的等待任务链表中删除。

当由于等待的事件标志有效使得当前任务恢复，根据标志对事件标志进行相应的置 0 或者置 1 操作后，函数返回当前的事件标志状态。

9.04 置位或清除事件标志组的事件标志位

内核提供系统函数 OSFlagPost()来置位或者清除事件标志组中的事件标志位，具体代码如下：

Listing 9.7 Setting or clearing bits (i.e., events) in an event flag group.

```
OS_FLAGS  OSFlagPost (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR     cpu_sr;
#endif
    OS_FLAG_NODE *pnode;
    BOOLEAN       sched;
    OS_FLAGS      flags_cur;
    OS_FLAGS      flags_rdy;


#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                                          (1)
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {                            (2)
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    OS_ENTER_CRITICAL();
    switch (opt) {                                                           (3)
        case OS_FLAG_CLR:
             pgrp->OSFlagFlags &= ~flags;                                    (4)
             break;

        case OS_FLAG_SET:
             pgrp->OSFlagFlags |=  flags;                                    (5)
             break;
```

Listing 9.7 Setting or clearing bits (i.e., events) in an event flag group. (Continued)

```
        default:
             OS_EXIT_CRITICAL();
             *err = OS_FLAG_INVALID_OPT;
             return ((OS_FLAGS)0);
    }
    sched = FALSE;                                                          (6)
    pnode = pgrp->OSFlagWaitList;
    while (pnode != (OS_FLAG_NODE *)0) {                                    (7)
        switch (pnode->OSFlagNodeWaitType) {
            case OS_FLAG_WAIT_SET_ALL:                                      (8)
                 flags_rdy = pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                 if (flags_rdy == pnode->OSFlagNodeFlags) {                 (9)
                     if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {       (10)
                         sched = TRUE;                                     (11)
                     }
                 }
                 break;

            case OS_FLAG_WAIT_SET_ANY:
                 flags_rdy = pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                 if (flags_rdy != (OS_FLAGS)0) {
                     if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
                         sched = TRUE;
                     }
                 }
                 break;

#if OS_FLAG_WAIT_CLR_EN > 0
            case OS_FLAG_WAIT_CLR_ALL:
                 flags_rdy = ~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                 if (flags_rdy == pnode->OSFlagNodeFlags) {
                     if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
                         sched = TRUE;
                     }
                 }
                 break;
```

Listing 9.7 Setting or clearing bits (i.e., events) in an event flag group. (Continued)

```
            case OS_FLAG_WAIT_CLR_ANY:
                 flags_rdy = ~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
                 if (flags_rdy != (OS_FLAGS)0) {
                     if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
                         sched = TRUE;
                     }
                 }
                 break;
#endif
        }
        pnode = pnode->OSFlagNodeNext;                                  (12)
    }
    OS_EXIT_CRITICAL();
    if (sched == TRUE) {                                                (13)
        OS_Sched();                                                     (14)
    }
    OS_ENTER_CRITICAL();
    flags_cur = pgrp->OSFlagFlags;                                      (15)
    OS_EXIT_CRITICAL();
    *err      = OS_NO_ERR;
    return (flags_cur);                                                 (16)
}
```

系统函数 OSFlagPost()的输入参数 opt 决定了任务对事件标志组的标志位是置 1 操作还是置 0 操作，当 opt 为 OS_FLAG_SET 时，根据输入参数 flag 对相应的事件标志位置 1 操作，当 opt 为 OS_FLAG_CLR 时，根据输入参数 flag 对相应的事件标志位置 0 操作。

函数对局部布尔变量 sched 置为 FALSE，当由于对事件标志的操作使得一个更高的优先级任务就绪，则置局部布尔变量 sched 为 TRUE。当有任务在等待这个事件标志组，则遍历事件标志组的所有等待任务链表 OS_FLAG_NODE 进行处理，以检查新设定的事件标志是否满足某个任务所期待的条件。每个任务可能等待 OS_FLAG_WAIT_SET_ALL（等待所有指定的事件标志组中的事件标志位置 1）、OS_FLAG_WAIT_SET_ANY（等待任意一个指定的事件标志组中的事件标志位置 1）、OS_FLAG_WAIT_CLR_ALL（等待所有指定的事件标志组中的事件标志位置 0）、OS_FLAG_WAIT_CLR_ANY（等待任意一个指定的事件标志组中的事件标志位置 0）。

如果检查到某个任务等待的事件标志位得到满足，通过调用函数 OS_FlagTaskRdy()标志该任务进入就绪态。此时需要进行任务调度，但不是每检查一个 OS_FLAG_NODE 就进行一次任务调度，而是在遍历全部的等待任务后，进行一次总的调度，置局部布尔变量 sched 为 TRUE 以在后面进行任务调度。

遍历完成后，函数根据布尔变量 sched 决定是否进行任务调度，函数返回当前事件标志组

的事件标志状态。

另外，内核提供了函数 OS_FlagTaskRdy()使得等待事件标志组的任务进入就绪态，其代码如下：

Listing 9.8 Make a waiting task ready to run.

```
static  BOOLEAN  OS_FlagTaskRdy (OS_FLAG_NODE *pnode, OS_FLAGS flags_rdy)
{
    OS_TCB   *ptcb;
    BOOLEAN   sched;

    ptcb               = (OS_TCB *)pnode->OSFlagNodeTCB;
    ptcb->OSTCBDly     = 0;
    ptcb->OSTCBFlagsRdy = flags_rdy;
```

Listing 9.8 Make a waiting task ready to run. (Continued)

```
    ptcb->OSTCBStat    &= ~OS_STAT_FLAG;
    if (ptcb->OSTCBStat == OS_STAT_RDY) {                                    (1)
        OSRdyGrp               |= ptcb->OSTCBBitY;
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        sched                   = TRUE;                                      (2)
    } else {
        sched                   = FALSE;                                     (3)
    }
    OS_FlagUnlink(pnode);                                                    (4)
    return (sched);
}
```

当事件标志组中某个任务等待的事件标志发生后，为该任务建立的 OS_FLAG_NODE 数据结构就没有用，调用函数 OS_FlagUnlink()将 OS_FLAG_NODE 数据结构从等待任务链表中删除。并将 OS_FLAG_NODE 数据结构从该任务的事件控制块中删除。

函数 OS_FlagUnlink()的代码如下：

Listing 9.9 Unlinking an OS_FLAG_NODE.

```
void  OS_FlagUnlink (OS_FLAG_NODE *pnode)
{
#if OS_TASK_DEL_EN > 0
    OS_TCB       *ptcb;
#endif
    OS_FLAG_GRP  *pgrp;
    OS_FLAG_NODE *pnode_prev;
    OS_FLAG_NODE *pnode_next;



    pnode_prev = pnode->OSFlagNodePrev;                                  (1)
    pnode_next = pnode->OSFlagNodeNext;                                  (2)
    if (pnode_prev == (OS_FLAG_NODE *)0) {                               (3)
        pgrp                = pnode->OSFlagNodeFlagGrp;                  (4)
        pgrp->OSFlagWaitList = (void *)pnode_next;                       (5)
```

Listing 9.9 Unlinking an OS_FLAG_NODE. (Continued)

```
        if (pnode_next != (OS_FLAG_NODE *)0) {                           (6)
            pnode_next->OSFlagNodePrev = (OS_FLAG_NODE *)0;              (7)
        }
    } else {
        pnode_prev->OSFlagNodeNext = pnode_next;                         (8)
        if (pnode_next != (OS_FLAG_NODE *)0) {                           (9)
            pnode_next->OSFlagNodePrev = pnode_prev;                    (10)
        }
    }
#if OS_TASK_DEL_EN > 0
    ptcb                = (OS_TCB *)pnode->OSFlagNodeTCB;                (11)
    ptcb->OSTCBFlagNode = (void *)0;                                     (12)
#endif
}
```

9.05 无等待地获得事件标志组中的事件标志位

内核提供系统函数 OSFlagAccept()来无等待地获得事件标志组中的事件标志，如果等待的事件没有发生，则调用的任务并不挂起。系统函数 OSFlagAccept()可以在中断服务程序中调用。

如果等待的事件没有发生，则该函数并不会停止当前任务的继续运行，而是返回一个错误代码，调用函数判断返回的错误代码了解是否得到了期望的事件标志位。

系统函数 OSFlagAccept()的代码如下：

Listing 9.10 Looking for event flags without waiting.

```
OS_FLAGS  OSFlagAccept (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR     cpu_sr;
#endif
    OS_FLAGS      flags_cur;
    OS_FLAGS      flags_rdy;
    BOOLEAN       consume;


#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    if (wait_type & OS_FLAG_CONSUME) {
        wait_type &= ~OS_FLAG_CONSUME;
        consume    = TRUE;
    } else {
        consume    = FALSE;
    }

    OS_ENTER_CRITICAL();
    switch (wait_type) {
```

Listing 9.10 Looking for event flags without waiting. (Continued)

```
        case OS_FLAG_WAIT_SET_ALL:
             flags_rdy = pgrp->OSFlagFlags & flags;
             if (flags_rdy == flags) {
                 if (consume == TRUE) {
                     pgrp->OSFlagFlags &= ~flags_rdy;
                 }
                 flags_cur = pgrp->OSFlagFlags;
                 OS_EXIT_CRITICAL();
                 *err      = OS_NO_ERR;
             } else {
                 flags_cur = pgrp->OSFlagFlags;
                 OS_EXIT_CRITICAL();
                 *err      = OS_FLAG_ERR_NOT_RDY;
             }
             break;

        case OS_FLAG_WAIT_SET_ANY:
             flags_rdy = pgrp->OSFlagFlags & flags;
             if (flags_rdy != (OS_FLAGS)0) {
                 if (consume == TRUE) {
                     pgrp->OSFlagFlags &= ~flags_rdy;
                 }
                 flags_cur = pgrp->OSFlagFlags;
                 OS_EXIT_CRITICAL();
                 *err      = OS_NO_ERR;
             } else {
                 flags_cur = pgrp->OSFlagFlags;
                 OS_EXIT_CRITICAL();
                 *err      = OS_FLAG_ERR_NOT_RDY;
             }
             break;

#if OS_FLAG_WAIT_CLR_EN > 0
        case OS_FLAG_WAIT_CLR_ALL:
             flags_rdy = ~pgrp->OSFlagFlags & flags;
             if (flags_rdy == flags) {
                 if (consume == TRUE) {
                     pgrp->OSFlagFlags |= flags_rdy;
                 }
```

Listing 9.10 Looking for event flags without waiting. (Continued)

```
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
            } else {
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_FLAG_ERR_NOT_RDY;
            }
            break;

        case OS_FLAG_WAIT_CLR_ANY:
            flags_rdy = ~pgrp->OSFlagFlags & flags;
            if (flags_rdy != (OS_FLAGS)0) {
                if (consume == TRUE) {
                    pgrp->OSFlagFlags |= flags_rdy;
                }
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_NO_ERR;
            } else {
                flags_cur = pgrp->OSFlagFlags;
                OS_EXIT_CRITICAL();
                *err      = OS_FLAG_ERR_NOT_RDY;
            }
            break;
#endif

        default:
            OS_EXIT_CRITICAL();
            flags_cur = (OS_FLAGS)0;
            *err      = OS_FLAG_ERR_WAIT_TYPE;
            break;
    }
    return (flags_cur);
}
```

9.06 查询事件标志组的状态

内核提供系统函数 OSFlagQuery()得到当前事件标志组的事件标志状态。函数代码如下：

Listing 9.11 Obtaining the current flags of an event flag group.

```
OS_FLAGS  OSFlagQuery (OS_FLAG_GRP *pgrp, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_FLAGS   flags;


#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {                                    (1)
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
    if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {                      (2)
        *err = OS_ERR_EVENT_TYPE;
        return ((OS_FLAGS)0);
    }
#endif
    OS_ENTER_CRITICAL();
    flags = pgrp->OSFlagFlags;                                         (3)
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (flags);                                                    (4)
}
```

10 消息邮箱管理

UCOSII 可以通过消息邮箱，使得一个任务或者中断服务程序向另一个任务发送指针型的变量，该指针包含特点的消息内容。

内核提供系统函数 OSMboxCreate()、OSMboxDel()、OSMboxPend()、OSMboxPost()、OSMboxPostOpt()、OSMboxAccept()、OSMboxQuery()对消息邮箱进行操作。

消息邮箱包含的内容是一个指向一条消息的指针，指针指向的内容可以给定。

10.00 建立一个消息邮箱

内核提供系统函数 OSMboxCreate()来创建一个消息邮箱，消息邮箱包含的指针可以为 NULL，或者为一个特定的消息。函数的代码如下：

Listing 10.1 Creating a mailbox.

```
OS_EVENT  *OSMboxCreate (void *msg)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;                                          (1)
#endif
    OS_EVENT  *pevent;


    if (OSIntNesting > 0) {                                     (2)
        return ((OS_EVENT *)0);
    }
```

Listing 10.1 Creating a mailbox. (Continued)

```
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                   (3)
    if (OSEventFreeList != (OS_EVENT *)0) {                     (4)
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;  (5)
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                              (6)
        pevent->OSEventType = OS_EVENT_TYPE_MBOX;               (7)
        pevent->OSEventCnt  = 0;                                (8)
        pevent->OSEventPtr  = msg;                              (9)
        OS_EventWaitListInit(pevent);                           (10)
    }
    return (pevent);                                            (11)
}
```

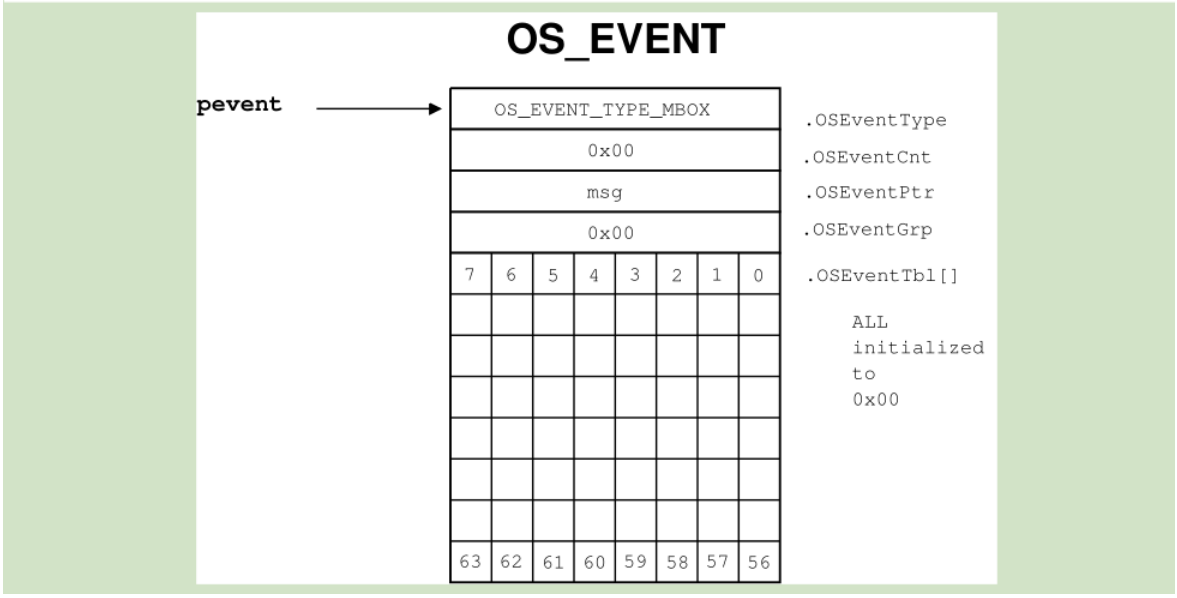
在中断服务程序中不允许建立消息邮箱；

系统函数 OSMboxCreat()从空闲的事件控制块链表中取得一个事件控制块 ECB，设置该事件控制块的参数为消息邮箱对应的参数值，调用函数 OS_EventWaitListInit()对 OSEventGrp

与 OSEventTbl[]初始化清 0 操作，创建时没有任务等待该消息邮箱。函数返回指向事件控制块 ECB 的指针。

函数返回前的事件控制块初始化如下：

Figure 10.2 ECB just before OSMboxCreate() returns.



10.01 删除一个消息邮箱

内核提供系统函数 OSMboxDel()删除一个消息邮箱。在删除消息邮箱之前，必须首先删除可能操作该消息邮箱的所有任务。函数的代码如下：

Listing 10.2 Deleting a mailbox.

```
OS_EVENT  *OSMboxDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    tasks_waiting;


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_DEL_ISR;
        return (pevent);
    }
```

Listing 10.2 Deleting a mailbox. (Continued)

```
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                    (2)
        *err = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                  (3)
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                 (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:
             if (tasks_waiting == FALSE) {
                 pevent->OSEventType = OS_EVENT_TYPE_UNUSED;          (5)
                 pevent->OSEventPtr  = OSEventFreeList;               (6)
                 OSEventFreeList     = pevent;                        (7)
                 OS_EXIT_CRITICAL();
                 *err = OS_NO_ERR;
                 return ((OS_EVENT *)0);                              (8)
             } else {
                 OS_EXIT_CRITICAL();
                 *err = OS_ERR_TASK_WAITING;
                 return (pevent);
             }

        case OS_DEL_ALWAYS:
             while (pevent->OSEventGrp != 0x00) {                     (9)
                 OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MBOX);   (10)
             }
             pevent->OSEventType = OS_EVENT_TYPE_UNUSED;             (11)
             pevent->OSEventPtr  = OSEventFreeList;                  (12)
             OSEventFreeList     = pevent;
             OS_EXIT_CRITICAL();
             if (tasks_waiting == TRUE) {
```

Listing 10.2 Deleting a mailbox. (Continued)

```
                 OS_Sched();                                         (13)
             }
             *err = OS_NO_ERR;
             return ((OS_EVENT *)0);                                 (14)

        default:
             OS_EXIT_CRITICAL();
             *err = OS_ERR_INVALID_OPT;
             return (pevent);
    }
}
```

在中断服务程序中不允许进行删除消息邮箱的操作；

系统函数 OSMboxDel()可以当没有任务等待消息邮箱时再删除（opt 为 OS_DEL_NO_PEND），或者尽管有任务等待也删除消息邮箱（opt 为 OS_DEL_ALWAYS）。

代码其他不详细论述。

10.02 等待消息邮箱中的消息

内核提供系统函数 OSMboxPend()来等待某一个消息邮箱中的消息。函数代码如下：

Listing 10.3 Waiting for a message at a mailbox (blocking), OSMboxPend( ).

```
void  *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    void      *msg;


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_PEND_ISR;
        return ((void *)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                   (2)
        *err = OS_ERR_PEVENT_NULL;
        return ((void *)0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                 (3)
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
#endif
```

Listing 10.3 Waiting for a message at a mailbox (blocking), OSMboxPend( ). (Continued)

```
    OS_ENTER_CRITICAL();
    msg = pevent->OSEventPtr;                                    (4)
    if (msg != (void *)0) {
        pevent->OSEventPtr = (void *)0;                          (5)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (msg);                                            (6)
    }
    OSTCBCur->OSTCBStat |= OS_STAT_MBOX;                         (7)
    OSTCBCur->OSTCBDly   = timeout;                              (8)
    OS_EventTaskWait(pevent);                                    (9)
    OS_EXIT_CRITICAL();
    OS_Sched();                                                 (10)
    OS_ENTER_CRITICAL();
    msg = OSTCBCur->OSTCBMsg;
    if (msg != (void *)0) {                                     (11)
        OSTCBCur->OSTCBMsg      = (void *)0;
        OSTCBCur->OSTCBStat     = OS_STAT_RDY;
        OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
        OS_EXIT_CRITICAL();
        *err                    = OS_NO_ERR;
        return (msg);                                           (12)
    }
    OS_EventTO(pevent);                                         (13)
    OS_EXIT_CRITICAL();
    *err = OS_TIMEOUT;
    return ((void *)0);                                         (14)
}
```

如果邮箱中有消息，即消息指针为非 NULL 指针，则从邮箱中取出消息，将 NULL 指针存入邮箱中，函数返回该消息指针。

如果邮箱中没有消息，即消息指针为 NULL 指针，则调用 OSMboxPend()的任务进入休眠态，等待其他任务或者中断服务程序通过该邮箱发送消息。系统函数 OSMboxPend()允许定义最长的超时时间。使得当前任务进入休眠态是通过调用函数 OS_EventTaskWait()实现。因为当前任务已经休眠，则进行任务调度。

当 OS_Sched()函数返回后，函数 OSMboxPend()需要检查消息是否已经存入任务控制块 TCB 中，如果消息非 NULL，则函数返回该消息指针，如果消息为 NULL，则超时时间使得当前任务恢复，OS_EventTO()函数将当前任务从消息邮箱的任务等待列表中删除。

10.03 向消息邮箱发送一则消息 OSMboxPost()

内核提供系统函数 OSMboxPost()来向消息邮箱发送一则消息。函数代码如下：

```
INT8U  OSMboxPost (OS_EVENT *pevent, void *msg)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                    (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (msg == (void *)0) {
        return (OS_ERR_POST_NULL_PTR);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                 (2)
        OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);                   (3)
        OS_EXIT_CRITICAL();
        OS_Sched();                                                   (4)
        return (OS_NO_ERR);
    }
    if (pevent->OSEventPtr != (void *)0) {                            (5)
        OS_EXIT_CRITICAL();
        return (OS_MBOX_FULL);
    }
    pevent->OSEventPtr = msg;                                         (6)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

函数首先检查是否有任务在等待该消息邮箱，如果有，则调用函数 OS_EventTaskRdy()，将其中的最高优先级任务从消息邮箱等待列表中删除，并加入到系统的就绪任务列表中，同时将消息存入任务控制块 TCB 中。随后进行任务切换。如果等待的任务为最高优先级任务，则执行任务切换，如果等待的任务不是最高优先级的任务，则当前任务继续运行。

当没有任务在等待该消息邮箱时，即代码序号 5 处，函数检查邮箱中是否有消息，由于邮箱里面只能存放一条消息，则如果当前邮箱中有消息，则返回邮箱已满的错误代码，如果当前邮箱中没有消息，则消息将存放到邮箱中。

10.04 向消息邮箱发送一则消息 OSMboxPostOpt()

内核提供系统函数 OSMboxPostOpt()，实现功能更强的向消息邮箱发送消息的操作。OSMboxPostOpt()可以向等待消息邮箱的所有任务发送消息，即实现“广播”的概念。函数的

代码如下：

Listing 10.5 Posting a message to a mailbox, OSMboxPostOpt().

```
INT8U  OSMboxPostOpt (OS_EVENT *pevent, void *msg, INT8U opt)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif

#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                  (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (msg == (void *)0) {
        return (OS_ERR_POST_NULL_PTR);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                               (2)
        if ((opt & OS_POST_OPT_BROADCAST) != 0x00) {                (3)
            while (pevent->OSEventGrp != 0x00) {                    (4)
                OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);         (5)
            }
        } else {
            OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);             (6)
        }
        OS_EXIT_CRITICAL();
        OS_Sched();                                                 (7)
        return (OS_NO_ERR);
    }
    if (pevent->OSEventPtr != (void *)0) {                          (8)
        OS_EXIT_CRITICAL();
        return (OS_MBOX_FULL);
    }
    pevent->OSEventPtr = msg;                                       (9)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

函数的输入参数 opt 是控制选项，如果 opt 为 OS_POST_OPT_BROADCAST，则所有等待任务都得到该信息。函数 OS_EventTaskRdy()将所有等待任务从等待列表中删除。如果没有广播的控制选项，则只有最高优先级的任务进入就绪态。

10.05 无等待地从消息邮箱得到一则消息

内核提供系统函数 OSMboxAccept()，使得任务从消息邮箱中得到消息，而不必使得任务

进入休眠态。函数的代码如下：

Listing 10.6 Getting a message without waiting.

```
void  *OSMboxAccept (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    void      *msg;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                  (1)
        return ((void *)0);
```

Listing 10.6 Getting a message without waiting. (Continued)

```
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                (2)
        return ((void *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    msg                = pevent->OSEventPtr;                        (3)
    pevent->OSEventPtr = (void *)0;                                 (4)
    OS_EXIT_CRITICAL();
    return (msg);                                                   (5)
}
```

OSMboxAccept()的调用函数必须检查函数的返回值，如果为 NULL，则说明邮箱是空的，如果为非 NULL，则说明邮箱中有消息可用。

OSMboxAccept()的另一个用途为清空邮箱中的现有内容。

10.06 查询一个消息邮箱的状态

内核提供系统函数 OSMboxQuery()来查询一个消息邮箱的状态。函数的代码如下：

Listing 10.7 Obtaining the status of a mailbox.

```
INT8U  OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U     *psrc;
    INT8U     *pdest;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                          (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                        (2)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    pdata->OSEventGrp = pevent->OSEventGrp;                                 (3)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];

#if OS_EVENT_TBL_SIZE > 0
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 1
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *pdest++          = *psrc++;
#endif
```

Listing 10.7 Obtaining the status of a mailbox. (Continued)

```
#if OS_EVENT_TBL_SIZE > 4
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *pdest            = *psrc;
#endif
    pdata->OSMsg = pevent->OSEventPtr;                                    (4)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

10.07 用消息邮箱作为二值信号量

消息邮箱可以作为二值信号量使用，初始化时将消息邮箱中设置一个非 NULL 的消息指针，当需要释放资源时，向消息邮箱中发送一个非 NULL 的消息指针。范例如下：

Listing 10.8 Using a mailbox as a binary semaphore.

```
OS_EVENT *MboxSem;


void Task1 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSMboxPend(MboxSem, 0, &err);   /* Obtain access to resource(s)  */
        .
        .    /* Task has semaphore, access resource(s)                   */
        .
        OSMboxPost(MboxSem, (void *)1); /* Release access to resource(s) */
    }
}
```

10.08 用消息邮箱实现延时

消息邮箱可以用来模仿 OSTimeDly()函数的功能。范例如下：

Listing 10.9 Using a mailbox as a time delay.

```
OS_EVENT *MboxTimeDly;


void Task1 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSMboxPend(MboxTimeDly, TIMEOUT, &err);   /* Delay task              */
        .
        .    /* Code executed after time delay or dummy message is received  */
        .
    }
}


void Task2 (void *pdata)
{
    INT8U err;


    for (;;) {
        OSMboxPost(MboxTimeDly, (void *)1);       /* Cancel delay for Task1  */
        .
        .
    }
}
```

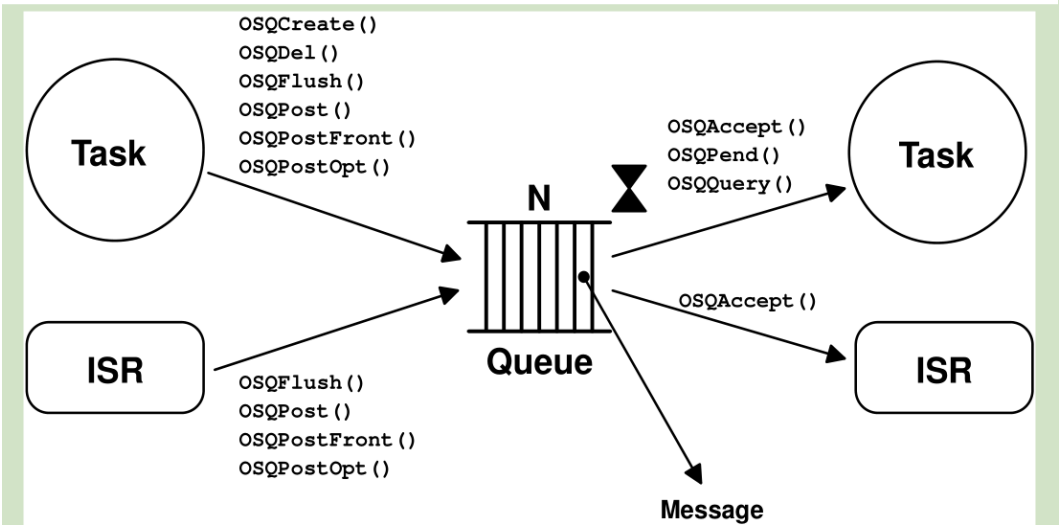
11 消息队列管理

UCOSII 提供消息队列的通信方式，允许一个任务或者中断服务程序向另一个任务发送以指针方式定义的变量或者其他任务。指针所指向的数据的类型也可以不相同。

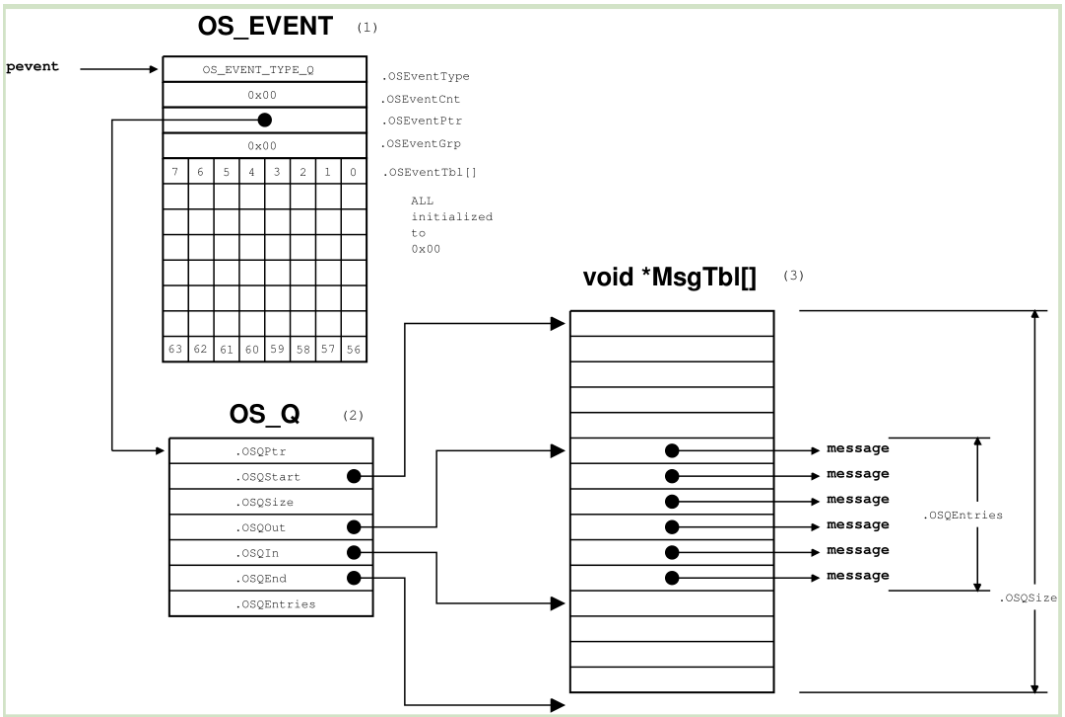
内核提供了 9 个系统函数，实现消息队列管理，分别为 OSQCreate()、OSQDel()、OSQPend()、OSQPost()、OSQPostFront()、OSQPostOpt()、OSQAccept()、OSQFlush()、OSQQuery()。

可以将消息队列看成是多个邮箱组成的邮箱数组，这多个邮箱共用一个等待任务列表。

Figure 11.1 Relationships between tasks, ISRs, and a message queue.



用于消息队列管理的数据结构如下：

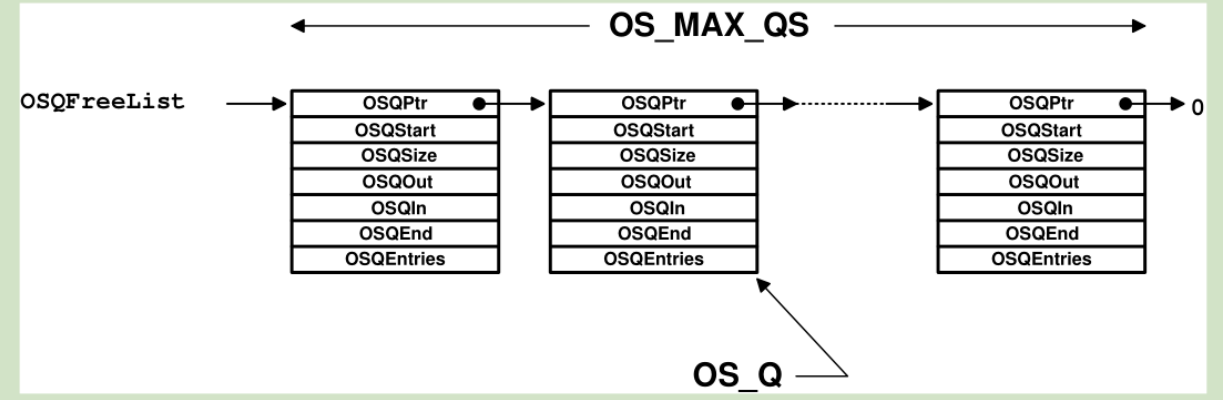


内核在管理消息队列时，使用事件控制块 ECB 记录等待任务列表，同时，建立一个队列控制块数据结构 OS_Q，事件控制块 ECB 的 OSEventPtr 指向该队列控制块数据结构 OS_Q。

消息队列管理的指针是一个与消息队列最大消息数相同的指针数组，该数组的起始地址，数组的大小，数组元素的数量等信息作为输入参数传递给 OSQCreat()。

内核设置空闲消息队列控制块链表，链表个数为 OS_MAX_QS。如下：

Figure 11.3 List of free queue control blocks.



在内核代码中，消息队列控制块数据结构如下：

```
/*
*********************************************************************************************************
*                                          MESSAGE QUEUE DATA
*********************************************************************************************************
*/

#if OS_Q_EN > 0u
typedef struct os_q {                   /* QUEUE CONTROL BLOCK                                      */
    struct os_q   *OSQPtr;              /* Link to next queue control block in list of free blocks  */
    void         **OSQStart;            /* Pointer to start of queue data                           */
    void         **OSQEnd;              /* Pointer to end   of queue data                           */
    void         **OSQIn;               /* Pointer to where next message will be inserted  in   the Q  */
    void         **OSQOut;              /* Pointer to where next message will be extracted from the Q  */
    INT16U         OSQSize;             /* Size of queue (maximum number of entries)                */
    INT16U         OSQEntries;          /* Current number of entries in the queue                   */
} OS_Q;
```

数据结构参数 OSQPtr 用来在空闲消息队列控制块链表中链接所有的队列控制块，当建立消息队列后，该参数不再使用；

数据结构参数 OSQStart 指向消息队列的指针数组的起始地址的指针。该参数为指向指针的指针；

数据结构参数 OSQEnd 指向消息队列的指针数组结束的下一个地址；

数据结构参数 OSQIn 指向消息队列中插入下一条消息的位置的指针，当 OSQIn 等于 OSQEnd 时，说明该消息队列消息填满一个循环，则 OSQIn 被调整到消息队列的起始单元；

数据结构参数 OSQOut 指向消息队列中下一个取出消息的位置的指针。当 OSQOut 等于 OSQEnd 时，说明该消息队列消息已经取了一个循环，则 OSQOut 被调整到消息队列的起始

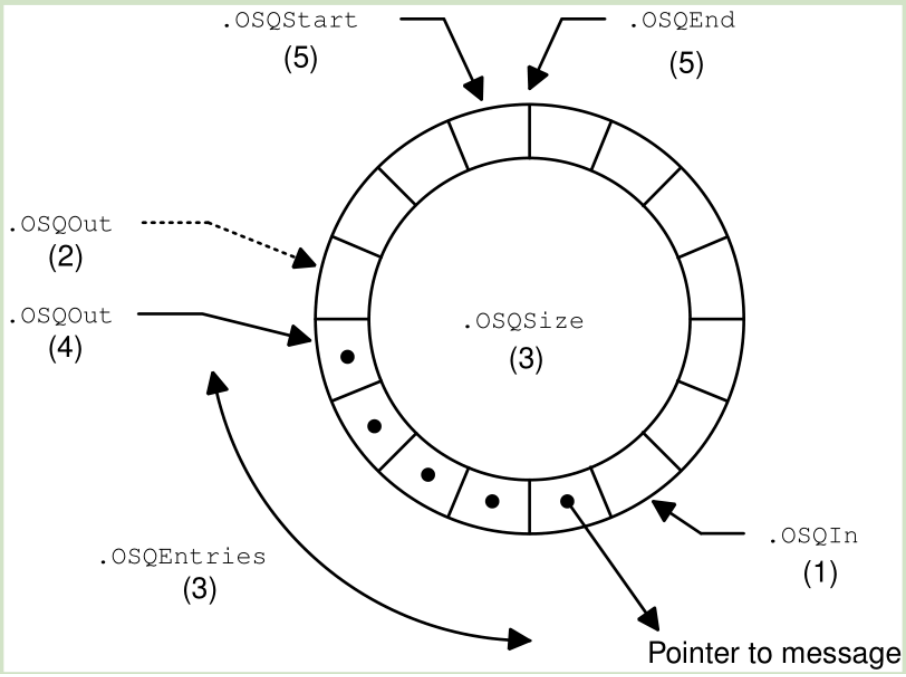
单元；

数据结构参数 OSQSize 表示消息队列中可以容纳的总的消息数量；

数据结构参数 OSQEntries 表示消息队列中当前的消息数量。当消息队列为空时，该参数为 0，当消息队列填满时，该参数等于 OSQSize。

消息队列是由指针组成的循环缓存区，示意图如下：

Figure 11.4 A message queue as a circular buffer of pointers.



11.00 建立一个消息队列

内核提供系统函数 OSQCreate()来建立一个消息队列，该参数的输入参数包括 2 个，分别为指向消息数组的指针和数组的大小，指针数组的类型必须为 void 类型，比如声明如下：

```
void *MyArrayOfMsg[SIZE];
```

需要将指针数组 MyArrayofMsg 的地址及数组大小 SIZE 传递给系统函数 OSQCreat()。消息队列被初始化为空，不含任何消息数据。代码如下：

Listing 11.1 Creating a message queue.

```
OS_EVENT  *OSQCreate (void **start, INT16U size)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;                                                  (1)
#endif
    OS_EVENT  *pevent;
    OS_Q      *pq;


    if (OSIntNesting > 0) {                                             (2)
        return ((OS_EVENT *)0);
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                           (3)
    if (OSEventFreeList != (OS_EVENT *)0) {
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                                      (4)
        OS_ENTER_CRITICAL();
        pq = OSQFreeList;
        if (pq != (OS_Q *)0) {
            OSQFreeList         = OSQFreeList->OSQPtr;
            OS_EXIT_CRITICAL();
            pq->OSQStart        = start;                                (5)
            pq->OSQEnd          = &start[size];
            pq->OSQIn           = start;
            pq->OSQOut          = start;
            pq->OSQSize         = size;
            pq->OSQEntries      = 0;
            pevent->OSEventType = OS_EVENT_TYPE_Q;                      (6)
            pevent->OSEventCnt  = 0;
            pevent->OSEventPtr  = pq;
            OS_EventWaitListInit(pevent);                               (7)
        } else {
            pevent->OSEventPtr = (void *)OSEventFreeList;               (8)
            OSEventFreeList    = pevent;
            OS_EXIT_CRITICAL();
```

Listing 11.1 Creating a message queue. (Continued)

```
            pevent = (OS_EVENT *)0;
        }
    }
    return (pevent);                                                    (9)
}
```

内核不允许在中断服务程序中调用系统函数 OSQCreat()；

系统函数 OSQCreat()试图从空闲事件控制块 ECB 链表中取得一个事件控制块 ECB，如果事件控制块 ECB 可用，则再试图从空闲消息队列控制块链表中取得一个消息队列控制块 OS_Q；

函数返回指向事件控制块 ECB 的指针。

11.01 删除一个消息队列

内核提供系统函数 OSQDel()来删除一个消息队列。多任务可能使用需要删除的消息队列，因此一般在删除消息队列之前，首先应该删除所有用到该消息队列的任务。

系统函数 OSQDel()的代码如下：

Listing 11.2 Deleting a message queue.

```
OS_EVENT  *OSQDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
```

Listing 11.2 Deleting a message queue. (Continued)

```
#endif
    BOOLEAN    tasks_waiting;
    OS_Q      *pq;


    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_DEL_ISR;
        return ((OS_EVENT *)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                   (2)
        *err = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (3)
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
        case OS_DEL_NO_PEND:
             if (tasks_waiting == FALSE) {
                 pq                  = pevent->OSEventPtr;           (5)
                 pq->OSQPtr          = OSQFreeList;
                 OSQFreeList         = pq;
                 pevent->OSEventType = OS_EVENT_TYPE_UNUSED;         (6)
                 pevent->OSEventPtr  = OSEventFreeList;              (7)
                 OSEventFreeList     = pevent;
                 OS_EXIT_CRITICAL();
                 *err = OS_NO_ERR;
                 return ((OS_EVENT *)0);                             (8)
```

Listing 11.2 Deleting a message queue. (Continued)

```
            } else {
                OS_EXIT_CRITICAL();
                *err = OS_ERR_TASK_WAITING;
                return (pevent);
            }

        case OS_DEL_ALWAYS:
            while (pevent->OSEventGrp != 0x00) {                          (9)
                OS_EventTaskRdy(pevent, (void *)0, OS_STAT_Q);            (10)
            }
            pq                  = pevent->OSEventPtr;                     (11)
            pq->OSQPtr          = OSQFreeList;
            OSQFreeList         = pq;
            pevent->OSEventType = OS_EVENT_TYPE_UNUSED;                   (12)
            pevent->OSEventPtr  = OSEventFreeList;                        (13)
            OSEventFreeList     = pevent;
            OS_EXIT_CRITICAL();
            if (tasks_waiting == TRUE) {
                OS_Sched();                                               (14)
            }
            *err = OS_NO_ERR;
            return ((OS_EVENT *)0);                                       (15)

        default:
            OS_EXIT_CRITICAL();
            *err = OS_ERR_INVALID_OPT;
            return (pevent);
    }
}
```

系统函数的代码不再详细分析，与删除信号量、删除消息邮箱、删除互斥型信号量等代码类似。

11.02 等待消息队列中的消息

内核提供系统函数 OSQPend()来等待消息队列中的消息，具体代码如下：

Listing 11.3 Waiting for a message to arrive at a queue.

```
void  *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    void      *msg;
    OS_Q      *pq;
```

Listing 11.3 Waiting for a message to arrive at a queue. (Continued)

```
    if (OSIntNesting > 0) {                                          (1)
        *err = OS_ERR_PEND_ISR;
        return ((void *)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                   (2)
        *err = OS_ERR_PEVENT_NULL;
        return ((void *)0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (3)
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {                                        (4)
        msg = *pq->OSQOut++;                                         (5)
        pq->OSQEntries--;                                            (6)
        if (pq->OSQOut == pq->OSQEnd) {                              (7)
            pq->OSQOut = pq->OSQStart;                               (8)
        }
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (msg);                                                (9)
    }
    OSTCBCur->OSTCBStat |= OS_STAT_Q;                                (10)
    OSTCBCur->OSTCBDly   = timeout;                                  (11)
    OS_EventTaskWait(pevent);                                        (12)
    OS_EXIT_CRITICAL();
    OS_Sched();                                                      (13)
    OS_ENTER_CRITICAL();
    msg = OSTCBCur->OSTCBMsg;                                        (14)
```

Listing 11.3 Waiting for a message to arrive at a queue. (Continued)

```
    if (msg != (void *)0) {
        OSTCBCur->OSTCBMsg      = (void *)0;                         (15)
        OSTCBCur->OSTCBStat     = OS_STAT_RDY;
        OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
        OS_EXIT_CRITICAL();
        *err                    = OS_NO_ERR;
        return (msg);
    }
    OS_EventTO(pevent);                                              (16)
    OS_EXIT_CRITICAL();
    *err = OS_TIMEOUT;
    return ((void *)0);                                              (17)
}
```

系统函数OSQPend()判断消息队列控制块数据结构OS_Q的参数OSQEntries是否大于0，如果是则说明当前消息队列中有消息可用，OSQPend()将 OSQOut 指向的消息指针复制到 msg 变量中，并让 OSQOut 指向消息队列中的下一个单元。然后检查 OSQOut 是否超出了消息队

列的最后一个单元，当发生越界，则 OSQOut 需要调整到消息队列的起始单元。函数返回指向消息的指针变量 msg。

当消息队列中消息为空，则调用本函数的任务将被挂起，直到消息队列中收到新的消息，OSQPend()允许定义等待超时时间。

当 OS_Sched()返回后，系统函数 OSQPend()会检查 OSQPost()是否已经将消息放在任务控制块 TCB 中，即代码序号 14、15 处。

11.03 向消息队列发送一则消息（FIFO）

向消息队列发送一则消息，可以采用 FIFO 形式，也可以采用 LIFO 形式，如果在 OSQIn 指向的单元插入新的指向消息的指针，则采用的为 FIFO 形式，而如果在 OSQOut 指向的单元的下一个单元插入新的指向消息的指针，则采用的为 LIFO 形式。

内核提供系统函数 OSQPost()，实现以 FIFO 形式向消息队列发送一则消息。系统函数 OSQPost()的代码如下：

Listing 11.4 Depositing a message in a queue (FIFO), OSQPost().

```
INT8U  OSQPost (OS_EVENT *pevent, void *msg)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_Q      *pq;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                    (1)
        return (OS_ERR_PEVENT_NULL);
    }
```

Listing 11.4 Depositing a message in a queue (FIFO), OSQPost(). (Continued)

```
    if (msg == (void *)0) {                                          (2)
        return (OS_ERR_POST_NULL_PTR);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (3)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                                (4)
        OS_EventTaskRdy(pevent, msg, OS_STAT_Q);                     (5)
        OS_EXIT_CRITICAL();
        OS_Sched();                                                  (6)
        return (OS_NO_ERR);
    }
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries >= pq->OSQSize) {                             (7)
        OS_EXIT_CRITICAL();
        return (OS_Q_FULL);
    }
    *pq->OSQIn++ = msg;                                              (8)
    pq->OSQEntries++;                                                (9)
    if (pq->OSQIn == pq->OSQEnd) {                                   (10)
        pq->OSQIn = pq->OSQStart;
    }
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

系统函数 OSQPost()在 OSQIn 指向的单元插入新的指向消息的指针，则采用的为 FIFO 形式，函数首先检查是否有任务在等待该消息队列中的消息，如果有，则调用 OSEventTaskRdy 函数，从等待列表中取出最高优先级的任务，并置于就绪态，并把消息指针 msg 存入任务控制块 TCB 的 OSTCBMsg 变量中。随后进行任务调度。

如果没有任务在等待该消息队列中的消息，且消息队列未满，则在 OSQIn 指向的单元插入新的指向消息的指针 msg。

11.04 向消息队列发送一则消息（LIFO）

向消息队列发送一则消息，可以采用 FIFO 形式，也可以采用 LIFO 形式，如果在 OSQIn 指向的单元插入新的指向消息的指针，则采用的为 FIFO 形式，而如果在 OSQOut 指向的单元的下一个单元插入新的指向消息的指针，则采用的为 LIFO 形式。

内核提供系统函数 OSQPostFront()，实现以 FIFO 形式向消息队列发送一则消息。系统函

数 OSQPostFront()的代码如下：

Listing 11.5 Depositing a message in a queue (LIFO), OSQPostFront().

```
INT8U  OSQPostFront (OS_EVENT *pevent, void *msg)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_Q      *pq;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {
        return (OS_ERR_PEVENT_NULL);
    }
    if (msg == (void *)0) {
        return (OS_ERR_POST_NULL_PTR);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
        return (OS_ERR_EVENT_TYPE);
```

Listing 11.5 Depositing a message in a queue (LIFO), OSQPostFront(). (Continued)

```
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {
        OS_EventTaskRdy(pevent, msg, OS_STAT_Q);
        OS_EXIT_CRITICAL();
        OS_Sched();
        return (OS_NO_ERR);
    }
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries >= pq->OSQSize) {
        OS_EXIT_CRITICAL();
        return (OS_Q_FULL);
    }
    if (pq->OSQOut == pq->OSQStart) {                    (1)
        pq->OSQOut = pq->OSQEnd;                         (2)
    }
    pq->OSQOut--;                                        (3)
    *pq->OSQOut = msg;
    pq->OSQEntries++;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

系统函数 OSQPostFront()在 OSQOut 指向的单元的下一个单元插入新的指向消息的指针，则采用的为 LIFO 形式。OSQOut 指针指向的是已经插入消息指针的单元，因此在插入新的消息指针之前，必须先将 OSQOut 指针在消息队列中前移一个单元。

11.05 向消息队列发送一则消息（FIFO 或 LIFO）

内核提供系统函数 OSQPostOpt()，可以实现按照 FIFO 方式发送消息队列，或者按照 LIFO 方式发送消息队列，或者通过广播方式给所有等待消息队列的任务发送消息。可以替代 OSQPost()与 OSQPostFront()。

系统函数 OSQPostOpt()代码如下：

Listing 11.6 Depositing a message in a queue (Broadcast, FIFO, or LIFO), OSQPostOpt().

```
INT8U  OSQPostOpt (OS_EVENT *pevent, void *msg, INT8U opt)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_Q      *pq;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                  (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (msg == (void *)0) {                                         (2)
        return (OS_ERR_POST_NULL_PTR);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                   (3)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                               (4)
        if ((opt & OS_POST_OPT_BROADCAST) != 0x00) {                (5)
            while (pevent->OSEventGrp != 0x00) {                    (6)
                OS_EventTaskRdy(pevent, msg, OS_STAT_Q);
            }
        } else {
            OS_EventTaskRdy(pevent, msg, OS_STAT_Q);                (7)
        }
        OS_EXIT_CRITICAL();
        OS_Sched();                                                 (8)
        return (OS_NO_ERR);
    }
```

Listing 11.6 Depositing a message in a queue (Broadcast, FIFO, or LIFO), OSQPostOpt(). (Continued)

```
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries >= pq->OSQSize) {                                (9)
        OS_EXIT_CRITICAL();
        return (OS_Q_FULL);
    }
    if ((opt & OS_POST_OPT_FRONT) != 0x00) {                            (10)
        if (pq->OSQOut == pq->OSQStart) {                               (11)
            pq->OSQOut = pq->OSQEnd;
        }
        pq->OSQOut--;
        *pq->OSQOut = msg;
    } else {
        *pq->OSQIn++ = msg;                                             (12)
        if (pq->OSQIn == pq->OSQEnd) {
            pq->OSQIn = pq->OSQStart;
        }
    }
    pq->OSQEntries++;                                                   (13)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

11.06 无等待地从消息队列中获得消息

内核提供系统函数 OSQAccept()实现无等待地从消息队列中获得消息数据。具体代码如下：

Listing 11.7 Getting a message without waiting (non-blocking), OSQAccept().

```
void  *OSQAccept (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    void      *msg;
    OS_Q      *pq;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                      (1)
        return ((void *)0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                       (2)
        return ((void *)0);
    }
```

Listing 11.7 Getting a message without waiting (non-blocking), OSQAccept(). (Continued)

```
#endif
    OS_ENTER_CRITICAL();
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {                                    (3)
        msg = *pq->OSQOut++;                                     (4)
        pq->OSQEntries--;                                        (5)
        if (pq->OSQOut == pq->OSQEnd) {                          (6)
            pq->OSQOut = pq->OSQStart;
        }
    } else {
        msg = (void *)0;                                         (7)
    }
    OS_EXIT_CRITICAL();
    return (msg);
}
```

11.07 清空消息队列

内核提供系统函数 OSQFlush()清空一个消息队列中的所有的消息，以重新开始使用。系统函数 OSQFlush()的代码如下：

Listing 11.8 Flushing the contents of a queue.

```
INT8U  OSQFlush (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_Q      *pq;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                               (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                (2)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    pq             = (OS_Q *)pevent->OSEventPtr;                 (3)
    pq->OSQIn      = pq->OSQStart;
    pq->OSQOut     = pq->OSQStart;
    pq->OSQEntries = 0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

11.08 获取消息队列的状态

内核提供系统函数 OSQQuery()来获取消息队列的当前状态。其代码如下：

Listing 11.9 Obtaining the status of a queue.

```
INT8U  OSQQuery (OS_EVENT *pevent, OS_Q_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_Q      *pq;
    INT8U     *psrc;
    INT8U     *pdest;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                                  (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                   (2)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    pdata->OSEventGrp = pevent->OSEventGrp;                         (3)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
#if OS_EVENT_TBL_SIZE > 0
    *pdest++          = *psrc++;
#endif
```

Listing 11.9 Obtaining the status of a queue. (Continued)

```
#if OS_EVENT_TBL_SIZE > 1
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 4
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *pdest            = *psrc;
#endif
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {                                          (4)
        pdata->OSMsg = *pq->OSQOut;
    } else {
        pdata->OSMsg = (void *)0;
    }
    pdata->OSNMsgs = pq->OSQEntries;                                   (5)
    pdata->OSQSize = pq->OSQSize;                                      (6)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

12 内存管理

在 C 库函数中，通过 malloc()与 free()函数来动态地分配与释放内存，但在嵌入式操作系统中，多次的使用 malloc()与 free()函数来动态地分配与释放内存，会使得内存支离破碎，即存在内存碎片，无法得到连续内存。

UCOSII 操作系统把连续的大块内存按分区管理，每个分区包含整数个大小相同的内存块，在一个系统中可以有多个内存分区，应用程序可以从不同的分区得到不同大小的内存块，内存块释放时重新放回到之前所属的内存分区中。

内核提供 4 个函数实现内存管理，分别是 OSMemCreate()、OSMemGet()、OSMemPut()、OSMemQuery()。

12.00 内存控制块

内核使用内存控制块 OS_MEM 的数据结构来跟踪每一个内存分区，每个内存分区均有对应的内存控制块 OS_MEM。内存控制块 OS_MEM 的数据结构如下：

Listing 12.1　Memory control block data structure.

```
typedef struct {
    void   *OSMemAddr;
    void   *OSMemFreeList;
    INT32U  OSMemBlkSize;
    INT32U  OSMemNBlks;
    INT32U  OSMemNFree;
} OS_MEM;
```

数据结构变量 OSMemAddr 指向内存分区起始地址的指针；

数据结构变量 OSMemFreeList 指向下一个空闲的内存控制块，或者下一个空余内存块的指针；

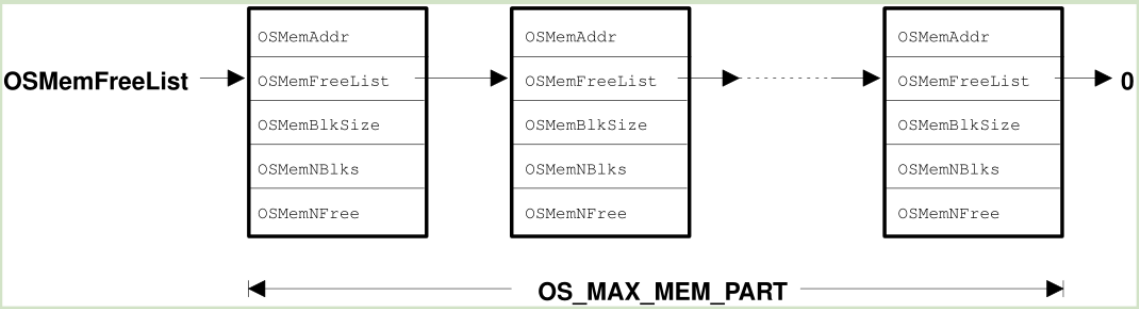
数据结构变量 OSMemBlkSize 是内存分区中内存块的大小；

数据结构变量 OSMemNBlks 是内存分区中内存块的数量；

数据结构变量 OSMemNFree 是内存分区中当前可以使用的空余的内存块的数量；

在系统进行初始化设置时，初始化函数 OSInit()会调用内存初始化函数 OSMemInit()来建立一个内存控制块链表，宏 OS_MAX_MEM_PART 定义了最大的内存分区数。

Figure 12.3 List of free memory control blocks.



12.01 建立一个内存分区

在使用一个内存分区之前，必须先建立该内存分区，内核提供系统函数 OSMemCreate()来建立一个内存分区。如果要建立一个含有 100 个内存块的内存分区，每个内存块为 32 字节，应用程序范例如下：

Listing 12.2 Creating a memory partition.

```
OS_MEM *CommTxBuf;
INT8U   CommTxPart[100][32];


void main (void)
{
    INT8U err;


    OSInit();
    .
    .
    CommTxBuf = OSMemCreate(CommTxPart, 100, 32, &err);
    .
    .
    OSStart();
}
```

以上范例调用了系统函数 OSMemCreate()来建立内存分区，第一个输入参数为内存分区的起始地址，第二个输入参数为分区内的内存块总数 100，第三个输入参数为每个内存块的字节数，第四个输入参数为指向错误代码的指针。系统函数 OSMemCreate()返回指向内存控制块的指针。

系统函数 OSMemCreate()的代码如下：

Listing 12.3 *OSMemCreate().*

```
OS_MEM  *OSMemCreate (void *addr, INT32U nblks, INT32U blksize, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    OS_MEM    *pmem;
    INT8U     *pblk;
    void     **plink;
    INT32U     i;


#if OS_ARG_CHK_EN > 0
    if (addr == (void *)0) {                                          (1)
        *err = OS_MEM_INVALID_ADDR;
        return ((OS_MEM *)0);
    }
    if (nblks < 2) {                                                  (2)
        *err = OS_MEM_INVALID_BLKS;
        return ((OS_MEM *)0);
    }
    if (blksize < sizeof(void *)) {                                   (3)
        *err = OS_MEM_INVALID_SIZE;
        return ((OS_MEM *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    pmem = OSMemFreeList;                                             (4)
    if (OSMemFreeList != (OS_MEM *)0) {
        OSMemFreeList = (OS_MEM *)OSMemFreeList->OSMemFreeList;
    }
    OS_EXIT_CRITICAL();
    if (pmem == (OS_MEM *)0) {                                        (5)
        *err = OS_MEM_INVALID_PART;
        return ((OS_MEM *)0);
    }
    plink = (void **)addr;                                            (6)
    pblk  = (INT8U *)addr + blksize;
```

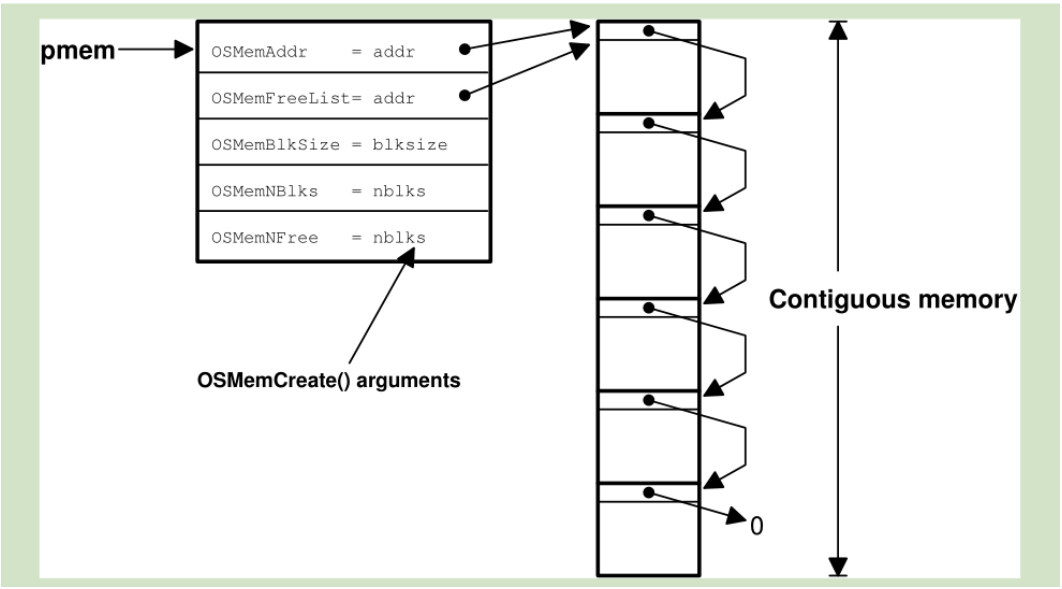
Listing 12.3 *OSMemCreate()* ***(Continued).***

```
    for (i = 0; i < (nblks - 1); i++) {
        *plink = (void *)pblk;
        plink  = (void **)pblk;
        pblk   = pblk + blksize;
    }
    *plink              = (void *)0;
    pmem->OSMemAddr     = addr;                                       (7)
    pmem->OSMemFreeList = addr;
    pmem->OSMemNFree    = nblks;
    pmem->OSMemNBlks    = nblks;
    pmem->OSMemBlkSize  = blksize;
    *err                = OS_NO_ERR;
    return (pmem);                                                    (8)
}
```

每个内存分区必须含有至少 2 个内存块（代码序号 2），每个内存块至少需要容下一个指针的空间，同一分区中所有空闲内存块由指针链接起来的（代码序号 3）。

系统函数 OSMemCreate()建立的内存分区中的所有内存块链接成一个单向的链表，如下：

Figure 12.4 Memory partition created by OSMemCreate().



12.02 分配一个内存块

内核提供系统函数 OSMemGet()从已经建立的内存分区中申请一个内存块，应用程序在使用内存块时，必须知道内存块的大小且不能超过其容量。应用程序使用完内存块后需重新释放内存块。系统函数 OSMemGet()代码如下：

Listing 12.4 OSMemGet().

```
void  *OSMemGet (OS_MEM *pmem, INT8U *err)                              (1)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    void      *pblk;
```

Listing 12.4 OSMemGet(). (Continued)

```
#if OS_ARG_CHK_EN > 0
    if (pmem == (OS_MEM *)0) {                                          (2)
        *err = OS_MEM_INVALID_PMEM;
        return ((OS_MEM *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree > 0) {                                         (3)
        pblk                = pmem->OSMemFreeList;                      (4)
        pmem->OSMemFreeList = *(void **)pblk;                           (5)
        pmem->OSMemNFree--;                                             (6)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (pblk);                                                  (7)
    }
    OS_EXIT_CRITICAL();
    *err = OS_MEM_NO_FREE_BLKS;
    return ((void *)0);
}
```

系统函数 OSMemGet()检查内存分区中是否有空余的内存块，如果有，则第一个内存块将会从空余的内存块链表中删除，空余的内存块链表被更新，链表头指针后移 1 个元素，空余内存块数量减去 1。函数返回已经分配的内存块的指针。

12.03 释放一个内存块

当应用程序不使用某个内存块，需要及时地释放当相应的内存分区中，内核提供系统函数 OSMemPut()完成此操作。应用程序需注意内存块需要放回到正确的内存分区中，否则可能引起系统崩溃。系统函数 OSMemPut()的代码如下：

Listing 12.5 OSMemPut().

```
INT8U  OSMemPut (OS_MEM  *pmem, void *pblk)                          (1)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


#if OS_ARG_CHK_EN > 0
    if (pmem == (OS_MEM *)0) {                                       (2)
        return (OS_MEM_INVALID_PMEM);
    }
    if (pblk == (void *)0) {
        return (OS_MEM_INVALID_PBLK);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree >= pmem->OSMemNBlks) {                      (3)
        OS_EXIT_CRITICAL();
        return (OS_MEM_FULL);
    }
    *(void **)pblk      = pmem->OSMemFreeList;                       (4)
    pmem->OSMemFreeList = pblk;
    pmem->OSMemNFree++;                                              (5)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

系统函数 OSMemPut()检查内存分区是否已经满，如满则说明系统在分配和释放内存时出现了错误，如果未满，则将释放的内存块插入到该内存分区的空余的内存块链表中。

12.04 查询一个内存分区的状态

内核提供系统函数 OSMemQuery()来查询一个内存分区的状态信息。代码如下：

Listing 12.6 Data structure used to obtain status from a partition.

```
typedef struct {
    void  *OSAddr;     /* Points to beginning address of memory partition     */
    void  *OSFreeList; /* Points to beginning of free list of memory blocks   */
    INT32U OSBlkSize;  /* Size (in bytes) of each memory block                */
    INT32U OSNBlks;    /* Total number of blocks in the partition             */
    INT32U OSNFree;    /* Number of memory blocks free                        */
    INT32U OSNUsed;    /* Number of memory blocks used                        */
} OS_MEM_DATA;
```

Listing 12.7 *OSMemQuery().*

```
INT8U  OSMemQuery (OS_MEM *pmem, OS_MEM_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif


#if OS_ARG_CHK_EN > 0
    if (pmem == (OS_MEM *)0) {                                          (1)
        return (OS_MEM_INVALID_PMEM);
    }
    if (pdata == (OS_MEM_DATA *)0) {
        return (OS_MEM_INVALID_PDATA);
    }
#endif
    OS_ENTER_CRITICAL();
    pdata->OSAddr     = pmem->OSMemAddr;                                (2)
    pdata->OSFreeList = pmem->OSMemFreeList;
```

Listing 12.7 *OSMemQuery().*

```
    pdata->OSBlkSize  = pmem->OSMemBlkSize;
    pdata->OSNBlks    = pmem->OSMemNBlks;
    pdata->OSNFree    = pmem->OSMemNFree;
    OS_EXIT_CRITICAL();
    pdata->OSNUsed    = pdata->OSNBlks - pdata->OSNFree;                (3)
    return (OS_NO_ERR);
}
```

12.05 使用内存分区范例

以下介绍一个使用动态分配的内存分区进行消息传递的例子。

第一个任务读取并检查模拟量输入的值，如果该值超过阈值，就向第二个任务发送一则消息，该消息中含有出错时的系统时间，出错代码等信息。该消息即通过内存动态分配得到的内存块进行传递的。

Listing 12.8 Scanning analog inputs and reporting errors.

```
AnalogInputTask()
{
    for (;;) {
        for (all analog inputs to read) {
            Read analog input;                                          (1)
            if (analog input exceeds threshold) {
                Get memory block;                                       (2)
                Get current system time (in clock ticks);               (3)
                Store the following items in the memory block.          (4)
                    System time (i.e. a time stamp);
                    The channel that exceeded the threshold;
                    An error code;
                    The severity of the error;
                    Etc.
                Post the error message to error queue;                  (5)
                    (A pointer to the memory block containing the data)
            }
        }
        Delay task until it's time to sample analog inputs again;
    }
}



ErrorHandlerTask()
{
    for (;;) {
        Wait for message from error queue;                              (6)
            (Gets a pointer to a memory block containing information
             about the error reported)
```

Listing 12.8 Scanning analog inputs and reporting errors. (Continued)

```
        Read the message and take action based on error reported;       (7)
        Return the memory block to the memory partition;                (8)
    }
}
```

12.06 等待内存分区中的一个内存块

当内存分区中暂时没有可用的内存块时，可以让申请内存块的任务等待，可以让特点的内存分区增加计数型信号量的方法实现等待的功能。

应用程序为了申请分配内存块，首先需要得到相应的信号量，才能调用 OSMemGet()函数，当释放内存块时，需要发送一个信号量即可。

Listing 12.9 Waiting for memory blocks from a partition.

```
OS_EVENT  *SemaphorePtr;                                                (1)
OS_MEM    *PartitionPtr;
INT8U      Partition[100][32];
OS_STK     TaskStk[1000];


void main (void)
{
    INT8U err;

    OSInit();                                                           (2)
    .
    .
    SemaphorePtr = OSSemCreate(100);                                    (3)
    PartitionPtr = OSMemCreate(Partition, 100, 32, &err);               (4)
    .
    OSTaskCreate(Task, (void *)0, &TaskStk[999], &err);                 (5)
    .
    OSStart();                                                          (6)
}

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

Listing 12.9 Waiting for memory blocks from a partition. (Continued)

```
    INT8U *pblock;

    for (;;) {
        OSSemPend(SemaphorePtr, 0, &err);                               (7)
        pblock = OSMemGet(PartitionPtr, &err);                          (8)
        .
        .  /* Use the memory block */
        .
        OSMemPut(PartitionPtr, pblock);                                 (9)
        OSSemPost(SemaphorePtr);                                       (10)
    }
}
```