第3章 内核结构

本章给出μC/OS-II的主要结构概貌。读者将学习以下一些内容;

- μC/OS-II是怎样处理临界段代码的;
- 什么是任务，怎样把用户的任务交给μC/OS-II;
- 任务是怎样调度的;
- 应用程序CPU的利用率是多少，μC/OS-II是怎样知道的;
- 怎样写中断服务子程序;
- 什么是时钟节拍，μC/OS-II是怎样处理时钟节拍的;
- μC/OS-II是怎样初始化的，以及
- 怎样启动多任务;

本章还描述以下函数,这些服务于应用程序:

- `OS_ENTER_CRITICAL()` 和 `OS_EXIT_CRITICAL()`,
- `OSInit()`,
- `OSStart()`,
- `OSIntEnter()` 和 `OSIntExit()`,
- `OSSchedLock()` 和 `OSSchedUnlock()`, 以及
- `OSVersion()`.

3.0 临界段(Critical Sections)

和其它内核一样，μC/OS-II为了处理临界段代码需要关中断，处理完毕后再开中断。这使得μC/OS-II能够避免同时有其它任务或中断服务进入临界段代码。关中断的时间是实时内核开发商应提供的最重要的指标之一，因为这个指标影响用户系统对实时事件的响应性。μC/OS-II努力使关中断时间降至最短，但就使用μC/OS-II而言，关中断的时间很大程度上取决于微处理器的架构以及编译器所生成的代码质量。

微处理器一般都有关中断/开中断指令，用户使用的C语言编译器必须有某种机制能够在C中直接实现关中断/开中断地操作。某些C编译器允许在用户的C源代码中插入汇编语言的语句。这使得插入微处理器指令来关中断/开中断很容易实现。而有的编译器把从C语言中关中断/开中断放在语言的扩展部分。μC/OS-II定义两个宏(macros)来关中断和开中断，以便避开不同C编译器厂商选择不同的方法来处理关中断和开中断。μC/OS-II中的这两个宏调用分别是：OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()。因为这两个宏的定义取决于所用的微处理器，故在文件OS_CPU.H中可以找到相应宏定义。每种微处理器都有自己的OS_CPU.H文件。

3.1 任务

一个任务通常是一个无限的循环[L3.1(2)]，如程序清单3.1所示。一个任务看起来像其它C的函数一样，有函数返回类型，有形式参数变量，但是任务是绝不会返回的。故返回参数必须定义成void[L3.1(1)]。

程序清单 L3.1 任务是一个无限循环

```
void YourTask (void *pdata)                                    (1)
{
  for (;;) {                                                   (2)
    /* 用户代码 */
    调用uC/OS-II的某种系统服务:
    OSMboxPend();
    OSQPend();
    OSSemPend();
    OSTaskDel(OS_PRIO_SELF);
    OSTaskSuspend(OS_PRIO_SELF);
    OSTimeDly();
    OSTimeDlyHMSM();
    /* 用户代码 */
  }
}
```

不同的是，当任务完成以后，任务可以自我删除，如清单 L3.2 所示。注意任务代码并非真的删除了，μC/OS-Ⅱ只是简单地不再理会这个任务了，这个任务的代码也不会再运行，如果任务调用了 OSTaskDel()，这个任务绝不会返回什么。

程序清单 L 3.2 ． 任务完成后自我删除

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

形式参数变量[L3.1(1)]是由用户代码在第一次执行的时候带入的。请注意，该变量的类型是一个指向 void 的指针。这是为了允许用户应用程序传递任何类型的数据给任务。这个指针好比一辆万能的车子，如果需要的话，可以运载一个变量的地址，或一个结构，甚至是一个函数的地址。也可以建立许多相同的任务，所有任务都使用同一个函数（或者说是同一个任务代码程序），见第一章的例 1。例如，用户可以将四个串行口安排成每个串行口都是一个单独的任务,而每个任务的代码实际上是相同的。并不需要将代码复制四次，用户可以建立一个任务，向这个任务传入一个指向某数据结构的指针变量，这个数据结构定义串行口的参数（波特率、I/O 口地址、中断向量号等）。

μC/OS-Ⅱ可以管理多达 64 个任务，但目前版本的μC/OS-Ⅱ有两个任务已经被系统占用了。作者保留了优先级为 0、1、2、3、OS_LOWEST_PRIO-3、OS_LOWEST_PRIO-2，OS_LOWEST_PRIO-1 以及 OS_LOWEST_PRIO 这 8 个任务以被将来使用。OS_LOWEST_PRIO 是作为定义的常数在 OS_CFG.H 文件中用定义常数语句#define constant 定义的。因此用户可以有多达 56 个应用任务。必须给每个任务赋以不同的优先级，优先级可以从 0 到 OS_LOWEST_PRIO-2。优先级号越低，任务的优先级越高。μC/OS-Ⅱ总是运行进入就绪态的

优先级最高的任务。目前版本的μC/OS-Ⅱ中，任务的优先级号就是任务编号（ID）。优先级号（或任务的 ID 号）也被一些内核服务函数调用，如改变优先级函数 OSTaskChangePrio()，以及任务删除函数 OSTaskDel()。

为了使μC/OS-Ⅱ能管理用户任务，用户必须在建立一个任务的时候，将任务的起始地址与其它参数一起传给下面两个函数中的一个：OSTastCreat 或 OSTaskCreatExt()。OSTaskCreateExt()是 OSTaskCreate()的扩展，扩展了一些附加的功能。，这两个函数的解释见第四章，任务管理。

3.2 任务状态

图 3.1 是μC/OS-Ⅱ控制下的任务状态转换图。在任一给定的时刻，任务的状态一定是在这五种状态之一。

睡眠态（DORMANT）指任务驻留在程序空间之中，还没有交给μC/OS-Ⅱ管理，（见程序清单 L3.1 或 L3.2）。把任务交给μC/OS-Ⅱ是通过调用下述两个函数之一：OSTaskCreate()或 OSTaskCreateExt()。当任务一旦建立，这个任务就进入就绪态准备运行。任务的建立可以是在多任务运行开始之前，也可以是动态地被一个运行着的任务建立。如果一个任务是被另一个任务建立的，而这个任务的优先级高于建立它的那个任务，则这个刚刚建立的任务将立即得到 CPU 的控制权。一个任务可以通过调用 OSTaskDel()返回到睡眠态，或通过调用该函数让另一个任务进入睡眠态。

调用 OSStart()可以启动多任务。OSStart()函数运行进入就绪态的优先级最高的任务。就绪的任务只有当所有优先级高于这个任务的任务转为等待状态，或者是被删除了，才能进入运行态。

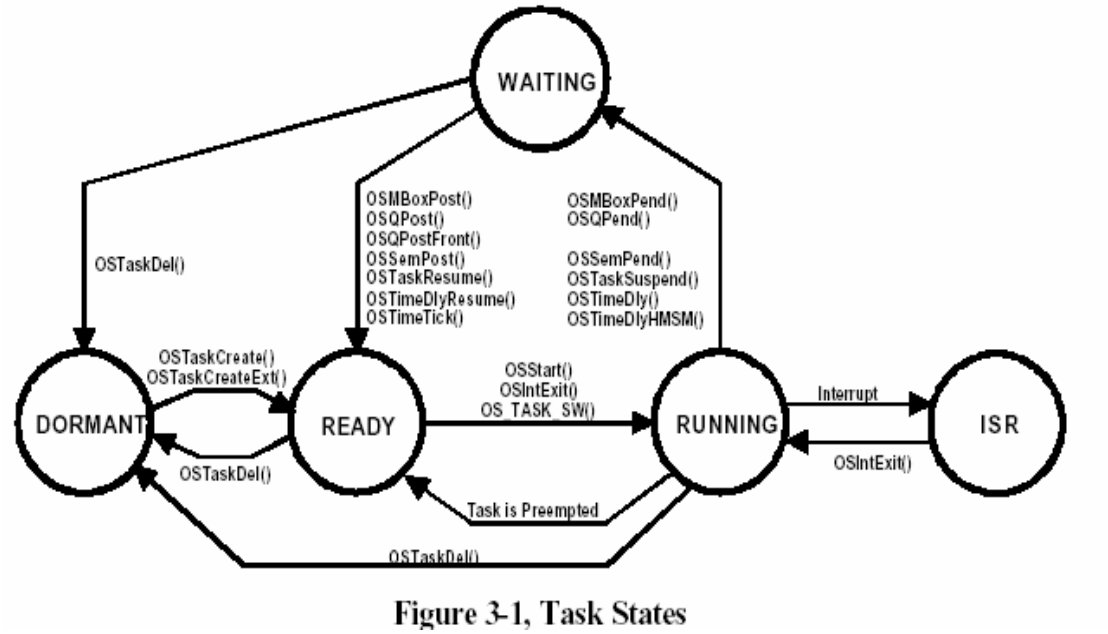

Figure 3-1, Task States

图 3.1 任务的状态

正在运行的任务可以通过调用两个函数之一将自身延迟一段时间，这两个函数是 OSTimeDly()或 OSTimeDlyHMSM()。这个任务于是进入等待状态，等待这段时间过去，下一个优先级最高的、并进入了就绪态的任务立刻被赋予了 CPU 的控制权。等待的时间过去以后，系统服务函数 OSTimeTick()使延迟了的任务进入就绪态（见 3.10 节，时钟节拍）。

正在运行的任务期待某一事件的发生时也要等待，手段是调用以下 3 个函数之一：OSSemPend()，OSMboxPend()，或 OSQPend()。调用后任务进入了等待状态（WAITING）。当任务因等待事件被挂起（Pend），下一个优先级最高的任务立即得到了 CPU 的控制权。当事件发生了，被挂起的任务进入就绪态。事件发生的报告可能来自另一个任务，也可能来自中断服务子程序。

正在运行的任务是可以被中断的，除非该任务将中断关了，或者μC/OS-II将中断关了。被中断了的任务就进入了中断服务态（ISR）。响应中断时，正在执行的任务被挂起，中断服务子程序控制了 CPU 的使用权。中断服务子程序可能会报告一个或多个事件的发生，而使一个或多个任务进入就绪态。在这种情况下，从中断服务子程序返回之前，μC/OS-II要判定，被中断的任务是否还是就绪态任务中优先级最高的。如果中断服务子程序使一个优先级更高的任务进入了就绪态，则新进入就绪态的这个优先级更高的任务将得以运行，否则原来被中断了的任务才能继续运行。

当所有的任务都在等待事件发生或等待延迟时间结束，μC/OS-II执行空闲任务（idle task），执行 OSTaskIdle()函数。

3.3 任务控制块（Task Control Blocks，OS_TCBs）

一旦任务建立了，任务控制块 OS_TCBs 将被赋值（程序清单 3.3）。任务控制块是一个数据结构，当任务的 CPU 使用权被剥夺时，μC/OS-II用它来保存该任务的状态。当任务重新得到 CPU 使用权时，任务控制块能确保任务从当时被中断的那一点丝毫不差地继续执行。OS_TCBs 全部驻留在 RAM 中。读者将会注意到笔者在组织这个数据结构时，考虑到了各成员的逻辑分组。任务建立的时候，OS_TCBs 就被初始化了（见第四章 任务管理）。

程序清单 L 3.3 μC/OS-II任务控制块.

```
typedef struct os_tcb {
    OS_STK       *OSTCBStkPtr;

#if OS_TASK_CREATE_EXT_EN
    void         *OSTCBExtPtr;
    OS_STK       *OSTCBStkBottom;
    INT32U        OSTCBStkSize;
    INT16U        OSTCBOpt;
    INT16U        OSTCBId;
#endif

    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;

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

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void         *OSTCBMsg;
```

```
#endif

    INT16U        OSTCBDly;
    INT8U         OSTCBStat;
    INT8U         OSTCBPrio;

    INT8U         OSTCBX;
    INT8U         OSTCBY;
    INT8U         OSTCBBitX;
    INT8U         OSTCBBitY;

#if OS_TASK_DEL_EN
    BOOLEAN       OSTCBDelReq;
#endif
} OS_TCB;
```

.OSTCBStkPtr 是指向当前任务栈顶的指针。μC/OS-Ⅱ允许每个任务有自己的栈，尤为重要的是，每个任务的栈的容量可以是任意的。有些商业内核要求所有任务栈的容量都一样，除非用户写一个复杂的接口函数来改变之。这种限制浪费了 RAM，当各任务需要的栈空间不同时，也得按任务中预期栈容量需求最多的来分配栈空间。OSTCBStkPtr 是 OS_TCB 数据结构中唯一的一个能用汇编语言来处置的变量（在任务切换段的代码 Context-switching code 之中，）把 OSTCBStkPtr 放在数据结构的最前面，使得从汇编语言中处理这个变量时较为容易。

.OSTCBExtPtr 指向用户定义的任务控制块扩展。用户可以扩展任务控制块而不必修改μC/OS-Ⅱ的源代码。.OSTCBExtPtr 只在函数 OstaskCreateExt()中使用，故使用时要将 OS_TASK_CREAT_EN 设为 1，以允许建立任务函数的扩展。例如用户可以建立一个数据结构，这个数据结构包含每个任务的名字，或跟踪某个任务的执行时间，或者跟踪切换到某个任务的次数（见例 3）。注意，笔者将这个扩展指针变量放在紧跟着堆栈指针的位置，为的是当用户需要在汇编语言中处理这个变量时，从数据结构的头上算偏移量比较方便。

.OSTCBStkBottom 是指向任务栈底的指针。如果微处理器的栈指针是递减的，即栈存储器从高地址向低地址方向分配，则 OSTCBStkBottom 指向任务使用的栈空间的最低地址。类似地，如果微处理器的栈是从低地址向高地址递增型的，则 OSTCBStkBottom 指向任务可以使用的栈空间的最高地址。函数 OSTaskStkChk()要用到变量 OSTCBStkBottom，在运行中检验栈空间的使用情况。用户可以用它来确定任务实际需要的栈空间。这个功能只有当用户在任务建立时允许使用 OSTaskCreateExt()函数时才能实现。这就要求用户将 OS_TASK_CREATE_EXT_EN 设为 1，以便允许该功能。

.OSTCBStkSize 存有栈中可容纳的指针元数目而不是用字节（Byte）表示的栈容量总数。也就是说，如果栈中可以保存 1,000 个入口地址，每个地址宽度是 32 位的，则实际栈容量是 4,000 字节。同样是 1,000 个入口地址，如果每个地址宽度是 16 位的，则总栈容量只有 2,000 字节。在函数 OSStakChk()中要调用 OSTCBStkSize。同理，若使用该函数的话，要将 OS_TASK_CREAT_EXT_EN 设为 1。

.OSTCBOpt 把“选择项”传给 OSTaskCreateExt()，只有在用户将 OS_TASK_CREATE_EXT_EN 设为 1 时，这个变量才有效。μC/OS-Ⅱ目前只支持 3 个选择项（见 uCOS_II.H）：OS_TASK_OTP_STK_CHK, OS_TASK_OPT_STK_CLR 和 OS_TASK_OPT_SAVE_FP 。

OS_TASK_OTP_STK_CHK 用于告知 TaskCreateExt()，在任务建立的时候任务栈检验功能得到了允许。OS_TASK_OPT_STK_CLR 表示任务建立的时候任务栈要清零。只有在用户需要有栈检验功能时，才需要将栈清零。如果不定义 OS_TASK_OPT_STK_CLR，而后又建立、删除了任务，栈检验功能报告的栈使用情况将是错误的。如果任务一旦建立就决不会被删除，而用户初始化时，已将 RAM 清过零，则 OS_TASK_OPT_STK_CLR 不需要再定义，这可以节约程序执行时间。传递了 OS_TASK_OPT_STK_CLR 将增加 TaskCreateExt()函数的执行时间，因为要将栈空间清零。栈容量越大，清零花的时间越长。最后一个选择项 OS_TASK_OPT_SAVE_FP 通知 TaskCreateExt()，任务要做浮点运算。如果微处理器有硬件的浮点协处理器，则所建立的任务在做任务调度切换时，浮点寄存器的内容要保存。

.OSTCBId 用于存储任务的识别码。这个变量现在没有使用，留给将来扩展用。

.OSTCBNext 和 **.OSTCBPrev** 用于任务控制块 OS_TCBs 的双重链接，该链表在时钟节拍函数 OSTimeTick()中使用，用于刷新各个任务的任务延迟变量.OSTCBDly，每个任务的任务控制块 OS_TCB 在任务建立的时候被链接到链表中，在任务删除的时候从链表中被删除。双重连接的链表使得任一成员都能被快速插入或删除。

.OSTCBEventPtr 是指向事件控制块的指针，后面的章节中会有所描述（见第 6 章 任务间通讯与同步）。

.OSTCBMsg 是指向传给任务的消息的指针。用法将在后面的章节中提到（见第 6 章任务间通讯与同步）。

.OSTCBDly 当需要把任务延时若干时钟节拍时要用到这个变量，或者需要把任务挂起一段时间以等待某事件的发生，这种等待是有超时限制的。在这种情况下，这个变量保存的是任务允许等待事件发生的最多时钟节拍数。如果这个变量为 0，表示任务不延时，或者表示等待事件发生的时间没有限制。

.OSTCBStat 是任务的状态字。当.OSTCBStat 为 0，任务进入就绪态。可以给.OSTCBStat 赋其它的值，在文件 uCOS_II.H 中有关于这个值的描述。

.OSTCBPrio 是任务优先级。高优先级任务的.OSTCBPrio 值小。也就是说，这个值越小，任务的优先级越高。

.OSTCBX, .OSTCBY, .OSTCBBitX 和 .OSTCBBitY 用于加速任务进入就绪态的过程或进入等待事件发生状态的过程（避免在运行中去计算这些值）。这些值是在任务建立时算好的，或者是在改变任务优先级时算出的。这些值的算法见程序清单 L3.4。

程序清单 L 3.4 任务控制块OS_TCB中几个成员的算法

```
OSTCBY          = priority >> 3;
OSTCBBitY       = OSMapTbl[priority >> 3];
OSTCBX          = priority & 0x07;
OSTCBBitX       = OSMapTbl[priority & 0x07];
```

.OSTCBDelReq 是一个布尔量，用于表示该任务是否需要删除，用法将在后面的章节中描述（见第 4 章 任务管理）

应用程序中可以有的最多任务数（OS_MAX_TASKS）是在文件 OS_CFG.H 中定义的。这个最多任务数也是 μC/OS-II 分配给用户程序的最多任务控制块 OS_TCBs 的数目。将 OS_MAX_TASKS 的数目设置为用户应用程序实际需要的任务数可以减小 RAM 的需求量。所有的任务控制块 OS_TCBs 都是放在任务控制块列表数组 OSTCBTbl[]中的。请注意，μC/OS-II

分配给系统任务 OS_N_SYS_TASKS 若干个任务控制块，见文件μC/OS-Ⅱ.H，供其内部使用。目前，一个用于空闲任务，另一个用于任务统计（如果 OS_TASK_STAT_EN 是设为 1 的）。在μC/OS-Ⅱ初始化的时候，如图 3.2 所示，所有任务控制块 OS_TCBs 被链接成单向空任务链表。当任务一旦建立，空任务控制块指针 OSTCBFreeList 指向的任务控制块便赋给了该任务，然后 OSTCBFreeList 的值调整为指向下链表中下一个空的任务控制块。一旦任务被删除，任务控制块就还给空任务链表。

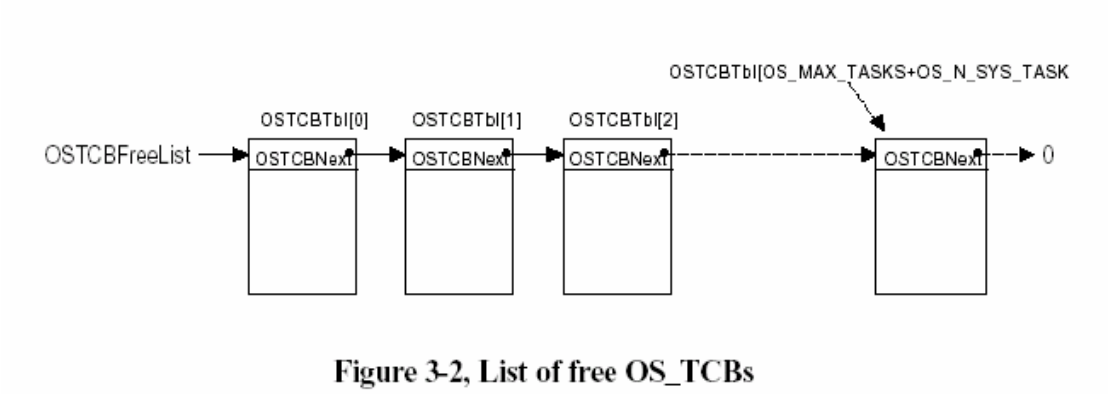

Figure 3-2, List of free OS_TCBs

图 3.2 空任务列表

3.4 就绪表（Ready List）

每个任务被赋予不同的优先级等级，从 0 级到最低优先级 OS_LOWEST_PRIO，包括 0 和 OS_LOWEST_PRIO 在内（见文件 OS_CFG.H）。当μC/OS-Ⅱ初始化的时候，最低优先级 OS_LOWEST_PRIO 总是被赋给空闲任务 idle task。注意，最多任务数目 OS_MAX_TASKS 和最低优先级数是没有关系的。用户应用程序可以只有 10 个任务，而仍然可以有 32 个优先级的级别（如果用户将最低优先级数设为 31 的话）。

每个任务的就绪态标志都放入就绪表中的，就绪表中有两个变量 OSRedyGrp 和 OSRdyTbl[]。在 OSRdyGrp 中，任务按优先级分组，8 个任务为一组。OSRdyGrp 中的每一位表示 8 组任务中每一组中是否有进入就绪态的任务。任务进入就绪态时，就绪表 OSRdyTbl[]中的相应元素的相应位也置位。就绪表 OSRdyTbl[]数组的大小取决于 OS_LOWEST_PRIO(见文件 OS_CFG.H)。当用户的应用程序中任务数目比较少时，减少 OS_LOWEST_PRIO 的值可以降低μC/OS-Ⅱ对 RAM（数据空间）的需求量。

为确定下次该哪个优先级的任务运行了，内核调度器总是将 OS_LOWEST_PRIO 在就绪表中相应字节的相应位置 1。OSRdyGrp 和 OSRdyTbl[]之间的关系见图 3.3，是按以下规则给出的：

当 OSRdyTbl[0]中的任何一位是 1 时，OSRdyGrp 的第 0 位置 1，
当 OSRdyTbl[1]中的任何一位是 1 时，OSRdyGrp 的第 1 位置 1，
当 OSRdyTbl[2]中的任何一位是 1 时，OSRdyGrp 的第 2 位置 1，
当 OSRdyTbl[3]中的任何一位是 1 时，OSRdyGrp 的第 3 位置 1，
当 OSRdyTbl[4]中的任何一位是 1 时，OSRdyGrp 的第 4 位置 1，
当 OSRdyTbl[5]中的任何一位是 1 时，OSRdyGrp 的第 5 位置 1，
当 OSRdyTbl[6]中的任何一位是 1 时，OSRdyGrp 的第 6 位置 1，
当 OSRdyTbl[7]中的任何一位是 1 时，OSRdyGrp 的第 7 位置 1，

程序清单 3.5 中的代码用于将任务放入就绪表。Prio 是任务的优先级。

程序清单 L3.5 使任务进入就绪态

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

表 T3.1 OSMapTbl[]的值

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

读者可以看出，任务优先级的低三位用于确定任务在总就绪表 OSRdyTbl[]中的所在位。接下去的三位用于确定是在 OSRdyTbl[]数组的第几个元素。OSMapTbl[]是在 ROM 中的（见文件 OS_CORE.C）屏蔽字，用于限制 OSRdyTbl[]数组的元素下标在 0 到 7 之间，见表 3.1

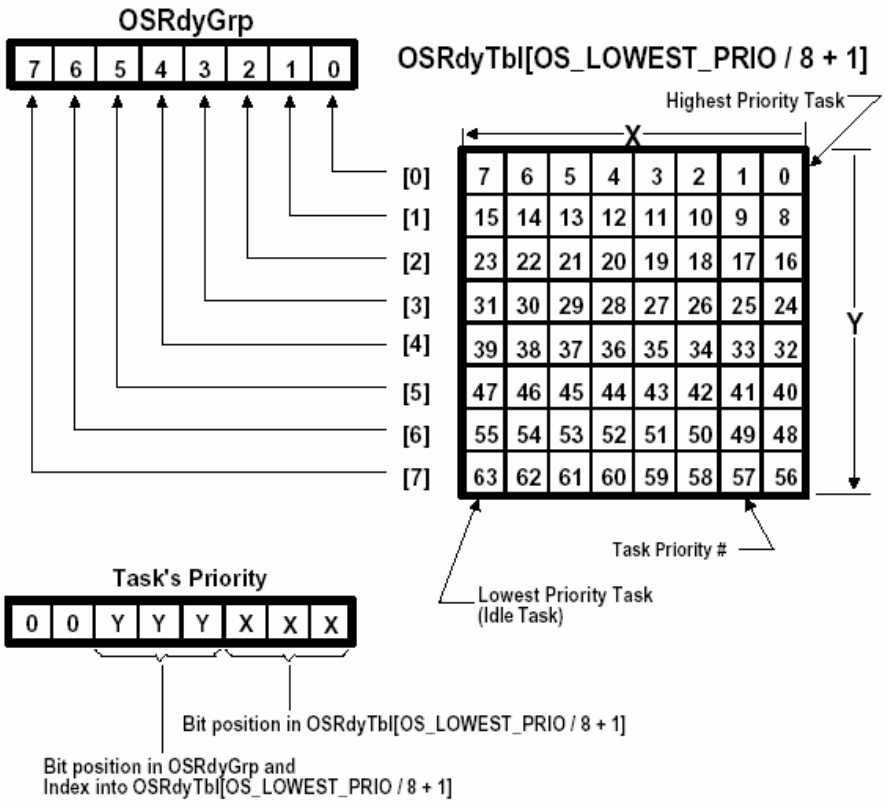

Figure 3-3, µC/OS-II's Ready List

图 3.3 µC/OS-Ⅱ就绪表

如果一个任务被删除了，则用程序清单 3.6 中的代码做求反处理。

程序清单 L3.6 从就绪表中删除一个任务

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

以上代码将就绪任务表数组 OSRdyTbl[]中相应元素的相应位清零，而对于 OSRdyGrp，只有当被删除任务所在任务组中全组任务一个都没有进入就绪态时，才将相应位清零。也就是说 OSRdyTbl[prio>>3]所有的位都是零时，OSRdyGrp 的相应位才清零。为了找到那个进入就绪态的优先级最高的任务，并不需要从 OSRdyTbl[0]开始扫描整个就绪任务表，只需要查另外一张表，即优先级判定表 OSUnMapTbl([256])(见文件 OS_CORE.C)。OSRdyTbl[]中每个字节的 8 位代表这一组的 8 个任务哪些进入就绪态了，低位的优先级高于高位。利用这个字节为下标来查 OSUnMapTbl 这张表，返回的字节就是该组任务中就绪态任务中优先级最高的那个任务所在的位置。这个返回值在 0 到 7 之间。确定进入就绪态的优先级最高的任务是用以下代码完成的，如程序清单 L3.7 所示。

程序清单 L3.7 找出进入就绪态的优先级最高的任务

```
y    = OSUnMapTbl[OSRdyGrp];
x    = OSUnMapTbl[OSRdyTbl[y]];
```

```
prio = (y << 3) + x;
```

例如，如果 OSRdyGrp 的值为二进制 01101000，查 OSUnMapTbl[OSRdyGrp]得到的值是 3，它相应于 OSRdyGrp 中的第 3 位 bit3，这里假设最右边的一位是第 0 位 bit0。类似地，如果 OSRdyTbl[3]的值是二进制 11100100,则 OSUnMapTbl[OSRdyTbc[3]]的值是 2，即第 2 位。于是任务的优先级 Prio 就等于 26（3*8+2）。利用这个优先级的值。查任务控制块优先级表 OSTCBPrioTbl[]，得到指向相应任务的任务控制块 OS_TCB 的工作就完成了。

3.5 任务调度（Task Scheduling）

μC/OS-Ⅱ总是运行进入就绪态任务中优先级最高的那一个。确定哪个任务优先级最高，下面该哪个任务运行了的工作是由调度器（Scheduler）完成的。任务级的调度是由函数 OSSched()完成的。中断级的调度是由另一个函数 OSIntExt()完成的，这个函数将在以后描述。OSSched()的代码如程序清单 L3.8 所示。

程序清单 L3.8 任务调度器（the Task Scheduler）

```
void OSSched (void)
{
    INT8U y;



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

μC/OS-Ⅱ任务调度所花的时间是常数，与应用程序中建立的任务数无关。如程序清单中[L3.8(1)]条件语句的条件不满足，任务调度函数 OSSched()将退出，不做任务调度。这个条件是：如果在中断服务子程序中调用 OSSched()，此时中断嵌套层数 OSIntNesting>0，或者由于用户至少调用了一次给任务调度上锁函数 OSSchedLock()，使 OSLockNesting>0。如果不是在中断服务子程序调用 OSSched()，并且任务调度是允许的，即没有上锁，则任务调度函数将找出那个进入就绪态且优先级最高的任务[L3.8(2)]，进入就绪态的任务在就绪任务表中有相应的位置位。一旦找到那个优先级最高的任务，OSSched()检验这个优先级最高的任务是不是当前正在运行的任务，以此来避免不必要的任务调度[L3.8(3)]。注意，在μC/OS 中曾经是先得到 OSTCBHighRdy 然后和 OSTCBCur 做比较。因为这个比较是两个指针型变量的比较，在 8 位和一些 16 位微处理器中这种比较相对较慢。而在μC/OS-Ⅱ中是两个整数的比较。并且，除非用户实际需要做任务切换，在查任务控制块优先级表 OSTCBPrioTbl[]

时，不需要用指针变量来查 OSTCBHighRdy。综合这两项改进，即用整数比较代替指针的比较和当需要任务切换时再查表，使得μC/OS-Ⅱ比μC/OS 在 8 位和一些 16 位微处理器上要更快一些。

为实现任务切换，OSTCBHighRdy 必须指向优先级最高的那个任务控制块 OS_TCB，这是通过将以 OSPrioHighRdy 为下标的 OSTCBPrioTbl[]数组中的那个元素赋给 OSTCBHighRdy 来实现的[L3.8(4)]。接着，统计计数器 OSCtxSwCtr 加 1，以跟踪任务切换次数[L3.8(5)]。最后宏调用 OS_TASK_SW()来完成实际上的任务切换[L3.8(6)]。

任务切换很简单，由以下两步完成，将被挂起任务的微处理器寄存器推入堆栈，然后将较高优先级的任务的寄存器值从栈中恢复到寄存器中。在μC/OS-Ⅱ中，就绪任务的栈结构总是看起来跟刚刚发生过中断一样，所有微处理器的寄存器都保存在栈中。换句话说，μC/OS-Ⅱ运行就绪态的任务所要做的一切，只是恢复所有的 CPU 寄存器并运行中断返回指令。为了做任务切换，运行 OS_TASK_SW(),人为模仿了一次中断。多数微处理器有软中断指令或者陷阱指令 TRAP 来实现上述操作。中断服务子程序或陷阱处理（Trap hardler），也称作事故处理（exception handler），必须提供中断向量给汇编语言函数 OSCtxSw()。OSCtxSw()除了需要 OS_TCBHighRdy 指向即将被挂起的任务，还需要让当前任务控制块 OSTCBCur 指向即将被挂起的任务，参见第 8 章，移植μC/OS-Ⅱ，有关于 OSCtxSw()的更详尽的解释。

OSSched()的所有代码都属临界段代码。在寻找进入就绪态的优先级最高的任务过程中，为防止中断服务子程序把一个或几个任务的就绪位置位，中断是被关掉的。为缩短切换时间，OSSched()全部代码都可以用汇编语言写。为增加可读性，可移植性和将汇编语言代码最少化，OSSched()是用 C 写的。

3.6 给调度器上锁和开锁(Locking and UnLocking the Scheduler)

给调度器上锁函数 OSSchedlock()（程序清单 L3.9）用于禁止任务调度，直到任务完成后调用给调度器开锁函数 OSSchedUnlock()为止，（程序清单 L3.10）。调用 OSSchedlock()的任务保持对 CPU 的控制权，尽管有个优先级更高的任务进入了就绪态。然而，此时中断是可以被识别的，中断服务也能得到(假设中断是开着的)。OSSchedlock()和 OSSchedUnlock()必须成对使用。变量 OSLockNesting 跟踪 OSSchedLock()函数被调用的次数，以允许嵌套的函数包含临界段代码，这段代码其它任务不得干预。μC/OS-Ⅱ允许嵌套深度达 255 层。当 OSLockNesting 等于零时，调度重新得到允许。函数 OSSchedLock()和 OSSchedUnlock()的使用要非常谨慎，因为它们影响μC/OS-Ⅱ对任务的正常管理。

当 OSLockNesting 减到零的时候，OSSchedUnlock()调用 OSSched[L3.10(2)]。OSSchedUnlock()是被某任务调用的，在调度器上锁的期间，可能有什么事件发生了并使一个更高优先级的任务进入就绪态。

调用 OSSchedLock()以后，用户的应用程序不得使用任何能将现行任务挂起的系统调用。也就是说，用户程序不得调用 OSMboxPend()、OSQPend()、OSSemPend()、OSTaskSuspend(OS_PRIO_SELF)、OSTimeDly()或 OSTimeDlyHMSM(),直到 OSLockNesting 回零为止。因为调度器上了锁，用户就锁住了系统，任何其它任务都不能运行。

当低优先级的任务要发消息给多任务的邮箱、消息队列、信号量时(见第 6 章 任务间通讯和同步)，用户不希望高优先级的任务在邮箱、队列和信号量没有得到消息之前就取得了 CPU 的控制权，此时，用户可以使用禁止调度器函数。

程序清单 L3.9 给调度器上锁

```
void OSSchedLock (void)
```

```
{
    if (OSRunning == TRUE) {
        OS_ENTER_CRITICAL();
        OSLockNesting++;
        OS_EXIT_CRITICAL();
    }
}
```

程序清单 L3.10 给调度器开锁.

```
void OSSchedUnlock (void)
{
    if (OSRunning == TRUE) {
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {
            OSLockNesting--;
            if ((OSLockNesting | OSIntNesting) == 0) {                (1)
                OS_EXIT_CRITICAL();
                OSSched();                                            (2)
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}
```

3.7 空闲任务(Idle Task)

μC/OS-Ⅱ总是建立一个空闲任务，这个任务在没有其它任务进入就绪态时投入运行。这个空闲任务[OSTaskIdle()]永远设为最低优先级，即 OS_LOWEST_PRIO。空闲任务 OSTaskIdle()什么也不做，只是在不停地给一个 32 位的名叫 OSIdleCtr 的计数器加 1，统计任务（见 3.08 节，统计任务）使用这个计数器以确定现行应用软件实际消耗的 CPU 时间。程序清单 L3.11 是空闲任务的代码。在计数器加 1 前后，中断是先关掉再开启的，因为 8 位以及大多数 16 位微处理器的 32 位加 1 需要多条指令，要防止高优先级的任务或中断服务子程序从中打入。空闲任务不可能被应用软件删除。

程序清单 L3.11 μC/OS-Ⅱ的空闲任务.

```
void OSTaskIdle (void *pdata)
{
    pdata = pdata;
    for (;;) {
        OS_ENTER_CRITICAL();
```

```
        OSIdleCtr++;
        OS_EXIT_CRITICAL();
    }
}
```

3.8 统计任务

μC/OS-Ⅱ有一个提供运行时间统计的任务。这个任务叫做 OSTaskStat(),如果用户将系统定义常数 OS_TASK_STAT_EN（见文件 OS_CFG.H）设为 1，这个任务就会建立。一旦得到了允许，OSTaskStat()每秒钟运行一次（见文件 OS_CORE.C），计算当前的 CPU 利用率。换句话说，OSTaskStat()告诉用户应用程序使用了多少 CPU 时间，用百分比表示，这个值放在一个有符号 8 位整数 OSCPUsage 中，精读度是 1 个百分点。

如果用户应用程序打算使用统计任务，用户必须在初始化时建立一个唯一的任务，在这个任务中调用 OSStatInit()(见文件 OS_CORE.C)。换句话说，在调用系统启动函数 OSStart()之前，用户初始代码必须先建立一个任务，在这个任务中调用系统统计初始化函数 OSStatInit()，然后再建立应用程序中的其它任务。程序清单 L3.12 是统计任务的示意性代码。

程序清单 L3.12 初始化统计任务.

```
void main (void)
{
    OSInit();                /* 初始化uC/OS-II                              (1)*/
    /* 安装uC/OS-II的任务切换向量                                              */
    /* 创建用户起始任务(为了方便讨论，这里以TaskStart()作为起始任务)          (2)*/
    OSStart();               /* 开始多任务调度                              (3)*/
}




void TaskStart (void *pdata)
{
    /* 安装并启动uC/OS-II的时钟节拍                                          (4)*/
    OSStatInit();             /* 初始化统计任务                            (5)*/
    /* 创建用户应用程序任务                                                    */
    for (;;) {
        /* 这里是TaskStart()的代码!                                           */
    }
}
```

因为用户的应用程序必须先建立一个起始任务[TaskStart（）]，当主程序 main()调用系统启动函数 OSStcnt()的时候，μC/OS-Ⅱ只有 3 个要管理的任务：TaskStart()、OSTaskIdle()和 OSTaskStat()。请注意，任务 TaskStart()的名称是无所谓的，叫什么名字都可以。因为μC/OS-Ⅱ已经将空闲任务的优先级设为最低，即 OS_LOWEST_PRIO，统计任务

的优先级设为次低，OS_LOWEST_PRIO-1。启动任务 TaskStart()总是优先级最高的任务。

图 F3.4 解释初始化统计任务时的流程。用户必须首先调用的是μC/OS-Ⅱ中的系统初始化函数 OSInit()，该函数初始化μC/OS-Ⅱ[图 F3.4(2)]。有的处理器（例如 Motorola 的 MC68HC11），不需要“设置”中断向量，中断向量已经在 ROM 中有了。用户必须调用 OSTaskCreat()或者 OSTaskCreatExt()以建立 TaskStart()[图 F3.4(3)]。进入多任务的条件准备好了以后，调用系统启动函数 OSStart()。这个函数将使 TaskStart()开始执行，因为 TaskStart()是优先级最高的任务[图 F3.4(4)]]。

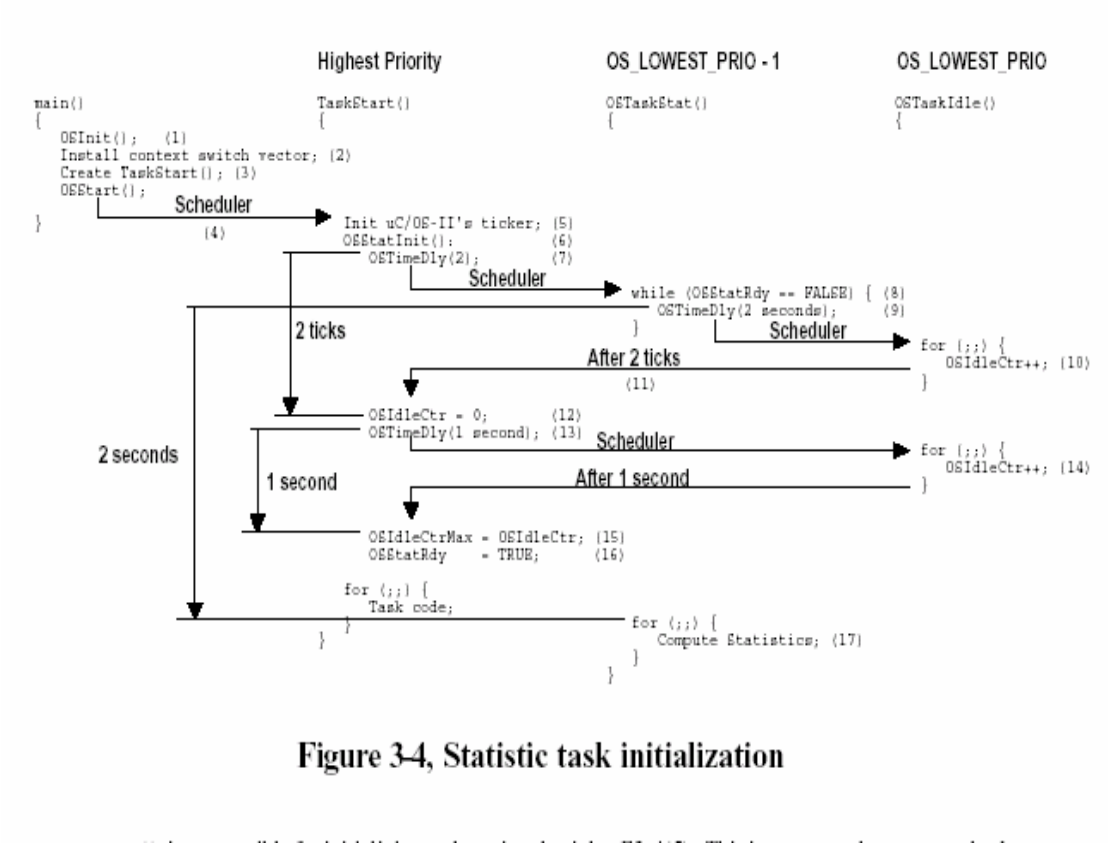

Figure 3-4, Statistic task initialization

图 F3.4 统计任务的初始化

TaskStart()负责初始化和启动时钟节拍[图 F3.4(5)]。在这里启动时钟节拍是必要的，因为用户不会希望在多任务还没有开始时就接收到时钟节拍中断。接下去 TaskStart()调用统计初始化函数 OSStatInit()[图 F3.4（6）]。统计初始化函数 OSStatInit()决定在没有其它应用任务运行时，空闲计数器（OSIdleCtr）的计数有多快。奔腾 II 微处理器以 333MHz 运行时，加 1 操作可以使该计数器的值达到每秒 15,000,000 次。OSIdleCtr 的值离 32 位计数器的溢出极限值 4,294,967,296 还差得远。微处理器越来越快，用户要注意这里可能会是将来的一个潜在问题。

系统统计初始化任务函数OSStatInit()调用延迟函数OSTimeDly()将自身延时2个时钟节拍以停止自身的运行[图 F3.4(7)]。这是为了使 OSStatInit()与时钟节拍同步。μC/OS-Ⅱ然后选下一个优先级最高的进入就绪态的任务运行，这恰好是统计任务 OSTaskStat()。读者会在后面读到 OSTaskStat()的代码，但粗看一下，OSTaskStat()所要做的第一件事就是查看统计任务就绪标志是否为“假”，如果是的话，也要延时两个时钟节拍[图 F3.4(8)]。一定会是这样，因为标志 OSStatRdy 已被 OSInit()函数初始化为“假”，所以实际上 DSTaskStat 也将自己推入休眠态（Sleep）两个时钟节拍[图 F3.4(9)]。于是任务切换到空

闲任务,OSTaskIdle()开始运行,这是唯一一个就绪态任务了。CPU 处在空闲任务 OSTaskIdle 中，直到 TaskStart()的延迟两个时钟节拍完成[图 3.4(10)]。两个时钟节拍之后，TaskStart()恢复运行[图 F3.4(11)]。 在执行 OSStartInit()时，空闲计数器 OSIdleCtr 被清零[图 F3.4(12)]。然后，OSStatInit()将自身延时整整一秒[图 F3.4(13)]。因为没有其它进入就绪态的任务，OSTaskIdle()又获得了 CPU 的控制权[图 F3.4(14)]。一秒钟以后，TaskStart()继续运行，还是在 OSStatInit()中，空闲计数器将 1 秒钟内计数的值存入空闲计数器最大值 OSIdleCtrMax 中[图 F3.4(15)]。

OSStarInit()将统计任务就绪标志 OSStatRdy 设为“真”[图 F3.4（16）]，以此来允许两个时钟节拍以后 OSTaskStat()开始计算 CPU 的利用率。

统计任务的初始化函数 OSStatInit()的代码如程序清单L3.13 所示。

程序清单 L3.13 统计任务的初始化.

```
void OSStatInit (void)
{
    OSTimeDly(2);
    OS_ENTER_CRITICAL();
    OSIdleCtr    = 0L;
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC);
    OS_ENTER_CRITICAL();
    OSIdleCtrMax = OSIdleCtr;
    OSStatRdy    = TRUE;
    OS_EXIT_CRITICAL();
}
```

统计任务 OSStat()的代码程序清单 L3.14 所示。在前面一段中，已经讨论了为什么要等待统计任务就绪标志 OSStatRdy[L3.14(1)]。这个任务每秒执行一次，以确定所有应用程序中的任务消耗了多少 CPU 时间。当用户的应用程序代码加入以后，运行空闲任务的 CPU 时间就少了，OSIdleCtr 就不会像原来什么任务都不运行时有那么多计数。要知道，OSIdleCtr 的最大计数值是 OSStatInit()在初始化时保存在计数器最大值 OSIdleCtrMax 中的。CPU 利用率（表达式[3.1]）是保存在变量 OSCPUsage[L3.14(2)]中的：

[3.1]表达式 Need to typeset the equation.

一旦上述计算完成，OSTaskStat()调用任务统计外界接入函数 OSTaskStatHook()[L3.14(3)]，这是一个用户可定义的函数，这个函数能使统计任务得到扩展。这样，用户可以计算并显示所有任务总的执行时间，每个任务执行时间的百分比以及其它信息(参见 1.09 节例 3)。

程序清单 L3.14 统计任务

```
void OSTaskStat (void *pdata)
{
    INT32U run;
    INT8S usage;
```

```
    pdata = pdata;
    while (OSStatRdy == FALSE) {                                        (1)
        OSTimeDly(2 * OS_TICKS_PER_SEC);
    }
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtrRun = OSIdleCtr;
        run          = OSIdleCtr;
        OSIdleCtr    = 0L;
        OS_EXIT_CRITICAL();
        if (OSIdleCtrMax > 0L) {
            usage = (INT8S)(100L - 100L * run / OSIdleCtrMax);          (2)
            if (usage > 100) {
                OSCPUUsage = 100;
            } else if (usage < 0) {
                OSCPUUsage = 0;
            } else {
                OSCPUUsage = usage;
            }
        } else {
            OSCPUUsage = 0;
        }
        OSTaskStatHook();                                               (3)
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

3.9 μC/OS 中的中断处理

μC/OS 中，中断服务子程序要用汇编语言来写。然而，如果用户使用的 C 语言编译器支持在线汇编语言的话，用户可以直接将中断服务子程序代码放在C 语言的程序文件中。中断服务子程序的示意码如程序清单 L3.15 所示。

程序清单 L3.15 μC/OS-II中的中断服务子程序.

```
用户中断服务子程序:
    保存全部CPU寄存器;                                (1)
    调用OSIntEnter或OSIntNesting直接加1;              (2)
    执行用户代码做中断服务;                           (3)
    调用OSIntExit();                                  (4)
    恢复所有CPU寄存器;                                (5)
    执行中断返回指令;                                 (6)
```

用户代码应该将全部CPU寄存器推入当前任务栈[L3.15(1)]。注意，有些微处理器，例如Motorola68020(及68020以上的微处理器)，做中断服务时使用另外的堆栈。μC/OS-Ⅱ可以用在这类微处理器中，当任务切换时，寄存器是保存在被中断了的那个任务的栈中的。

μC/OS-Ⅱ需要知道用户在做中断服务，故用户应该调用OSIntEnter()，或者将全程变量OSIntNesting[L3.15(2)]直接加1,如果用户使用的微处理器有存储器直接加1的单条指令的话。如果用户使用的微处理器没有这样的指令，必须先将OSIntNesting读入寄存器，再将寄存器加 1，然后再写回到变量 OSIatNesting 中去，就不如调用 OSIatEnter()。OSIntNesting 是共享资源。OSIntEnter()把上述三条指令用开中断、关中断保护起来，以保证处理OSIntNesting时的排它性。直接给OSIntNesting加1比调用OSIntEnter()快得多，可能时，直接加1更好。要当心的是，在有些情况下，从OSIntEnter()返回时，会把中断开了。遇到这种情况，在调用 OSIntEnter()之前要先清中断源，否则，中断将连续反复打入，用户应用程序就会崩溃!

上述两步完成以后，用户可以开始服务于叫中断的设备了[L3.15(3)]。这一段完全取决于应用。μC/OS-Ⅱ允许中断嵌套，因为μC/OS-Ⅱ跟踪嵌套层数 OSIntNesting。然而，为允许中断嵌套，在多数情况下，用户应在开中断之前先清中断源。

调用脱离中断函数OSIntExit()[L3.15(4)]标志着中断服务子程序的终结，OSIntExit()将中断嵌套层数计数器减1。当嵌套计数器减到零时，所有中断，包括嵌套的中断就都完成了，此时μC/OS-Ⅱ要判定有没有优先级较高的任务被中断服务子程序(或任一嵌套的中断)唤醒了。如果有优先级高的任务进入了就绪态，μC/OS-Ⅱ就返回到那个高优先级的任务，OSIntExit()返回到调用点[L3.15(5)]。保存的寄存器的值是在这时恢复的，然后是执行中断返回指令[L3.16(6)]。注意，如果调度被禁止了(OSIntNesting>0)，μC/OS-Ⅱ将被返回到被中断了的任务。

以上描述的详细解释如图F3.5所示。中断来到了[F3.5(1)]但还不能被被CPU识别，也许是因为中断被μC/OS-Ⅱ或用户应用程序关了，或者是因为CPU还没执行完当前指令。一旦 CPU 响应了这个中断[F3.5(2)]，CPU 的中断向量（至少大多数微处理器是如此）跳转到中断服务子程序[F3.5(3)]。如上所述，中断服务子程序保存 CPU 寄存器（也叫做 CPU context）[F3.5(4)]，一旦做完，用户中断服务子程序通知μC/OS-Ⅱ进入中断服务子程序了，办法是调用OSIntEnter()或者给OSIntNesting直接加1[F3.5(5)]。然后用户中断服务代码开始执行[F3.5(6)]。用户中断服务中做的事要尽可能地少，要把大部分工作留给任务去做。中断服务子程序通知某任务去做事的手段是调用以下函数之一：OSMboxPost()，OSQPost()，OSQPostFront()，OSSemPost()。中断发生并由上述函数发出消息时，接收消息的任务可能是，也可能不是挂起在邮箱、队列或信号量上的任务。用户中断服务完成以后，要调用 OSIntExit()[F3.5(7)]。从时序图上可以看出，对被中断了的任务说来，如果没有高优先级的任务被中断服务子程序激活而进入就绪态,OSIntExit()只占用很短的运行时间。进而，在这种情况下，CPU寄存器只是简单地恢复[F3.5(8)]并执行中断返回指令[F3.5(9)]。如果中断服务子程序使一个高优先级的任务进入了就绪态,则OSIntExit()将占用较长的运行时间，因为这时要做任务切换[F3.5(10)]。新任务的寄存器内容要恢复并执行中断返回指令[F3.5(12)]。

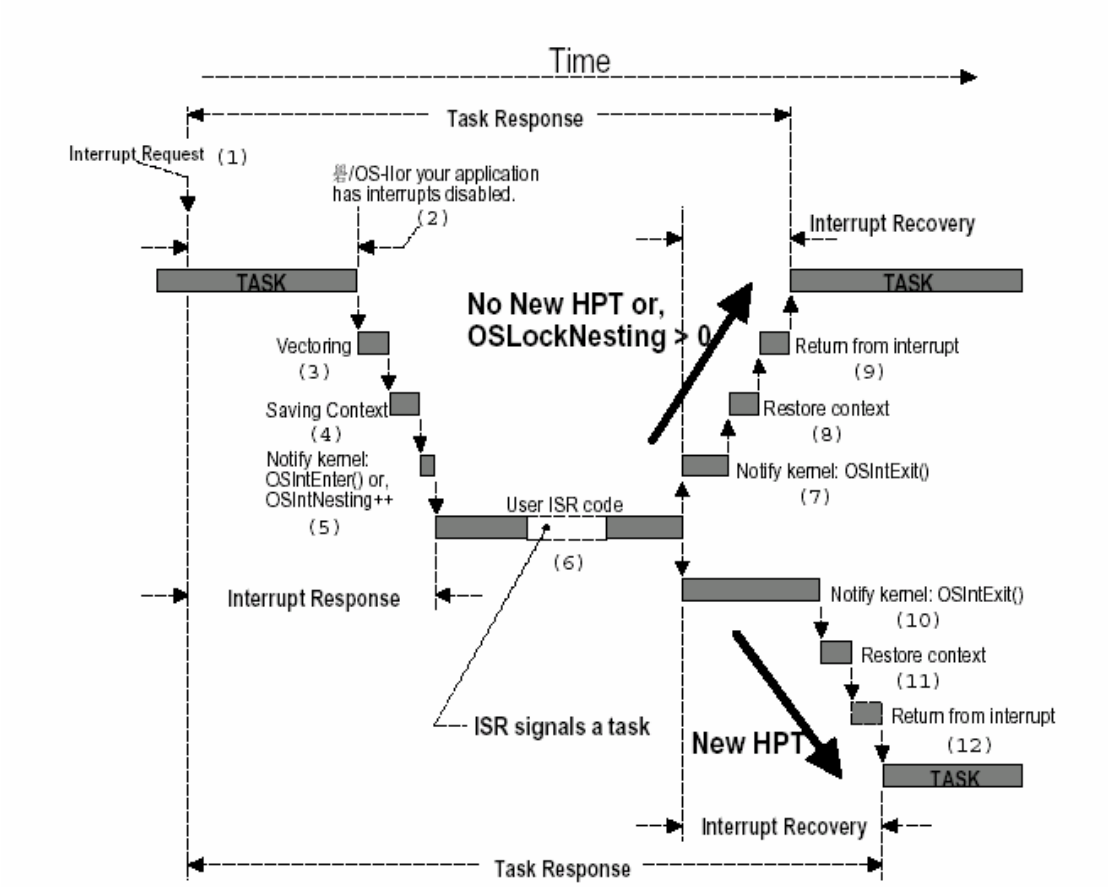

Figure 3-5, Servicing an interrupt

图 3.5 中断服务

进入中断函数 OSIntEnter()的代码如程序清单 L3.16 所示，从中断服务中退出函数 OSIntExit()的代码如程序清单 L3.17 所示。如前所述，OSIntEnter()所做的事是非常少的。

程序清单 L3.16 通知μC/OS-Ⅱ，中断服务子程序开始了.

```
void OSIntEnter (void)
{
    OS_ENTER_CRITICAL();
    OSIntNesting++;
    OS_EXIT_CRITICAL();
}
```

程序清单 L3.17 通知μC/OS-Ⅱ，脱离了中断服务

```
void OSIntExit (void)
{
    OS_ENTER_CRITICAL();                                          (1)
    if ((--OSIntNesting | OSLockNesting) == 0) {                  (2)
        OSIntExitY    = OSUnMapTbl[OSRdyGrp];                     (3)
```

```
        OSPrioHighRdy = (INT8U)((OSIntExitY << 3) +
                        OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OSIntCtxSw();                                              (4)
        }
    }
    OS_EXIT_CRITICAL();
}
```

OSIntExit()看起来非常像 OSSched()。但有三点不同。第一点，OSIntExit()使中断嵌套层数减 1[L3.17(2)]而调度函数 OSSched()的调度条件是：中断嵌套层数计数器和锁定嵌套计数器（OSLockNesting）二者都必须是零。第二个不同点是，OSRdyTbl[]所需的检索值 Y 是保存在全程变量 OSIntExitY 中的[L3.17(3)]。这是为了避免在任务栈中安排局部变量。这个变量在哪儿和中断任务切换函数 OSIntCtxSw()有关，（见 9.04.03 节，中断任务切换函数）。最后一点，如果需要做任务切换，OSIntExit()将调用 OSIntCtxSw()[L3.17（4）]而不是调用 OS_TASK_SW(),正像在 OSSched()函数中那样。

调用中断切换函数 OSIntCtxSw()而不调用任务切换函数 OS_TASK_SW()，有两个原因，首先是，如程序清单中 L3.5(1)和图 F3.6(1)所示，一半的工作，即 CPU 寄存器入栈的工作已经做完了。第二个原因是，在中断服务子程序中调用 OSIntExit()时，将返回地址推入了堆栈[L3.15(4)和 F3.6(2)]。OSIntExit()中的进入临界段函数 OS_ENTER_CRITICAL()或许将 CPU 的状态字也推入了堆栈 L3.7(1)和 F3.6(3)。这取决于中断是怎么被关掉的（见第 8 章移植μC/OS-Ⅱ）。最后，调用 OSIntCtxSw()时的返回地址又被推入了堆栈[L3.17(4)和 F3.1(4)]，除了栈中不相关的部分，当任务挂起时，栈结构应该与μC/OS-Ⅱ所规定的完全一致。OSIntCtxSw()只需要对栈指针做简单的调整，如图 F3.6(5)所示。换句话说，调整栈结构要保证所有挂起任务的栈结构看起来是一样的。

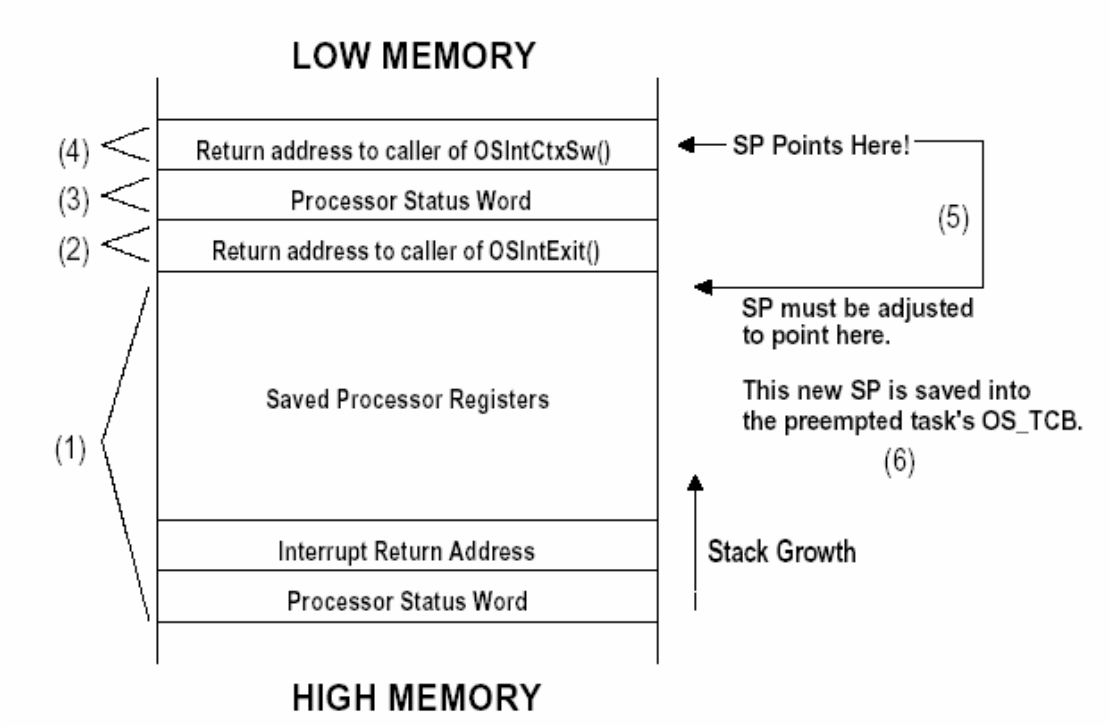

Figure 3-6, Cleanup by OSIntCtxSw().

图 3.6 中断中的任务切换函数 OSIntCtxSw()调整栈结构

有的微处理器，像 Motorola 68HC11 中断发生时 CPU 寄存器是自动入栈的，且要想允许中断嵌套的话，在中断服务子程序中要重新开中断，这可以视作一个优点。确实，如果用户中断服务子程序执行得非常快，用户不需要通知任务自身进入了中断服务，只要不在中断服务期间开中断，也不需要调用 OSIntEnter()或 OSIntNesting 加 1。程序清单 L3。18 中的示意代码表示这种情况。一个任务和这个中断服务子程序通讯的唯一方法是通过全程变量。

程序清单 L3.18 Motorola 68HC11中的中断服务子程序

```
M68HC11_ISR:                    /* 快中断服务程序，必须禁止中断*/
    所有寄存器被CPU自动保存;
    执行用户代码以响应中断;
    执行中断返回指令;
```

3.10 时钟节拍

μC/OS 需要用户提供周期性信号源，用于实现时间延时和确认超时。节拍率应在每秒 10 次到 100 次之间，或者说 10 到 100Hz。时钟节拍率越高，系统的额外负荷就越重。时钟节拍的实际频率取决于用户应用程序的精度。时钟节拍源可以是专门的硬件定时器，也可以是来自 50/60Hz 交流电源的信号。

用户**必须**在多任务系统启动**以后**再开启时钟节拍器，也就是在调用 OSStart()之后。换句话说，在调用 OSStart()之后做的第一件事是初始化定时器中断。通常，容易犯的错误是将允许时钟节拍器中断放在系统初始化函数 OSInit()之后，在调启动多任务系统启动函数

OSStart()之前，如程序清单 L3.19 所示。

程序清单 L3.19 启动时钟就节拍器的不正确做法.

```
void main(void)
{
    .
    .
    OSInit();                /* 初始化uC/OS-II                                  */
    .
    .
    /* 应用程序初始化代码 ...                                                    */
    /* ... 通过调用OSTaskCreate()创建至少一个任务                                */
    .
    .
    允许时钟节拍（TICKER）中断; /* 千万不要在这里允许时钟节拍中断!!!             */
    .
    .
    OSStart();               /* 开始多任务调度                                  */
}
```

这里潜在地危险是，时钟节拍中断有可能在μC/OS-II 启动第一个任务之前发生，此时μC/OS-II 是处在一种不确定的状态之中，用户应用程序有可能会崩溃。

μC/OS-II 中的时钟节拍服务是通过在中断服务子程序中调用 OSTimeTick()实现的。时钟节拍中断服从所有前面章节中描述的规则。时钟节拍中断服务子程序的示意代码如程序清单 L3.20 所示。这段代码必须用汇编语言编写，因为在 C 语言里不能直接处理 CPU 的寄存器。

程序清单 L3.20 时钟节拍中断服务子程序的示意代码

```
void OSTickISR(void)
{
    保存处理器寄存器的值;
    调用OSIntEnter()或是将OSIntNesting加1;
    调用OSTimeTick();

    调用OSIntExit();
    恢复处理器寄存器的值;
    执行中断返回指令;
}
```

时钟节拍函数 OSTimeTick()的代码如程序清单 3.21 所示。OSTimtick()以调用可由用户定义的时钟节拍外连函数 OSTimTickHook()开始，这个外连函数可以将时钟节拍函数 OSTimtick()予以扩展[L3.2(1)]。笔者决定首先调用 OSTimTickHook()是打算在时钟节拍中断服务一开始就给用户一个可以做点儿什么的机会，因为用户可能会有一些时间要求苛刻的工作要做。OSTimtick()中量大的工作是给每个用户任务控制块 OS_TCB 中的时间延时项

OSTCBDly 减 1（如果该项不为零的话）。OSTimTick()从 OSTCBList 开始，沿着 OS_TCB 链表做，一直做到空闲任务[L3.21(3)]。当某任务的任务控制块中的时间延时项 OSTCBDly 减到了零，这个任务就进入了就绪态[L3.21(5)]。而确切被任务挂起的函数 OSTaskSuspend()挂起的任务则不会进入就绪态[L3.21(4)]。OSTimTick()的执行时间直接与应用程序中建立了多少个任务成正比。

程序清单 L3.21 时钟节拍函数 OSTimtick() 的一个节拍服务

```
void OSTimeTick (void)
{
    OS_TCB *ptcb;

    OSTimeTickHook();                                                    (1)
    ptcb = OSTCBList;                                                    (2)
    while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {                            (3)
        OS_ENTER_CRITICAL();
        if (ptcb->OSTCBDly != 0) {
            if (--ptcb->OSTCBDly == 0) {
                if (!(ptcb->OSTCBStat & OS_STAT_SUSPEND)) {              (4)
                    OSRdyGrp             |= ptcb->OSTCBBitY;             (5)
                    OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                } else {
                    ptcb->OSTCBDly = 1;
                }
            }
        }
        ptcb = ptcb->OSTCBNext;
        OS_EXIT_CRITICAL();
    }
    OS_ENTER_CRITICAL();                                                 (6)
    OSTime++;                                                            (7)
    OS_EXIT_CRITICAL();
}
```

OSTimeTick()还通过调用 OSTime()[L3.21(7)]累加从开机以来的时间，用的是一个无符号 32 位变量。注意，在给 OSTime 加 1 之前使用了关中断，因为多数微处理器给 32 位数加 1 的操作都得使用多条指令。

中断服务子程序似乎就得写这么长，如果用户不喜欢将中断服务程序写这么长，可以从任务级调用 OSTimeTick()，如程序清单 L3.22 所示。要想这么做，得建立一个高于应用程序中所有其它任务优先级的任务。时钟节拍中断服务子程序利用信号量或邮箱发信号给这个高优先级的任务。

程序清单 L3.22 时钟节拍任务 TickTask() 作时钟节拍服务.

```
void TickTask (void *pdata)
{
```

```
    pdata = pdata;
    for (;;) {
        OSMboxPend(...);    /* 等待从时钟节拍中断服务程序发来的信号 */
        OSTimeTick();
    }
}
```

用户当然需要先建立一个邮箱（初始化成 NULL）用于发信号给上述任何告知时钟节拍中断已经发生了（程序清单 L3.23）。

程序清单L3.23时钟节拍中断服务函数OSTickISR()做节拍服务。

```
void OSTickISR(void)
{
    保存处理器寄存器的值;
    调用OSIntEnter()或是将OSIntNesting加1;

    发送一个‘空’消息(例如， (void *)1)到时钟节拍的邮箱;

    调用OSIntExit();
    恢复处理器寄存器的值;
    执行中断返回指令;
}
```

3.11 μC/OS-Ⅱ初始化

在调用μC/OS-Ⅱ的任何其它服务之前，μC/OS-Ⅱ要求用户首先调用系统初始化函数 OSIint()。OSIint()初始化μC/OS-Ⅱ所有的变量和数据结构（见 OS_CORE.C）。

OSInit()建立空闲任务 idle task，这个任务总是处于就绪态的。空闲任务 OSTaskIdle（）的优先级总是设成最低，即 OS_LOWEST_PRIO。如果统计任务允许 OS_TASK_STAT_EN 和任务建立扩展允许都设为 1，则 OSInit()还得建立统计任务 OSTaskStat()并且让其进入就绪态。OSTaskStat 的优先级总是设为 OS_LOWEST_PRIO-1。

图 F3.7 表示调用 OSInit()之后，一些μC/OS-Ⅱ变量和数据结构之间的关系。其解释是基于以下假设的：

- 在文件 OS_CFG.H 中，OS_TASK_STAT_EN 是设为 1 的。
- 在文件 OS_CFG.H 中，OS_LOWEST_PRIO 是设为 63 的。
- 在文件 OS_CFG.H 中, 最多任务数 OS_MAX_TASKS 是设成大于 2 的。

以上两个任务的任务控制块（OS_TCBs）是用双向链表链接在一起的。OSTCBList 指向这个链表的起始处。当建立一个任务时，这个任务总是被放在这个链表的起始处。换句话说，OSTCBList 总是指向最后建立的那个任务。链的终点指向空字符 NULL(也就是零)。

因为这两个任务都处在就绪态，在就绪任务表 OSRdyTbl[]中的相应位是设为 1 的。还有，因为这两个任务的相应位是在 OSRdyTbl[]的同一行上，即属同一组，故 OSRdyGrp 中只有 1 位是设为 1 的。

μC/OS-Ⅱ还初始化了4个空数据结构缓冲区，如图F3.8所示。每个缓冲区都是单向链表，允许μC/OS-Ⅱ从缓冲区中迅速得到或释放一个缓冲区中的元素。注意，空任务控制块在空缓冲区中的数目取决于最多任务数OS_MAX_TASKS，这个最多任务数是在OS_CFG.H文件中定义的。μC/OS-Ⅱ自动安排总的系统任务数 OS_N_SYS_TASKS(见文件μC/OS-Ⅱ.H)。控制块OS_TCB的数目也就自动确定了。当然，包括足够的任务控制块分配给统计任务和空闲任务。指向空事件表OSEventFreeList和空队列表OSFreeList的指针将在第6章，任务间通讯与同步中讨论。指向空存储区的指针表OSMemFreeList将在第7章存储管理中讨论。

3.12 μC/OS-Ⅱ的启动

多任务的启动是用户通过调用OSStart()实现的。然而，启动μC/OS-Ⅱ之前，用户至少要建立一个应用任务，如程序清单L3.24所示。

程序清单 L3.24 初始化和启动μC/OS-Ⅱ

```
void main (void)
{
    OSInit();          /* 初始化uC/OS-II                          */
    .
    .
    通过调用OSTaskCreate()或OSTaskCreateExt()创建至少一个任务;
    .
    .
    OSStart();         /* 开始多任务调度!OSStart()永远不会返回 */
}
```

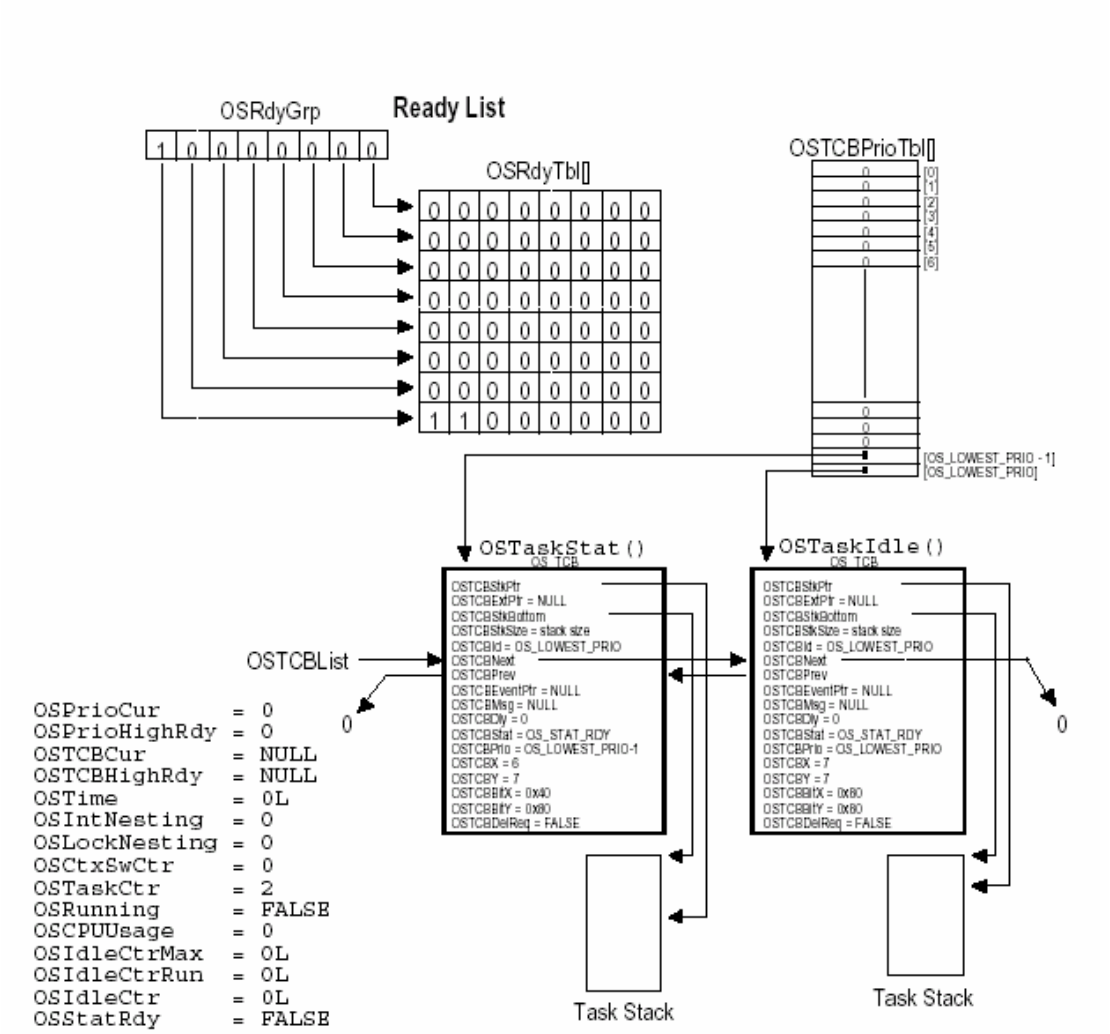

Figure 3-7, Data structures after calling OSInit()

图 3.7 调用 OSInit()之后的数据结构

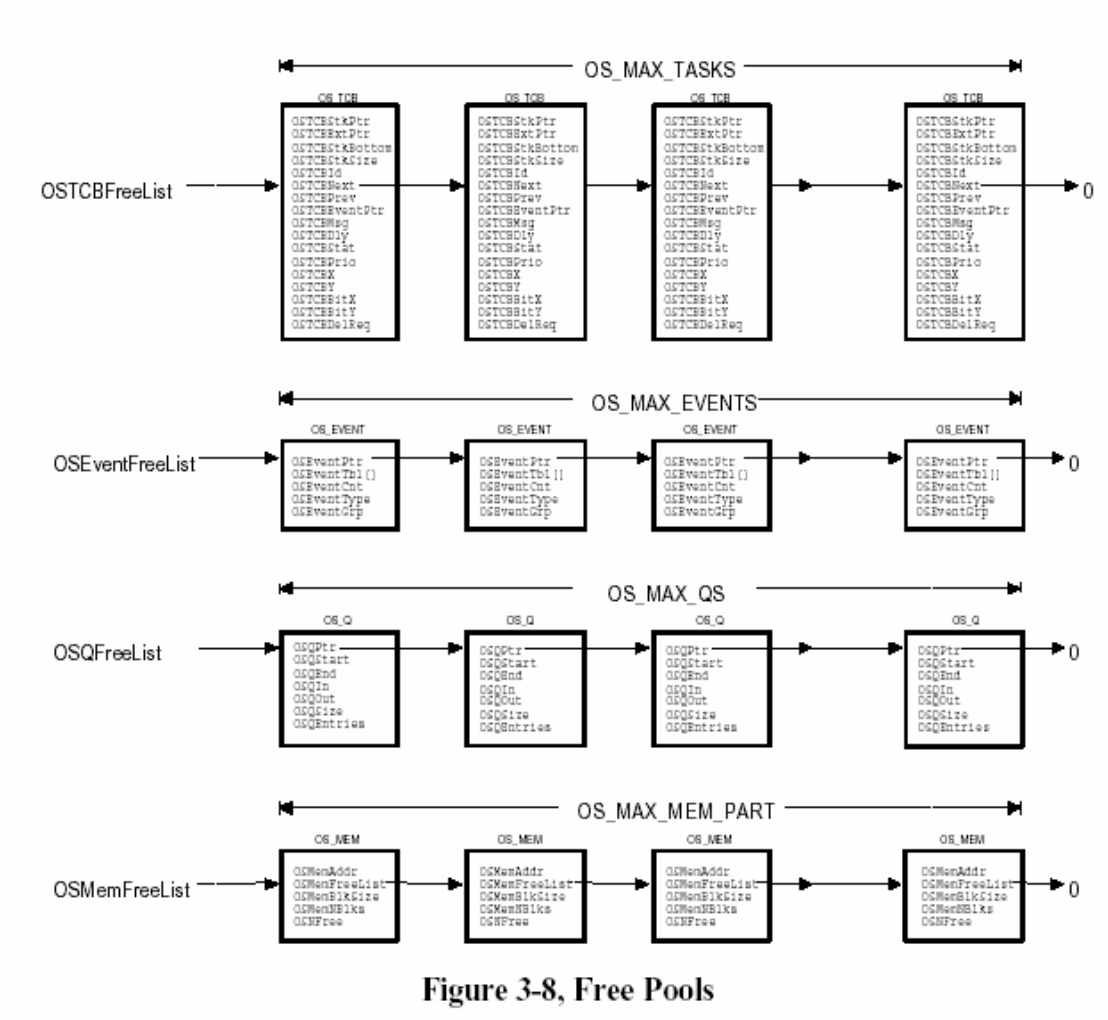

Figure 3-8, Free Pools

图 3.8 空缓冲区

OSStart()的代码如程序清单 L3.25 所示。当调用 OSStart()时，OSStart()从任务就绪表中找出那个用户建立的优先级最高任务的任务控制块[L3.25(1)]。然后，OSStart()调用高优先级就绪任务启动函数 OSStartHighRdy()[L3,25(2)]，(见汇编语言文件 OS_CPU_A.ASM)，这个文件与选择的微处理器有关。实质上，函数 OSStartHighRdy()是将任务栈中保存的值弹回到 CPU 寄存器中，然后执行一条中断返回指令，中断返回指令强制执行该任务代码。见 9.04.01 节，高优先级就绪任务启动函数 OSStartHighRdy()。那一节详细介绍对于 80x86 微处理器是怎么做的。注意，OSStartHighRdy()将永远不返回到 OSStart()。

程序清单 L3.25 启动多任务.

```
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)
    }
}
```

多任务启动以后变量与数据结构中的内容如图 F3.9 所示。这里笔者假设用户建立的任务优先级为 6，注意，OSTaskCtr 指出已经建立了 3 个任务。OSRunning 已设为“真”，指出多任务已经开始，OSPrioCur 和 OSPrioHighRdy 存放的是用户应用任务的优先级，OSTCBCur 和 OSTCBHighRdy 二者都指向用户任务的任务控制块。

3.13 获取当前μC/OS-Ⅱ的版本号

应用程序调用 OSVersion()[程序清单 L3.26]可以得到当前μC/OS-Ⅱ的版本号。OSVersion()函数返回版本号值乘以 100。换言之，200 表示版本号 2.00。

程序清单 L3.26 得到μC/OS-Ⅱ当前版本号

```
INT16U OSVersion (void)
{
    return (OS_VERSION);
}
```

为找到μC/OS-Ⅱ的最新版本以及如何做版本升级，用户可以与出版商联系，或者查看μC/OS-Ⅱ得正式网站 WWW. uCOS-II.COM

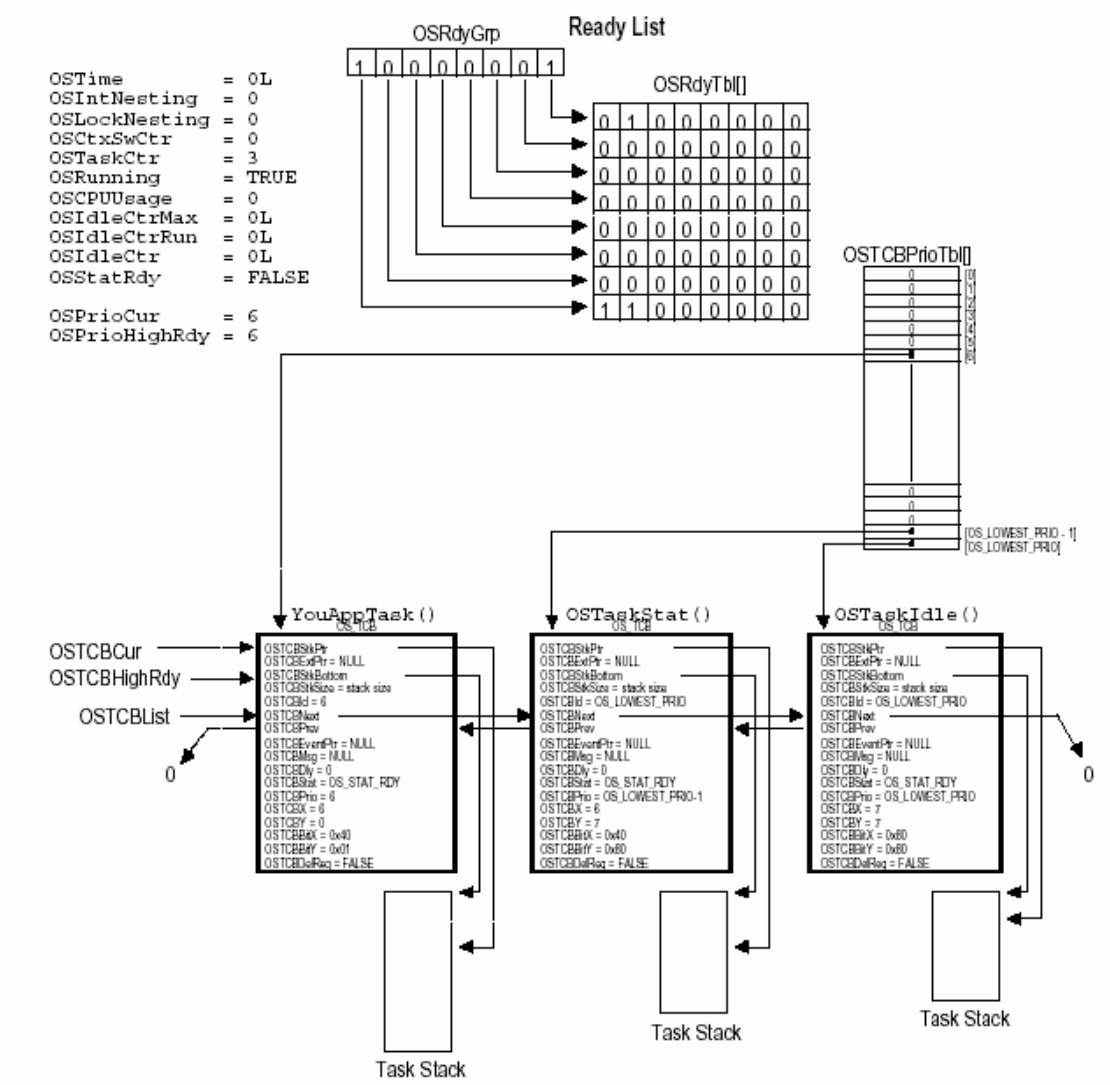

Figure 3-9, Variables and Data Structures after calling OSStart()

图 3.9 调用 OSStart()以后的变量与数据结构

3.14 OSEvent???()函数

读者或许注意到有 4 个 OS_CORE.C 中的函数没有在本章中提到。这 4 个函数是 OSEventWaitListInit()，OSEventTaskRdy()，OSEventTaskWait()，OSEventTO()。这几个函数是放在文件 OS_CORE.C 中的，而对如何使用这个函数的解释见第 6 章，任务间的通讯与同步。