第 5 章 时间管理 1
5.0 任务延时函数，OSTimeDly() 1
5.1 按时分秒延时函数 OSTimeDlyHMSM() 3
5.2 让处在延时期的任务结束延时，OSTimeDlyResume() 5
5.3 系统时间，OSTimeGet()和 OSTimeSet() 7

第5章 时间管理

在 3.10 节时钟节拍中曾提到，μC/OS-Ⅱ(其它内核也一样)要求用户提供定时中断来实现延时与超时控制等功能。这个定时中断叫做时钟节拍，它应该每秒发生 10 至 100 次。时钟节拍的实际频率是由用户的应用程序决定的。时钟节拍的频率越高，系统的负荷就越重。

3.10 节讨论了时钟的中断服务子程序和节时钟节函数 OSTimeTick——该函数用于通知μC/OS-Ⅱ发生了时钟节拍中断。本章主要讲述五个与时钟节拍有关的系统服务：

- OSTimeDly()
- OSTimeDlyHMSM()
- OSTimeDlyResume()
- OSTimeGet()
- OSTimeSet()

本章所提到的函数可以在 OS_TIME.C 文件中找到。

5.0 任务延时函数，OSTimeDly()

μC/OS-Ⅱ提供了这样一个系统服务：申请该服务的任务可以延时一段时间，这段时间的长短是用时钟节拍的数目来确定的。实现这个系统服务的函数叫做 OSTimeDly()。调用该函数会使 μC/OS-Ⅱ进行一次任务调度，并且执行下一个优先级最高的就绪态任务。任务调用 OSTimeDly()后，一旦规定的时间期满或者有其它的任务通过调用 OSTimeDlyResume()取消了延时，它就会马上进入就绪状态。注意，只有当该任务在所有就绪任务中具有最高的优先级时，它才会立即运行。

程序清单 L5.1 所示的是任务延时函数 OSTimeDly()的代码。用户的应用程序是通过提供延时的时钟节拍数——一个 1 到 65535 之间的数，来调用该函数的。如果用户指定 0 值[L5.1(1)]，则表明用户不想延时任务，函数会立即返回到调用者。非 0 值会使得任务延时函数 OSTimeDly()将当前任务从就绪表中移除[L5.1(2)]。接着，这个延时节拍数会被保存在当前任务的 OS_TCB 中[L5.1(3)]，并且通过 OSTimeTick()每隔一个时钟节拍就减少一个延时节拍数。最后，既然任务已经不再处于就绪状态，任务调度程序会执行下一个优先级最高的就绪任务。

程序清单 L 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)
    }
}
```

清楚地认识0到一个节拍之间的延时过程是非常重要的。换句话说，如果用户只想延时一个时钟节拍，而实际上是在0到一个节拍之间结束延时。即使用户的处理器的负荷不是很重，这种情况依然是存在的。图F5.1详细说明了整个过程。系统每隔10ms发生一次时钟节拍中断[F5.1(1)]。假如用户没有执行其它的中断并且此时的中断是开着的，时钟节拍中断服务就会发生[F5.1(2)]。也许用户有好几个高优先级的任务(HPT)在等待延时期满，它们会接着执行[F5.1(3)]。接下来，图5.1中所示的低优先级任务(LPT)会得到执行的机会，该任务在执行完后马上调用[F5.1(4)]所示的OSTimeDly(1)。μC/OS-Ⅱ会使该任务处于休眠状态直至下一个节拍的到来。当下一个节拍到来后，时钟节拍中断服务子程序会执行[F5.1(5)]，但是这一次由于没有高优先级的任务被执行，μC/OS-Ⅱ会立即执行申请延时一个时钟节拍的任务[F5.1(6)]。正如用户所看到的，该任务实际的延时少于一个节拍！在负荷很重的系统中，任务甚至有可能会在时钟中断即将发生时调用 OSTimeDly(1)，在这种情况下，任务几乎就没有得到任何延时，因为任务马上又被重新调度了。如果用户的应用程序至少得延时一个节拍，必须要调用OSTimeDly(2)，指定延时两个节拍！

Figure 5.1 Delay resolution.

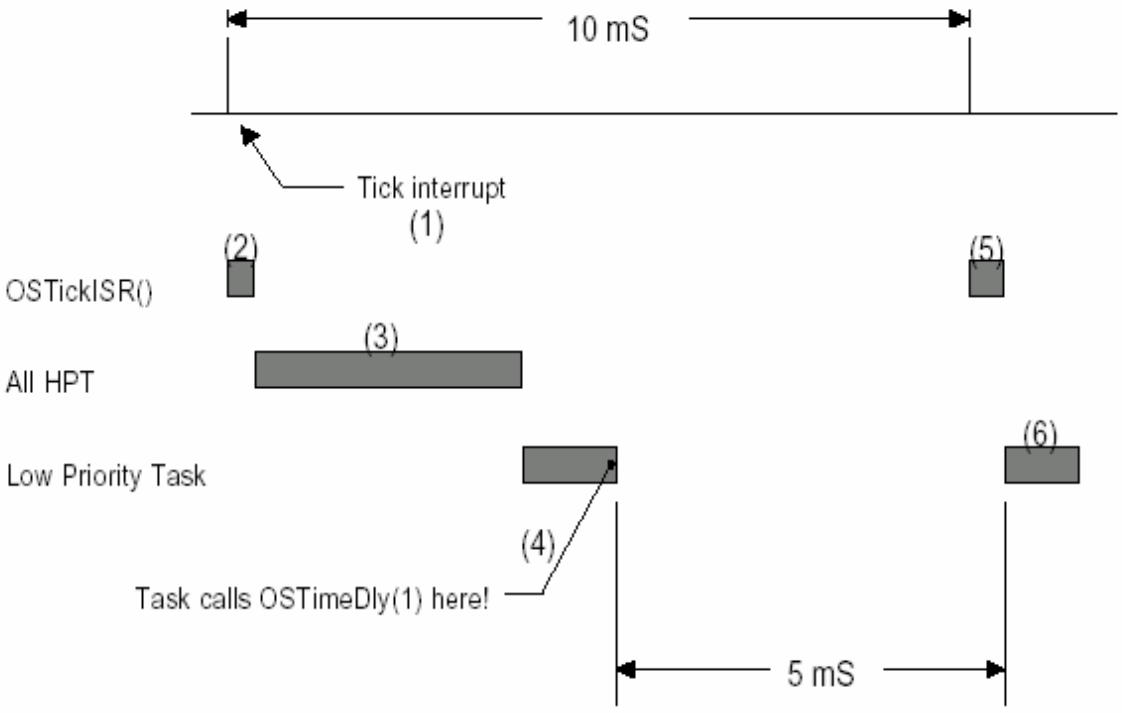

Figure 5-1, Delay resolution

5.1 按时分秒延时函数 OSTimeDlyHMSM()

OSTimeDly()虽然是一个非常有用的函数，但用户的应用程序需要知道延时时间对应的时钟节拍的数目。用户可以使用定义全局常数 OS_TICKS_PER_SEC(参看 OS_CFG.H)的方法将时间转换成时钟段，但这种方法有时显得比较愚笨。笔者增加了 OSTimeDlyHMSM()函数后，用户就可以按小时(H)、分(M)、秒(S)和毫秒(m)来定义时间了，这样会显得更自然些。与 OSTimeDly()一样，调用 OSTimeDlyHMSM()函数也会使 μC/OS-Ⅱ进行一次任务调度，并且执行下一个优先级最高的就绪态任务。任务调用 OSTimeDlyHMSM()后，一旦规定的时间期满或者有其它的任务通过调用 OSTimeDlyResume()取消了延时(参看 5.02,恢复延时的任务 OSTimeDlyResume())，它就会马上处于就绪态。同样，只有当该任务在所有就绪态任务中具有最高的优先级时，它才会立即运行。

程序清单 L5.2 所示的是 OSTimeDlyHMSM()的代码。从中可以看出，应用程序是通过用小时、分、秒和毫秒指定延时来调用该函数的。在实际应用中，用户应避免使任务延时过长的时间,因为从任务中获得一些反馈行为(如减少计数器,清除 LED 等等)经常是很不错的事。但是，如果用户确实需要延时长时间的话，μC/OS-Ⅱ可以将任务延时长达 256 个小时(接近 11 天)。

OSTimeDlyHMSM()一开始先要检验用户是否为参数定义了有效的值[L5.2(1)]。与

OSTimeDly()一样，即使用户没有定义延时，OSTimeDlyHMSM()也是存在的[L5.2(9)]。因为μC/OS-Ⅱ只知道节拍，所以节拍总数是从指定的时间中计算出来的[L5.2(3)]。很明显，程序清单 L5.2 中的程序并不是十分有效的。笔者只是用这种方法告诉大家一个公式，这样用户就可以知道怎样计算总的节拍数了。真正有意义的只是 OS_TICKS_PER_SEC。[L5.2(3)]决定了最接近需要延迟的时间的时钟节拍总数。500/OS_TICKS_PER_SECOND 的值基本上与 0.5 个节拍对应的毫秒数相同。例如，若将时钟频率(OS_TICKS_PER_SEC)设置成 100Hz(10ms)，4ms 的延时不会产生任何延时！而 5ms 的延时就等于延时 10ms。

μC/OS-Ⅱ支持的延时最长为 65,535 个节拍。要想支持更长时间的延时，如 L5.2(2)所示，OSTimeDlyHMSM()确定了用户想延时多少次超过 65,535 个节拍的数目[L5.2(4)]和剩下的节拍数[L5.2(5)]。例如，若 OS_TICKS_PER_SEC 的值为 100，用户想延时 15 分钟，则 OSTimeDlyHMSM()会延时 15x60x100=90,000 个时钟。这个延时会被分割成两次 32,768 个节拍的延时(因为用户只能延时 65,535 个节拍而不是 65536 个节拍)和一次 24,464 个节拍的延时。在这种情况下，OSTimeDlyHMSM()首先考虑剩下的节拍，然后是超过 65,535 的节拍数[L5.2(7)和(8)](即两个 32,768 个节拍延时)。

程序清单 L 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);
        }
        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);
    } else {
        return (OS_TIME_ZERO_DLY);                                 (9)
    }
}
```

由于 OSTimeDlyHMSM()的具体实现方法，用户不能结束延时调用 OSTimeDlyHMSM()要求延时超过 65535 个节拍的任务。换句话说，如果时钟节拍的频率是 100Hz，用户不能让调用 OSTimeDlyHMSM(0，10，55，350)或更长延迟时间的任务结束延时。

5.2 让处在延时期的任务结束延时，OSTimeDlyResume()

μC/OS-Ⅱ允许用户结束延时正处于延时期的任务。延时的任务可以不等待延时期满，而是通过其它任务取消延时来使自己处于就绪态。这可以通过调用 OSTimeDlyResume()和指定要恢复的任务的优先级来完成。实际上，OSTimeDlyResume()也可以唤醒正在等待事件(参看第六章——任务间的通讯和同步)的任务，虽然这一点并没有提到过。在这种情况下，等待事件发生的任务会考虑是否终止等待事件。

OSTimeDlyResume()的代码如程序清单 L5.3 所示，它首先要确保指定的任务优先级有效 [L5.3(1)]。接着，OSTimeDlyResume()要确认要结束延时的任务是确实存在的[L5.3(2)]。如果任务存在，OSTimeDlyResume()会检验任务是否在等待延时期满[L5.3(3)]。只要 OS_TCB 域中的 OSTCBDly 包含非 0 值就表明任务正在等待延时期满，因为任务调用了 OSTimeDly()，OSTimeDlyHMSM()或其它在第六章中所描述的 PEND 函数。然后延时就可以通过强制命令 OSTCBDly 为 0 来取消[L5.3(4)]。延时的任务有可能已被挂起了，这样的话，任务只有在没有被挂起的情况下才能处于就绪状态[L5.3(5)]。当上面的条件都满足后，任务就会被放在就绪表中[L5.3(6)]。这时，OSTimeDlyResume()会调用任务调度程序来看被恢复的任务是否拥有比当前任务更高的优先级[L5.3(7)]。这会导致任务的切换。

程序清单 L 5.3 恢复正在延时的任务

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

```
{
    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)) {                  (5)
                OSRdyGrp             |= ptcb->OSTCBBitY;                 (6)
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                OS_EXIT_CRITICAL();
                OSSched();                                               (7)
            } else {
                OS_EXIT_CRITICAL();
            }
            return (OS_NO_ERR);
        } else {
            OS_EXIT_CRITICAL();
            return (OS_TIME_NOT_DLY);
        }
    } else {
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);
    }
}
```

注意，用户的任务有可能是通过暂时等待信号量、邮箱或消息队列来延时自己的(参看第六章)。可以简单地通过控制信号量、邮箱或消息队列来恢复这样的任务。这种情况存在的唯一问题是它要求用户分配事件控制块(参看 6.00),因此用户的应用程序会多占用一些RAM。

5.3 系统时间，OSTimeGet()和 OSTimeSet()

无论时钟节拍何时发生，μC/OS-Ⅱ都会将一个 32 位的计数器加 1。这个计数器在用户调用 OSStart()初始化多任务和 4,294,967,295 个节拍执行完一遍的时候从 0 开始计数。在时钟节拍的频率等于 100Hz 的时候，这个 32 位的计数器每隔 497 天就重新开始计数。用户可以通过调用 OSTimeGet()来获得该计数器的当前值。也可以通过调用 OSTimeSet()来改变该计数器的值。OSTimeGet()和 OSTimeSet()两个函数的代码如程序清单 L5.4 所示。注意，在访问 OSTime 的时候中断是关掉的。这是因为在大多数 8 位处理器上增加和拷贝一个 32 位的数都需要数条指令，这些指令一般都需要一次执行完毕，而不能被中断等因素打断。

程序清单 L 5.4 得到和改变系统时间

```
INT32U OSTimeGet (void)
{
    INT32U ticks;


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


void OSTimeSet (INT32U ticks)
{
    OS_ENTER_CRITICAL();
    OSTime = ticks;
    OS_EXIT_CRITICAL();
}
```