第 3 章

CHAPTER 3

μC/OS-Ⅲ中断、时钟和时间管理

本章讲述 μC/OS-Ⅲ操作系统中的中断管理、时钟管理以及时间管理机制。在实时嵌入式系统中，中断管理机制是保证系统能够对外部或内部事件迅速响应的关键环节；时钟管理机制则通过系统时钟节拍中断驱动时钟节拍任务，实现任务切换与定时功能，为多任务调度提供精确的时间基准；时间管理机制负责任务的延时、唤醒以及系统时间的设置与获取等操作。在 μC/OS-Ⅲ中，这三大机制相互协作，不仅确保了系统的高响应性和实时性，也为复杂任务调度和时间敏感的数据处理提供了可靠保障。

3.1 μC/OS-Ⅲ的中断管理

要理解 μC/OS-Ⅲ的中断管理机制，需要对其整体中断处理过程有一个清晰的认识，这有助于更好地理解系统如何在保持高效性的同时，确保任务的正确执行和系统的稳定性。

3.1.1 μC/OS-Ⅲ中断处理过程

ARM Cortex 系列微处理器是支持中断处理的，中断是一个硬件机制，主要用来向 CPU 通知一个异步事件发生了，这时 CPU 就会将当前 CPU 寄存器值入栈，然后转而去执行中断服务程序，在 CPU 执行中断服务程序时，有可能有更高优先级的任务就绪，那么当中断服务程序执行结束时，CPU 就会直接执行这个就绪的更高优先级的任务。

μC/OS-Ⅲ是支持中断嵌套的，即高优先级的中断可以打断低优先级的中断，在 μC/OS-Ⅲ中使用全局变量中断嵌套计数器(OSIntNestingCtr)来记录中断嵌套层数，最大支持 250 级的中断嵌套，每进入一次中断服务程序，OSIntNestingCtr 就会加 1，当退出中断服务程序时，OSIntNestingCtr 就会减 1。

在编写 μC/OS-Ⅲ的中断服务程序时，需要使用到两个函数：OSIntEnter()和 OSIntExit()。

1. OSIntEnter()函数

OSIntEnter()函数通常被称为进入中断服务函数，它通常位于中断服务程序的开头。该函数的主要作用包括以下几方面。

1) 通知操作系统中断处理开始

这个函数向操作系统指示一个中断处理正在开始。

2）递增中断嵌套层数

OSIntEnter()函数会递增中断嵌套计数器(OSIntNestingCtr)，将 OSIntNestingCtr 加1，OSIntNestingCtr 用来记录中断嵌套的层数。当 OSIntNestingCtr 不为 0 时，意味着有中断正在处理。中断嵌套发生在一个中断处理程序运行时，第二个中断请求被触发，并且第二个中断具有更高的优先级，从而暂停当前的中断处理。也就是说，OSIntEnter()函数可以管理多个中断同时发生的情况。与 OSIntEnter()函数配对使用的是 OSIntExit()函数，这两个函数共同确保操作系统在中断处理的开始和结束时都能正确地管理中断的状态。这对于保持系统的稳定性和效率至关重要。

3）防止任务调度

在执行中断服务程序期间，系统不应进行任务切换。通过递增中断嵌套计数器，OSIntEnter()函数确保了在 ISR 执行期间操作系统不会进行任务调度。

在 μC/OS-Ⅲ的源码文件中，OSIntEnter()函数定义在 os_core.c 文件中，其代码和关键句的解释说明如下：

```
void OSIntEnter (void)
{
    if (OSRunning != OS_STATE_OS_RUNNING) {                          (1)
      return;                                }

    if (OSIntNestingCtr >= (OS_NESTING_CTR)250u) {                   (2)
        return;                              }

    OSIntNestingCtr++;                                               (3)
}
```

(1) 判断 μC/OS-Ⅲ是否运行，如果 μC/OS-Ⅲ未运行的话就直接跳出 OSIntEnter()。

(2) 用于判断当前的中断嵌套层数是否已经达到系统允许的最大值。μC/OS-Ⅲ最大支持 250 层中断嵌套，如果 OSIntNestingCtr 大于或等于 250，则直接跳出 OSIntEnter()。

(3) 将中断嵌套计数器加 1，用于记录当前系统处于第几层中断中。这一计数对于中断退出判断、任务调度等操作至关重要，确保操作系统能够正确管理中断嵌套和响应。

对于某些特定的 CPU 架构，OSIntEnter()函数需要用汇编语言来实现，以满足性能和效率的需求。这是因为汇编语言允许程序直接与硬件交互，可以精确控制处理器的行为。

2. OSIntExit()函数

OSIntExit()函数通常被称为退出中断服务函数，它通常位于中断服务程序的末尾。该函数的主要作用包括以下几方面。

1）通知操作系统中断处理完成

这个函数向操作系统指示一个中断服务已经处理完毕，准备回到普通的任务调度。

2）递减中断嵌套层数

OSIntExit()函数会递减中断嵌套计数器(OSIntNestingCtr)，将 OSIntNestingCtr 减1，中断嵌套计数器记录当前系统中正在处理的中断层次，当这个计数器降至零时，意味着所有嵌套中断都已经处理完毕。当所有中断处理完毕(即计数器为零)时，操作系统恢复正常任务调度。

3）触发任务调度

通过 OSIntExit()，操作系统会检查在中断期间是否有任何更高优先级的任务变得就

绪，如果有，操作系统会在退出中断时进行相应的任务切换；否则，继续执行被中断的任务。

因为 OSIntExit()函数在中断处理完成后负责检查并执行任务调度，确保高优先级任务能够及时得到执行，因此，OSIntExit()函数也被称为中断级任务调度器。

在 μC/OS-Ⅲ的源码文件中，OSIntExit()函数定义在 os_core. c 文件中，它的代码和关键句的解释说明如下：

```
void OSIntExit (void)
{
CPU_SR_ALLOC( );
if (OSRunning != OS_STATE_OS_RUNNING) {                                    (1)
return;
}
CPU_INT_DIS( );
if (OSIntNestingCtr == (OS_NESTING_CTR)0) {                                (2)
CPU_INT_EN( );
return;
}
OSIntNestingCtr -- ;                                                       (3)
if (OSIntNestingCtr > (OS_NESTING_CTR)0) {                                 (4)
CPU_INT_EN( );
return;
}
if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {                           (5)
CPU_INT_EN( );
return;
}
OSPrioHighRdy = OS_PrioGetHighest( );                                      (6)
OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
if (OSTCBHighRdyPtr == OSTCBCurPtr) {
CPU_INT_EN( );
return;
}
# if OS_CFG_TASK_PROFILE_EN > 0u
OSTCBHighRdyPtr - > CtxSwCtr++;
# endif
OSTaskCtxSwCtr++;
# if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
OS_TLS_TaskSw( );
# endif
OSIntCtxSw( );                                                             (7)
CPU_INT_EN( );
}
```

(1) 判断 μC/OS-Ⅲ操作系统是否运行，如果 μC/OS-Ⅲ操作系统当前的状态不是运行状态，则直接返回，不执行后续的代码。

(2) OSIntNestingCtr 为中断嵌套计数器，这里检查 OSIntNestingCtr 是否为 0，以确保在退出中断服务程序时调用 OSIntExit()后 OSIntNestingCtr 不会等于负数；如果 OSIntNestingCtr 等于 0，说明没有嵌套的中断，那么重新启用 CPU 的中断并返回，不执行后续的代码。

(3) 如果还有其他未完成的中断嵌套，则 OSIntNestingCtr 减 1。因为 OSIntExit()是

在退出中断服务程序时调用的，因此中断嵌套计数器要减1；然后重新检查是否还有未完成的中断嵌套。

(4) 如果还有未完成的中断嵌套，即OSIntNestingCtr还大于0，那么重新启用CPU的中断并返回；不需要做任务切换。

(5) 检查调度器是否被加锁，如果调度器被加锁，那么OSIntExit()不会启动调度器，而是重新启用CPU的中断并返回。

(6) 接下来的5行程序和任务级调度器OSSechd()是一样的；如果这是最后一层嵌套的中断(再返回即到任务级)，并且调度器没有上锁，那么将找出就绪的最高优先级的任务；从OSRdyList[]中取出最高优先级任务的任务控制块OS_TCB；如果就绪的优先级最高的任务是当前任务，就重新启用CPU的中断并直接返回，不执行后续的代码，不需要做任务切换。

(7) 如果就绪的优先级最高的任务并不是当前任务，那么μC/OS-Ⅲ将调用中断级任务切换函数OSIntCtxSW()，进行一次中断级的任务切换。中断级的任务切换和任务级的任务切换不同，因为被中断的任务的寄存器已经在进入中断服务程序时保存了，程序需要做的仅是恢复新任务的现场并执行。

在中断级任务调度器中真正完成任务切换的就是中断级任务切换函数OSIntCtxSW()，与任务级切换函数OSCtxSW()不同的是，由于进入中断时现场已经保存过，所以OSIntCtxSW()不需要像OSCtxSW()一样先保存当前任务现场，只需要做OSCtxSW()的后半部分工作，也就是从将要执行的任务堆栈中恢复CPU寄存器的值。

从上面代码中，我们可以看出该代码中的OSIntExit()函数的作用如下。

(1) 把全局变量OSIntNestingCtr减1，从而用它来记录中断嵌套的层数。

(2) 在中断嵌套层数计数器OSIntNestingCtr为0，调度器未被锁定(即OSSchedLock NestingCtr=0)的情况下：从任务就绪表OSRdyList中，获取最高就绪任务；若其(获取的最高就绪任务)不是被中断的任务，则进行任务切换，否则重新启用CPU的中断并返回。

那么在μC/OS-Ⅲ环境中如何编写基于μC/OS-Ⅲ的中断服务程序呢？我们按照下面所示代码编写中断服务程序：

```
void XXX_Handler(void)                                                    (1)
{
  OSIntEnter();                    //进入中断                             (2)
  用户自行编写的中断服务程序;      //这部分就是我们的中断服务程序         (3)
  OSIntExit();                     //退出中断,并发起一次中断级任务切换    (4)
}
```

(1) 定义中断服务程序名。若中断为内核中断，其中断服务程序名为XXX_Handler(如NMI_Handler(void)等)，若中断为外部中断通道，其中断服务程序名为XXX_IRQHandler(如USART1_IRQHandler(void)等)。中断服务程序名是严格按照startup_stm32f10x_hd.s文件中的中断向量表对应的向量名来命名，两者必须一致。

(2) 调用OSIntEnter()函数，来标记进入中断服务程序，并且记录中断嵌套次数。

(3) 这部分就是用户自行编写的中断服务程序。

(4) 退出中断服务程序时，调用OSIntExit()，发起一次中断级任务切换。

3.1.2 直接发布和延迟发布

中断服务程序可以通过调用 OSSemPost()、OSTaskSemPost()、OSMutexPost()、OSFlagPost()、OSQPost()或 OSTaskQPost()等发布函数向任务发送信号或消息等。从中断向任务发布信号或消息的模式，μC/OS-Ⅱ采用的是直接发布模式，μC/OS-Ⅲ在保留了直接发布模式的基础上又新增了延迟发布模式。

我们可以通过宏 OS_CFG_ISR_POST_DEFERRED_EN 选择采用直接发布或是延迟发布的模式。当该宏的值为 1 时，系统启用中断服务管理任务(采用延迟发布模式)；当其值为 0 时，系统不启用该任务(采用直接发布模式)。需要注意的是，OS_CFG_ISR_POST_DEFERRED_EN 实际上是通过控制中断服务管理任务的启用状态，从而间接决定了事件的发布模式，而不是直接指定发布模式。宏 OS_CFG_ISR_POST_DEFERRED_EN 定义在 os_cfg.h 文件中。

1. 直接发布

在 μC/OS-Ⅲ中，当设置宏 OS_CFG_ISR_POST_DEFERRED_EN 为 0 时，从中断向任务发布信号或消息的模式，采用的是直接发布模式(也称常规发布模式)。

直接发布模式，顾名思义，就是直接从中断服务程序中向任务发布信号或消息。直接发布模式示意图如图 3-1 所示，下面对其进行详细地解释说明。

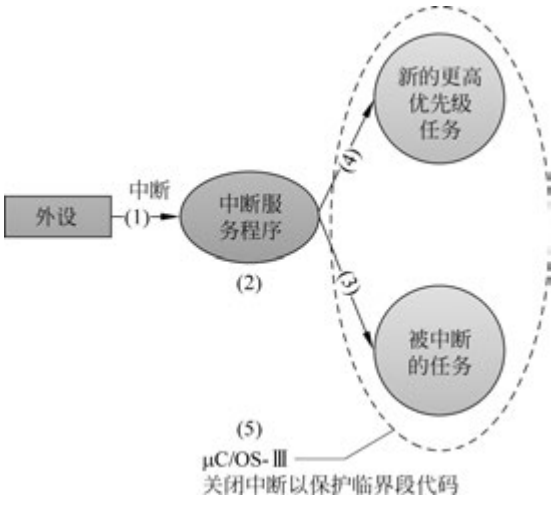

图 3-1 直接发布模式示意图

(1) 外设产生中断请求：一个外设产生中断请求。

(2) CPU 响应中断请求：CPU 响应该外设中断请求，挂起当前运行的任务，转去执行该外设对应的中断服务程序(假设中断是允许的)，该中断服务程序中包含任务正在等待的事件，等待这个事件的任务优先级要么比当前被中断的任务高，要么比其低或相同。

(3) 事件触发低优先级或相同优先级任务：如果中断服务程序中包含的事件使得比被中断的任务优先级低(或相同)的任务进入就绪态，则中断退出后仍恢复运行被中断的任务。

(4) 事件触发高优先级任务：如果中断服务程序中包含的事件使得比被中断的任务优

先级更高的任务进入就绪态，则μC/OS-Ⅲ将进行任务切换，中断服务程序退出后就执行更高优先级的任务。

(5) 直接发布模式中的临界段保护：如果使用直接发布模式，则μC/OS-Ⅲ必须关闭中断以保护临界段代码，防止中断处理程序访问这些临界段代码。

直接发布模式通过关闭中断来保护临界段代码，这样会延长中断的响应时间。虽然μC/OS-Ⅲ已经采用了所有可能的措施来降低中断关闭时间，但仍然有一些复杂的功能会使得中断关闭相对较长的时间。

2. 延迟发布

在μC/OS-Ⅲ中，当设置宏 OS_CFG_ISR_POST_DEFERRED_EN 为 1 时，从中断向任务发布信号或消息的模式，采用的是延迟发布模式。

延迟发布模式，顾名思义，就是延迟从中断服务程序中向任务发布信号或消息。它是先将发布函数和相应的参数写入一个专用的队列中，然后由中断队列处理任务进行发布信号或消息。延迟发布模式示意图如图 3-2 所示，对其详细的解释说明如下。

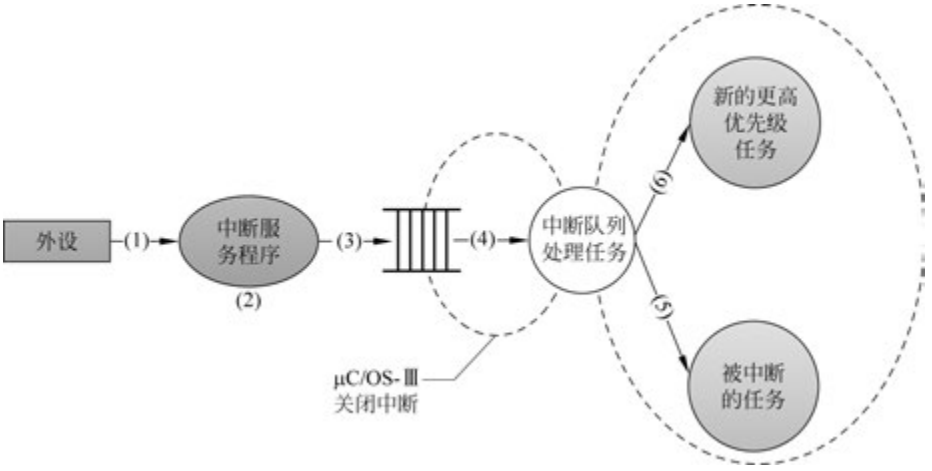

图 3-2 延迟发布模式示意图

(1) 外设产生中断请求：一个外设产生中断请求。

(2) CPU 响应中断请求：CPU 响应该外设中断请求，挂起当前运行的任务，转去执行该外设对应的中断服务程序(假设中断是允许的)，若该中断服务程序中包含任务正在等待的事件，等待这个事件的任务的优先级要么比当前被中断的任务高，要么比其低或相同。

(3) 将发布服务函数写入中断队列中：中断服务程序通过调用系统的发布服务函数(简称发布函数)向任务发布信号或消息，在延迟发布模式下，这个过程不是直接进行发布操作，而是先将这个发布函数和相应的参数写入专用队列(该队列称为中断队列)中，然后使中断队列处理任务(即中断服务管理任务)进入就绪态，这个任务是μC/OS-Ⅲ的内部任务，并且具有最高优先级 0。

(4) 处理中断队列和执行发布函数：当中断服务程序处理结束时，μC/OS-Ⅲ切换执行中断队列处理任务，该任务从中断队列中提取出发布函数，此时仍需要关闭中断以防止其他中断服务程序同时对中断队列进行访问；中断队列处理任务提取出发布函数后重新开中断，并给任务调度器上锁，禁止任务调度，然后执行发布函数发布信号或消息给相应的任务，

相当于发布函数的提取和执行一直是在任务级代码中进行的，这样本来应该在中断服务程序中处理的代码就被放到了任务级完成。

(5) 恢复任务调度：中断队列处理任务将中断队列处理完，将自身挂起，并给调度器解锁，重新启动任务调度来运行处于最高优先级的就绪任务；若原先被中断的任务仍然是最高优先级的就绪任务，则 μC/OS-Ⅲ恢复运行这个任务。

(6) 任务切换：若由于执行中断队列处理任务的发布函数使得更高优先级的任务进入就绪态，内核将切换到更高优先级的任务运行。

延迟发布模式额外增加的操作都是为了避免使用关闭中断。这些额外增加的操作仅包括将发布函数及其参数复制到中断队列中、从中断队列提取发布函数及其参数以及一次额外的任务切换。采用延迟发布模式的好处，就是可以减少中断关闭的时间，进而缩短中断的响应时间。延迟发布模式不是通过关闭中断，而是通过给任务调度器上锁的方法来保护临界段代码。

为何说做额外的延迟操作可以减少中断关闭的时间，详见第 2 章 2.1.5 节“系统任务”中的第 5 项“中断服务管理任务”。

3. 直接发布和延迟发布的对比

(1) 在直接发布模式下，μC/OS-Ⅲ通过关闭中断来保护临界段代码。

(2) 在延迟发布模式下，μC/OS-Ⅲ通过锁定任务调度来保护临界段代码；此模式下，μC/OS-Ⅲ在访问中断队列时，仍然需要关闭中断，但这个时间是非常短的。

如果应用中存在非常快速的中断请求源，则当 μC/OS-Ⅲ在直接发布模式下的中断关闭时间不能满足要求时，可以使用延迟发布模式来降低中断关闭时间。

3.1.3 μC/OS-Ⅲ的临界段代码保护

临界段代码，又称临界区，是指那些必须完整连续运行，不可被打断的代码段。在 μC/OS-Ⅲ中，使用宏 OS_CRITICAL_ENTER()进入临界区；使用宏 OS_CRITICAL_EXIT()或 OS_CRITICAL_EXIT_NO_SCHED()退出临界区。

宏 OS_CRITICAL_EXIT()和宏 OS_CRITICAL_EXIT_NO_SCHED()的区别在于：宏 OS_CRITICAL_EXIT()使 OSSchedLockNestingCtr(调度器被锁住的嵌套层数)减 1，并且当此值达到 0 时，调用调度器；宏 OS_CRITICAL_EXIT_NO_SCHED()也能使 OSSchedLockNestingCtr 减 1，但是当此值为 0 时，也不调用调度器。

μC/OS-Ⅲ可以使用两种方式保护临界段代码：关中断或锁定调度器。相关设置如下：

(1) 当宏 OS_CFG_ISR_POST_DEFERRED_EN 被设置为 0 时，使用关中断的方式来保护临界段代码，具体来说，μC/OS-Ⅲ会在进入临界段代码前关闭中断，并在退出临界段代码后重新打开中断。

(2) 当宏 OS_CFG_ISR_POST_DEFERRED_EN 被设置为 1 时，使用锁定调度器的方式来保护临界段代码，具体来说，μC/OS-Ⅲ会在进入临界段代码前给调度器上锁，并在退出临界段代码后给调度器解锁，即重新允许调度。

在 μC/OS-Ⅲ中存在大量的临界段代码。中断处理程序和任务都会访问的临界段代码，需要使用关中断的方法加以保护；仅由任务访问的临界段代码，可以通过给调度器上锁的方法来保护。

进入和退出临界区的宏定义在 os.h 文件中,示例代码如下:

```
//延迟发布模式
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u     /* 延迟发布模式 */
//进入临界区
#define  OS_CRITICAL_ENTER()
          do {
              CPU_CRITICAL_ENTER();
              OSSchedLockNestingCtr++;
              if (OSSchedLockNestingCtr == 1u) {
                  OS_SCHED_LOCK_TIME_MEAS_START();
              }
              CPU_CRITICAL_EXIT();
          } while (0)
//退出临界区
#define  OS_CRITICAL_EXIT()
          do {
              CPU_CRITICAL_ENTER();
              OSSchedLockNestingCtr--;
              if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {
                  OS_SCHED_LOCK_TIME_MEAS_STOP();
                  if (OSIntQNbrEntries > (OS_OBJ_QTY)0) {
                      CPU_CRITICAL_EXIT();
                      OS_Sched0();
                  } else {
                      CPU_CRITICAL_EXIT();
                  }
              } else {
                  CPU_CRITICAL_EXIT();
              }
          } while (0)

//退出临界区,禁止任务调度
#define  OS_CRITICAL_EXIT_NO_SCHED()
          do {
              CPU_CRITICAL_ENTER();
              OSSchedLockNestingCtr--;
              if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {
                  OS_SCHED_LOCK_TIME_MEAS_STOP();
              }
              CPU_CRITICAL_EXIT();
          } while (0)
```

如以上代码所示,当宏 OS_CFG_ISR_POST_DEFERRED_EN 被设置为 1 时,系统采用延迟发布模式,使用锁定调度器的方式来保护临界段代码。宏 OS_CRITICAL_ENTER()递增全局变量 OSSchedLockNestingCtr(调度器被锁住的嵌套层数),如果该变量的值不为零,则表明调度器已经被锁定;宏 OS_CRITICAL_EXIT()递减全局变量 OSSchedLockNestingCtr,当该变量递减到零时,会调用调度函数进行任务调度;宏 OS_CRITICAL_EXIT_NO_SCHED()也递减全局变量 OSSchedLockNestingCtr,但当该变量递减到零时,不调用调度函数。

```
//直接发布模式
#else              /* 即 OS_CFG_ISR_POST_DEFERRED_ED 被设置为 0: 直接发布模式 */
//进入临界区
```

```
#define OS_CRITICAL_ENTER()                    CPU_CRITICAL_ENTER()
//退出临界代码段
#define OS_CRITICAL_EXIT()                     CPU_CRITICAL_EXIT()
//退出临界代码段,禁止任务调度
#define OS_CRITICAL_EXIT_NO_SCHED()            CPU_CRITICAL_EXIT()
#endif
```

如以上代码所示，当宏 OS_CFG_ISR_POST_DEFERRED_EN 被设置为 0 时，系统采用直接发布模式，使用关中断的方式来保护临界段代码。这里使用的三个宏 OS_CRITICAL_ENTER()、OS_CRITICAL_EXIT()和 OS_CRITICAL_EXIT_NO_SCHED()不同于延迟发布模式里采用的三个宏，宏 OS_CRITICAL_ENTER()调用了另一个宏 CPU_CRITICAL_ENTER()，最终又调用了宏 CPU_SR_Save()，宏 OS_CRITICAL_EXIT()及 OS_CRITICAL_EXIT_NO_SCHED()都调用了另一个宏 CPU_CRITICAL_EXIT()，并最终调用了宏 CPU_SR_Restore()。

宏 CPU_SR_Save()定义在 cpu_a.asm 文件中，通常用汇编语言编写，它首先暂存 CPU 状态寄存器的当前状态，接着关闭所有可屏蔽的中断。宏 CPU_SR_Save()的返回值(即暂存的 CPU 状态寄存器的当前状态)保存在局部变量 cpu_sr 中，而该变量是在其任务栈中动态分配的。

宏 CPU_SR_Restore()定义在 cpu_a.asm 文件中，通常也使用汇编语言编写，它通过将局部变量 cpu_sr 的内容复制到 CPU 状态寄存器中，将 CPU 状态寄存器恢复到调用 OS_CRITICAL_ENTER()之前的状态。

3.2 μC/OS-Ⅲ的时钟管理

3.2.1 μC/OS-Ⅲ的系统时钟、系统节拍、时钟节拍

操作系统通常需要底层平台提供一个周期性的定时信号，驱动操作系统运行，用于系统定时、延时、超时判断和时间片轮转调度等，这个周期性的定时信号，被称为系统时钟、系统节拍或时钟节拍(Clock Tick)。

为了产生时钟节拍，通常需要使用一个硬件定时器，对于基于 ARM Cortex 系列微处理器来说，硬件定时器就是 SysTick 定时器。从 ARM Cortex-M0 内核开始，一直到现在的 ARM Cortex-H7，这些内核中都搭载了 SysTick 定时器，它放在了 NVIC 中，主要目的是给操作系统提供一个滴答中断。SysTick 定时器又称为 SystemTick 定时器，它的中文全称为系统滴答定时器、系统节拍定时器或时钟节拍定时器(SysTick Timer，STK)。

硬件定时器以时钟节拍的定时周期为周期，定时地产生中断，该中断称为时钟节拍中断，该中断的中断服务程序叫作时钟节拍中断的中断服务程序，用 OS_CPU_SysTickHandler()或 OSTickISR()表示。该中断服务程序通过调用时钟节拍函数 OSTimeTick()来完成系统在每个时钟节拍时需要做的工作。

系统时钟可以视为系统的心跳，系统时钟的频率越高，系统的额外开销就越大，其频率取决于应用所需要的定时精度。

3.2.2 μC/OS-Ⅲ的系统时钟节拍中断的中断服务程序

μC/OS-Ⅲ的系统时钟节拍中断的中断服务程序 OS_CPU_SysTickHandler()的源代码如下：

```
void OS_CPU_SysTickHandler (void)
{   CPU_SR_ALLOC();

    CPU_CRITICAL_ENTER();
    OSIntNestingCtr++;              //中断嵌套层数加 1
    CPU_CRITICAL_EXIT();

    OSTimeTick();
    OSIntExit();
}
```

从以上代码可知，时钟节拍中断的中断服务程序通过调用时钟节拍（服务）函数 OSTimeTick()来完成系统在每个时钟节拍时需要做的工作。OSTimeTick()的源代码如下：

```
void OSTimeTick (void)
{
    OS_ERR err;
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    CPU_TS ts;
#endif

    OSTimeTickHook();                                   /* Call user definable hook */

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    ts = OS_TS_GET();                                   /* Get timestamp */
    OS_IntQPost((OS_OBJ_TYPE) OS_OBJ_TYPE_TICK,         /* Post to ISR queue */
                (void       *)&OSRdyList[OSPrioCur],
                (void       *) 0,
                (OS_MSG_SIZE)  0u,
                (OS_FLAGS   )  0u,
                (OS_OPT     )  0u,
                (CPU_TS     )  ts,
                (OS_ERR    *)&err);

#else
   (void)OSTaskSemPost((OS_TCB *)&OSTickTaskTCB, /* Signal tick task */
                       (OS_OPT ) OS_OPT_POST_NONE,
                       (OS_ERR *)&err);
#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
    OS_SchedRoundRobin(&OSRdyList[OSPrioCur]);
#endif

#if OS_CFG_TMR_EN > 0u
    OSTmrUpdateCtr--;
    if (OSTmrUpdateCtr == (OS_CTR)0u) {
        OSTmrUpdateCtr = OSTmrUpdateCnt;
        OSTaskSemPost((OS_TCB *)&OSTmrTaskTCB, /* Signal timer task */
                      (OS_OPT ) OS_OPT_POST_NONE,
```

```
                    (OS_ERR *)&err);
    }
#endif

#endif
}
```

OSTimeTick()是 μC/OS-Ⅲ中的核心函数,它负责管理任务的延时和超时、时间片轮转调度、所有的软件定时器有序的运行以及触发任务调度。

OSTimeTick()函数在其开头处就调用 OSTimeTickHook()函数,这个函数是一个钩子函数,给用户提供了更大的灵活性,用户不需要修改 OSTimeTick()内核代码程序,只需要在钩子函数中添加代码就可以扩充 OSTimeTick()的功能,并且给了 μC/OS-Ⅲ移植人员机会,使其能够在时钟节拍中断服务刚开始时就做出反应。

OSTimeTick()函数根据宏 OS_CFG_ISR_POST_DEFERRED_EN 是否定义为 1,产生 2 个分支,若该宏定义为 1,即系统采用延迟发布模式,则在每次时钟节拍中断发生时,读取当前的时间戳信息,将发布函数(即 POST 函数)和相应的参数(如当前任务的任务就绪表指针和当前的时间戳)写入中断队列中,延迟向时钟节拍任务发信号的操作,然后使中断队列处理任务进入就绪态,该任务具有最高的优先级(优先级 0)。当所有嵌套的中断服务程序执行结束时,μC/OS-Ⅲ切换执行中断队列处理任务,该任务从中断队列中提取出发布函数和相应的参数,然后调用该发布函数,向时钟节拍任务发信号。当时钟节拍任务接收到这个信号后,由等待状态转变为就绪状态,并进行一次任务切换,执行时钟节拍任务,来更新任务的延时和超时。

若该宏定义为 0,即系统采用直接发布模式,则在每次时钟节拍中断发生时,主要执行以下操作:

(1) 向时钟节拍任务发信号。

将任务信号量发送给时钟节拍任务 OSTickTask(),触发其执行,以更新任务的延时与超时状态,如图 2-4 所示。

(2) 若启用时间片轮转调度。

若宏 OS_CFG_SCHED_ROUND_ROBIN_EN 定义为大于 0 的值,表示启用了时间片轮转调度,则通过调用 OS_SchedRoundRobin(&OSRdyList[OSPrioCur])来对任务就绪表 OSRdyList 中的优先级为 OSPrioCur 的所有就绪任务,实行时间片轮转调度。

(3) 若启用定时器任务。

若宏 OS_CFG_TMR_EN 定义为大于 0 的值,表示在编译的代码中会包括定时器任务的代码,该任务用于管理所有的软件定时器有序运行,则每次时钟节拍中断发生时,OSTimeTick( )函数会将 OSTmrUpdateCtr 减 1,当它减到 0 时,系统才发送信号量给定时器任务,并把 OSTmrUpdateCnt 的值赋予 OSTmrUpdateCtr,形成以 OSTmrUpdateCnt * 时钟节拍周期为时基的定时节奏,如图 2-7 所示。

在 μC/OS-Ⅱ中,时钟节拍服务是在时钟节拍中断服务程序中完成的,每次时钟节拍服务都会遍历整个任务链表,递减所有延时任务的延时计数器。当任务数目较多时,时钟节拍服务处理时间很长,会造成中断延迟时间和任务延迟时间都变得很长,影响系统的实时性。

针对 μC/OS-Ⅱ时钟节拍服务的问题,μCOS-Ⅲ主要做了两点改进:

(1) 用时钟节拍任务 OSTickTask()来做时钟节拍处理,更新任务的延时和超时;

(2) 采用了哈希散列表机制来管理延时任务以及指定了超时时限的等待任务。具体来说,在 μC/OS-Ⅲ中,OSTimeTick()函数不需要遍历任务链表,只是通过 OSTaskSemPost()函数向时钟节拍任务 OSTickTask()发信号。而时钟节拍任务绝大部分时间内都处于等待该信号的状态,每次收到该信号时,时钟节拍任务会恢复运行,其只需要处理时钟节拍轮盘的某个特定表项所指向的任务链表,然后再次进入等待该信号的状态。哈希散列表机制详见第 2 章 2.1.5 节"系统任务"中的第 2 项"时钟节拍任务"的部分。

所以,在 μC/OS-Ⅲ中,时钟节拍服务不再在时钟节拍中断服务程序中完成,而是放到一个时钟节拍任务 OSTickTask()中完成;而且,为了提高时钟节拍的处理速度,采用了哈希散列表机制,用时钟节拍轮盘来管理所有正在延时的任务和指定了超时时限的等待任务。每次时钟节拍服务只需要处理极少数的延时任务,从而大大减少了时钟节拍服务花费的时间,提高了系统的实时性。

3.3 μC/OS-Ⅲ的时间管理

μC/OS-Ⅲ提供了一系列管理时间的函数,归纳见表 3-1,其代码见 os_time.c 文件。

表 3-1 时间管理 API 函数

函 数 名	功 能
OSTimeDly()	任务延时 n 个时钟节拍
OSTimeDlyHMSM()	任务延时指定的时间,采用"时:分:秒.毫秒"方式
OSTimeDlyResume()	取消任务延时
OSTimeSet()	设置当前时钟节拍计数器的值
OSTimeGet()	获取当前时钟节拍计数器的值

为了方便理解与使用,μC/OS-Ⅲ的时间管理函数可分为三类:任务延时函数(OSTimeDly()和 OSTimeDlyHMSM())、取消任务延时函数(OSTimeDlyResume())以及设置和获取系统时间函数(OSTimeSet()和 OSTimeGet())。下面将分别介绍这三类函数。

3.3.1 任务延时

因为 μC/OS-Ⅲ中的任务是一个无限循环,所以为了使高优先级的任务不至于独占 CPU,需要给其他优先级较低的任务获取 CPU 使用权的机会。μC/OS-Ⅲ规定,除了空闲任务之外的所有任务必须在任务体中合适的位置调用系统提供的延时函数,使当前任务的运行延时(即暂停)一段时间,即挂起当前任务一段时间,并进行一次任务调度,以让出 CPU 的使用权,调度到(即切换到)当前优先级最高的处于就绪状态任务。延时的一段时间到,将当前任务恢复为就绪状态,如果恢复之后,当前任务是优先级最高的任务,便运行之。

任务延时函数有两种:OSTimeDly()和 OSTimeDlyHMSM()。

1. OSTimeDly()函数

该函数用于将一个任务延时若干时钟节拍。如果延时时间大于 0,系统将立即进行任务调度。延时时间可以为 0～65 535 个时钟节拍。延时时间为 0 表示不进行延时,函数将立即返回调用者。

延时的具体时间依赖系统每秒有多少个时钟节拍，OSTimeDly()函数原型如下。

```
void OSTimeDly (OS_TICK   dly,
                OS_OPT    opt,
                OS_ERR  * p_err);
```

1）函数的参数

（1）dly：所需的延时长度，以时钟节拍为单位。

（2）opt：该参数用来指定延时模式。其可以为以下内容之一：

OS_OPT_TIME_DLY　　相对模式

OS_OPT_TIME_PERIODIC　周期模式

OS_OPT_TIME_MATCH　　绝对模式

① 相对模式，准确地说，相对当前时钟节拍计数器(OSTickCtr)的值而言，开始计时，延时结束时间为“当前时间＋dly”；即延时时间为 dly，并且只延时一次。

② “周期模式”，用于在任务循环中反复调用 OSTimeDly()。若仅调用 OSTimeDly()一次，只会发生一次任务挂起与唤醒，不会自动进入周期延时。“周期模式”，顾名思义，以用户指定的非零周期 dly 为间隔工作：每当 dly 到时任务被唤醒，并在任务循环中再次调用 OSTimeDly()，从而再过 dly 时间再次唤醒任务，周而复始。延时周期为 dly，延时非一次性。

③ “绝对模式”，顾名思义，用户指定的时间是一个绝对的时间。具体来说，当 OSTickCtr 达到 dly 值时，任务将被唤醒。这里延时时间不一定是 dly，只有当执行 OSTimeDly()函数的时刻，OSTickCtr 等于 0，延时时间才是 dly。并且只延时一次。

归纳一下，延时任务在时钟节拍计数器(OSTickCtr)的值达到其任务控制块中记录的“任务唤醒时刻”(TickCtrMatch，见表 3-2)时，将被唤醒。

表 3-2　延时任务的任务唤醒时刻

Opt 选项	任务唤醒时刻(TickCtrMatch)
OS_OPT_TIME_DLY	OSTickCtr＋dly
OS_OPT_TIME_PERIODIC	OSTCBCurPtr-> TickCtrPrev＋dly
OS_OPT_TIME_MATCH	dly

其中，若 Opt 选项等于 OS_OPT_TIME_PERIODIC(即周期模式)，当调用 OSTimeDly()时，μC/OS-Ⅲ保存当前时间 OSTickCtr 的值到 OSTCBCurPtr-> TickCtrPrev 中，并延时 dly 个时钟节拍，该延时时间一到唤醒延时任务，即每当 OSTickCtr 达到“OSTCBCurPtr-> TickCtrPrev＋ dly”时刻，将唤醒延时任务，并且每当延时时间到，μC/OS-Ⅲ将保存“OSTCBCurPtr-> TickCtrPrev＋dly”值到 OSTCBCurPtr-> TickCtrPrev 中。当下一次调用 OSTimeDly()时，在 OSTCBCurPtr-> TickCtrPrev＋dly 时刻唤醒任务。即使下一次调用时间有偏差，也会尽量“校正”到周期节奏上。周而复始。

在“相对模式”下，当系统负荷较重时有可能延时会少一个节拍，甚至偶尔差多个节拍。在“周期模式”下，任务仍然可能会被推迟执行，但它尽量和预期的“匹配值”同步。因此，推荐使用“周期模式”来实现长时间运行的周期性延时。

（3）* p_err：p_err 是一个指针，指向调用该函数后返回的错误码。

2）函数的返回值

该函数无返回值。

3）用法示例

（1）OSTimeDly(10,OS_OPT_TIME_PERIODIC,&err);//延时模式采用周期模式；周期=10个时钟节拍，每次唤醒任务=上次计划到期点(OSTCBCurPtr-> TickCtrPrev)+10(无抖动)；若仅调用OSTimeDly()一次，只会发生一次10个时钟节拍的任务挂起与唤醒，不会自动进入周期延时。

（2）OSTimeDly(100,OS_OPT_TIME_DLY,&err);//延时100个时钟节拍，延时模式采用相对模式

（3）OSTimeDly(500,OS_OPT_TIME_MATCH,&err);//延时模式采用绝对模式：等待直到OSTickCtr==500(若当前已>=500，通常会立即返回)

（4）

```
while(1)
{
LED1=~LED1;
OSTimeDly(100, OS_OPT_TIME_DLY, &err);
}
```

/*执行效果：每次翻转之间的间隔≈100时钟节拍+每次任务代码执行耗时(例如翻转、其他代码、任务切换等)。*/

（5）

```
while(1)
{
LED1=~LED1;
OSTimeDly(10, OS_OPT_TIME_PERIODIC, &err);
}
```

/*这段循环程序的作用：按固定节拍翻转LED，并用"周期模式"精确对齐时钟节拍，避免因任务执行耗时造成的累计漂移。

执行效果：每次翻转之间的间隔≈10时钟节拍。

在周期模式(OS_OPT_TIME_PERIODIC)下，本次挂起后，下一次唤醒目标时刻=上次"计划到期点"(TickCtrPrev)+10时钟节拍。这意味着唤醒以固定周期(10时钟节拍)推进(无抖动/无漂移)，不会因为代码执行耗时而累计漂移。但若任务执行时间超过10时钟节拍，则会跳过某些节拍，造成丢周期。*/

2. OSTimeDlyHMSM()函数

该函数用于将一个任务延时若干时间。延时的单位是小时、分、秒、毫秒。调用OSTimeDlyHMSM()后，如果延时时间不为0，系统将立即进行任务调度。该函数仅在“相对模式”下工作，从函数被调用的时刻开始延时。

OSTimeDlyHMSM()函数原型如下：

```
void OSTimeDlyHMSM (CPU_INT16U    hours,
                    CPU_INT16U    minutes,
                    CPU_INT16U    seconds,
                    CPU_INT32U    milli,
```

```
OS_OPT          opt,
OS_ERR        * p_err);
```

1）函数的参数

（1）opt：该参数用于同时指定延时模式与参数校验方式两类选项。相比 OSTimeDly()，本函数的 opt 参数保留了 OS_OPT_TIME_DLY 和 OS_OPT_TIME_PERIODIC 两个延时模式选项，并新增 OS_OPT_TIME_HMSM_STRICT 和 OS_OPT_TIME_HMSM_NON_STRICT 两个参数校验选项。两类选项用位或运算符连接。

① 延时模式

OSTimeDlyHMSM()函数的延时均以调用该函数的时刻为基准开始计算（相对延时），延时时间由参数 hours、minutes、seconds、milli 共同指定。其 opt 参数不支持 OS_OPT_TIME_MATCH（绝对模式）选项，只能使用 OS_OPT_TIME_DLY（相对模式）选项，实现单次相对延时，这种模式的基准点并非“固定点”，而是每次调用时把当前调用点作为基准点；OS_OPT_TIME_PERIODIC（周期模式）选项，实现周期性相对延时，这种模式的基准点是“固定点”，把第一次调用的时间点作为基准点，以后每次调用都会按照固定周期执行延时，其实现机制与 OSTimeDly()的周期模式相同（固定基准点＋固定周期），区别仅在于延时单位：OSTimeDlyHMSM()使用时/分/秒/毫秒，而 OSTimeDly()使用时钟节拍。两个选项互斥，不要将 OS_OPT_TIME_DLY| OS_OPT_TIME_PERIODIC 同时设置。OS_OPT_TIME_DLY 宏定义为 0，所以，若 opt 参数没有选择 OS_OPT_TIME_DLY 或 OS_OPT_TIME_PERIODIC，系统默认选 OS_OPT_TIME_DLY。

② 参数校验方式

严格模式（OS_OPT_TIME_HMSM_STRICT，宏值为 0，且为默认选项）表示对传递给该函数的参数有严格限制，具体范围如下：

参数“hours”：0～99

参数“minutes”：0～59

参数“seconds”：0～59

参数“milli”：0～999

若任一参数超出范围，函数将立即返回错误码。

非严格模式（OS_OPT_TIME_HMSM_NON_STRICT）表示对参数的限制放宽，具体范围如下：

参数“hours”：0～999

参数“minutes”：0～9999

参数“seconds”：0～65 535

参数“milli”：0～4 294 967 295

在此范围内均可接受，超限则返回错误码。

因为 OS_OPT_TIME_HMSM_STRICT 宏定义为 0，所以，若这两个选项未选择，系统默认选 OS_OPT_TIME_HMSM_STRICT（严格模式）。

（2）hours：任务要延时的小时数。

当 opt 参数为 OS_OPT_TIME_HMSM_STRICT 时，该值有效范围是 0～99；若超出范围，函数将立即返回错误码。

当opt参数为OS_OPT_TIME_HMSM_NON_STRICT时，该值有效范围是0～999；在此范围内均可接受，超限则返回错误码。

(3) minutes：任务要延时的分钟数。

当opt参数为OS_OPT_TIME_HMSM_STRICT时，该值有效范围是0～59；若超出范围，函数将立即返回错误码。

当opt参数为OS_OPT_TIME_HMSM_NON_STRICT时，该值有效范围是0～9999；在此范围内均可接受，超限则返回错误码。

(4) seconds：任务要延时的秒数。

当opt参数为OS_OPT_TIME_HMSM_STRICT时，该值有效范围是0～59；若超出范围，函数将立即返回错误码。

当opt参数为OS_OPT_TIME_HMSM_NON_STRICT时，该值有效范围是0～65 535；在此范围内均可接受，超限则返回错误码。

(5) milli：任务要延时的毫秒数。

当opt参数为OS_OPT_TIME_HMSM_STRICT时，该值有效范围是0～999；若超出范围，函数将立即返回错误码。

当opt参数为OS_OPT_TIME_HMSM_NON_STRICT时，该值有效范围是0～4 294 967 295。在此范围内均可接受，超限则返回错误码。

请注意，该参数的精度是时钟周期的整数倍。例如，如果时钟频率是100Hz，则时钟周期是10ms，延时4ms的实际效果是没有延时，因为延时将被四舍五入到零。同样道理，如果延时15ms，则实际效果将是20ms。

(6) * p_err：p_err是一个指针，指向调用该函数后返回的错误码。

2) 函数的返回值

该函数无返回值。

3) 用法示例

(1) OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);//延时模式采用周期模式；周期＝1s，每次唤醒任务＝上次计划到期点(OSTCBCurPtr-> TickCtrPrev)＋1s(无抖动)；参数校验模式＝默认严格模式(OS_OPT_TIME_HMSM_STRICT)；若仅调用OSTimeDlyHMSM()一次，只会发生一次1s的任务挂起与唤醒，不会自动进入周期延时。

(2) OSTimeDlyHMSM(0,0,0,500,OS_OPT_TIME_HMSM_STRICT,&err);//延时500ms，延时模式为默认的相对模式，参数校验模式为默认的严格模式

(3) OSTimeDlyHMSM(0,1,1500,1200,OS_OPT_TIME_DLY|OS_OPT_TIME_HMSM_NON_STRICT,&err);//延时1分1500秒1200毫秒，延时模式为默认的相对模式，参数校验模式为非严格模式

(4) OSTimeDlyHMSM(0,0,2,0,0,&err); //延时2s，延时模式为默认的相对模式，参数校验模式为默认的严格模式

(5)

```
while(1)
{
```

```
LED1=~LED1;
OSTimeDlyHMSM(0, 0, 1, 0, OS_OPT_TIME_DLY, &err); //延时 1s,延时模式为默认的相对模式,参数校验模式为默认的严格模式
}
/*这段循环程序的作用:让 LED 以约 1 秒的间隔翻转一次。
执行效果:每次翻转之间的间隔≈1 秒+每次任务代码执行耗时(例如翻转、其他代码、任务切换等)。
在相对模式(OS_OPT_TIME_DLY)下,每次延时的起点=每次调用 OSTimeDlyHMSM()的时刻,因此基准点不是固定的。这意味着,如果任务代码执行耗时较长,实际翻转周期会在 1 秒的基础上额外增加执行时间,从而产生"累计漂移"。*/
(6)
while(1)
{
LED1=~LED1;
OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err); //延时 1s,延时模式为周期模式,参数校验模式为默认的严格模式
}
/*这段循环程序的作用:以固定周期翻转 LED,并用"周期模式"精确对齐 1 秒的时间间隔,避免因任务执行耗时造成的累计漂移。
执行效果:每次翻转之间的间隔≈1 秒。
在周期模式(OS_OPT_TIME_PERIODIC)下,本次挂起后,下一次唤醒目标时刻=上次"计划到期点"(TickCtrPrev)+1 秒。这意味着唤醒以固定周期(1 秒)推进(无抖动/无漂移),不会因为代码执行耗时而累计漂移。但若任务执行时间超过 1 秒,则会跳过某些周期,造成丢周期。*/
```

3.3.2 取消任务延时

因为调用了 OSTimeDly()或者 OSTimeDlyHMSM()函数而进入等待态的任务,可通过在其他任务中调用 OSTimeDlyResume()函数取消任务延时而使其进入就绪状态。

取消任务延时函数仅有一种,即 OSTimeDlyResume()函数。

OSTimeDlyResume()函数原型如下:

```
void OSTimeDlyResume (OS_TCB *p_tcb,
                      OS_ERR *p_err);
```

1) 函数的参数

(1) *p_tcb:p_tcb 是一个指针,指向要被唤醒任务的任务控制块对象。若 p_tcb 为 NULL,则为无效指针,函数不会进行唤醒操作。注意,当前任务本不可能处于延时状态,因此无须唤醒。

(2) *p_err:p_err 是一个指针,指向调用该函数后返回的错误码。

2) 函数的返回值

该函数无返回值。

下面是使用 OSTimeDlyResume()函数的示意性代码：

```
//task1 任务函数
void task1_task(void * p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;
    …
    while(1)
    {
        …
        OSTimeDly(2,
              OS_OPT_TIME_DLY,
              &err)
    }
}

//task2 任务函数
void task2_task(void * p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;
    …
    while(1)
    {
        …
   OSTimeDlyResume (&Task1_TaskTCB,
                    &err);
     …

    }
}
```

注意：延时的任务并不知道它是被其他任务恢复的，会认为是延时结束而得到恢复的。因此，请谨慎使用该函数。

3.3.3 设置和获取系统时间

μC/OS-Ⅲ定义了一个 CPU_INT32U 类型的全局变量 OSTickCtr(即时钟节拍计数器)来记录系统时钟节拍数，在调用 OSInit()时被初始化为 0，以后每发生 1 个时钟节拍，OSTickCtr 加 1。

1. OSTimeSet()函数

该函数用于用户设置系统时间，即用于用户改变当前时钟节拍计数器 OSTickCtr 的值。

OSTimeSet()函数原型如下。

```
void OSTimeSet (OS_TICK   ticks,
                OS_ERR   * p_err);
```

1) 函数的参数

(1) ticks：要设置的系统时钟的值，以时钟节拍为单位。

(2) * p_err：p_err 是一个指针，指向调用该函数后返回的错误码。

2) 函数的返回值

该函数无返回值。

注意：当使用 OSTimeSet()函数时需要谨慎，因为其他任务可能依赖时钟节拍计数器 OSTickCtr 的当前值。

2. OSTimeGet()函数

该函数用于用户获取系统时间，即用于用户获取当前时钟节拍计数器 OSTickCtr 的值。

OSTimeGet()函数原型如下。

```
OS_TICK OSTimeGet (OS_ERR * p_err);
```

1) 函数的参数

* p_err：p_err 是一个指针，指向调用该函数后返回的错误码。

2) 函数的返回值

OSTickCtr 的当前值，以时钟节拍为单位。

习题

1. 请简述 μC/OS-Ⅲ的中断机制？它为什么需要中断？
2. 请写出 μC/OS-Ⅲ中断服务程序的示意性代码。
3. 中断机制与轮询机制的区别是什么？它们分别适用什么场合？
4. 在 μC/OS-Ⅲ中，全局变量 OSIntNestingCtr 的作用是什么？
5. 在 μC/OS-Ⅲ中，全局变量 OSSchedLockNestingCtr 的作用是什么？
6. 在 μC/OS-Ⅲ中，全局变量 OSTickCtr 的作用是什么？
7. μC/OS-Ⅲ的中断服务程序何时返回被中断的任务？何时不返回被中断的任务？
8. 在 μC/OS-Ⅲ中，中断发布信号或者消息的两种模式，分别是直接发布模式和延迟发布模式，它们之间有何区别？如何看待延迟发布模式？
9. 在 μC/OS-Ⅲ中，如何进入和退出临界区？
10. μC/OS-Ⅲ的系统时钟是如何实现的？μC/OS-Ⅲ的系统时钟节拍中断的中断服务程序实现什么功能？
11. 请说明延时函数 OSTimeDly()和 OSTimeDlyHMSM()的区别。
12. 请用思维导图总结本章所学的知识并撰写学习体会。