



The Real-Time Kernel

Timer Management



中断处理

- 中断：由于某种事件的发生而导致程序流程的改变。产生中断的事件称为中断源。
- CPU响应中断的条件：
 - 至少有一个中断源向CPU发出中断信号；
 - 系统允许中断，且对此中断信号未予屏蔽。

中断服务程序ISR

- 中断一旦被识别，CPU会保存部分（或全部）运行上下文（context，即寄存器的值），然后跳转到专门的子程序去处理此次事件，称为中断服务子程序(ISR)。
- μC/OS-II 中，中断服务子程序要用汇编语言来编写，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。

用户ISR的框架

1. 保存全部CPU寄存器的值;
2. 调用OSIntEnter(), 或直接把全局变量OSIntNesting
(中断嵌套层次) 加1;
3. 执行用户代码做中断服务;
4. 调用OSIntExit();
5. 恢复所有CPU寄存器;
6. 执行中断返回指令。

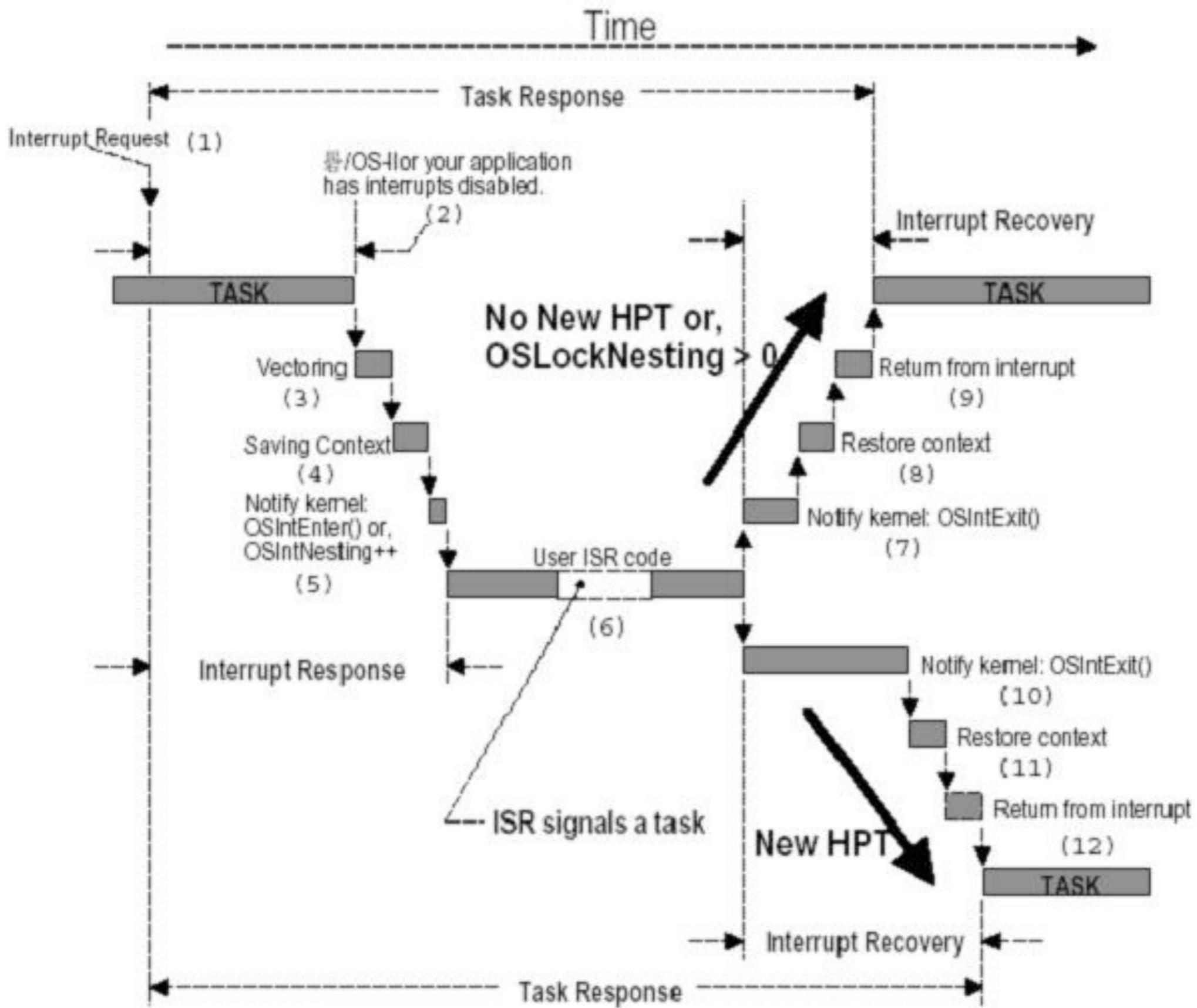


Figure 3-5, Servicing an interrupt

OSIntEnter()

```
/* 在调用本函数之前必须先将中断关闭 */
```

```
void OSIntEnter (void)
```

```
{
```

```
    if (OSRunning == TRUE) {
```

```
        if (OSIntNesting < 255) {
```

```
            OSIntNesting++;
```

```
}
```

```
}
```

```
}
```

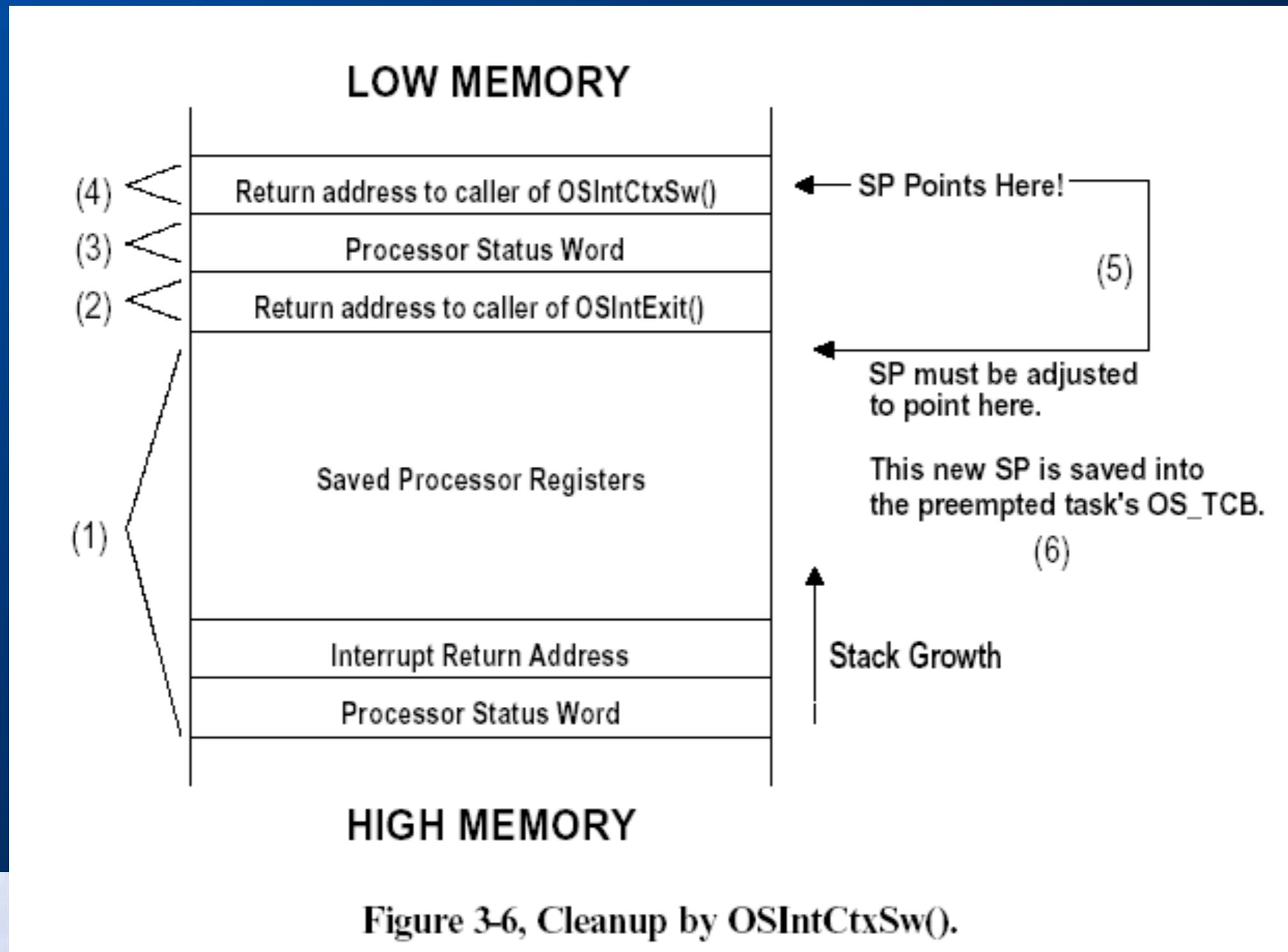
OSIntExit()

```
void OSIntExit (void)
{
    OS_ENTER_CRITICAL(); //关中断
    if ((--OSIntNesting | OSLockNesting) == 0) //判断嵌套是否为零
    { //把高优先级任务装入
        OSIntExitY = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy=(INT8U)((OSIntExitY<< 3) +
            OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OSIntCtxSw();
        }
    }
    OS_EXIT_CRITICAL(); //开中断返回
}
```

OSIntCtxSw()

- 在任务切换时，为什么使用OSIntCtxSw()而不是调度函数中的OS_TASK_SW()？
- 原因如下：
 - 一半的任务切换工作，即CPU寄存器入栈，已经在前面做完了；
 - 需要保证所有被挂起任务的栈结构是一样的。

调用中断切换函数OSIntCtxSw() 后的堆栈情况



时钟节拍

- 时钟节拍是一种特殊的中断；
- 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10到100Hz之间，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，或是来自50/60Hz交流电源的信号。

时钟节拍ISR

```
void OSTickISR(void)
```

```
{
```

- (1)保存处理器寄存器的值;
- (2)调用OSIntEnter()或将OSIntNesting加1;
- (3)调用OSTimeTick(); /*检查每个任务的时间延时*/
- (4)调用OSIntExit();
- (5)恢复处理器寄存器的值;
- (6)执行中断返回指令;

```
}
```

时钟节拍函数 OSTimetick()

```
void OSTimeTick (void)
{
    OS_TCB *ptcb;
    OSTimeTickHook();                                         (1)
    ptcb = OSTCBLList;                                       (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++;
    OS_EXIT_CRITICAL();                                    (7)
}
```

时间管理

- 与时间管理相关的系统服务：
 - OSTimeDLY()
 - OSTimeDLYHMSM()
 - OSTimeDlyResume()
 - OSTimeGet()
 - OSTimeSet()

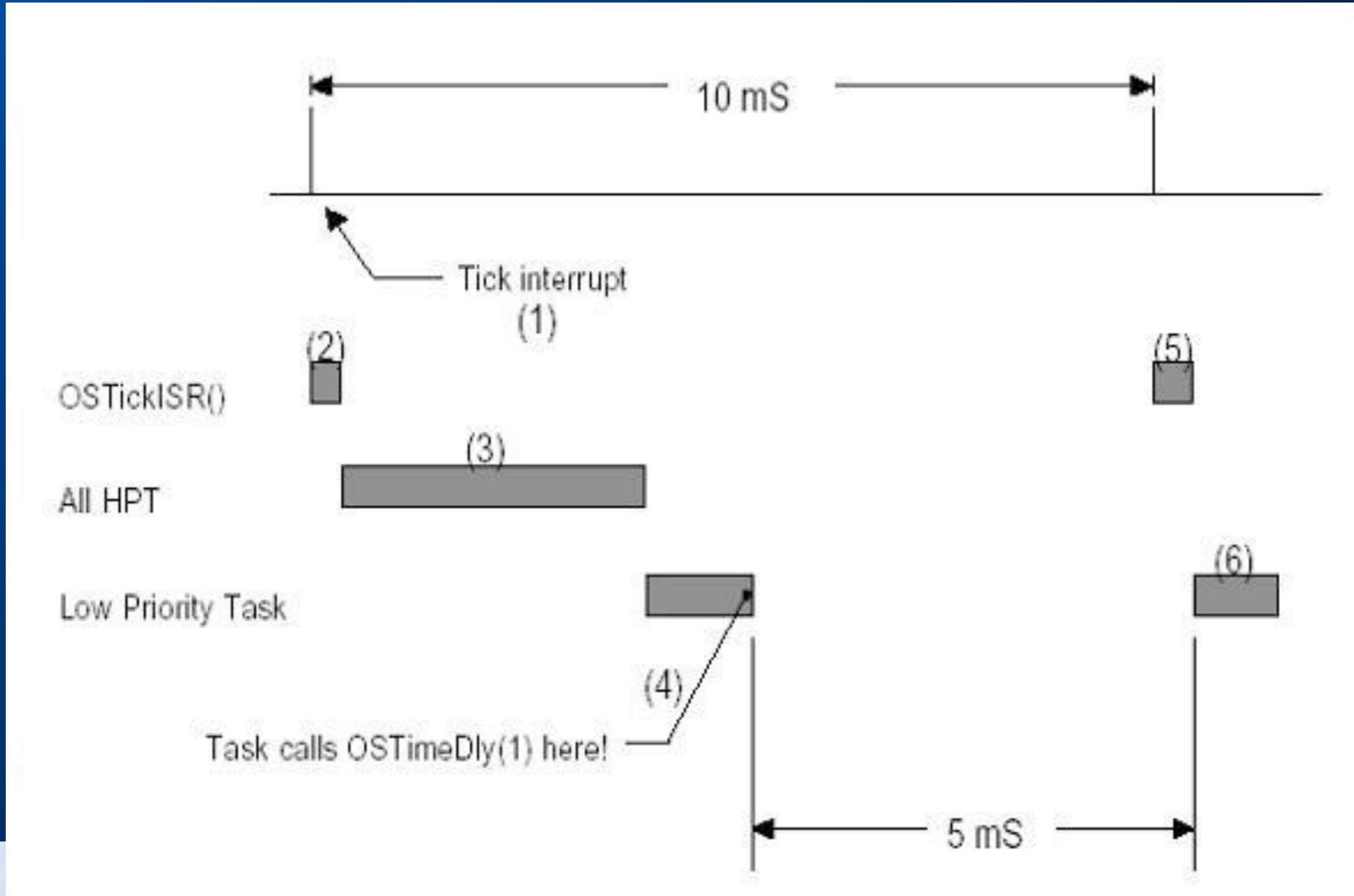
OSTimeDLY()

- OSTimeDLY(): 任务延时函数，申请该服务的任务可以延时一段时间；
- 调用OSTimeDLY后，任务进入等待状态；
- 使用方法
 - void OSTimeDly (INT16U ticks);
 - ticks表示需要延时的时间长度，用时钟节拍的个数来表示。

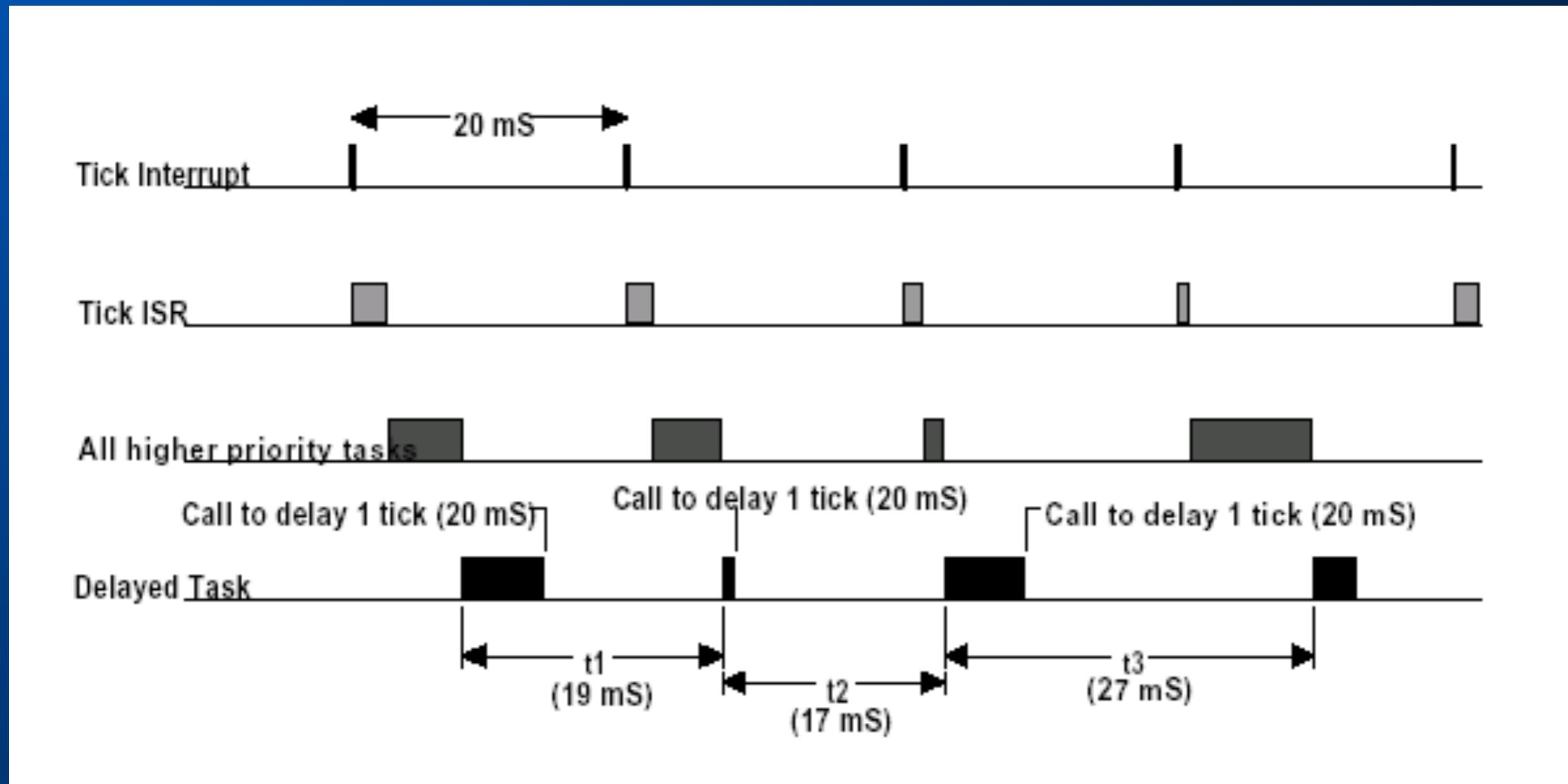
OSTimeDLY()

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

OSTimeDLY(1)的问题

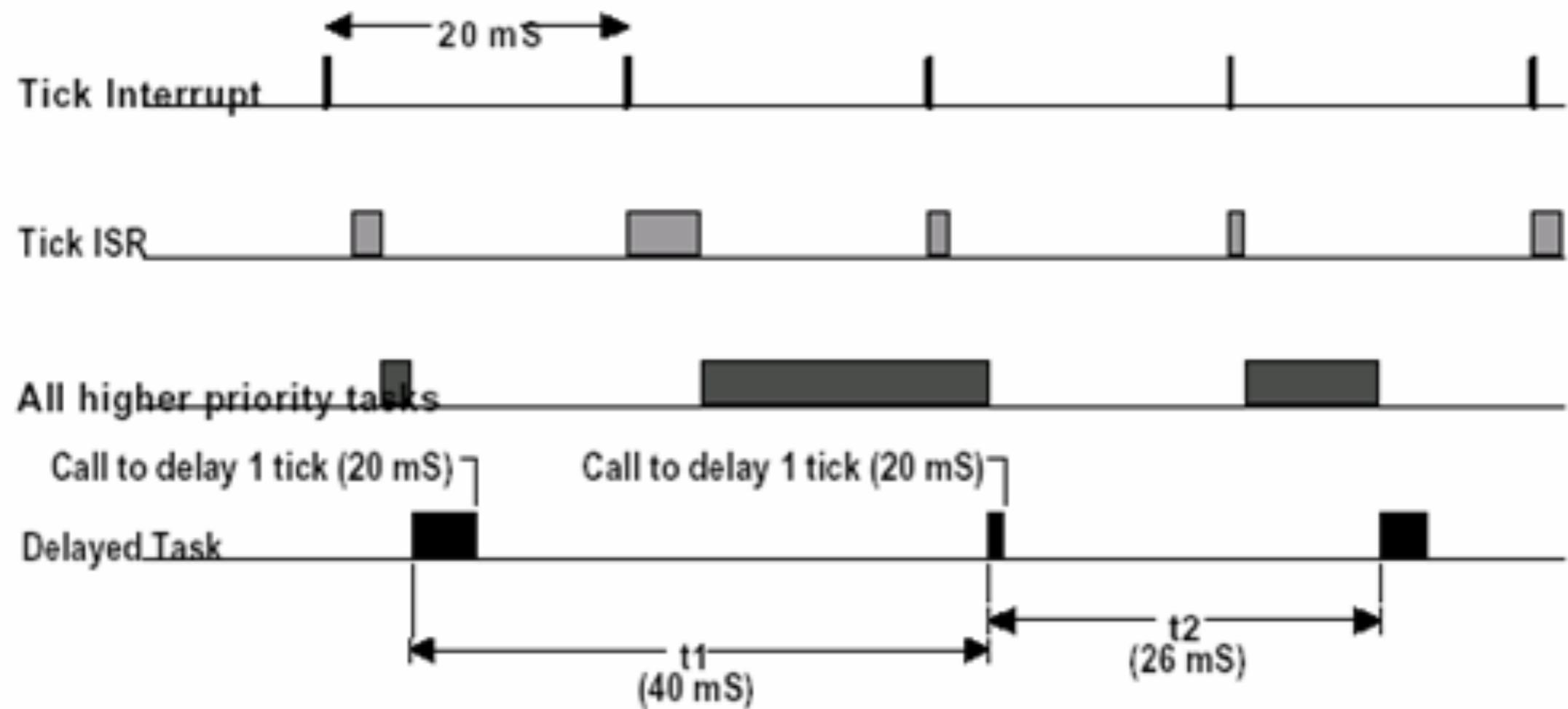


OSTimeDLY的问题(I)



将任务延迟一个时钟节拍(第一种情况)

OSTimeDLY的问题(2)



将任务延迟一个时钟节拍(第二种情况)

解决方案

- 增加微处理器的时钟频率
- 增加时钟节拍的频率
- 重新安排任务的优先级
- 避免使用浮点运算(如果非使用不可,尽量用单精度数)
- 使用能较好地优化程序代码的编译器
- 时间要求苛刻的代码用汇编语言写
- 如果可能,用同一家族的更快的微处理器做系统升级。
- 不管怎么样, 抖动总是存在的

OSTimeDlyHMSM()

- OSTimeDlyHMSM(): OSTimeDly()的另一个版本，即按时分秒延时函数；
- 使用方法

```
INT8U OSTimeDlyHMSM(  
    INT8U hours, // 小时  
    INT8U minutes, // 分钟  
    INT8U seconds, // 秒  
    INT16U milli // 毫秒  
);
```

OSTimeDlyResume()

- OSTimeDlyResume(): 让处在延时期的任务提前结束延时，进入就绪状态；
- 使用方法
 - INT8U OSTimeDlyResume (INT8U prio);
 - prio表示需要提前结束延时的任务的优先级/任务ID。

系统时间

- 每隔一个时钟节拍，发生一个时钟中断，将一个32位的计数器OSTime加1；
- 该计数器在用户调用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。若时钟节拍的频率等于100Hz，该计数器每隔497天就重新开始计数；
- OSTimeGet(): 获得该计数器的当前值；
 - INT32U OSTimeGet (void);
- OSTimeSet(): 设置该计数器的值。
 - void OSTimeSet (INT32U ticks);

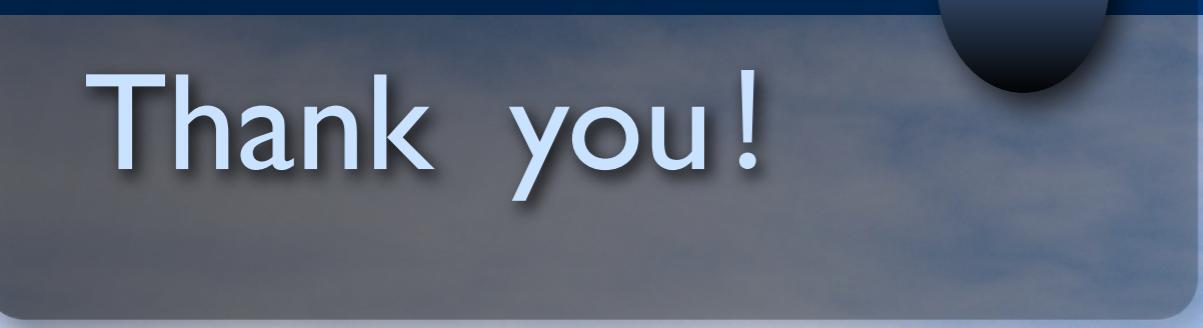
何时启动系统定时器

- 如果在OSStart之前启动定时器，则系统可能无法正确执行完OSStartHighRdy
- OSStart函数直接调用OSStartHighRdy去执行最高优先级的任务，OSStart不返回
- 系统定时器应该在系统的最高优先级任务中启动
- 使用OSRunning变量来控制操作系统的运行

时钟节拍的启动

- 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
- 在调用OSStart()之后做的第一件事是初始化定时器中断

```
void main(void)
{
    ...
    OSInit(); /* 初始化uC/OS-II*/
    /* 应用程序初始化代码... */
    /* 调用OSTaskCreate()创建至少一个任务*/
    允许时钟节拍中断; /* 错误！可能crash!*/
    OSStart(); /* 开始多任务调度 */
}
```



Thank you!

