(163条消息) ThreadX内核源码分析 - 定时器及线程时间片调度(arm)_arm7star的博客-CSDN博客_threadx源码

blog.csdn.net/arm7star/article/details/122952009

1、线程时间片介绍(tx_thread_time_slice)

ThreadX内核同优先级线程之间是按时间片调度的，tx_thread_new_time_slice记录线程的时间片(一次调度的总的时间片)，tx_thread_time_slice记录线程的剩余时间片(ThreadX内核每次调度线程时，并不是从tx_thread_new_time_slice，而是从上次换出cpu时的剩余时间片继续计时，只有当时间片用尽时，tx_thread_time_slice才会从tx_thread_new_time_slice开始)；

ThreadX内核正在执行的线程在优先级链表表头，线程被切换出cpu时，线程并不会移动到链表末尾，如果内核线程执行还没多久就被高优先级线程抢占，把线程移动到链表末尾的话，该线程就要等很久才会得到调度，所以线程被抢占的话，线程仍在优先级链表的表头，下次该优先级成为最高优先级时，取表头线程也就是上次被切换出去的线程继续执行；另外，正是由于被换出去的线程仍在表头，如果线程以新的时间片执行的话，如果每次线程都没用尽时间片就被高优先级抢占，那么同优先级链表后面的线程就得不到调度，所以每次线程被调度都是接着上次没有用完的时间片tx_thread_time_slice继续计时，直到时间片用完才分配新的时间片tx_thread_new_time_slice(如果同优先级有其他线程就绪就将线程移动到末尾，调度下一个同优先级线程，否则接着执行当前线程)。

2、定时器中断

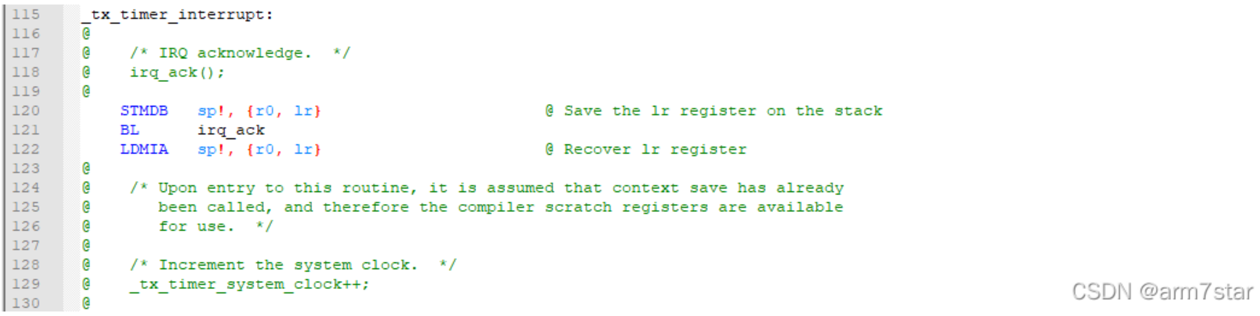
2.1、中断介绍

前一篇文章已经介绍过中断上下文保存、中断处理、中断上下文恢复相关内容，汇编代码具体实现参考：
ThreadX内核源码分析 - ports线程上下文相关代码分析(arm)_arm7star的博客-CSDN博客
1、ports源码介绍内核与cpu相关的关键代码基本都是用汇编语言实现的，c语言可能实现不了或者不好编写。ThreadX官网针对ARM9 gcc的移植代码在threadx-6.1.2_rel\ports\arm9\gnu\src目录下，ThreadX文件命名规则基本是以该文件包含的函数名命名的(函数名多了一个"_"前缀，文件名里面没有"_"前缀)，每个源文件通常只实现一个函数；ports代码目录如下：tx_thread_context_restore.S是_tx_thread_context
https://blog.csdn.net/arm7star/article/details/122930850?spm=1001.2014.3001.5502

IRQ中断处理顶层代码逻辑如下，__tx_irq_handler为IRQ中断处理函数入口，__tx_irq_handler主要就是保存必要的中断上下文，调用中断处理函数，恢复中断上下文(如果有高优先级唤醒，那么要将IRQ栈里面保存的寄存器以及其他没有修改的寄存器保存到被中断线程的栈里面，_tx_thread_context_save只保存了会影响到的必要的寄存器(保存在IRQ栈里面)，因为中断返回时并不一定会切换线程，保存过多的寄存器反而影响性能；如果线程没有被抢占或者切换出去，那么恢复保存在IRQ栈里面的寄存器即可)：

本文只用到了定时器中断，所以中断服务程序就直接是调用_tx_timer_interrupt；另外ThreadX官网的tx_timer_interrupt是针对一类cpu的，而定时器是处理器相关的，所以官网的tx_timer_interrupt仅是内核相关的，需要自己在tx_timer_interrupt合适的地方加上硬件中断清理工作，否则定时器中断将不断触发；

针对s3c2440的定时器中断清理代码如下，保存r0, lr寄存器(BL指令会修改lr寄存器；c函数r0-r3之外的寄存器如果有用到，编译器会保护及恢复的，所以调用中断清除C函数，通用寄存器只需要考虑r0-r3，_tx_timer_interrupt调用irq_ack之后的代码很显然没有用到r0-r3的旧的值，所以r0-r3是不需要保护的，但是仿照ThreadX的其他代码，为了让栈保持8 byte对齐，还是把r0保存到栈里面了)，irq_ack就是c代码清除定时器中断：



```
_tx_timer_interrupt:
@
@    /* IRQ acknowledge.  */
@    irq_ack();
@
    STMDB   sp!, {r0, lr}                       @ Save the lr register on the stack
    BL      irq_ack
    LDMIA   sp!, {r0, lr}                       @ Recover lr register
@
@    /* Upon entry to this routine, it is assumed that context save has already
@       been called, and therefore the compiler scratch registers are available
@       for use.  */
@
@    /* Increment the system clock.  */
@    _tx_timer_system_clock++;
@
```

2.2、定时器中断服务程序(_tx_timer_interrupt)

_tx_timer_interrupt定时器中断服务程序主要是对正在执行的线程的时间片减1，检查当前线程时间片是否用尽，检查当前是否有timer定时器超时(应用程序定时器，非硬件定时器，线程的sleep、阻塞超时等的唤醒都是通过定时器唤醒的，例如等待某个互斥锁，如果超时了就不继续等待，那么内核就启动了一个定时器，在超时时间内等到了互斥锁，那么就取消定时器，否则定时器超时就唤醒阻塞的线程，返回获取互斥锁失败)；如果有定时器超时，调用_tx_timer_expiration_process处理超时定时器，如果线程时间片用尽，调用_tx_thread_time_slice处理时间片(有同优先级的其他线程的话需要调度下一个线程，否则不需要调度，重新设置当前线程的时间片即可，新一轮开始计时)，_tx_timer_interrupt代码如下：

```
115 _tx_timer_interrupt:
116 @
117 @    /* IRQ acknowledge.  */
118 @    irq_ack();
119 @
120     STMDB   sp!, {r0, lr}                      @ Save the lr register on the stack
121     BL      irq_ack // 清除定时器中断
122     LDMIA   sp!, {r0, lr}                      @ Recover lr register
123 @
124 @    /* Upon entry to this routine, it is assumed that context save has already
125 @       been called, and therefore the compiler scratch registers are available
126 @       for use.  */
127 @
128 @    /* Increment the system clock.  */
129 @    _tx_timer_system_clock++;
130 @
131     LDR     r1, =_tx_timer_system_clock        @ Pickup address of system clock
132     LDR     r0, [r1]                           @ Pickup system clock
133     ADD     r0, r0, #1                         @ Increment system clock // _tx_timer_system_clock++，每次定时器中断，系统的时钟加1
134     STR     r0, [r1]                           @ Store new system clock
135 @
136 @    /* Test for time-slice expiration.  */
137 @    if (_tx_timer_time_slice)
138 @    {
139 @
140     LDR     r3, =_tx_timer_time_slice          @ Pickup address of time-slice
141     LDR     r2, [r3]                           @ Pickup time-slice
142     CMP     r2, #0                             @ Is it non-active?
143     BEQ     __tx_timer_no_time_slice            @ Yes, skip time-slice processing // 检查_tx_timer_time_slice是否激活，_tx_timer_time_slice不为0，表示线程使用了时间片，每次定时器中断要对线程时间片计时，_tx_timer_time_slice为0，表示线程没有使用时间片，不对线程运行时间计时，只有线程被抢占或者线程自己让出cpu，否则同优先级的其他就绪线程可能就得不到调度，一般不启用时间片的线程都会有某些阻塞调用
144 @
145 @       /* Decrement the time_slice.  */
146 @       _tx_timer_time_slice--;
147 @
148     SUB     r2, r2, #1                          @ Decrement the time-slice // 线程启用了时间片，对线程时间片减1(线程调度时，线程剩余时间片保存在_tx_timer_time_slice里面，线程被换出时，_tx_timer_time_slice保存到线程的剩余时间片tx_thread_time_slice里面)
149     STR     r2, [r3]                            @ Store new time-slice value
150 @
151 @       /* Check for expiration.  */
152 @       if (__tx_timer_time_slice == 0)
153 @
154     CMP     r2, #0                              @ Has it expired? // 检查是否超时(线程时间片用尽了)
155     BNE     __tx_timer_no_time_slice            @ No, skip expiration processing // _tx_timer_time_slice不为0表示还有时间片，跳转到__tx_timer_no_time_slice
156 @
157 @       /* Set the time-slice expired flag.  */
158 @       _tx_timer_expired_time_slice =  TX_TRUE;
159 @
160     LDR     r3, =_tx_timer_expired_time_slice   @ Pickup address of expired flag // _tx_timer_time_slice为0，线程时间片用尽了，设置_tx_timer_expired_time_slice线程时间片用尽标志
161     MOV     r0, #1                              @ Build expired value
162     STR     r0, [r3]                            @ Set time-slice expiration flag
163 @
164 @    }
165 @
166 __tx_timer_no_time_slice: // 检查完了线程时间片
167 @
168 @    /* Test for timer expiration.  */
```

```
169 @    if (*_tx_timer_current_ptr)

170 @    {

171 @

172     LDR     r1, =_tx_timer_current_ptr          @ Pickup current timer pointer address

173     LDR     r0, [r1]                            @ Pickup current timer

174     LDR     r2, [r0]                            @ Pickup timer list entry // *_tx_timer_current_ptr

175     CMP     r2, #0                              @ Is there anything in the list? // 检查*_tx_timer_current_ptr是否为空，是否有定时器，_tx_timer_current_ptr类似墙上的一个挂钟，每次定时器中断移动一格，每个格子下面挂的是超时的定时器，如果没有定时器的话，就是空的，否则有定时器超时

176     BEQ     __tx_timer_no_timer                 @ No, just increment the timer // 没有定时器的话，跳转到__tx_timer_no_timer

177 @

178 @        /* Set expiration flag.  */

179 @        _tx_timer_expired =  TX_TRUE;

180 @

181     LDR     r3, =_tx_timer_expired              @ Pickup expiration flag address // 有定时器超时，设置_tx_timer_expired标志

182     MOV     r2, #1                              @ Build expired value

183     STR     r2, [r3]                            @ Set expired flag

184     B       __tx_timer_done                     @ Finished timer processing

185 @

186 @    }

187 @    else

188 @    {

189 __tx_timer_no_timer: // 没有定时器超时

190 @

191 @        /* No timer expired, increment the timer pointer.  */

192 @        _tx_timer_current_ptr++;

193 @

194     ADD     r0, r0, #4                          @ Move to next timer // _tx_timer_current_ptr移动到下一个元素(_tx_timer_current_ptr指向_tx_timer_list数组的一个节点，_tx_timer_list的每个节点下是一个超时定时器链表，该链表里面的定时器全是在同一个时间点超时，_tx_timer_list每个节点间超时时间相差一个定时器中断)，类似秒针走一格

195 @
```

```
196 @        /* Check for wraparound.  */
197 @        if (_tx_timer_current_ptr == _tx_timer_list_end)
198 @
199     LDR     r3, =_tx_timer_list_end             @ Pickup address of timer list end // 因为_tx_timer_list使用数组实现链表的，_tx_timer_current_ptr指向_tx_timer_list的元素，_tx_timer_current_ptr超过_tx_timer_list的最后一个元素时，需要重新指向_tx_timer_list的第1个元素，即wrap回环，跟秒针一样，秒针走到59之后，下一次就回到0
200     LDR     r2, [r3]                            @ Pickup list end
201     CMP     r0, r2                              @ Are we at list end?
202     BNE     __tx_timer_skip_wrap                @ No, skip wraparound logic
203 @
204 @            /* Wrap to beginning of list.  */
205 @            _tx_timer_current_ptr =  _tx_timer_list_start;
206 @
207     LDR     r3, =_tx_timer_list_start           @ Pickup address of timer list start
208     LDR     r0, [r3]                            @ Set current pointer to list start
209 @
210 __tx_timer_skip_wrap:
211 @
212     STR     r0, [r1]                            @ Store new current timer pointer
213 @    }
214 @
215 __tx_timer_done: // 检查完了定时器
216 @
217 @
218 @    /* See if anything has expired.  */
219 @    if ((_tx_timer_expired_time_slice) || (_tx_timer_expired))
220 @    {
221 @
222     LDR     r3, =_tx_timer_expired_time_slice   @ Pickup address of expired flag
223     LDR     r2, [r3]                            @ Pickup time-slice expired flag
```

```
    CMP     r2, #0                                   @ Did a time-slice expire? // 检查线程时间片超时标志_tx_timer_expired_time_slice(如果线程时间片用尽，前面会设置_tx_timer_expired_time_slice)
    BNE     __tx_something_expired                   @ If non-zero, time-slice expired // 如果__tx_something_expired不为0，线程时间片用尽了，跳转到_tx_timer_expired，需要检查是否换出当前执行的线程
    LDR     r1, =_tx_timer_expired                   @ Pickup address of other expired flag // 检查是否有设置定时器超时标志
    LDR     r0, [r1]                                 @ Pickup timer expired flag
    CMP     r0, #0                                   @ Did a timer expire?
    BEQ     __tx_timer_nothing_expired               @ No, nothing expired // 定时器没有超时，跳转到__tx_timer_nothing_expired(需要注意，线程时间片没有用尽的情况才会走到这里的指令，也就是__tx_timer_nothing_expired是线程时间片没有用尽而且没有定时器超时)
@
__tx_something_expired: // 走到这里是至少有一个超时(线程时间片用尽了、有定时器超时)
@
@
    STMDB   sp!, {r0, lr}                            @ Save the lr register on the stack // lr入栈(BL指令会修改lr，r0入栈只是为了让栈保持8 byte对齐)
                                                     @   and save r0 just to keep 8-byte alignment
@
@    /* Did a timer expire?  */
@    if (_tx_timer_expired)
@    {
@
    LDR     r1, =_tx_timer_expired                   @ Pickup address of expired flag
    LDR     r0, [r1]                                 @ Pickup timer expired flag
    CMP     r0, #0                                   @ Check for timer expiration
    BEQ     __tx_timer_dont_activate                 @ If not set, skip timer activation // 再次检查一下_tx_timer_expired是否超时，没有超时的话，跳转到_tx_timer_expired
@
@        /* Process timer expiration.  */
@        _tx_timer_expiration_process();
@
```

```
    BL      _tx_timer_expiration_process        @ Call the timer expiration handling routine // 调用定时器超时处理函数_tx_timer_expiration_process，处理_tx_timer_current_ptr下面挂载的超时定时器
@
@    }
__tx_timer_dont_activate:
@
@    /* Did time slice expire?  */
@    if (_tx_timer_expired_time_slice)
@    {
@
    LDR     r3, =_tx_timer_expired_time_slice   @ Pickup address of time-slice expired
    LDR     r2, [r3]                            @ Pickup the actual flag
    CMP     r2, #0                              @ See if the flag is set // 检查线程时间片是否用尽
    BEQ     __tx_timer_not_ts_expiration        @ No, skip time-slice processing // 线程时间片没有用尽，跳转到__tx_timer_not_ts_expiration
@
@        /* Time slice interrupted thread.  */
@        _tx_thread_time_slice();
@
    BL      _tx_thread_time_slice               @ Call time-slice processing // 调用线程时间片处理函数_tx_thread_time_slice
@
@    }
@
__tx_timer_not_ts_expiration:
@
    LDMIA   sp!, {r0, lr}                       @ Recover lr register (r0 is just there for
                                                @   the 8-byte stack alignment
@
@    }
@
```

```
277 __tx_timer_nothing_expired:
278 @
279 #ifdef __THUMB_INTERWORK
280     BX      lr                                 @ Return to caller
281 #else
282     MOV     pc, lr                             @ Return to caller // 中断处理函数返回(中断服务程序不会切换线程，线程切换在上下文恢复的时候判断)
283 #endif
284 @
```

3、线程时间片用尽(_tx_thread_time_slice)

当前线程没有启用抢占情况下，调度下一个就绪线程，当前线程开启抢占的话，可以抢占自己优先级的线程，那么就算有其他同优先级就绪线程，也不调度下一个就绪线程，当前线程抢占同优先级的其他就绪线程；调度下一个就绪线程主要是更新_tx_thread_execute_ptr，中断返回时用到_tx_thread_execute_ptr；_tx_thread_time_slice代码如下：

```
VOID  _tx_thread_time_slice(VOID)
{

TX_INTERRUPT_SAVE_AREA

TX_THREAD       *thread_ptr;
#ifdef TX_ENABLE_STACK_CHECKING
TX_THREAD       *next_thread_ptr;
#endif
#ifdef TX_ENABLE_EVENT_TRACE
ULONG           system_state;
UINT            preempt_disable;
#endif

    /* Pickup thread pointer.  */
    TX_THREAD_GET_CURRENT(thread_ptr) // 获取当前正在执行的线程 _tx_thread_current_ptr

#ifdef TX_ENABLE_STACK_CHECKING

    /* Check this thread's stack.  */
    TX_THREAD_STACK_CHECK(thread_ptr)

    /* Set the next thread pointer to NULL.  */
    next_thread_ptr =  TX_NULL;
#endif

    /* Lockout interrupts while the time-slice is evaluated.  */
    TX_DISABLE // 关闭中断

    /* Clear the expired time-slice flag.  */
    _tx_timer_expired_time_slice =  TX_FALSE; // 清除线程时间片用尽标志

    /* Make sure the thread pointer is valid.  */
    if (thread_ptr != TX_NULL) // 再次检查_tx_thread_current_ptr是否为空(应该是关中断前_tx_thread_current_ptr可能被设置为TX_NULL，定时器超时或者嵌套中断服务程序可能把正在执行的线程删掉也是可能的)
    {

        /* Make sure the thread is still active, i.e. not suspended.  */
        if (thread_ptr -> tx_thread_state == TX_READY) // 再次检查线程状态(如果线程非就绪状态，是不用管时间片的，唤醒线程时应该会重新设置时间片，毕竟线程睡眠这么久了，给刚唤醒的线程多一点点时间片也是合理的)
        {

            /* Setup a fresh time-slice for the thread.  */
            thread_ptr -> tx_thread_time_slice =  thread_ptr -> tx_thread_new_time_slice; // 重新设置线程时间片

            /* Reset the actual time-slice variable.  */
            _tx_timer_time_slice =  thread_ptr -> tx_thread_time_slice; // 重新设置_tx_timer_time_slice，如果当前线程不切换出去的话，设置_tx_timer_time_slice是有意义的，中断返回不会重新设置_tx_timer_time_slice，如果切换到别的线程，_tx_timer_time_slice会被再次设置，这个也不影响

            /* Determine if there is another thread at the same priority and preemption-threshold
               is not set.  Preemption-threshold overrides time-slicing.  */
            if (thread_ptr -> tx_thread_ready_next != thread_ptr) // 检查同一优先级线程就绪链表是否有下一个就绪线程；之前文章有介绍，正在执行的线程在就绪链表的表头，thread_ptr -> tx_thread_ready_next就是下一个就绪的线程
            {

                /* Check to see if preemption-threshold is not being used.  */
                if (thread_ptr -> tx_thread_priority == thread_ptr -> tx_thread_preempt_threshold) // 当前正在执行的线程的抢占阈值等于线程优先级(没有启用抢占)，就调度下一个就绪线程，设置_tx_thread_execute_ptr；有抢占的话就不管后续就绪线程；一般情况下，这两个值是一样的，如果不相等，抢占阈值优先级应该高于线程优先级，那么只看下一个就绪线程优先级的话，当前线程明显可以抢占下一个线程，这个判断一定意义上等价于比较当前线程的抢占阈值与下一个就绪线程的优先级
                {

                    /* Preemption-threshold is not being used by this thread. */

                    /* There is another thread at this priority, make it the highest at
                       this priority level.  */
                    _tx_thread_priority_list[thread_ptr -> tx_thread_priority] = thread_ptr -> tx_thread_ready_next; // 就绪链表表头指针指向下一个就绪线程(下一个就绪线程移到了表头，当前线程移到了表尾)

                    /* Designate the highest priority thread as the one to execute.  Don't use this
                       thread's priority as an index just in case a higher priority thread is now
                       ready!  */
                    _tx_thread_execute_ptr = _tx_thread_priority_list[_tx_thread_highest_priority]; // _tx_thread_execute_ptr指向下一个就绪线程，下次调度将要执行的线程

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

                    /* Increment the thread's time-slice counter.  */
                    thread_ptr -> tx_thread_performance_time_slice_count++;

                    /* Increment the total number of thread time-slice operations.  */
                    _tx_thread_performance_time_slice_count++;
#endif


#ifdef TX_ENABLE_STACK_CHECKING

                    /* Pickup the next execute pointer.  */
                    next_thread_ptr =  _tx_thread_execute_ptr;
#endif
```

```
                }
            }
        }
    }

#ifdef TX_ENABLE_EVENT_TRACE

    /* Pickup the volatile information.  */
    system_state =  TX_THREAD_GET_SYSTEM_STATE();
    preempt_disable =  _tx_thread_preempt_disable;

    /* Insert this event into the trace buffer.  */
    TX_TRACE_IN_LINE_INSERT(TX_TRACE_TIME_SLICE, _tx_thread_execute_ptr, system_state, preempt_disable, TX_POINTER_TO_ULONG_CONVERT(&thread_ptr), TX_TRACE_INTERNAL_EVENTS)
#endif

    /* Restore previous interrupt posture.  */
    TX_RESTORE // 开启中断

#ifdef TX_ENABLE_STACK_CHECKING

    /* Determine if there is a next thread pointer to perform stack checking on.  */
    if (next_thread_ptr != TX_NULL)
    {

        /* Yes, check this thread's stack.  */
        TX_THREAD_STACK_CHECK(next_thread_ptr)
    }
#endif
}
```

4、软件定时器timer

4.1、定时器结构

_tx_timer_list是一个超时定时器链表数组，首先_tx_timer_list是个数组，数组的每个元素是个超时定时器链表，其次_tx_timer_list是一个用数组实现的单向循环链表，逻辑上，数组前一个元素指向后一个元素，最末尾的元素指向第一个元素；

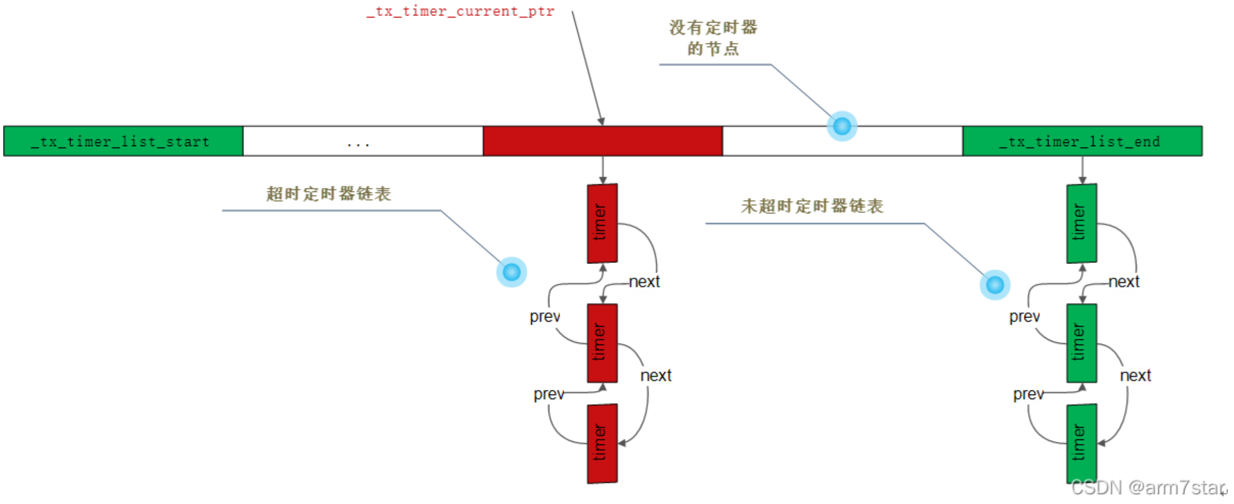
_tx_timer_current_ptr指向超时的定时器链表(_tx_timer_list对应的元素)。

假设硬件定时器中断时间周期为1秒，_tx_timer_current_ptr当前指向_tx_timer_list[0]，此时要加入一个n秒的定时器，那么内核就会将定时器挂载到_tx_timer_current_ptr + n的超时定时器链表(_tx_timer_current_ptr + n没有超过_tx_timer_list最后一个元素，超过后要从第一个元素开始计算)；

定时器中断一次_tx_timer_current_ptr加1，中断n次_tx_timer_current_ptr加n，中断n次之后，此时正好已经过了n秒了，内核检测到_tx_timer_current_ptr + n链表不为空，就会去处理_tx_timer_current_ptr + n指向的超时定时器链表。

定时器超时时间超过_tx_timer_list能表示的范围的情况，假如_tx_timer_list只有4个元素，_tx_timer_current_ptr指向_tx_timer_list[0]，下一次定时器中断就认为_tx_timer_list[0]超时了，如果定时器时间为4秒，那么计算之后定时器似乎挂载到_tx_timer_list[0]，很明显这个超时时间不对，最多只能挂载到_tx_timer_list[3]，也就是最多只有3秒，内核做了个简单处理，就是先延迟3秒，先挂个超时定时器到_tx_timer_list[3]，还剩下1秒先记录下来，_tx_timer_list[3]超时的时候，定时器处理函数检查到还有1秒时间，那么再挂一个超时定时器到_tx_timer_list[0]即可。

ThreadX超时定时器数据结构大致如下图所示：



4.2、定时器超时处理函数(_tx_timer_expiration_process)

前面代码注释里面已经讲过了_tx_timer_current_ptr，在此略过。前面可以看到_tx_timer_expiration_process是在中断上下文里面调用的，定时器处理需要较长时间，因此_tx_timer_expiration_process唤醒一个专门处理超时定时器的线程_tx_timer_thread，线程入口为_tx_timer_thread_entry；_tx_timer_thread是个无限循环线程，没有超时定时器时就睡眠。

前面大致介绍过了定时器原理，_tx_timer_thread_entry主要就是判断定时器是否真正超时，是否需要重新激活，然后调用定时器超时回调函数；

因为_tx_timer_thread_entry与中断服务程序使用共同的变量，所以开关中断比较频繁，既要互斥也不能关中断太久，_tx_timer_thread_entry代码实现如下：

```
VOID  _tx_timer_thread_entry(ULONG timer_thread_input)
{

TX_INTERRUPT_SAVE_AREA

TX_TIMER_INTERNAL           *expired_timers;
TX_TIMER_INTERNAL           *reactivate_timer;
TX_TIMER_INTERNAL           *next_timer;
TX_TIMER_INTERNAL           *previous_timer;
TX_TIMER_INTERNAL           *current_timer;
VOID                        (*timeout_function)(ULONG id);
ULONG                       timeout_param =  ((ULONG) 0);
TX_THREAD                   *thread_ptr;
#ifdef TX_REACTIVATE_INLINE
TX_TIMER_INTERNAL           **timer_list;               /* Timer list pointer */
UINT                        expiration_time;            /* Value used for pointer offset*/
ULONG                       delta;
#endif
#ifdef TX_TIMER_ENABLE_PERFORMANCE_INFO
TX_TIMER                    *timer_ptr;
#endif


    /* Make sure the timer input is correct.  This also gets rid of the
       silly compiler warnings.  */
    if (timer_thread_input == TX_TIMER_ID)
    {

        /* Yes, valid thread entry, proceed...  */

    /* Now go into an infinite loop to process timer expirations.  */
    while (TX_LOOP_FOREVER)
    {

        /* First, move the current list pointer and clear the timer
           expired value.  This allows the interrupt handling portion
           to continue looking for timer expirations.  */
        TX_DISABLE // 关中断，中断服务程序会访问修改_tx_timer_current_ptr，有定时器超时时，中断服务程序不会移动_tx_timer_current_ptr，_tx_timer_current_ptr没有被取走的情况下，发生再多中断，_tx_timer_current_ptr也不会变，所以_tx_timer_current_ptr要被尽快取走，否则定时器就不准了

        /* Save the current timer expiration list pointer.  */
        expired_timers =  *_tx_timer_current_ptr; // 取出超时定时器链表

        /* Modify the head pointer in the first timer in the list, if there
           is one!  */
        if (expired_timers != TX_NULL)
        {

            expired_timers -> tx_timer_internal_list_head = &expired_timers;
        }

        /* Set the current list pointer to NULL.  */
        *_tx_timer_current_ptr =  TX_NULL; // _tx_timer_current_ptr设置为空，中断服务程序才会对_tx_timer_current_ptr进行移动，软件定时器才会计时

        /* Move the current pointer up one timer entry wrap if we get to
           the end of the list.  */
        _tx_timer_current_ptr =  TX_TIMER_POINTER_ADD(_tx_timer_current_ptr, 1); // 下一次中断时的超时定时器链表
        if (_tx_timer_current_ptr == _tx_timer_list_end) // 链表回环
        {

            _tx_timer_current_ptr =  _tx_timer_list_start;
        }

        /* Clear the expired flag.  */
        _tx_timer_expired =  TX_FALSE; // 清除定时器超时_tx_timer_expired

        /* Restore interrupts temporarily.  */
        TX_RESTORE // 允许中断，可以对软件定时器计时了

        /* Disable interrupts again.  */
        TX_DISABLE // 紧接着又立即禁止了中断(主要是前面关闭时间可能有点长，如果有中断等待处理，开启中断就会使cpu立即处理中断，避免中断等待太久；另外接下来的关中断也可能很久，有中断的话先处理中断)

        /* Next, process the expiration of the associated timers at this
           time slot.  */
        while (expired_timers != TX_NULL) // 超时定时器不为空，循环处理当前链表里面的所有超时定时器
        {

            /* Something is on the list.  Remove it and process the expiration.  */
            current_timer =  expired_timers;

            /* Pickup the next timer.  */
            next_timer =  expired_timers -> tx_timer_internal_active_next;

            /* Set the reactivate_timer to NULL.  */
            reactivate_timer =  TX_NULL;

            /* Determine if this is the only timer.  */
            if (current_timer == next_timer) // 超时定时器下一个定时器指向自己(只有一个超时定时器)
            {
```

```

            /* Yes, this is the only timer in the list.  */

            /* Set the head pointer to NULL.  */
            expired_timers =  TX_NULL; // 清空超时定时器链表即可
        }
        else // 将当前处理的超时定时器从链表中删除
        {

            /* No, not the only expired timer.  */

            /* Remove this timer from the expired list.  */
            previous_timer = current_timer -> tx_timer_internal_active_previous;
            next_timer -> tx_timer_internal_active_previous = previous_timer;
            previous_timer -> tx_timer_internal_active_next = next_timer;

            /* Modify the next timer's list head to point at the current list head.  */
            next_timer -> tx_timer_internal_list_head = &expired_timers;

            /* Set the list head pointer.  */
            expired_timers =  next_timer;
        }

        /* In any case, the timer is now off of the expired list.  */

        /* Determine if the timer has expired or if it is just a really
           big timer that needs to be placed in the list again.  */
        if (current_timer -> tx_timer_internal_remaining_ticks > TX_TIMER_ENTRIES) // 定时器剩余超时时间大于_tx_timer_list最大超时时间
        {
```

```

                    /* Timer is bigger than the timer entries and must be
                       rescheduled.  */

#ifdef TX_TIMER_ENABLE_PERFORMANCE_INFO

                    /* Increment the total expiration adjustments counter.  */
                    _tx_timer_performance__expiration_adjust_count++;

                    /* Determine if this is an application timer.  */
                    if (current_timer -> tx_timer_internal_timeout_function != &_tx_thread_timeout)
                    {

                        /* Derive the application timer pointer.  */

                        /* Pickup the application timer pointer.  */
                        TX_USER_TIMER_POINTER_GET(current_timer, timer_ptr)

                        /* Increment the number of expiration adjustments on this timer.  */
                        if (timer_ptr -> tx_timer_id == TX_TIMER_ID)
                        {

                            timer_ptr -> tx_timer_performance__expiration_adjust_count++;
                        }
                    }
#endif

                    /* Decrement the remaining ticks of the timer.  */
                    current_timer -> tx_timer_internal_remaining_ticks =
                            current_timer -> tx_timer_internal_remaining_ticks - TX_TIMER_ENTRIES; // 需要先超时TX_TIMER_ENTRIES，剩余超时时间保留下来

                    /* Set the timeout function to NULL in order to bypass the
                       expiration.  */
                    timeout_function =  TX_NULL; // 定时器没有真正超时，还不需要调用定时器超时函数，timeout_function设置为空即可

                    /* Make the timer appear that it is still active while interrupts
                       are enabled.  This will permit proper processing of a timer
                       deactivate from an ISR.  */
                    current_timer -> tx_timer_internal_list_head =  &reactivate_timer;
                    current_timer -> tx_timer_internal_active_next =  current_timer;

                    /* Setup the temporary timer list head pointer.  */
                    reactivate_timer =  current_timer; // 需要重新激活的定时器
                }
                else // 剩余时间少于TX_TIMER_ENTRIES是不会再起定时器了，毕竟代码调用要花费时间，这些时间加上关中断的延迟，甚至就可以弥补这些偏差了，没办法计算到很精确
                {

                    /* Timer did expire.  */

#ifdef TX_TIMER_ENABLE_PERFORMANCE_INFO

                    /* Increment the total expirations counter.  */
                    _tx_timer_performance_expiration_count++;

                    /* Determine if this is an application timer.  */
                    if (current_timer -> tx_timer_internal_timeout_function != &_tx_thread_timeout)
                    {

                        /* Derive the application timer pointer.  */

                        /* Pickup the application timer pointer.  */
                        TX_USER_TIMER_POINTER_GET(current_timer, timer_ptr)

                        /* Increment the number of expirations on this timer. */
                        if (timer_ptr -> tx_timer_id == TX_TIMER_ID)
                        {

                            timer_ptr -> tx_timer_performance_expiration_count++;
                        }
                    }
#endif

                    /* Copy the calling function and ID into local variables before interrupts
                       are re-enabled.  */
                    timeout_function =  current_timer -> tx_timer_internal_timeout_function; // 定时器超时回调函数
                    timeout_param =     current_timer -> tx_timer_internal_timeout_param; // 定时器超时回调函数指针

                    /* Copy the reinitialize ticks into the remaining ticks.  */
                    current_timer -> tx_timer_internal_remaining_ticks = current_timer -> tx_timer_internal_re_initialize_ticks; // tx_timer_internal_re_initialize_ticks不为0的话就是个循环定时器

                    /* Determine if the timer should be reactivated.  */
                    if (current_timer -> tx_timer_internal_remaining_ticks != ((ULONG) 0))
                    {

                        /* Make the timer appear that it is still active while processing
```

```
                           the expiration routine and with interrupts enabled.  This will
                           permit proper processing of a timer deactivate from both the
                           expiration routine and an ISR.  */
                        current_timer -> tx_timer_internal_list_head =  &reactivate_timer;
                        current_timer -> tx_timer_internal_active_next =  current_timer;

                        /* Setup the temporary timer list head pointer.  */
                        reactivate_timer =  current_timer; // 需要重新激活的定时器
                    }
                    else
                    {

                        /* Set the list pointer of this timer to NULL.  This is used to indicate
                           the timer is no longer active.  */
                        current_timer -> tx_timer_internal_list_head =  TX_NULL;
                    }
                }

                /* Set pointer to indicate the expired timer that is currently being processed.  */
                _tx_timer_expired_timer_ptr =  current_timer; // 正在处理的定时器，避免其他线程操作该定时器

                /* Restore interrupts for timer expiration call.  */
                TX_RESTORE

                /* Call the timer-expiration function, if non-NULL.  */
                if (timeout_function != TX_NULL)
                {

                    (timeout_function) (timeout_param); // 定时器回调函数
                }

                /* Lockout interrupts again.  */
                TX_DISABLE

                /* Clear expired timer pointer.  */
                _tx_timer_expired_timer_ptr =  TX_NULL;

                /* Determine if the timer needs to be reactivated.  */
                if (reactivate_timer == current_timer) // reactivate_timer要么为空，要么为current_timer，实际就是判断是否需要重新激活current_timer
                {

                    /* Reactivate the timer.  */

#ifdef TX_TIMER_ENABLE_PERFORMANCE_INFO

                    /* Determine if this timer expired.  */
                    if (timeout_function != TX_NULL)
                    {

                        /* Increment the total reactivations counter.  */
                        _tx_timer_performance_reactivate_count++;

                        /* Determine if this is an application timer.  */
                        if (current_timer -> tx_timer_internal_timeout_function != &_tx_thread_timeout)
                        {

                            /* Derive the application timer pointer.  */

                            /* Pickup the application timer pointer.  */
                            TX_USER_TIMER_POINTER_GET(current_timer, timer_ptr)

                            /* Increment the number of expirations on this timer.  */
                            if (timer_ptr -> tx_timer_id == TX_TIMER_ID)
                            {

                                timer_ptr -> tx_timer_performance_reactivate_count++;
                            }
                        }
                    }
#endif

#ifdef TX_REACTIVATE_INLINE

                    /* Calculate the amount of time remaining for the timer.  */
                    if (current_timer -> tx_timer_internal_remaining_ticks > TX_TIMER_ENTRIES)
                    {

                        /* Set expiration time to the maximum number of entries.  */
                        expiration_time =  TX_TIMER_ENTRIES - ((UINT) 1);
                    }
                    else
                    {

                        /* Timer value fits in the timer entries.  */

                        /* Set the expiration time.  */
                        expiration_time =  ((UINT) current_timer -> tx_timer_internal_remaining_ticks) - ((UINT) 1);
                    }
```

```

                /* At this point, we are ready to put the timer back on one of
                   the timer lists.  */

                /* Calculate the proper place for the timer.  */
                timer_list =  TX_TIMER_POINTER_ADD(_tx_timer_current_ptr, expiration_time);
                if (TX_TIMER_INDIRECT_TO_VOID_POINTER_CONVERT(timer_list) >= TX_TIMER_INDIRECT_TO_VOID_POINTER_CONVERT(_tx_timer_list_end))
                {

                    /* Wrap from the beginning of the list.  */
                    delta =  TX_TIMER_POINTER_DIF(timer_list, _tx_timer_list_end);
                    timer_list =  TX_TIMER_POINTER_ADD(_tx_timer_list_start, delta);
                }

                /* Now put the timer on this list.  */
                if ((*timer_list) == TX_NULL)
                {

                    /* This list is NULL, just put the new timer on it.  */

                    /* Setup the links in this timer.  */
                    current_timer -> tx_timer_internal_active_next =  current_timer;
                    current_timer -> tx_timer_internal_active_previous =  current_timer;

                    /* Setup the list head pointer.  */
                    *timer_list =  current_timer;
                }
                else
```

```
            {

                /* This list is not NULL, add current timer to the end. */
                next_timer = *timer_list;
                previous_timer = next_timer -> tx_timer_internal_active_previous;
                previous_timer -> tx_timer_internal_active_next = current_timer;
                next_timer -> tx_timer_internal_active_previous = current_timer;
                current_timer -> tx_timer_internal_active_next = next_timer;
                current_timer -> tx_timer_internal_active_previous = previous_timer;
            }

            /* Setup list head pointer.  */
            current_timer -> tx_timer_internal_list_head =  timer_list;
#else

            /* Reactivate through the timer activate function.  */

            /* Clear the list head for the timer activate call.  */
            current_timer -> tx_timer_internal_list_head = TX_NULL;

            /* Activate the current timer.  */
            _tx_timer_system_activate(current_timer); // 激活当前定时器(挂载到超时定时器链表上面去)
#endif
        }

        /* Restore interrupts.  */
        TX_RESTORE

```

```
                /* Lockout interrupts again.  */
                TX_DISABLE
            }

            /* Finally, suspend this thread and wait for the next expiration.  */

            /* Determine if another expiration took place while we were in this
               thread.  If so, process another expiration.  */
            if (_tx_timer_expired == TX_FALSE) // 定时器中断没有激活新的超时定时器，那么需要挂起自己
            {

                /* Otherwise, no timer expiration, so suspend the thread.  */

                /* Build pointer to the timer thread.  */
                thread_ptr =  &_tx_timer_thread;

                /* Set the status to suspending, in order to indicate the
                   suspension is in progress.  */
                thread_ptr -> tx_thread_state =  TX_SUSPENDED;

#ifdef TX_NOT_INTERRUPTABLE

                /* Call actual non-interruptable thread suspension routine.  */
                _tx_thread_system_ni_suspend(thread_ptr, ((ULONG) 0));

                /* Restore interrupts.  */
                TX_RESTORE
#else

                /* Set the suspending flag. */
                thread_ptr -> tx_thread_suspending =  TX_TRUE;

                /* Increment the preempt disable count prior to suspending.  */
                _tx_thread_preempt_disable++; // 禁止抢占(挂起线程过程中，线程自己把自己切换出去，没必要被其他线程抢占，保存上下文，调度线程恢复上下文，再睡眠保存上下文，禁止抢占避免了一些不必要的操作)

                /* Restore interrupts.  */
                TX_RESTORE

                /* Call actual thread suspension routine.  */
                _tx_thread_system_suspend(thread_ptr); // 挂起自己(挂起线程需要一些时间，所以中断是打开的，但是又不想别其他线程抢占，所以前面的抢占是禁止的，_tx_thread_system_suspend会对_tx_thread_preempt_disable进行减1操作)
#endif
            }
            else
            {

                /* Restore interrupts.  */
                TX_RESTORE
            }
        }
    }

#ifdef TX_SAFETY_CRITICAL

    /* If we ever get here, raise safety critical exception.  */
    TX_SAFETY_CRITICAL_EXCEPTION(__FILE__, __LINE__, 0);
#endif

}
#endif

```