(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

C 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 irg 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IRO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ro, lr寄存器(BL指令会修改lr寄存器;c函数ro-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ro-r3,_tx_timer_interrupt调用irq_ack之后的代码很显然没有用到ro-r3的旧的值,所以ro-r3是不需要保护的,但是仿照ThreadX的其他代码,为了让栈保持8byte对齐,还是把ro保存到栈里面了),irg_ack就是c代码清除定时器中断:

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:

```
1. 115 _tx_timer_interrupt:
2. 116 @
3. 117 @ /* IRQ acknowledge. */
4. 118 @ irq_ack();
5. 119 @
 6. 120
          STMDB sp!, \{r0, lr\}
                                                   @ Save the lr register on the
   stack
7. 121
          BL
             irq ack // 清除定时器中断
8. 122
          LDMIA sp!, {r0, lr}
                                                   @ Recover lr register
9. 123 @
10. 124 @ /* Upon entry to this routine, it is assumed that context save has already
11. 125 @
            been called, and therefore the compiler scratch registers are available
12. 126 @
            for use. */
13. 127 @
14. 128 @ /* Increment the system clock. */
15. 129 @ _tx_timer_system_clock++;
16. 130 @
17. 131
               r1, =_tx_timer_system_clock @ Pickup address of system clock
          LDR
18. 132
          LDR
               r0, [r1]
                                                   @ Pickup system clock
19. 133
                  r0, r0, #1
                                                   @ Increment system clock //
          ADD
   _tx_timer_system_clock++,每次定时器中断,系统的时钟加1
20. 134
          STR
                r0, [r1]
                                                 @ Store new system clock
21. 135 @
22. 136 @ /* Test for time-slice expiration. */
23. 137 @ if (_tx_timer_time_slice)
24. 138 @
         {
25. 139 @
26. 140
          LDR
                 r3, =_tx_timer_time_slice
                                                 @ Pickup address of time-slice
                                                  @ Pickup time-slice
27. 141
               r2, [r3]
          LDR
28. 142
               r2, #0
                                                   @ Is it non-active?
          CMP
```

```
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
30. 144 @
31. 145 @
           /* Decrement the time slice. */
32. 146 @
        tx timer time slice--;
33. 147 @
34. 148
                                              @ Decrement the time-slice // 线
         SUB
                r2, r2, #1
   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)
35. 149
         STR
               r2, [r3]
                                              @ Store new time-slice value
36. 150 @
        /* Check for expiration. */
37. 151 @
38. 152 @ if ( tx timer time slice == 0)
39. 153 @
40. 154 CMP r2, #0
                                              @ Has it expired? // 检查是否超
   时(线程时间片用尽了)
                tx timer no time slice @ No, skip expiration processing
41. 155
         BNE
   // _tx_timer_time_slice不为0表示还有时间片, 跳转到__tx_timer_no_time_slice
42. 156 @
43. 157 @ /* Set the time-slice expired flag. */
44. 158 @
        tx timer expired time slice = TX TRUE;
45. 159 @
                r3, =_tx_timer_expired_time_slice @ Pickup address of expired flag
46. 160
         LDR
   // _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
47. 161
         MOV
                r0, #1
                                              @ Build expired value
48. 162
         STR
                r0, [r3]
                                              @ Set time-slice expiration flag
49. 163 @
50. 164 @
         }
51. 165 @
52. 166 __tx_timer_no_time_slice: // 检查完了线程时间片
53. 167 @
54. 168 @ /* Test for timer expiration. */
```

__tx_timer_no_time_slice

@ Yes, skip time-slice

29. 143

BEQ

```
55. 169 @ if (*_tx_timer_current_ptr)
56. 170 @
         {
57. 171 @
58. 172
                r1, =_tx_timer_current_ptr
                                              @ Pickup current timer pointer
          LDR
   address
59. 173
                 r0, [r1]
                                                @ Pickup current timer
          LDR
60. 174
                 r2, [r0]
                                                @ Pickup timer list entry //
          LDR
   *_tx_timer_current_ptr
61. 175
                                                @ Is there anything in the list?
          CMP
                 r2, #0
   // 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
62. 176
          BEO
                 tx timer no timer
                                                @ No, just increment the timer
   // 没有定时器的话, 跳转到 tx timer no timer
63. 177 @
         /* Set expiration flag. */
64. 178 @
65. 179 @
              tx timer expired = TX TRUE;
66. 180 @
67. 181
          LDR
                 r3, = tx timer expired
                                                @ Pickup expiration flag address
   // 有定时器超时,设置 tx timer expired标志
68. 182
          MOV
                r2, #1
                                                @ Build expired value
69. 183
          STR
                r2, [r3]
                                                @ Set expired flag
70. 184
                 tx timer done
                                                @ Finished timer processing
71. 185 @
72. 186 @
          }
73. 187 @
         else
74. 188 @
         {
75. 189 __tx_timer_no_timer: // 没有定时器超时
76. 190 @
             /* No timer expired, increment the timer pointer. */
77. 191 @
              _tx_timer_current_ptr++;
78. 192 @
79. 193 @
80. 194
                 r0, r0, #4
          ADD
                                                @ 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
```

81. 195 @

```
82. 196 @
               /* Check for wraparound. */
83. 197 @
               if (_tx_timer_current_ptr == _tx_timer_list_end)
84. 198 @
85. 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
86. 200
            LDR
                   r2, [r3]
                                                      @ Pickup list end
87. 201
                   r0, r2
                                                      @ Are we at list end?
            CMP
88. 202
            BNE
                   __tx_timer_skip_wrap
                                                      @ No, skip wraparound logic
89. 203 @
90. 204 @
                   /* Wrap to beginning of list. */
91. 205 @
                    _tx_timer_current_ptr = _tx_timer_list_start;
92. 206 @
93. 207
                   r3, = tx timer list start
                                                     @ Pickup address of timer list
            LDR
    start
94. 208
                   r0, [r3]
                                                      @ Set current pointer to list
            LDR
    start
95. 209 @
96. 210 __tx_timer_skip_wrap:
97. 211 @
98. 212
                                                      @ Store new current timer
            STR
                   r0, [r1]
    pointer
99. 213 @
            }
100. 214 @
101. 215 __tx_timer_done: // 检查完了定时器
102. 216 @
103. 217 @
104. 218 @
            /* See if anything has expired. */
105. 219 @
            if (( tx timer expired time slice) || ( tx timer expired))
106. 220 @
             {
107. 221 @
108. 222
                   r3, =_tx_timer_expired_time_slice @ Pickup address of expired flag
            LDR
109. 223
            LDR
                   r2, [r3]
                                                      @ Pickup time-slice expired flag
```

```
110. 224
           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)
111. 225
                  __tx_something_expired
                                                @ If non-zero, time-slice
           BNE
    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
112. 226
          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
113. 227
           LDR
                 r0, [r1]
                                                @ Pickup timer expired flag
114. 228
          CMP
                 r0, #0
                                                @ Did a timer expire?
115. 229
                  tx timer nothing expired
                                                @ No, nothing expired // 定时器
          BEO
    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)
116. 230 @
117. 231 __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)
118. 232 @
119. 233 @
120, 234
         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)
121. 235
                                                    and save r0 just to keep 8-
    byte alignment
122. 236 @
         /* Did a timer expire? */
123. 237 @
124. 238 @
          if ( tx timer expired)
125. 239 @
           {
126. 240 @
127. 241
           LDR
               r1, = tx timer expired
                                               @ Pickup address of expired flag
128. 242
                 r0, [r1]
                                                @ Pickup timer expired flag
           LDR
129. 243
          CMP
                 r0, #0
                                                @ Check for timer expiration
130. 244
                  __tx_timer_dont_activate
                                                @ If not set, skip timer
           BEQ
    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
131. 245 @
132. 246 @
          /* Process timer expiration. */
133. 247 @
             _tx_timer_expiration_process();
134. 248 @
```

```
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
136. 250 @
137. 251 @ }
138. 252 tx timer dont activate:
139. 253 @
          /* Did time slice expire? */
140. 254 @
141. 255 @ if (tx timer expired time slice)
142. 256 @
          {
143. 257 @
144. 258
           LDR
                 r3, = tx timer expired time slice @ Pickup address of time-slice
    expired
145. 259
           LDR
                 r2, [r3]
                                                  @ Pickup the actual flag
146. 260
           CMP
                  r2, #0
                                                  @ See if the flag is set // 检查
    线程时间片是否用尽
                  __tx_timer_not_ts_expiration @ No, skip time-slice processing
147. 261
           BEQ
    // 线程时间片没有用尽,跳转到__tx_timer_not_ts_expiration
148. 262 @
149. 263 @
          /* Time slice interrupted thread. */
150. 264 @
               _tx_thread_time_slice();
151. 265 @
152. 266 BL
                  _tx_thread_time_slice
                                                 @ Call time-slice processing //
    调用线程时间片处理函数_tx_thread_time_slice
153. 267 @
154. 268 @
          }
155. 269 @
156. 270 __tx_timer_not_ts_expiration:
157. 271 @
158. 272 LDMIA sp!, {r0, lr}
                                                  @ Recover lr register (r0 is
    just there for
159. 273
                                                     the 8-byte stack alignment
160. 274 @
161. 275 @
          }
162. 276 @
```

_tx_timer_expiration_process

@ Call the timer expiration

135. 249

BL

```
163. 277 __tx_timer_nothing_expired:

164. 278 @

165. 279 #ifdef __THUMB_INTERWORK

166. 280 BX lr @ Return to caller

167. 281 #else

168. 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)
```

169. 283 #endif

170. 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:

```
1. 079 VOID _tx_thread_time_slice(VOID)
2. 080 {
3. 081
4. 082 TX_INTERRUPT_SAVE_AREA
5. 083
6. 084 TX THREAD
                       *thread ptr;
7. 085 #ifdef TX ENABLE STACK CHECKING
8. 086 TX THREAD
                     *next thread ptr;
9. 087 #endif
10. 088 #ifdef TX_ENABLE_EVENT_TRACE
11. 089 ULONG
                     system state;
12. 090 UINT
                       preempt_disable;
13. 091 #endif
14. 092
15. 093
       /* Pickup thread pointer. */
16. 094
           TX_THREAD_GET_CURRENT(thread_ptr) // 获取当前正在执行的线程
   _tx_thread_current_ptr
17. 095
18. 096 #ifdef TX_ENABLE_STACK_CHECKING
19. 097
20. 098
          /* Check this thread's stack. */
21. 099
        TX_THREAD_STACK_CHECK(thread_ptr)
22. 100
23. 101
          /* Set the next thread pointer to NULL. */
24. 102
          next_thread_ptr = TX_NULL;
25. 103 #endif
26. 104
27. 105
          /* Lockout interrupts while the time-slice is evaluated. */
28. 106
          TX_DISABLE // 美闭中断
29. 107
30. 108
          /* Clear the expired time-slice flag. */
           _tx_timer_expired_time_slice = TX_FALSE; // 清除线程时间片用尽标志
31. 109
```

```
/* Make sure the thread pointer is valid. */
33. 111
34. 112
         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)
35. 113
         {
36, 114
37. 115
            /* Make sure the thread is still active, i.e. not suspended. */
38. 116
            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)
39. 117
            {
40. 118
41. 119
               /* Setup a fresh time-slice for the thread. */
42. 120
               thread ptr -> tx thread time slice = thread ptr ->
   tx thread new time slice; // 重新设置线程时间片
43. 121
44. 122
               /* Reset the actual time-slice variable. */
45. 123
                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
46. 124
47. 125
                /* Determine if there is another thread at the same priority and
   preemption-threshold
48. 126
                  is not set. Preemption-threshold overrides time-slicing. */
49. 127
                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
50. 128
                {
51. 129
52. 130
                   /* Check to see if preemption-threshold is not being used. */
53. 131
                   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
```

54. 132

{

```
56. 134
                          /* Preemption-threshold is not being used by this thread.
   */
57. 135
58. 136
                          /* There is another thread at this priority, make it the
   highest at
59, 137
                             this priority level. */
60. 138
                          _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)
61. 139
62. 140
                           /* Designate the highest priority thread as the one to
   execute. Don't use this
63. 141
                             thread's priority as an index just in case a higher
   priority thread is now
                             ready! */
64, 142
65. 143
                           _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
66, 144
67. 145 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
68. 146
69. 147
                          /* Increment the thread's time-slice counter. */
70. 148
                          thread ptr -> tx thread performance time slice count++;
71. 149
72. 150
                          /* Increment the total number of thread time-slice
   operations. */
73. 151
                          tx thread performance time slice count++;
74. 152 #endif
75. 153
76. 154
77. 155 #ifdef TX ENABLE STACK CHECKING
78. 156
79. 157
                          /* Pickup the next execute pointer. */
80. 158
                          next_thread_ptr = _tx_thread_execute_ptr;
81. 159 #endif
```

```
82. 160
                         }
 83. 161
                     }
 84. 162
                 }
 85. 163
             }
 86. 164
 87. 165 #ifdef TX ENABLE EVENT TRACE
 88. 166
 89. 167
             /* Pickup the volatile information. */
 90. 168
             system_state = TX_THREAD_GET_SYSTEM_STATE();
 91. 169
             preempt_disable = _tx_thread_preempt_disable;
 92. 170
 93. 171
            /* Insert this event into the trace buffer. */
             TX_TRACE_IN_LINE_INSERT(TX_TRACE_TIME_SLICE, _tx_thread_execute_ptr,
 94. 172
     system_state, preempt_disable, TX_POINTER_TO_ULONG_CONVERT(&thread_ptr),
     TX TRACE INTERNAL EVENTS)
 95. 173 #endif
 96. 174
            /* Restore previous interrupt posture. */
 97. 175
             TX RESTORE // 开启中断
 98. 176
 99. 177
100. 178 #ifdef TX_ENABLE_STACK_CHECKING
101. 179
102. 180
             /* Determine if there is a next thread pointer to perform stack checking on.
     */
103. 181
             if (next_thread_ptr != TX_NULL)
104. 182
             {
105. 183
106. 184
                 /* Yes, check this thread's stack. */
                TX_THREAD_STACK_CHECK(next_thread_ptr)
107. 185
108. 186
             }
109. 187 #endif
110. 188 }
```

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:

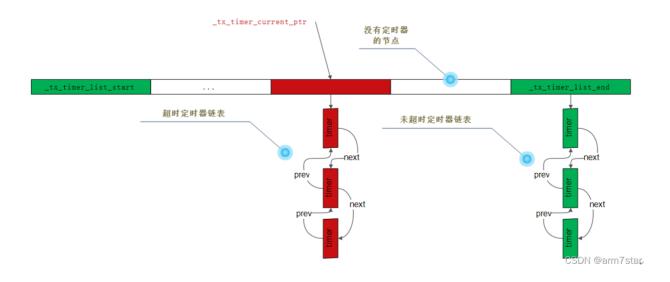
_tx_timer_current_ptr指向超时的定时器链表(_tx_timer_list对应的元素)。

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[o],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[o],下一次定时器中断就认为 _tx_timer_list[o]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[o],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[o]即可。

ThreadX招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:

```
1. 077 VOID _tx_timer_thread_entry(ULONG timer_thread_input)
2. 078 {
3. 079
4. 080 TX_INTERRUPT_SAVE_AREA
5. 081
6. 082 TX TIMER INTERNAL
                              *expired timers;
7. 083 TX_TIMER_INTERNAL
                                   *reactivate_timer;
8. 084 TX TIMER INTERNAL
                                   *next timer;
9. 085 TX TIMER INTERNAL
                                  *previous_timer;
10. 086 TX_TIMER_INTERNAL
                                   *current_timer;
11. 087 VOID
                                   (*timeout function)(ULONG id);
12. 088 ULONG
                                   timeout_param = ((ULONG) 0);
13. 089 TX_THREAD
                                   *thread_ptr;
14. 090 #ifdef TX REACTIVATE INLINE
15. 091 TX TIMER INTERNAL
                         **timer list;
                                                             /* Timer list pointer
   */
16. 092 UINT
                                                             /* Value used for
                                   expiration time;
   pointer offset*/
17. 093 ULONG
                                   delta;
18. 094 #endif
19. 095 #ifdef TX_TIMER_ENABLE_PERFORMANCE_INFO
20. 096 TX_TIMER
                                   *timer_ptr;
21. 097 #endif
22. 098
23. 099
24. 100
        /* Make sure the timer input is correct. This also gets rid of the
25. 101
             silly compiler warnings. */
26. 102
          if (timer_thread_input == TX_TIMER_ID)
27. 103
           {
28. 104
              /* Yes, valid thread entry, proceed... */
29. 105
30. 106
```

```
31. 107
              /* Now go into an infinite loop to process timer expirations. */
32. 108
              while (TX_LOOP_FOREVER)
33. 109
              {
34. 110
35. 111
                  /* First, move the current list pointer and clear the timer
36, 112
                     expired value. This allows the interrupt handling portion
37. 113
                     to continue looking for timer expirations. */
38. 114
                  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
39. 115
40. 116
                  /* Save the current timer expiration list pointer. */
41. 117
                  expired_timers = *_tx_timer_current_ptr; // 取出超时定时器链表
42. 118
43. 119
                  /* Modify the head pointer in the first timer in the list, if there
44. 120
                     is one! */
45. 121
                  if (expired timers != TX NULL)
46. 122
                  {
47. 123
48. 124
                      expired timers -> tx timer internal list head =
   &expired timers;
49. 125
50. 126
51. 127
                  /* Set the current list pointer to NULL. */
                  *_tx_timer_current_ptr = TX_NULL; // _tx_timer_current_ptr设置为
52. 128
   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
53. 129
54. 130
                  /* Move the current pointer up one timer entry wrap if we get to
55. 131
                     the end of the list. */
56. 132
                  _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
57. 133
                  if (_tx_timer_current_ptr == _tx_timer_list_end) // 链表回环
58. 134
                  {
59. 135
```

```
60. 136
                     _tx_timer_current_ptr = _tx_timer_list_start;
61. 137
                 }
62. 138
63. 139
                /* Clear the expired flag. */
64. 140
                 _tx_timer_expired = TX_FALSE; // 清除定时器超时_tx_timer_expired
65. 141
66. 142
                 /* Restore interrupts temporarily. */
67. 143
                 TX RESTORE // 允许中断,可以对软件定时器计时了
68. 144
69. 145
                 /* Disable interrupts again. */
70. 146
                 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)
71. 147
72. 148
                 /* Next, process the expiration of the associated timers at this
73. 149
                    time slot. */
74. 150
                 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
75. 151
76. 152
77. 153
                     /* Something is on the list. Remove it and process the
   expiration. */
78. 154
                     current_timer = expired_timers;
79. 155
80. 156
                    /* Pickup the next timer. */
81. 157
                     next_timer = expired_timers -> tx_timer_internal_active_next;
82. 158
83. 159
                     /* Set the reactivate_timer to NULL. */
84. 160
                     reactivate_timer = TX_NULL;
85, 161
86. 162
                     /* Determine if this is the only timer. */
87. 163
                     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)
88. 164
                     {
```

```
89. 165
90. 166
                            /* Yes, this is the only timer in the list. */
91. 167
92. 168
                            /* Set the head pointer to NULL. */
93. 169
                            expired_timers = TX_NULL; // 清空超时定时器链表即可
94, 170
                        }
95. 171
                        else // 将当前处理的超时定时器从链表中删除
96. 172
97. 173
                            /* No, not the only expired timer. */
98. 174
99. 175
100, 176
                            /* Remove this timer from the expired list. */
101. 177
                            previous_timer =
    current_timer -> tx_timer_internal_active_previous;
102. 178
                            next_timer -> tx_timer_internal_active_previous =
    previous_timer;
103. 179
                            previous_timer -> tx_timer_internal_active_next =
    next timer;
104. 180
105. 181
                            /* Modify the next timer's list head to point at the current
    list head. */
106. 182
                            next_timer -> tx_timer_internal_list_head =
    &expired_timers;
107. 183
108. 184
                            /* Set the list head pointer. */
109. 185
                            expired_timers = next_timer;
110. 186
                        }
111. 187
112. 188
                        /* In any case, the timer is now off of the expired list. */
113. 189
114. 190
                        /* Determine if the timer has expired or if it is just a really
115. 191
                           big timer that needs to be placed in the list again. */
116. 192
                        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
117. 193
                        {
```

```
118. 194
119. 195
                            /* Timer is bigger than the timer entries and must be
120. 196
                               rescheduled. */
121. 197
122. 198 #ifdef TX TIMER ENABLE PERFORMANCE INFO
123. 199
124. 200
                            /* Increment the total expiration adjustments counter. */
125. 201
                            _tx_timer_performance__expiration_adjust_count++;
126. 202
                            /* Determine if this is an application timer. */
127. 203
128. 204
                            if (current_timer -> tx_timer_internal_timeout_function !=
    & tx thread timeout)
129. 205
                             {
130. 206
131. 207
                                /* Derive the application timer pointer. */
132. 208
133. 209
                                /* Pickup the application timer pointer. */
134. 210
                                TX_USER_TIMER_POINTER_GET(current_timer, timer_ptr)
135. 211
136. 212
                                /* Increment the number of expiration adjustments on
    this timer. */
137. 213
                                if (timer_ptr -> tx_timer_id == TX_TIMER_ID)
138. 214
                                {
139. 215
140. 216
                                    timer_ptr ->
    tx_timer_performance__expiration_adjust_count++;
141. 217
                                }
142. 218
                             }
143. 219 #endif
144. 220
145. 221
                            /* Decrement the remaining ticks of the timer. */
146. 222
                            current_timer -> tx_timer_internal_remaining_ticks =
147. 223
                                     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
```

```
149. 225
                          /* Set the timeout function to NULL in order to bypass the
150. 226
                             expiration. */
151. 227
                           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
152. 228
153, 229
                          /* Make the timer appear that it is still active while
    interrupts
154. 230
                             are enabled. This will permit proper processing of a
    timer
155. 231
                             deactivate from an ISR. */
156. 232
                          current timer -> tx timer internal list head =
    &reactivate_timer;
157. 233
                          current_timer -> tx_timer_internal_active_next =
    current_timer;
158. 234
                          /* Setup the temporary timer list head pointer. */
159. 235
160. 236
                          reactivate_timer = current_timer; // 需要重新激活的定时器
161. 237
                       }
                       else // 剩余时间少于TX_TIMER_ENTRIES是不会再起定时器了, 毕竟代码
162. 238
    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
163. 239
                       {
164. 240
165. 241
                          /* Timer did expire. */
166. 242
167. 243 #ifdef TX_TIMER_ENABLE_PERFORMANCE_INFO
168. 244
169. 245
                          /* Increment the total expirations counter. */
170. 246
                          _tx_timer_performance_expiration_count++;
171. 247
172. 248
                          /* Determine if this is an application timer. */
173. 249
                          if (current_timer -> tx_timer_internal_timeout_function !=
    &_tx_thread_timeout)
174. 250
                           {
```

175. 251

```
176. 252
                                /* Derive the application timer pointer. */
177. 253
178. 254
                                /* Pickup the application timer pointer. */
179. 255
                                TX_USER_TIMER_POINTER_GET(current_timer, timer_ptr)
180. 256
181. 257
                                /* Increment the number of expirations on this timer.
     */
182. 258
                                if (timer_ptr -> tx_timer_id == TX_TIMER_ID)
183. 259
                                {
184. 260
185. 261
                                    timer ptr ->
     tx timer performance expiration count++;
186. 262
                                }
187. 263
                            }
188. 264 #endif
189. 265
190. 266
                            /* Copy the calling function and ID into local variables
    before interrupts
191. 267
                               are re-enabled. */
192. 268
                            timeout function = current timer ->
    tx timer internal timeout function; // 定时器超时回调函数
193. 269
                            timeout param =
                                               current timer ->
     tx_timer_internal_timeout_param; // 定时器超时回调函数指针
194. 270
195. 271
                            /* Copy the reinitialize ticks into the remaining ticks. */
196. 272
                            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
197. 273
198. 274
                            /* Determine if the timer should be reactivated. */
199. 275
                            if (current_timer -> tx_timer_internal_remaining_ticks !=
    ((ULONG) 0))
200. 276
                            {
201. 277
202. 278
                                /* Make the timer appear that it is still active while
     processing
```

```
203. 279
                                   the expiration routine and with interrupts enabled.
    This will
204. 280
                                   permit proper processing of a timer deactivate from
    both the
205. 281
                                   expiration routine and an ISR. */
206. 282
                                current timer -> tx timer internal list head =
    &reactivate timer;
207. 283
                                current_timer -> tx_timer_internal_active_next =
    current timer;
208. 284
209. 285
                                /* Setup the temporary timer list head pointer. */
210. 286
                                reactivate_timer = current_timer; // 需要重新激活的定时
    器
211. 287
                            }
212. 288
                            else
213. 289
                            {
214. 290
215. 291
                                /* Set the list pointer of this timer to NULL. This is
    used to indicate
216. 292
                                   the timer is no longer active. */
217. 293
                                current_timer -> tx_timer_internal_list_head = TX_NULL;
218. 294
                            }
219. 295
                        }
220. 296
                        /* Set pointer to indicate the expired timer that is currently
221. 297
    being processed. */
                        _tx_timer_expired_timer_ptr = current_timer; // 正在处理的定时
222. 298
     器,避免其他线程操作该定时器
223. 299
224. 300
                        /* Restore interrupts for timer expiration call. */
225. 301
                        TX RESTORE
226. 302
227. 303
                        /* Call the timer-expiration function, if non-NULL. */
228. 304
                        if (timeout_function != TX_NULL)
229. 305
                        {
230. 306
```

```
231. 307
                            (timeout_function) (timeout_param); // 定时器回调函数
232. 308
                        }
233. 309
234. 310
                        /* Lockout interrupts again. */
235. 311
                        TX DISABLE
236. 312
237. 313
                        /* Clear expired timer pointer. */
238. 314
                        _tx_timer_expired_timer_ptr = TX_NULL;
239. 315
                        /* Determine if the timer needs to be reactivated. */
240. 316
241. 317
                        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
242. 318
                        {
243. 319
244. 320
                            /* Reactivate the timer. */
245. 321
246. 322 #ifdef TX TIMER ENABLE PERFORMANCE INFO
247. 323
248. 324
                            /* Determine if this timer expired. */
249. 325
                            if (timeout function != TX NULL)
250. 326
                            {
251. 327
252. 328
                                /* Increment the total reactivations counter. */
253. 329
                                _tx_timer_performance_reactivate_count++;
254. 330
255. 331
                                /* Determine if this is an application timer. */
256. 332
                                if (current_timer -> tx_timer_internal_timeout_function
    != &_tx_thread_timeout)
257. 333
                                {
258. 334
259. 335
                                    /* Derive the application timer pointer. */
260. 336
261. 337
                                    /* Pickup the application timer pointer. */
```

```
262. 338
                                     TX_USER_TIMER_POINTER_GET(current_timer, timer_ptr)
263. 339
264. 340
                                     /* Increment the number of expirations on this
    timer. */
265. 341
                                     if (timer_ptr -> tx_timer_id == TX_TIMER_ID)
266. 342
                                     {
267. 343
268. 344
                                         timer_ptr ->
    tx timer performance reactivate count++;
269. 345
                                     }
270. 346
                                 }
271. 347
                             }
272. 348 #endif
273. 349
274. 350 #ifdef TX REACTIVATE INLINE
275. 351
276. 352
                             /* Calculate the amount of time remaining for the timer. */
                             if (current timer -> tx timer internal remaining ticks >
277. 353
    TX TIMER ENTRIES)
278. 354
                             {
279. 355
280. 356
                                 /* Set expiration time to the maximum number of entries.
    */
281. 357
                                 expiration time = TX TIMER ENTRIES - ((UINT) 1);
282. 358
                             }
283. 359
                             else
284. 360
                             {
285. 361
286. 362
                                 /* Timer value fits in the timer entries. */
287. 363
288. 364
                                 /* Set the expiration time. */
289. 365
                                 expiration_time = ((UINT) current_timer ->
    tx_timer_internal_remaining_ticks) - ((UINT) 1);
290. 366
                             }
```

```
291. 367
292. 368
                            /* At this point, we are ready to put the timer back on one
    of
293. 369
                                the timer lists. */
294. 370
295. 371
                            /* Calculate the proper place for the timer. */
296. 372
                             timer_list = TX_TIMER_POINTER_ADD(_tx_timer_current_ptr,
     expiration_time);
297. 373
                             if (TX TIMER INDIRECT TO VOID POINTER CONVERT(timer list) >=
    TX_TIMER_INDIRECT_TO_VOID_POINTER_CONVERT(_tx_timer_list_end))
298. 374
                             {
299. 375
300.376
                                 /* Wrap from the beginning of the list. */
301. 377
                                 delta = TX_TIMER_POINTER_DIF(timer_list,
    tx timer list end);
302. 378
                                 timer_list = TX_TIMER_POINTER_ADD(_tx_timer_list_start,
    delta);
303. 379
                             }
304. 380
305. 381
                            /* Now put the timer on this list. */
306. 382
                            if ((*timer list) == TX NULL)
307. 383
                             {
308. 384
309.385
                                 /* This list is NULL, just put the new timer on it. */
310. 386
                                 /* Setup the links in this timer. */
311. 387
312. 388
                                 current timer -> tx timer internal active next =
     current_timer;
313. 389
                                 current_timer -> tx_timer_internal_active_previous =
     current_timer;
314. 390
315. 391
                                 /* Setup the list head pointer. */
316. 392
                                 *timer_list = current_timer;
317. 393
                             }
318. 394
                             else
```

```
319. 395
                             {
320. 396
321. 397
                                /* This list is not NULL, add current timer to the end.
    */
322. 398
                                next_timer =
    *timer list;
323. 399
                                previous_timer =
     next_timer -> tx_timer_internal_active_previous;
324. 400
                                previous_timer -> tx_timer_internal_active_next =
    current timer;
325. 401
                                next_timer -> tx_timer_internal_active_previous =
    current timer;
326. 402
                                current timer -> tx timer internal active next =
    next_timer;
                                current_timer -> tx_timer_internal_active_previous =
327. 403
    previous timer;
328. 404
                            }
329. 405
                            /* Setup list head pointer. */
330. 406
331. 407
                            current_timer -> tx_timer_internal_list_head = timer_list;
332. 408 #else
333. 409
                            /* Reactivate through the timer activate function. */
334. 410
335. 411
336. 412
                            /* Clear the list head for the timer activate call. */
337. 413
                            current_timer -> tx_timer_internal_list_head = TX_NULL;
338. 414
339. 415
                            /* Activate the current timer. */
340. 416
                            _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)
341. 417 #endif
342. 418
                        }
343. 419
344. 420
                        /* Restore interrupts. */
345. 421
                        TX_RESTORE
346. 422
```

```
347. 423
                        /* Lockout interrupts again. */
348. 424
                        TX_DISABLE
349. 425
                    }
350. 426
351. 427
                    /* Finally, suspend this thread and wait for the next expiration.
    */
352. 428
353. 429
                   /* Determine if another expiration took place while we were in this
                       thread. If so, process another expiration. */
354. 430
                    if (tx timer expired == TX FALSE) // 定时器中断没有激活新的超时定时
355. 431
    器,那么需要挂起自己
356. 432
                    {
357. 433
358. 434
                        /* Otherwise, no timer expiration, so suspend the thread. */
359. 435
360. 436
                        /* Build pointer to the timer thread. */
361. 437
                        thread_ptr = &_tx_timer_thread;
362. 438
363. 439
                        /* Set the status to suspending, in order to indicate the
364. 440
                          suspension is in progress. */
365. 441
                        thread_ptr -> tx_thread_state = TX_SUSPENDED;
366. 442
367. 443 #ifdef TX NOT INTERRUPTABLE
368. 444
369. 445
                       /* Call actual non-interruptable thread suspension routine. */
                        _tx_thread_system_ni_suspend(thread_ptr, ((ULONG) 0));
370. 446
371. 447
372. 448
                        /* Restore interrupts. */
373. 449
                        TX RESTORE
374. 450 #else
375. 451
376. 452
                        /* Set the suspending flag. */
377. 453
                        thread_ptr -> tx_thread_suspending = TX_TRUE;
```

```
379. 455
                     /* Increment the preempt disable count prior to suspending. */
380. 456
                     _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)
381. 457
382. 458
                     /* Restore interrupts. */
383. 459
                     TX RESTORE
384. 460
385. 461
                     /* Call actual thread suspension routine. */
386. 462
                     _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)
387. 463 #endif
388. 464
                  }
389. 465
                  else
390. 466
                  {
391. 467
                     /* Restore interrupts. */
392. 468
                     TX_RESTORE
393. 469
394. 470
                  }
395. 471
             }
396. 472
        }
397. 473
398. 474 #ifdef TX_SAFETY_CRITICAL
399. 475
        /* If we ever get here, raise safety critical exception. */
400. 476
401. 477
           TX_SAFETY_CRITICAL_EXCEPTION(__FILE__, __LINE__, 0);
402. 478 #endif
403. 479
404. 480 }
405. 481 #endif
406. 482
```