(163条消息) ThreadX中断处理(基于ARM处理器)_arm7star的博客-CSDN博客_threadx 中断

1、s3c2440定时器初始化

定时器用于内核时钟计数，一用于对当前正在运行的任务的时间片进行更新，二用于对内核timer更新。

定时器初始化函数如下，调用s3c2440_timer_init设置时钟周期，调用unmask_irq允许定时器4中断。

```
1. void SMDK2440_Timer_Initialize(void)
2. {
3.     s3c2440_timer_init();   // initialize timer4
4.     unmask_irq(INT_TIMER4); // unmask timer4 interrupt
5. }
```

```
1. void s3c2440_timer_init(void)
2. {
3.     // Timer input clock Frequency = PCLK / {prescaler value+1} / {divider value}
4.     TCFG0 |= (99 << 8); // 预分频器2,3,4 = 99
5.     TCFG1  = (3 << 16); // MUX 4 0011 = 1/16, 16分频
6.     TCNTB4 = 625;
7.     TCON  |= (1 << 21); // auto reload/Update TCNTB4/Start for Timer 4
8.     TCON   =  5 << 20;  // No operation
9. }
```

2、irq中断入口

中断向量入口如下，irq中断发生时，硬件自动跳转到0x00000018地址执行代码，通过"ldr pc, _irq"指令跳转到irq处理函数。

```
1. .global _start
2. _start:
3.         ldr pc, ResetAddr
4.         ldr pc, _undefined_instruction
5.         ldr pc, _software_interrupt
6.         ldr pc, _prefetch_abort
7.         ldr pc, _data_abort
8.         ldr pc, _not_used
9.         ldr pc, _irq
10.        ldr pc, _fiq
11. ResetAddr:
12.        .word reset
13. _undefined_instruction:
14.        .word undefined_instruction
15. _software_interrupt:
16.        .word software_interrupt
17. _prefetch_abort:
18.        .word prefetch_abort
19. _data_abort:
20.        .word data_abort
21. _not_used:
22.        .word not_used
23. _irq:
24.        .word irq
25. _fiq:
26.        .word fiq
27. _pad:
28.        .word 0x12345678 /* now 16*4=64 */
29. .global _end_vect
30. _end_vect:
31.        .balignl 16,0xdeadbeef
```

irq处理函数如下，发生irq中断时，硬件将当前cpu的pc寄存器减4并保存到irq模式的lr寄存器中，cpu发送中断时的pc指向正在取指的指令地址而不是正在执行的指令的地址，硬件将pc - 4后，指向正在译码的指令的地址，正在执行的指令的地址仍需要再减4。

"sub　lr, lr, #4"计算中断返回地址；"stmfd sp!, {r0-r3, r12, lr}"，将{r0,r1,r2,r3,r12,pc}入IRQ模式的栈，这几个寄存器都是通用寄存器，后续代码需要用到这些寄存器，需要先保存一下；"bl　_tx_timer4_interrupt"调用定时器处理函数(本文中没有用到其他中断，暂时没有处理其他中断的必要，简化处理过程，直接调用定时器中断处理函数即可)；

```
1. /*
2.  * IRQ interrupt
3.  */
4.         .align  5
5. irq:
6.         sub    lr, lr, #4
7.         stmfd  sp!, {r0-r3, r12, lr} // 保存中断寄存器，IRQ栈:
   r0,r1,r2,r3,r12,pc(lr)
8.         bl     _tx_timer4_interrupt
9.         ldmfd  sp!, {r0-r3, r12, lr}
10.        movs   pc,  lr
```

3、定时器中断处理

编译器使用r0-r3作为通用寄存器，c函数内部不保存恢复r0-r3寄存器，对于其他c函数需要用到的寄存器，编译器会在使用前保存并在函数返回前恢复，因此如果r0-r3有数据的话，需要在调用c函数前自己保存，在irq函数中已经保存了r0-r3寄存器，因此_tx_timer4_interrupt调用c函数前都没有做寄存器保存工作。

"bl Timer4_Exception"调用Timer4_Exception清除定时器中断，否则中断将持续发生；

”bl　_tx_timer_interrupt“调用内核的_tx_timer_interrupt函数，对系统时钟、当前执行的任务的时间片、定时器进行更新(定时器超时，resume _tx_timer_thread线程；任务剩余时间片为0，调用_tx_thread_time_slice，在可抢占情况下，调度同一优先级的下一个任务，任务时间片用完了并不会唤醒更高优先级的任务，因此只需检查同优先级是否有就绪任务即可)

"bl　__tx_thread_preempt_check"检查可执行的任务是否为当前被中断的任务(当前任务不可抢占、当前任务时间片没有用完、同一优先级只有当前任务)，如果是(__tx_thread_preempt_check返回0)，则这需要恢复中断上下文返回被中断的任务即可，如果不是(__tx_thread_preempt_check返回非0)，则需要保存当前被中断的任务的上下文，切换到另外一个就绪任务执行。

__tx_timer_nothing_expired: 中断后仍返回当前被中断任务继续执行，"ldmfd sp!, {r0-r3, r12, pc}^"恢复保存在irq中的寄存器及cpsr(spsr)，ldmfd指令恢复的寄存器包含pc，那么^后缀将同时将spsr恢复到cpsr。

_irq_thread_swith_save_no: 中断后没有需要上下文保存，仅恢复IRQ模式的sp寄存器即可，在irq函数保存了部分寄存器并修改了IRQ的sp寄存器，这部分寄存器没有任何用处，简单修改sp即可；(在没有任何任务执行的情况下，_tx_thread_schedule在不停的循环，与NucleusPlus调度一样，_tx_thread_schedule函数的上下文不需要恢复，_tx_thread_schedule不在任何进程的上下文，_tx_thread_schedule被中断后不需要再恢复，只需要再次调用_tx_thread_schedule即可)

_irq_thread_swich_save_all: 保存当前被中断的任务上下文，重新调度；"stmfd sp!, {r0}"任务的cpsr寄存器入栈，"add sp, sp, #4*7"恢复IRQ模式的sp寄存器，后续将切换到SVC模式来保存上下文，将无法访问IRQ的sp寄存器。

```
    .global _tx_timer4_interrupt
    .type   _tx_timer4_interrupt,function
_tx_timer14_interrupt:
    bl      Timer4_Exception
    bl      _tx_timer_interrupt
    bl      __tx_thread_preempt_check
    cmp     r0, #4
    ldr     pc, [pc, r0, lsl #2] // ldr pc, _irq_thread_exit + __tx_thread_preempt_check返回值*4, 执行该命令时pc指向取值的指令地址，即 _irq_thread_exit
    /* Restore from interrupt without scheduing. */
__tx_timer_nothing_expired: // 直接恢复中断上下文
    ldmfd   sp!, {r0-r3, r12, pc}^         // Restore from interrupt
_irq_thread_exit:
    .word     __tx_timer_nothing_expired
    .word     _irq_thread_swith_save_no
    .word     _irq_thread_swich_save_all
    /* Pop minimal context on the current stack. */
_irq_thread_swith_save_no: // 不需要保存中断上下文(中断上下文已经保存过，不需要保存；没有就绪任务时，进入_tx_thread_schedule循环过程中，_tx_thread_current_ptr并没有并设置为NULL，因此_tx_thread_schedule被中断时，_tx_thread_current_ptr指向的并非正在执行的任务，另外_tx_thread_schedule并没有上下文要保存，_tx_thread_schedule不运行在任务上下文，_tx_thread_schedule每次调用都会重新设置栈，因此不存在内存泄漏问题)
    add     sp, sp, #4*6                   // Skip r0-r3, r12, lr on the current stack
    msr     cpsr_cxsf, #(SUP_MODE | I_BIT) // Switch to supervisor mode (SVC)
    b       _irq_thread_swith_exit
   /* Save complete context on the current stack. */
_irq_thread_swich_save_all: // 换出当前执行的进程，保存当前任务的上下文
    mrs     r0, spsr
    stmfd   sp!, {r0}                      // Save cpsr on the current stack, IRQ栈: cpsr(spsr),r0,r1,r2,r3,r12,pc(lr)
    add     sp, sp, #4*7                   // Restore sp
    bl      _tx_thread_context_save        // Save context on thead stack
_irq_thread_swith_exit:
```

```
28.     ldr     pc, =_tx_thread_schedule // 执行调度程序，调度新的任务
```

4、中断上下文保存

_tx_thread_context_save仅用于保存中断的上下文，在_tx_thread_context_save之前的函数都是用的stmfd将寄存器入栈，即sp指针指向有效的数据，寄存器入栈前，先将sp减4然后将寄存器存入sp指向的内存地址，实际数据存储在sp - 4的地址。

在_tx_thread_context_save之前，irq的栈已经通过"add sp, sp, #4*7"恢复了，因此被中断的任务的寄存器实际保存在sp - 4的地址。再看看ldmfa指令，该指令先出栈然后再修改sp寄存器，即先将sp指向的内存的数据加载到寄存器中，然后sp减4。

当前内存中已保存的数据为"cpsr(spsr),r0,r1,r2,r3,r12,pc(lr)"。

上下文保存看代码注释，主要是取出保存在内存中的寄存器的值并存储到当前任务的栈里面，然后设置任务的栈指针_tx_thread_current_ptr->tx_stack_ptr = sp，下次任务执行时，从_tx_thread_current_ptr->tx_stack_ptr即可恢复任务的上下文。

```
1. _tx_thread_context_save:

2.     sub     r0, sp, #4                      // Pickup stack bottom of saved context

3.     msr     cpsr_cxsf, #(SUP_MODE | I_BIT) // Switch to supervisor mode (SVC)，切换到SVC模式，ThreadX的任务运行在SVC模式(切换前cpsr,r0,r1,r2,r3,r12,pc已经保存在内存中)

4.     /* Save complete context on the current stack. */

5.     ldmfa   r0!, {r1}                       // Pop PC to r1, r0执行保存上下文内存的高地址，即pc，恢复中断的pc到r1寄存器，r0减4

6.     stmfd   sp!, {r1}                       // Push PC on the current stack，pc保存到任务的栈中(此时的sp已经是任务的sp了)，任务栈：pc

7.     ldmfa   r0!, {r1-r3, r12}               // Pop r1-r3, r12 to r1-r3, r12，从内存恢复r1,r2,r3,r12的内容到r1,r2,r3,r12寄存器

8.     stmfd   sp!, {r1-r12}                   // Push r1-r12 on the current stack，r1-r12保存到任务栈(r1-r3,r12上一条指令已经从内存恢复，r4-r11汇编代码没有使用，c函数即使使用了也会自动恢复，因此直接保存可以r4-r12中断程序并没有改变)，任务栈：r1-r12,pc

9.     ldmfa   r0!, {r1-r2}                    // Pop cpsr, r0 to r1-r2，cpsr,r0从内存恢复到cpsr,r0寄存器

10.    stmfd   sp!, {r2}                       // Push r0 on the current stack，r0入任务栈，任务栈：r0-r12,pc

11.    stmfd   sp!, {r1, lr}                   // Push cpsr, lr on the current stack，cpsr,lr入栈，任务栈：cpsr,lr,r0-r12,pc

12.    ldr     r0, =_tx_thread_current_ptr // 获取当前任务指针

13.    ldr     r0, [r0]

14.    str     sp, [r0, #8]                    // Store sp on tx_stack_ptr, _tx_thread_current_ptr->tx_stack_ptr = sp

15.    msr     cpsr_c, #(IRQ_MODE | I_BIT)    // Return to Interrupt Request mode (IRQ)

16.    mov     pc, lr                          // Return to Caller
```

5、任务上下文保存

任务处理时间片用完、被高优先级任务抢占外，还存在主动让出cpu的情况，例如：任务获取不到互斥锁等都会导致进程阻塞进而重新调度。_tx_thread_system_return即用于保存任务主动让出cpu时保存任务的上下文。

```
1.      .global _tx_thread_system_return

2.      .type   _tx_thread_system_return,function

3.  _tx_thread_system_return:

4.      mrs     r0,         spsr // 保存spsr到r0(c函数调用_tx_thread_system_return，上一级函数自己保存r0-r3的值，如果需要保存，调用_tx_thread_system_return前，r0-r3已经保存在栈里面了，调用完_tx_thread_system_return后，上级函数将自己恢复r0-r3，因此_tx_thread_system_return可以任意修改r0-r3寄存器，_tx_thread_system_return保存到任务栈里面的r0-r3没有实际意义

5.      mov     r1,         lr // 任务返回地址保存到lr(_tx_thread_system_return返回地址)

6.      /* Lockout interrupts.  */

7.      msr     cpsr_cxsf, #(SUP_MODE | I_BIT) // 关闭中断

8.     /* Save complete context on the current stack. */

9.      stmfd   sp!, {r1}                       // Push PC on the current stack，pc入栈，任务栈：pc

10.     stmfd   sp!, {r0-r12}                   // Push r0-r12 on the current stack，r0-r12入栈，任务栈：r0-r12,pc

11.     stmfd   sp!, {lr}                       // Push lr on the current stack，lr寄存器入栈，此处的lr寄存器也没有实际意义，任务恢复时直接返回到_tx_thread_system_return函数，任务栈：lr,r0-r12,pc

12.     stmfd   sp!, {r0}                       // Push cpsr on the current stack，cpsr入栈，任务栈：cpsr,lr,r0-r12,pc

13.     ldr     r0, =_tx_thread_current_ptr     // Pickup address of _tx_thread_current_ptr

14.     ldr     r0, [r0]                        // Pickup _tx_thread_current_ptr value

15.     str     sp, [r0, #8]                    // Store sp on tx_stack_ptr, _tx_thread_current_ptr->tx_stack_ptr = sp

16.     ldr     pc, =_tx_thread_schedule
```

6、任务上下文恢复

_tx_thread_context_restore用于任务上下文的恢复。

```
1.      .global _tx_thread_context_restore

2.      .type   _tx_thread_context_restore,function

3.  _tx_thread_context_restore:

4.      /* Restore previous context. */

5.      ldr     r0, =_tx_thread_current_ptr    // Pickup address of
    _tx_thread_current_ptr，获取任务指针(在_tx_thread_to_schedule中会将
    _tx_thread_execute_ptr的内容赋值给_tx_thread_current_ptr，_tx_thread_current_ptr用于
    进程上下文的保存恢复及时间片计数等，_tx_thread_execute_ptr代表当前可执行的最高优先级
    的任务)

6.      mov     r2, #0

7.      ldr     r0, [r0]                       // Pickup _tx_thread_current_ptr value

8.      ldr     r1, [r0, #8]                   // Pickup tx_stack_ptr value, r1 =
    _tx_thread_current_ptr->tx_stack_ptr

9.      str     r2, [r0, #8]                   // Store new tx_stack_ptr,
    _tx_thread_current_ptr->tx_stack_ptr = 0

10.     ldmfd   r1!, {r0, lr}                  // Pop cpsr, lr to r0, lr, cpsr,lr恢复到
    r0,lr寄存器

11.     mov     sp, r1                         // Store new sp value, r1赋值给sp

12.     msr     spsr_cxsf, r0                  // Store previous cpsr to spsr，r0恢复到
    spsr

13.     ldmfd   sp!, {r0-r12, pc}^             // Restore from context stack，恢复r0-
    r12,pc及spsr到cpsr
```

7、任务上下文创建

任务上下文保存及恢复已经清楚了，下面看下任务上下文的创建。内核初始化之后，在调用_tx_initialize_kernel_enter及_tx_thread_schedule来调度任务，_tx_thread_schedule调用_tx_thread_context_restore来执行任务，因此创建任务的上下文与_tx_thread_context_restore的上下文一致(有的系统可能有标记是被中断恢复上下文还是任务主动/第一次创建切换上下文，之前主动让出cpu时任务保存上下文中有介绍r0-r3没有实际意义，因此有的系统不同情况下寄存器恢复不一样，有的不恢复r0-r3，ThreaX没有区分，上下文全一样)。

ARM_STACK结构体定义如下，该结构体各寄存器顺序与上下文保存顺序一致。

```
typedef struct arm_stack_struct {
    unsigned int cpsr;
    unsigned int lr;
    unsigned int r0;
    unsigned int r1;
    unsigned int r2;
    unsigned int r3;
    unsigned int r6;
    unsigned int r4;
    unsigned int r5;
    unsigned int r7;
    unsigned int r8;
    unsigned int r9;
    unsigned int r10;
    unsigned int r11;
    unsigned int r12;
    unsigned int pc;
} ARM_STACK;
```

任务上下文的创建。

"ArmRegister = (ARM_STACK *)(((int)thread_ptr->tx_stack_end - sizeof(ARM_STACK) - 1) & 0xfffffffc);"将tx_stack_end减去一个值用于保存线程上下文(tx_stack_end - sizeof(ARM_STACK) - 1到tx_stack_end这段空间保存线程上下文)；

"ArmRegister->cpsr = MODE_SVC32;"设置任务运行于SVC模式；

"ArmRegister->lr = (int)(thread_ptr->tx_thread_entry); "任务执行完的返回地址，此处任务执行完后有重新执行该任务；

"ArmRegister->r0 = (unsigned int)thread_ptr->tx_entry_parameter; " 任务的参数；

"ArmRegister->pc = (int)(thread_ptr->tx_thread_entry); "任务的入口；

thread_ptr->tx_stack_ptr = (VOID_PTR)ArmRegister;任务栈顶，栈由高往低递减，栈底指向高地址，栈底指向低地址；

任务创建完成后，内核调用_tx_thread_context_restore恢复ArmRegister到cpu即可启动任务。

```
1. void _tx_thread_stack_build(TX_THREAD *thread_ptr, void (*function_ptr)(void))
2. {
3.     ARM_STACK    *ArmRegister   = 0;
4.     ArmRegister         = (ARM_STACK *)(((int)thread_ptr->tx_stack_end -
   sizeof(ARM_STACK) - 1) & 0xfffffffc);
5.     ArmRegister->cpsr   = MODE_SVC32;
6.     ArmRegister->lr     = (int)(thread_ptr->tx_thread_entry);
7.     ArmRegister->r0     = (unsigned int)thread_ptr->tx_entry_parameter;
8.     ArmRegister->r1     = 0;
9.     ArmRegister->r2     = 0;
10.    ArmRegister->r3     = 0;
11.    ArmRegister->r4     = 0;
12.    ArmRegister->r5     = 0;
13.    ArmRegister->r6     = 0;
14.    ArmRegister->r7     = 0;
15.    ArmRegister->r8     = 0;
16.    ArmRegister->r9     = 0;
17.    ArmRegister->r10    = 0;
18.    ArmRegister->r11    = 0;
19.    ArmRegister->r12    = 0;
20.    ArmRegister->pc     = (int)(thread_ptr->tx_thread_entry);
21.    thread_ptr->tx_stack_ptr    = (VOID_PTR)ArmRegister;
22. }
```

8、任务时间片

每次硬件定时器中断都调用_tx_timer_interrupt函数，对任务时间片减1。

_tx_timer_system_clock: 系统时钟，类似linux内核的jiffies，每个时钟周期加1；

_tx_timer_time_slice: 任务时间片，每个时钟周期减1。

```
1. VOID _tx_timer_interrupt(VOID)
2. {
3.     _tx_timer_system_clock++; // 系统时钟加1
4.     if (_tx_timer_time_slice) { // _tx_timer_time_slice不为0表示有任务时间片在计数，没有任务运行时_tx_timer_time_slice为0，每次任务调度时会将任务的时间片赋值给_tx_timer_time_slice，_tx_timer_time_slice记录当前任务的剩余时间片
5.                 _tx_timer_time_slice--; // 当前任务时间片减1
6.                 if (_tx_timer_time_slice == 0) { // 当前任务时间片已经耗尽
7.                         _tx_timer_expired_time_slice =  TX_TRUE; // 设置_tx_timer_expired_time_slice，任务时间片耗尽
8.                 }
9.     }
10.    if (*_tx_timer_current_ptr) {
11.        _tx_timer_expired =  TX_TRUE;
12.    } else {
13.        _tx_timer_current_ptr++;
14.        if (_tx_timer_current_ptr == _tx_timer_list_end){
15.            _tx_timer_current_ptr =  _tx_timer_list_start;
16.                }
17.    }
18.    if ((_tx_timer_expired_time_slice) || (_tx_timer_expired)) {
19.                if (_tx_timer_expired) {
20.                    _tx_timer_expired   = TX_FALSE;
21.                        _tx_thread_preempt_disable++;
22.                        _tx_thread_resume(&_tx_timer_thread);
23.                }
24.                if (_tx_timer_expired_time_slice) { // 任务时间片耗尽
25.                _tx_timer_expired_time_slice    = TX_FALSE; // _tx_timer_expired_time_slice设置为TX_FALSE，任务时间片耗尽后，要么重新调度新的任务(_tx_timer_expired_time_slice为FALSE)，要么没有其他可调度的任务(仍然执行当前任务，_tx_timer_expired_time_slice为FALSE)，所以此处必须设置_tx_thread_time_slice为FALSE
26.                        if (_tx_thread_time_slice() == TX_FALSE) { // _tx_thread_time_slice重新设置时间片，检查是否需要重新调度(如果禁止抢占，则任务时间片设置为1，等抢占允许后，其他就绪任务就可能更快得到调度，因为下一个时钟中断当前任务的时间片就用完了；如果允许抢占，则获取同一优先级的下一个任务执行，更新_tx_thread_execute_ptr，_tx_thread_execute_ptr执行需要的任务)
```

```
27.                                 _tx_timer_time_slice =  _tx_thread_current_ptr ->
    tx_time_slice; // 禁止抢占/没有其他可执行任务的情况，更新_tx_timer_time_slice为当前
    任务的时间片

28.                         }

29.                 }

30.     }

31. }
```

9、抢占检查

任务时间片用尽且有同优先级的就绪任务时会导致任务发生抢占，当定时器超时唤醒进程时也可能导致任务抢占(例如一个高优先级的任务sleep一段时间，正好超时，那么当前正在运行的低优先级的任务应该让出cpu给更高优先级的任务运行)。

在任务时间片用尽有同优先级任务就绪或者定时器超时唤醒高优先级任务时，都会更新_tx_thread_execute_ptr，判断是否抢占的依据是_tx_thread_execute_ptr是否等于_tx_timer_current_ptr，不相等则需要重新调度；ThreadX通过___tx_thread_preempt_check来判断抢占，该函数还判断上下文如何保存。

_tx_thread_execute_ptr等于_tx_thread_current_ptr表示_tx_thread_execute_ptr没有更新，不需要抢占，返回0给上一级函数，上一级函数直接恢复中断上下文即可恢复当前运行的任务；

tx_stack_ptr、tx_stack_end等检查用于检测任务栈越界，如果越界则进入无限while循环；

_tx_thread_current_ptr->tx_stack_ptr不为0表示任务栈已经保存，返回1表示不需要保存上下文；(当最后一个就绪任务挂起时，进入_tx_thread_schedule循环，_tx_thread_current_ptr并没有被设置为0，因此在_tx_thread_schedule等待任务就绪的过程中，_tx_thread_current_ptr始终保存的是上一个挂起任务的任务指针，不在执行的任务的_tx_thread_current_ptr->tx_stack_ptr指针指向保存任务上下文的内存地址，正在指向的任务的_tx_thread_current_ptr->tx_stack_ptr指针在_tx_thread_context_restore恢复执行时被设置为0，因此_tx_thread_current_ptr->tx_stack_ptr不为0，实际代表没有任务在运行，_tx_thread_current_ptr->tx_stack_ptr为0表示有任务在运行；_tx_thread_schedule循环过程，没有任务执行情况下当然不需要保存上下文)

总结起来就是返回0表示恢复被中断的任务继续执行(仅恢复中断上下文)；返回1表示发生中断时没有任务在执行，恢复丢弃IRQ栈里面的内容，调用_tx_thread_schedule等待就绪任务即可；返回2表示需要切换到新的任务执行，需要保存当前执行的任务的上下文。

```
int __tx_thread_preempt_check (void)
{
    if (_tx_thread_execute_ptr == _tx_thread_current_ptr) { // 恢复中断上下文继续执行即可
        return 0;
    } else {
        if (_tx_thread_execute_ptr->tx_stack_ptr >
            _tx_thread_execute_ptr->tx_stack_end ||
            _tx_thread_execute_ptr->tx_stack_ptr <
            _tx_thread_execute_ptr->tx_stack_start){
            while(1) { // 任务栈错误，进入无限循环
            }
        }
        if (_tx_thread_current_ptr->tx_stack_ptr != 0) {
            return 1;  // 没有任务在运行
        } else {
            return 2; // 需要切换任务
        }
    }
}
```

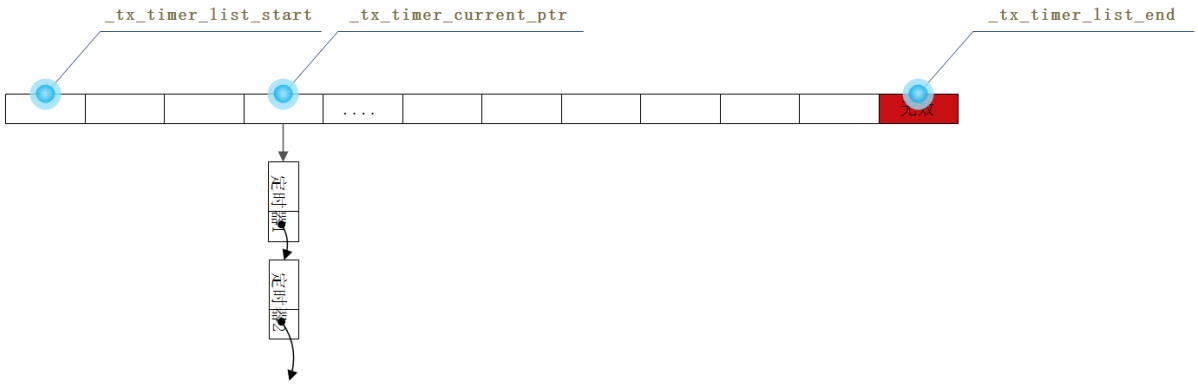
10、定时器

内核定时器timer原理有点类似linux内核定时器，每个硬件时钟中断移动一个节点，定时器可以理解为一个TX_TIMER_ENTRIES的计数器，一个数值在0-TX_TIMER_ENTRIES来回重复计数(跟指针时钟一个原理，秒钟到了60后又从0开始)，举个例子：

假如当前定时器计数到10了，现在有个任务要sleep 10秒，那么理论上我们只要在计数器计数到20即可唤醒该进程，ThreadX用_tx_timer_current_ptr指针作为计数器，每次时钟中断_tx_timer_current_ptr加1，到_tx_timer_list_end后又从_tx_timer_list_start开始计数，可以理解为一个循环链表。假如_tx_timer_current_ptr代表10，一个硬件定时中断1秒，那么sleep 10秒的进程需要在_tx_timer_current_ptr加10次之后触发唤醒操作，那么将sleep 10秒的进程的定时器及任务相关信息存储在(_tx_timer_current_ptr + 10)这个指针指向的定时器节点即可，通过判断(_tx_timer_current_ptr + 10)指向的定时器节点的内容是否为空即可知道是否在该时刻是否有超时需要处理。

跟指针时钟一个原理，计数超过TX_TIMER_ENTRIES就又要重新开始，那么怎么计时超过TX_TIMER_ENTRIES呢？这个跟秒钟进位，分钟加1有所不同，超过TX_TIMER_ENTRIES的定时器，ThreadX是先延时TX_TIMER_ENTRIES，剩余计数周期减去TX_TIMER_ENTRIES，在TX_TIMER_ENTRIES周期到后，触发定时器超时操作，简而言之就是60不够延迟180，那么我们先延迟60、再延迟60、再再延迟60即可。

简要定时器表示如下：



定时器计数在_tx_timer_interrupt在处理，代码如下。

```
1. VOID _tx_timer_interrupt(VOID)
2. {
3.     _tx_timer_system_clock++;
4.     if (_tx_timer_time_slice) {
5.                 _tx_timer_time_slice--;
6.                 if (_tx_timer_time_slice == 0) {
7.                         _tx_timer_expired_time_slice =  TX_TRUE;
8.                 }
9.     }
10.    if (*_tx_timer_current_ptr) { // 当前时间是否有定时器超时
11.        _tx_timer_expired =  TX_TRUE; // 有定时器挂在_tx_timer_current_ptr下面，有超时定时器需要处理(不移动_tx_timer_current_ptr，_tx_timer_current_ptr存储有超期定时器，在定时器超时处理任务需要用到!!!定时器超时任务获取到超时定时器链表后会移动_tx_timer_current_ptr到下一节点)
12.    } else {
13.        _tx_timer_current_ptr++; // 没有定时器需要处理_tx_timer_current_ptr计数加1
14.        if (_tx_timer_current_ptr == _tx_timer_list_end) { // 是否移动到_tx_timer_list_end(计数已达到最大值，犹如秒钟已经到了60)
15.            _tx_timer_current_ptr =  _tx_timer_list_start; // 从_tx_timer_list_start重新开始计数
16.                }
17.    }
18.    if ((_tx_timer_expired_time_slice) || (_tx_timer_expired)) {
19.                if (_tx_timer_expired) { // 有定时器超时
20.                    _tx_timer_expired   = TX_FALSE; // 重置_tx_timer_expired
21.                        _tx_thread_preempt_disable++; // 唤醒定时器任务前禁止抢占(_tx_thread_resume中有_tx_thread_preempt_disable--操作)
22.                        _tx_thread_resume(&_tx_timer_thread); // 唤醒_tx_timer_thread任务，用于处理超时定时器(定时器任务在_tx_timer_initialize创建，定时器任务入口为_tx_timer_thread_entry)
23.                }
24.                if (_tx_timer_expired_time_slice) {
25.                _tx_timer_expired_time_slice    = TX_FALSE;
26.                        if (_tx_thread_time_slice() == TX_FALSE) {
27.                                _tx_timer_time_slice =  _tx_thread_current_ptr -> tx_time_slice;
```

```
28.                         }
29.                 }
30.     }
31. }
```

11、定时器线程

_tx_timer_list有超时定时器时唤醒该线程，该线程取出超时定时器，调用对应定时器超时函数，对仍有剩余时间的定时器，减去一个TX_TIMER_ENTRIES后，重新将未实际超时的定时器添加到_tx_timer_list中。处理完所有超时定时器后，将自己挂起。

```
1. VOID    _tx_timer_thread_entry(ULONG timer_thread_input)
2. {
3. TX_INTERRUPT_SAVE_AREA
4. TX_INTERNAL_TIMER          *expired_timers;           /* Head of expired timers
   list  */
5. TX_INTERNAL_TIMER          *reactivate_timer;         /* Dummy list head pointer
   */
6. TX_INTERNAL_TIMER          **timer_list;              /* Timer list pointer
   */
7. REG_2  TX_INTERNAL_TIMER    *current_timer;           /* Current timer pointer
   */
8. REG_3 UINT                 expiration_time;           /* Value used for pointer
   offset*/
9. VOID                       (*timeout_function)(ULONG); /* Local timeout function
   ptr   */
10. ULONG                     timeout_param;             /* Local timeout parameter
    */
11.     /* Make sure the timer input is correct.  This also gets rid of the
12.        silly compiler warnings.  */
13.     if (timer_thread_input != TX_TIMER_ID)
14.         return;
15.     /* Set the reactivate_timer to NULL.  */
16.     reactivate_timer =  TX_NULL;
17.     /* Now go into an infinite loop to process timer expirations.  */
18.     do
19.     {
20.         /* First, move the current list pointer and clear the timer
21.            expired value.  This allows the interrupt handling portion
22.            to continue looking for timer expirations.  */
23.         TX_DISABLE // 关中断(中断里面以及定时器任务里面都处理_tx_timer_list)
24.         /* Save the current timer expiration list pointer.  */
25.         expired_timers =  *_tx_timer_current_ptr; // 超时定时器节点
26.         /* Modify the head pointer in the first timer in the list, if there
27.            is one!  */
28.         if (expired_timers)
```

```
29.             expired_timers -> tx_list_head =  &expired_timers; // expired_timers ->
    tx_list_head指向自己

30.         /* Set the current list pointer to NULL.  */

31.         *_tx_timer_current_ptr =  TX_NULL; // expired_timers从_tx_timer_list移除

32.         /* Move the current pointer up one timer entry wrap if we get to

33.            the end of the list.  */

34.         _tx_timer_current_ptr++; // _tx_timer_current_ptr定时器加1(移动到下一个节点)

35.         if (_tx_timer_current_ptr == _tx_timer_list_end)

36.             _tx_timer_current_ptr =  _tx_timer_list_start; // _tx_timer_current_ptr
    已经移动到_tx_timer_list_end，从_tx_timer_list_start重新开始

37.         /* Clear the expired flag.  */

38.         _tx_timer_expired =  TX_FALSE; // _tx_timer_list已经没有超时的定时器

39.         /* Now, restore previous interrupt posture.  */

40.         TX_RESTORE // 允许中断(接下的操作与_tx_timer_list不冲突)

41.         /* Next, process the expiration of the associated timers at this

42.            time slot.  */

43.         TX_DISABLE

44.         while (expired_timers) // expired_timers不为空

45.         {

46.             /* Something is on the list.  Remove it and process the expiration.  */

47.             current_timer =  expired_timers; // current_timer正在处理的timer

48.             /* Determine if this is the only timer.  */

49.             if (current_timer == expired_timers -> tx_active_next)

50.             {

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

52.                 /* Set the head pointer to NULL.  */

53.                 expired_timers =  TX_NULL; // 当前正处理最后一个超时timer，将
    expired_timers设置为NULL

54.             }

55.             else

56.             {

57.                 // 将当前处理的timer从expired_timers移除(多个同一时间的timer窜在一个
    链表上)

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

```
59.                /* Remove this timer from the expired list.  */
60.                (current_timer -> tx_active_next) -> tx_active_previous =
61.                                                    current_timer ->
    tx_active_previous;
62.                (current_timer -> tx_active_previous) -> tx_active_next =
63.                                                    current_timer -> tx_active_next;
64.                /* Modify the next timer's list head to point at the current list
    head.  */
65.                (current_timer -> tx_active_next) -> tx_list_head =
    &expired_timers;
66.                /* Set the list head pointer.  */
67.                expired_timers =  current_timer -> tx_active_next;
68.            }
69.            /* In any case, the timer is now off of the expired list.  */
70.            /* Determine if the timer has expired or if it is just a really
71.               big timer that needs to be placed in the list again.  */
72.            if (current_timer -> tx_remaining_ticks > TX_TIMER_ENTRIES) // 定时器剩
    余时大于TX_TIMER_ENTRIES
73.            {
74.                /* Timer is bigger than the timer entries and must be
75.                   re-scheduled.  */
76.                /* Decrement the remaining ticks of the timer.  */
77.                current_timer -> tx_remaining_ticks =
78.                        current_timer -> tx_remaining_ticks - TX_TIMER_ENTRIES; //
    定时器剩余时间减一个TX_TIMER_ENTRIES周期
79.                /* Set the timeout function to NULL in order to bypass the
80.                   expiration.  */
81.                timeout_function =  TX_NULL;
82.                /* Make the timer appear that it is still active while interrupts
83.                   are enabled.  This will permit proper processing of a timer
84.                   deactivate from an ISR.  */
85.                current_timer -> tx_list_head =    &reactivate_timer; // 定时器加入
    到reactivate_timer，需要再次激活
86.                current_timer -> tx_active_next =  current_timer;
87.            }
```

```
88.             else // 定时器超时
89.             {
90.                 /* Timer did expire.  Copy the calling function and ID
91.                    into local variables before interrupts are re-enabled.  */
92.                 timeout_function =  current_timer -> tx_timeout_function; // 定时器超时处理函数
93.                 timeout_param =     current_timer -> tx_timeout_param; // 定时器超时函数参数
94.                 /* Copy the re-initialize ticks into the remaining ticks.  */
95.                 current_timer -> tx_remaining_ticks =  current_timer -> tx_re_initialize_ticks; // tx_re_initialize_ticks由_tx_timer_change函数设置(一个定时器触发多次，一个定时器在计数时再次触发另外一次计数)
96.                 /* Determine if the timer should be re-activated.  */
97.                 if (current_timer -> tx_remaining_ticks) // 需要再次激活定时器
98.                 {
99.                     /* Make the timer appear that it is still active while processing
100.                       the expiration routine and with interrupts enabled.  This will
101.                       permit proper processing of a timer deactivate from both the
102.                       expiration routine and an ISR.  */
103.                    current_timer -> tx_list_head =    &reactivate_timer; // 当前定时器加入reactivate_timer链表
104.                    current_timer -> tx_active_next =  current_timer;
105.                }
106.                else
107.                {
108.                    /* Set the list pointer of this timer to NULL.  This is used to indicate
109.                       the timer is no longer active.  */
110.                    current_timer -> tx_list_head =  TX_NULL; // 定时器失效
111.                }
112.            }
113.            /* Restore interrupts for timer expiration call.  */
114.            TX_RESTORE
115.            /* Call the timer-expiration function, if non-NULL.  */
```

```
116.             if (timeout_function)

117.                 (timeout_function) (timeout_param); // 定时器超时处理(真正超时的定时器设置了timeout_function函数指针及timeout_param参数，对于sleep，此处应该是调用任务唤醒函数(_tx_thread_timeout、_tx_thread_resume)，timeout_function指向唤醒任务的函数，timeout_param指向被唤醒的进程的信息)

118.             /* Lockout interrupts again.  */

119.             TX_DISABLE

120.             /* Determine if the timer needs to be re-activated.  */

121.             if (current_timer -> tx_list_head == &reactivate_timer) // 重新激活定时器(还没真正超期的定时器(一次TX_TIMER_ENTRIES不够计数的定时器)，重新激活多次触发的定时器)

122.             {

123.                 /* Re-activate the timer.  */

124.                 /* Calculate the amount of time remaining for the timer.  */

125.                 if (current_timer -> tx_remaining_ticks > TX_TIMER_ENTRIES)

126.                 {

127.                     /* Set expiration time to the maximum number of entries.  */

128.                     expiration_time =  TX_TIMER_ENTRIES - 1; // 一次TX_TIMER_ENTRIES计数仍不够，先延迟TX_TIMER_ENTRIES这么多周期

129.                 }

130.                 else

131.                 {

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

133.                     /* Set the expiration time.  */

134.                     expiration_time =  (UINT) current_timer -> tx_remaining_ticks - 1; // 一次TX_TIMER_ENTRIES够计数

135.                 }

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

137.                    the timer lists.  */

138.                 /* Calculate the proper place for the timer.  */

139.                 timer_list =  _tx_timer_current_ptr + expiration_time; // 获取定时器需要加入的节点

140.                 if (timer_list >= _tx_timer_list_end) // 插入点超过_tx_timer_list_end

141.                 {

142.                     /* Wrap from the beginning of the list.  */
```

```
143.                     timer_list =  _tx_timer_list_start +
144.                                         (timer_list - _tx_timer_list_end); // 重新计
    算插入点
145.                 }
146.                 /* Now put the timer on this list.  */
147.                 if (*timer_list) // timer_list已有其他定时器，执行插入操作
148.                 {
149.                     /* This list is not NULL, add current timer to the end. */
150.                     current_timer -> tx_active_next =
    *timer_list;
151.                     current_timer -> tx_active_previous =
    (*timer_list) -> tx_active_previous;
152.                     (current_timer -> tx_active_previous) -> tx_active_next =
    current_timer;
153.                     (*timer_list) -> tx_active_previous =
    current_timer;
154.                     current_timer -> tx_list_head =
    timer_list;
155.                 }
156.                 else // timer_list为空，timer_list指向current_timer
157.                 {
158.                     /* This list is NULL, just put the new timer on it.  */
159.                     /* Setup the links in this timer.  */
160.                     current_timer -> tx_active_next =      current_timer;
161.                     current_timer -> tx_active_previous =  current_timer;
162.                     current_timer -> tx_list_head =        timer_list;
163.                     /* Setup the list head pointer.  */
164.                     *timer_list =  current_timer;
165.                 }
166.             }
167.             /* Restore interrupts.  */
168.             TX_RESTORE
169.             /* Lockout interrupts again.  */
170.             TX_DISABLE
171.         }
```

```
172.         /* Finally, suspend this thread and wait for the next expiration.  */
173.         /* Determine if another expiration took place while we were in this
174.            thread.  If so, process another expiration.  */
175.         if (!_tx_timer_expired) // _tx_timer_interrupt没有触发新的定时器超时
176.         {
177.             /* Otherwise, no timer expiration, so suspend the thread.  */
178.             /* Set the status to suspending, in order to indicate the
179.                suspension is in progress.  */
180.             _tx_thread_current_ptr -> tx_state =  TX_SUSPENDED;
181.             /* Set the suspending flag. */
182.             _tx_thread_current_ptr -> tx_suspending =  TX_TRUE;
183.             /* Increment the preempt disable count prior to suspending.  */
184.             _tx_thread_preempt_disable++;
185.             /* Restore interrupts.  */
186.             TX_RESTORE
187.             /* Call actual thread suspension routine.  */
188.             _tx_thread_suspend(_tx_thread_current_ptr); // 挂起自己，等定时器超时后
    被唤醒
189.         }
190.         else
191.         {
192.             /* Restore interrupts.  */
193.             TX_RESTORE // 允许中断，接着处理超时定时器
194.         }
195.     } while (TX_FOREVER);
196. }
```