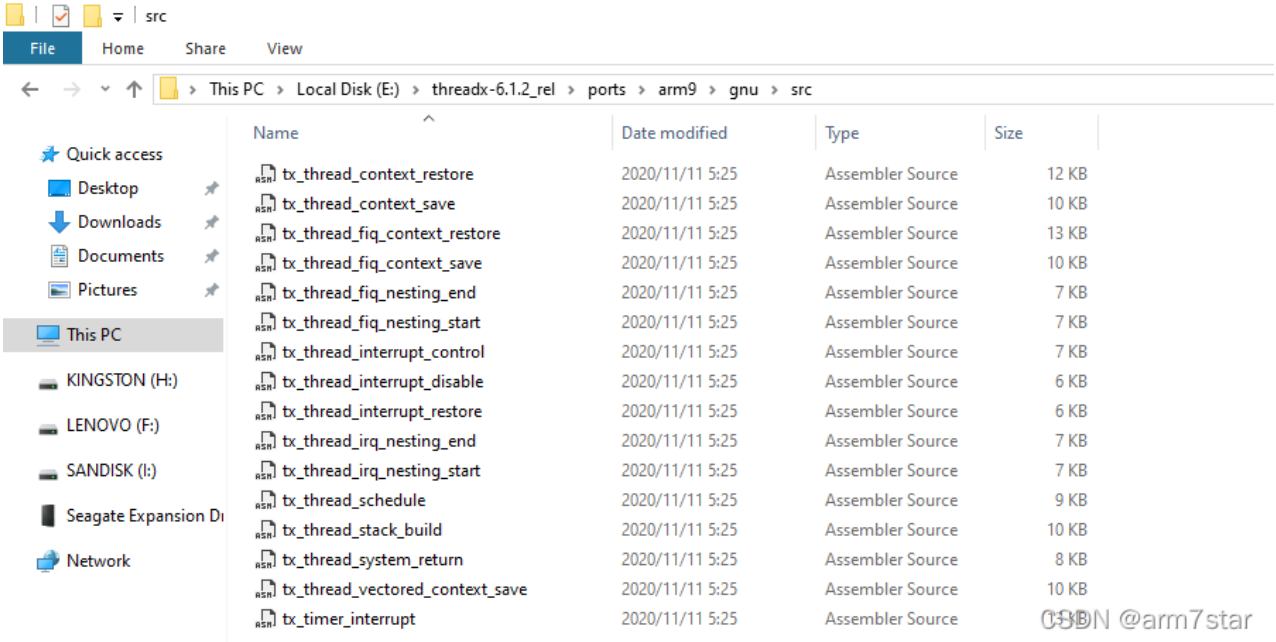
# (163条消息) ThreadX内核源码分析 - ports线程上下文相关代码分析(arm)_arm7star的博客-CSDN博客_threadx arm9



## 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_restore函数的实现。

## 2、ThreadX线程上下文

ThreadX内核在ARM上使用满递减栈("满"是指栈顶指针指向的内存地址是有数据的，下一个数据要入栈，则栈顶指针需要移动到下一个内存单元才行；"递减"是指栈是从高地址往低地址增长的，栈底在内存的高地址，栈顶在内存的低地址)。

## 2.1、中断线程上下文

ThreadX的线程中断上下文在内存栈里面的结构如下所示:

正在执行的线程进入睡眠状态/时间片用尽/被抢占等情况让出cpu的时候，线程的上下文主动或者被动保存到线程的栈里面，这里的栈也就是c语言意义上的栈(c函数的局部变量/参数的栈)，线程第一次执行前，内核会将分配给线程的栈的地址设置到sp栈指针(线程执行前，sp即指向线程栈的起始地址；以一个c函数为例，函数执行时，sp指向函数栈的起始地址，假如函数保存函数调用上下文需要2个byte，函数局部变量a占用1个byte，那么可以用sp - 1、sp - 2的地址保存函数调用上下文，sp - 3的地址保存变量a，然后将sp - 3设置为新

的sp(函数上下文/变量已经入栈)，函数返回时恢复旧的sp的值，sp到sp - 3这3个byte的栈空间被释放，局部变量在栈里面动态分配也就是这个原理，函数的局部变量在函数返回后就不存在了也是这个道理)。

中断上下文，中断上下文的栈顶为1，标志该栈为中断栈，cpu发生中断时，内核并没有办法判断线程正在使用那些寄存器，因此中断线程的上下文需要保存所有的寄存器。


## 2.2、非中断线程上下文

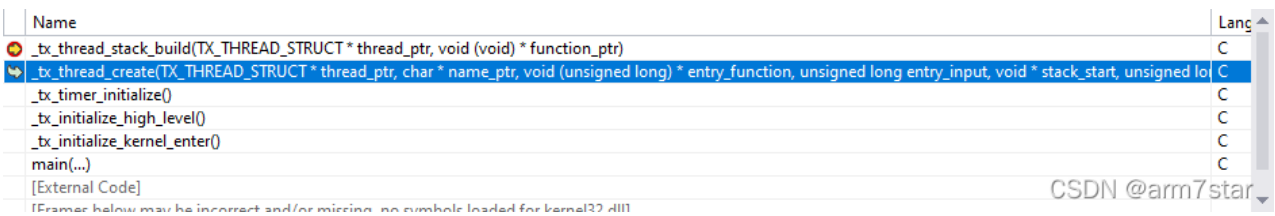
非中断线程上下文与此不同，非中断是指线程自己调用内核函数主动让出cpu，例如调用sleep/调用互斥锁等进入睡眠或者阻塞状态，下次线程执行时，需要返回的不是保存线程上下文的函数，例如调用sleep函数，下次线程执行是不需要回到sleep函数再返回调用sleep的函数，而是直接返回到调用sleep函数的下一条指令，线程在调用sleep函数前，一般r0-r3没有任何意义了(sleep后之后的代码不会用到r0-r3)或者编译器会把r0-r3保存到栈里面(如果寄存器r0-r3的值在sleep之后还有用到，从实际反汇编代码看，编译器实际在sleep之后的代码并没有用到sleep之前的r0-r3的值)，因此，线程主动让出cpu时，r0-r3寄存器如果有用的话，编译器已经保存到了栈里面，如果没有用的话，也没有保存的必要，sleep返回之后r0-r3的值并没有实际意义。

非中断线程上下文的栈顶的值为0，另外没有r0-r3。

## 3、线程栈创建(_tx_thread_stack_build)

线程运行的过程是“创建线程->线程执行”，线程执行简单理解就是把线程入口函数的上下文恢复到cpu上(cpu寄存器，然后让cpu跳转到线程入口执行；前面讲过了c函数执行的调用上下文及局部变量保存在栈里面，因此线程入口函数执行前sp寄存器必须指向线程的栈地址，对于linux带有mmu的内核，还需要设置mmu，ThreadX比较简单，所有线程在一个地址空间指向，不需要切换mmu)

线程栈的创建函数为_tx_thread_stack_build，下图是ThreadX启动过程创建线程的一个函数调用栈，_tx_thread_create创建线程，调用_tx_thread_stack_build创建线程的栈：



线程栈创建主要是线程的入口(pc)、线程的栈sp、线程运行时模式cpsr等寄存器设置，线程创建时的栈是按中断上下文栈类型保存寄存器的，虽然很多寄存器没有实际用到，创建栈的代码如下：

```
104     .global  _tx_thread_stack_build
105     .type    _tx_thread_stack_build,function
106 _tx_thread_stack_build:
107 @
108 @
109 @    /* Build a fake interrupt frame.  The form of the fake interrupt stack
110 @       on the ARM9 should look like the following after it is built:
111 @
112 @       Stack Top:      1           Interrupt stack frame type
113 @                       CPSR        Initial value for CPSR
114 @                       a1 (r0)     Initial value for a1
115 @                       a2 (r1)     Initial value for a2
116 @                       a3 (r2)     Initial value for a3
117 @                       a4 (r3)     Initial value for a4
118 @                       v1 (r4)     Initial value for v1
119 @                       v2 (r5)     Initial value for v2
120 @                       v3 (r6)     Initial value for v3
121 @                       v4 (r7)     Initial value for v4
122 @                       v5 (r8)     Initial value for v5
123 @                       sb (r9)     Initial value for sb
124 @                       sl (r10)    Initial value for sl
125 @                       fp (r11)    Initial value for fp
126 @                       ip (r12)    Initial value for ip
127 @                       lr (r14)    Initial value for lr
128 @                       pc (r15)    Initial value for pc
129 @                       0       For stack backtracing
130 @
131 @    Stack Bottom: (higher memory address)  */
132 @
133     LDR     r2, [r0, #16]                   @ Pickup end of stack area // r0为函数第一个参数的值thread_ptr，r2 = thread_ptr->tx_thread_stack_end，取函数栈的高地址(栈底)
```

31. 134     BIC     r2, r2, #7                      @ Ensure 8-byte alignment // r2(栈顶地址)的低3 bit位清0，     确保栈的起始地址按8 byte地址对齐，arm的入栈指令必须4 byte对齐(否则会产生总线错误)，8 byte对齐自然也是4 byte对齐

32. 135     SUB     r2, r2, #76                     @ Allocate space for the stack frame // 这里预留76 byte(19 word)个内存地址用于保存线程的栈，保存栈只需要18 word个内存，栈底保存了一个固定的0，因此多了1个word；另外线程创建时的栈保存了r0-r3寄存器，线程第一次运行并不是调用用户的线程入口函数，而是内核的_tx_thread_shell_entry，由_tx_thread_shell_entry调用用户线程入口函数，_tx_thread_shell_entry还要负责线程退出的一些清理工作，用户线程入口函数没办法做；r2减完之后执行栈顶

33. 136 @

34. 137 @    /* Actually build the stack frame.  */

35. 138 @

36. 139     MOV     r3, #1                          @ Build interrupt stack type // 栈类型设置为1(虽然不是中断上下文，但是栈里面保存了r0-r3寄存器，因此栈里面的内容与中断上下文相同，恢复时需要按照中断上下文恢复一样，恢复r0-r3寄存器)

37. 140     STR     r3, [r2, #0]                    @ Store stack type // 栈类型入栈(直接写入栈类型在栈中的内存地址)

38. 141     MOV     r3, #0                          @ Build initial register value // r3 = 0，没有用到的寄存器入栈时默认为0

39. 142     STR     r3, [r2, #8]                    @ Store initial r0 // 后续代码r0-r9入栈

40. 143     STR     r3, [r2, #12]                   @ Store initial r1

41. 144     STR     r3, [r2, #16]                   @ Store initial r2

42. 145     STR     r3, [r2, #20]                   @ Store initial r3

43. 146     STR     r3, [r2, #24]                   @ Store initial r4

44. 147     STR     r3, [r2, #28]                   @ Store initial r5

45. 148     STR     r3, [r2, #32]                   @ Store initial r6

46. 149     STR     r3, [r2, #36]                   @ Store initial r7

47. 150     STR     r3, [r2, #40]                   @ Store initial r8

48. 151     STR     r3, [r2, #44]                   @ Store initial r9

49. 152     LDR     r3, [r0, #12]                   @ Pickup stack starting address // r3 = thread_ptr->tx_thread_stack_start，获取栈的低内存地址

50. 153     STR     r3, [r2, #48]                   @ Store initial r10 (sl) // r10(栈顶指针寄存器)入栈，栈不能超过该地址，否则就非法访问别的内存了

51. 154     LDR     r3,=_tx_thread_schedule         @ Pickup address of _tx_thread_schedule for GDB backtrace // 取_tx_thread_schedule函数地址

52. 155     STR     r3, [r2, #60]                   @ Store initial r14 (lr) // lr = _tx_thread_schedule，用于gdb栈回溯使用，如果用gdb的backtrace命令查看栈的话，可以看到_tx_thread_schedule在调用栈的最顶端，仅给gdb用，没有实际用处

53. 156     MOV     r3, #0                          @ Build initial register value

```
54. 157     STR     r3, [r2, #52]                   @ Store initial r11 // r11入栈

55. 158     STR     r3, [r2, #56]                   @ Store initial r12

56. 159     STR     r1, [r2, #64]                   @ Store initial pc // r1为
    _tx_thread_stack_build的第二个参数function_ptr(_tx_thread_shell_entry)，线程第一次执
    行时，恢复_tx_thread_shell_entry到pc寄存器

57. 160     STR     r3, [r2, #68]                   @ 0 for back-trace

58. 161     MRS     r1, CPSR                        @ Pickup CPSR // 获取cpsr寄存器(cpsr
    里面保存了cpu模式等信息(ARM/Thumb)，恢复cpsr时不能随便恢复)

59. 162     BIC     r1, r1, #CPSR_MASK              @ Mask mode bits of CPSR // IRQ/FIQ
    禁止中断标志位等清零(线程运行时必须使能中断，否则硬件定时器中断没办法被相应，内核没
    办法计时)

60. 163     ORR     r3, r1, #SVC_MODE               @ Build CPSR, SVC mode, interrupts
    enabled // 设置为SVC模式(ThreadX线程全部运行在SVC特权模式，有开关中断等特权，如果线
    程运行在用户模式，开关中断需要切换到特权模式才行，影响性能)

61. 164     STR     r3, [r2, #4]                    @ Store initial CPSR // cpsr入栈

62. 165 @

63. 166 @    /* Setup stack pointer.  */

64. 167 @    thread_ptr -> tx_thread_stack_ptr =  r2;

65. 168 @

66. 169     STR     r2, [r0, #8]                    @ Save stack pointer in thread's //
    thread_ptr->tx_thread_stack_ptr = r8(栈顶地址)，线程恢复执行时，通过thread_ptr-
    >tx_thread_stack_ptr找到线程上下文的栈顶地址，从而恢复线程上下文寄存器

67. 170                                             @   control block

68. 171 #ifdef __THUMB_INTERWORK

69. 172     BX      lr                              @ Return to caller

70. 173 #else

71. 174     MOV     pc, lr                          @ Return to caller

72. 175 #endif

73. 176 @}

74. 177
```

## 4、线程上下文的恢复(_tx_thread_context_restore)

线程中断后/睡眠后/阻塞后，恢复执行都是由内核调用_tx_thread_context_restore恢复线程上下文；线程被创建/被切换出去时，thread_ptr->tx_thread_stack_ptr指向了线程的栈顶，而此时的栈顶保存的是线程的上下文；

## 4.1、保存被切换出去的线程的上下文

恢复一个线程时，必须先保存当前正在执行的线程的上下文，ThreadX用_tx_thread_execute_ptr指向当前正在执行的线程，_tx_thread_context_restore首先检查当前没有没线程正在执行(_tx_thread_execute_ptr是否不为空)，如果有线程正在执行，先保存正在执行线程的上下文，然后调用_tx_thread_schedule执行下一个线程，否则直接调用_tx_thread_schedule执行下一个线程。

_tx_thread_context_restore保存被切换出去的线程的中断上下文的代码如下：

```
100     .global _tx_thread_context_restore
101     .type   _tx_thread_context_restore,function
102 _tx_thread_context_restore:
103 @
104 @    /* Lockout interrupts.  */
105 @
106     MOV     r0, #IRQ_MODE                   @ Build disable interrupts CPSR
107     MSR     CPSR, r0                        @ Lockout interrupts // 禁止IRQ中断
108
109 #ifdef TX_ENABLE_EXECUTION_CHANGE_NOTIFY
110 @
111 @    /* Call the ISR exit function to indicate an ISR is complete.  */
112 @
113     BL      _tx_execution_isr_exit          @ Call the ISR exit function
114 #endif
115 @
116 @    /* Determine if interrupts are nested.  */
117 @    if (--_tx_thread_system_state)
118 @    {
119 @
120     LDR     r3, =_tx_thread_system_state    @ Pickup address of system state
    variable // 检查中断嵌套(如果是内核初始化过程，中断发生后应该返回到内核初始化，只有等系统初始化完成后才能调度线程，内核初始化相当于第一次中断，内核初始化时_tx_thread_system_state设置为非0，内核初始化完成后_tx_thread_system_state设置为0；如果是中断嵌套，要一层一层返回所有中断，也不能调度线程)
121     LDR     r2, [r3]                        @ Pickup system state
122     SUB     r2, r2, #1                      @ Decrement the counter // 中断退出，中断嵌套计数器_tx_thread_system_state减1(中断进入时加1)
123     STR     r2, [r3]                        @ Store the counter
124     CMP     r2, #0                          @ Was this the first interrupt? // 如果中断嵌套计数器_tx_thread_system_state为0，表明当前中断为最外层中断，外面没有中断了，没有嵌套中断
125     BEQ     __tx_thread_not_nested_restore  @ If so, not a nested restore // 没有嵌套中断要处理，跳转到__tx_thread_not_nested_restore恢复中断上下文或者调度线程
126 @
```

```
127 @    /* Interrupts are nested.  */
128 @
129 @    /* Just recover the saved registers and return to the point of
130 @       interrupt.  */
131 @
132     LDMIA   sp!, {r0, r10, r12, lr}         @ Recover SPSR, POI, and scratch
    regs // 恢复中断上下文
133     MSR     SPSR, r0                        @ Put SPSR back
134     LDMIA   sp!, {r0-r3}                    @ Recover r0-r3
135     MOVS    pc, lr                          @ Return to point of interrupt // 返
    回被中断的中断继续处理中断
136 @
137 @    }
138 __tx_thread_not_nested_restore: // 最外层中断退出，中断恢复
139 @
140 @    /* Determine if a thread was interrupted and no preemption is required.  */
141 @    else if (((_tx_thread_current_ptr) && (_tx_thread_current_ptr ==
    _tx_thread_execute_ptr)
142 @              || (_tx_thread_preempt_disable))
143 @    {
144 @
145     LDR     r1, =_tx_thread_current_ptr     @ Pickup address of current thread
    ptr
146     LDR     r0, [r1]                        @ Pickup actual current thread
    pointer
147     CMP     r0, #0                          @ Is it NULL? // 检查
    _tx_thread_current_ptr是否不为空，是否有线程要被执行(中断服务程序可能唤醒更高优先级
    线程或者当前执行的线程的时间片用完了)
148     BEQ     __tx_thread_idle_system_restore @ Yes, idle system was interrupted
    // 没有线程要被执行，跳转到__tx_thread_idle_system_restore
149 @
150     LDR     r3, =_tx_thread_preempt_disable @ Pickup preempt disable address
151     LDR     r2, [r3]                        @ Pickup actual preempt disable flag
152     CMP     r2, #0                          @ Is it set? // 检查
    _tx_thread_preempt_disable禁止抢占是否被设置(有些内核函数正在做重要的事情，即使当前
    执行的线程的时间片用完了或者有高优先级就绪了，也不能立即被抢占，当前线程做完重要的事
    情后，由当前线程来检查是否要重新调度别的线程)
```

```
153     BNE     __tx_thread_no_preempt_restore  @ Yes, don't preempt this thread // 有禁止抢占，跳转到__tx_thread_no_preempt_restore，不切换线程，继续当前正在执行的线程

154     LDR     r3, =_tx_thread_execute_ptr     @ Pickup address of execute thread ptr // 没有禁止抢占，获取正在执行的线程_tx_thread_execute_ptr

155     LDR     r2, [r3]                        @ Pickup actual execute thread pointer

156     CMP     r0, r2                          @ Is the same thread highest priority? // 比较是否为同一个线程(下一个要执行的线程是否为当前正在执行的线程)

157     BNE     __tx_thread_preempt_restore     @ No, preemption needs to happen // 不是同一个线程，_tx_thread_execute_ptr被_tx_thread_current_ptr抢占或者时间片轮转到_tx_thread_current_ptr执行，需要换出当前正在执行的线程_tx_thread_execute_ptr

158 @

159 @

160 __tx_thread_no_preempt_restore: // 正在执行的线程没有被抢占，继续被中断的线程

161 @

162 @    /* Restore interrupted thread or ISR.  */

163 @

164 @    /* Pickup the saved stack pointer.  */

165 @    tmp_ptr =  _tx_thread_current_ptr -> tx_thread_stack_ptr;

166 @

167 @   /* Recover the saved context and return to the point of interrupt.  */

168 @

169     LDMIA   sp!, {r0, r10, r12, lr}         @ Recover SPSR, POI, and scratch regs // 恢复r0(cpsr), r10, r12, lr(pc)

170     MSR     SPSR, r0                        @ Put SPSR back

171     LDMIA   sp!, {r0-r3}                    @ Recover r0-r3 // 恢复r0-r3(c函数如果用到r4-r12的寄存器的话，c函数进入时会保存这些寄存器，退出时会恢复这些寄存器，所以c函数不保护的r0-r3寄存器，中断汇编代码使用r0-r3及调用c函数前需要保存r0-r3寄存器，另外，很明显汇编代码只用到了r0-r3寄存器，因此中断退出时，只需要恢复r0-r3寄存器即可)

172     MOVS    pc, lr                          @ Return to point of interrupt // MOVS带S后缀时，会将spsr恢复到cpsr寄存器，lr为中断返回地址，该指令即返回到中断的线程继续执行

173 @

174 @    }

175 @    else

176 @    {

177 __tx_thread_preempt_restore: // 当前执行的线程被抢占或者轮转出去

178 @
```

```
179     LDMIA   sp!, {r3, r10, r12, lr}         @ Recover temporarily saved registers // 恢复中断的寄存器r3(cpsr), r10, r12, lr(pc)

180     MOV     r1, lr                          @ Save lr (point of interrupt) // 保存中断返回地址到r1(当前线程恢复地址pc，模式切换后lr(irq)寄存器不能访问)

181     MOV     r2, #SVC_MODE                   @ Build SVC mode CPSR

182     MSR     CPSR, r2                        @ Enter SVC mode // 切换到SVC模式

183     STR     r1, [sp, #-4]!                  @ Save point of interrupt // 切换到SVC模式后，进入到线程模式下面，此时的sp是线程的栈指针，pc入中断上下文的栈(跟创建线程是的栈不同，栈底没有预留0)，!执行指令后更新sp寄存器

184     STMDB   sp!, {r4-r12, lr}               @ Save upper half of registers // STMDB的DB为先减的意思，执行指令前sp指向了pc，r4-r12,、lr入栈，r4在低内存地址!!!

185     MOV     r4, r3                          @ Save SPSR in r4 // 线程的cpsr保存到r4寄存器

186     MOV     r2, #IRQ_MODE                   @ Build IRQ mode CPSR

187     MSR     CPSR, r2                        @ Enter IRQ mode // 切换回IRQ模式(需要访问IRQ的sp寄存器)

188     LDMIA   sp!, {r0-r3}                    @ Recover r0-r3 // 恢复r0-r3寄存器

189     MOV     r5, #SVC_MODE                   @ Build SVC mode CPSR

190     MSR     CPSR, r5                        @ Enter SVC mode // 切换到SVC模式(线程上下文)

191     STMDB   sp!, {r0-r3}                    @ Save r0-r3 on thread's stack // r0-r3入栈

192     MOV     r3, #1                          @ Build interrupt stack type // 栈的类型为中断栈(保存有r0-r3寄存器)

193     STMDB   sp!, {r3, r4}                   @ Save interrupt stack type and SPSR // 栈类型、cpsr入栈

194     LDR     r1, =_tx_thread_current_ptr     @ Pickup address of current thread ptr // 获取正在执行的线程(被中断的线程)_tx_thread_current_ptr

195     LDR     r0, [r1]                        @ Pickup current thread pointer

196     STR     sp, [r0, #8]                    @ Save stack pointer in thread control

197                                             @   block // 线程栈顶地址保存到thread_ptr->tx_thread_stack_ptr，至此线程的中断上下文已经保存到线程的栈里面了

198 @

199 @    /* Save the remaining time-slice and disable it.  */

200 @    if (_tx_timer_time_slice)

201 @    {

202 @

203     LDR     r3, =_tx_timer_time_slice       @ Pickup time-slice variable address
```

```
105. 204     LDR     r2, [r3]                        @ Pickup time-slice

106. 205     CMP     r2, #0                          @ Is it active? // 检查线程的时间片
     _tx_timer_time_slice是否为0，不为0的话，需要保存线程的时间片，时间片为0表示线程不使
     用时间片(可以无限执行)

107. 206     BEQ     __tx_thread_dont_save_ts        @ No, don't save it // 没有启用时间
     片，跳转到__tx_thread_dont_save_ts，不需要保存_tx_timer_time_slice

108. 207 @

109. 208 @        _tx_thread_current_ptr -> tx_thread_time_slice =  _tx_timer_time_slice;

110. 209 @        _tx_timer_time_slice =  0;

111. 210 @

112. 211     STR     r2, [r0, #24]                   @ Save thread's time-slice //
     _tx_thread_current_ptr -> tx_thread_time_slice =  _tx_timer_time_slice;

113. 212     MOV     r2, #0                          @ Clear value

114. 213     STR     r2, [r3]                        @ Disable global time-slice flag //
     _tx_timer_time_slice设置为0，正在执行的线程被切换出去，还没有线程被执行前，
     _tx_timer_time_slice设置为0，不需要对_tx_timer_time_slice进行计数

115. 214 @

116. 215 @    }

117. 216 __tx_thread_dont_save_ts:

118. 217 @

119. 218 @

120. 219 @    /* Clear the current task pointer.  */

121. 220 @    _tx_thread_current_ptr =  TX_NULL;

122. 221 @

123. 222     MOV     r0, #0                          @ NULL value

124. 223     STR     r0, [r1]                        @ Clear current thread pointer //
     _tx_thread_current_ptr =  TX_NULL; 线程信息已经保存了，当前cpu上没有线程在执行

125. 224 @

126. 225 @    /* Return to the scheduler.  */

127. 226 @    _tx_thread_schedule();

128. 227 @

129. 228     B       _tx_thread_schedule             @ Return to scheduler // 调用
     _tx_thread_schedule调度新的线程

130. 229 @    }

131. 230 @

132. 231 __tx_thread_idle_system_restore:
```

```
133. 232 @
134. 233 @    /* Just return back to the scheduler!  */
135. 234 @
136. 235     MOV     r0, #SVC_MODE                   @ Build SVC mode CPSR
137. 236     MSR     CPSR, r0                        @ Enter SVC mode // 切换到SVC模式
     (IRQ模式下跳转到__tx_thread_idle_system_restore，调用调度函数需要切换到SVC内核模式)
138. 237     B       _tx_thread_schedule             @ Return to scheduler // 调用
     _tx_thread_schedule
139. 238 @}
140. 239
```

## 5、线程调度(_tx_thread_schedule)

_tx_thread_schedule主要检查是否有线程需要执行，有的话就恢复线程上下文，没有的话就循环检测；ARM9没有看到省电唤醒指令，所以是不断循环等待，看到有cortex-m之类的处理器有类似睡眠指令，中断的时候唤醒cpu。

_tx_thread_schedule一定程度上可以看作是一个idle线程，没有其他线程可以执行的时候就执行idle线程。

有就绪线程的话，内核会设置_tx_thread_execute_ptr指向最高优先级的就绪线程，例如定时器计时检查到sleep的timer过期了，唤醒sleep线程，唤醒操作会设置_tx_thread_execute_ptr，_tx_thread_schedule只要检查_tx_thread_execute_ptr即可；

_tx_thread_schedule主要设置_tx_thread_current_ptr = _tx_thread_execute_ptr，恢复线程之前的时间片，恢复线程的寄存器，_tx_thread_schedule的代码实现如下：

```
1. 109     .global _tx_thread_schedule
2. 110         .type  _tx_thread_schedule,function
3. 111 _tx_thread_schedule:
4. 112 @
5. 113 @    /* Enable interrupts.  */
6. 114 @
7. 115     MRS     r2, CPSR                        @ Pickup CPSR
8. 116     BIC     r0, r2, #ENABLE_INTS            @ Clear the disable bit(s) // 清除禁
   止中断标志位，允许中断
9. 117     MSR     CPSR_cxsf, r0                   @ Enable interrupts // 设置cpsr中断
   标志位为0(允许中断)
10. 118 @
11. 119 @    /* Wait for a thread to execute.  */
12. 120 @    do
13. 121 @    {
14. 122     LDR     r1, =_tx_thread_execute_ptr     @ Address of thread execute ptr
15. 123 @
16. 124 __tx_thread_schedule_loop: // 等待_tx_thread_execute_ptr不为空(需要执行的线程)
17. 125 @
18. 126     LDR     r0, [r1]                        @ Pickup next thread to execute
19. 127     CMP     r0, #0                          @ Is it NULL?
20. 128     BEQ     __tx_thread_schedule_loop       @ If so, keep looking for a thread
21. 129 @
22. 130 @    }
23. 131 @    while(_tx_thread_execute_ptr == TX_NULL);
24. 132 @
25. 133 @    /* Yes! We have a thread to execute.  Lockout interrupts and
26. 134 @       transfer control to it.  */
27. 135 @
28. 136     MSR     CPSR_cxsf, r2                   @ Disable interrupts // 禁止中断，接
   着需要恢复线程
29. 137 @
30. 138 @    /* Setup the current thread pointer.  */
```

```
139 @    _tx_thread_current_ptr =  _tx_thread_execute_ptr;
140 @
141     LDR     r1, =_tx_thread_current_ptr     @ Pickup address of current thread
    // 当前正在执行的线程_tx_thread_current_ptr(新线程即将被执行，_tx_thread_current_ptr
    记录cpu上执行的线程)
142     STR     r0, [r1]                        @ Setup current thread pointer //
    _tx_thread_current_ptr =  _tx_thread_execute_ptr
143 @
144 @    /* Increment the run count for this thread.  */
145 @    _tx_thread_current_ptr -> tx_thread_run_count++;
146 @
147     LDR     r2, [r0, #4]                    @ Pickup run counter //
    _tx_thread_current_ptr -> tx_thread_run_count
148     LDR     r3, [r0, #24]                   @ Pickup time-slice for this thread
    // 取出线程的运行时间片_tx_thread_current_ptr->tx_thread_time_slice(上次运行时间，
    ThreadX的线程运行时间是累计的，如果每次都从一个全新的时间片开始运行，而且线程每次都
    没执行完一个时间片就被切换出去，那么该线程之后的同优先级的线程就得不到调度，因此，线
    程的时间片是累计的)
149     ADD     r2, r2, #1                      @ Increment thread run-counter // 线
    程运行次数加1
150     STR     r2, [r0, #4]                    @ Store the new run counter
151 @
152 @    /* Setup time-slice, if present.  */
153 @    _tx_timer_time_slice =  _tx_thread_current_ptr -> tx_thread_time_slice;
154 @
155     LDR     r2, =_tx_timer_time_slice       @ Pickup address of time-slice
156                                             @   variable
157     LDR     sp, [r0, #8]                    @ Switch stack pointers // sp =
    _tx_thread_current_ptr->tx_thread_stack_ptr，sp指向之前保存线程上下文的栈顶
158     STR     r3, [r2]                        @ Setup time-slice //
    _tx_timer_time_slice = _tx_thread_current_ptr -> tx_thread_time_slice，线程运行过程
    中_tx_timer_time_slice代表线程剩余的时间片，定时器对_tx_timer_time_slice计数，线程换
    出时，_tx_timer_time_slice写回到_tx_thread_current_ptr -> tx_thread_time_slice
159 @
160 @    /* Switch to the thread's stack.  */
161 @    sp =  _tx_thread_execute_ptr -> tx_thread_stack_ptr;
162 @
163 #ifdef TX_ENABLE_EXECUTION_CHANGE_NOTIFY
```

```
164 @
165 @    /* Call the thread entry function to indicate the thread is executing.  */
166 @
167     BL      _tx_execution_thread_enter      @ Call the thread execution enter
    function
168 #endif
169 @
170 @    /* Determine if an interrupt frame or a synchronous task suspension frame
171 @   is present.  */
172 @
173     LDMIA   sp!, {r0, r1}                   @ Pickup the stack type and saved
    CPSR // 栈类型、cpsr出栈
174     CMP     r0, #0                          @ Check for synchronous context
    switch // 判断栈类型(中断上下文栈保存了r0-r3，需要检查是否恢复r0-r3)
175     MSRNE   SPSR_cxsf, r1                   @   Setup SPSR for return // 中断栈
    类型，spsr = r1(cpsr)
176     LDMNEIA sp!, {r0-r12, lr, pc}^          @ Return to point of thread
    interrupt // 中断栈类型恢复r0-r12, lr, pc并且恢复cpsr(指令带有^)，此处pc已经恢复了，
    中断栈后面指令不会执行了
177     LDMIA   sp!, {r4-r11, lr}               @ Return to thread synchronously //
    非中断栈类型(主动让出cpu)，r0-r3由编译器恢复(编译器根据需要在调用函数前保存r0-r3，调
    用完后恢复)或者不需要恢复
178     MSR     CPSR_cxsf, r1                   @   Recover CPSR // 直接恢复cpsr
179 #ifdef __THUMB_INTERWORK
180     BX      lr                              @ Return to caller
181 #else
182     MOV     pc, lr                          @ Return to caller // 寄存器等都恢复
    好了，直接跳转到线程让出cpu前的下一个地址即可
183 #endif
```

## 6、线程上下文保存(_tx_thread_context_save)

### 6.1、中断处理流程

以下是中断的顶层代码，___tx_irq_handler为中断入口，IRQ中断向量直接调用"B ___tx_irq_handler"即可，这样_tx_thread_context_save函数调用时，除了irq专有寄存器被修改外(例如pc、cpsr)，其他寄存器都没有动过，因此_tx_thread_context_save读

取的就是没有被改过的寄存器，_tx_thread_context_save也是通过"B"指令跳转过去的，然后通过"B"指令跳转返回的，BL指令会修改lr，所以简单起见都用不修改要保护的寄存器的指令间接实现函数调用。

```
211     .global __tx_irq_handler
212     .global __tx_irq_processing_return
213 __tx_irq_handler:
214 @
215 @    /* Jump to context save to save system context.  */
216     B       _tx_thread_context_save
217 __tx_irq_processing_return:
218 @
219 @    /* At this point execution is still in the IRQ mode.  The CPSR, point of
220 @       interrupt, and all C scratch registers are available for use.  In
221 @       addition, IRQ interrupts may be re-enabled - with certain restrictions -
222 @       if nested IRQ interrupts are desired.  Interrupts may be re-enabled over
223 @       small code sequences where lr is saved before enabling interrupts and
224 @       restored after interrupts are again disabled.  */
225 @
226 @    /* Interrupt nesting is allowed after calling _tx_thread_irq_nesting_start
227 @       from IRQ mode with interrupts disabled.  This routine switches to the
228 @       system mode and returns with IRQ interrupts enabled.
229 @
230 @       NOTE:  It is very important to ensure all IRQ interrupts are cleared
231 @       prior to enabling nested IRQ interrupts.  */
232 #ifdef TX_ENABLE_IRQ_NESTING
233     BL      _tx_thread_irq_nesting_start
234 #endif
235 @
236 @    /* For debug purpose, execute the timer interrupt processing here.  In
237 @       a real system, some kind of status indication would have to be checked
238 @       before the timer interrupt handler could be called.  */
239 @
240     BL      _tx_timer_interrupt                 @ Timer interrupt handler
241 @
242 @
243 @    /* If interrupt nesting was started earlier, the end of interrupt nesting
244 @       service must be called before returning to _tx_thread_context_restore.
245 @       This routine returns in processing in IRQ mode with interrupts disabled.  */
246 #ifdef TX_ENABLE_IRQ_NESTING
247     BL      _tx_thread_irq_nesting_end
248 #endif
249 @
250 @    /* Jump to context restore to restore system context.  */
251     B       _tx_thread_context_restore
```



## 6.2、中断上下文入栈

_tx_thread_context_save函数主要把中断代码用到或者影响到的一些关键寄存器入栈(IRQ栈)，有中断发生并不一定会切换线程，如果有线程执行，需要在真正换出线程的时候在保存线程上下文。

嵌套中断上下文保存/线程中断上下文寄存器保存代码如下：

```
 1. 092     .global _tx_thread_context_save
 2. 093     .type   _tx_thread_context_save,function
 3. 094 _tx_thread_context_save:
 4. 095 @
 5. 096 @    /* Upon entry to this routine, it is assumed that IRQ interrupts are locked
 6. 097 @       out, we are in IRQ mode, and all registers are intact.  */
 7. 098 @
 8. 099 @    /* Check for a nested interrupt condition.  */
 9. 100 @    if (_tx_thread_system_state++)
10. 101 @    {
11. 102 @
12. 103     STMDB   sp!, {r0-r3}                    @ Save some working registers // r0-
    r3保存到IRQ栈里面
13. 104 #ifdef TX_ENABLE_FIQ_SUPPORT
14. 105     MRS     r0, CPSR                        @ Pickup the CPSR
15. 106     ORR     r0, r0, #DISABLE_INTS           @ Build disable interrupt CPSR
16. 107     MSR     CPSR_cxsf, r0                   @ Disable interrupts
17. 108 #endif
18. 109     LDR     r3, =_tx_thread_system_state    @ Pickup address of system state
    variable
19. 110     LDR     r2, [r3]                        @ Pickup system state
20. 111     CMP     r2, #0                          @ Is this the first interrupt?
21. 112     BEQ     __tx_thread_not_nested_save     @ Yes, not a nested context save //
    第一次进入中断，非嵌套中断，跳转到__tx_thread_not_nested_save
22. 113 @
23. 114 @    /* Nested interrupt condition.  */
24. 115 @
25. 116     ADD     r2, r2, #1                      @ Increment the interrupt counter //
    嵌套中断，嵌套中断计数器加1
26. 117     STR     r2, [r3]                        @ Store it back in the variable
27. 118 @
28. 119 @   /* Save the rest of the scratch registers on the stack and return to the
29. 120 @      calling ISR.  */
```

```
121 @
122     MRS     r0, SPSR                        @ Pickup saved SPSR
123     SUB     lr, lr, #4                      @ Adjust point of interrupt // 中断返回地址修正
124     STMDB   sp!, {r0, r10, r12, lr}         @ Store other registers // cprs, r10, r12, lr(pc)入栈，后面会用到或者有指令会修改这些寄存器
125 @
126 @    /* Return to the ISR.  */
127 @
128     MOV     r10, #0                         @ Clear stack limit
129
130 #ifdef TX_ENABLE_EXECUTION_CHANGE_NOTIFY
131 @
132 @    /* Call the ISR enter function to indicate an ISR is executing.  */
133 @
134     PUSH    {lr}                            @ Save ISR lr
135     BL      _tx_execution_isr_enter         @ Call the ISR enter function
136     POP     {lr}                            @ Recover ISR lr
137 #endif
138
139     B       __tx_irq_processing_return      @ Continue IRQ processing // 嵌套中断上下文已经保存，返回_tx_thread_context_save下一条指令
140 @
141 __tx_thread_not_nested_save: // 非嵌套中断
142 @    }
143 @
144 @    /* Otherwise, not nested, check to see if a thread was running.  */
145 @    else if (_tx_thread_current_ptr)
146 @    {
147 @
148     ADD     r2, r2, #1                      @ Increment the interrupt counter // 非嵌套中断，嵌套中断计数器加1
149     STR     r2, [r3]                        @ Store it back in the variable
```

```
150     LDR     r1, =_tx_thread_current_ptr     @ Pickup address of current thread ptr
151     LDR     r0, [r1]                        @ Pickup current thread pointer
152     CMP     r0, #0                          @ Is it NULL?
153     BEQ     __tx_thread_idle_system_save    @ If so, interrupt occurred in
154                                             @   scheduling loop - nothing needs saving! // 检查有没有线程在执行，没有的话跳转到__tx_thread_idle_system_save，不需要保存线程上下文
155 @
156 @    /* Save minimal context of interrupted thread.  */
157 @
158     MRS     r2, SPSR                        @ Pickup saved SPSR // r2 = cpsr
159     SUB     lr, lr, #4                      @ Adjust point of interrupt // 修正中断返回地址
160     STMDB   sp!, {r2, r10, r12, lr}         @ Store other registers // cpsr, r10, r12, lr(pc)入栈(IRQ栈)(前面r0-r3已经入栈了，所以这里可以使用r0-r3作为工作寄存器了)
161 @
162 @    /* Save the current stack pointer in the thread's control block.  */
163 @    _tx_thread_current_ptr -> tx_thread_stack_ptr =  sp;
164 @
165 @    /* Switch to the system stack.  */
166 @    sp =  _tx_thread_system_stack_ptr@
167 @
168     MOV     r10, #0                         @ Clear stack limit
169
170 #ifdef TX_ENABLE_EXECUTION_CHANGE_NOTIFY
171 @
172 @    /* Call the ISR enter function to indicate an ISR is executing.  */
173 @
174     PUSH    {lr}                            @ Save ISR lr
175     BL      _tx_execution_isr_enter         @ Call the ISR enter function
176     POP     {lr}                            @ Recover ISR lr
177 #endif
178
```

```
88. 179     B       __tx_irq_processing_return      @ Continue IRQ processing // 部分用到的寄存器已经保存到IRQ的栈里面了，返回到_tx_thread_context_save的下一条指令

89. 180 @
```

非嵌套中断也没有线程在执行时，没有保存上下文，如果中断没有唤醒线程，中断退出将调用_tx_thread_schedule，_tx_thread_schedule没有参数也没有局部变量(不需要栈)，_tx_thread_schedule需要的只是寄存器，而且所有寄存器的值都是从内存加载，根本不需要保存，被中断的_tx_thread_schedule也不需要被恢复，重新调用_tx_thread_schedule即可；

如果有线程唤醒，那么_tx_thread_schedule也是不需要恢复的，_tx_thread_schedule没有任何数据，跟再次调用_tx_thread_schedule一个道理；前面已经讲过_tx_thread_context_restore了，中断退出时就调用_tx_thread_context_restore。

## 6.3、中断服务程序调用

必要的寄存器保存完成后，就可以调用C语言的中断服务程序了，本文只使用了定时器中断，所以就是定时器中断服务程序，定时器函数对线程时间片计数以及内核定时器timer计时，这里面可能唤醒新的线程。

## 6.4、中断上下文恢复

_tx_thread_context_restore上面章节已经讲过了；中断服务程序如果有线程唤醒或者抢占等情况发生，只会设置或者改变_tx_thread_execute_ptr，_tx_thread_execute_ptr指向最高优先级的线程(不考虑抢占就是最高优先级，考虑抢占就要把抢占阈值计算进来)，_tx_thread_context_restore来判断是恢复中断、恢复线程、调度先线程还是什么，最终由_tx_thread_schedule恢复线程上下文。

主动让出cpu的上下文保存比较简单，根据上下文恢复代码，不难理解保存上下文代码，在此略过。

## 7、总结

ThreadX内核代码与Nucleus Plus内核代码很多地方非常相似，上下文保存恢复调度基本逻辑相同，Nucleus Plus非开源，ThreadX微软已经在github开源了，支持很多cpu，在裸机代码上，把定时器/IRQ中断入口代码稍作修改即可运行在开发板上。

针对ARM Versatile/PB移植好的代码，地址如下，里面有修改几行代码，官网新版本已经修复：GitHub - arm7star/ThreadXContribute to arm7star/ThreadX development by creating an account on GitHub.

https://github.com/arm7star/ThreadX