(163条消息) ThreadX内核源码分析 - 事件_arm7star的博客-CSDN博客_threadx源码

1、ThreadX内核事件介绍

ThreadX事件有点类似epoll，线程可以等待单个事件/多个事件等(epoll一个事件就绪即可返回，ThreadX可以等待多个事件都就绪才返回)，从代码实现上看，ThreadX可以多个线程等待同一个事件，获取事件之后还可以不清除事件，epoll在网络编程中，似乎还没看到多个线程对一个事件监听的情况，具体能否多个线程调用epoll监听同一事件还得看linux内核代码实现；

ThreadX等待多个事件就绪才返回，这个实现比较实用，在ceph中等待多个worker结束时，通常需要多次调用join操作，一个一个线程调用，如果在ThreadX里面实现，就给每个线程一个事件，每个线程结束时设置一下自己的事件，然后主程序等待所有worker线程的事件都设置了即可返回。

2、事件获取_tx_event_flags_get

ThreadX获取事件允许同时等待多个事件或者等待其中一个事件即可，如果等待不到事件允许阻塞就把当前线程挂载到事件组的等待链表里面，有线程设置事件就会检查事件是否满足阻塞线程等待的事件，如果满足就会将事件给阻塞的线程并唤醒获取到事件的线程；

获取到事件后，如果要清除获取到的事件，那么需要检查是否有其他线程也在检查事件，如果有，那么需要等待其他线程处理完事件再清除事件；

对于一个等待事件线程，多个设置事件线程，类似epoll的场景，延迟清除事件是不存在的，因为一个等待线程，从下面代码看，设置事件的函数不会提前情况等待队列，设置事件的线程在处理只有一个等待线程的时候，不存在被中断的情况(处理事件过程被其他获取/设置事件的线程中断)。

_tx_event_flags_get实现代码如下：

```
081 UINT  _tx_event_flags_get(TX_EVENT_FLAGS_GROUP *group_ptr, ULONG
requested_flags,
082                     UINT get_option, ULONG *actual_flags_ptr, ULONG wait_option)
083 {
084
085 TX_INTERRUPT_SAVE_AREA
086
087 UINT            status;
088 UINT            and_request;
089 UINT            clear_request;
090 ULONG           current_flags;
091 ULONG           flags_satisfied;
092 #ifndef TX_NOT_INTERRUPTABLE
093 ULONG           delayed_clear_flags;
094 #endif
095 UINT            suspended_count;
096 TX_THREAD       *thread_ptr;
097 TX_THREAD       *next_thread;
098 TX_THREAD       *previous_thread;
099 #ifndef TX_NOT_INTERRUPTABLE
100 UINT            interrupted_set_request;
101 #endif
102
103
104     /* Disable interrupts to examine the event flags group.  */
105     TX_DISABLE
106
107 #ifdef TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO
108
109     /* Increment the total event flags get counter.  */
110     _tx_event_flags_performance_get_count++;
111
```

```
112     /* Increment the number of event flags gets on this semaphore.  */
113     group_ptr -> tx_event_flags_group__performance_get_count++;
114 #endif
115
116     /* If trace is enabled, insert this event into the trace buffer.  */
117     TX_TRACE_IN_LINE_INSERT(TX_TRACE_EVENT_FLAGS_GET, group_ptr,
    requested_flags, group_ptr -> tx_event_flags_group_current, get_option,
    TX_TRACE_EVENT_FLAGS_EVENTS)
118
119     /* Log this kernel call.  */
120     TX_EL_EVENT_FLAGS_GET_INSERT
121
122     /* Pickup current flags.  */
123     current_flags =  group_ptr -> tx_event_flags_group_current; // 获取group_ptr
    已有的事件(一组事件)
124
125     /* Apply the event flag option mask.  */
126     and_request =  (get_option & TX_AND); // get_option中的TX_AND是否被设置(是需
    要等一组事件还是等待其中一个事件即可，例如epoll等待socket可读，就并不需要等待所有
    socket都可以读，只要一个socket可以读即可返回，然后处理可读的socket即可)
127
128 #ifdef TX_NOT_INTERRUPTABLE
129
130     /* Check for AND condition. All flags must be present to satisfy request.
    */
131     if (and_request == TX_AND)
132     {
133
134         /* AND request is present.  */
135
136         /* Calculate the flags present.  */
137         flags_satisfied =  (current_flags & requested_flags);
138
139         /* Determine if they satisfy the AND request.  */
140         if (flags_satisfied != requested_flags)
```

```
141         {
142
143             /* No, not all the requested flags are present. Clear the flags
    present variable.  */
144             flags_satisfied =  ((ULONG) 0);
145         }
146     }
147     else
148     {
149
150         /* OR request is present. Simply or the requested flags and the current
    flags.  */
151         flags_satisfied =  (current_flags & requested_flags);
152     }
153
154     /* Determine if the request is satisfied.  */
155     if (flags_satisfied != ((ULONG) 0))
156     {
157
158         /* Return the actual event flags that satisfied the request.  */
159         *actual_flags_ptr =  current_flags;
160
161         /* Pickup the clear bit.  */
162         clear_request =  (get_option & TX_EVENT_FLAGS_CLEAR_MASK);
163
164         /* Determine whether or not clearing needs to take place.  */
165         if (clear_request == TX_TRUE)
166         {
167
168              /* Yes, clear the flags that satisfied this request.  */
169              group_ptr -> tx_event_flags_group_current =
170                                          group_ptr ->
    tx_event_flags_group_current & (~requested_flags);
```

```
171         }
172
173         /* Return success.  */
174         status =  TX_SUCCESS;
175     }
176
177 #else
178
179     /* Pickup delayed clear flags.  */
180     delayed_clear_flags =  group_ptr -> tx_event_flags_group_delayed_clear; // 延迟清除的事件
181
182     /* Determine if there are any delayed clear operations pending.  */
183     if (delayed_clear_flags != ((ULONG) 0))
184     {
185
186         /* Yes, apply them to the current flags.  */
187         current_flags =  current_flags & (~delayed_clear_flags); // 再次调用获取事件，需要清除之前延迟清除的事件
188     }
189
190     /* Check for AND condition. All flags must be present to satisfy request.  */
191     if (and_request == TX_AND) // 如果设置了TX_AND(同时等待多个事件就绪)
192     {
193
194         /* AND request is present.  */
195
196         /* Calculate the flags present.  */
197         flags_satisfied =  (current_flags & requested_flags); // 已就绪的事件current_flags & 需要等待的事件requested_flags = 等待的事件有多少事件就绪(只保留等待事件中已经就绪的事件，其他非等待的就绪事件不会保留在flags_satisfied里面)
198
199         /* Determine if they satisfy the AND request.  */
```

```
120. 200         if (flags_satisfied != requested_flags) // 如果不是所有等待事件都就绪，那么设置flags_satisfied为0，否则保留flags_satisfied(后面还需要用到flags_satisfied判断是否等待到了事件)
121. 201         {
122. 202
123. 203             /* No, not all the requested flags are present. Clear the flags present variable.  */
124. 204             flags_satisfied =  ((ULONG) 0);
125. 205         }
126. 206     }
127. 207     else // 只要等待一个事件即可(有多个也无所谓，这里与epoll一样)
128. 208     {
129. 209
130. 210         /* OR request is present. Simply AND together the requested flags and the current flags
131. 211            to see if any are present.  */
132. 212         flags_satisfied =  (current_flags & requested_flags); // 不需要等待的事件清0，flags_satisfied只保留等待的并且就绪的事件(每个二进制位一个事件)
133. 213     }
134. 214
135. 215     /* Determine if the request is satisfied.  */
136. 216     if (flags_satisfied != ((ULONG) 0)) // 用flags_satisfied是否为0判断是否等待到了事件，这里也就是上面等待多个事件没有等待到所有事件时要清零flags_satisfied的原因，这里flags_satisfied不为0就是所有等待到了的事件
137. 217     {
138. 218
139. 219         /* Yes, this request can be handled immediately.  */
140. 220
141. 221         /* Return the actual event flags that satisfied the request.  */
142. 222         *actual_flags_ptr =  current_flags; // actual_flags_ptr记录当前所有就绪的事件(包括非等待的就绪事件)
143. 223
144. 224         /* Pickup the clear bit.  */
145. 225         clear_request =  (get_option & TX_EVENT_FLAGS_CLEAR_MASK); // 清除获取到事件的选项(获取到事件后是否清除事件)
146. 226
```

```
147. 227         /* Determine whether or not clearing needs to take place.  */

148. 228         if (clear_request == TX_TRUE) // 如果设置了清除事件选项
     TX_EVENT_FLAGS_CLEAR_MASK，那么清除当前获取到的事件(例如：有多个主线程，有多个work线
     程，多个主线程都等待work线程结束，每个work线程结束时都设置一下事件，那么这些事件不能
     清除，否则别的线程等不到线程结束的事件)

149. 229         {

150. 230

151. 231             /* Set interrupted set request flag to false.  */

152. 232             interrupted_set_request =  TX_FALSE;

153. 233

154. 234             /* Determine if the suspension list is being processed by an
     interrupted

155. 235                set request.  */

156. 236             if (group_ptr -> tx_event_flags_group_suspended_count !=
     TX_NO_SUSPENSIONS) // tx_event_flags_group_suspended_count不为0，那么有线程在等待事
     件

157. 237             {

158. 238

159. 239                 if (group_ptr -> tx_event_flags_group_suspension_list ==
     TX_NULL) // tx_event_flags_group_suspension_list为空，那么有其他线程在操作
     tx_event_flags_group_suspension_list，_tx_event_flags_set设置事件的线程处理等待链表
     时，会先取tx_event_flags_group_suspension_list，然后
     tx_event_flags_group_suspension_list设置为空，tx_event_flags_group_suspension_list，
     因为处理tx_event_flags_group_suspension_list比较耗时，不能锁住
     tx_event_flags_group_suspension_list

160. 240                 {

161. 241

162. 242                     /* Set the interrupted set request flag.  */

163. 243                     interrupted_set_request =  TX_TRUE;

164. 244                 }

165. 245             }

166. 246

167. 247             /* Was a set request interrupted?  */

168. 248             if (interrupted_set_request == TX_TRUE) // 调用_tx_event_flags_set设
     置事件的线程也正在检查当前的事件是否满足等待事件的线程，不能清除掉这些事件，也可以理
     解为，这时的事件是所有线程都可以获取的，等所有线程都检查完了，再清除这些事件

169. 249             {

170. 250
```

```
251                 /* A previous set operation is was interrupted, we need to defer
    the
252                    event clearing until the set operation is complete.  */
253
254                 /* Remember the events to clear.  */
255                 group_ptr -> tx_event_flags_group_delayed_clear =
256                                             group_ptr ->
    tx_event_flags_group_delayed_clear | requested_flags; // 先不清除事件，需要清除的事
    件保存到tx_event_flags_group_delayed_clear，下次设置事件或者获取事件的时候在清除这些
    事件(如果下次是先调用设置事件，那么先清除这些事件，下次就不会获取到旧的事件，如果下
    次是先调用获取事件，那么也先清除这些事件，这样也不会获取到旧的事件)
257             }
258             else
259             {
260
261                 /* Yes, clear the flags that satisfied this request.  */
262                 group_ptr -> tx_event_flags_group_current =
263                                             group_ptr ->
    tx_event_flags_group_current & ~requested_flags; // 没有等待事件的线程被中断，清除当
    前获取到的事件
264             }
265         }
266
267         /* Set status to success.  */
268         status =  TX_SUCCESS; // 获取到了事件，返回成功即可
269     }
270
271 #endif
272     else // 没有获取到事件
273     {
274
275         /* Determine if the request specifies suspension.  */
276         if (wait_option != TX_NO_WAIT) // 没有设置TX_NO_WAIT，也就是阻塞获取事件
277         {
278
```

```
199. 279             /* Determine if the preempt disable flag is non-zero.  */

200. 280             if (_tx_thread_preempt_disable != ((UINT) 0)) // 禁止了抢占，不能阻塞当前线程，否则其他线程也得不到调度

201. 281             {

202. 282

203. 283                 /* Suspension is not allowed if the preempt disable flag is non-zero at this point, return error completion.  */

204. 284                 status =  TX_NO_EVENTS; // 返回没有事件即可，由上层函数决定是再次获取事件还是怎么处理

205. 285             }

206. 286             else // 没有禁止抢占，需要阻塞当前线程，需要注意，到这里中断都是关闭的，线程没有挂到等待队列，如果允许中断，就保证不了被其他线程抢占，其他线程正好设置了事件，因为后面代码不再判断事件，所以在挂起当前线程前，不能有其他线程设置事件

207. 287             {

208. 288

209. 289                 /* Prepare for suspension of this thread.  */

210. 290

211. 291 #ifdef TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO

212. 292

213. 293                 /* Increment the total event flags suspensions counter.  */

214. 294                 _tx_event_flags_performance_suspension_count++;

215. 295

216. 296                 /* Increment the number of event flags suspensions on this semaphore.  */

217. 297                 group_ptr -> tx_event_flags_group___performance_suspension_count++;

218. 298 #endif

219. 299

220. 300                 /* Pickup thread pointer.  */

221. 301                 TX_THREAD_GET_CURRENT(thread_ptr) // 获取当前线程 _tx_thread_current_ptr

222. 302

223. 303                 /* Setup cleanup routine pointer.  */

224. 304                 thread_ptr -> tx_thread_suspend_cleanup =  &(_tx_event_flags_cleanup); // 设置等待超时以及线程终止等清理回调函数(等待事件超时要通过_tx_event_flags_cleanup回调函数唤醒当前线程，删除等待队列，线程终止也要删除等待队列)
```

```
225. 305

226. 306                /* Remember which event flags we are looking for.  */

227. 307                thread_ptr -> tx_thread_suspend_info =  requested_flags; // 等待
     的事件(如果有线程设置事件，那么会检查是否事件满足等待线程的等待事件)

228. 308

229. 309                /* Save the get option as well.  */

230. 310                thread_ptr -> tx_thread_suspend_option =  get_option; // 等待事
     件的选项(设置事件的线程也需要知道等待线程是等待一个事件，还是要等待所有事件)

231. 311

232. 312                /* Save the destination for the current events.  */

233. 313                thread_ptr -> tx_thread_additional_suspend_info =  (VOID *)
     actual_flags_ptr; // 线程获取到事件或者超被唤醒时，会设置当前所有就绪的事件到
     actual_flags_ptr

234. 314

235. 315                /* Setup cleanup information, i.e. this event flags group
     control

236. 316                   block.  */

237. 317                thread_ptr -> tx_thread_suspend_control_block =  (VOID *)
     group_ptr; // 等待的事件组

238. 318

239. 319 #ifndef TX_NOT_INTERRUPTABLE

240. 320

241. 321                /* Increment the suspension sequence number, which is used to
     identify

242. 322                   this suspension event.  */

243. 323                thread_ptr -> tx_thread_suspension_sequence++;

244. 324 #endif

245. 325

246. 326                /* Pickup the suspended count.  */

247. 327                suspended_count =  group_ptr ->
     tx_event_flags_group_suspended_count; // 获取有多少线程在等待事件

248. 328

249. 329                /* Setup suspension list.  */

250. 330                if (suspended_count == TX_NO_SUSPENSIONS) // 没有其他线程等待事
     件，那么新建一个等待链表，该链表只有当前线程

251. 331                {
```

```
252. 332
253. 333                    /* No other threads are suspended.  Setup the head pointer
     and
254. 334                       just setup this threads pointers to itself.  */
255. 335                    group_ptr -> tx_event_flags_group_suspension_list =
     thread_ptr;
256. 336                    thread_ptr -> tx_thread_suspended_next =
     thread_ptr;
257. 337                    thread_ptr -> tx_thread_suspended_previous =
     thread_ptr;
258. 338                }
259. 339                else // 有其他线程也在等待事件，将当前线程添加到等待链表末尾即可
260. 340                {
261. 341
262. 342                    /* This list is not NULL, add current thread to the end. */
263. 343                    next_thread =                                   group_ptr ->
     tx_event_flags_group_suspension_list;
264. 344                    thread_ptr -> tx_thread_suspended_next =        next_thread;
265. 345                    previous_thread =                               next_thread
     -> tx_thread_suspended_previous;
266. 346                    thread_ptr -> tx_thread_suspended_previous =
     previous_thread;
267. 347                    previous_thread -> tx_thread_suspended_next =   thread_ptr;
268. 348                    next_thread -> tx_thread_suspended_previous =   thread_ptr;
269. 349                }
270. 350
271. 351                /* Increment the number of threads suspended.  */
272. 352                group_ptr -> tx_event_flags_group_suspended_count++; // 等待事件
     的线程数加1
273. 353
274. 354                /* Set the state to suspended.  */
275. 355                thread_ptr -> tx_thread_state =    TX_EVENT_FLAG; // 线程状态设
     置为TX_EVENT_FLAG
276. 356
277. 357 #ifdef TX_NOT_INTERRUPTABLE
278. 358
```

```
279. 359                /* Call actual non-interruptable thread suspension routine.  */

280. 360                _tx_thread_system_ni_suspend(thread_ptr, wait_option);

281. 361

282. 362                /* Return the completion status.  */

283. 363                status =  thread_ptr -> tx_thread_suspend_status;

284. 364 #else

285. 365

286. 366                /* Set the suspending flag.  */

287. 367                thread_ptr -> tx_thread_suspending =  TX_TRUE; // 线程正在挂起
     中，后面挂起线程允许中断，可能有中断服务程序或者其他操作也修改当前线程(挂起或者唤醒
     当前线程等操作)，tx_thread_suspending禁止一些其他不必要的操作，例如线程不能被唤醒，
     线程唤醒也获取不到事件，没有必要也不能唤醒

288. 368

289. 369                /* Setup the timeout period.  */

290. 370                thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks
     =  wait_option; // 等待事件超时时间(_tx_thread_system_suspend需要检查
     tx_timer_internal_remaining_ticks，以确定是否要启动超时定时器)

291. 371

292. 372                /* Temporarily disable preemption.  */

293. 373                _tx_thread_preempt_disable++; // _tx_thread_system_suspend会对
     _tx_thread_preempt_disable减1，调用_tx_thread_system_suspend前必须对
     _tx_thread_preempt_disable加1

294. 374

295. 375                /* Restore interrupts.  */

296. 376                TX_RESTORE // 允许中断(到这里才开启中断，因此等待线程数目加1与挂
     载等待链表是在关中断情况下进行的，就如前面所说，想想不到什么情况等待计数器不为0但是
     等待链表为空的情况，出发有只增加计数器不挂载等待链表的操作)

297. 377

298. 378                /* Call actual thread suspension routine.  */

299. 379                _tx_thread_system_suspend(thread_ptr); // 挂起当前线程，切换到其
     他线程执行

300. 380

301. 381                /* Disable interrupts.  */

302. 382                TX_DISABLE

303. 383

304. 384                /* Return the completion status.  */
```

```
305. 385                 status =  thread_ptr -> tx_thread_suspend_status; // 别的线程设置事件，唤醒当前线程会设置tx_thread_suspend_status为成功，定时器超时会设置tx_thread_suspend_status为超时，与计数信号量处理一样
306. 386 #endif
307. 387             }
308. 388         }
309. 389         else // 非阻塞获取不到事件，设置为TX_NO_EVENTS，等待的事件没有就绪，返回
310. 390         {
311. 391
312. 392             /* Immediate return, return error completion.  */
313. 393             status =  TX_NO_EVENTS;
314. 394         }
315. 395     }
316. 396
317. 397     /* Restore interrupts.  */
318. 398     TX_RESTORE
319. 399
320. 400     /* Return completion status.  */
321. 401     return(status);
322. 402 }
323. 403
```

3、事件设置_tx_event_flags_set

设置事件比获取事件复杂一些，ThreadX内核设置事件的时候是直接把事件给阻塞的等待事件的线程，而不是唤醒所有等待线程，让所有线程重新去获取事件，设置事件把事件给等待事件线程，效率高一些，代码也复杂一点点。

检查事件过程有一个tx_event_flags_group_reset_search变量，这个变量主要是标志是事件/线程状态是否有更新；_tx_event_flags_set在检查等待事件线程链表时，会把等待链表及就绪事件取出到本地，中断服务程序等没办法从等待链表删除线程，别的线程设置事件也不会检查被取出的等待链表，所以，_tx_event_flags_set当前的事件或者处理的等待线程状态可能有变化(线程终止了或者不再等待事件)，处理等待链表的线程检测到tx_event_flags_group_reset_search为真，就得重新检查事件及等待事件线程链表；

检查事件过程还有一个preempt_check变量，在有线程获取到事件的时候会设置(获取到事件的线程会被唤醒，但是唤醒过程是禁止抢占的，当前线程可能被抢占，处理完事件后，允许抢占时，需要检查抢占)(这个代码似乎有个bug，preempt_check只在有线程获取到事

件才设置，但是禁止抢占期间没有完全关中断，中断服务程序也可能唤醒高优先级线程，所以只要禁止抢占期间开了中断，都要检查抢占)。

_tx_event_flags_set代码实现如下：

```
1. 080 UINT  _tx_event_flags_set(TX_EVENT_FLAGS_GROUP *group_ptr, ULONG flags_to_set,
   UINT set_option)
2. 081 {
3. 082
4. 083 TX_INTERRUPT_SAVE_AREA
5. 084
6. 085 TX_THREAD       *thread_ptr;
7. 086 TX_THREAD       *next_thread_ptr;
8. 087 TX_THREAD       *next_thread;
9. 088 TX_THREAD       *previous_thread;
10. 089 TX_THREAD       *satisfied_list;
11. 090 TX_THREAD       *last_satisfied;
12. 091 TX_THREAD       *suspended_list;
13. 092 UINT            suspended_count;
14. 093 ULONG           current_event_flags;
15. 094 ULONG           requested_flags;
16. 095 ULONG           flags_satisfied;
17. 096 ULONG           *suspend_info_ptr;
18. 097 UINT            and_request;
19. 098 UINT            get_option;
20. 099 UINT            clear_request;
21. 100 UINT            preempt_check;
22. 101 #ifndef TX_NOT_INTERRUPTABLE
23. 102 UINT            interrupted_set_request;
24. 103 #endif
25. 104 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
26. 105 VOID            (*events_set_notify)(struct TX_EVENT_FLAGS_GROUP_STRUCT
    *notify_group_ptr);
27. 106 #endif
28. 107
29. 108
30. 109     /* Disable interrupts to remove the semaphore from the created list.  */
```

```
110     TX_DISABLE
111
112 #ifdef TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO
113
114     /* Increment the total event flags set counter.  */
115     _tx_event_flags_performance_set_count++;
116
117     /* Increment the number of event flags sets on this semaphore.  */
118     group_ptr -> tx_event_flags_group_performance_set_count++;
119 #endif
120
121     /* If trace is enabled, insert this event into the trace buffer.  */
122     TX_TRACE_IN_LINE_INSERT(TX_TRACE_EVENT_FLAGS_SET, group_ptr, flags_to_set,
    set_option, group_ptr -> tx_event_flags_group_suspended_count,
    TX_TRACE_EVENT_FLAGS_EVENTS)
123
124     /* Log this kernel call.  */
125     TX_EL_EVENT_FLAGS_SET_INSERT
126
127     /* Determine how to set this group's event flags.  */
128     if ((set_option & TX_EVENT_FLAGS_AND_MASK) == TX_AND) // TX_AND从后面代码
    看，这个TX_AND在设置事件函数里面应该是清除事件的作用，flags_to_set为0的事件被清除，
    flags_to_set为1的事件被保留(如果原来就绪的话)
129     {
130
131 #ifndef TX_NOT_INTERRUPTABLE
132
133         /* Set interrupted set request flag to false.  */
134         interrupted_set_request =  TX_FALSE;
135
136         /* Determine if the suspension list is being processed by an interrupted
137            set request.  */
138         if (group_ptr -> tx_event_flags_group_suspended_count !=
    TX_NO_SUSPENSIONS) // 与获取事件一样...
```

```
139         {
140
141             if (group_ptr -> tx_event_flags_group_suspension_list == TX_NULL)
142             {
143
144                 /* Set the interrupted set request flag.  */
145                 interrupted_set_request =  TX_TRUE;
146             }
147         }
148
149         /* Was a set request interrupted?  */
150         if (interrupted_set_request == TX_TRUE)
151         {
152
153             /* A previous set operation was interrupted, we need to defer the
154                event clearing until the set operation is complete.  */
155
156             /* Remember the events to clear.  */
157             group_ptr -> tx_event_flags_group_delayed_clear =
158                                             group_ptr ->
    tx_event_flags_group_delayed_clear | ~flags_to_set;
159         }
160         else
161         {
162 #endif
163
164             /* Previous set operation was not interrupted, simply clear the
165                specified flags by "ANDing" the flags into the current events
166                of the group.  */
167             group_ptr -> tx_event_flags_group_current =
168                  group_ptr -> tx_event_flags_group_current & flags_to_set; // 清
    除事件(注意这里不是设置事件，这里用的是&)
169
```

```
170 #ifndef TX_NOT_INTERRUPTABLE
171
172         }
173 #endif
174
175         /* Restore interrupts.  */
176         TX_RESTORE // 开中断，返回即可
177     }
178     else // 设置事件
179     {
180
181 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
182
183         /* Pickup the notify callback routine for this event flag group.  */
184         events_set_notify =  group_ptr -> tx_event_flags_group_set_notify;
185 #endif
186
187         /* "OR" the flags into the current events of the group.  */
188         group_ptr -> tx_event_flags_group_current =
189             group_ptr -> tx_event_flags_group_current | flags_to_set; // 设置事件(|)
190
191 #ifndef TX_NOT_INTERRUPTABLE
192
193         /* Determine if there are any delayed flags to clear.  */
194         if (group_ptr -> tx_event_flags_group_delayed_clear != ((ULONG) 0))
195         {
196
197             /* Yes, we need to neutralize the delayed clearing as well.  */
198             group_ptr -> tx_event_flags_group_delayed_clear =
199                                         group_ptr -> tx_event_flags_group_delayed_clear & ~flags_to_set; // 清除延迟清除的事件
200         }
```

```
122. 201 #endif

123. 202

124. 203         /* Clear the preempt check flag.  */

125. 204         preempt_check =  TX_FALSE; // 抢占检查设置为TX_FALSE，设置事件后可能唤醒
     等待线程，可能存在抢占

126. 205

127. 206         /* Pickup the thread suspended count.  */

128. 207         suspended_count =  group_ptr -> tx_event_flags_group_suspended_count; //
     多少线程在等待事件

129. 208

130. 209         /* Determine if there are any threads suspended on the event flag group.
     */

131. 210         if (group_ptr -> tx_event_flags_group_suspension_list != TX_NULL) // 如
     果等待链表不为空，那么有线程等待事件

132. 211         {

133. 212

134. 213             /* Determine if there is just a single thread waiting on the event

135. 214                flag group.  */

136. 215             if (suspended_count == ((UINT) 1)) // 如果只有一个线程等待事件，那么
     只有检查事件是否满足该等待事件的线程即可

137. 216             {

138. 217

139. 218                 /* Single thread waiting for event flags.  Bypass the multiple
     thread

140. 219                    logic.  */

141. 220

142. 221                 /* Setup thread pointer.  */

143. 222                 thread_ptr =  group_ptr -> tx_event_flags_group_suspension_list;
     // 等待事件的线程

144. 223

145. 224                 /* Pickup the current event flags.  */

146. 225                 current_event_flags =  group_ptr ->
     tx_event_flags_group_current; // 当前的所有就绪事件

147. 226

148. 227                 /* Pickup the suspend information.  */
```

```
228                 requested_flags =  thread_ptr -> tx_thread_suspend_info; // 阻塞线程等待的事件
229
230                 /* Pickup the suspend option.  */
231                 get_option =  thread_ptr -> tx_thread_suspend_option; // 阻塞线程等待事件选项(等待一个或者等待多个就绪)
232
233                 /* Isolate the AND selection.  */
234                 and_request =  (get_option & TX_AND); // 等待事件的线程thread_ptr等待多个事件就绪?
235
236                 /* Check for AND condition. All flags must be present to satisfy request.  */
237                 if (and_request == TX_AND) // thread_ptr等待多个事件就绪
238                 {
239
240                     /* AND request is present.  */
241
242                     /* Calculate the flags present.  */
243                     flags_satisfied =  (current_event_flags & requested_flags); // flags_satisfied获取thread_ptr等待的就绪的事件(没有就绪的事件为0)
244
245                     /* Determine if they satisfy the AND request.  */
246                     if (flags_satisfied != requested_flags) // 不相等，则requested_flags等待事件有没有就绪的
247                     {
248
249                         /* No, not all the requested flags are present. Clear the flags present variable.  */
250                         flags_satisfied =  ((ULONG) 0); // 设置为0，表示thread_ptr没有等待到事件(只部分事件等到了，但是设置了TX_AND，要继续等待所有事件就绪才行)
251                     }
252                 }
253                 else // 只要一个事件就绪即可(类似epoll等待一个socket就绪即可)
254                 {
```

```
176. 255

177. 256                    /* OR request is present. Simply or the requested flags and
     the current flags.  */

178. 257                    flags_satisfied =  (current_event_flags & requested_flags);
     // flags_satisfied记录所有等待就绪的事件

179. 258                }

180. 259

181. 260                /* Determine if the request is satisfied.  */

182. 261                if (flags_satisfied != ((ULONG) 0)) // 如果有等待到事件，那么需
     要唤醒等待线程thread_ptr

183. 262                {

184. 263

185. 264                    /* Yes, resume the thread and apply any event flag

186. 265                       clearing.  */

187. 266

188. 267                    /* Set the preempt check flag.  */

189. 268                    preempt_check =  TX_TRUE; // 事件满足阻塞的等待事件线程等待
     的事件，需要唤醒阻塞线程，唤醒就可能有抢占，因此抢占检查preempt_check设置为TX_TRUE

190. 269

191. 270                    /* Return the actual event flags that satisfied the request.
     */

192. 271                    suspend_info_ptr =
     TX_VOID_TO_ULONG_POINTER_CONVERT(thread_ptr -> tx_thread_additional_suspend_info);
     // 当前所有就绪事件保存到actual_flags_ptr(等待事件的线程在挂起前把actual_flags_ptr保
     存到了tx_thread_additional_suspend_info)

193. 272                    *suspend_info_ptr =  current_event_flags; // 所有就绪事件保
     存到actual_flags_ptr(包括非等待的事件)

194. 273

195. 274                    /* Pickup the clear bit.  */

196. 275                    clear_request =  (get_option & TX_EVENT_FLAGS_CLEAR_MASK);
     // 获取到事件之后清除获取到的事件? TX_EVENT_FLAGS_CLEAR_MASK

197. 276

198. 277                    /* Determine whether or not clearing needs to take place.
     */

199. 278                    if (clear_request == TX_TRUE) // 获取到事件之后清除已经获取
     到的事件(该事件将被处理)

200. 279                    {

201. 280
```

```
281                         /* Yes, clear the flags that satisfied this request.  */
282                         group_ptr -> tx_event_flags_group_current =  group_ptr -> tx_event_flags_group_current & (~requested_flags); // 清除获取到的事件
283                     }
284
285                     /* Clear the suspension information in the event flag group.  */
286                     group_ptr -> tx_event_flags_group_suspension_list = TX_NULL; // 只有一个线程等待事件，现在该线程获取到了事件，那么等待队列就设置为空(没有线程等待事件)
287                     group_ptr -> tx_event_flags_group_suspended_count = TX_NO_SUSPENSIONS; // 没有线程等待事件，等待计数器设置为0即可
288
289                     /* Clear cleanup routine to avoid timeout.  */
290                     thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL; // 清理函数设置为空
291
292                     /* Put return status into the thread control block.  */
293                     thread_ptr -> tx_thread_suspend_status =  TX_SUCCESS; // 获取到事件，thread_ptr阻塞状态设置为成功(线程唤醒后，用tx_thread_suspend_status判断是超时还是获取到了事件)
294
295 #ifdef TX_NOT_INTERRUPTABLE
296
297                     /* Resume the thread!  */
298                     _tx_thread_system_ni_resume(thread_ptr);
299 #else
300
301                     /* Temporarily disable preemption.  */
302                     _tx_thread_preempt_disable++;
303
304                     /* Restore interrupts.  */
305                     TX_RESTORE
306
307                     /* Resume thread.  */
```

```
308                     _tx_thread_system_resume(thread_ptr); // 唤醒等待事件的线程 thread_ptr
309
310                     /* Disable interrupts to remove the semaphore from the created list.  */
311                     TX_DISABLE
312 #endif
313                 }
314             }
315             else // 有多个线程在等待事件(每个等待线程等待的事件不完全一样，需要逐个检查事件是否满足等待的线程)
316             {
317
318                 /* Otherwise, the event flag requests of multiple threads must be
319                    examined.  */
320
321                 /* Setup thread pointer, keep a local copy of the head pointer. */
322                 suspended_list =  group_ptr -> tx_event_flags_group_suspension_list; // 获取等待线程链表
323                 thread_ptr =      suspended_list; // 第一个等待事件的线程
324
325                 /* Clear the suspended list head pointer to thwart manipulation of
326                    the list in ISR's while we are processing here.  */
327                 group_ptr -> tx_event_flags_group_suspension_list =  TX_NULL; // tx_event_flags_group_suspension_list设置为空，等待链表已经取到suspended_list里面了；tx_event_flags_group_suspended_count没有改变，因此获取事件的函数可以检测到有线程在处理事件，不能立即清除事件
328
329                 /* Setup the satisfied thread pointers.  */
330                 satisfied_list =  TX_NULL; // 记录获取到事件的线程链表
331                 last_satisfied =  TX_NULL; // satisfied_list指向satisfied_list的最后一个线程，以便快速在satisfied_list末尾插入线程
332
333                 /* Pickup the current event flags.  */
```

```
255. 334                current_event_flags =  group_ptr ->
     tx_event_flags_group_current;

256. 335

257. 336                /* Disable preemption while we process the suspended list.  */

258. 337                _tx_thread_preempt_disable++; // 禁止抢占(后面会允许中断，不禁止
     抢占的话，处理过程就可能被切换出去)

259. 338

260. 339                /* Loop to examine all of the suspended threads. */

261. 340                do

262. 341                {

263. 342

264. 343 #ifndef TX_NOT_INTERRUPTABLE

265. 344

266. 345                    /* Restore interrupts temporarily.  */

267. 346                    TX_RESTORE // 允许中断，避免阻塞中断处理

268. 347

269. 348                    /* Disable interrupts again.  */

270. 349                    TX_DISABLE // 再次关闭中断，开中断之后的中断都处理完了，暂时
     再次关闭中断

271. 350 #endif

272. 351

273. 352                    /* Determine if we need to reset the search.  */

274. 353                    if (group_ptr -> tx_event_flags_group_reset_search !=
     TX_FALSE) // 搜索过程被中断了，前面禁止了抢占，另外根据
     tx_event_flags_group_reset_search的注释，应该只有中断服务程序ISR会设置
     tx_event_flags_group_reset_search为TX_TRUE，也就是如果中断服务程序改变了等待链表，那
     么需要重新检查事件及等待线程链表

275. 354                    {

276. 355

277. 356                        /* Clear the reset search flag.  */

278. 357                        group_ptr -> tx_event_flags_group_reset_search =
     TX_FALSE;

279. 358

280. 359                        /* Move the thread pointer to the beginning of the
     search list.  */
```

281. 360   thread_ptr = suspended_list; // thread_ptr重新指向阻塞链表表头(suspended_list是当前线程的局部变量，中断服务程序等改变不了suspended_list，所以被中断后，还可以从suspended_list重新遍历等待事件的线程链表)

282. 361

283. 362   /* Reset the suspended count. */

284. 363   suspended_count = group_ptr -> tx_event_flags_group_suspended_count; // 重新获取等待事件的线程数(从代码上下文看，中断服务程序不会改变tx_event_flags_group_suspended_count，也就是tx_event_flags_group_suspended_count一直等于suspended_list的大小，另外中断服务程序也不会有获取事件操作，最多应该是改变等待事件的线程状态)

285. 364

286. 365   /* Update the current events with any new ones that might

287. 366   have been set in a nested set events call from an ISR. */

288. 367   current_event_flags = current_event_flags | group_ptr -> tx_event_flags_group_current; // 更新事件(current_event_flags已经记录了旧的事件，tx_event_flags_group_current为当前最新的事件，也就是如果有新的事件，那么加入旧的事件里面)

289. 368   }

290. 369

291. 370   /* Save next thread pointer. */

292. 371   next_thread_ptr = thread_ptr -> tx_thread_suspended_next; // 下一个等待事件的线程

293. 372

294. 373   /* Pickup the suspend information. */

295. 374   requested_flags = thread_ptr -> tx_thread_suspend_info; // thread_ptr等待的事件

296. 375

297. 376   /* Pickup this thread's suspension get option. */

298. 377   get_option = thread_ptr -> tx_thread_suspend_option; // thread_ptr等待事件的选项(一个事件or多个事件)

299. 378

300. 379   /* Isolate the AND selection. */

301. 380   and_request = (get_option & TX_AND); // TX_AND选项

302. 381

303. 382   /* Check for AND condition. All flags must be present to satisfy request. */

```
304. 383                    if (and_request == TX_AND) // 设置了TX_AND选项，thread_ptr需要一次等待所有等待事件就绪才行(后面if...else...获取就绪事件前面已经前面章节已经介绍了，不再介绍...)
305. 384                    {
306. 385
307. 386                        /* AND request is present.  */
308. 387
309. 388                        /* Calculate the flags present.  */
310. 389                        flags_satisfied =  (current_event_flags & requested_flags);
311. 390
312. 391                        /* Determine if they satisfy the AND request.  */
313. 392                        if (flags_satisfied != requested_flags)
314. 393                        {
315. 394
316. 395                            /* No, not all the requested flags are present.  Clear the flags present variable.  */
317. 396                            flags_satisfied =  ((ULONG) 0);
318. 397                        }
319. 398                    }
320. 399                    else
321. 400                    {
322. 401
323. 402                        /* OR request is present. Simply or the requested flags and the current flags.  */
324. 403                        flags_satisfied =  (current_event_flags & requested_flags);
325. 404                    }
326. 405
327. 406                    /* Check to see if the thread had a timeout or wait abort during the event search processing.
328. 407                       If so, just set the flags satisfied to ensure the processing here removes the thread from
329. 408                       the suspension list.  */
```

```
330. 409                    if (thread_ptr -> tx_thread_state != TX_EVENT_FLAG) // 线程的状态已经不是TX_EVENT_FLAG了(中断服务程序可能终止了线程，不管是否获取到事件，都设置为获取到了事件，这样才能从等待链表删除线程，前面已经将等待链表取出到suspended_list了，中断服务程序不能操作suspended_list，所以，中断服务程序最多改变线程状态，还得当前线程从等待链表删除该线程)

331. 410                    {

332. 411

333. 412                       /* Simply set the satisfied flags to 1 in order to remove the thread from the suspension list.  */

334. 413                        flags_satisfied =  ((ULONG) 1);

335. 414                    }

336. 415

337. 416                    /* Determine if the request is satisfied.  */

338. 417                    if (flags_satisfied != ((ULONG) 0))

339. 418                    {

340. 419

341. 420                        /* Yes, this request can be handled now.  */

342. 421

343. 422                        /* Set the preempt check flag.  */

344. 423                        preempt_check =  TX_TRUE; // 有线程获取到了事件(或者线程状态改变了，因为前面禁止了抢占，那么可能有更高优先级线程就绪，需要检查抢占...)

345. 424

346. 425                        /* Determine if the thread is still suspended on the event flag group. If not, a wait

347. 426                           abort must have been done from an ISR.  */

348. 427                        if (thread_ptr -> tx_thread_state == TX_EVENT_FLAG) // 如果线程还在等待事件(没有被中断服务程序改变状态或者终止)，那么把事件给线程thread_ptr

349. 428                        {

350. 429

351. 430                            /* Return the actual event flags that satisfied the request.  */

352. 431                            suspend_info_ptr = TX_VOID_TO_ULONG_POINTER_CONVERT(thread_ptr -> tx_thread_additional_suspend_info);

353. 432                            *suspend_info_ptr =  current_event_flags;

354. 433

355. 434                            /* Pickup the clear bit.  */
```

```
356. 435                            clear_request =  (get_option &
     TX_EVENT_FLAGS_CLEAR_MASK);

357. 436

358. 437                            /* Determine whether or not clearing needs to take
     place.  */

359. 438                            if (clear_request == TX_TRUE)

360. 439                            {

361. 440

362. 441                                /* Yes, clear the flags that satisfied this
     request.  */

363. 442                                group_ptr -> tx_event_flags_group_current =
     group_ptr -> tx_event_flags_group_current & ~requested_flags; // 这里清除了获取到的
     事件!!!

364. 443                            }

365. 444

366. 445                            /* Prepare for resumption of the first thread.  */

367. 446

368. 447                            /* Clear cleanup routine to avoid timeout.  */

369. 448                            thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL;

370. 449

371. 450                            /* Put return status into the thread control block.
     */

372. 451                            thread_ptr -> tx_thread_suspend_status =
     TX_SUCCESS; // 线程阻塞状态更新为成功状态(已经获取到了事件，后面再唤醒)

373. 452                        }

374. 453

375. 454                        /* We need to remove the thread from the suspension list
     and place it in the

376. 455                           expired list.  */

377. 456

378. 457                        /* See if this is the only suspended thread on the list.
     */

379. 458                        if (thread_ptr == thread_ptr ->
     tx_thread_suspended_next) // 只有一个等待事件的线程，没有其他线程了，清空
     suspended_list

380. 459                        {

381. 460
```

```
382. 461                            /* Yes, the only suspended thread.  */
383. 462
384. 463                            /* Update the head pointer.  */
385. 464                            suspended_list =  TX_NULL;
386. 465                        }
387. 466                        else // 有其他线程也等待事件(从suspended_list删除
     thread_ptr，如果thread_ptr是表头还得更新表头)
388. 467                        {
389. 468
390. 469                            /* At least one more thread is on the same
     expiration list.  */
391. 470
392. 471                            /* Update the links of the adjacent threads.  */
393. 472                            next_thread =
     thread_ptr -> tx_thread_suspended_next;
394. 473                            previous_thread =
     thread_ptr -> tx_thread_suspended_previous;
395. 474                            next_thread -> tx_thread_suspended_previous =
     previous_thread;
396. 475                            previous_thread -> tx_thread_suspended_next =
     next_thread;
397. 476
398. 477                            /* Update the list head pointer, if removing the
     head of the
399. 478                               list.  */
400. 479                            if (suspended_list == thread_ptr)
401. 480                            {
402. 481
403. 482                                /* Yes, head pointer needs to be updated.  */
404. 483                                suspended_list =  thread_ptr ->
     tx_thread_suspended_next;
405. 484                            }
406. 485                        }
407. 486
408. 487                        /* Decrement the suspension count.  */
```

```
409. 488                        group_ptr -> tx_event_flags_group_suspended_count--; //
     等待事件的线程数减1

410. 489

411. 490                        /* Place this thread on the expired list.  */

412. 491                        if (satisfied_list == TX_NULL) // 满足事件的线程链表为
     空，thread_ptr加入该链表(到这里，thread_ptr还没被唤醒，后面检查完所有线程后统一对获
     取到事件的线程进行唤醒)

413. 492                        {

414. 493

415. 494                            /* First thread on the satisfied list.  */

416. 495                            satisfied_list =  thread_ptr;

417. 496                            last_satisfied =  thread_ptr;

418. 497

419. 498                            /* Setup initial next pointer.  */

420. 499                            thread_ptr -> tx_thread_suspended_next =  TX_NULL;

421. 500                        }

422. 501                        else // 添加thread_ptr到satisfied_list末尾
     (last_satisfied指向satisfied_list链表末尾)

423. 502                        {

424. 503

425. 504                            /* Not the first thread on the satisfied list.  */

426. 505

427. 506                            /* Link it up at the end.  */

428. 507                            last_satisfied -> tx_thread_suspended_next =
     thread_ptr;

429. 508                            thread_ptr -> tx_thread_suspended_next =
     TX_NULL;

430. 509                            last_satisfied =
     thread_ptr;

431. 510                        }

432. 511                    }

433. 512

434. 513                    /* Copy next thread pointer to working thread ptr.  */

435. 514                    thread_ptr =  next_thread_ptr; // 获取下一个阻塞线程

436. 515
```

```
437. 516                    /* Decrement the suspension count.  */

438. 517                    suspended_count--; // suspended_count个数减1

439. 518

440. 519                } while (suspended_count != TX_NO_SUSPENSIONS); // 这里用
     suspended_count表示suspended_list的个数，所以前面的更新suspended_count必须保证
     suspended_count等于suspended_list的个数!!!

441. 520

442. 521                /* Setup the group's suspension list head again.  */

443. 522                group_ptr -> tx_event_flags_group_suspension_list =
     suspended_list; // 把没有获取到事件的线程重新挂载到
     tx_event_flags_group_suspension_list链表(禁止抢占期间ISR不会操作
     tx_event_flags_group_suspension_list，tx_event_flags_group_suspension_list没有阻塞线
     程；获取事件的函数没有检查是否在中断上下文，但是从代码上看，ISR程序就不能调用获取事
     件操作，否则ISR会被阻塞!!!)

444. 523

445. 524 #ifndef TX_NOT_INTERRUPTABLE

446. 525

447. 526                /* Determine if there is any delayed event clearing to perform.
     */

448. 527                if (group_ptr -> tx_event_flags_group_delayed_clear != ((ULONG)
     0))

449. 528                {

450. 529

451. 530                    /* Perform the delayed event clearing.  */

452. 531                    group_ptr -> tx_event_flags_group_current =

453. 532                        group_ptr -> tx_event_flags_group_current & ~(group_ptr
     -> tx_event_flags_group_delayed_clear);

454. 533

455. 534                    /* Clear the delayed event flag clear value.  */

456. 535                    group_ptr -> tx_event_flags_group_delayed_clear =  ((ULONG)
     0);

457. 536                }

458. 537 #endif

459. 538

460. 539                /* Restore interrupts.  */

461. 540                TX_RESTORE

462. 541
```

```
542                 /* Walk through the satisfied list, setup initial thread
    pointer. */
543                 thread_ptr =  satisfied_list; // 获取到事件的线程链表
544                 while(thread_ptr != TX_NULL) // 逐个唤醒获取到事件的线程(前面的
    禁止抢占还没取消，唤醒过程不会被抢占)
545                 {
546
547                     /* Get next pointer first.  */
548                     next_thread_ptr =  thread_ptr -> tx_thread_suspended_next;
549
550                     /* Disable interrupts.  */
551                     TX_DISABLE
552
553 #ifdef TX_NOT_INTERRUPTABLE
554
555                     /* Resume the thread!  */
556                     _tx_thread_system_ni_resume(thread_ptr);
557
558                     /* Restore interrupts.  */
559                     TX_RESTORE
560 #else
561
562                     /* Disable preemption again.  */
563                     _tx_thread_preempt_disable++;
564
565                     /* Restore interrupt posture.  */
566                     TX_RESTORE
567
568                     /* Resume the thread.  */
569                     _tx_thread_system_resume(thread_ptr); // 唤醒获取到事件的线
    程
570 #endif
571
```

```
493. 572                    /* Move next thread to current.  */

494. 573                    thread_ptr =  next_thread_ptr;

495. 574                }

496. 575

497. 576                /* Disable interrupts.  */

498. 577                TX_DISABLE

499. 578

500. 579                /* Release thread preemption disable.  */

501. 580                _tx_thread_preempt_disable--; // 取消禁止抢占(唤醒等待事件的线程及禁止抢占期间，可能有更高优先级就绪线程就绪了)

502. 581            }

503. 582        }

504. 583        else // 等待事件线程链表为空(可能被其他设置事件的线程设置为空，其他线程在处理等待链表，检查tx_event_flags_group_suspended_count才能真正确定是否有线程等待事件)

505. 584        {

506. 585

507. 586            /* Determine if we need to set the reset search field.  */

508. 587            if (group_ptr -> tx_event_flags_group_suspended_count != TX_NO_SUSPENSIONS) // 当前线程设置了事件，但是没有检查等待事件线程链表(别的线程在处理)，那么要设置tx_event_flags_group_reset_search，有新的事件，处理等待线程链表的线程需要再次检查一遍

509. 588            {

510. 589

511. 590                /* We interrupted a search of an event flag group suspension

512. 591                   list.  Make sure we reset the search.  */

513. 592                group_ptr -> tx_event_flags_group_reset_search =  TX_TRUE; // 设置tx_event_flags_group_reset_search，处理suspended_list的线程需要更新事件，重新检查是否有事件满足等待事件的线程

514. 593            }

515. 594        }

516. 595

517. 596        /* Restore interrupts.  */

518. 597        TX_RESTORE

519. 598

520. 599 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
```

```
600
601         /* Determine if a notify callback is required.  */
602         if (events_set_notify != TX_NULL)
603         {
604
605             /* Call application event flags set notification.  */
606             (events_set_notify)(group_ptr);
607         }
608 #endif
609
610         /* Determine if a check for preemption is necessary.  */
611         if (preempt_check == TX_TRUE) // 是否要检查抢占(这个抢占检查有点问题，上面有线程获取到事件的时候设置preempt_check为TX_TRUE，如果整个过程是关中断的，那么这里没有问题，但是TX_NOT_INTERRUPTABLE没有定义的情况下，很多地方是允许中断的，只是禁止了抢占，中断服务程序可能唤醒更高优先级线程)
612         {
613
614             /* Yes, one or more threads were resumed, check for preemption.  */
615             _tx_thread_system_preempt_check();
616         }
617     }
618
619     /* Return completion status.  */
620     return(TX_SUCCESS);
621 }
```