(163条消息) ThreadX内核源码分析 - 消息队列_arm7star的博客-CSDN博客_tx_queue_receive

1、消息队列介绍

ThreadX内核的消息可以多线程收发，每个消息的大小固定；消息队列有一定的大小，超过大小之后，发送消息的线程需要等待消息被取走才能往消息队列里面再次发送消息。

2、消息的接收_tx_queue_receive

消息接收主要检查有没有消息，没有消息就要等待消息或者返回消息队列为空的错误码；

如果消息队列不为空，并且没有发送消息的线程等待消息队列(消息队列不为空)，那么直接从消息队列最前面读消息即可；

如果消息队列满了，有线程等待消息队列，那么检查等待消息队列的第一个线程是不是要将消息发送到消息队列最前面(一般消息都是追加的，但是ThreadX内核支持插入消息的最前面)，如果是插入到消息队列最前面，那么就从发送消息的线程直接读取消息并唤醒发送消息的线程，否则还得从消息队列最前面开始读消息。

读完一个消息，就可以让一个等待消息队列的线程发送消息，有读消息的函数直接将发送消息的线程的消息数据拷贝到消息队列，然后唤醒该发送消息的线程。

很多内核通用重复代码及原理前面文章都有介绍，在此仅介绍不一样的部分关键代码，详细看代码里面的注释，对着代码行及说明看代码更容易理解。

_tx_queue_receive实现代码如下：

```
082 UINT  _tx_queue_receive(TX_QUEUE *queue_ptr, VOID *destination_ptr, ULONG
    wait_option)
083 {
084
085 TX_INTERRUPT_SAVE_AREA
086
087 TX_THREAD       *thread_ptr;
088 ULONG           *source;
089 ULONG           *destination;
090 UINT            size;
091 UINT            suspended_count;
092 TX_THREAD       *next_thread;
093 TX_THREAD       *previous_thread;
094 UINT            status;
095
096
097     /* Default the status to TX_SUCCESS.  */
098     status =  TX_SUCCESS;
099
100     /* Disable interrupts to receive message from queue.  */
101     TX_DISABLE
102
103 #ifdef TX_QUEUE_ENABLE_PERFORMANCE_INFO
104
105     /* Increment the total messages received counter.  */
106     _tx_queue_performance__messages_received_count++;
107
108     /* Increment the number of messages received from this queue.  */
109     queue_ptr -> tx_queue_performance_messages_received_count++;
110
111 #endif
112
```

```
32. 113     /* If trace is enabled, insert this event into the trace buffer.  */

33. 114     TX_TRACE_IN_LINE_INSERT(TX_TRACE_QUEUE_RECEIVE, queue_ptr,
    TX_POINTER_TO_ULONG_CONVERT(destination_ptr), wait_option, queue_ptr ->
    tx_queue_enqueued, TX_TRACE_QUEUE_EVENTS)

34. 115

35. 116     /* Log this kernel call.  */

36. 117     TX_EL_QUEUE_RECEIVE_INSERT

37. 118

38. 119     /* Pickup the thread suspension count.  */

39. 120     suspended_count =  queue_ptr -> tx_queue_suspended_count; // 等待消息队列线
    程数

40. 121

41. 122     /* Determine if there is anything in the queue.  */

42. 123     if (queue_ptr -> tx_queue_enqueued != TX_NO_MESSAGES) // 消息数不为0，有消息

43. 124     {

44. 125

45. 126         /* Determine if there are any suspensions.  */

46. 127         if (suspended_count == TX_NO_SUSPENSIONS) // 没有线程等待消息

47. 128         {

48. 129

49. 130             /* There is a message waiting in the queue and there are no
    suspensi.  */

50. 131

51. 132             /* Setup source and destination pointers.  */

52. 133             source =       queue_ptr -> tx_queue_read; // tx_queue_read消息数据
    源地址(消息队列的数据的地址)

53. 134             destination =  TX_VOID_TO_ULONG_POINTER_CONVERT(destination_ptr); //
    消息数据目的地址(本次读消息保存数据的地址)

54. 135             size =         queue_ptr -> tx_queue_message_size; // 每个消息的大
    小，多少个unsigned long大小(消息大小固定，只能按这么大小的消息收发，单位不是byte!!!)

55. 136

56. 137             /* Copy message. Note that the source and destination pointers are

57. 138                incremented by the macro.  */

58. 139             TX_QUEUE_MESSAGE_COPY(source, destination, size) // 拷贝size大小消息
    到destination，拷贝消息的时候，source、destination都在往后移动，拷贝完后，source指向
    下一个消息(因为消息按消息大小收发的，有消息的话，那么肯定有size大小的数据)
```

```
140
141             /* Determine if we are at the end.  */
142             if (source == queue_ptr -> tx_queue_end) // 当前消息已经到消息的末尾(消息发送到内存的末尾后，接下来的消息从消息内存地址开始的地址存储，一块连续的内存组成一个单向循环链表)
143             {
144
145                 /* Yes, wrap around to the beginning.  */
146                 source =  queue_ptr -> tx_queue_start; // 下一个消息从tx_queue_start开始
147             }
148
149             /* Setup the queue read pointer.   */
150             queue_ptr -> tx_queue_read =  source; // 更新tx_queue_read指向下一个消息
151
152             /* Increase the amount of available storage.  */
153             queue_ptr -> tx_queue_available_storage++; // 消息队列的容量加1(可以一个消息已经被读取，增加消息队列的容量，这里单位是消息个数，不是byte!!!)
154
155             /* Decrease the enqueued count.  */
156             queue_ptr -> tx_queue_enqueued--; // 消息队列里面的消息个数减1(不包含没有发送到消息队列里面的，可能消息队列已满，还有线程阻塞在发送过程中)
157
158             /* Restore interrupts.  */
159             TX_RESTORE
160         }
161         else // 有消息并且有消息等待消息队列(此时只可能是消息队列满了，有线程等待消息队列可以发送消息)
162         {
163
164             /* At this point we know the queue is full.  */
165
166             /* Pickup thread suspension list head pointer.  */
167             thread_ptr =  queue_ptr -> tx_queue_suspension_list; // 第一个发送消息的阻塞线程
```

```
 87. 168
 88. 169             /* Now determine if there is a queue front suspension active.   */
 89. 170
 90. 171             /* Is the front suspension flag set?  */
 91. 172             if (thread_ptr -> tx_thread_suspend_option == TX_TRUE) // 如果
     tx_thread_suspend_option为TX_TRUE，那么表明thread_ptr是要将消息发送到消息队列最前
     面，_tx_queue_front_send会设置tx_thread_suspend_option为TX_TRUE，那么直接从该线程读
     取消息即可
 92. 173             {
 93. 174
 94. 175                 /* Yes, a queue front suspension is present.  */
 95. 176
 96. 177                 /* Return the message associated with this suspension.  */
 97. 178
 98. 179                 /* Setup source and destination pointers.  */
 99. 180                 source =       TX_VOID_TO_ULONG_POINTER_CONVERT(thread_ptr ->
     tx_thread_additional_suspend_info); // 发送消息的线程的消息直接保存在
     tx_thread_additional_suspend_info里面(tx_thread_additional_suspend_info指向线程待发
     送的消息，因为消息队列不够，该消息还在线程里面，还没发送到消息队列)
100. 181                 destination =
     TX_VOID_TO_ULONG_POINTER_CONVERT(destination_ptr);
101. 182                 size =         queue_ptr -> tx_queue_message_size;
102. 183
103. 184                 /* Copy message. Note that the source and destination pointers
     are
104. 185                    incremented by the macro.  */
105. 186                 TX_QUEUE_MESSAGE_COPY(source, destination, size) // 拷贝消息数据
106. 187
107. 188                 /* Message is now in the caller's destination. See if this is
     the only suspended thread
108. 189                    on the list.  */
109. 190                 suspended_count--; // 等待发送消息的线程数减1
110. 191                 if (suspended_count == TX_NO_SUSPENSIONS) // 没有更多线程等待发
     送消息，那么tx_queue_suspension_list设为空即可(tx_queue_suspension_list目前都是发送
     阻塞的线程)
111. 192                 {
112. 193
```

```
194                     /* Yes, the only suspended thread.  */
195
196                     /* Update the head pointer.  */
197                     queue_ptr -> tx_queue_suspension_list =  TX_NULL;
198                 }
199                 else // 有其他线程阻塞在发送消息，挂在tx_queue_suspension_list链表上面，那么将已经取走消息的线程thread_ptr从链表删除即可
200                 {
201
202                     /* At least one more thread is on the same expiration list.  */
203
204                     /* Update the list head pointer.  */
205                     next_thread =                           thread_ptr -> tx_thread_suspended_next;
206                     queue_ptr -> tx_queue_suspension_list =  next_thread;
207
208                     /* Update the links of the adjacent threads.  */
209                     previous_thread =                           thread_ptr -> tx_thread_suspended_previous;
210                     next_thread -> tx_thread_suspended_previous = previous_thread;
211                     previous_thread -> tx_thread_suspended_next =  next_thread;
212                 }
213
214                 /* Decrement the suspension count.  */
215                 queue_ptr -> tx_queue_suspended_count =  suspended_count; // 更新等待消息队列的线程数
216
217                 /* Prepare for resumption of the first thread.  */
218
219                 /* Clear cleanup routine to avoid timeout.  */
220                 thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL; // tx_thread_suspend_cleanup设置为空
221
```

```
141. 222                /* Put return status into the thread control block.  */
142. 223                thread_ptr -> tx_thread_suspend_status =  TX_SUCCESS; // 消息被读取，状态设置为成功状态
143. 224
144. 225 #ifdef TX_NOT_INTERRUPTABLE
145. 226
146. 227                /* Resume the thread!  */
147. 228                _tx_thread_system_ni_resume(thread_ptr);
148. 229
149. 230                /* Restore interrupts.  */
150. 231                TX_RESTORE
151. 232 #else
152. 233
153. 234                /* Temporarily disable preemption.  */
154. 235                _tx_thread_preempt_disable++; // 禁止抢占
155. 236
156. 237                /* Restore interrupts.  */
157. 238                TX_RESTORE
158. 239
159. 240                /* Resume thread.  */
160. 241                _tx_thread_system_resume(thread_ptr); // 唤醒取走了消息的线程
161. 242 #endif
162. 243            }
163. 244            else // 阻塞线程thread_ptr不是要将数据发送到消息最前面(追加消息到已有消息的末尾)，那么还得从消息队列读消息
164. 245            {
165. 246
166. 247                /* At this point, we know that the queue is full and there
167. 248                   are one or more threads suspended trying to send another
168. 249                   message to this queue.  */
169. 250
170. 251                /* Setup source and destination pointers.  */
171. 252                source =       queue_ptr -> tx_queue_read;
```

```
172. 253                destination =
     TX_VOID_TO_ULONG_POINTER_CONVERT(destination_ptr);

173. 254                size =         queue_ptr -> tx_queue_message_size;

174. 255

175. 256                /* Copy message. Note that the source and destination pointers
     are

176. 257                   incremented by the macro.  */

177. 258                TX_QUEUE_MESSAGE_COPY(source, destination, size)

178. 259

179. 260                /* Determine if we are at the end.  */

180. 261                if (source == queue_ptr -> tx_queue_end)

181. 262                {

182. 263

183. 264                    /* Yes, wrap around to the beginning.  */

184. 265                    source =  queue_ptr -> tx_queue_start;

185. 266                }

186. 267

187. 268                /* Setup the queue read pointer.   */

188. 269                queue_ptr -> tx_queue_read =  source; // 更新tx_queue_read，这之
     前的几行读消息代码与之前代码一样，略过...

189. 270

190. 271                /* Disable preemption.  */

191. 272                _tx_thread_preempt_disable++; // 禁止抢占(消息处理花了一些时间，
     后面需要临时开一下中断，让中断得到处理)

192. 273

193. 274 #ifdef TX_NOT_INTERRUPTABLE

194. 275

195. 276                /* Restore interrupts.  */

196. 277                TX_RESTORE

197. 278

198. 279                /* Interrupts are enabled briefly here to keep the interrupt

199. 280                   lockout time deterministic.  */

200. 281

201. 282                /* Disable interrupts again.  */
```

```
202. 283                TX_DISABLE

203. 284 #endif

204. 285

205. 286                /* Decrement the preemption disable variable.  */

206. 287                _tx_thread_preempt_disable--; // 取消禁止抢占(中断已经关了，不需要禁止抢占)

207. 288

208. 289                /* Setup source and destination pointers.  */

209. 290                source =       TX_VOID_TO_ULONG_POINTER_CONVERT(thread_ptr -> tx_thread_additional_suspend_info); // 阻塞线程thread_ptr的消息地址

210. 291                destination =  queue_ptr -> tx_queue_write; // 消息队列写消息的地址(消息队列满的情况，这里就指向刚才已经被读取的消息地址)

211. 292                size =         queue_ptr -> tx_queue_message_size;

212. 293

213. 294                /* Copy message. Note that the source and destination pointers are

214. 295                   incremented by the macro.  */

215. 296                TX_QUEUE_MESSAGE_COPY(source, destination, size) // 拷贝消息到消息队列(末尾)，拷贝过程destination同样在更新，更新指向下一个消息地址(下一个消息可以写入的地址)

216. 297

217. 298                /* Determine if we are at the end.  */

218. 299                if (destination == queue_ptr -> tx_queue_end) // 写地址已经到内存地址的末尾，再从消息队列内存的起始地址开始

219. 300                {

220. 301

221. 302                    /* Yes, wrap around to the beginning.  */

222. 303                    destination =  queue_ptr -> tx_queue_start;

223. 304                }

224. 305

225. 306                /* Adjust the write pointer.  */

226. 307                queue_ptr -> tx_queue_write =  destination; // 更新写消息的地址tx_queue_write

227. 308

228. 309                /* Pickup thread pointer.  */
```

```
310                 thread_ptr =  queue_ptr -> tx_queue_suspension_list; // 前面已经读取了thread_ptr，这里为什么还要再次读取?
311
312                 /* Message is now in the queue.  See if this is the only suspended thread
313                    on the list.  */
314                 suspended_count--; // 阻塞线程数减1
315                 if (suspended_count == TX_NO_SUSPENSIONS) // 没有线程等待写队列，tx_queue_suspension_list清空即可
316                 {
317
318                   /* Yes, the only suspended thread.  */
319
320                     /* Update the head pointer.  */
321                     queue_ptr -> tx_queue_suspension_list =  TX_NULL;
322                 }
323                 else // 有其他线程等待写队列，将thread_ptr从等待队列中删除即可
324                 {
325
326                     /* At least one more thread is on the same expiration list.  */
327
328                     /* Update the list head pointer.  */
329                     next_thread =                           thread_ptr -> tx_thread_suspended_next;
330                     queue_ptr -> tx_queue_suspension_list =  next_thread;
331
332                     /* Update the links of the adjacent threads.  */
333                     previous_thread =                               thread_ptr -> tx_thread_suspended_previous;
334                     next_thread -> tx_thread_suspended_previous =  previous_thread;
335                     previous_thread -> tx_thread_suspended_next =   next_thread;
336                 }
337
```

```
338                 /* Decrement the suspension count.  */
339                 queue_ptr -> tx_queue_suspended_count =  suspended_count; // 更新tx_queue_suspended_count
340
341                 /* Prepare for resumption of the first thread.  */
342
343                 /* Clear cleanup routine to avoid timeout.  */
344                 thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL; // thread_ptr的消息已经放入消息队列了，清除tx_thread_suspend_cleanup
345
346                 /* Put return status into the thread control block.  */
347                 thread_ptr -> tx_thread_suspend_status =  TX_SUCCESS; // 设置tx_thread_suspend_status状态为成功，表示线程的消息已经发送成功
348
349 #ifdef TX_NOT_INTERRUPTABLE
350
351                 /* Resume the thread!  */
352                 _tx_thread_system_ni_resume(thread_ptr);
353
354                 /* Restore interrupts.  */
355                 TX_RESTORE
356 #else
357
358                 /* Temporarily disable preemption.  */
359                 _tx_thread_preempt_disable++;
360
361                 /* Restore interrupts.  */
362                 TX_RESTORE
363
364                 /* Resume thread.  */
365                 _tx_thread_system_resume(thread_ptr); // 唤醒线程(因为本次只读取一个消息，然后thread_ptr的消息写完后，消息队列又满了，所以一次只能有一个阻塞线程写消息成功)
366 #endif
```

```
367             }
368         }
369     }
370
371     /* Determine if the request specifies suspension.  */
372     else if (wait_option != TX_NO_WAIT) // 消息队列没有消息，并且有设置阻塞选项，需要阻塞等待有消息可读
373     {
374
375         /* Determine if the preempt disable flag is non-zero.  */
376         if (_tx_thread_preempt_disable != ((UINT) 0)) // 有禁止抢占，不能阻塞当前线程
377         {
378
379             /* Restore interrupts.  */
380             TX_RESTORE
381
382             /* Suspension is not allowed if the preempt disable flag is non-zero at this point - return error completion.  */
383             status =  TX_QUEUE_EMPTY; // 返回消息队列为空即可
384         }
385         else // 没有禁止抢占，需要等待消息
386         {
387
388             /* Prepare for suspension of this thread.  */
389
390 #ifdef TX_QUEUE_ENABLE_PERFORMANCE_INFO
391
392             /* Increment the total queue empty suspensions counter.  */
393             _tx_queue_performance_empty_suspension_count++;
394
395             /* Increment the number of empty suspensions on this queue.  */
396             queue_ptr -> tx_queue_performance_empty_suspension_count++;
```

```
316. 397 #endif

317. 398

318. 399             /* Pickup thread pointer.  */

319. 400             TX_THREAD_GET_CURRENT(thread_ptr) // 获取当前线程
     _tx_thread_current_ptr

320. 401

321. 402             /* Setup cleanup routine pointer.  */

322. 403             thread_ptr -> tx_thread_suspend_cleanup =  &(_tx_queue_cleanup); //
     等待消息超时或者线程终止时需要调用_tx_queue_cleanup唤醒或者清理当前线程(当前线程挂在
     阻塞链表里面，需要从阻塞链表删除)

323. 404

324. 405             /* Setup cleanup information, i.e. this queue control

325. 406                block and the source pointer.  */

326. 407             thread_ptr -> tx_thread_suspend_control_block =    (VOID *)
     queue_ptr; // 阻塞在消息队列queue_ptr上

327. 408             thread_ptr -> tx_thread_additional_suspend_info =  (VOID *)
     destination_ptr; // 消息接收地址(别的线程有发送消息，会将数据直接拷贝到
     destination_ptr里面)

328. 409             thread_ptr -> tx_thread_suspend_option =           TX_FALSE; // 读消
     息的时候，这个没起作用，都是从头读消息，不存在从末尾先读消息的情况

329. 410

330. 411 #ifndef TX_NOT_INTERRUPTABLE

331. 412

332. 413             /* Increment the suspension sequence number, which is used to
     identify

333. 414                this suspension event.  */

334. 415             thread_ptr -> tx_thread_suspension_sequence++;

335. 416 #endif

336. 417

337. 418             /* Setup suspension list.  */

338. 419             if (suspended_count == TX_NO_SUSPENSIONS) // 没有其他线程等待读消息
     (当前线程组成一个元素的阻塞链表)

339. 420             {

340. 421

341. 422                 /* No other threads are suspended.  Setup the head pointer and

342. 423                    just setup this threads pointers to itself.  */
```

```
343. 424                queue_ptr -> tx_queue_suspension_list =         thread_ptr;

344. 425                thread_ptr -> tx_thread_suspended_next =        thread_ptr;

345. 426                thread_ptr -> tx_thread_suspended_previous =    thread_ptr;

346. 427            }

347. 428            else // 有其他线程等待读消息，将当前线程添加到等待队列末尾即可

348. 429            {

349. 430

350. 431                /* This list is not NULL, add current thread to the end. */

351. 432                next_thread =                                   queue_ptr ->
     tx_queue_suspension_list;

352. 433                thread_ptr -> tx_thread_suspended_next =        next_thread;

353. 434                previous_thread =                               next_thread ->
     tx_thread_suspended_previous;

354. 435                thread_ptr -> tx_thread_suspended_previous =    previous_thread;

355. 436                previous_thread -> tx_thread_suspended_next =   thread_ptr;

356. 437                next_thread -> tx_thread_suspended_previous =   thread_ptr;

357. 438            }

358. 439

359. 440            /* Increment the suspended thread count.  */

360. 441            queue_ptr -> tx_queue_suspended_count =  suspended_count + ((UINT)
     1); // 挂起线程的数目加1

361. 442

362. 443            /* Set the state to suspended.  */

363. 444            thread_ptr -> tx_thread_state =    TX_QUEUE_SUSP; // 线程状态设置为
     等待消息队列挂起状态

364. 445

365. 446 #ifdef TX_NOT_INTERRUPTABLE

366. 447

367. 448            /* Call actual non-interruptable thread suspension routine.  */

368. 449            _tx_thread_system_ni_suspend(thread_ptr, wait_option);

369. 450

370. 451            /* Restore interrupts.  */

371. 452            TX_RESTORE

372. 453 #else
```

```
373. 454

374. 455             /* Set the suspending flag.  */

375. 456             thread_ptr -> tx_thread_suspending =  TX_TRUE; // 设置挂起中操作，线
     程还没真正挂起，还在就绪线程链表

376. 457

377. 458             /* Setup the timeout period.  */

378. 459             thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks =
     wait_option; // 等待选项(_tx_thread_system_suspend根据
     tx_timer_internal_remaining_ticks来启动超时定时器，如果是无限等待就不启动定时器)

379. 460

380. 461             /* Temporarily disable preemption.  */

381. 462             _tx_thread_preempt_disable++;

382. 463

383. 464             /* Restore interrupts.  */

384. 465             TX_RESTORE

385. 466

386. 467             /* Call actual thread suspension routine.  */

387. 468             _tx_thread_system_suspend(thread_ptr); // 挂起当前线程

388. 469 #endif

389. 470

390. 471             /* Return the completion status.  */

391. 472             status =  thread_ptr -> tx_thread_suspend_status; // 发送消息的线程
     把消息拷贝给当前线程会设置成功状态，等待超时会设置超时状态(与内存、信号量、互斥锁等
     操作一样...)

392. 473         }

393. 474     }

394. 475     else // 非阻塞，没有消息的时候返回消息队列为空即可

395. 476     {

396. 477

397. 478         /* Restore interrupts.  */

398. 479         TX_RESTORE

399. 480

400. 481         /* Immediate return, return error completion.  */

401. 482         status =  TX_QUEUE_EMPTY;
```

```
402. 483     }
403. 484
404. 485     /* Return completion status.  */
405. 486     return(status);
406. 487 }
```

3、消息的发送_tx_queue_send

发送消息过程与接收消息类型，消息队列没有满没有线程等待消息，那么将消息拷贝到消息队列即可；

如果有线程等待消息，那么消息队列就为空，当前消息就是第一个消息，拷贝消息到第一个等待消息的线程并唤醒该线程即可；

如果消息队列满了，如果设置了等待选项并允许阻塞的话，那么需要挂载到等待链表，发送消息与接收消息的线程共用一个等待链表tx_queue_suspension_list，只可能有发送消息的线程在等待或者接收消息的线程等待或者没有线程等待，不存在发送消息的线程和接收消息的线程都等待的清空。

_tx_queue_send是将消息加到消息队列末尾，实现代码比较简单，具体分析看代码中的注释。

_tx_queue_send实现代码如下：

```
080 UINT  _tx_queue_send(TX_QUEUE *queue_ptr, VOID *source_ptr, ULONG wait_option)
081 {
082
083 TX_INTERRUPT_SAVE_AREA
084
085 TX_THREAD       *thread_ptr;
086 ULONG           *source;
087 ULONG           *destination;
088 UINT            size;
089 UINT            suspended_count;
090 TX_THREAD       *next_thread;
091 TX_THREAD       *previous_thread;
092 UINT            status;
093 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
094 VOID            (*queue_send_notify)(struct TX_QUEUE_STRUCT *notify_queue_ptr);
095 #endif
096
097
098     /* Default the status to TX_SUCCESS.  */
099     status =  TX_SUCCESS;
100
101     /* Disable interrupts to place message in the queue.  */
102     TX_DISABLE
103
104 #ifdef TX_QUEUE_ENABLE_PERFORMANCE_INFO
105
106     /* Increment the total messages sent counter.  */
107     _tx_queue_performance_messages_sent_count++;
108
109     /* Increment the number of messages sent to this queue.  */
110     queue_ptr -> tx_queue_performance_messages_sent_count++;
```

```
111 #endif
112
113     /* If trace is enabled, insert this event into the trace buffer.  */
114     TX_TRACE_IN_LINE_INSERT(TX_TRACE_QUEUE_SEND, queue_ptr,
    TX_POINTER_TO_ULONG_CONVERT(source_ptr), wait_option, queue_ptr ->
    tx_queue_enqueued, TX_TRACE_QUEUE_EVENTS)
115
116     /* Log this kernel call.  */
117     TX_EL_QUEUE_SEND_INSERT
118
119     /* Pickup the thread suspension count.  */
120     suspended_count =  queue_ptr -> tx_queue_suspended_count; // 等待队列线程数
    (发送线程或者接收线程)
121
122     /* Determine if there is room in the queue.  */
123     if (queue_ptr -> tx_queue_available_storage != TX_NO_MESSAGES) //
    tx_queue_available_storage不为0，消息队列还可以接收tx_queue_available_storage个消息
124     {
125
126         /* There is room for the message in the queue.  */
127
128         /* Determine if there are suspended on this queue.  */
129         if (suspended_count == TX_NO_SUSPENSIONS) // 没有等待线程(消息队列可以接
    收数据，那么只等待队列只可能是接收消息的线程，没有等待消息的线程，那么就直接发送消息
    到消息队列即可)
130         {
131
132             /* No suspended threads, simply place the message in the queue.  */
133
134             /* Reduce the amount of available storage.  */
135             queue_ptr -> tx_queue_available_storage--; // 消息队列可接收消息的个
    数tx_queue_available_storage减1
136
137             /* Increase the enqueued count.  */
138             queue_ptr -> tx_queue_enqueued++; // 消息队列里面消息的个数
    tx_queue_enqueued加1
```

```
139
140             /* Setup source and destination pointers.  */
141             source =       TX_VOID_TO_ULONG_POINTER_CONVERT(source_ptr);
142             destination =  queue_ptr -> tx_queue_write;
143             size =         queue_ptr -> tx_queue_message_size;
144
145             /* Copy message. Note that the source and destination pointers are
146                incremented by the macro.  */
147             TX_QUEUE_MESSAGE_COPY(source, destination, size) // 拷贝消息到消息队列内存里面(destination更新到下一个消息)
148
149             /* Determine if we are at the end.  */
150             if (destination == queue_ptr -> tx_queue_end) // 下一个消息的地址已经到了消息队列内存的末尾，下一个消息地址要从消息队列内存的起始地址开始
151             {
152
153                 /* Yes, wrap around to the beginning.  */
154                 destination =  queue_ptr -> tx_queue_start;
155             }
156
157             /* Adjust the write pointer.  */
158             queue_ptr -> tx_queue_write =  destination; // 更新写地址tx_queue_write，下一个消息从tx_queue_write开始写
159
160 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
161
162             /* Pickup the notify callback routine for this queue.  */
163             queue_send_notify =  queue_ptr -> tx_queue_send_notify;
164 #endif
165
166             /* No thread suspended, just return to caller.  */
167
168             /* Restore interrupts.  */
```

```
169             TX_RESTORE
170
171 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
172
173             /* Determine if a notify callback is required.  */
174             if (queue_send_notify != TX_NULL)
175             {
176
177                 /* Call application queue send notification.  */
178                 (queue_send_notify)(queue_ptr);
179             }
180 #endif
181         }
182         else // 消息队列可接收消息个数不为0，有线程挂起，那么只可能是消息队列为空，有读消息的线程挂起，那么直接将当前发送的消息拷贝到读消息的线程即可，不需要先拷贝到消息队列，减少一次拷贝操作
183         {
184
185             /* There is a thread suspended on an empty queue. Simply
186                copy the message to the suspended thread's destination
187                pointer.  */
188
189             /* Pickup the head of the suspension list.  */
190             thread_ptr =  queue_ptr -> tx_queue_suspension_list; // 第一个读消息线程
191
192             /* See if this is the only suspended thread on the list.  */
193             suspended_count--; // 挂起线程个数减1
194             if (suspended_count == TX_NO_SUSPENSIONS) // 没有其他线程等待消息，那么等待队列tx_queue_suspension_list设置为空即可
195             {
196
197                 /* Yes, the only suspended thread.  */
198
```

```
120. 199                /* Update the head pointer.  */
121. 200                queue_ptr -> tx_queue_suspension_list =  TX_NULL;
122. 201            }
123. 202            else // 有其他线程等待消息，将thread_ptr从等待链表删除(发送的消息将直接拷贝给thread_ptr线程)
124. 203            {
125. 204
126. 205                /* At least one more thread is on the same expiration list.  */
127. 206
128. 207                /* Update the list head pointer.  */
129. 208                queue_ptr -> tx_queue_suspension_list =  thread_ptr -> tx_thread_suspended_next;
130. 209
131. 210                /* Update the links of the adjacent threads.  */
132. 211                next_thread =                           thread_ptr -> tx_thread_suspended_next;
133. 212                queue_ptr -> tx_queue_suspension_list =  next_thread;
134. 213
135. 214                /* Update the links of the adjacent threads.  */
136. 215                previous_thread =                           thread_ptr -> tx_thread_suspended_previous;
137. 216                next_thread -> tx_thread_suspended_previous =   previous_thread;
138. 217                previous_thread -> tx_thread_suspended_next =   next_thread;
139. 218            }
140. 219
141. 220            /* Decrement the suspension count.  */
142. 221            queue_ptr -> tx_queue_suspended_count =  suspended_count; // 更新等待线程个数(前面减1了，减掉了thread_ptr)
143. 222
144. 223            /* Prepare for resumption of the thread.  */
145. 224
146. 225            /* Clear cleanup routine to avoid timeout.  */
147. 226            thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL; // thread_ptr即将获取到消息，清空tx_thread_suspend_cleanup
148. 227
```

```
149. 228            /* Setup source and destination pointers.  */
150. 229            source =       TX_VOID_TO_ULONG_POINTER_CONVERT(source_ptr); // 发送消息的消息地址
151. 230            destination =  TX_VOID_TO_ULONG_POINTER_CONVERT(thread_ptr -> tx_thread_additional_suspend_info); // thread_ptr接收消息的地址
152. 231            size =         queue_ptr -> tx_queue_message_size;
153. 232
154. 233            /* Copy message. Note that the source and destination pointers are
155. 234               incremented by the macro.  */
156. 235            TX_QUEUE_MESSAGE_COPY(source, destination, size) // 直接将发送到消息拷贝到接收消息的线程
157. 236
158. 237            /* Put return status into the thread control block.  */
159. 238            thread_ptr -> tx_thread_suspend_status =  TX_SUCCESS; // 设置接收消息的线程的状态tx_thread_suspend_status为成功
160. 239
161. 240 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
162. 241
163. 242            /* Pickup the notify callback routine for this queue.  */
164. 243            queue_send_notify =  queue_ptr -> tx_queue_send_notify;
165. 244 #endif
166. 245
167. 246 #ifdef TX_NOT_INTERRUPTABLE
168. 247
169. 248            /* Resume the thread!  */
170. 249            _tx_thread_system_ni_resume(thread_ptr);
171. 250
172. 251            /* Restore interrupts.  */
173. 252            TX_RESTORE
174. 253 #else
175. 254
176. 255            /* Temporarily disable preemption.  */
177. 256            _tx_thread_preempt_disable++;
178. 257
```

```
179. 258            /* Restore interrupts.  */
180. 259            TX_RESTORE
181. 260
182. 261            /* Resume thread.  */
183. 262            _tx_thread_system_resume(thread_ptr); // 唤醒获取到消息的线程 thread_ptr
184. 263 #endif
185. 264
186. 265 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
187. 266
188. 267            /* Determine if a notify callback is required.  */
189. 268            if (queue_send_notify != TX_NULL)
190. 269            {
191. 270
192. 271                /* Call application queue send notification.  */
193. 272                (queue_send_notify)(queue_ptr);
194. 273            }
195. 274 #endif
196. 275        }
197. 276    }
198. 277
199. 278    /* At this point, the queue is full. Determine if suspension is requested. */
200. 279    else if (wait_option != TX_NO_WAIT) // 消息队列满了，等待选项不是不等待，那么需要阻塞当前线程
201. 280    {
202. 281
203. 282        /* Determine if the preempt disable flag is non-zero.  */
204. 283        if (_tx_thread_preempt_disable != ((UINT) 0)) // 禁止了抢占，那么不能阻塞调度，不能挂起当前线程，返回队列满了即可
205. 284        {
206. 285
207. 286            /* Restore interrupts.  */
208. 287            TX_RESTORE
```

```
288
289             /* Suspension is not allowed if the preempt disable flag is non-zero
    at this point - return error completion.  */
290             status =  TX_QUEUE_FULL; // 消息队列满了
291         }
292         else // 没有禁止抢占，可以阻塞当前线程
293         {
294
295             /* Yes, prepare for suspension of this thread.  */
296
297 #ifdef TX_QUEUE_ENABLE_PERFORMANCE_INFO
298
299             /* Increment the total number of queue full suspensions.  */
300             _tx_queue_performance_full_suspension_count++;
301
302             /* Increment the number of full suspensions on this queue.  */
303             queue_ptr -> tx_queue_performance_full_suspension_count++;
304 #endif
305
306             /* Pickup thread pointer.  */
307             TX_THREAD_GET_CURRENT(thread_ptr)
308
309             /* Setup cleanup routine pointer.  */
310             thread_ptr -> tx_thread_suspend_cleanup =  &(_tx_queue_cleanup);
311
312             /* Setup cleanup information, i.e. this queue control
313                block and the source pointer.  */
314             thread_ptr -> tx_thread_suspend_control_block =    (VOID *)
    queue_ptr; // 消息队列
315             thread_ptr -> tx_thread_additional_suspend_info =  (VOID *)
    source_ptr; // 消息的地址
316             thread_ptr -> tx_thread_suspend_option =           TX_FALSE; // 正常
    发送消息，这个设置为TX_FALSE，是发送消息到消息队列末尾，不是插入到消息队列最前面
317
```

```
239. 318 #ifndef TX_NOT_INTERRUPTABLE

240. 319

241. 320             /* Increment the suspension sequence number, which is used to
     identify

242. 321                this suspension event.  */

243. 322             thread_ptr -> tx_thread_suspension_sequence++;

244. 323 #endif

245. 324

246. 325             /* Setup suspension list.  */

247. 326             if (suspended_count == TX_NO_SUSPENSIONS) // if...else...当前线程插
     入消息等待队列

248. 327             {

249. 328

250. 329                 /* No other threads are suspended.  Setup the head pointer and

251. 330                    just setup this threads pointers to itself.  */

252. 331                 queue_ptr -> tx_queue_suspension_list =         thread_ptr;

253. 332                 thread_ptr -> tx_thread_suspended_next =        thread_ptr;

254. 333                 thread_ptr -> tx_thread_suspended_previous =    thread_ptr;

255. 334             }

256. 335             else

257. 336             {

258. 337

259. 338                 /* This list is not NULL, add current thread to the end. */

260. 339                 next_thread =                                   queue_ptr ->
     tx_queue_suspension_list;

261. 340                 thread_ptr -> tx_thread_suspended_next =        next_thread;

262. 341                 previous_thread =                               next_thread ->
     tx_thread_suspended_previous;

263. 342                 thread_ptr -> tx_thread_suspended_previous =    previous_thread;

264. 343                 previous_thread -> tx_thread_suspended_next =   thread_ptr;

265. 344                 next_thread -> tx_thread_suspended_previous =   thread_ptr;

266. 345             }

267. 346

268. 347             /* Increment the suspended thread count.  */
```

```
348             queue_ptr -> tx_queue_suspended_count =  suspended_count + ((UINT) 1); // 等待线程加1(等待消息队列的线程个数)
349
350             /* Set the state to suspended.  */
351             thread_ptr -> tx_thread_state =    TX_QUEUE_SUSP; // 线程状态设置为等待消息队列
352
353 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
354
355             /* Pickup the notify callback routine for this queue.  */
356             queue_send_notify =  queue_ptr -> tx_queue_send_notify;
357 #endif
358
359 #ifdef TX_NOT_INTERRUPTABLE
360
361             /* Call actual non-interruptable thread suspension routine.  */
362             _tx_thread_system_ni_suspend(thread_ptr, wait_option);
363
364             /* Restore interrupts.  */
365             TX_RESTORE
366 #else
367
368             /* Set the suspending flag.  */
369             thread_ptr -> tx_thread_suspending =  TX_TRUE; // 线程挂起中
370
371             /* Setup the timeout period.  */
372             thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks = wait_option; // 等待选项(等待时间或者无限等待)
373
374             /* Temporarily disable preemption.  */
375             _tx_thread_preempt_disable++;
376
377             /* Restore interrupts.  */
```

```
299. 378             TX_RESTORE
300. 379
301. 380             /* Call actual thread suspension routine.  */
302. 381             _tx_thread_system_suspend(thread_ptr); // 挂起当前线程
303. 382 #endif
304. 383
305. 384 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
306. 385
307. 386             /* Determine if a notify callback is required.  */
308. 387             if (thread_ptr -> tx_thread_suspend_status == TX_SUCCESS)
309. 388             {
310. 389
311. 390                 /* Determine if there is a notify callback.  */
312. 391                 if (queue_send_notify != TX_NULL)
313. 392                 {
314. 393
315. 394                     /* Call application queue send notification.  */
316. 395                     (queue_send_notify)(queue_ptr);
317. 396                 }
318. 397             }
319. 398 #endif
320. 399
321. 400             /* Return the completion status.  */
322. 401             status =  thread_ptr -> tx_thread_suspend_status;
323. 402         }
324. 403     }
325. 404     else // 消息队列满了，不等待消息队列，返回消息队列满了即可
326. 405     {
327. 406
328. 407         /* Otherwise, just return a queue full error message to the caller.  */
329. 408
330. 409 #ifdef TX_QUEUE_ENABLE_PERFORMANCE_INFO
```

```
410
411         /* Increment the number of full non-suspensions on this queue.  */
412         queue_ptr -> tx_queue_performance_full_error_count++;
413
414         /* Increment the total number of full non-suspensions.  */
415         _tx_queue_performance_full_error_count++;
416 #endif
417
418         /* Restore interrupts.  */
419         TX_RESTORE
420
421         /* Return error completion.  */
422         status =  TX_QUEUE_FULL; // 消息队列满了
423     }
424
425     /* Return completion status.  */
426     return(status);
427 }
```

_tx_queue_front_send将消息发送到消息队列最前面，这个实现也比较简单，挂起线程时，是将线程插入队列表头，在此略过，通用技术可以参考比较早的文章，核心代码比较多重复的，不再重复介绍。