**LiteOS内核函数解析**

[一、 任务相关函数 3](#_Toc9873)

[1.1 基本任务函数 3](#_Toc19858)

[1.1.1 数据结构解析 3](#_Toc5263)

[1.1.2 osTaskInit( )函数 5](#_Toc11659)

[1.1.3 LOS\_TaskCreate( )函数 7](#_Toc29428)

[1.1.4 LOS\_TaskCreateOnly( )函数 8](#_Toc8974)

[1.1.5 LOS\_TaskDelete( )函数 10](#_Toc15724)

[1.1.6 LOS\_TaskSuspend( )函数 12](#_Toc11073)

[1.1.7 LOS\_TaskResume( )函数 13](#_Toc13345)

[1.1.8 LOS\_TaskLock( )和LOS\_TaskUnlock( )函数 14](#_Toc18135)

[1.1.9 LOS\_TaskPriSet( )和LOS\_TaskPriGet( )函数 15](#_Toc22886)

[1.1.10 osTaskPriModify( )函数 16](#_Toc2454)

[1.1.11 osTskStackInit( )函数 16](#_Toc10657)

[1.1.12 osTaskEntry( )函数 18](#_Toc16157)

[1.1.13 osTaskSelfDelete( )函数 20](#_Toc18302)

[1.2 任务延时相关函数 21](#_Toc5840)

[1.2.1 LOS\_TaskDelay( )函数 21](#_Toc26285)

[1.2.2 osTaskAdd2TimerList( )函数 22](#_Toc6154)

[1.2.3 osTimerListDelete( )函数 25](#_Toc20933)

[1.3 系统调度相关函数 25](#_Toc26894)

[1.3.1 系统tick相关函数 25](#_Toc3067)

[1.3.2 时间片的检测及轮转调度 27](#_Toc1609)

[1.3.3 cortex-m4寄存器组简单说明 29](#_Toc21966)

[1.3.4 PEND\_SV中断说明 31](#_Toc13542)

[1.3.5 系统调度的启动：LOS\_StartToRun( )函数 33](#_Toc21464)

[1.3.6 系统调度的实现：LOS\_Schedule( )和osTaskSchedule( )函数 35](#_Toc4805)

[1.3.7 排序队列的扫描及调度：osTaskScan( )函数 41](#_Toc20835)

[二、 内存池管理 44](#_Toc30540)

[2.1 内存池管理概述 44](#_Toc9335)

[2.2 静态内存池 44](#_Toc13235)

[2.2.1 宏定义和结构体解析 44](#_Toc10103)

[2.2.2 LOS\_MemboxInit( )函数 45](#_Toc30389)

[2.2.3 LOS\_MemboxAlloc( )函数 46](#_Toc7305)

[2.2.4 LOS\_MemboxFree( )函数 47](#_Toc30418)

[2.2.5 osCheckBoxMem( )函数 48](#_Toc12636)

[2.2.6 LOS\_MemboxClr( )函数 48](#_Toc24445)

[2.3 动态内存池 49](#_Toc19981)

[2.3.1 相关数据结构解析 49](#_Toc8433)

[2.3.2 LOS\_MemInit( )函数 50](#_Toc28726)

[2.3.3 LOS\_MemAlloc( )函数 52](#_Toc32646)

[2.3.4 osMemAllocWithCheck( )函数 53](#_Toc7018)

[2.3.5 osMemFindSuitableFreeBlock( )函数 54](#_Toc26291)

[2.3.6 osMemSpitNode( )函数 55](#_Toc29360)

[2.3.7 osMemMergeNode( )函数 55](#_Toc25891)

[2.3.8 LOS\_MemAllocAlign( )函数 56](#_Toc27323)

[2.3.9 LOS\_MemFree( )函数 57](#_Toc8847)

[2.3.10 osMemCheckUsedNode( )函数 57](#_Toc17133)

[2.3.11 osMemIsNodeValid( )函数 58](#_Toc25908)

[2.3.12 osMemFreeNode( )函数 59](#_Toc9066)

[三、 IPC通信(消息队列和事件) 60](#_Toc11290)

[3.1 消息队列 60](#_Toc28866)

[3.1.1 QUEUE\_CB\_S结构解析 60](#_Toc19602)

[3.1.2 osQueueInit( )函数 61](#_Toc10752)

[3.1.3 LOS\_QueueCreate( )函数 61](#_Toc13449)

[3.1.4 osQueueCreate( )函数 62](#_Toc3106)

[3.1.5 LOS\_QueueDelete( )函数 64](#_Toc16557)

[3.1.6 osQueuePend( )函数 65](#_Toc20027)

[3.1.7 osQueueWakeUp( )函数 65](#_Toc19569)

[3.1.8 LOS\_QueueRead( )函数 66](#_Toc9177)

[3.1.9 LOS\_QueueWrite( )函数 69](#_Toc4979)

[3.1.10 osQueueMailAlloc( )函数 71](#_Toc24418)

[3.1.11 osQueueMailFree( )函数 73](#_Toc7293)

[3.2 事件 74](#_Toc3700)

[3.2.1 EVENT\_CB\_S结构解析 74](#_Toc23253)

[3.2.2 LOS\_EventInit( )函数 74](#_Toc27988)

[3.2.3 LOS\_EventWrite( )函数 74](#_Toc29602)

[3.2.4 LOS\_EventRead( )函数 75](#_Toc31044)

[3.2.5 LOS\_EventPoll( )函数 78](#_Toc26856)

[3.2.6 LOS\_EventClear( )函数 78](#_Toc31184)

[3.2.7 LOS\_EventDestory( )函数 79](#_Toc17230)

[四、 IPC之任务同步(互斥锁和信号量) 79](#_Toc24353)

[4.1 优先级翻转问题概述 79](#_Toc7153)

[4.2 互斥锁 80](#_Toc2322)

[4.2.1 MUX\_CB\_S结构解析 80](#_Toc6573)

[4.2.2 osMuxInit( )函数 81](#_Toc22486)

[4.2.3 LOS\_MuxCreate( )函数 81](#_Toc9632)

[4.2.4 LOS\_MuxDelete( )函数 82](#_Toc23262)

[4.2.5 LOS\_MuxPend( )函数 83](#_Toc30981)

[4.2.6 LOS\_MuxPost( )函数 86](#_Toc22495)

[4.3 信号量 87](#_Toc30351)

[4.3.1 SEM\_CB\_S结构解析 87](#_Toc29488)

[4.3.2 osSemInit( )函数 88](#_Toc5099)

[4.3.3 LOS\_SemCreate( )函数 88](#_Toc19406)

[4.3.4 LOS\_SemDelete( )函数 89](#_Toc31770)

[4.3.5 LOS\_SemPend( )函数 90](#_Toc12488)

[4.3.6 LOS\_SemPost( )函数 92](#_Toc6584)

[五、 软件定时器 94](#_Toc750)

[5.1 软件定时器任务函数 94](#_Toc28785)

[5.1.1 SWTMR\_CTRL\_S结构解析 94](#_Toc11494)

[5.1.2 osSwTmrInit( )函数 95](#_Toc10039)

[5.1.3 osSwTmrTaskCreate( )函数 96](#_Toc11557)

[5.1.4 osSwTmrTask( )函数 96](#_Toc28907)

[5.2 软件定时器基本函数 97](#_Toc18515)

[5.2.1 LOS\_SwtmrCreate( )函数 97](#_Toc12192)

[5.2.2 LOS\_SwtmrStart( )函数 98](#_Toc28703)

[5.2.3 LOS\_SwtmrStop( )函数 99](#_Toc20126)

[5.2.4 LOS\_SwtmrDelete( )函数 100](#_Toc2464)

[5.2.5 osSwTmrStart( )函数 101](#_Toc29130)

[5.2.6 osSwtmrStop( )函数 102](#_Toc22440)

[5.2.7 osSwtmrDelete( )函数 102](#_Toc19756)

[5.3 软件定时器扫描及处理 103](#_Toc12466)

[5.3.1 osSwtmrScan( )函数 103](#_Toc19868)

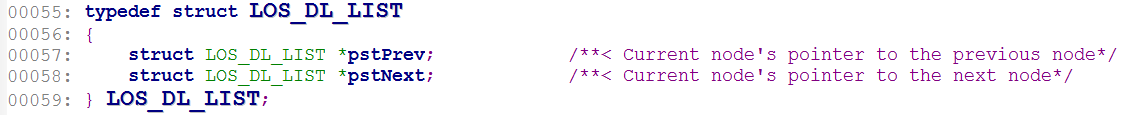
[5.3.2 osSwTmrTimeoutHandle( )函数 103](#_Toc9562)

# 任务相关函数

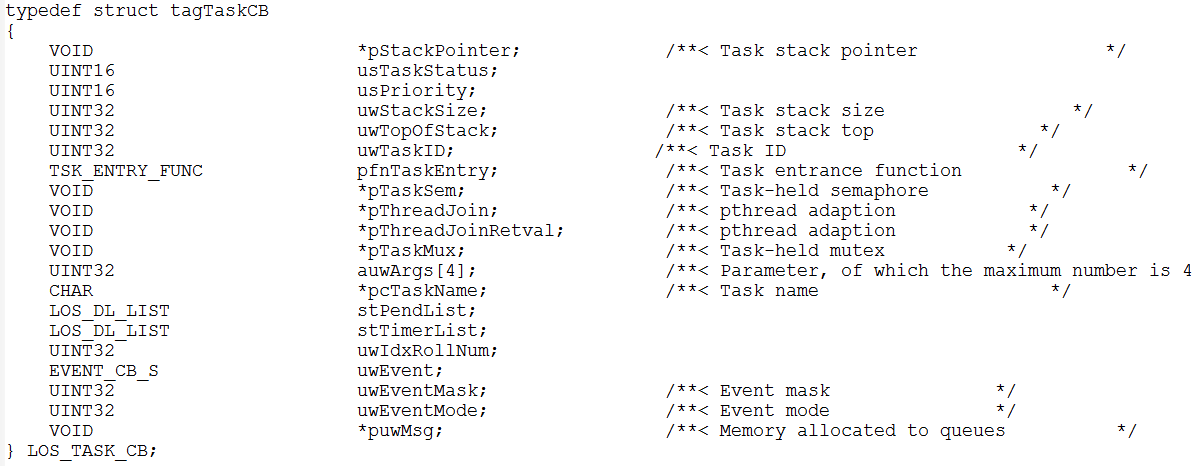
## 1.1 基本任务函数

### 1.1.1 数据结构解析

链接指针，双链表形式，用于将特定的数据结构链接在特定的链表中。



TASK的任务控制块TCB：



\*pStackPointer：TASK当前的栈指针，在任务调出时用于保存PSP，任务调入时由保存下来的PSP恢复现场。

usTaskStatus：TASK的状态，共14种状态，系统使用了其中的9种状态，分别如下：

OS\_TASK\_STATUS\_UNUSED：值为0x0001，未使用状态，表示TCB目前处于空闲队列中，未被分配出去。

OS\_TASK\_STATUS\_SUSPEND：值为0x0002，挂起状态，TASK不再参与系统调度，处于SUSPEND状态的TASK一定不在优先级队列中，但有可能存在于其它阻塞队列中，故在其它阻塞队列中需要对SUSPEND状态做出判断。

OS\_TASK\_STATUS\_READY：值为0x0004，就绪状态，TASK位于优先级队列参与系统调度。

OS\_TASK\_STATUS\_PEND：值为0x0008，阻塞状态，TASK位于对应的阻塞队列中等待唤醒。

OS\_TASK\_STATUS\_RUNNING：值为0x0010，运行状态，TASK正在被CPU调度运行。

OS\_TASK\_STATUS\_DELAY：值为0x0020，延时状态，TASK延时等待，位于超时排序队列中等待超时唤醒。

OS\_TASK\_STATUS\_TIMEOUT：值为0x0040，超时状态，TASK被设置了阻塞超时，位于超时排序队列中等待超时唤醒。

OS\_TASK\_STATUS\_DETACHED：值为0x0080，属性分离状态，设置该状态后，TASK在其工作函数退出后需执行自删除过程来删除自己。未设置该状态时，当TASK工作函数退出后，需要等待其父TASK回收该TASK。

OS\_TASK\_STATUS\_PEND\_QUEUE：值为0x2000，队列阻塞状态，TASK被阻塞在消息队列上等待唤醒。

usPriority：TASK的优先级，最高为0，最低为31，共32个优先级。

uwStackSize：TASK的栈大小，按8字节对齐。

uwTopOfStack：TASK的栈顶指针，永远指示栈顶位置。

uwTaskID：TASK的ID号，标识唯一的TASK。

pfnTaskEntry：TASK的工作函数，由TASK的入口函数调用执行。

\*pTaskSem：指向信号量控制块，表示TASK正在等待获取某一信号量，在TASK被阻塞至信号量阻塞链表时设置，被唤醒后清除。

\*pThreadJoin：指示父TASK。

\*pThreadJoinRetval：工作函数的返回值。

\*pTaskMux：指向互斥锁控制块，表示TASK正在等待获取某一互斥锁，在TASK被阻塞至互斥锁阻塞链表时设置，被唤醒后清除。

auwArgs[4]：TASK工作函数的参数。

\*pcTaskName：指向TASK的名称。

stPendList：TASK的指示链表，用于将该TASK链接在除了排序队列以外的其它队列上。

stTimerList：TASK的指示链表，用于将该TASK链接在排序队列上。

备注：TASK有可能同时处于阻塞队列和排序队列上，所以需要两个链接指针指示该TASK同时被加入了两个队列上。

uwIdxRollNum：TASK在排序队列中的超时时间控制信息。系统依据该值为TASK设置超时时间，扫描并处理超时状态。

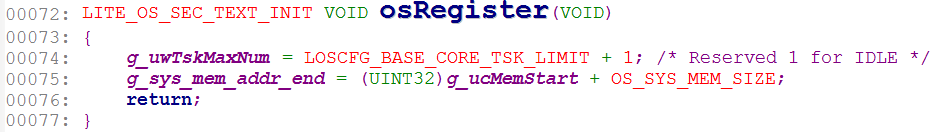
uwEvent：事件标识，表示该TASK等待哪些事件发生。

uwEventMask：事件掩码位，置1表示对应的事件发生。

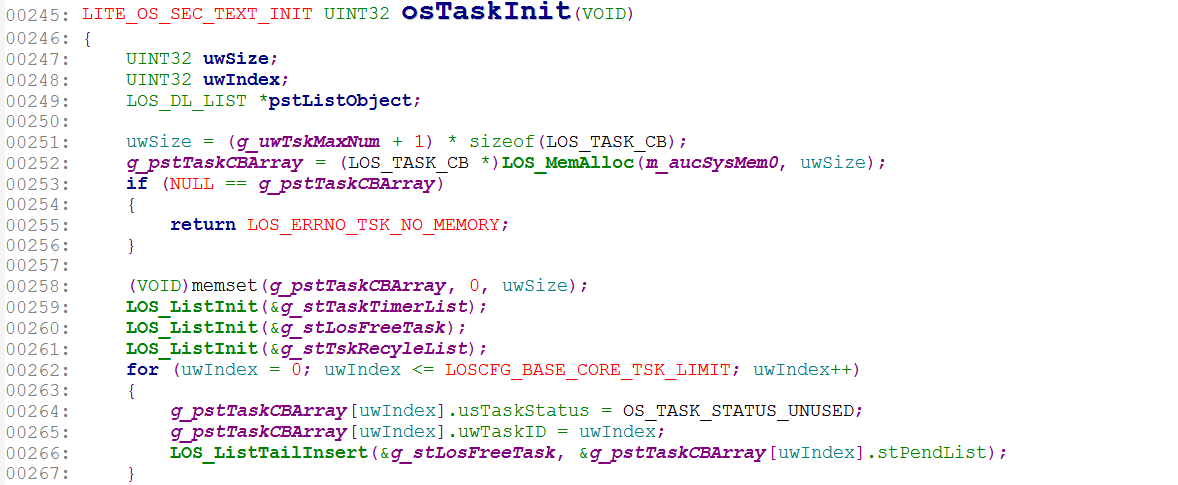
uwEventMode：等待事件的模式，有OR、AND、CLR三种模式。

\*puwMsg：在邮件队列中用于暂存邮件内存块。TASK因申请邮件队列中的内存块失败被阻塞在邮件队列上，当被唤醒时唤醒者会申请内存块并将申请结果保存在此处，该TASK被唤醒后从此处取出申请内存块的结果并做相应的处理。

### 1.1.2 osTaskInit( )函数



g\_uwTskMaxNum为用户可用的最大任务数加1，1将分配给系统的IDLE TASK，IDLE TASK也将加入到空闲链表中。



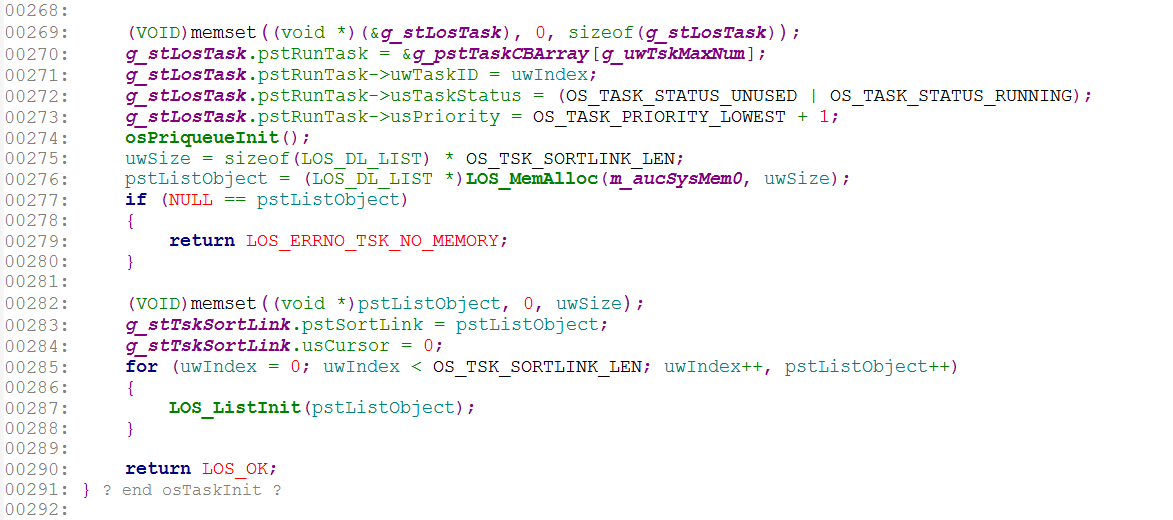
Line 251：此处加1的用途：多申请出来的这个TCB不会加入到空闲链表中，也不会加入到其它任何链表中，只独立存在于g\_pstTaskCBArray数组中，且是该数组下标最大的位置。此处加1的目的有两个，一是为了给下面的默认的正在运行的任务使用，二是为了给删除处于RUNNING状态的任务时临时过渡使用（具体参见任务删除函数）。

Line 259：初始化g\_stTaskTimerList为空链表，目前程序中未使用到该链表，具体用途未知。

Line 260：初始化g\_stLosFreeTask为空链表，用于链接处于空闲状态的TCB。

Line 261：初始化g\_stTskRecyleList为空链表，用于链接自删除TASK的TCB，自删除TASK在删除自己时将TCB加入到该链表，等到有任务需要创建时回收这个链表中的TCB。

Line 262 - 267：每一个任务都有唯一的任务控制块LOS\_TASK\_CB结构体，初始时为所有的任务设置状态为OS\_TASK\_STATUS\_UNUSED，并初始化任务的ID号，然后将所有TCB (由结构体元素LOS\_DL\_LIST stPendList 指示) 加入到空闲链表g\_stLosFreeTask中，表示所有的任务目前处于空闲状态。

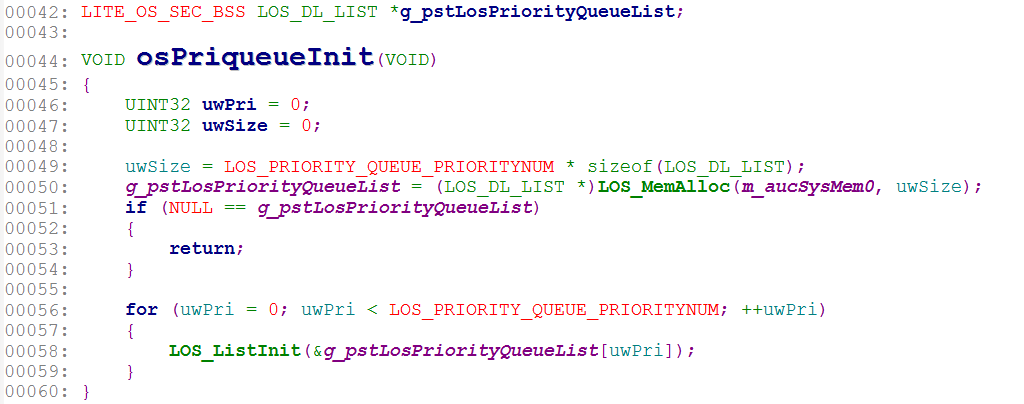


Line 269：g\_stLosTask用于指示系统调度相关内容，\*pstRunTask指示系统当前正在运行的TASK，\*pstNewTask指示系统从优先级队列中新取出的优先级最高的TASK，由这两个TCB比较是否相同来决定是否需要产生任务切换。结构体如下：



Line 270 - 273：设置默认的当前正在运行的TASK为多申请出来的那一个TASK，并设置其ID、状态和优先级，实际上该TASK没有任何内容存在，也未加入到后面的优先级队列中，所以不具备被调度到的条件。

Line 274：初始化优先级队列：优先级队列是系统调度的依据，只有处于优先级队列中的TASK才有被系统调度的可能，当某一TASK满足条件需要执行时，必须要加入到优先级队列参与调度，当某一TASK被阻塞时，必须要从优先级队列移除到相应条件的阻塞队列。每一个优先级自成一个队列，初始时设置所有优先级队列为空队列。函数实现如下：

 Line 275 - 288：初始化排序队列：排序队列主要用于与系统tick相关的时间延迟任务的排序等待，排序队列的每一个序号自成一个队列，初始时设置所有的排序队列为空。数据结构如下：



TSK\_SORTLINK\_ATTRIBUTE\_S g\_stTskSortLink;

其中：\*pstSortLink：指示排序队列的数组，32个元素，每个元素是LOS\_DL\_LIST型，每个元素链接一个排序队列，共32个排序队列。

usCursor：当前的扫描光标，一个tick中断扫描一个排序队列，32个排序队列编号为0 - 31，usCursor用于指示当前的扫描位置在哪一个排序队列上。

usUnUsed：程序中未使用到，用途未知。

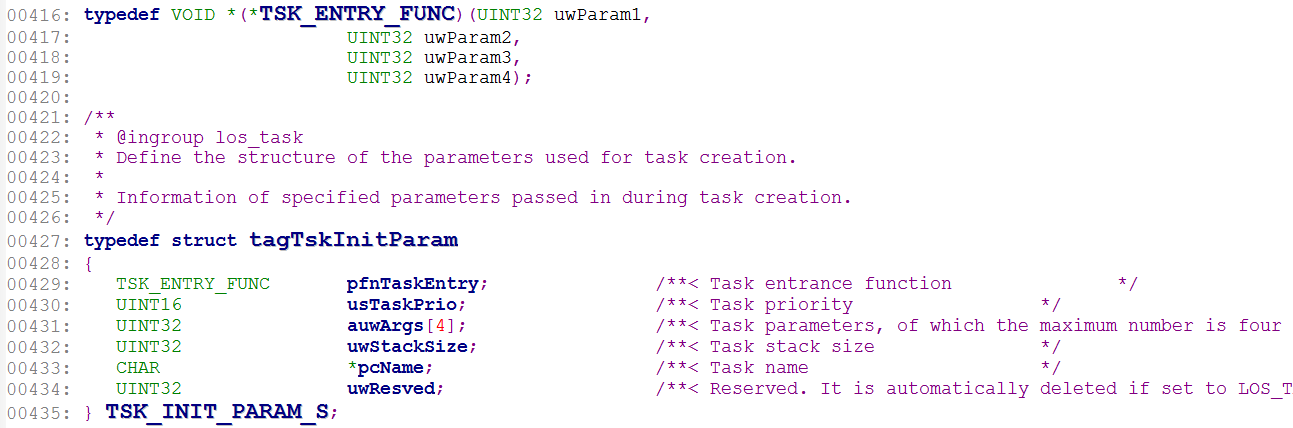
### 1.1.3 LOS\_TaskCreate( )函数

功能：创建一个TASK并启动TASK运行。

传入参数：

\*puwTaskID：用于接收系统创建好的TASK的ID号，后续用户通过该ID号操作TASK。

\*pstInitParam：用户创建TASK时需要指定的用户级参数。参数列表如下：



pfnTaskEntry：创建的TASK的工作函数。

usTaskPrio：TASK的优先级。

auwArgs[4]：TASK工作函数的4个参数。

uwStackSize：TASK栈的大小，此处没有按照8字节对齐的要求。

\*pcName：TASK的名称。

uwResved：设置LOS\_TASK\_STATUS\_DETACHED的值。表示TASK的工作函数退出后是否自行删除该TASK，置1表示自行删除，置0表示不删除。

创建过程分为两部分，第一部分是创建一个单纯的TASK，第二部分是让创建好的这个TASK参与系统调度进而真正的开始工作。



Line 589：调用函数LOS\_TaskCreateOnly( )单纯创建一个TASK。

创建TASK完成以后，该TASK的TCB已从空闲链表取下，需要将其加入到排序队列中使其参与系统调度：

Line 594：由TASK ID获取TASK的TCB。

Line 597 - 600：在LOS\_TaskCreateOnly( )函数中会设置SUSPEND状态，故此处需清除SUSPEND状态，设置READY状态并将创建好的TASK加入到对应的优先级队列中等待调度。

Line 601：从当前的优先级队列中取出优先级最高的TASK。

Line 603：根据条件启动任务切换。尝试调度新创建的TASK执行。

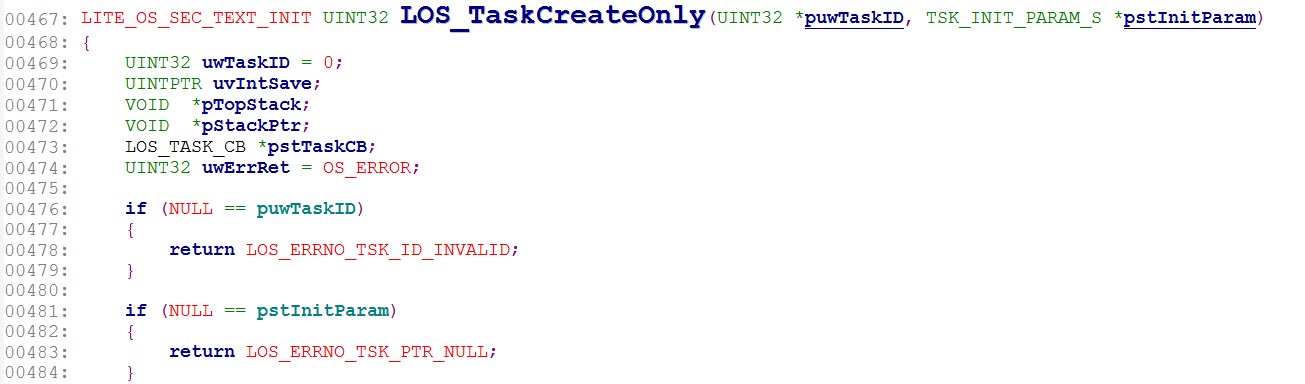
### 1.1.4 LOS\_TaskCreateOnly( )函数

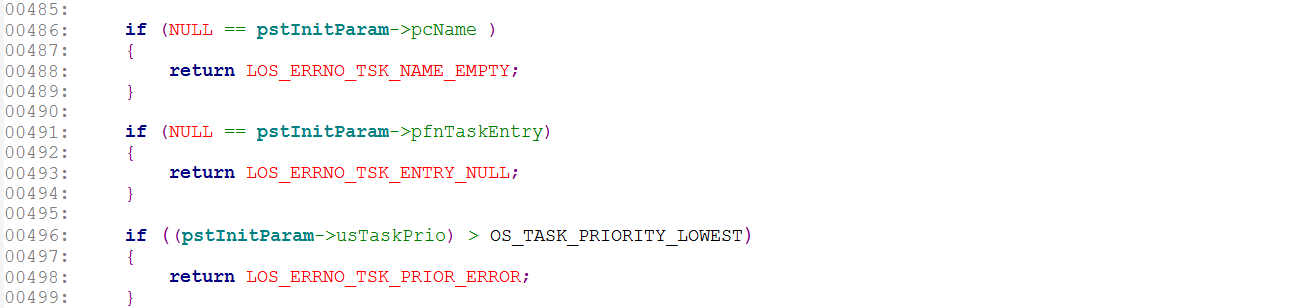
功能：单纯创建一个TASK。

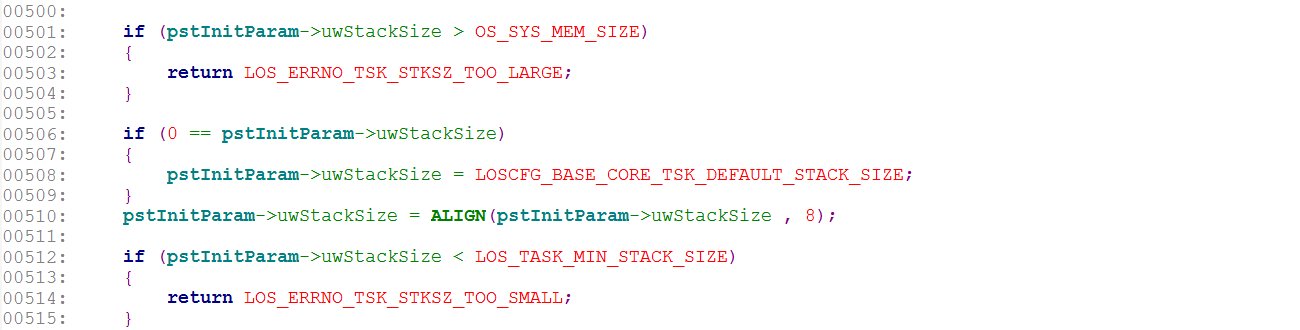
传入参数：

\*puwTaskID：用于接收系统创建好的TASK的ID号。

\*pstInitParam：用户创建TASK时需要指定的用户级参数。参数列表见LOS\_TaskCreate( )函数。



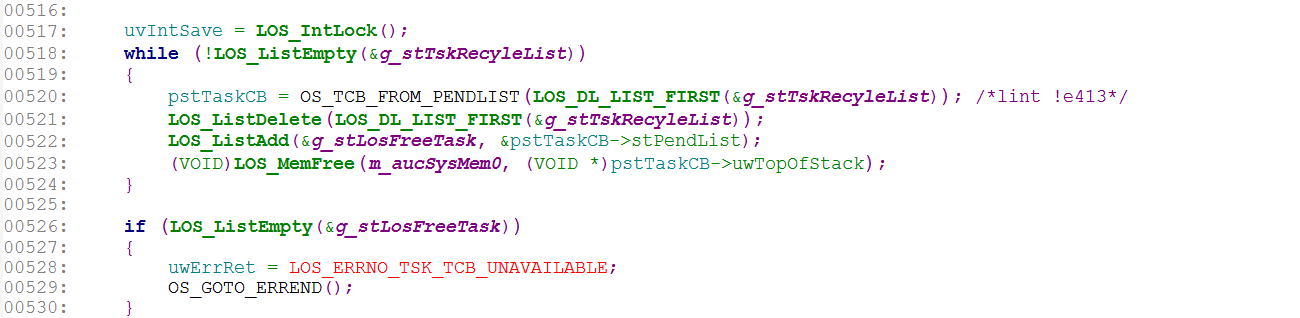




前面是创建TASK的一些合法性检测。

Line 501 - 515：如果用户传入的栈大小为0，则使用系统默认的TASK栈大小，栈的大小需在系统允许的最大值和最小值之间。

Line 510：栈的大小按照8字节对齐。如用户传入的栈大小为100，则按照8字节对齐后为104。



Line 518 - 524：检测循环队列g\_stTskRecyleList，主要目的是对之前自删除的任务控制块和任务栈进行回收。该队列主要存储被删除时处于RUNNING状态的TASK（因为删除RUNNING状态的TASK时栈信息涉及到任务切换，当时不能释放，所以不能直接进入空闲队列，故加入到循环队列，在此处对这些任务进行回收）。

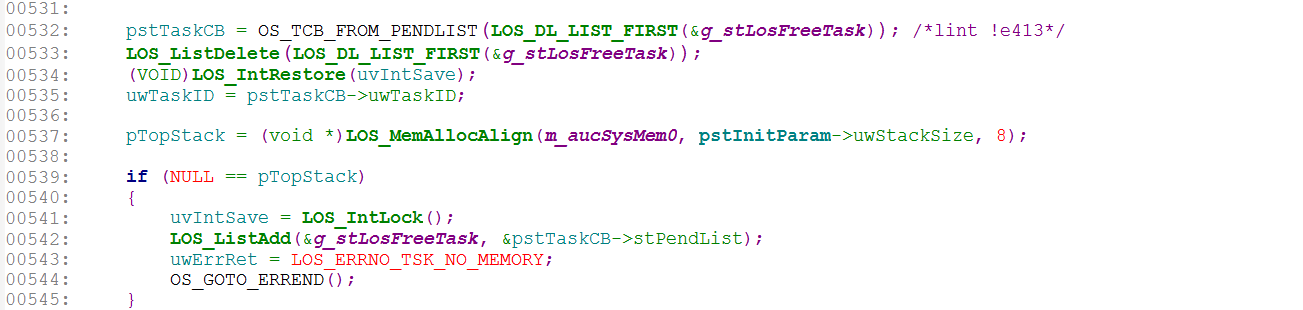
回收自删除TASK时：

Line 521：将TASK的TCB从循环队列中删除。

Line 522：将TCB重新加入到空闲队列的队头。

Line 523：释放TASK的栈空间。

Line 526：空闲队列为空，表示已经没有可用的TCB了，无法再创建新的TASK。

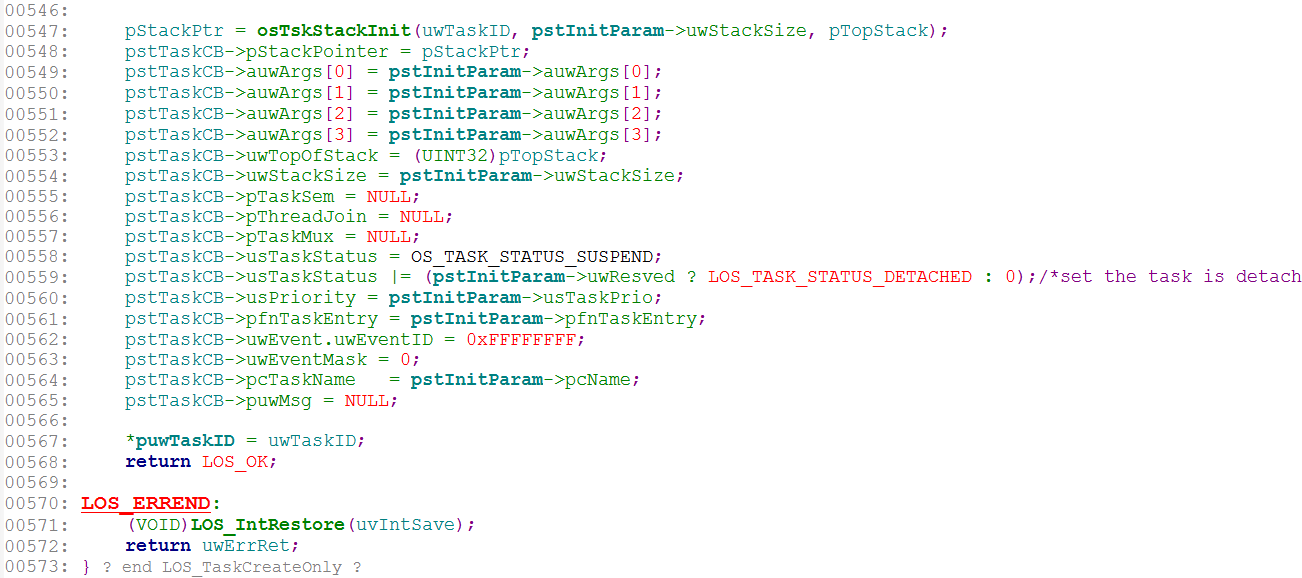


创建新的TASK：

Line 532 - 533：从空闲队列的队头取下一个TCB。

Line 537：申请TASK需要的栈空间。

Line 539：栈空间申请失败，无法创建用户的TASK，将TCB重新加入到空闲队列的队头位置。



正式创建用户的TASK：

Line 547：初始化TASK的栈。预置默认的任务切换现场，该现场在TASK第一次被调度时使用。

Line 548：保存当前的栈指针，任务切换时通过该指针来恢复任务现场。

Line 548 - 565：初始化TASK的TCB，TCB中各成员的含义在数据结构解析部分说明。

Line 558：默认设置TASK状态为SUSPEND状态，是否启动调度由调用者决定。

Line 559：设置LOS\_TASK\_STATUS\_DETACHED状态，表明TASK在工作函数退出时是否需要自删除。

Line 567：将创建好的TASK的ID号返回给用户。

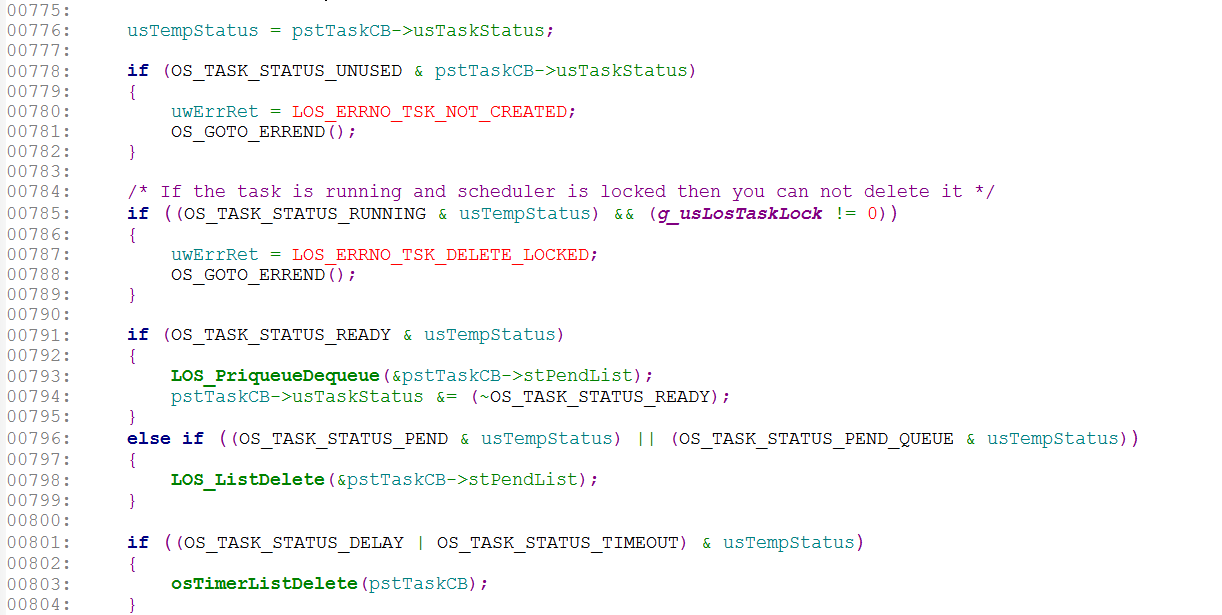
### 1.1.5 LOS\_TaskDelete( )函数

功能：删除一个TASK。传入参数：要删除的TASK的ID号。



Line 762：IDLE TASK不允许被删除。

Line 774：根据ID号找到TASK对应的TCB。



Line 778：TASK的状态若为OS\_TASK\_STATUS\_UNUSED，表示该TASK还在空闲队列中，未被分配出，不能删除。

Line 785：若TASK正在运行并且系统调度被锁，则不能删除。因为删除正在运行的TASK时需要切换任务，CPU调度新的TASK执行，任务调度被锁时无法切换任务。

Line 791：处于READY状态的TASK会放在优先级队列中等待调度，需将其从优先级队列中删除并清除READY状态。

Line 796：TASK处于PEND或PEND\_QUEUE状态，表示TASK存在于阻塞队列中，需要将其从对应的阻塞队列中删除。

Line 801：TASK处于DELAY或TIMEOUT状态，则表示该TASK同时被加入到了排序队列中等待超时，需将其从排序队列中删除，排序队列由TCB->stTimerList指示。

备注：READY、PEND、PEND\_QUEUE状态不能并存，TASK只能处于这几个状态中的其中之一。



Line 806 - 809：删除时设置要删除TASK的TCB的相关标志。

Line 811：从优先级队列中取出新的优先级最高的TASK，以备切换任务使用。如果要删除的TASK不处于RUNNING状态，则当前正在运行的TASK是优先级队列中优先级最高的TASK，所以取出的仍然是正在运行的这个TASK，所以这条语句没有实际意义。

Line 812：如果要删除的TASK处于RUNNING状态（自删除，正在运行的任务删除自己），则需要完成一次任务切换，切入新的任务执行，否则将该TASK删除以后系统不知道该执行什么任务了。

Line 814：将该TASK加入到循环队列g\_stTskRecyleList中，表示该TASK的栈在任务切换时还要使用（任务切换时需要将当前TASK的现场保存在其栈中），暂时不能完全释放。

Line 815 - 819：设置当前正在运行的TASK为初始化时默认的TASK，但默认TASK的ID、状态、栈、名称等信息却是要自删除的这个TASK的（任务切换时要用栈，其它信息不会用到）。这样，就将要删除TASK的控制块和控制块中的信息分离开，由默认的那个TASK代替要自删除的TASK继续运行，完成最后任务切换的工作。

Line 820：设置默认TASK(应该是要删除任务???)的状态为UNUSED，表示默认的TASK只是临时替代，不是真正有效的TASK。

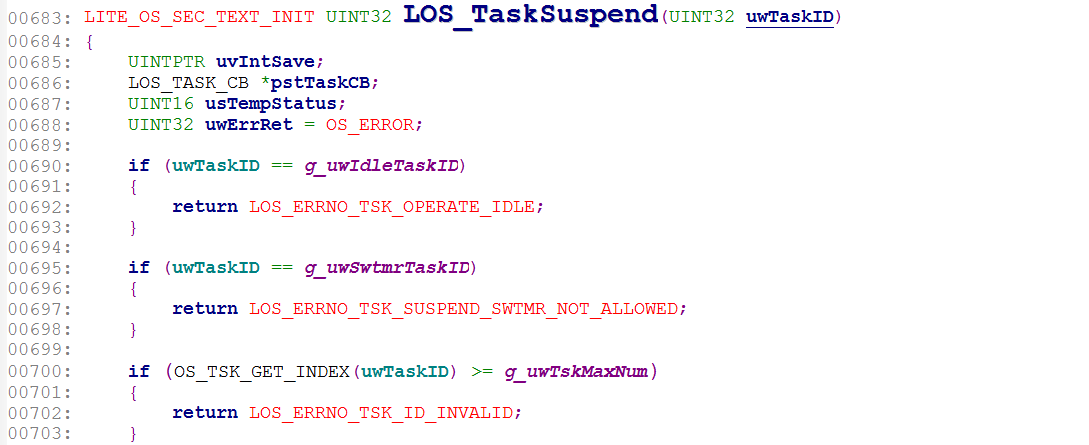
Line 822：启动一次任务切换，CPU调度新的TASK执行，因为要删除的TASK已经从优先级队列中取下了，默认的TASK只是暂时替代要删除的TASK完成最后的工作，并没有加入到优先级队列中，所以完成切换以后的后面一条语句（Line 823）永远都不会被执行到。

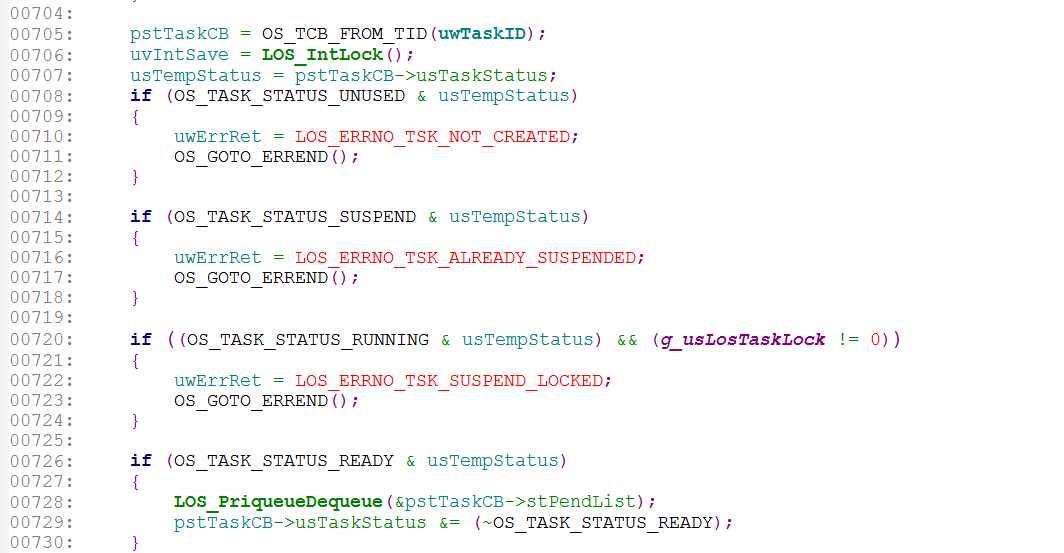
备注：TASK被自删除以后，该TASK的信息一直未被释放掉，直到有用户创建TASK时再来回收循环队列中的自删除TASK（具体参见任务创建函数）。

Line 825 - 830：若要删除的TASK不处于RUNNING状态，则不需要再保存该TASK的任何信息，直接将该TASK的TCB加入到空闲队列中，并释放该TASK所占用的栈空间，而且也不需要再产生一次任务切换，因为当前处于RUNNING状态的任务没有被删除，后续的任务调度都可以正常进行。

### 1.1.6 LOS\_TaskSuspend( )函数

功能：挂起一个TASK。传入参数：要挂起的TASK的ID号。

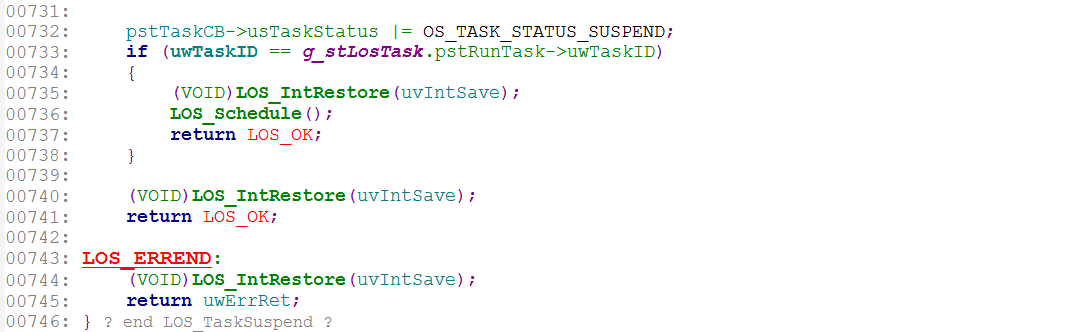
 Line 690、695：IDLE TASK和软件定时器TASK不允许被挂起



Line 714：处于SUSPEND状态的TASK不允许被再次挂起。

Line 720：如果要挂起的TASK处于RUNNING状态并且调度被锁，因为挂起后无法完成任务切换，所以不允许被挂起。

Line 726：要挂起处于READY状态的TASK时需要将其从优先级队列中移除并清除READY状态。



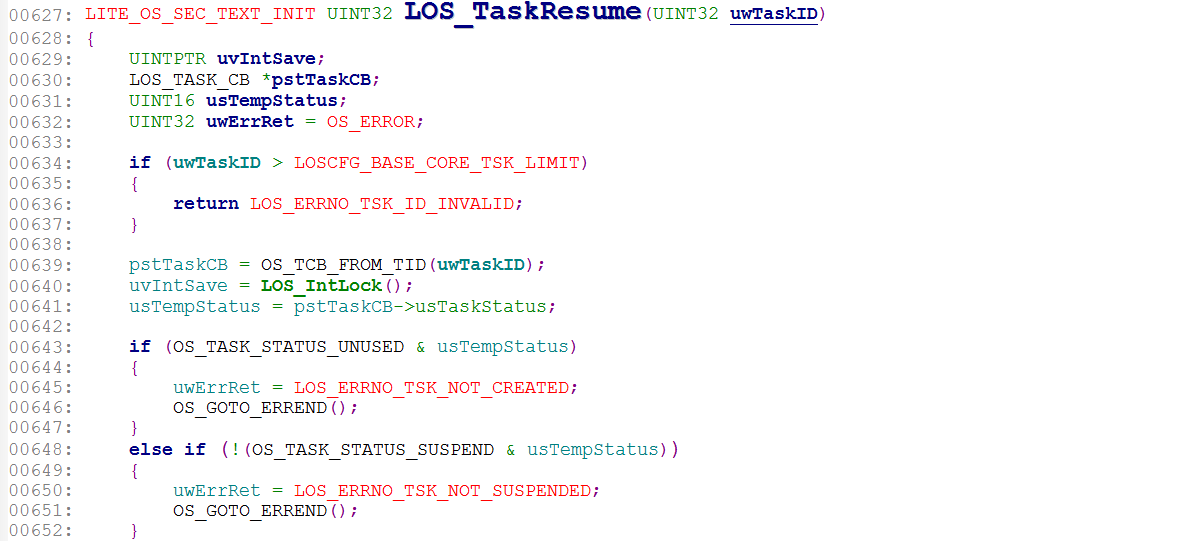
Line 732：设置TASK状态为SUSPEND状态。

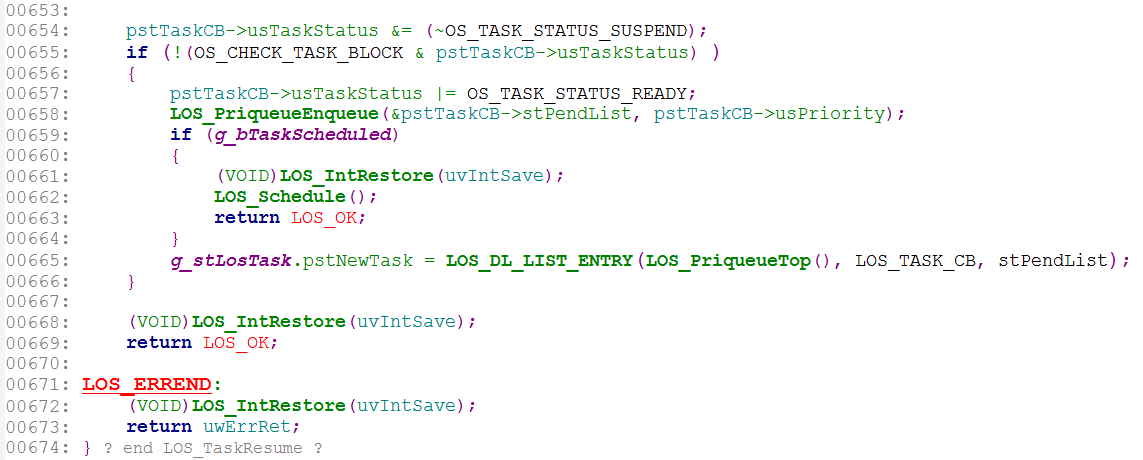
备注：如果挂起前的状态为READY状态，则挂起后该TASK将不处于任何队列中，只存在于g\_pstTaskCBArray数组中。如果挂起前的状态中有阻塞状态，则挂起后该TASK既存在于g\_pstTaskCBArray数组中，也存在于相应的阻塞队列中，所以在阻塞队列中被唤醒时需要判断是否处于SUSPEND状态。如果挂起前的状态中有DELAY或TIMEOUT状态，则挂起后该TASK既存在于g\_pstTaskCBArray数组中，也存在于排序队列中，所以在排序队列中因超时时间到被唤醒时需要判断是否处于SUSPEND状态。

如果要挂起的TASK是正在运行的TASK，说明要挂起的TASK是优先级队列中优先级最高的任务，则需要产生一次任务切换，因为要挂起的TASK已经从优先级队列中删除了，所以这一次任务切换一定会调度新的任务执行。

### 1.1.7 LOS\_TaskResume( )函数

功能：恢复一个处于挂起状态的TASK。传入参数：要恢复的TASK的ID号。





Line 654：恢复被挂起的TASK时，只需要清除SUSPEND状态即可。

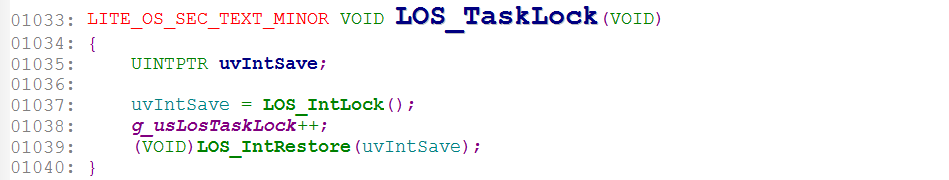
Line 655：清除SUSPEND状态后，如果TASK不处于阻塞状态，说明该TASK满足参与调度的条件，则需要设置READY状态并将其重新加入到优先级队列中参与调度。

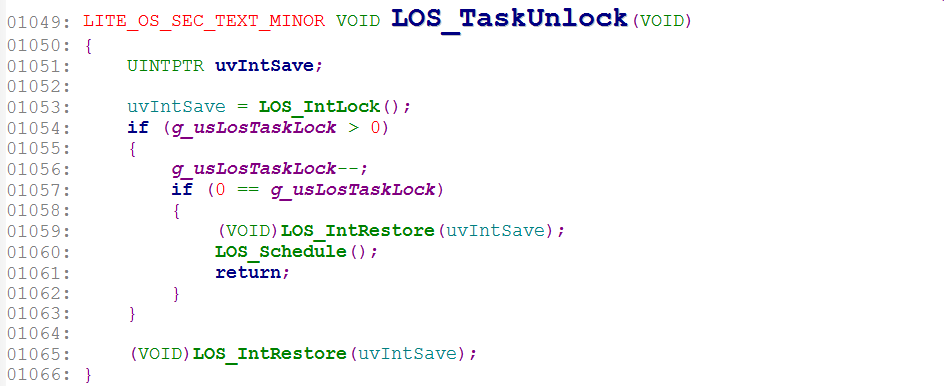
Line 662：启动一次任务切换，CPU调度优先级队列中新的优先级最高的TASK执行。

Line 665：系统还未启动调度，先保存新的优先级最高的TASK，等到系统启动调度以后再行调度。

Line 669：清除SUSPEND状态后，如果TASK处于阻塞状态，则此处不需要做任何操作，TASK会被相应的阻塞条件唤醒进而重新参与系统调度。被阻塞条件唤醒时会判断SUSPEND状态。

### 1.1.8 LOS\_TaskLock( )和LOS\_TaskUnlock( )函数



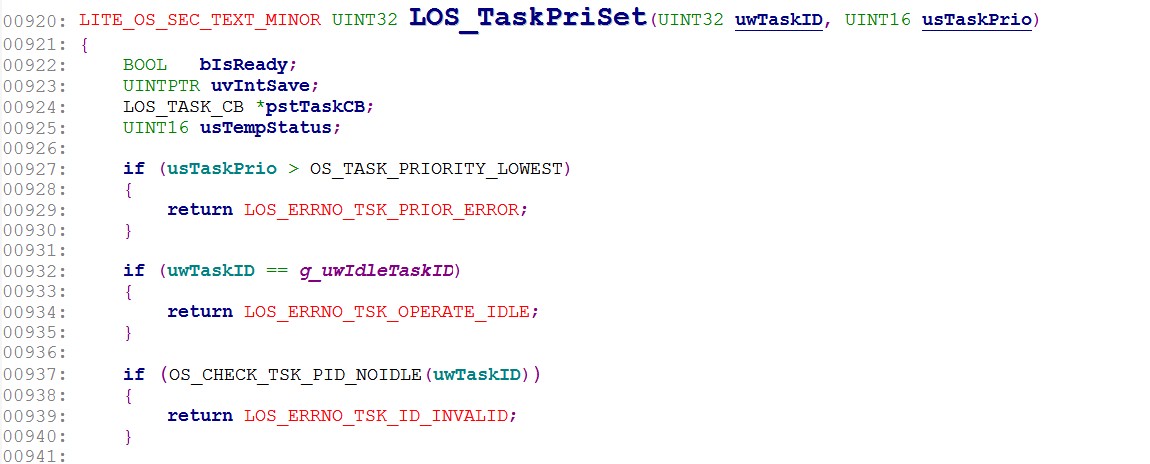


Lock只是锁定任务的调度，而不是锁中断。如果任务的调度被锁，则满足任务调度条件时的任务调度被放弃，需要等到解锁任务调度时由解锁函数启动任务切换。在解锁任务调度时即便没有因锁定而放弃的调度，也会产生一次任务调度。

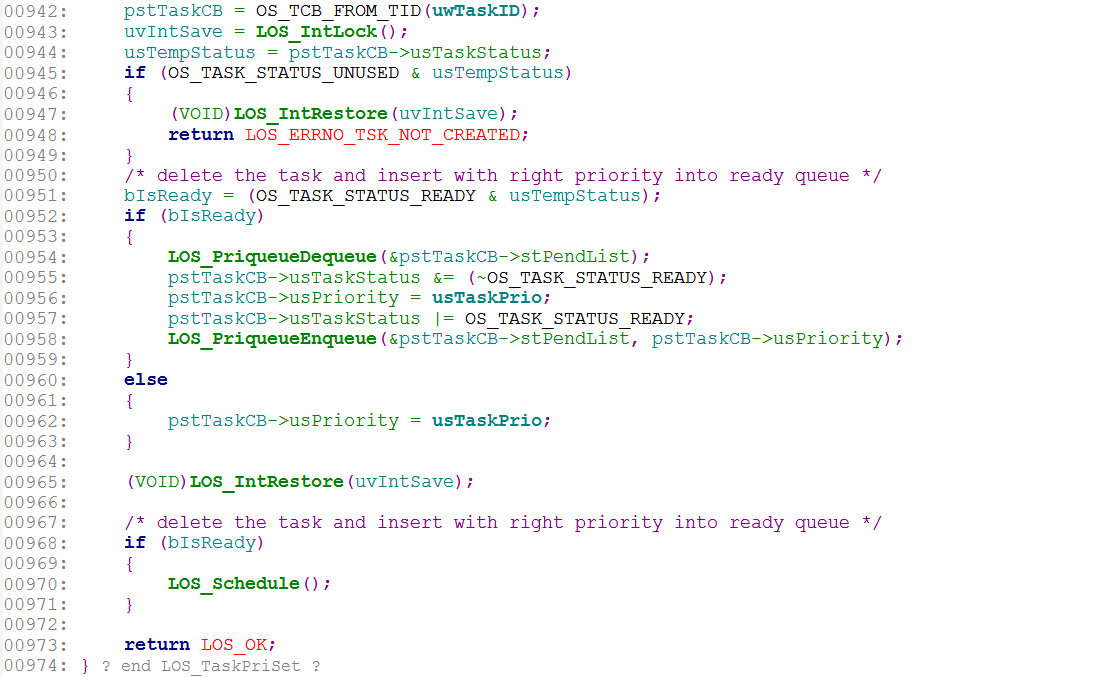
### 1.1.9 LOS\_TaskPriSet( )和LOS\_TaskPriGet( )函数



由TASK ID找到TASK对应的TCB，然后从TCB中取出优先级返回。



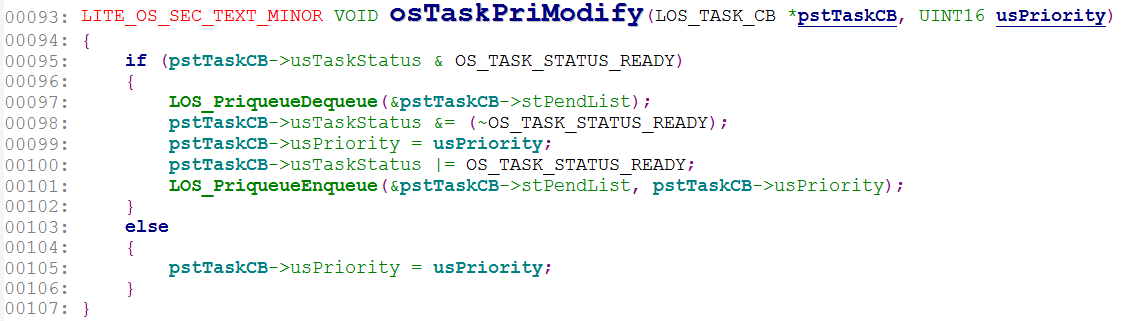
Line 932：IDLE TASK的优先级最低且不允许被更改。



Line 952 - 959：如果要设置的TASK当前处于优先级队列，需要将该TASK从原有的优先级队列中取下，并设置新的优先级，然后将其加入到新的优先级队列等待调度，并启动一次任务切换。

### 1.1.10 osTaskPriModify( )函数

功能：更改一个TASK的优先级。与函数LOS\_TaskPriGet( )实现相同的功能。



### 1.1.11 osTskStackInit( )函数

功能：初始化TASK的栈空间。栈顶在内存低地址位置，栈底在内存高地址位置。栈空间的初始化主要是设置默认的任务现场，也就是填充TSK\_CONTEXT\_S结构体，该结构体的元素排列与任务切换恢复现场时出栈的顺序一致。这样，在任务第一次被调度时就会将栈中默认的信息弹出，CPU按照默认信息的指示执行这个TASK。当TASK启动以后，这些默认值将不复存在。

传入参数：

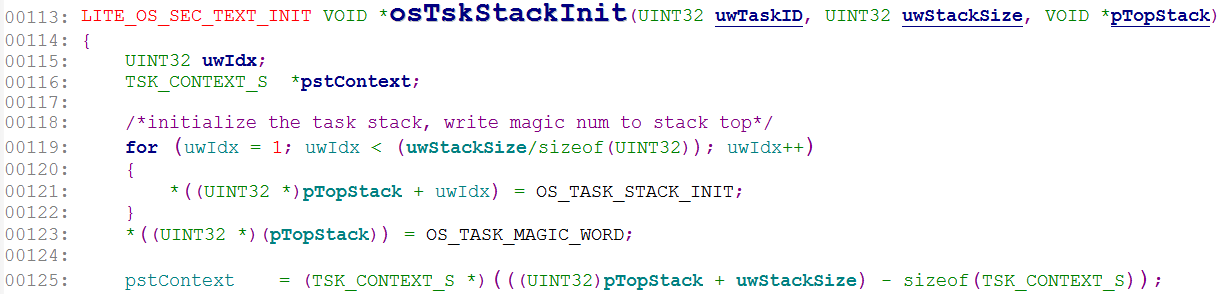
uwTaskID：TASK的ID号。

uwStackSize：栈大小，需按8字节对齐。

\*pTopStack：栈顶指针，指示栈的位置。

返回值：

\*pstContext：返回当前栈指针，也就是当前TASK的PSP。

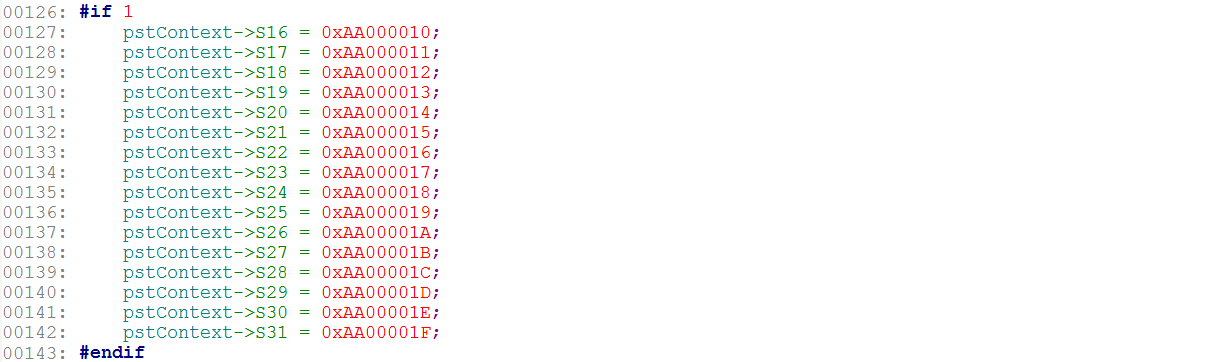


OS\_TASK\_STACK\_INIT：0xCACACACA。OS\_TASK\_MAGIC\_WORD：0xCCCCCCCC。

Line 123：栈顶四个字节初始化为0xCCCCCCCC。

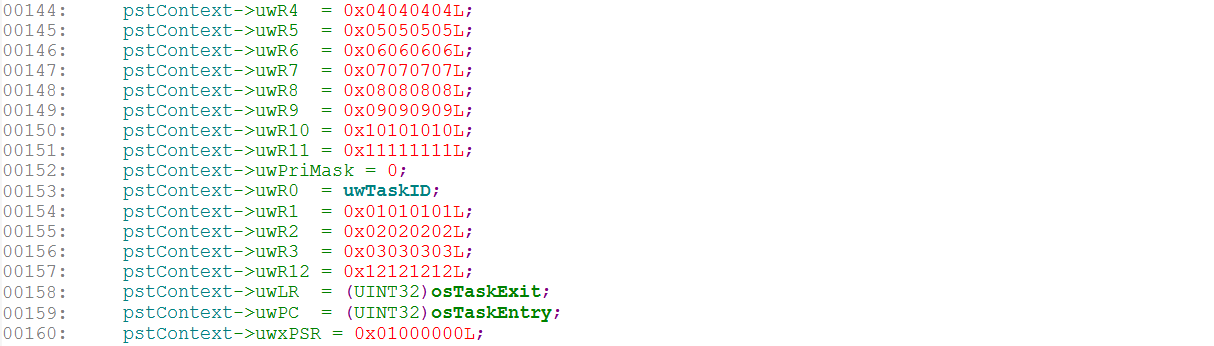
Line 119 - 122：从栈顶向下4字节位置到栈底的所有空间初始化为0xCACACACA。

Line 125：从栈底向上开辟大小为sizeof(TSK\_CONTEXT\_S)空间，用于填充下面的信息。



Line 126 - 143：

S16 - S31：对应于FPU浮点处理器单元的浮点寄存器S16 - S31。填入的默认值无实际意义。也表示64位双精度浮点运算的D8 - D15。



Line 144 - 160：

uwR0 - uwR12：对应通用寄存器R0 - R12，uwR0被填入TASK的ID号，uwR1 - uwR12的默认值无实际意义。

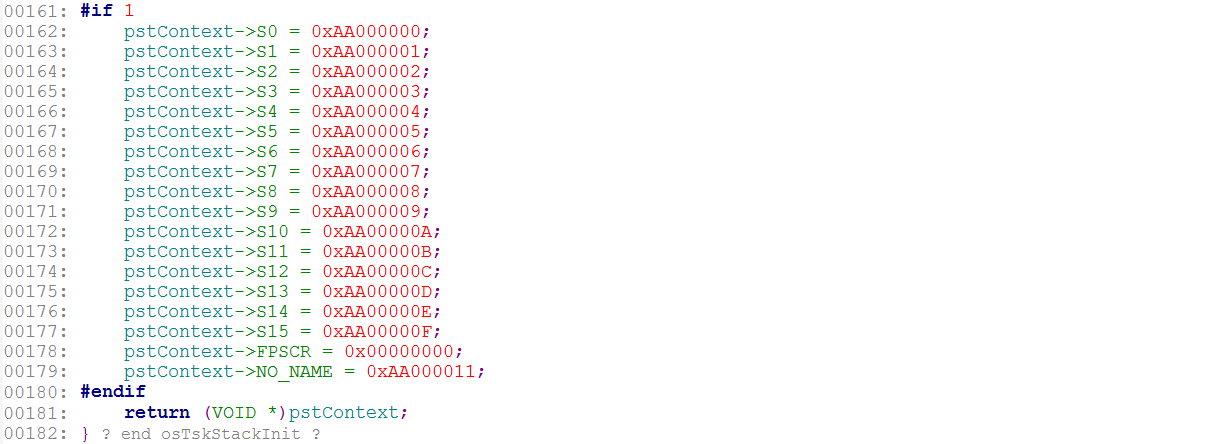
uwLR：对应寄存器R14，是连接寄存器（LR），主要作用是保存子程序的返回地址，以便在执行完子程序时恢复现场。此处设置默认值为osTaskExit( )函数的地址。

uwPC：对应寄存器R15，是程序计数器（PC），指向当前的程序地址。此处设置默认值为osTaskEntry( )函数的地址，表示该TASK被调度时，从函数osTaskEntry( )处开始执行。

uwPriMask：对应于R12寄存器，用于保存当前TASK的PRIMASK寄存器的值，当系统再次调度时用于恢复PRIMASK寄存器。

寄存器R13无对应关系。R13表示堆栈指针，可通过PUSH和POP操作实现栈存储的访问。栈指针包括两个：主栈指针MSP和进程栈指针PSP。MSP为默认指针，复位后或Handler模式时只能是MSP，而PSP只能在线程模式使用。

uwxPSR：对应寄存器xPSR，是程序状态寄存器。



Line 161 - 180：

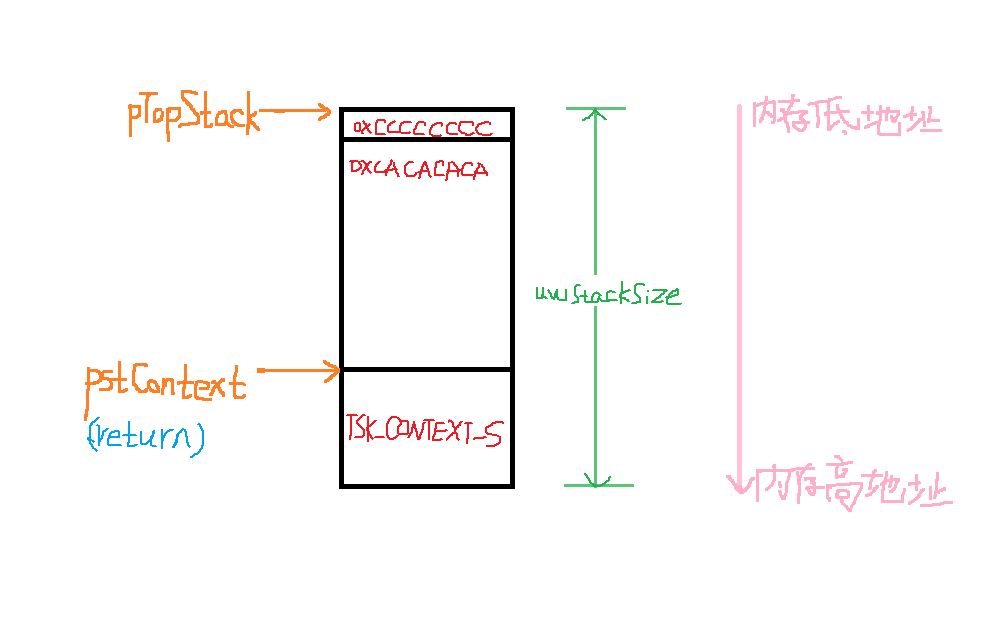
S0 - S15：对应于浮点寄存器的S0 - S15。填入的默认值无实际意义。也表示64位双精度浮点运算的D0 - D7。

FPSCR：对应于浮点状态和控制寄存器FPSCR，FPSCR有两个功能：一是提供浮点运算结果的状态信息，如负标志、进位标志等。二是定义一些浮点运算动作，如何舍入等。

NO\_NAME：对应于中断产生时CPU自动入栈的最开始的那个未知的字。

Line 181：返回用户可用的栈的栈底。

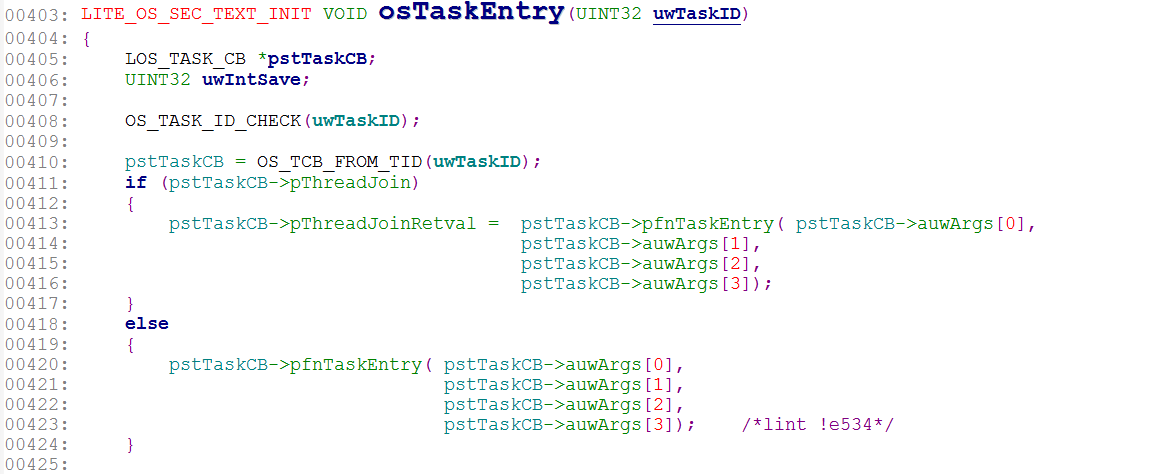
栈空间示意图如下：



### 1.1.12 osTaskEntry( )函数

功能：TASK的入口函数，用于调用执行ID号对应的TASK的工作函数及工作函数退出后的处理。

传入参数：uwTaskID是TASK的ID号。

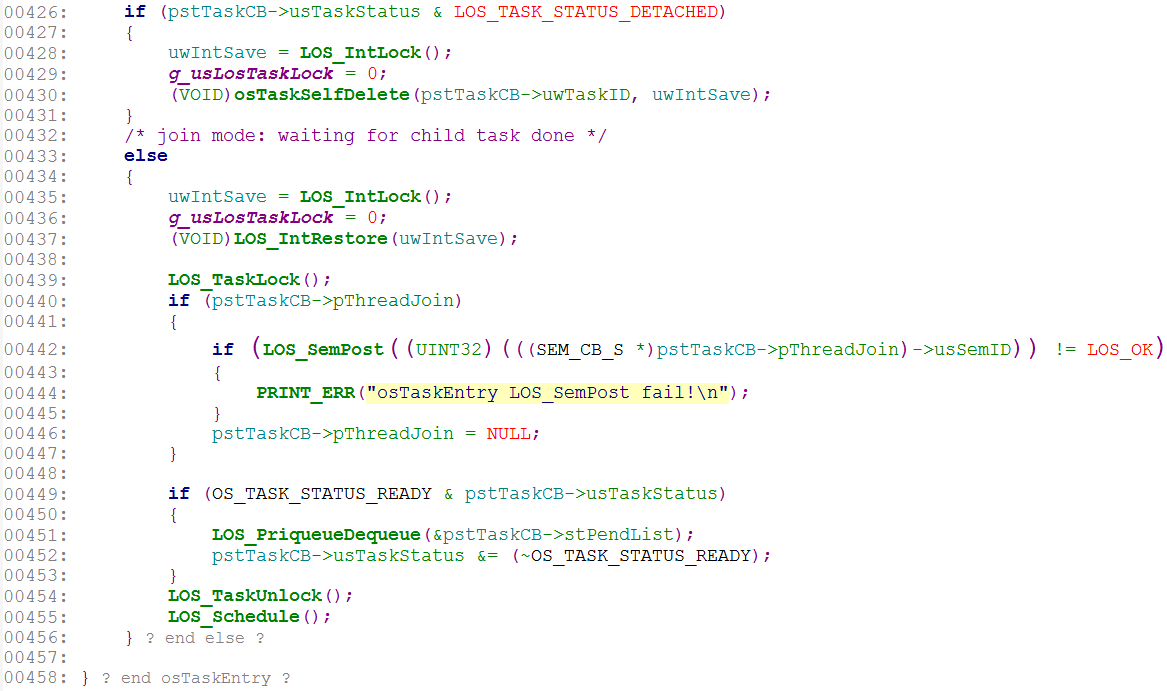


Line 410：由ID号找到对应的TASK的TCB。

Line 411：父TASK存在。

Line 413、420：执行TASK的工作函数。

TASK的工作函数退出后该TASK继续向下执行。一般情况下，TASK的工作函数是死循环形式，循环完成用户的任务，永远不会退出，CPU对该TASK的调度会一直陷在pfnTaskEntry( )函数中。但如果用户对工作函数设置了退出条件，则工作函数退出后该TASK的使命已经完成了，需要根据条件做自删除或等待其它TASK删除这个TASK。



Line 426：TASK在创建时被设置了DETACHED属性，表示该TASK不是在其它TASK中创建的，不属于其它某一TASK的子TASK，所以工作函数退出后，需要将这个TASK自删除。

Line 433：TASK在创建时未设置DETACHED属性，表示该TASK是被某一个TASK创建的子TASK，父TASK通过信号量来回收子TASK，所以在这种情况下，不能直接将自己删除，需要发信号量给父TASK，由父TASK来回收子TASK。

Line 440：父TASK存在，则向父TASK发送信号量，父TASK被唤醒后回收子TASK。LOS\_SemPost( )唤醒时会产生一次任务切换，尝试调度父TASK，至于父TASK能否立即被调度到，则要看其优先级而定。

Line 449：该TASK还未被父TASK回收，则需要将其从优先级队列中取下并清除READY状态。

Line 455：启动一次任务切换。

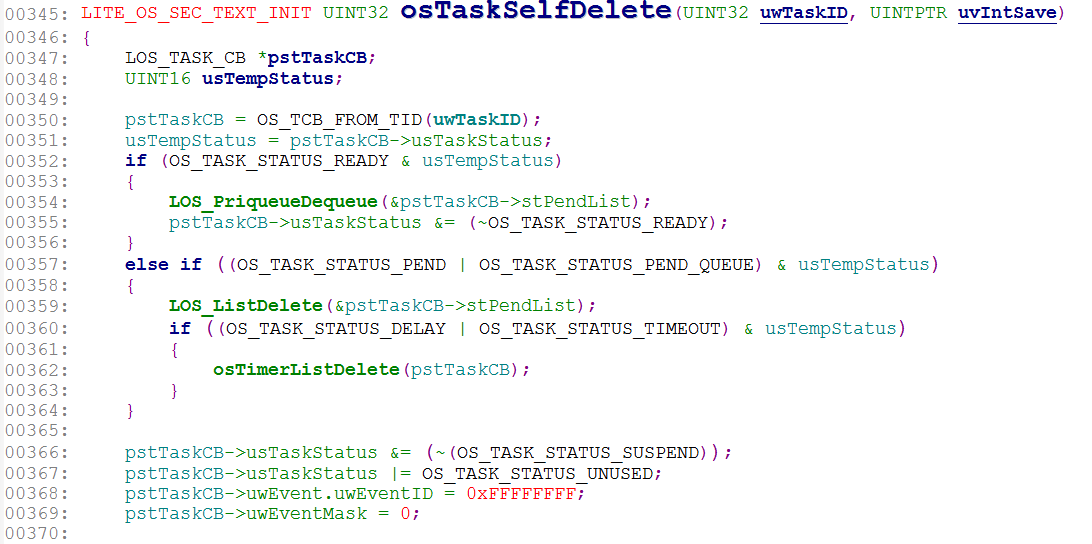
### 1.1.13 osTaskSelfDelete( )函数

功能：TASK自删除。

传入参数：

uwTaskID：TASK ID号。

uvIntSave：中断相关变量，调用该函数时中断已经被禁止，需要在该函数中通过该值来恢复被禁止中断。



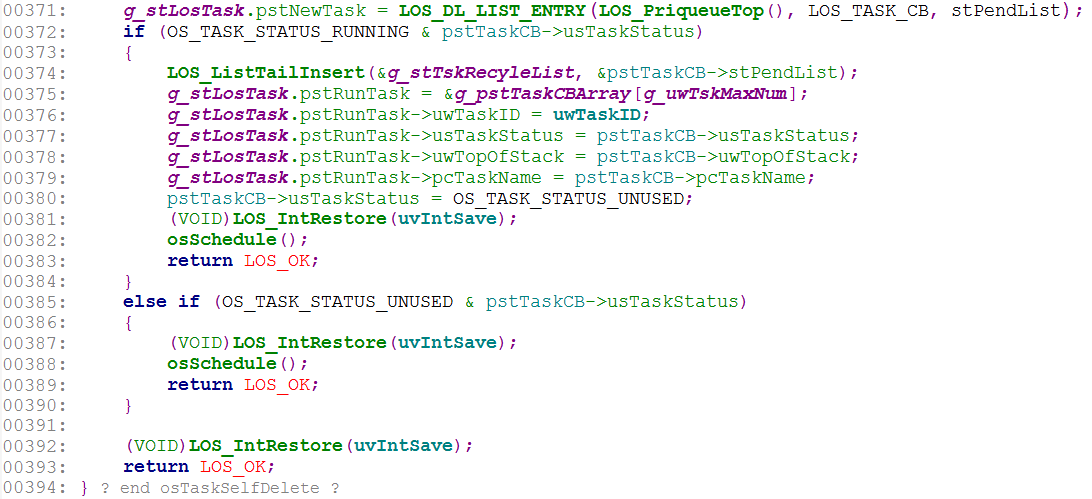
Line 350：由TASK的ID号获取到TCB。

Line 352：READY状态说明TASK处于优先级队列参与系统调度，需要将其从优先级队列删除并清除READY状态。

Line 357：PEND或PEND\_QUEUE状态说明TASK不在优先级队列，被挂在阻塞队列上等待唤醒，需要将其从阻塞队列中删除。自删除的TASK一定是处于RUNNING状态的，这种情况应该是不存在的，这种写法用意不明。

Line 360：DELAY或TIMEOUT状态说明TASK不但在阻塞队列上，同时还处于排序队列中等待超时唤醒，需要将其从排序队列中删除。

Line 366 - 369：清除TCB的状态和事件相关标志。



Line 371：从优先级队列中取出目前优先级最高的TASK。

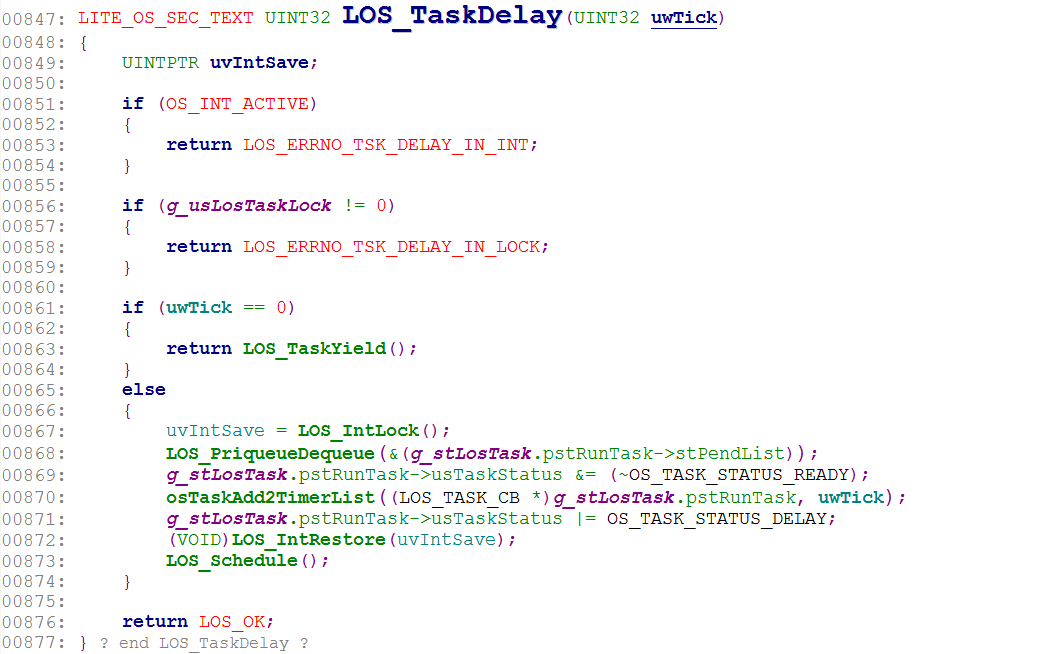
Line 372：要删除的TASK处于RUNNING状态，删除方式与LOS\_TaskDelete( )函数中一致。

Line 385：要删除的TASK不在RUNNING状态，因为是自删除过程，所以要删除的TASK一定是正在运行的，这种情况是不存在的，这个写法用意不明。

## 1.2 任务延时相关函数

### 1.2.1 LOS\_TaskDelay( )函数

使任务延时等待一段时间，时间单位是tick。



Line 851：中断中不允许延时操作。

Line 856：因为任务延时需要涉及到任务切换，故任务调度被锁时不允许延时任务。

Line 863：若传入的延时tick为0，则调用函数LOS\_TaskYield( )，主要功能是尝试调度与当前TASK同优先级的任务，如果没有与当前TASK同优先级的任务，则CPU继续执行这个任务，不产生任务切换，延时为0个tick，若有与当前TASK同优先级的任务，则会依次调度这些任务，这时的延时就无法确定了。故传入延时tick为0的延时有可能是0个tick，也有可能是n个tick。

当延时tick大于0时，需要将该TASK加入到排序队列中，等待延时时间到后由超时唤醒：

Line 868 - 869：将当前的TASK从优先级队列中取下，清除READY状态，这样该TASK就不再参与系统调度了。

Line 870 - 871：调用函数osTaskAdd2TimerList( ) 将该TASK加入到排序队列中，设置该TASK状态为DELAY（在其它阻塞条件下，比如读消息队列超时，同样会将TASK加入到排序队列中，但是设置的状态是TIMEOUT）。

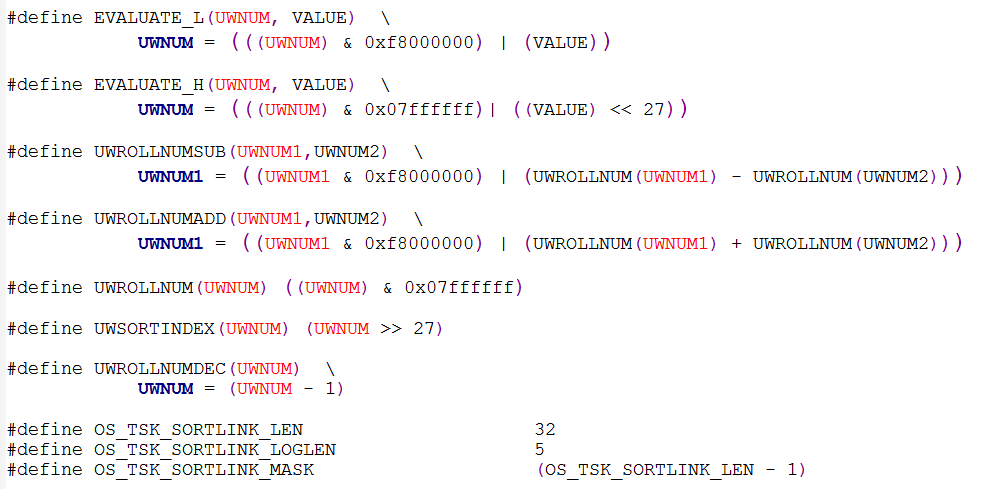
Line 873：当前任务延时执行，启动一次任务切换调度优先级队列中优先级最高的任务执行。

调用函数LOS\_Schedule( ) 切换任务后，CPU调度执行新的任务，(???)当前这个任务被停在了LOS\_Schedule( ) 函数的后面一条语句上（Line 876），当超时被唤醒时，该任务继续执行。

注意：延时的时间不一定绝对准确，有可能会因为高优先级任务工作而致使低优先级任务的延时比用户期望的延时长。

### 1.2.2 osTaskAdd2TimerList( )函数

将某一TASK加入到排序队列中等待超时，此处只是将TASK按照超时时间加入到排序队列的相应位置，具体的超时判断在osTaskScan( ) 函数中实现。



EVALUATE\_L(UWNUM, VALUE)：UWNUM的高5bit不变，低27bit变为VALUE的值。

EVALUATE\_H(UWNUM, VALUE)：UWNUM的高5bit变为VALUE的值，低27bit不变。

UWROLLNUMSUB(UWNUM1,UWNUM2)：UWNUM1 = 高5bit不变，低27bit变为 ( UWNUM1的低27bit ) - ( UWNUM2的低27bit )的值。

UWROLLNUMADD(UWNUM1,UWNUM2)：UWNUM1 = 高5bit不变，低27bit变为( UWNUM1的低27bit ) + ( UWNUM2的低27bit )的值。

UWROLLNUM(UWNUM)：获取临时值 = UWNUM的高5bit清0，低27bit不变，即获取UWNUM的低27bit的值，但UWNUM的原值不变。

UWSORTINDEX(UWNUM)：获取临时值 = UWNUM的高5bit的值，UWNUM原值不变。

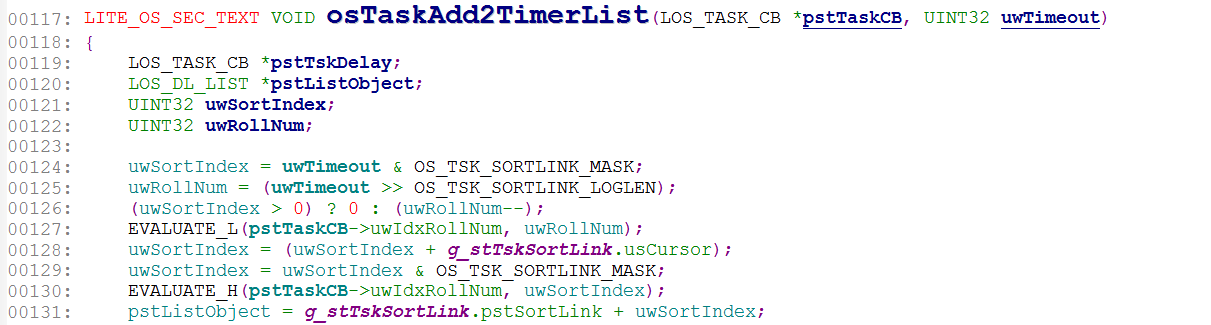
UWROLLNUMDEC(UWNUM)：UWNUM = UWNUM - 1。

排序队列由g\_stTskSortLink.pstSortLink指示，初始化时一共申请32个排序队列，编号为0-31，每个tick中断扫描一个排序队列，所有排序队列按照序号依次循环扫描，故完整扫描一次排序队列的时间是32个tick。

传入参数：

\*pstTaskCB：要加入到排序队列中的TASK的任务控制块。

uwTimeout：超时时间，以tick为单位，如果用户希望延时为秒或毫秒，则需要先将时间单位转换成对应的tick数。



pstTaskCB->uwIdxRollNum由两部分组成，高5位是Index，表示该TASK将加入到序号为Index的排序队列中（这个序号是绝对序号，从0到31号）。低27位是RollNum，表示循环扫描次数，即需要完全扫描RollNum次排序队列（每次扫描耗时32个tick）。

Line 124 - 125：uwSortIndex获取uwTimeout的低五位，uwRollNum获取uwTimeout的高27位。

Line 126：若uwSortIndex = 0，则需要将uwRollNum减1，这个与扫描排序队列的方式有关，具体原因在下面的例值2中说明。

Line 127：uwRollNum与排序队列的序号没有关系，所以直接调用EVALUATE\_L将其放入到pstTaskCB->uwIdxRollNum的低27位。

Line128 - 130：g\_stTskSortLink.usCursor是扫描光标，表示当前的扫描点在哪一个排序队列上，uwSortIndex是TASK要放入的排序队列的序号，这个序号是相对序号，需要按照当前的扫描光标调整。调整以后将最后得到的绝对序号通过调用EVALUATE\_H放入pstTaskCB->uwIdxRollNum的高5位。

例值1：以用户传入uwTimeout = 200为例，uwSortIndex = 200 & 31 = 8，uwRollNum = 200 >> 5 = 6。所以200个tick的延时被分解为200 = 6\*32 + 8，即完整扫描排序队列6次（每次32个tick），然后再多扫描8个tick即可完成200个tick的延时。所以需要将该TASK加入到相对序号为8的排序队列中，由相对序号和当前的扫描光标就可以得到绝对序号，假设当前的usCursor是15，那么绝对序号就是23，需要将其加入到23号排序队列上，这样的话光标从当前的15号扫描到23号需要8个tick，然后再完整的扫描6次就可以完成200个tick的延时了。假设当前的usCursor是27，则绝对序号就是3，排序队列是循环扫描的。

注意：假设当前的usCursor是15，则绝对序号是23，在osTaskScan( )函数中，对该TASK的扫描是从当前光标的下一个位置开始的，即16号开始，这样的话从16号扫到23号一共扫了7个tick，从此处设置延时到开始扫描（下个tick中断）需要的时间在0到1个tick之间，粗略按照1个tick计算，故8个tick的延时并不完全准确，在7到8个tick之间。

例值2：以用户传入uwTimeout = 192为例，uwSortIndex = 192 &31 = 0，uwRollNum = 192 >> 5 = 6。所以192个tick的延时被分解为192 = 6\*32 + 0，即完整的扫描排序队列6次即可完成192个tick的延时，需要将该TASK加入到相对序号为0的排序队列中。也就是当前usCursor所在的位置，但由于扫描排序队列的方式问题，开始扫描时是从当前光标的下一个位置开始，这样的话在扫描判断中需要扫描uwRollNum + 1次才能扫描完成，具体的扫描过程在osTaskScan( )函数中说明，所以对于uwSortIndex = 0的这种情况，需要将uwRollNum减1。



确定了要放入的排序队列后，需要将TASK加入到对应的排序队列中去等待扫描。

Line132：如果要加入的排序队列为空，直接将其加入排序队列的队尾，若不为空，需要对完整的扫描周期数做一些处理，具体处理如下。

Line138 - 154：将完整的扫描周期数按照从短到长的顺序依次排列，且当前这个TASK的RollNum是去除了前面所有TASK的RollNum之和后的数字，这样如果第一个TASK超时时间未到，那么当前排序队列的后续所有的TASK的超时时间均不可能到，这样每次只需要扫描当前排序队列的第一个TASK即可，可以减少不必要的扫描耗时。

例值3：假设有5个TASK等待延时，TASK的信息如下：

TASK NAME： A B C D E

TASK ROLLNUM： 5 2 6 10 6

TASK INDEX： 8 8 8 8 8

TASK DELAY TICK： 168 72 200 328 200

这5个TASK最终都被加入到相对序号为8的排序队列上，加入后最后在排序队列上的顺序和ROLLNUM信息如下：

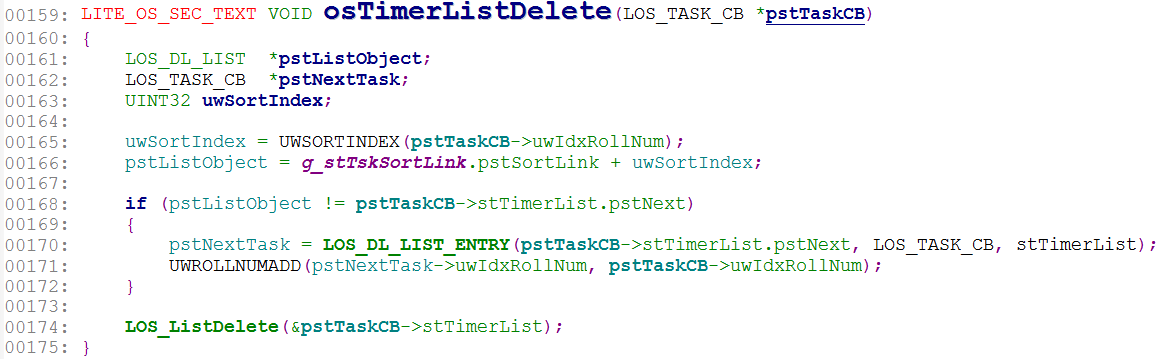
TASK NAME： B A C E D

TASK ROLLNUM： 2 3 1 0 4

在5个TASK中，B需要的循环扫描次数最少，所以在最前面，A需要5次，当B扫描完成以后A只要再扫描3次也就完成了，C需要6次，当A扫描完成后只需要再扫描1次，C就完成了，E也需要6次，所以C完成以后E也就完成了，C和E会在同一个tick中断中被唤醒，D需要10次，前面的总次数是6次，则前面的TASK完成以后D只需要再扫描4次就完成了，若有新加入或删除的TASK，有些值会被重新调整。

### 1.2.3 osTimerListDelete( )函数

将TASK从排序队列中删除。



Line 165：wSortIndex用于获取TASK所在排序队列的绝对序号。

删除时有两种情况：

Line 174：第一种情况是该TASK在排序队列的尾部，后面没有其它TASK存在，删除该TASK不会影响到其它TASK，则直接将其删除就可以了。

Line 168 - 174：第二种情况是该TASK后还有TASK存在，后面的TASK的RollNum依赖于该TASK的RollNum，（参见前面的例值3）所以需要将该TASK的RollNum的值加到该TASK后面的一个TASK上去，然后再将该TASK删除。

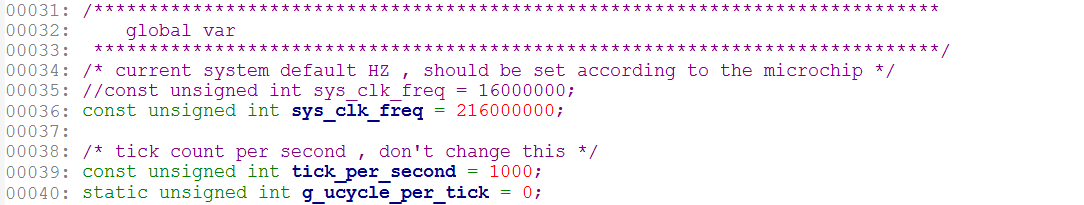
以前面的例值3为例：假设要删除A，则需要将A的RollNum值加到C上去，删除后新的排序队列如下：

TASK NAME： B C E D

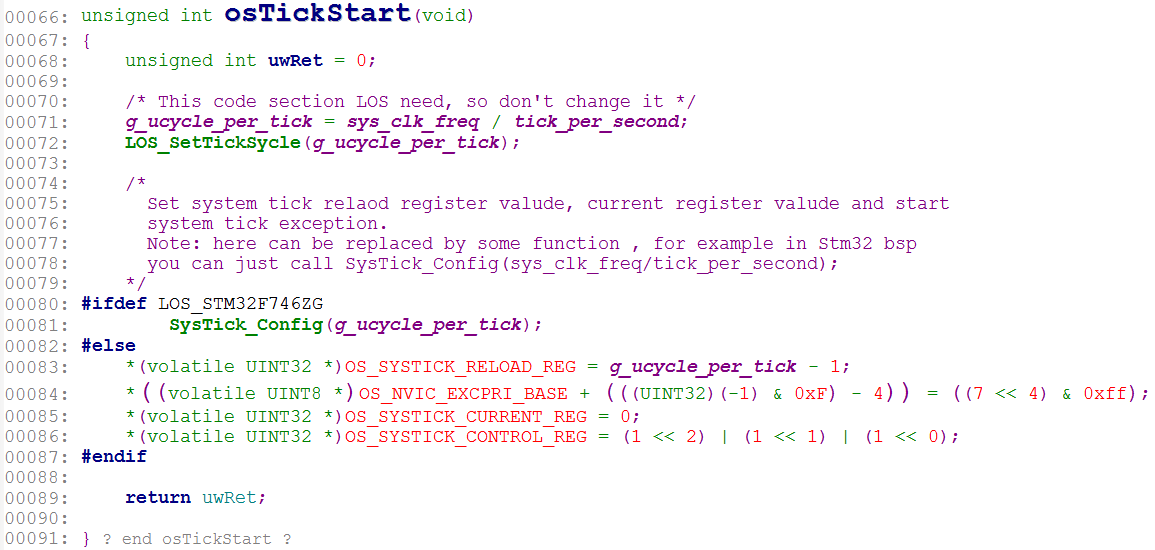
TASK ROLLNUM： 2 4 0 4

## 1.3 系统调度相关函数

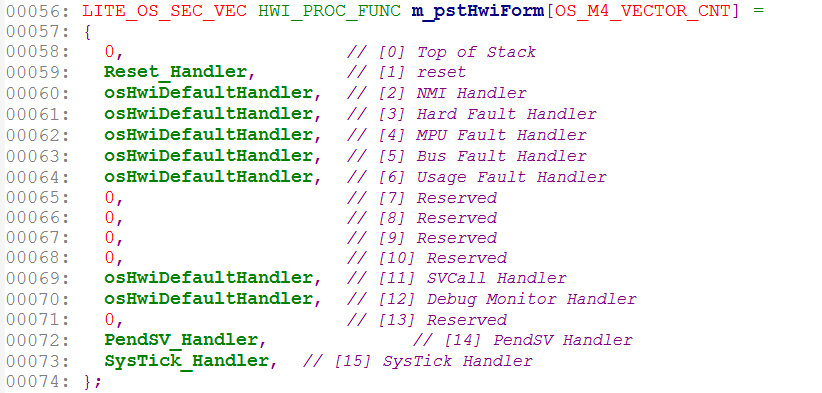
### 1.3.1 系统tick相关函数



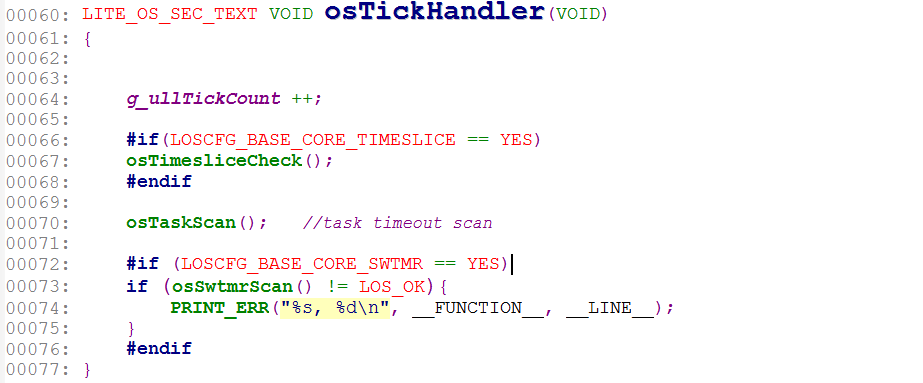
主频216MHz，tick\_per\_second = 1000表示每秒产生1000个tick中断，所以tick中断每1ms产生一次，系统tick是系统最小的时间单位，延时等待和软件定时器都依赖于系统tick。



Tick的配置和启动。



中断向量表，每1ms的tick中断产生时，执行tick的中断处理函数SysTick\_Handler( )。



中断处理函数最终调用到函数osTickHandler( )，在这个函数中完成以下任务：

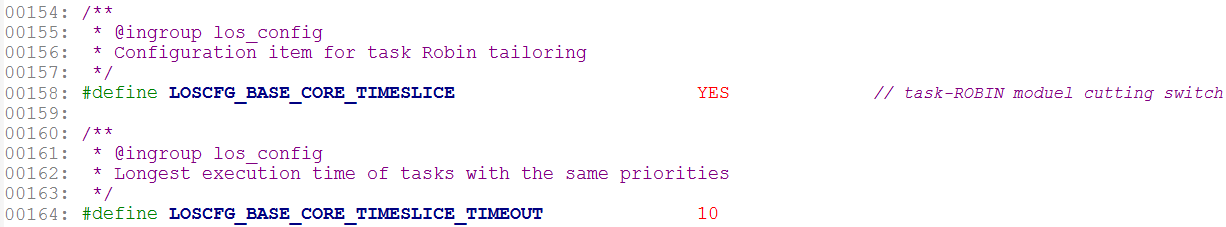
Line 67：时间片的检测及轮转调度。

Line 70：系统排序队列（超时链表）的扫描及调度。

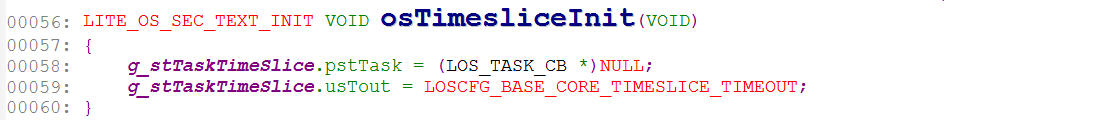
Line 73：软件定时器超时队列的扫描及处理，这部分内容在软件定时器部分说明。

### 1.3.2 时间片的检测及轮转调度

1.3.2.1 osTimesliceInit( )函数



使能系统调度时间片检测，配置时间片为10个tick。



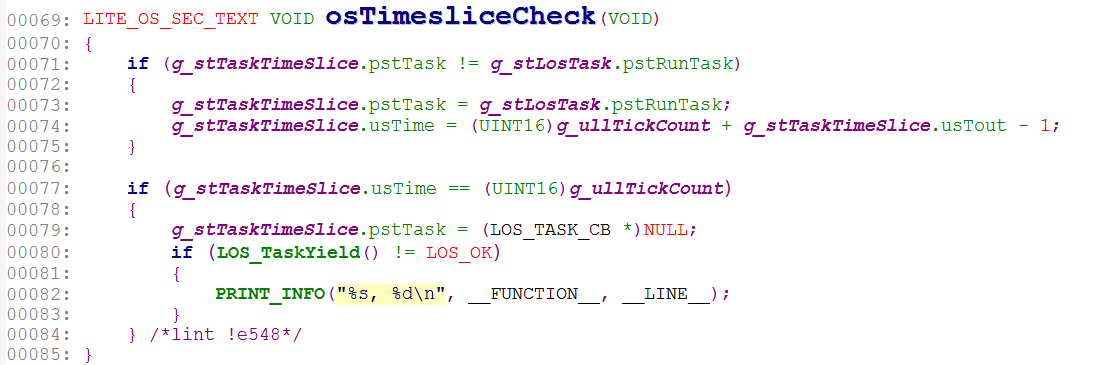
\*pstTask：指示当前正在占用时间片的TASK。

usTime：时间片的计时。

usTout：时间片的大小，以tick为单位。

初始时配置无任何TASK占用时间片，系统时间片为10个tick，即同优先级的任务需要严格按照10个tick的时间片轮转调度。

1.3.2.2 osTimesliceCheck( )函数



Line 71 - 75：时间片检测时，若当前正在运行的TASK与当前占用时间片的TASK不一致，则需要设置当前占用时间片的TASK为当前正在运行的TASK并为其分配时间片。

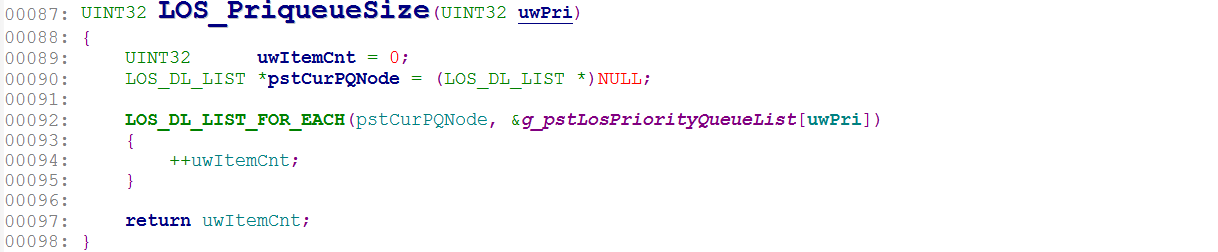
Line 74：为TASK重新分配时间片时有一个减1个tick的动作，减1的原因：同优先级之间按照时间片轮转的任务切换的工作在LOS\_TaskYield( )函数中完成，该函数在分配时间片后面执行，所以为新的TASK分配时间片需要等到下一个tick中断才可以，这时该TASK已经运行了一个tick时间了，所以需要少分配一个tick的时间片。其他抢占任务的切换在任何时间都有可能发生，调度发生以后同样需要等到下一个tick中断才能为其分配时间片，分配时间片时该TASK已经运行的时间在0到1个tick之间，所以直接减1，这样该TASK最后得到的时间片在9-10个tick之间，粗略按照10个tick计算。

Line 77 - 84：g\_stTaskTimeSlice.usTime == (UINT16)g\_ullTickCount表示当前正在占用时间片的TASK的时间片已经用完了，这时需要严格按照时间片轮转的调度算法调度与该TASK同优先级的TASK执行。如果有同优先级的TASK等待调度，则产生任务切换调度新的TASK执行并在下一个tick中断中为新的TASK分配时间片，如果没有同优先级的TASK存在，则不产生任务切换，CPU继续执行当前的这个任务，同样在下一个tick中断时为这个TASK再次分配时间片（这就是语句g\_stTaskTimeSlice.pstTask = (LOS\_TASK\_CB \*)NULL的用意，即便仍然执行这个TASK，也能正确的分配出时间片）。也就是说这个任务将一直执行下去，除非有同优先级或更高优先级的任务打断。

1.3.2.3 LOS\_TaskYield( )函数

这个函数的主要作用是调度与当前正在执行的TASK同优先级的TASK，以保证同优先级TASK之间调度的公平性。





Line 1008：调用函数LOS\_PriqueueSize( ) 获取当前正在运行的TASK对应的优先级队列中的TASK的个数。

Line 1009：uwTskCount > 1 说明当前优先级队列中除了正在运行的这个TASK以外还有其它同优先级的TASK等待调度，故需要严格按照时间片轮转的调度算法依次调度这些同优先级的TASK执行。将当前正在运行的TASK从优先级队列的队头取下，并设置READY状态，然后将其加入到优先级队列的队尾等待下次调度。

Line 1018：若uwTskCount == 1，说明当前优先级队列中只有正在运行的这个TASK，则不再产生任务切换（因为系统每次都调度最高优先级的TASK执行，所以这个TASK也是目前优先级队列中优先级最高的任务），返回后CPU继续执行这个TASK，下一次的时间片也将继续分配给这个TASK，这个TASK将一直执行下去，除非有同优先级的任务需要切换或者更高优先级的任务打断。

### 1.3.3 cortex-m4寄存器组简单说明

寄存器组包括通用寄存器和特殊控制寄存器两大类：

通用寄存器为32位寄存器，包括R0 - R15共15个寄存器。其中R13用作堆栈指针SP，R14用作连接寄存器LR，R15用作程序计数器PC。

特殊控制寄存器有三组，分别是程序状态寄存器PSR，中断屏蔽寄存器组PRIMASK、FAULTMASK和BASEPRI，控制寄存器CONTROL。特殊功能寄存器必须通过MRS/MSR指令来访问。

1.3.3.1 通用寄存器R0 - R15

R0 - R7：低组寄存器，所有指令都可以访问，大小为32位，复位后初始值不定。

R8 - R12：高组寄存器，只有部分的16位Thumb指令可以访问，32位Thumb-2指令不受限制，大小为32位，复位后初始值仍然不定。

R13：堆栈指针SP，有两个堆栈指针，分别是主堆栈指针MSP和进程堆栈指针PSP，MSP和PSP共用R13，MSP和PSP不能同时访问，即将哪一个装入R13，就访问哪一块区域。

MSP是上电复位后默认的堆栈指针（将MSP装入R13中），一般由OS内核、异常服务例程以及所有需要特权访问的应用程序代码使用，PSP用于常规的应用代码（即不处于异常服务例程中时）。

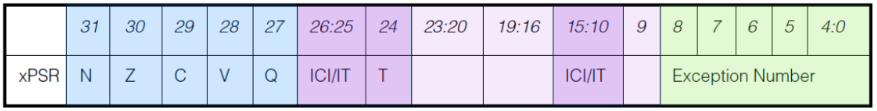
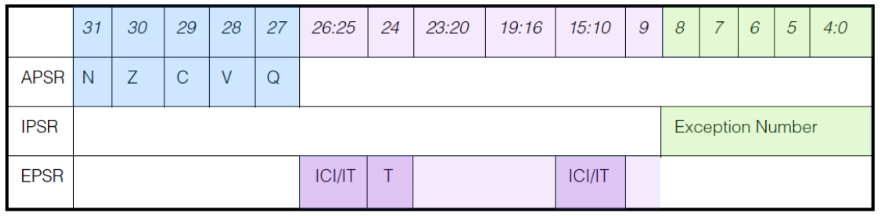
当引用R13/SP时，引用的是当前正在使用的那一个，另一个必须使用MRS/MSR指令来访问。如当前正在使用PSP（R13中装的是PSP），则对MSP必须使用MRS/MSR指令来访问。

R14：连接寄存器LR，主要作用是保存子程序的返回地址，以便在执行完子程序时恢复现场。当调用一个子程序时，由R14存储返回地址，不像大多数其它处理器，ARM为了减少访问内存的次数，把返回地址直接存储在寄存器中。这样足以使很多只有1级子程序调用的代码无需访问栈内存即可直接从R14处返回，从而提高了子程序调用的效率。如果多于 1 级，则需要把前一级的R14值压到堆栈里。

R15：程序计数器PC。

1.3.3.2 程序状态寄存器PSR

PSR包括应用程序PSR（APSR），用于记录ALU的标志，包括0标志，负数标志，进位标志，溢出标志等。中断号PSR（IPSR），用于指示当前正在服务的中断号。执行PSR（EPSR），用于指示当前的执行状态。可单独访问，也可组合访问，三合一方式访问时需使用名字xPSR或PSR。

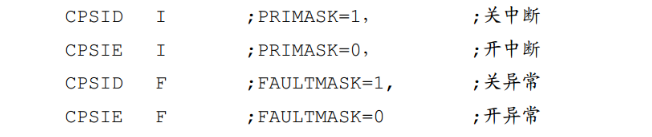


1.3.3.3 中断屏蔽组寄存器PRIMASK、FAULTMASK和BASEPRI

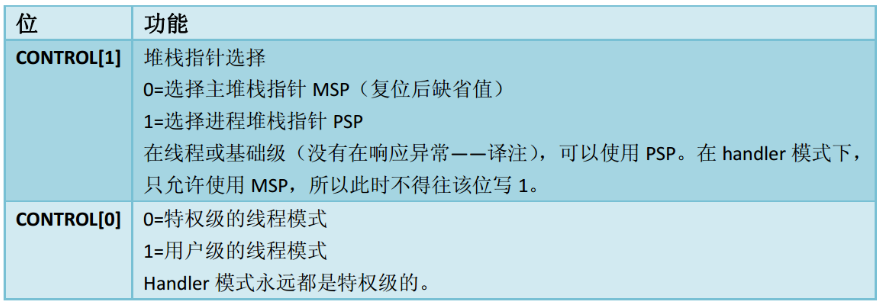
说明如下：



另外，为了快速开关中断及异常，使用如下CPS指令



1.3.3.4 控制寄存器CONTROL



说明：

CONTROL[1]：在Handler模式下该位总是0，在线程模式中可以是0，也可以是1。即Handler模式只能使用MSP，线程模式可以使用MSP，也可以使用PSP。仅当处于特权级的线程模式下时此位才可写。

MSP是系统默认分配的一块栈空间，在系统复位以后从中断向量表的起始处读取4字节的MSP地址，赋给R13。然后紧接着从reset向量处再读取4字节，赋给PC，指示系统复位后要执行的第一条指令。然后CPU跳转至第一条指令处执行。

PSP一般指向堆内存池中为每一个TASK分配出来的一块连续空间。

CONTROL[0]：仅当在特权级下操作时才允许写该位，一旦进入用户级，唯一返回特权级的途径，就是触发一个中断，再由中断服务程序改写。

特权级和非特权级的区别：特权级模式可以访问系统的所有资源，非特权级模式对一部分系统资源没有访问权限，要访问这部分资源，必须切换为特权级模式。

默认值：系统复位后CONTROL[1]和CONTROL[0]的值默认都是0，即系统工作在特权级线程模式下，使用主堆栈指针MSP。

在LiteOS中，启动系统调度前系统工作在内核态，使用默认的线程模式和堆栈指针。启动调度后系统工作在用户态，线程模式不变，仍使用特权级的线程模式，但使用进程堆栈指针PSP。这样做的好处是系统不需要在特权级和非特权级模式之间来回切换，坏处是用户态的任务拥有特权级模式，可以访问系统的所有资源，使用资源不当有可能会对内核或其它任务造成破坏。具体参见LOS\_StartToRun( )函数。

### 1.3.4 PEND\_SV中断说明

PendSV中断是为系统设备而设的可悬挂请求，通常用于系统调度的任务切换。此处只说明中断产生时现场入栈和出栈的问题，不涉及任务切换。

中断产生时，CPU会自动将一部分寄存器压入当前R13（PSP）所指示的栈中，然后将R13的指向由PSP切为MSP，用于执行handler模式。然后执行PEND\_SV中断处理函数PendSV\_Handler( )。在中断处理函数中需要将另一部分寄存器信息手动压入当前PSP（不是R13）所指示的栈中，然后处理中断。中断处理函数退出前，需要将手动压入栈中的寄存器再次手动从栈中弹出去，然后再退出中断处理函数，中断处理函数退出后先将R13的指向由MSP切为PSP，然后CPU自动弹出调用中断处理函数之前自动压入栈中的寄存器。

CPU自动将一部分寄存器入栈而不是全部寄存器入栈的原因：中断处理函数需要用到寄存器，所以需要将寄存器入栈，但是一般不会用到所有的寄存器，所以只需要入栈一部分。故CPU自动入栈一部分，中断处理函数中如果寄存器不够的话需要手动入栈另外的部分。

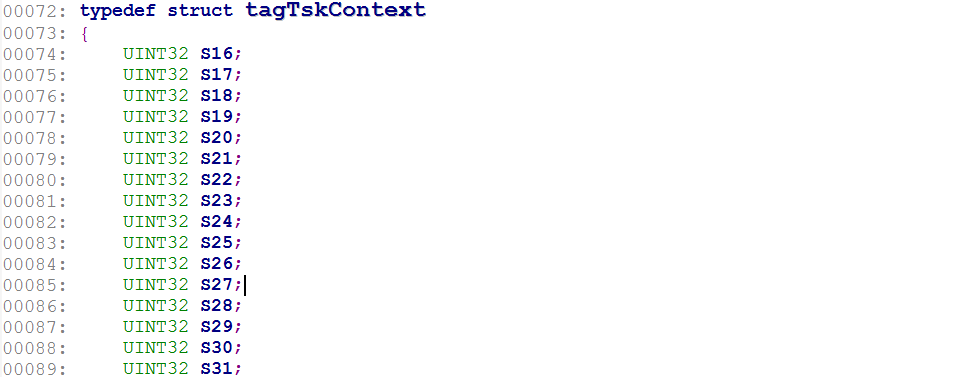
有些CPU有FPU浮点处理器单元，对应有一些浮点寄存器，与FPU相关的寄存器包括32位浮点寄存器S0 - S31，浮点状态寄存器FPSCR。S0 - S31也可以按照64位双精度浮点寄存器D0 - D15访问。有些CPU没有FPU，同样也就没有浮点寄存器。这两种情况下中断产生时入栈和出栈的寄存器顺序是不一样的。

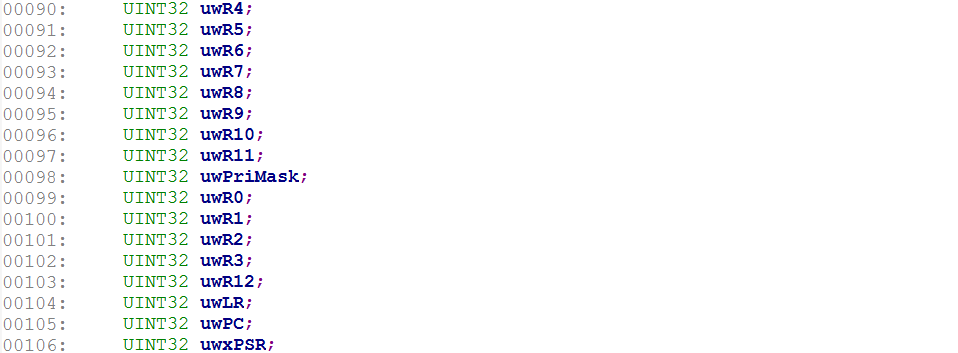
第一种情况，CPU无FPU单元：中断产生时CPU自动入栈的寄存器的顺序是：【xPSR - PC - LR - R12 - R3 - R2 - R1 - R0】，共8个寄存器，PEND\_SV中断处理函数中需要将【R11 - R4】依次入栈。中断处理函数退出前，手动将【R4 - R11】依次弹栈，中断退出后由CPU自动弹出【R0 - xPSR】至对应的寄存器中，恢复进入中断前的现场继续执行。

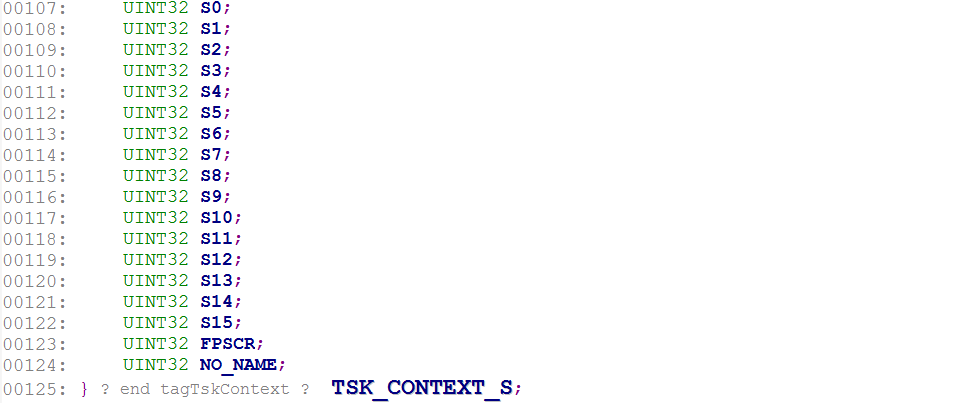
第二种情况，CPU有FPU单元：这种情况下CPU会将部分FPU的寄存器也一起入栈，包括一个未知的字（下文以NO\_NAME代替），一个FPSCR寄存器，以及S15 - S0。剩下的S31 - S16需要在中断中手动入栈。此时当中断产生时CPU自动入栈的寄存器的顺序是：【NO\_NAME - FPSCR - S15 - S14 - S13 - ... - S2 - S1 - S0 - xPSR - PC - LR - R12 - R3 - R2 - R1 - R0】，PEND\_SV中断处理函数中需要将剩下的【R11 - R4，S31 - S16】依次入栈。中断处理函数退出前，手动将【S16 -- S31，R4 - R11】依次弹栈，中断退出后由CPU自动弹出【R0 - NO\_NAME】至对应的寄存器中，恢复进入中断前的现场继续执行。

cortex-m0和cortex-m3没有FPU单元，cortex-m4和cortex-m7有FPU单元，所以他们的栈的指示和任务切换的指令序列是不一样的。

在用户创建TASK时会初始化TASK的栈空间，初始化栈时有一个结构体TSK\_CONTEXT\_S，这个结构体中的成员刚好是PEND\_SV中断执行时的全部入栈顺序，初始化栈空间时会从栈底位置开始依次初始化这些值，主要是为了给出一组默认的TASK的执行现场，用于任务切换时启动TASK执行，具体过程参见调度汇编语言解析。结构体如下：







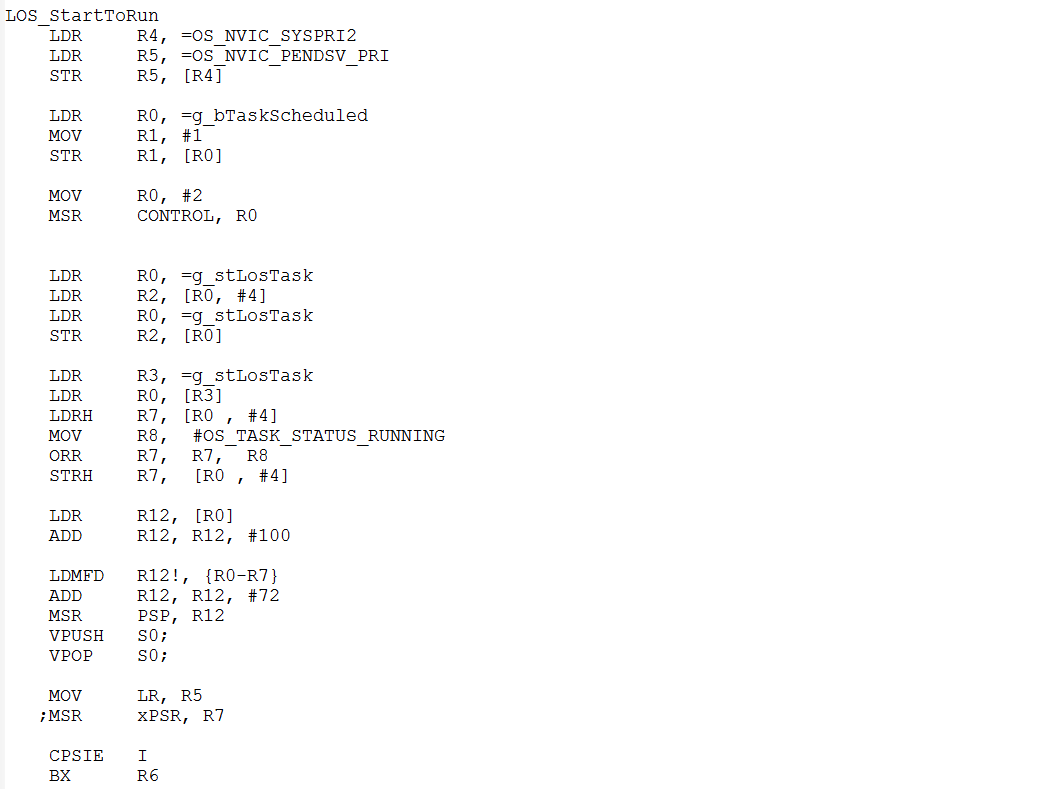
比较发现，这个顺序比前面的入栈顺序在Line 98行多了一个uwPriMask，原因是在

TaskSwitch中R12寄存器多操作了一次。按照正常的入栈顺序，进入中断处理函数后只需手动将R11 - R4入栈即可，但是此处也将R12入栈了，将R12再次入栈的原因是在调用TaskSwitch之前R12保存了PRIMASK的值，所以需要一起入栈。所以结构体中需要多出来一个成员表示这个R12，就是uwPriMask。



### 1.3.5 系统调度的启动：LOS\_StartToRun( )函数

CPU在调用并执行函数LOS\_StartToRun( )之后，才能启动系统调度，通过PENDSV中断的方式切换不同的TASK执行。在此之前，CPU的执行是单任务的模式，CPU使用的栈是系统默认分配的栈空间，是MSP，也就是主堆栈。TASK运行时使用的栈是从动态内存池中分配出的一段连续空间，是PSP，也就是进程堆栈。这两种工作方式下使用的栈空间是不一样的。



代码解析如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CODE START \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LOS\_StartToRun完成由内核态切入到用户态的功能。在此之前，CPU工作在内核态，使用内核态的堆栈空间MSP，内核态的堆栈空间在执行main函数之前就已经分配好了，在此之前，可以认为系统只有一个处于内核态的TASK在运行，这个TASK完成一些列的系统初始化工作，比如系统时钟的初始化等。也包括用户态相关内容的初始化，比如TASK的创建，内存池、信号量、消息队列的初始化等。

该函数执行时，特权级线程模式不变，堆栈指针切为用户态堆栈指针PSP，然后CPU转入用户态中目前优先级最高的那个TASK执行，其它处于用户态的TASK都由系统调度的任务切换来启动执行（其它TASK的第一次启动在切换任务中描述）。至此以后CPU一直处于用户态，当有异常或中断或必须由内核态来处理的工作时，CPU重新再切回内核态处理，处理完成以后再返回用户态继续执行用户态的任务。

LOS\_StartToRun

LDR R4, =OS\_NVIC\_SYSPRI2 //OS\_NVIC\_SYSPRI2 = 0xE000ED20

LDR R5, =OS\_NVIC\_PENDSV\_PRI //OS\_NVIC\_PENDSV\_PRI = 0xF0F00000

STR R5, [R4] //将PENDSV中断优先级的值写入到中断优先级寄存器中

/\* 以上设置PENDSV中断的优先级 \*/

LDR R0, =g\_bTaskScheduled //变量g\_bTaskScheduled的地址装入R0

MOV R1, #1 //立即数1装入R1

STR R1, [R0] //将变量g\_bTaskScheduled置1，至此表示系统调度已经启动

/\* 以上设置变量g\_bTaskScheduled的值为1，表示系统调度启动 \*/

MOV R0, #2 //立即数2装入R0

MSR CONTROL, R0 //将2写入控制寄存器CONTROL。表示R13用作进程堆栈指针，在此之前用作主堆栈指针。线程模式不变，仍为特权级线程模式。更改后，R13指向系统默认的PSP，默认的PSP是一个不能访问的地址值，若使用则会更改里边原来保存的信息，其结果不可预知。

/\* 以上切换堆栈指针，R13的指向由MSP切为PSP \*/

LDR R0, =g\_stLosTask //将g\_stLosTask的地址装入R0

LDR R2, [R0, #4] //将g\_stLosTask中的pstNewTask指针装入R2中

LDR R0, =g\_stLosTask //将g\_stLosTask的地址装入R0

STR R2, [R0] //将pstNewTask指针赋值给pstRunTask指针，表示当前正在运行的TASK变成pstNewTask

/\* 以上设置pstRunTask，以使后续的任务切换可以正常进行 \*/

LDR R3, =g\_stLosTask //将g\_stLosTask的地址装入R0

LDR R0, [R3] //将g\_stLosTask中的pstRunTask的地址装入R0

LDRH R7, [R0 , #4] //将pstRunTask的TCB中变量usTaskStatus的值装入R7

MOV R8, #OS\_TASK\_STATUS\_RUNNING //将RUNNING表示的立即数装入R8

ORR R7, R7, R8 //设置pstRunTask的RUNNING状态，表示正在运行

STRH R7, [R0 , #4] //将设置RUNNING后的状态的值写回到pstRunTask的TCB的usTaskStatus变量中

/\* 以上设置pstRunTask的RUNNING状态，表示正在运行 \*/

LDR R12, [R0] //将pstRunTask指向的TCB中的pStackPointer指针装入R12，此时pStackPointer指向栈空间的pstContext处

ADD R12, R12, #100 //pstRunTask的栈向下偏移100字节，指向pstContext->uwR0处

LDMFD R12!, {R0-R7} //将R12指示的地址开始的值依次装入R0 - R7，并将新的栈指针重新装入R12。即将pstContext->uwR0到pstContext->uwxPSR的值依次装入R0 - R7

/\* 以上从TASK初始的栈空间中取出TASK的初始信息 \*/

ADD R12, R12, #72 //pstRunTask的栈再向下偏移72字节，指向栈底位置

MSR PSP, R12 //将栈底位置的指针装入PSP，即用户级线程的栈空间从栈底开始访问，将栈初始化时保存的一个TSK\_CONTEXT\_S结构体的信息清除掉。此时R13由默认的PSP指向用户TASK的PSP。

VPUSH S0; //S0寄存器的值入栈

VPOP S0; //S0寄存器的值出栈，这两句话应该无实际意义，只是为了测试栈空间而存在

/\* 以上清除TASK的栈空间中保存的初始信息，将栈底位置赋给PSP，因为在前面R13已经指向了PSP，所以此处更改PSP的值会同步更改R13的指向 \*/

MOV LR, R5 //将osTaskExit( )函数的地址赋值给LR，表示默认的返回值

;MSR xPSR, R7 //;;;;;;;;;;将uwxPSR的初始值装入xPSR寄存器中

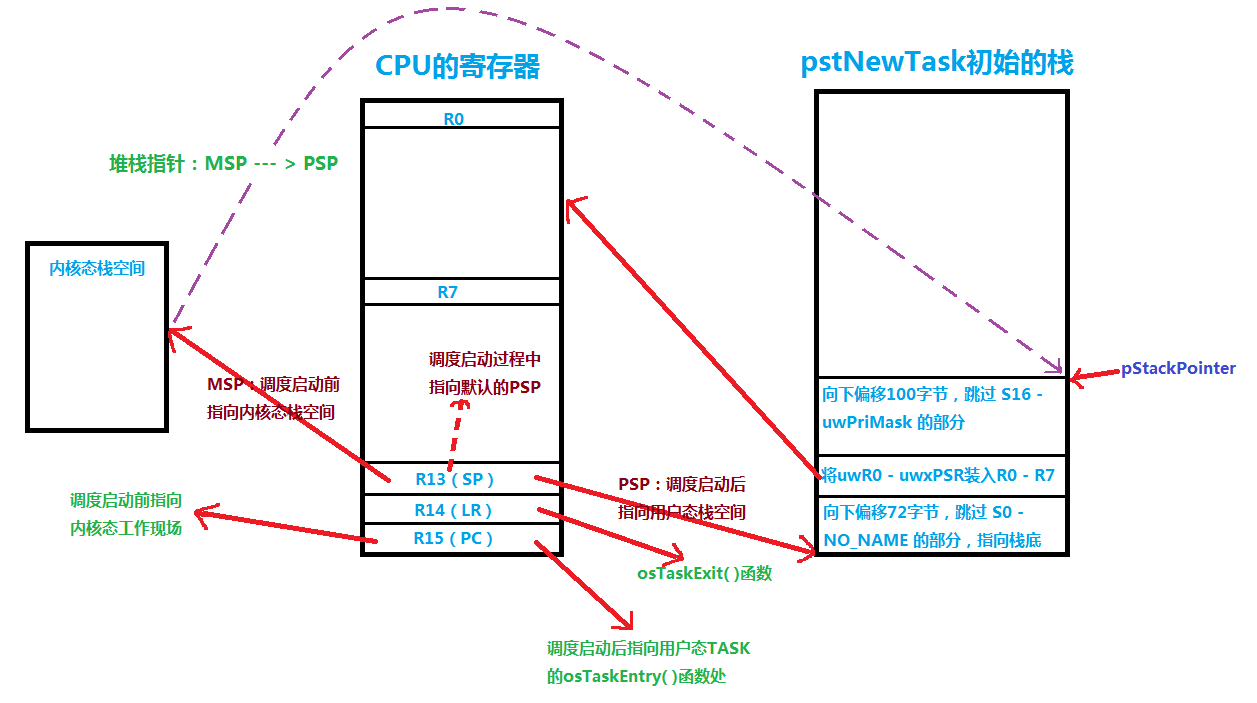
CPSIE I //使能全局中断

BX R6 //跳转至函数osTaskEntry( )处运行，执行TASK的工作函数

/\* 以上设置R14（LR）和R15（PC）\*/

至此，用户态第一个TASK的启动工作完成，CPU离开内核态，进入用户态运行。该TASK有用的初始信息（保存于自己的栈中）已经被CPU提取完毕，栈中的初始信息已被销毁。R13（PSP）指向TASK的栈底位置。R15（PC）指向osTaskEntry( )函数，其中R0用于传入osTaskEntry( )函数的参数uwTaskID。R14（LR）指向默认的TASK退出函数osTaskExit( )。

示意图如下：



/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CODE END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 1.3.6 系统调度的实现：LOS\_Schedule( )和osTaskSchedule( )函数

LiteOS内核任务调度基于高优先级抢占的时间片轮转调度算法，时间片基于系统tick产生。当有TASK需要被调度时，需将该TASK加入到对应的优先级队列中，并启动调度，每次调度都从优先级队列中选择优先级最高的TASK，同优先级之间严格按照时间片进行轮转调度，高优先级TASK强制抢占处理器，而不管正在执行的低优先级TASK的时间片是否结束，以达到系统实时响应的目的。

当正在运行的TASK因为某种原因需要进入阻塞状态时，该TASK会将自己从优先级队列中移除，移除后该TASK不再参与调度，并设置等待条件同时将该TASK加入相对应的阻塞队列中（比如超时等待会加入到排序队列中，读取消息队列阻塞会加入到消息阻塞队列中），每当处理系统tick中断时都会检测排序队列中的TASK的状态（排序队列中都是与时间延时有关的任务，所以需要在每个tick中断中都检测），如果有状态满足被调度条件的TASK，则将该TASK重新加入到优先级队列的尾部进而参与调度，其它的阻塞状态会被相应的阻塞条件唤醒进而参与系统调度（比如读消息队列阻塞会被写消息队列函数唤醒）。

所有调度相关的队列都采用双循环链表数据结构实现，以便能够快速的在队尾插入。

1.3.3.1 LOS\_Schedule( ) 函数

调用该函数启动一次任务切换。



Line 73：系统调度时需要新调入的TASK是当前优先级队列中优先级最高的那个TASK。

Line 76：优先级队列中优先级最高的那个TASK与当前正在运行的TASK相比较，这两个不相等的情况只有三种，第一种情况是正在执行的TASK为NULL，需要新调入TASK来执行。第二种情况是同优先级的TASK被时间片切换调度。第三种情况是新得到的TASK的优先级比当前正在执行的TASK的优先级高，需要紧急处理。

即便有低优先级的TASK在排队等待，只要高优先级的TASK没有离开优先级队列，那么低优先级的TASK就没有被调度到的可能。

如果高优先级的TASK长时间不进入阻塞状态而被调出，那么时间片一直会分配给高优先级的TASK，那些低优先级的TASK就会因为无法被调度而长时间处于饥饿状态，甚至被饿死。

Line78：若调度被锁，即便有新的TASK需要调度，也不产生任务切换，等到释放调度锁时再产生任务切换。

LIne82：如果需要任务切换，则调用函数osTaskSchedule( ) 来完成任务的切换，当该函数结束时，CPU调度新的TASK执行。

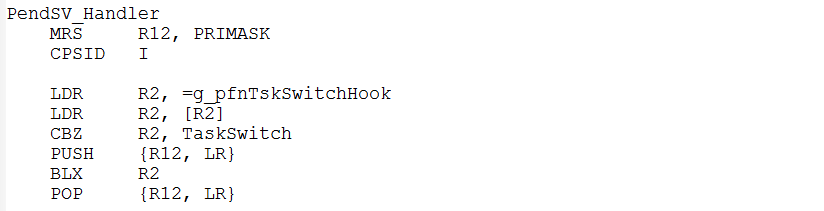
1.3.3.2 osTaskSchedule( )函数



将OS\_NVIC\_PENDSVSET的值装入OS\_NVIC\_INT\_CTRL寄存器中，使系统产生PENDSV中断。

BX LR：返回到调用处，即程序返回到调用osTaskSchedule的地方继续执行，只是此处设置了PENDSV中断，所以返回后会执行PENDSV中断的处理函数。

系统调度通过产生比tick中断更高优先级的PENDSV中断来完成，触发PENDSV中断后CPU执行中断处理函数PendSV\_Handler( )。



PENDSV中断中最后调用TaskSwitch来完成任务切换，PENDSV中断退出后CPU执行新的TASK。

代码解析：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CODE START \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**中断中MSP与PSP的切换过程如下：**

在PENDSV中断产生之前，R13中保存的是PSP，指向PSP。当PENDSV中断产生后，CPU自动保存一些寄存器的信息至R13所指向的栈中，保存完后将MSP赋给R13，此时R13指向MSP，然后执行PendSV\_Handler，即中断处理函数的执行全过程中R13均指向MSP。

中断处理函数执行过程中R13一直指向MSP，不会更改，中断中所有的栈操作都是操作MSP。PendSV\_Handler中需要用到PSP来切换任务，而PSP此时不在R13中，所以需要通过MRS指令将PSP装到R0中，这样R0就指向了PSP，由R0来完成对用户任务栈的操作。任务切换完成以后需要通过MSR将新的栈指针从寄存器R1中装入到PSP中，这样PSP就指向了新的TASK的栈，但此时R13仍指向MSP，并不指向新的TASK的栈。

中断处理函数执行完毕，中断退出前，CPU先将R13的指向由MSP改为PSP，这样R13就指向了新的任务的栈空间，然后CPU自动恢复此时R13指向的栈中的寄存器信息，也就是恢复新的TASK的现场，然后中断退出，CPU执行新的TASK，R13重新指向PSP。

PendSV\_Handler

MRS R12, PRIMASK //状态寄存器PRIMASK（只有一个位的寄存器，用于开关全局中断）的值暂存在R12中，因为下面要关全局中断，会设置PRIMASK寄存器，所以需要先将该寄存器的值保存下来，最后直接执行指令MSR PRIMASK, R12可以恢复该寄存器的值，同时也会开启全局中断

CPSID I //关全局中断，任务切换过程不允许被除了NMI和硬件FAULT中断以外的任何中断打断

LDR R2, =g\_pfnTskSwitchHook //将钩子函数的函数指针装入R2

LDR R2, [R2] //将用户指定的钩子函数的地址装入R2

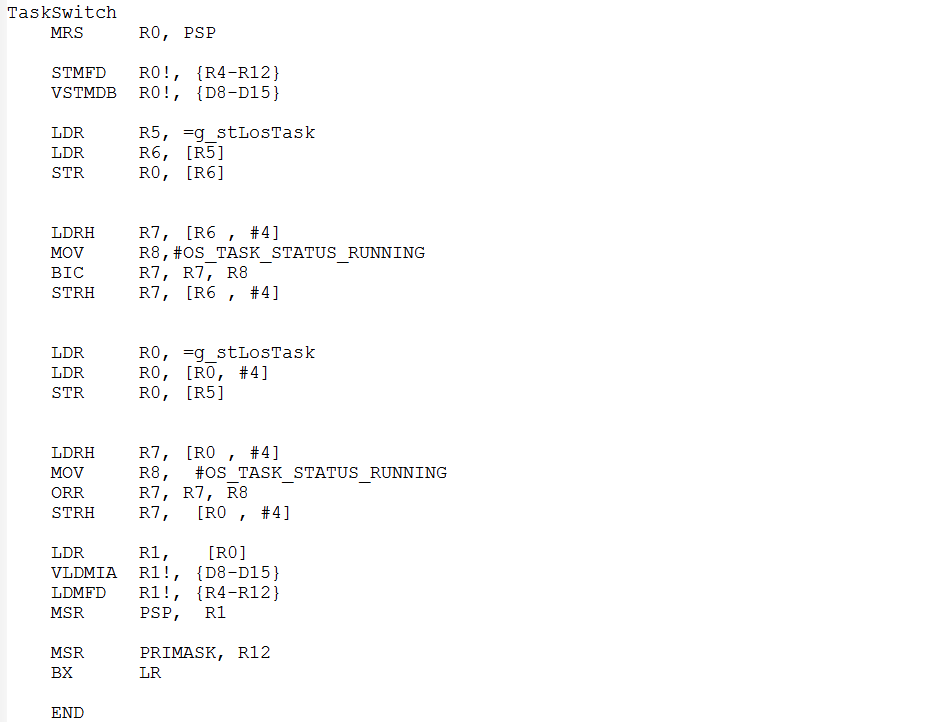
CBZ R2, TaskSwitch //若R2中的值为0，则跳转至TaskSwitch处执行，否则继续向下执行，不产生任务切换，即钩子函数和任务切换函数只能执行其中一个

PUSH {R12, LR} //将R12和LR的值入当前正在运行的TASK的栈

BLX R2 //跳转至用户钩子函数处执行

POP {R12, LR} //用户钩子函数执行完毕，R12和LR出栈，恢复现场继续执行

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CODE END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/



TaskSwitch用于完成任务的切换，代码解析如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CODE START \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

TaskSwitch

MRS R0, PSP //将pstRunTask的栈指针PSP装入R0。在此之前，R13已经被装入了MSP，所以对PSP的访问需要通过MRS指令将其装入到R0中，这时R0就代替原来的R13指向了pstRunTask的栈。

STMFD R0!, {R4-R12} //将R4-R12的值保存至pstRunTask的栈中，并将新的栈指针写回到R0中。

VSTMDB R0!, {D8-D15} //将D8-D15的值保存至pstRunTask的栈中，并将新的栈指针写回到R0中。

/\* 以上代码：执行PendSV\_Handler之前，CPU已经将pstRunTask的一些寄存器信息【NO\_NAME - FPSCR - S15 - ... - S0 - xPSR - PC - LR - R12 - R3 - R2 - R1 - R0】保存在了pstRunTask的栈中，并且使PSP指向当前的栈顶。并且切换了栈指针使R13指向MSP。在PendSV\_Handler中断处理函数中，需要将剩下的寄存器【R11 - R4，S31 - S16】保存在pstRunTask的栈中，并使R0指向当前的栈顶。此处多保存了一次R12，是因为R12在调用TaskSwitch之前保存了PRIMASK的值，需要将其压入栈中，等下次调度这个TASK时根据R12中的值来恢复PRIMASK的值。S31 - S16是按照双精度浮点寄存器D15 - D8保存的。 \*/

LDR R5, =g\_stLosTask //将g\_stLosTask的地址装入R5

LDR R6, [R5] //将g\_stLosTask中的pstRunTask指针装入R6中

STR R0, [R6] //将pstRunTask的当前栈指针赋给TCB的\*pStackPointer

/\* 以上代码：将pstRunTask当前的栈指针PSP赋给pstRunTask对应的TCB的指针pStackPointer，当下次调度到这个TASK时CPU根据pStackPointer保存的PSP恢复现场 \*/

LDRH R7, [R6 , #4] //将pstRunTask的TCB中变量usTaskStatus的值装入R7

MOV R8,#OS\_TASK\_STATUS\_RUNNING //将RUNNING表示的立即数装入R8

BIC R7, R7, R8 //清除pstRunTask的RUNNING状态，表示停止运行

STRH R7, [R6 , #4] //将清除RUNNING后的状态的值写回到pstRunTask的TCB的usTaskStatus变量中

/\* 以上代码：清除pstRunTask的RUNNING状态，表示pstRunTask停止运行 \*/

/\* 以上步骤是任务切换的第一部分，将pstRunTask指示的当前正在运行的TASK调出，主要完成以下工作：一是保存CPU在进入中断处理函数之前未自动保存的pstRunTask的寄存器信息到pstRunTask的栈中。二是将pstRunTask的当前的栈指针赋给pstRunTask的TCB的指针pStackPointer，以便下一次调度该TASK时恢复现场使用。三是将pstRunTask的RUNNING状态清除，表示该TASK停止运行。这样，后面会在PSP中装入pstNewTask的pStackPointer，中断处理程序返回后，CPU会从pstNewTask的栈中自动恢复现场，进而达到任务切换的目的 \*/

LDR R0, =g\_stLosTask //将g\_stLosTask的地址装入R0

LDR R0, [R0, #4] //将g\_stLosTask中的pstNewTask指针装入R0中

STR R0, [R5] //将pstNewTask指针赋值给pstRunTask指针，表示当前正在运行的TASK变成pstNewTask

/\* 以上代码：将pstNewTask赋给pstRunTask，切换当前正在运行的TASK \*/

LDRH R7, [R0 , #4] //将pstNewTask的TCB中变量usTaskStatus的值装入R7中

MOV R8, #OS\_TASK\_STATUS\_RUNNING //将RUNNING表示的立即数装入R8

ORR R7, R7, R8 //设置pstNewTask的RUNNING状态，表示正在运行

STRH R7, [R0 , #4] //将设置RUNNING后的状态的值写回到pstNewTask的TCB的usTaskStatus变量中

/\* 以上代码：设置pstNewTask的RUNNING状态，表示pstNewTask正在运行 \*/

LDR R1, [R0] //将pstNewTask的当前的栈指针pStackPointer装入R1中。

VLDMIA R1!, {D8-D15} //将R1指示的地址开始的值恢复至D8-D15寄存器，并将新的栈指针重新装入R1中

LDMFD R1!, {R4-R12} //将R1地址开始的值恢复至R4-R12寄存器，并将新的栈指针重新装入R1中

MSR PSP, R1 //将pstNewTask的当前栈指针装入PSP中，这样当中断处理函数执行完毕中断退出前，CPU会自动将R13的指向由MSP切为PSP，即R13指向pstNewTask的栈。

/\* 以上代码：从pstNewTask的PSP指示的栈顶开始，依次手动弹出寄存器【S16 - S31，R4 - R12】，并将最新的栈指针赋给PSP，R12用于在后面恢复PRIMASK的值，其余寄存器的值是该TASK在前一次被调出时手动保存的，需要手动恢复。等中断处理函数退出之后CPU自动恢复寄存器【R0 - R1 - R2 - R3 - R12 - LR - PC - xPSR - S0 - ... S15 - FPSCR - NO\_NAME】。 \*/

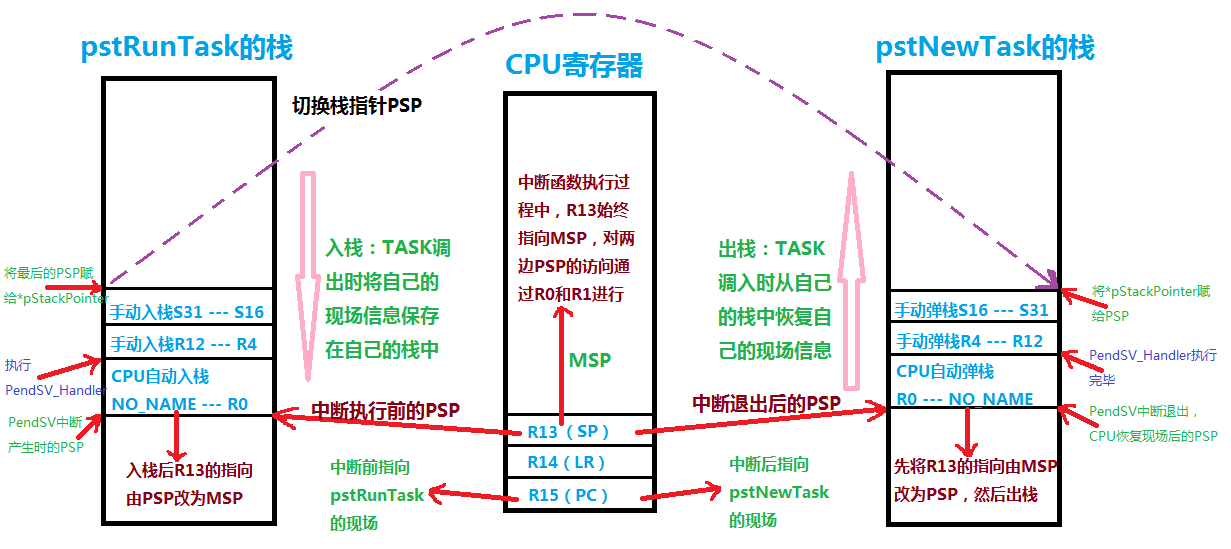
/\* 以上步骤是任务切换的第二部分，调入pstNewTask指示的新的TASK执行，主要完成以下工作：一是将pstNewTask赋给pstRunTask，设置新的正在运行的TASK。二是设置pstNewTask的RUNNING状态，表示正在运行。三是从pstNewTask的栈中恢复该TASK上次被调出时手动保存的寄存器的信息。四是将pstNewTask的栈指针装入PSP中，这样当PendSV中断处理程序返回时，CPU自动从当前PSP指示的栈中，也就是pstNewTask的栈中恢复pstNewTask的任务现场，进而达到任务切换的目的 \*/

MSR PRIMASK, R12 //将R12中暂存的状态寄存器的值恢复至状态寄存器PRIMASK中，在刚开始执行PendSV\_Handler时保存了PRIMASK的值，这个值是对应于pstRunTask的，而此处的R12是从pstNewTask的栈中恢复的，所以这个PRIMASK的值对应于pstNewTask。

BX LR //等价于MOV PC,LR指令，将LR的值赋给PC，子程序返回，程序跳转至调用PendSV\_Handler的地方继续执行。PendSV\_Handler退出后，CPU自动从PSP指示的pstNewTask的栈中恢复现场，进而执行pstNewTask。至此，任务切换完成，CPU由执行pstRunTask转为执行pstNewTask。

END //程序结束，这句话永远不会被执行

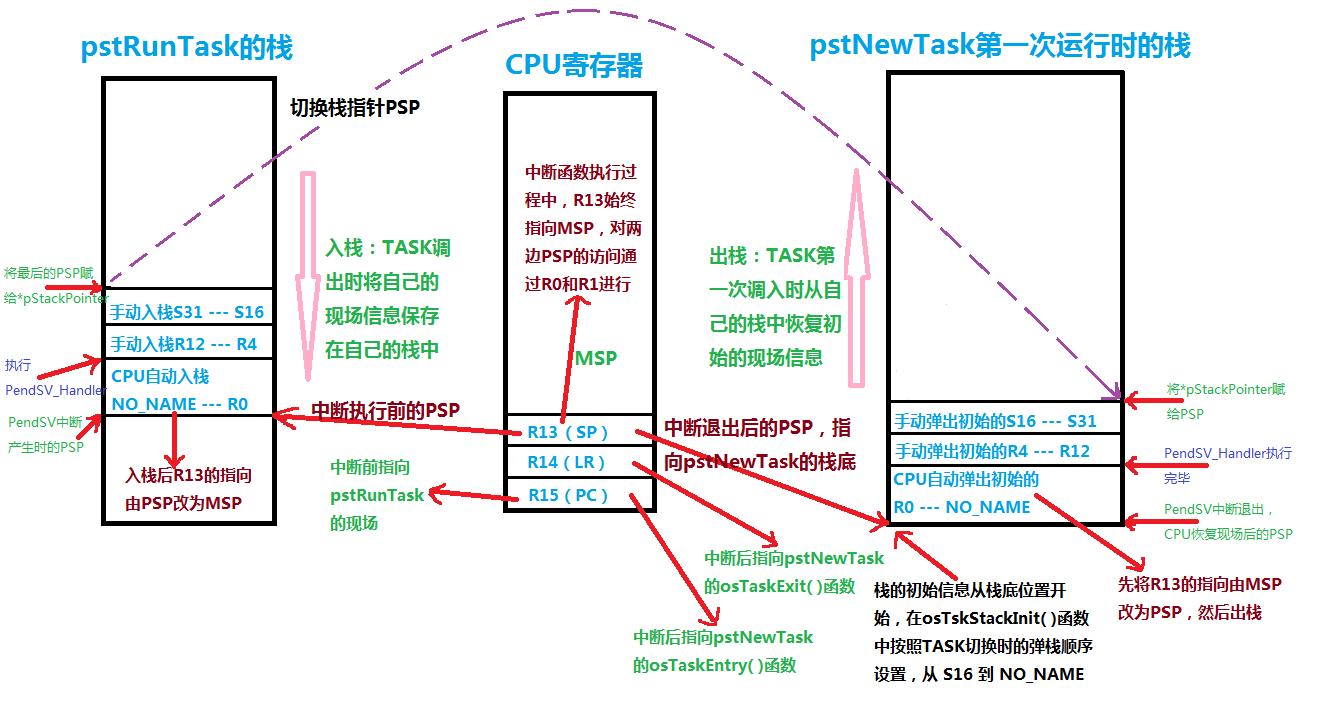
栈的示意图如下（假设pstNewTask已经被调度过）：



TASK第一次启动执行的情况：CPU由内核态切入用户态时，通过调用LOS\_StartToRun( )函数启动了用户态中优先级最高的一个TASK，剩余的TASK以及后续创建的TASK的启动都是通过系统调度最终调用TaskSwitch实现的。

TaskSwitch在切换任务的后半部分，需要将pstNewTask的现场信息依次弹栈，因该TASK以前从未执行过，故没有保存过现场信息，所以默认情况下pstNewTask没有现场信息可以恢复。为了TaskSwitch能正确启动pstNewTask，在pstNewTask的栈中需要给出一个默认的现场信息，并且这个默认的现场信息要和TaskSwitch后半部分的弹栈顺序一致。这样，CPU在启动pstNewTask时恢复其默认的现场信息，也就是TASK的初始信息，进而启动该TASK执行。该TASK启动以后，栈中保存的默认信息被销毁，后续的切换直接通过PEND\_SV中断进行。

这就是osTaskStackInit( )函数和TSK\_CONTEXT\_S结构体的作用。栈的示意图如下：



/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CODE END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

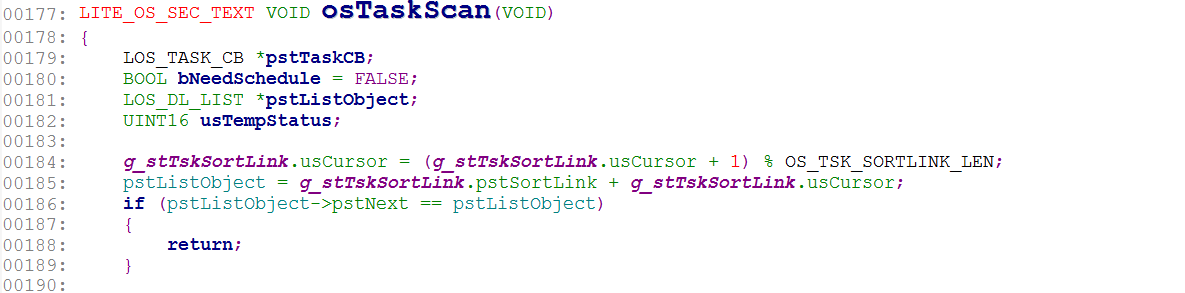
### 1.3.7 排序队列的扫描及调度：osTaskScan( )函数

排序队列中存放的都是因需要延时等待而被挂起的TASK，时间的判断需要依赖于系统tick，所以在每个tick中断时需要完成等待时间的减少以及等待时间到时对TASK状态的判断，以便在超时时间到时能唤醒TASK。

进入排序队列的TASK会有两种状态，一种是主动延时的状态OS\_TASK\_STATUS\_DELAY，另一种是等待超时的状态OS\_TASK\_STATUS\_TIMEOUT，这两种状态都会唤醒被阻塞的TASK。

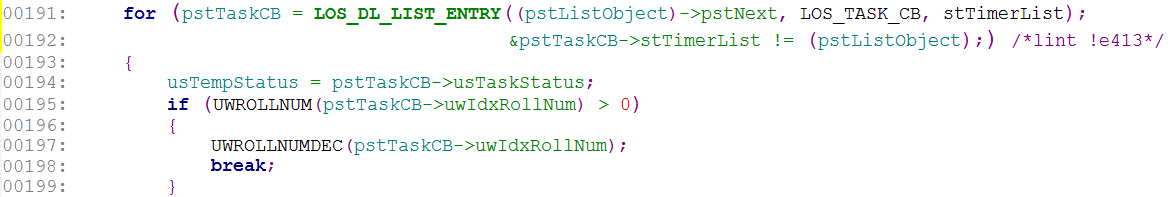
注意：在延时时间到时，DELAY状态会在扫描函数中清除，因为处于DELAY状态的TASK被唤醒以后不需要判断是哪种状态唤醒的，但TIMEOUT状态不会在扫描时清除，因为处于TIMEOUT状态的TASK一般同时还会在其它阻塞队列中，当该TASK被唤醒后需要判断是否是超时唤醒，所以TIMEOUT状态在唤醒后还需要用到，故不在此处清除。

排序队列的扫描原理：系统一共32个排序队列（由宏OS\_TSK\_SORTLINK\_LEN决定，可以更改），编号为0到31号。每一个tick中断扫描一个排序队列，将整个排序队列扫一圈需要耗时32个tick。TASK的tick延时被分解为tickCount = RollNum\*32 + RollIndex的方式，其中RollNum表示需要完整扫描的圈数，RollIndex表示不够扫描一圈的tick个数。



Line 184：每个tick中断扫描一个排序队列，循环扫描32个排序队列，每次扫描的都是当前光标的下一个位置（加入排序队列时是按照当前光标的位置加入的）。

Line 186：如果当前tick中要扫描的排序队列为空，则直接返回，下一个排序队列在下一个tick中断中扫描。



Line 191：若排序队列中有TASK存在，则需要从头开始循环扫描排序队列。

Line 195：第一个TASK的RollNum > 0，则将RollNum值减1后直接返回，不需要再扫描后面的TASK。若扫到RollNum为0的TASK，则表示该TASK的超时时间已到，需要将其从排序队列的队头删除，检查状态并唤醒，然后继续扫描，直到排序队列为空或者第一个TASK的RollNum > 0为止。

例值1：以当前usCursor为15，用户传入的延时时间为200模拟扫描过程。200 = 6\*32+8，故TASK被加入到23号排序队列中（添加时确定的），uwIdxRollNum的低27位为6。

第一圈：从16号扫到23号用7个tick，加上设置延时到下个tick中断的时间共约8个tick，到23号后RollNum = 6 > 0，故RollNum - 1 = 5。

第二圈：扫一圈，耗时32个tick，此时RollNum = 5 > 0，故RollNum - 1 = 4。

第三圈：扫一圈，耗时32个tick，此时RollNum = 4 > 0，故RollNum - 1 = 3。

第四圈：扫一圈，耗时32个tick，此时RollNum = 3 > 0，故RollNum - 1 = 2。

第五圈：扫一圈，耗时32个tick，此时RollNum = 2 > 0，故RollNum - 1 = 1。

第六圈：扫一圈，耗时32个tick，此时RollNum = 1 > 0，故RollNum - 1 = 0。

第七圈：扫一圈，耗时32个tick，此时RollNum = 0，表示延时时间已经完成，将该TASK从排序队列的队头取下。

例值2：以当前usCursor为15，用户传入的延时时间为192模拟扫描过程。192 = 6\*32+0，故TASK被加入到15号排序队列中，uwIdxRollNum的低27位为5。当uwIdx为0时，RollNum需要减1。

第一圈：从16号扫到15号用31个tick，加上设置延时到下个tick中断的时间共约32个tick，到15号后RollNum = 5 > 0，故RollNum - 1 = 4。

第二圈：扫一圈，耗时32个tick，此时RollNum = 4 > 0，故RollNum - 1 = 3。

第三圈：扫一圈，耗时32个tick，此时RollNum = 3 > 0，故RollNum - 1 = 2。

第四圈：扫一圈，耗时32个tick，此时RollNum = 2 > 0，故RollNum - 1 = 1。

第五圈：扫一圈，耗时32个tick，此时RollNum = 1 > 0，故RollNum - 1 = 0。

第六圈：扫一圈，耗时32个tick，此时RollNum = 0，表示时间已经完成，共扫了6个周期，如果不减去1，对于uwIdx为0的这种延时就需要多扫32个tick。

注意：如果同一序号的排序队列中有多个TASK，则加入时按照等待时间从短到长的顺序加入队列(确保扫描时可以只扫描队列的第一个TASK)，且当前TASK的RollNum的值是该TASK需要的总的RollNum的值与前面所有TASK的RollNum值的总和的差值。具体参见osTaskAdd2TimerList( ) 函数。所以对于同一个排序队列，只要第一个TASK的超时时间未到，那么后面所有TASK的超时时间就一定未到，故只要第一个TASK的RollNum > 0，就直接跳出返回，不再扫描后面的TASK。



当某一TASK的超时时间完成时：

Line 201：将该TASK从排序队列中删除。

Line 202 - 208：如果该TASK还处于PEND（阻塞）状态，说明该TASK还存在于其他阻塞队列中（比如读事件阻塞队列、互斥锁阻塞队列等），需要清除PEND状态并将该TASK从其所在的阻塞队列中删除。并设置pstTaskCB->pTaskSem = NULL; pstTaskCB->pTaskMux = NULL，表示该TASK不再等待任何信号量和互斥锁（TASK在进入信号量和互斥锁阻塞队列时会设置等待的信号量和互斥锁）。

Line 209：若该TASK处于EVENT状态，则清除EVENT状态（目前程序中没有设置该状态的地方，所以EVENT状态目前没有任何作用）。

Line 213 - 217：若该TASK处于PEND\_QUEUE状态，说明该TASK还存在于消息阻塞队列中（读阻塞或写阻塞队列），需要将其从消息阻塞队列中清除。

Line220：如果前面的几种阻塞状态都不存在，则说明该TASK只是处于DELAY状态，则只需要清除DELAY状态即可。

PEND、EVENT、PEND\_QUEUE、DELAY这四种阻塞条件是互斥的。

PEND状态和PEND\_QUEUE状态在排序队列中对应的状态为TIMEOUT，而不是DELAY，这两种状态被唤醒时都需要判断是否是超时唤醒的，所以此处不清除TIMEOUT状态，而是等到唤醒后判断完以后再清除TIMEOUT状态。

Line 223 - 228：最后，若该TASK不处于SUSPEND（挂起）状态，则将该TASK重新加入到优先级队列中参与调度并启动一次任务切换，CPU重新调度当前优先级队列中优先级最高的TASK 执行，至于该TASK能否被调度到，则由其优先级决定。

Line 235：当满足调度条件时启动一次任务切换。

# 内存池管理

## 2.1 内存池管理概述

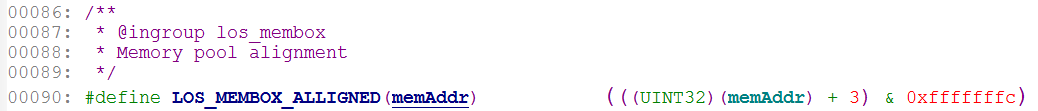
内存池是一块连续的内存空间，这块内存空间可以是一块静态数组，也可以是从堆内存中申请出来的一块连续空间。内存池实现动态的申请与释放。

内存池分为静态内存池和动态内存池，这两种内存池都是一块连续的内存空间，用户使用时从内存池中分配出合适大小的空间，使用完毕后归还。不同的是静态内存池中的内存块大小是固定的，动态内存池中内存块大小是可变的，用户可以按需申请。

用户在使用这两种内存池前必须先初始化，用户可以根据需要初始化多块内存池。

## 2.2 静态内存池

### 2.2.1 宏定义和结构体解析



LOS\_MEMBOX\_ALLIGNED：将memAddr的值按4字节对齐，得到对齐后的值。

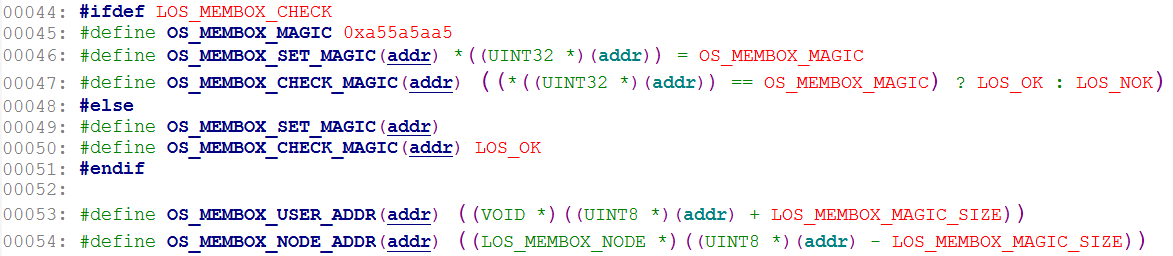
例：若memAddr = 24，已经是4字节对齐了，得到的结果还是24。若memAddr = 25或26或27，未按4字节对齐，则补足至4字节对齐，最后得到的结果是28。

需要按照4字节对齐的原因：初始时每个内存块被设置成LOS\_MEMBOX\_NODE类型，该类型存储指向下一个内存块的指针。即初始时每个内存块都只存储指向下一个内存块的指针（用内存块最开始的4个字节存储），内存池头部信息的stFreeList存储第一个内存块的指针，最后一个内存块存储NULL指针。内存块被分配给用户以后若有魔法字，则最开始的四个字节需要存储魔法字。一个指针或魔法字需要4个字节存储，所以按照4字节对齐，以使当内存块的大小小于4时能正确存储指针和魔法字信息。

next

OS\_MEMBOX\_NEXT：将addr向后偏移uwBlkSize个字节后的地址强转为LOS\_MEMBOX\_NODE型的指针。

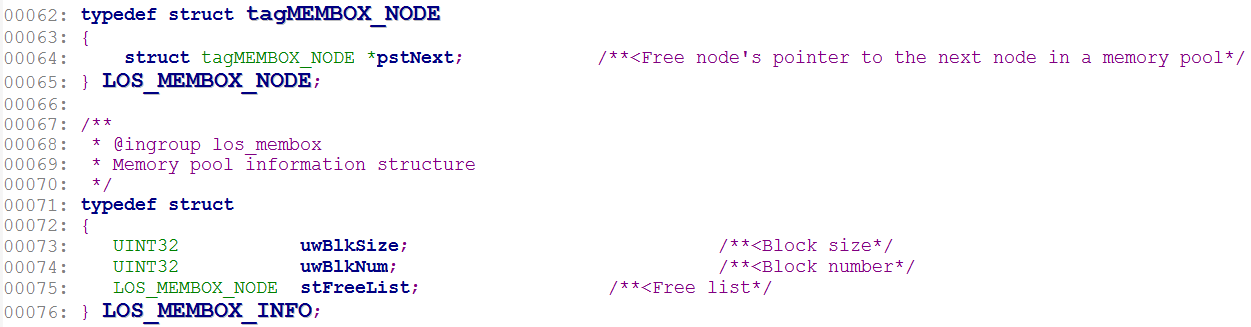
也就是将内存池中当前内存块的下一个内存块的起始地址强转为LOS\_MEMBOX\_NODE型的指针，用于存储指向下一个内存块的地址。



设置和检测静态内存池的魔法字：魔法字在用户申请内存块时设置，在释放内存块时检测，魔法字对用户透明。

OS\_MEMBOX\_USER\_ADDR：分配内存块时设置用户可用内存块的起始地址。若有魔法字，则用户使用内存块的起始地址是内存块起始地址加4字节魔法字后的地址。若无魔法字，则两者一致。

OS\_MEMBOX\_NODE\_ADDR：释放内存块时还原内存块起始地址。若有魔法字，则内存块起始地址在用户传入的内存块使用地址减4的位置。若无魔法字，则两者一致。



LOS\_MEMBOX\_NODE：每一个内存块的节点信息，内存块处于空闲状态时被强转为该类型，所以内存块的前4个字节存储的是指向下一个内存块起始地址的指针。当内存块被分配出去以后，如果有魔法字，则起始地址开始的4个字节填入魔法字，用户使用地址从魔法字之后开始，如果没有魔法字，则用户使用地址就是内存块起始地址。

LOS\_MEMBOX\_INFO：表示静态内存池信息的结构体，是内存池的头部，共12字节，存放于内存池最开始的位置。其中：

uwBlkSize：内存池中每个内存块的大小，4字节对齐。

uwBlkNum：内存池中内存块的个数，是内存池的总大小减去头部12字节后整除uwBlkSize的结果，整除后若存在不够一个内存块的区域，则该区域直接浪费掉。

stFreeList：内存块链接指针，链接内存池中的空闲内存块，初始时所有内存块处于空闲状态，并且都被链接在这个链表上。用户申请时从头部取下一个内存块，用户释放时将内存块重新加入到该链表的头部。

### 2.2.2 LOS\_MemboxInit( )函数

按照用户的要求，将用户传入的一块内存区域初始化为一块静态内存池。

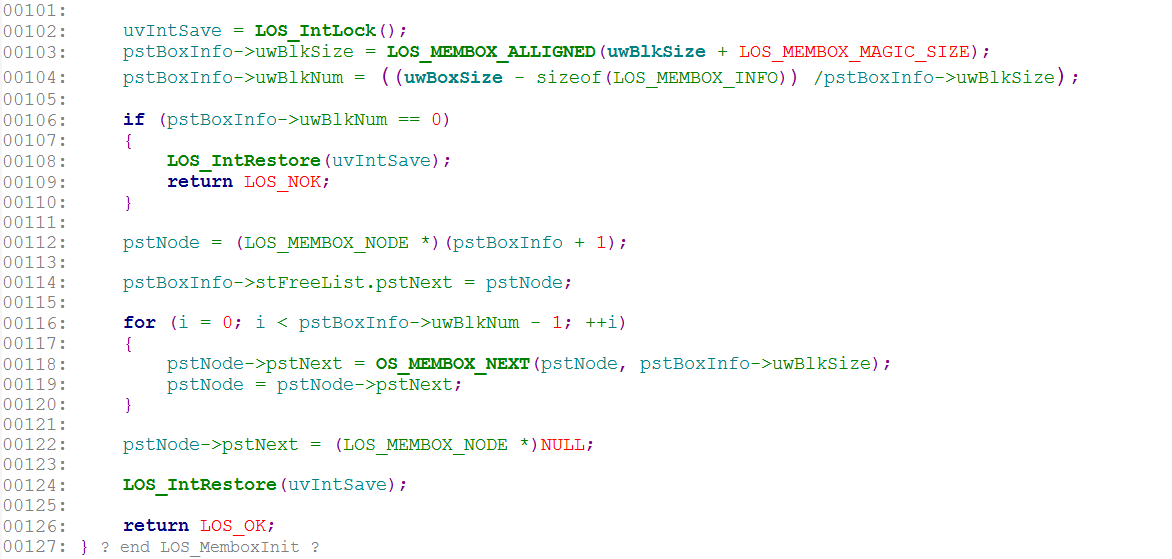
传入参数：\*pPool：静态内存池的起始地址。

uwBoxSize：静态内存池的总大小。

uwBlkSize：静态内存池中每一个内存块的大小。



Line 97：内存池的最小值为12字节，此时内存池只能存储头信息，对用户来讲这种大小的内存池没有任何意义。



填充内存池头部的信息：

Line 103：设置内存块大小uwBlkSize，考虑MAGIC的情况，并按4字节对齐。

Line 104：计算满足用户要求的完整的内存块的个数，取整，最后不够一块的区域直接浪费掉。

Line 106：uwBlkNum == 0说明用户传入的内存池无法分配出一个完整的内存块。有可能能分配出一块，但是大小无法满足，直接浪费掉。

Line 112：使pstNode指向内存池中跳过头部后的第一个内存块的起始地址。

Line 114：将内存池中第一个内存块链接到头部的stFreeList链表上。

Line 118：将pstNode指向的内存块的下一个内存块强转为LOS\_MEMBOX\_NODE型的指针并将当前内存块和下一个内存块通过pstNode->pstNext链接起来。

Line116：循环将内存池中所有的内存块依次链接起来。

Line122：最后一个内存块的pstNext置NULL，表示后面再没有内存块。

### 2.2.3 LOS\_MemboxAlloc( )函数

从特定的静态内存池中申请一个内存块。

传入参数：pPool指示对应内存池。

返回值：申请到的内存块的用户可以使用的首地址。



Line 143：从内存池中空闲链表的头部取下一个内存块，因为每个内存块大小一致，所以从头部分配不需要遍历链表，效率最高。

Line 145：pstRet指向将要分配给用户的内存块。

Line 146：将分配出去的内存块从空闲链表的头部删除。

Line 147：为分配出去的内存块设置魔法字，若有魔法字，则在内存块的开始4个字节处填入魔法字。

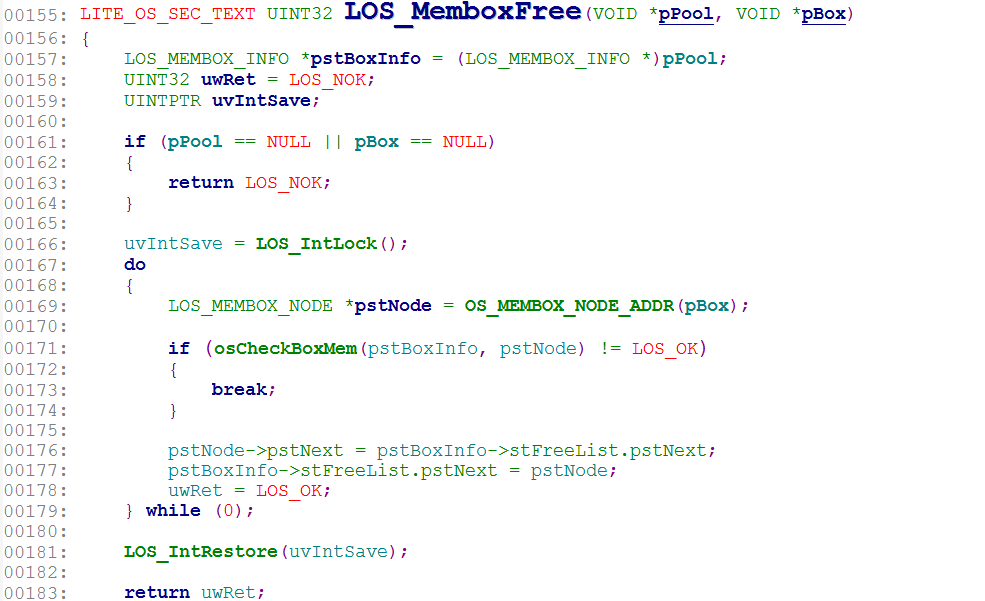
Line 152：返回给用户内存块的可用起始地址，就是内存块跳过魔法字后的地址。

### 2.2.4 LOS\_MemboxFree( )函数

将内存块释放回对应的内存池中。

传入参数：\*pPool：内存池起始地址。

\*pBox：要释放的内存块地址。



Line 169：根据用户传入的释放地址及系统魔法字配置状态还原内存块的起始地址。

Line 171：用户要释放的内存块的起始地址和魔法字的检测，只有检测通过才能正确释放内存块。

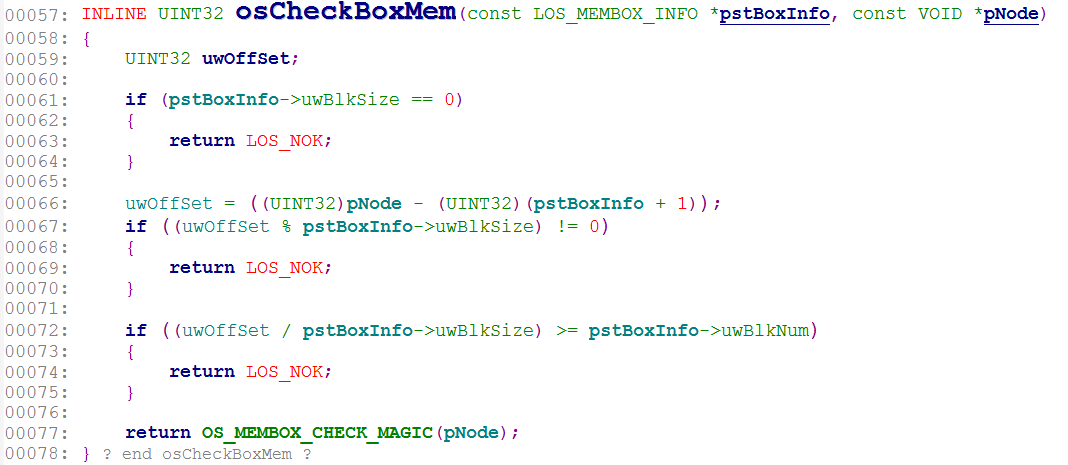
Line 176 - 177：将用户释放的内存块重新加入到内存块空闲链表的头部，内存块开始的4个字节重新存储指向下一个内存块的指针。

### 2.2.5 osCheckBoxMem( )函数

内存池检测，主要检测用户要释放的内存块的地址信息是否合法，魔法字信息是否匹配。

传入参数：\*pstBoxInfo：内存池起始地址。

\*pNode：用户要释放的内存块的起始地址，已经还原了魔法字。



Line 67：成立则说明内存块起始地址错误，即用户释放的地址不在内存块地址列表中。

Line 72：成立则说明内存块的起始地址超出了内存池的地址范围，当释放内存池中处于最后位置的内存块时uwOffSet / pstBoxInfo->uwBlkSize的值最大，为pstBoxInfo->uwBlkNum减1。

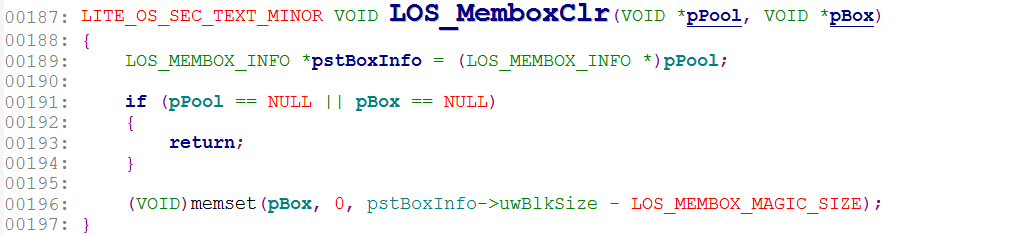
Line 77：内存块地址没有问题后检测魔法字。

### 2.2.6 LOS\_MemboxClr( )函数

将内存池中某一内存块的空间清0，不包括魔法字区域。主要用于用户申请内存块成功以后将可用内存块区域清0。

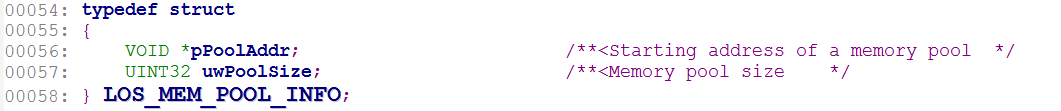
传入参数：\*pPool：内存池首地址。

\*pBox：内存块的用户使用地址，不是内存块的原始地址。

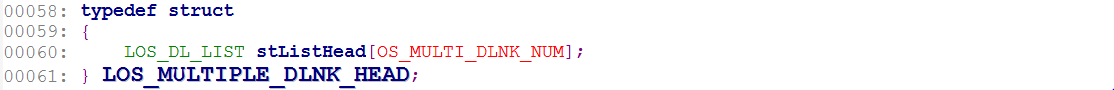


## 2.3 动态内存池

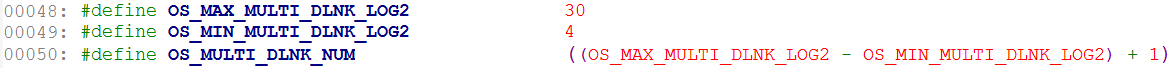
### 2.3.1 相关数据结构解析



LOS\_MEM\_POOL\_INFO是动态内存池的第一部分，存储动态内存池的起始地址和动态内存池的总大小。



LOS\_MULTIPLE\_DLINK\_HEAD是动态内存池的第二部分，是一个元素为LOS\_DL\_LIST的数组，每一个元素指示一个空闲链表，链接处于空闲状态的内存块。内存块处于USED状态时会从对应的空闲链表中取下，当处于UNUSED状态时会被重新加入到空闲链表中。



备注：空闲块选择空闲链表的依据：每一个空闲链表有一个允许的空闲块大小的范围，每个链表只链接在自己允许范围内的空闲块，这样用户申请空闲块时，就可以根据用户申请的大小找到对应的链表，而不需要遍历整个空闲链表。

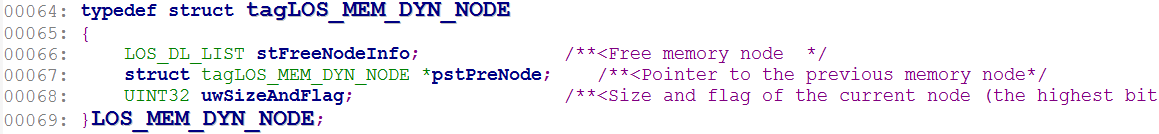
假设内存池允许的最小内存块大小为2^min字节，则第一个双链表链接的是所有size为2^min <= size < 2^min+1的空闲块，第二个双链表链接的是所有size为2^min+1 <= size < 2^min+2的空闲块，依次类推，第n个双链表链接的是所有size为2^min+n-1 <= size < 2^min+n的空闲块。每次申请内存时，会从空闲链表中检索最合适大小的空闲块，进行内存分配。每次释放内存时，会将该块内存作为空闲块存储至对应的空闲链表中，以便下次再利用。

OS\_MIN\_MULTI\_DLNK\_LOG2定义内存块大小的最小值，4表示大小为2的4次方，也就是16字节，即sizeof(LOS\_MEM\_DYN\_NODE)，这种内存块只有头部信息，没有可用的内存空间。OS\_MAX\_MULTI\_DLNK\_LOG2定义内存块大小的最大值为2的30次方。

按照上面的解释，第一个空闲链表存储所有size为2^4 <= size < 2^5（16 <= size < 32）的空闲块，第二个空闲链表存储所有size为2^5 <= size < 2^6（32 <= size < 64）的空闲块，依次类推，第27个空闲链表存储所有size为2^30 <= size < 2^31的空闲块，因为系统允许的最大空闲块为2^30，故第27个空闲链表实际上只存储大小为2^30的空闲块。

假设：空闲块大小为4800字节，因为2^12 < 4800 < 2^13（4096 < 4800 < 8192），所以大小为4800的空闲块需要链接在数组下标为8（12 - 4）的空闲链表上，也就是第九个空闲链表上。

假设：空闲块大小为64字节，因为2^6 <= 64 < 2^7（64 <= 64 < 128），所以大小为64的空闲块需要链接在数组下标为2（6 - 4）的空闲链表上，也就是第三个空闲链表上。



LOS\_MEM\_DYN\_NODE是内存块的头部信息，内存池中有多个用户申请的内存块，同时又有多个处于空闲状态的内存块，每个内存块都有一个头部信息，指示这个内存块的相关信息。其中：

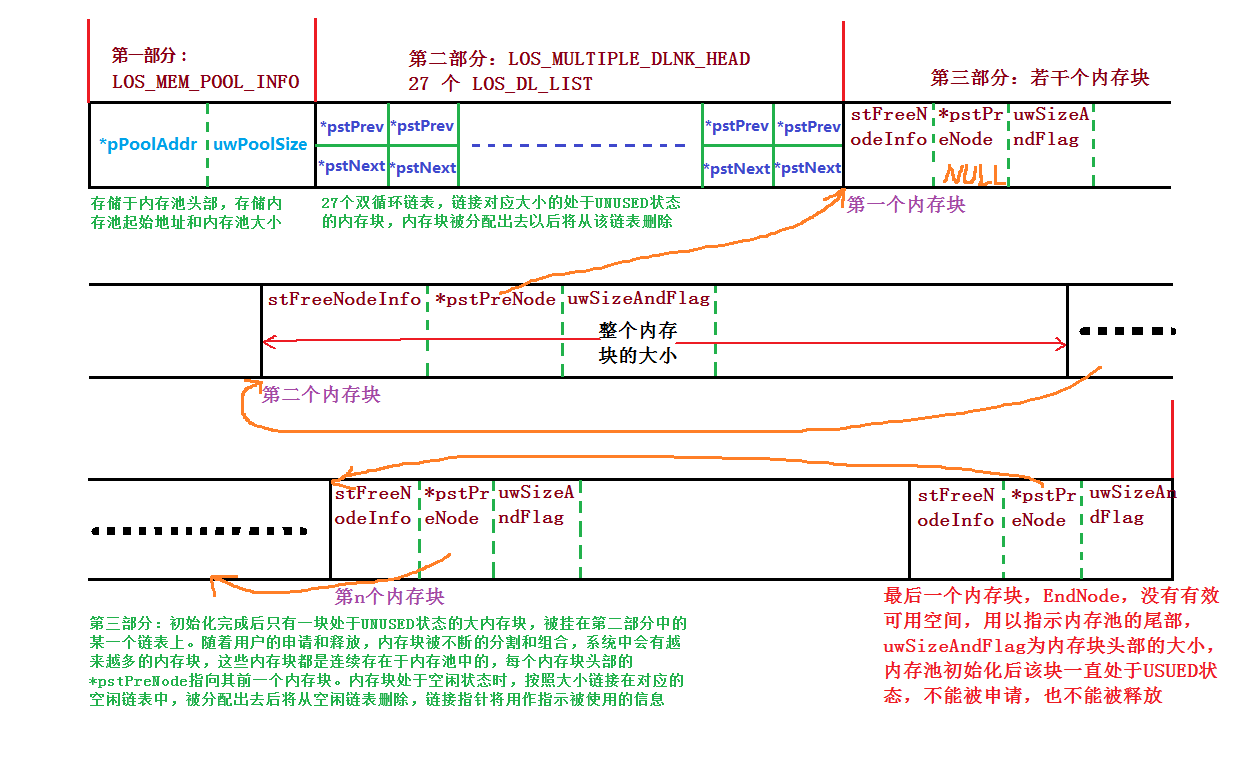
stFreeNodeInfo：内存块处于UNUSED状态时将其用双链表的方式链接在第二部分对应的空闲链表上。处于USED状态时该内存块会从空闲链表上取下，stFreeNodeInfo会存储新的信息，其中\*pstPrev用于存储魔法字，\*pstNext用于指示当前正在使用这个内存块的TASK。

\*pstPrevNode：用于指示当前内存块的前一个内存块的头部，因为内存池是连续的空间，内存块将内存池分为若干个块，所以第一个内存块的\*pstPrevNode = NULL，后面所有的\*pstPrevNode都指向其前一个内存块。\*pstPrevNode主要用于访问前面的内存块，当前内存块的后面一个内存块不用指针指示，而是通过当前内存块地址加大小的方式访问，这样做可以节减小头部信息的大小。

备注：\*pstPrevNode只是用于链接前一个内存块，与前一个内存块处于USED还是UNUSED状态没有关系。

uwSizeAndFlag：当前内存块的大小，包括头部信息的大小，其中最高的1个bit指示USED还是UNUSED状态，剩下的指示内存块大小，最小为sizeof(LOS\_MEM\_DYN\_NODE)，也就是16字节，最大为2^30。

内存池示意图如下：



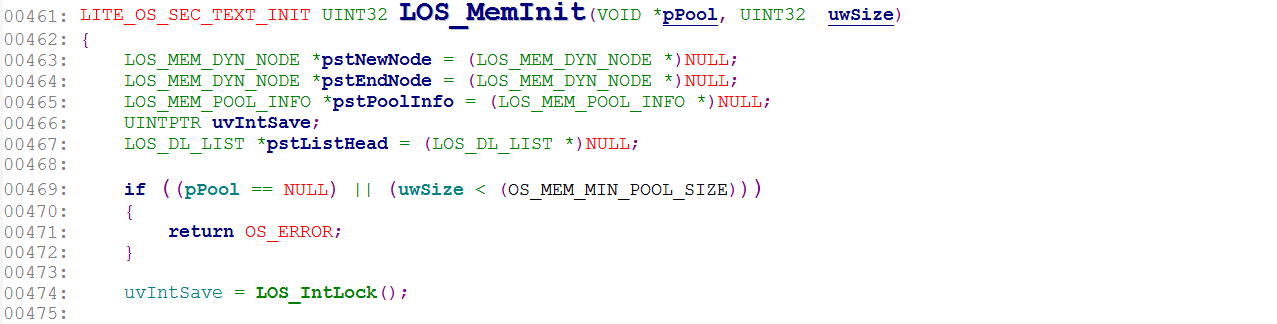
### 2.3.2 LOS\_MemInit( )函数

功能：将一段连续的内存空间初始化为一个动态内存池。

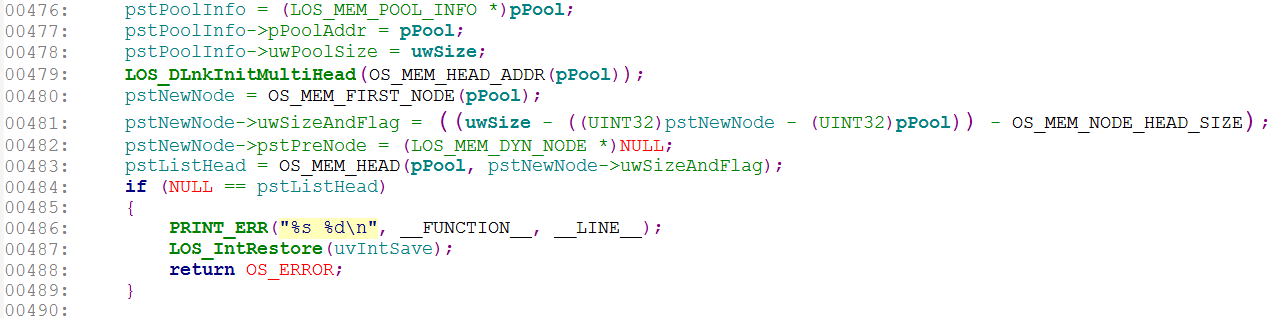
传入参数：

\*pPool：动态内存池起始地址。

uwSize：动态内存池大小。

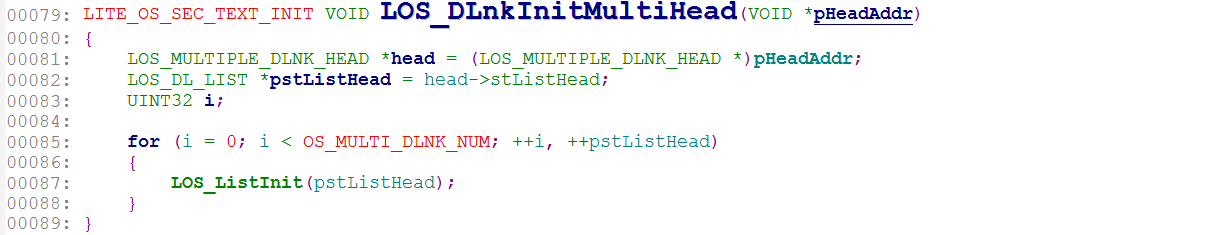


Line 469：内存池大小的最小值OS\_MEM\_MIN\_POOL\_SIZE被定义为：内存池第一部分大小+第二部分大小+两个内存块头部的大小（一个用于分配第一个空闲内存块，一个用于分配默认的最后一个内存块）。



Line 476 - 478：填充内存池的第一部分（内存池的起始地址和内存池的大小）。

Line 479：代码如下图，填充并初始化内存池的第二部分（将所有的空闲链表初始化为空链表）。

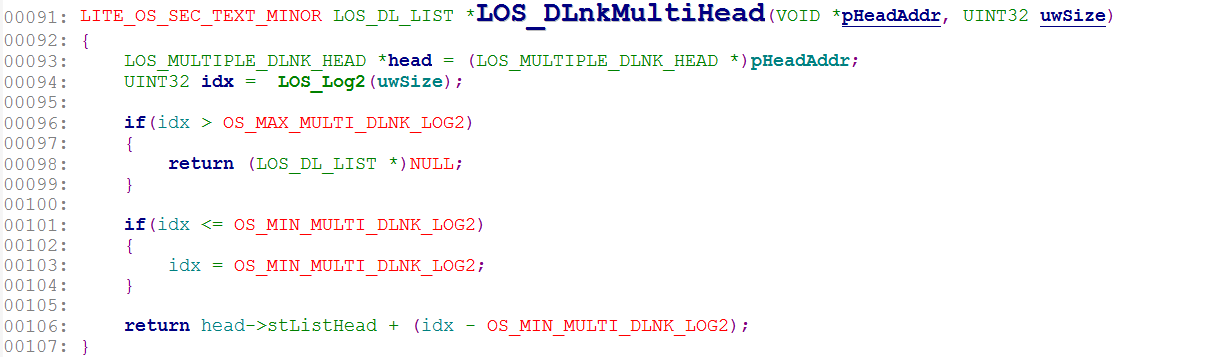


Line 480 - 482：填充内存池的第三部分（设置默认的最大的空闲块）。

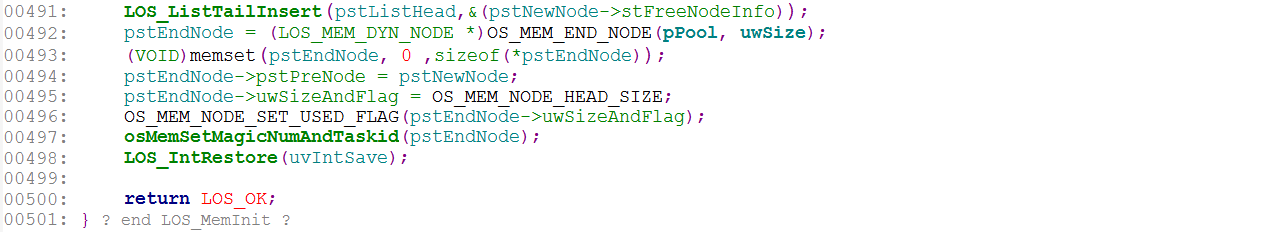
Line 481：最后减去OS\_MEM\_NODE\_HEAD\_SIZE，是为了给最后的EndNode使用。

Line 482：第一个空闲块的前面没有内存块，所以\*pstPreNode置NULL。

Line 483：按照空闲块的大小去查找该块要加入的空闲链表。OS\_MEM\_HEAD最终被转换为调用函数LOS\_DLnkMultiHead( )，函数实现如下图：



传入参数：\*pHeadAddr是指向空闲链表数组的指针，uwSize是空闲块的大小。调用LOS\_Log2( )来计算大小为uwSize的块要加入的空闲链表的编号。例如uwSize = 4800，则LOS\_Log2( )返回的idx为12，所以最终返回的空闲链表的数组下标为8，就是将该块加入到数组下标为8的空闲链表中，即第9个空闲链表。



Line 491：将空闲块加入到对应的空闲双链表的尾部。

Line 492 - 497：设置内存块的最后一个节点，用于标识内存池的结束位置。

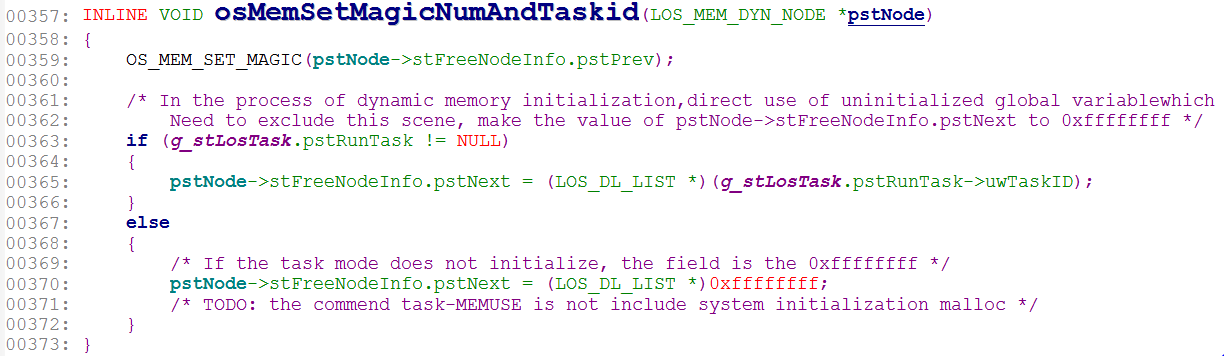
Line 492：由内存池起始地址和大小确定最后一个内存块的起始地址。

Line 494：此时内存池中有两个内存块，一个是默认的最大的空闲块，已经被链接在空闲链表上了，另一个是EndNode，所以需要设置EndNode的pstPreNode指针指向前一个内存块。

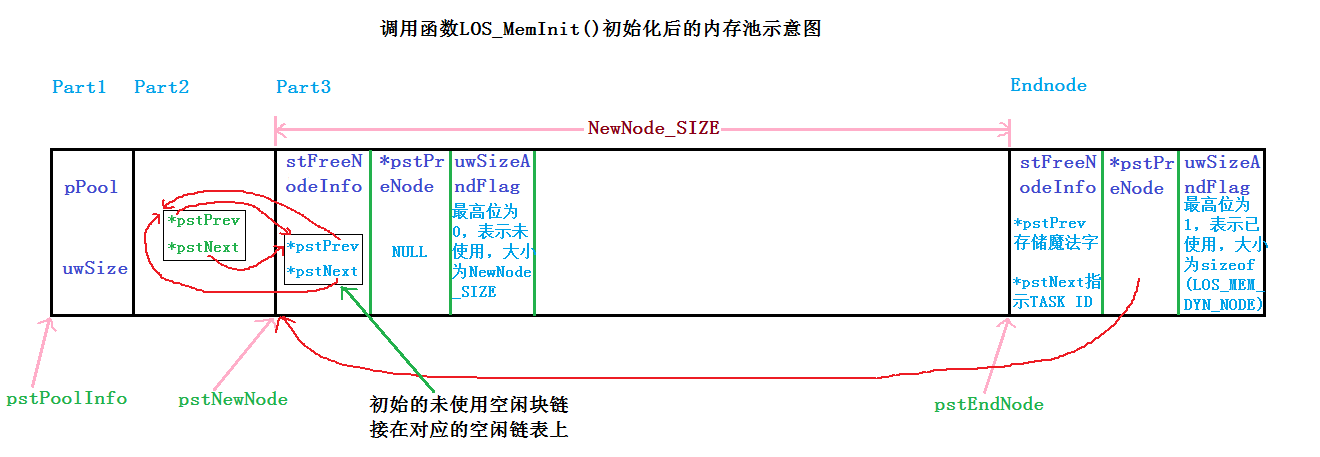
Line 495：EndNode这个内存块处于内存池的尾部，没有有效可用空间，只有头部信息，所以大小为头部信息的大小。

Line 496：设置EndNode的状态为USED，此后这个块一直处于USED状态，没有被加入到任何空闲链表中。

Line 497：osMemSetMagicNumAndTaskid( )函数实现如下图，处于USED状态的内存块需要设置stFreeNodeInfo的pstPrev指针为魔法字信息，pstNext指针指示使用这个内存块的TASK的信息。（处于UNUSED状态的内存块的stFreeNodeInfo用于将内存块链接在空闲链表上）



调用函数LOS\_MemInit( )初始化后的内存池的示意图如下：



### 2.3.3 LOS\_MemAlloc( )函数

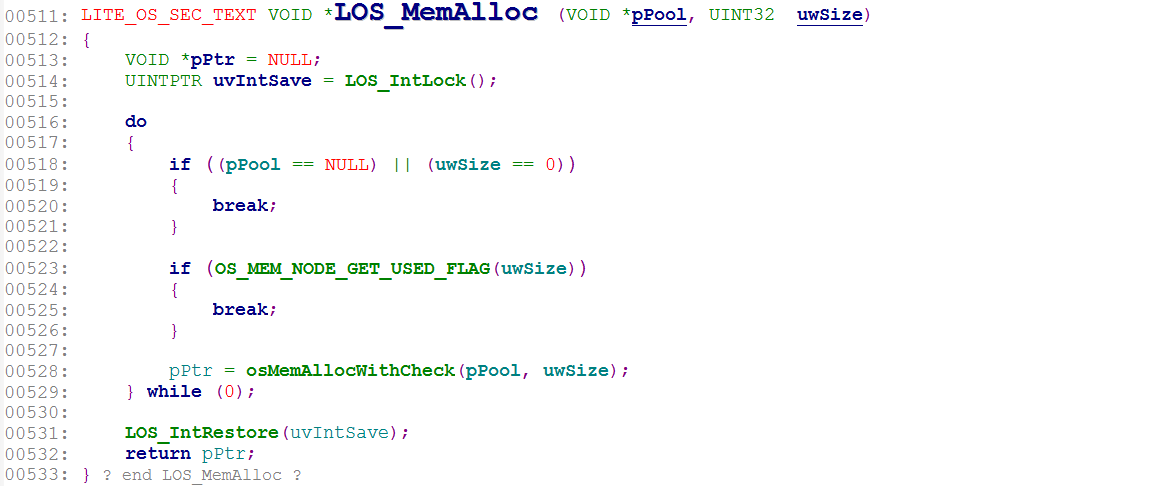
功能：从pPool指示的动态内存池中申请大小为uwSize的内存块

传入参数：

\*pPool：动态内存池的首地址，使用前必须先初始化。

uwSize：用户申请的内存大小，因为内存块有头部信息和对齐的空间，所以实际分配出来的内存块大于用户申请的大小。

返回值：返回用户可用空间的起始地址。



Line 518：不允许用户申请大小为0的内存块。有的平台允许这么做，申请大小为0的内存块时返回一个只有头部信息的内存块，返回给用户的指针是下一个内存块的起始地址，用户使用这个指针会产生不可预知的后果。

Line 523：uwSize的最高位表示USED或UNUSED状态，用户不能在申请时传入USED状态的标志。

Line 528：从内存池中分配一个内存块，并将用户可用空间的地址返回给用户。

### 2.3.4 osMemAllocWithCheck( )函数

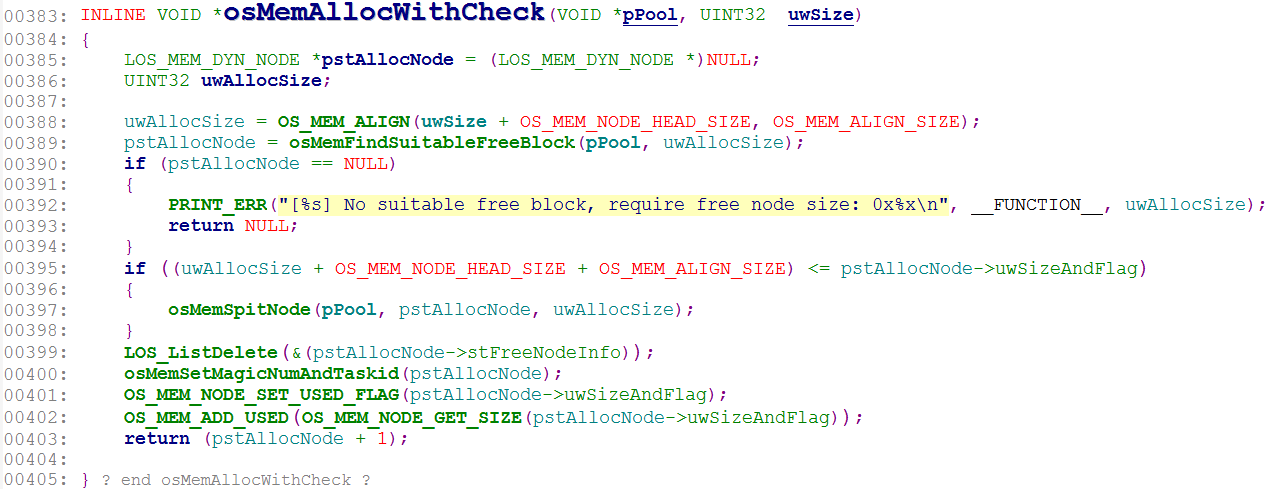
功能：从动态内存池中分配一块内存，并将结果返回。

传入参数：

\*pPool：动态内存池的首地址。

uwSize：申请的内存的大小。

返回值：内存块跳过头部信息后的指针。



Line 388：uwSize只是用户需要的内存大小，系统需要为这块内存设置头部信息，并且内存块按照4字节对齐，所以本次申请需要申请的内存总大小为uwAllocSize。

Line 389：从内存池的空闲块中查找一块合适的内存块（头部信息在加入空闲链表时已经填充好）并将内存块起始地址返回，不对内存块做任何操作。

Line 395：条件成立表示：查找出的内存块比较大，可以将当前内存块拆分为两块，前一块返回给用户使用，后一块重新加入到空闲链表中供后续的申请匹配。若内存块比用户申请的稍大，但是无法拆分为两块，直接将多余的空间一并返回给用户，但用户不能使用多出来的空间，否则可能会造成不可预知的后果。

Line 397：将当前的大内存块拆分为两块，同时设置后面一块的头部信息，并将其加入到与块大小相匹配的空闲链表中。

Line 399：将满足本次申请的内存块从空闲链表中取下。

Line 400：设置当前内存块的魔法字信息和使用该内存块的TASK的信息。

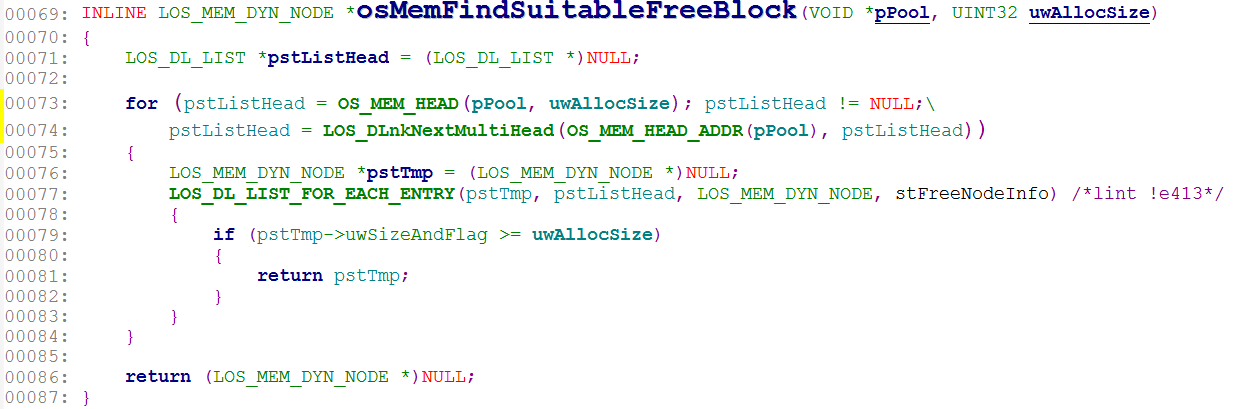
Line 401：设置当前内存块的状态为USED状态。

Line 402：每个TASK都有一个统计该TASK使用动态内存大小的变量uwMemUsed，当申请一块内存时，将内存块大小的值累加到uwMemUsed上，释放一块内存时，将内存块的大小的值从uwMemUsed中减去。

Line 403：将内存块跳过头部信息后的指针，也就是用户可用空间的地址返回给用户。

### 2.3.5 osMemFindSuitableFreeBlock( )函数

功能：从\*pPool指示的内存池中查找满足uwAllocSize大小的内存块，并将该内存块的起始地址返回，不对该内存块做任何操作。



Line 73 - 74：遍历满足条件的链表，从uwAllocSize对应的链表开始查找，如果当前链表为空，则转入下一个链表，直到找到合适的内存块或将所有链表遍历完为止。

假设uwAllocSize为280字节，因为2^8 < 280 < 2^9，所以第一次遍历时查找数组下标为4（8 - 4）的空闲链表，若为空，则继续遍历数组下标为5的空闲链表，依次类推，直到找到合适的空闲块为止。

Line 77：当前链表不为空，依次遍历当前链表中的空闲块。

Line 79：找到一个满足条件的空闲块，将其返回，不再遍历后面的块。

备注：释放空闲块时直接将其加入到对应空闲链表的尾部，所以空闲链表中的空闲块不是按照块大小排列的，故需要一直遍历链表中的空闲块，直到找到一块满足条件的空闲块为止。这种排列方式是不是最优的？

空闲块按照由小到大排列：这种排列方式查找速度慢，但是可以保证每次分配给用户的空闲块是最合适的一块。

空闲块按照由大到小排列：这种排列方式查找速度快，只要第一块不符合要求，那么后面的所有块都不符合要求。但是每次分配给用户的是当前链表中最大的一块。

### 2.3.6 osMemSpitNode( )函数

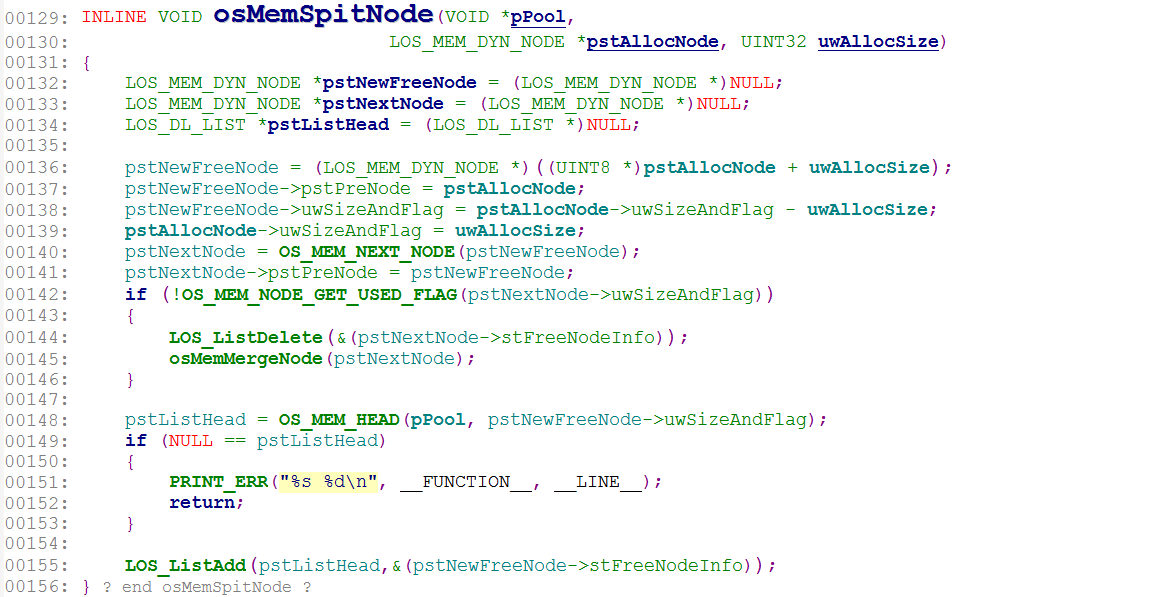
功能：将内存块一分为二，同时设置两个内存块的头部信息，并将后一个内存块链接到空闲链表上。假设要拆分的大内存块为AB，拆分后的两个小内存块依次为A、B，则拆分后A将返回给用户使用，B将被链接到空闲链表上。

传入参数：

\*pPool：内存池起始地址，用于查找空闲链表。

\*pstAllocNode：AB的起始地址。

uwAllocSize：A的总大小。



\*pstAllocNode在拆分前指示AB，拆分后指示A。\*pstNewFreeNode用于在拆分后指示B。pstNextNode用于指示B的后面一个内存块，假设为C。

Line 136：由AB的起始地址和A的总大小设置B的起始地址。

Line 137：设置B的前指针指向A。

Line 138：设置B的总大小为AB的大小减去A的总大小。

Line 139：设置A的大小。

Line 140：由B的起始地址和B的大小查找C。

Line 141：设置C的前指针指向B。

Line 142：若C的状态为UNUSED，则B和C都是空闲块，需要将两块合并成一个大的空闲块（假设为BC）并使C后面的一个块的前指针指向BC。

Line 144：将C从空闲队列中删除，准备与B合并。

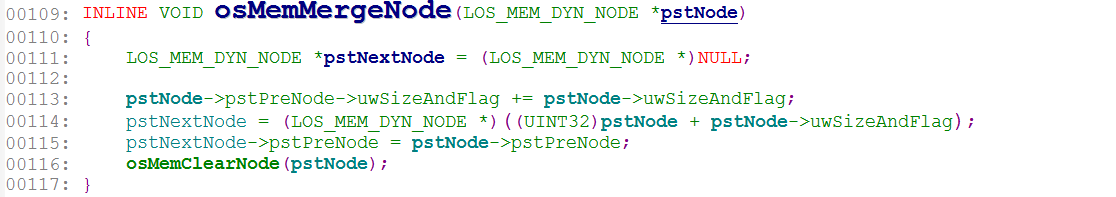
Line 145：将B与C合并，去除C的头部信息，合并成大块BC。

Line 148：按照B或者BC的大小查找对应的空闲链表。

Line 155：将B或BC链接到对应的空闲链表中。

### 2.3.7 osMemMergeNode( )函数

功能：传入当前内存块的起始地址（假设为C），将当前内存块与其前一个内存块（假设为B）合并，并使当前内存块的后面一个内存块（假设为D）的前指针指向合并后的内存块（假设为BC）。

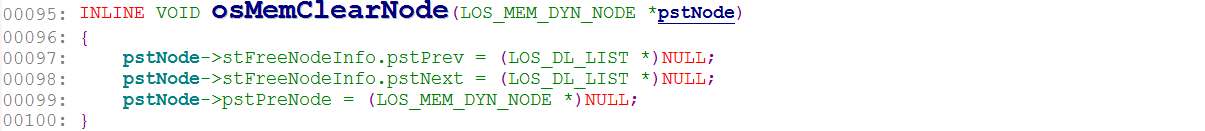


Line 113：重新设置BC的大小。

Line 114：由C的起始地址和C的大小查找D。

Line 115：使D的前指针由指向C改为指向BC。

Line 116：消除C的头部信息，函数实现如下图。



### 2.3.8 LOS\_MemAllocAlign( )函数

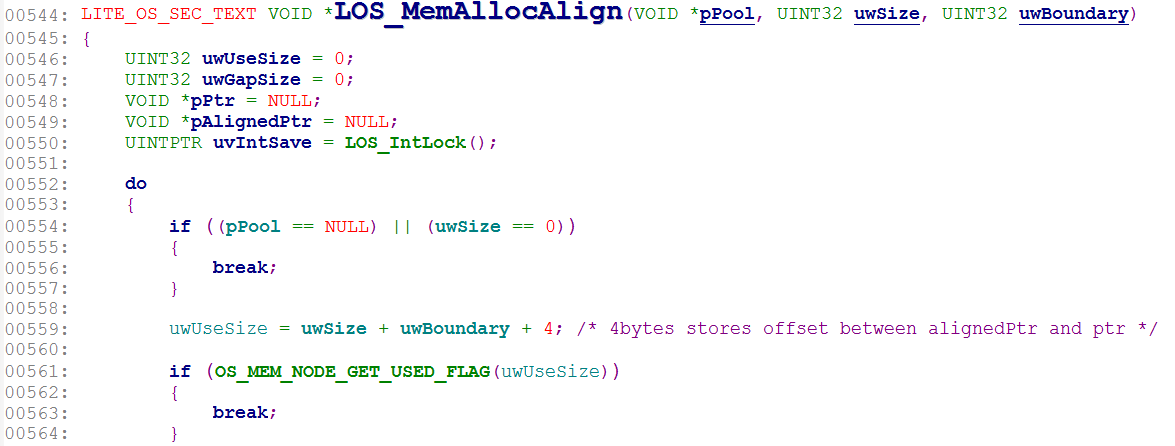
功能：向内存池申请一块内存块，返回的内存块的用户可用的首地址必须按照用户的要求对齐在某一边界上。

传入参数：

\*pPool：内存池首地址。

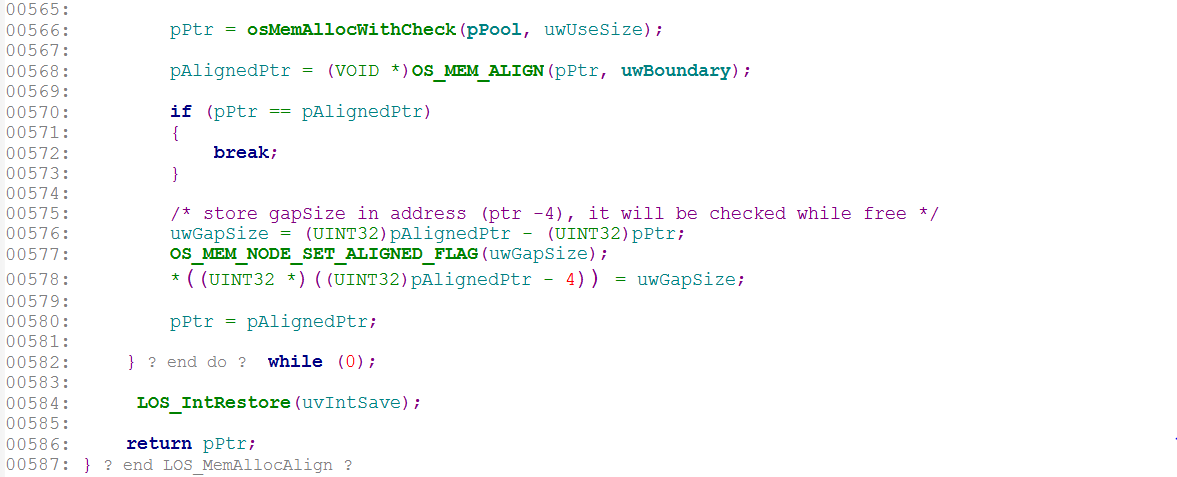
uwSize：用户申请内存块的用户可用空间大小。

uwBoundary：对齐要求，返回的地址必须对齐到uwBoundary的边界上。



Line 559：4个字节用于存储对齐前和对齐后的指针中间的大小。

Line 561：用户的申请大小中不允许带有USED标志。



Line 566：从内存池中分配一块内存块，并返回用户可用的首地址，不是内存块在内存池中的起始地址。

Line 568：将内存块用户可用首地址按照uwBoundary对齐。

Line 570：内存块分配出来以后返回的用户可用地址已经按照uwBoundary对齐了，则不需在做任何工作，直接将用户可用地址返回给用户。

Line 576：获取对齐后的地址和对齐前的地址之间的空间大小。

Line 577：对这个块设置对齐标志。

Line 578：将中间的大小存储起来。

Line 580：将对齐后的地址返回给用户。

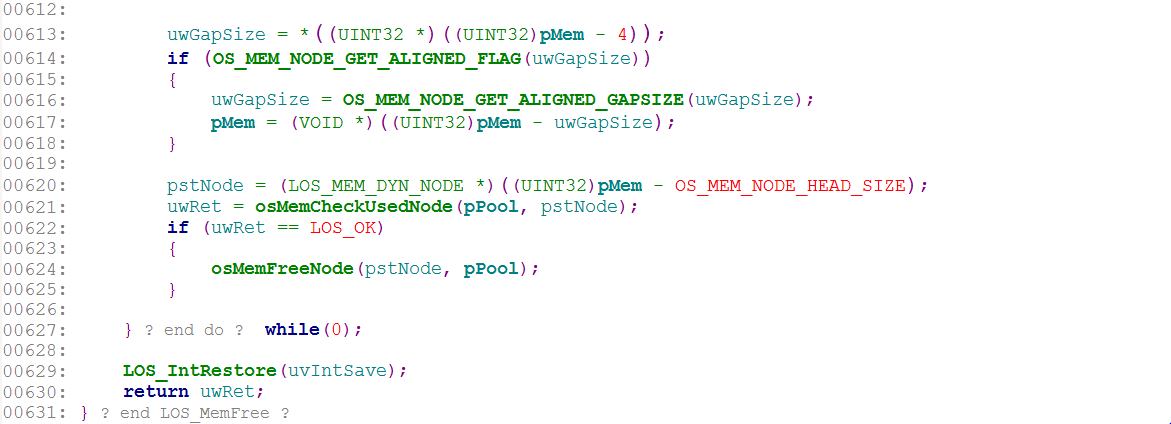
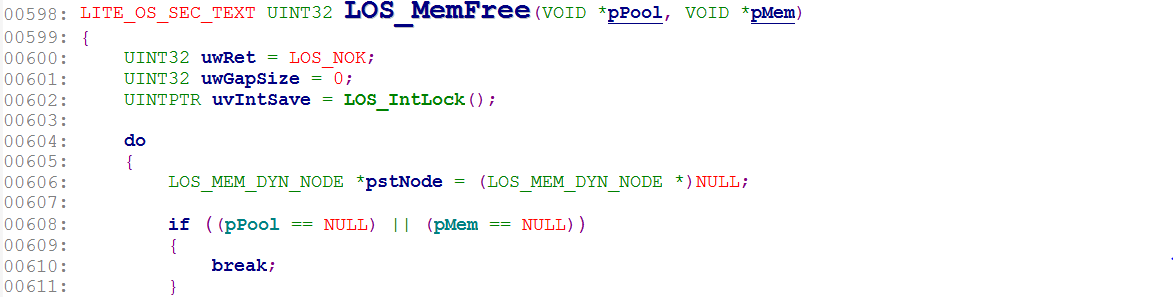
### 2.3.9 LOS\_MemFree( )函数

功能：向动态内存池释放一个内存块。

传入参数：

\*pPool：动态内存池首地址。

\*pMem：用户申请时返回给用户的内存块地址。



Line 613：将用户传入的释放地址前推四个字节，获取到的值有两种可能：若用户在申请时没有设置对齐，则会获取到内存块头部信息中的uwSizeAndFlag。若用户在申请时设置了对齐，则会获取到对齐地址和非对齐地址之间的空间大小，该值的最高位下面的一位表示对齐标志。

Line 614：若在申请时用户未设置对齐，则此条件不成立，若设置了对齐，则此条件成立，需要通过uwGapSize的值找到未对齐前的地址。

Line 616：获取对齐地址和未对齐地址之间的差值。

Line 617：由对齐地址和差值恢复未对齐地址，这个地址是内存块除了头部信息后的首地址。

Line 620：由内存块用户使用地址获取内存块起始地址。

Line 621：检验用户要释放内存块的合法性。

Line 624：释放内存块，将内存块重新还回内存池中。

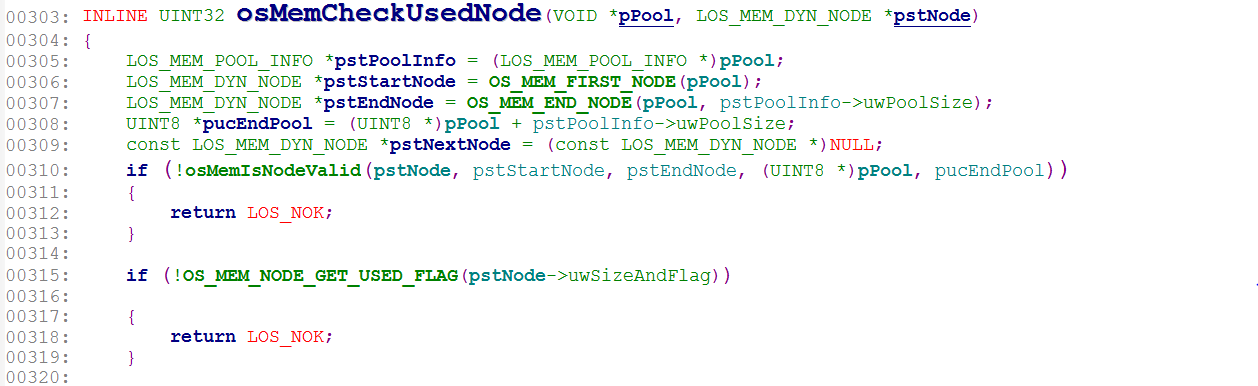
### 2.3.10 osMemCheckUsedNode( )函数

功能：检验内存块的合法性。

传入参数：

\*pPool：内存池首地址。

\*pstNode：要检验的内存块的起始地址。



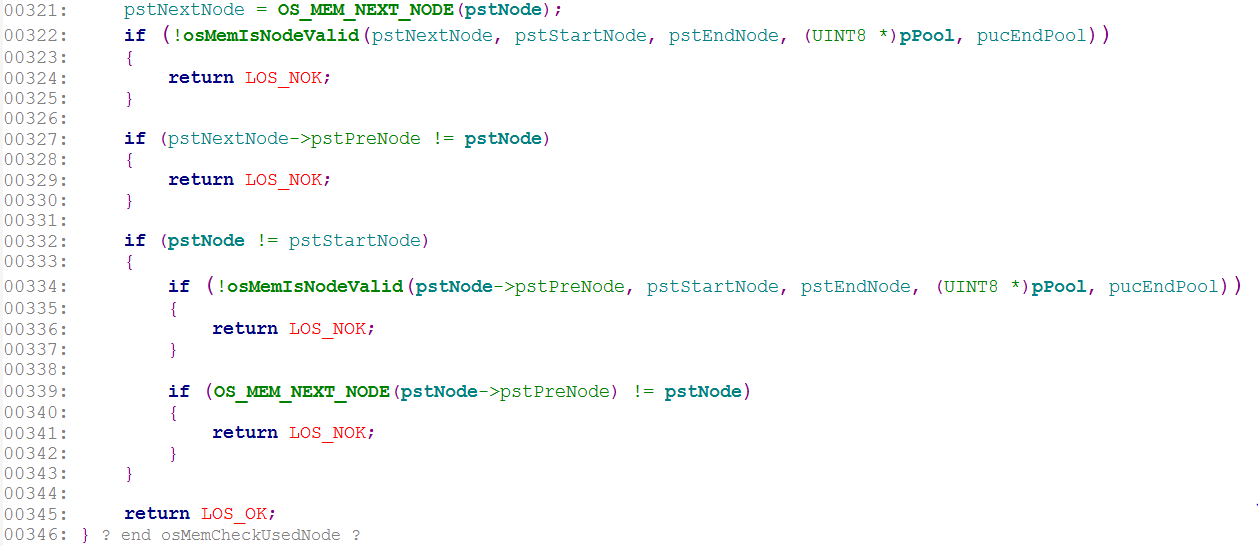
Line 306：由内存池起始地址查找第一个内存块的起始地址。

Line 307：由内存池起始地址和内存池总大小查找最后一个内存块的起始地址。

Line 308：由内存池起始地址和内存池总大小查找内存池的结束地址。

Line 310：检测要释放的内存块的有效性。

Line 315：判断内存块的状态是否为USED状态。



Line 321：由当前内存块获取下一个内存块的起始地址。

Line 322：检测下一个内存块的有效性。

Line 327：判断下一个内存块的pstPreNode指针是否指向当前内存块。

Line 332：判断当前内存块是否是内存池中的第一个内存块。

Line 334：检测当前内存块的前一个内存块是否有效。

Line 339：判断由前一个内存块的起始地址得到的当前内存块的起始地址与传入的当前内存块的起始地址是否相等。

Line 345：内存块合法。

### 2.3.11 osMemIsNodeValid( )函数

功能：检测某一内存块的有效性。

传入参数：

\*pstNode：要检测的内存块的起始地址。

\*pstStartNode：内存池中第一个内存块的起始地址。

\*pstEndNode：内存池中最后一个内存块的起始地址。

\*pucStartPool：内存池的起始地址。

\*pucEndPool：内存池的结尾地址。



Line 277：判断要检测的内存块是否位于内存池中第一个内存块和最后一个内存块之间。

Line 282 - 291：对于处于USED状态内存块，需要检测内存块的魔法字信息并返回检测结果。

Line 293：对于处于UNUSED状态的内存块，需要检测链接该内存块的链表是否处于内存池的起始地址和结尾地址之间。

### 2.3.12 osMemFreeNode( )函数

功能：释放内存块，将内存块还回内存池中

传入参数：

\*pstNode：要释放的内存块的起始地址

\*pPool：内存池首地址



Line 172：释放内存块后该内存块不再为TASK所用，所以需要将该内存块的大小从TASK使用动态内存的总大小中除去。

Line 173：获得要释放的内存块的总大小。

Line 174：要释放的内存块不是内存池中的第一个内存块，并且要释放的内存块（假设为B）的前一个内存块（假设为A）处于UNUSED状态。

Line 177：由B的前指针找到A起始地址。

Line 178：将B与A合并为一个空闲内存块，假设为AB，合并后B不再存在。

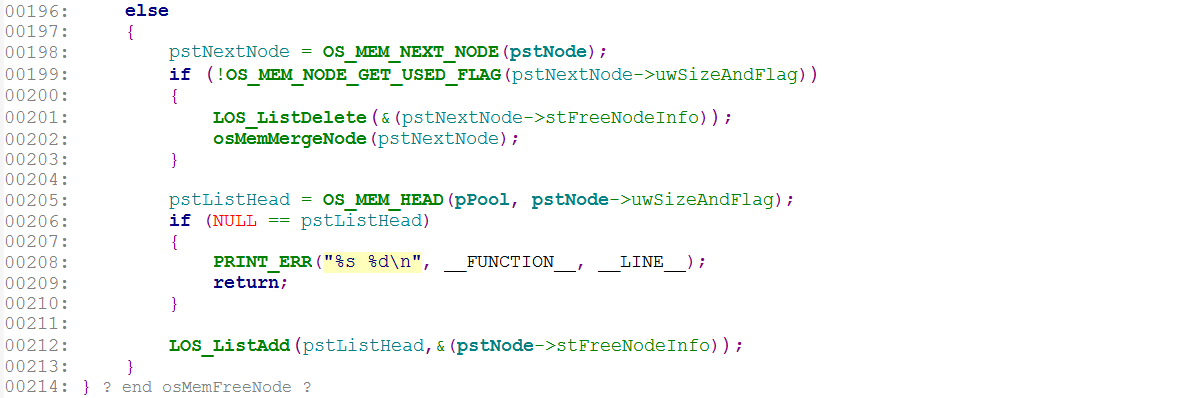
Line 179：由AB的起始地址获取AB的后一个内存块（假设为C）的起始地址。

Line 180 - 184：C为UNUSED状态，则需要将C从空闲链表中删除，并将C与AB合并成一个大的空闲内存块（假设为ABC）。

Line 186：将合并后的AB或ABC从空闲链表中取下。

Line 187：重新寻找适合AB或ABC的空闲链表。

Line 194：将AB或ABC链接到新的空闲链表中



Line 196：要释放的内存块是内存池中的第一个内存块。

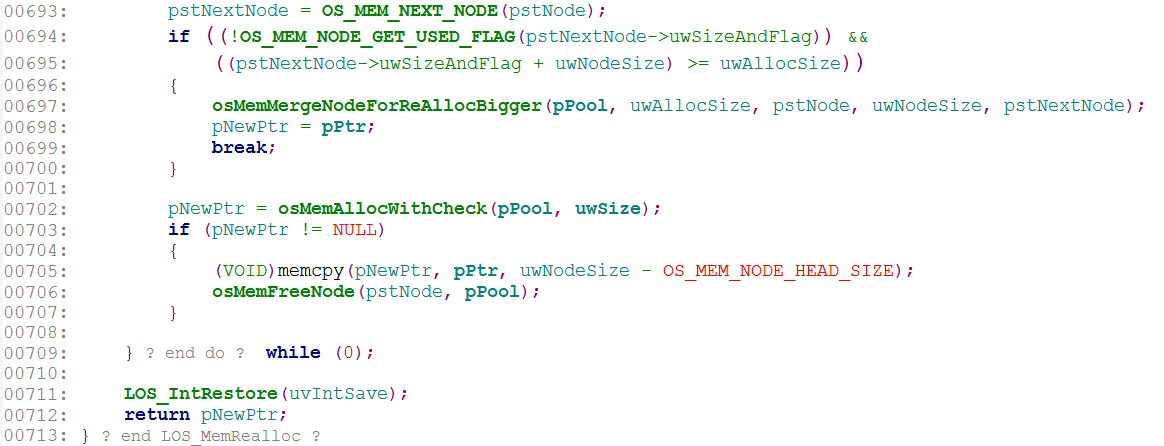
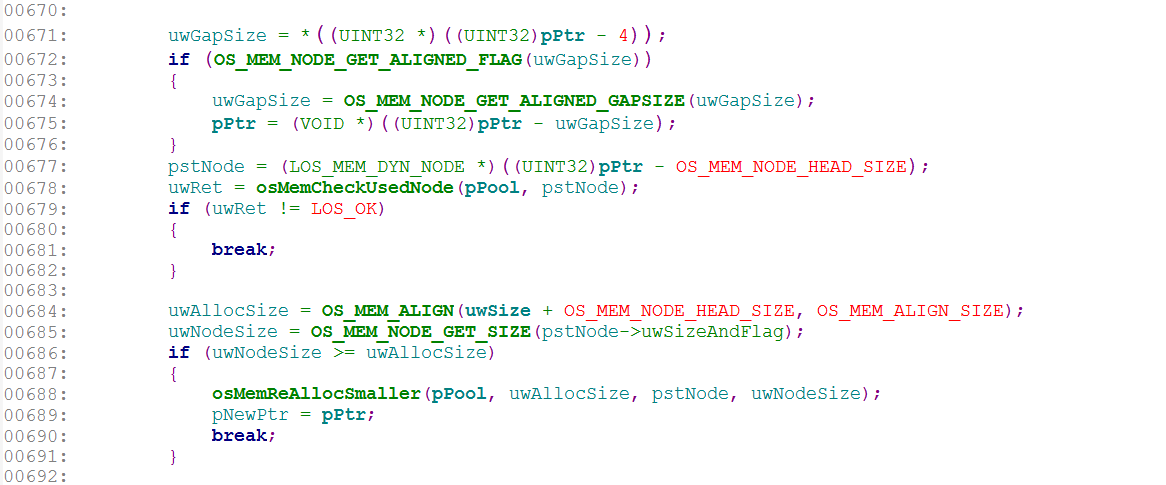
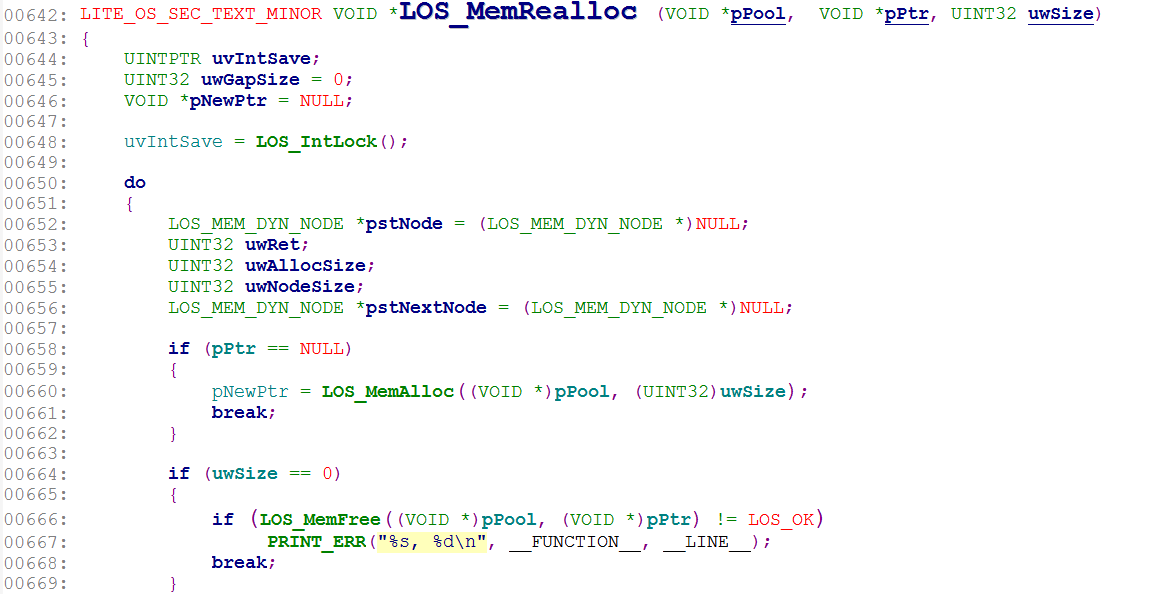
Line 198：由当前内存块（假设为A）的起始地址查找下一个内存块的起始地址（假设为B）。

Line 199 - 203：B处于UNUSED状态，则将B从空闲链表中删除，并将B与A合并为一个新的空闲内存块（假设为AB）。

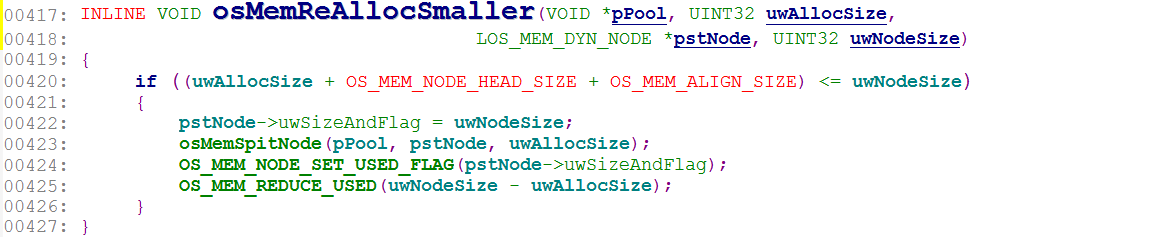
Line 205：由AB的大小查找适合的空闲链表。

Line 212：将AB链接在新的空闲链表中。

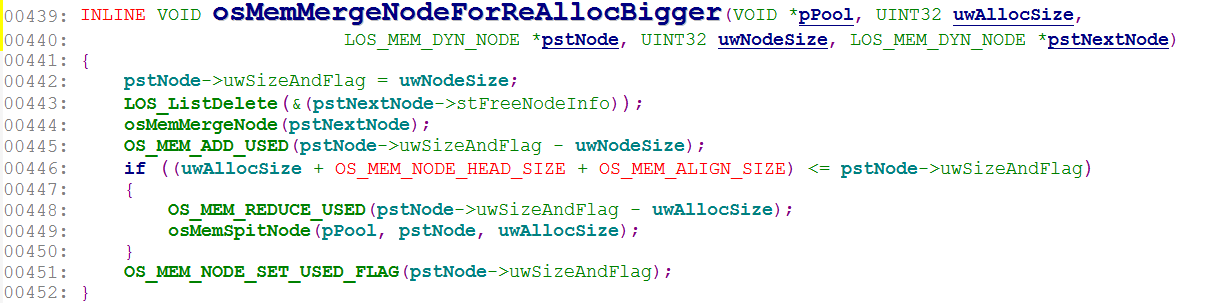
### 2.3.13 LOS\_MemRealloc( )函数



### 2.3.14 osMemReAllocSmaller( )函数



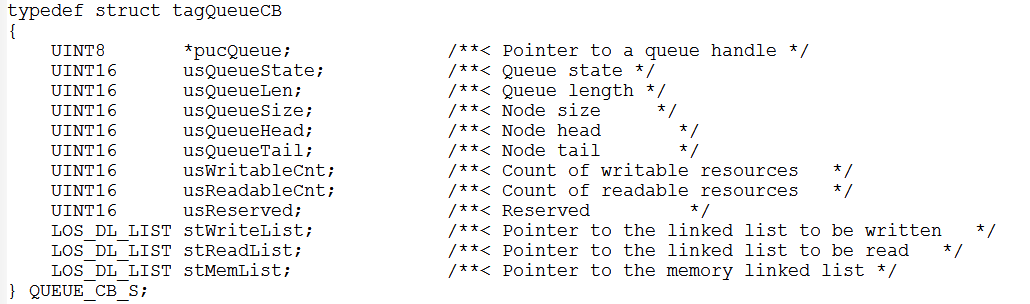
### 2.3.15 osMemMergeNodeForReAllocBigger( )函数

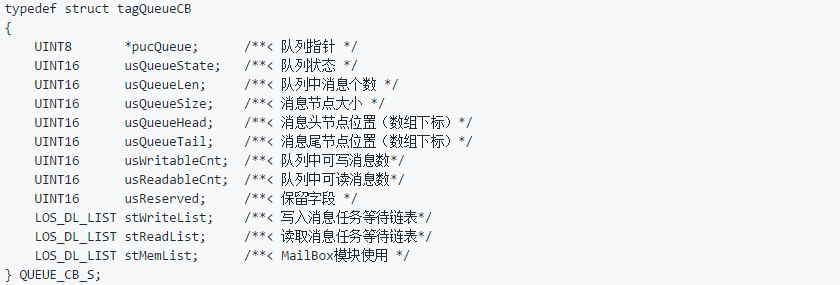


# IPC通信(消息队列和事件)

## 3.1 消息队列

### 3.1.1 QUEUE\_CB\_S结构解析





\*pucQueue：指向消息节点区域，该区域是在创建队列时按照消息节点个数和节点大小从动态内存池中申请出来的一块空间。

usQueueState：队列状态，标明队列控制块是否被使用，有OS\_QUEUE\_INUSED和OS\_QUEUE\_UNUSED两种状态。

usQueueLen：消息节点个数，表示该消息队列最大可存储多少个消息。

usQueueSize：消息节点大小，表示每个消息节点可存储信息的大小。

消息节点按照循环队列的方式访问，队列中的每个节点以数组下标表示，下面的四个成员与消息节点循环队列有关：

usQueueHead：指示消息节点循环队列的头部。

usQueueTail：指示消息节点循环队列的尾部。

usWritableCnt：消息节点循环队列中可写的消息个数，为0表示循环队列为满，等于usQueueLen表示循环队列为空。

usReadableCnt：消息节点循环队列中可读的消息个数，为0表示循环队列为空，等于usQueueLen表示消息队列为满。

usReserved：保留字段。

stWriteList：写消息阻塞链表，链接因消息队列满而无法写入时需要挂起的TASK，按照优先级由高到低排列。

stReadList：读消息阻塞链表，链接因消息队列空而无法读取时需要挂起的TASK，按照优先级由高到低排列。

stMemList：申请内存块阻塞链表，链接因申请某一静态内存池中的内存块失败而需要挂起的TASK，按照优先级由高到低排列。

### 3.1.2 osQueueInit( )函数



LOSCFG\_BASE\_IPC\_QUEUE\_LIMIT是系统允许的消息的最大个数，每一个消息都由唯一的消息控制块QUEUE\_CB\_S结构体管理。初始化时g\_pstAllQueue指向系统消息控制块数组。所有的消息控制块连续存储，每一个消息控制块将对应一个消息。

### 3.1.3 LOS\_QueueCreate( )函数

传入参数：

\*pcQueueName：要创建的消息队列的名称（程序中未做任何处理，为将来留用）。

usLen：消息队列中的消息节点的个数。

\*puwQueueID：用于保存最后创建完成的消息的ID号，后续由用户使用。

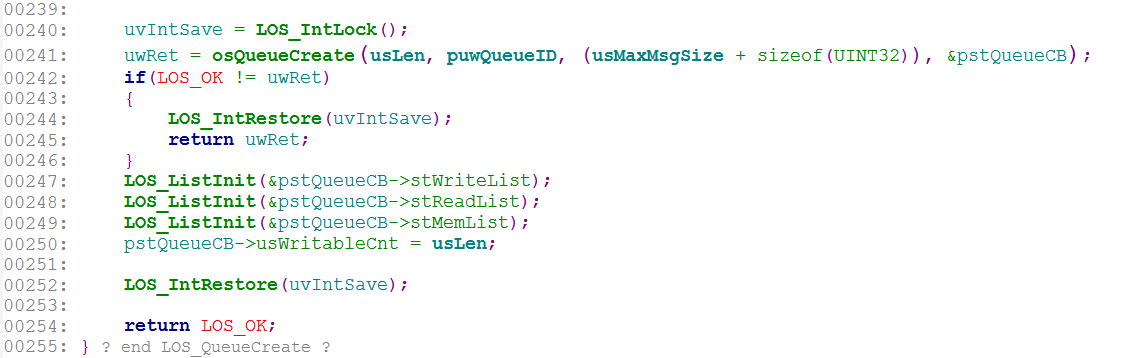
uwFlags：消息队列的类型，FIFO或PRIO方式（程序中未做任何处理，默认为FIFO模式）。

usMaxMsgSize：每个消息节点的大小。

功能：

创建一个消息队列时，包含usLen个消息节点，每个消息节点的大小为usMaxMsgSize。





Line 241：调用函数osQueueCreate( )创建一个消息队列，创建成功以后的消息控制块通过参数&pstQueueCB接收。

Line 247 - 249：初始化写消息阻塞队列、读消息阻塞队列、用于邮件队列的申请内存块阻塞队列均为空队列。

Line 250：设置队列中可写消息个数为usLen（初始时消息队列为空，所以可写值为消息节点个数的最大值），默认的可读消息个数为0。

备注：调用osQueueCreate( )创建消息队列时传入的参数：usLen是消息节点的个数。puwQueueID用于接收消息的ID号，该ID号直接返回给用户使用，用户通过消息ID号操作对应的消息队列。(usMaxMsgSize + sizeof(UINT32))是每个消息节点的大小，每个节点加4字节的原因：写消息时只写入一个UINT32的数据或指针，这个数据写入消息节点最开始的位置，读消息时对消息有效数据的操作也只是读取这个UINT32的数据或指针，至于这个UINT32的数据或指针代表的具体含义，由收发消息的双方自行定义，为了保证这个UINT32的数据或指针能够正确写入，多加了4个字节的空间，实际上usMaxMsgSize所指示的空间目前并未使用。&pstQueueCB用于接收创建好的消息控制块。

### 3.1.4 osQueueCreate( )函数

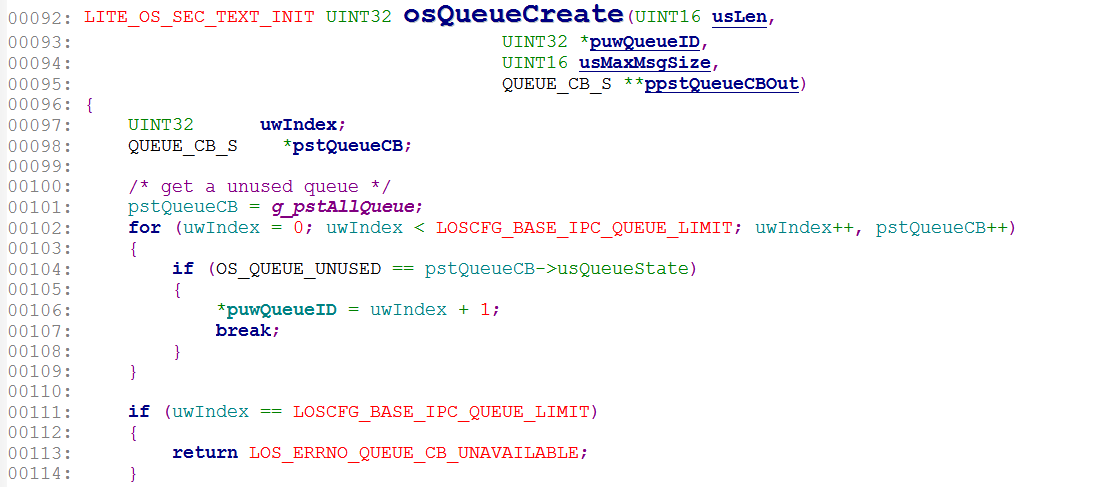
传入参数：

usLen：消息队列中消息节点的个数。

\*puwQueueID：用于保存最后创建好的消息的ID号，进而返回给用户，最终用户通过该ID号操作对应的消息队列。

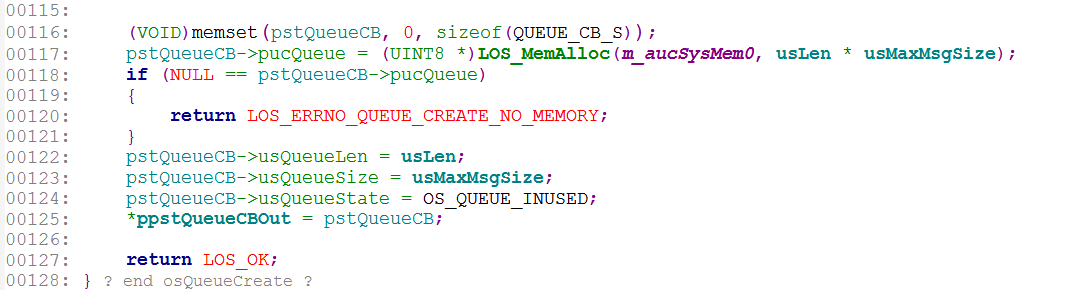
usMaxMsgSize：每个消息节点的大小。

\*\*ppstQueueCBOut：用于给创建者返回创建好的消息控制块。



Line 102 - 109：OS\_QUEUE\_UNUSED定义为0，从g\_pstAllQueue中取出一个消息号最小的未使用的消息队列控制块，并设置消息ID号传给\*puwQueueID。

注意，此处的ID号是在g\_pstAllQueue下标的基础上加了1，所以在对消息队列进行操作时（比如读队列、写队列、删除队列等）需要减去1后再根据ID号查找对应的消息控制块。消息ID号并不存储在QUEUE\_CB\_S中。



Line 117：usLen \* usMaxMsgSize = 消息队列的节点个数 \* 节点大小，分配出来的这块内存实际是一个下标为usLen，每个元素的大小为usMaxMsgSize的数组，数组按照下标组成一个消息循环队列，由pstQueueCB->usQueueHead指示循环队列的队头，由pstQueueCB->usQueueTail指示循环队列的队尾，初始时队头和队尾都指向下标0，表示循环队列为空。

备注：Head和Tail指向相同的数组下标表示队列为空，Tail指向位置的下一个位置为Head表示队列为满，但是这样的话需要浪费一个空间作为队列为空或为满的条件，所以在此处的应用环境下不使用这种方式判断队列为空还是满，而是使用usWriteableCnt和usReadableCnt组合作为队列为满或为空的判断条件。

Line 122：设置队列中消息个数pstQueueCB->usQueueLen = usLen。

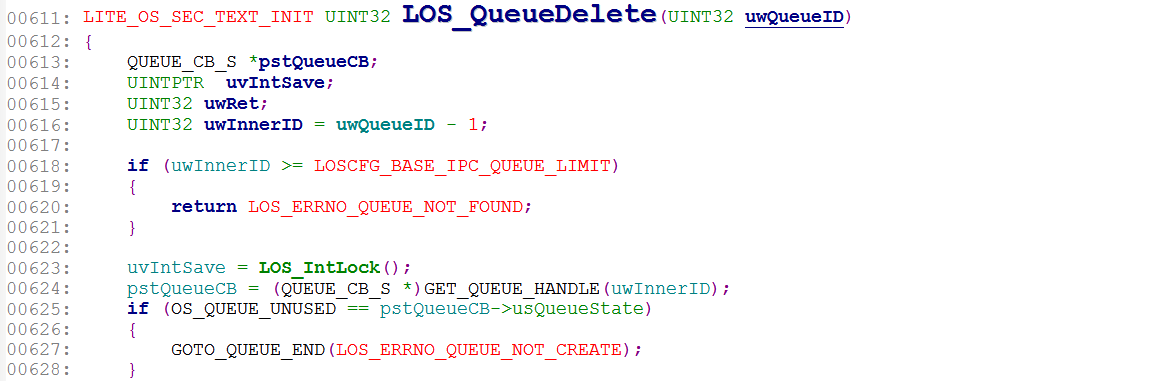
Line 123：设置每个消息节点的大小pstQueueCB->usQueueSize = usMaxMsgSize。

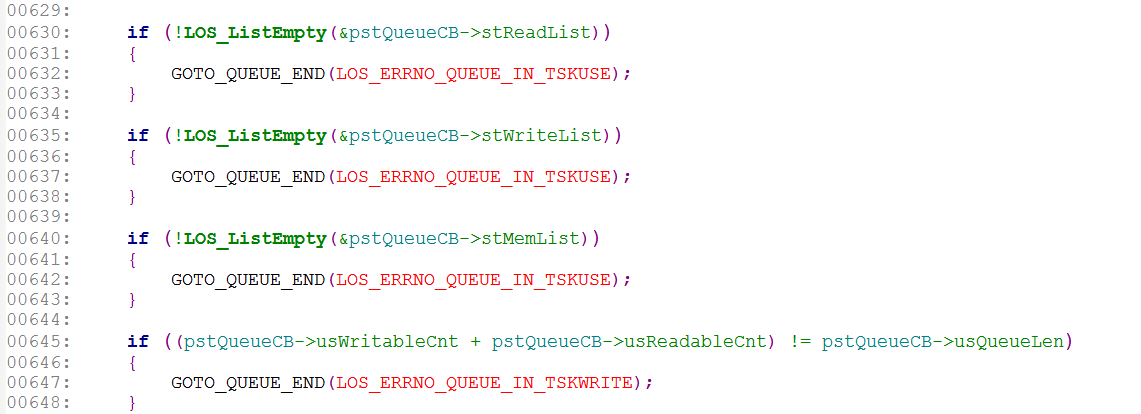
Line124：设置消息队列的状态为pstQueueCB->usQueueState = OS\_QUEUE\_INUSED，OS\_QUEUE\_INUSED被定义为1，表示该消息控制块已经被使用。

Line 125：最后将新分配出的消息控制块通过\*ppstQueueCBOut传送给调用者，由调用者处理后续的事情。

### 3.1.5 LOS\_QueueDelete( )函数

删除消息队列时传入消息队列的ID号。





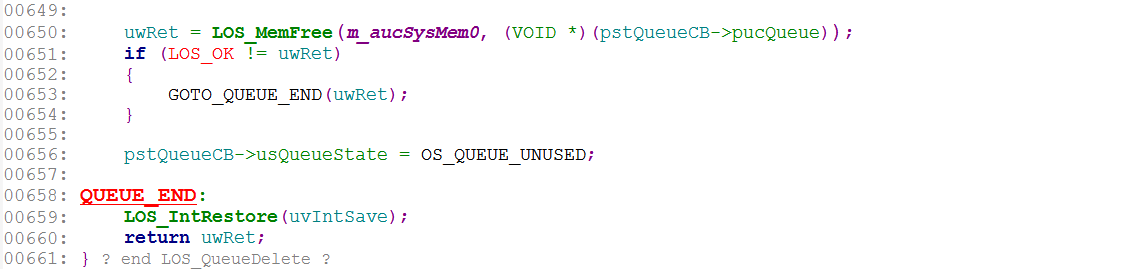
Line 630：读阻塞队列不为空，说明有TASK在这个消息队列上因未读到消息而进入阻塞状态。

Line 635：写阻塞队列不为空，说明有TASK向这个消息队列写入消息时因消息队列满而被挂起。

Line 640：申请内存块阻塞队列不为空，说明有TASK因使用邮件队列而申请静态内存池中的内存块失败而被挂起。

Line645：可写的消息个数加可读的消息个数与总消息个数不相等，说明当前有TASK正在写消息到这个消息队列中。

在这些情况下，说明还有TASK在使用这个消息队列，所以不能删除。



Line 650 - 656：删除消息队列时只需要将申请的消息节点个数和节点大小的相关内存释放并设置该消息为OS\_QUEUE\_UNUSED即可，其他状态不需要再关心，因为在创建消息队列时OS\_QUEUE\_UNUSED作为这个消息未被使用的标志被判断，其他的状态会在创建时清除并重新赋值。

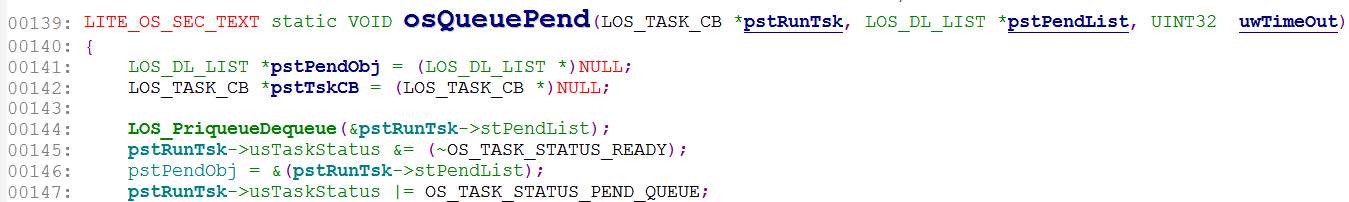
### 3.1.6 osQueuePend( )函数

因为与消息队列相关的原因挂起一个正在运行的TASK，传入参数：

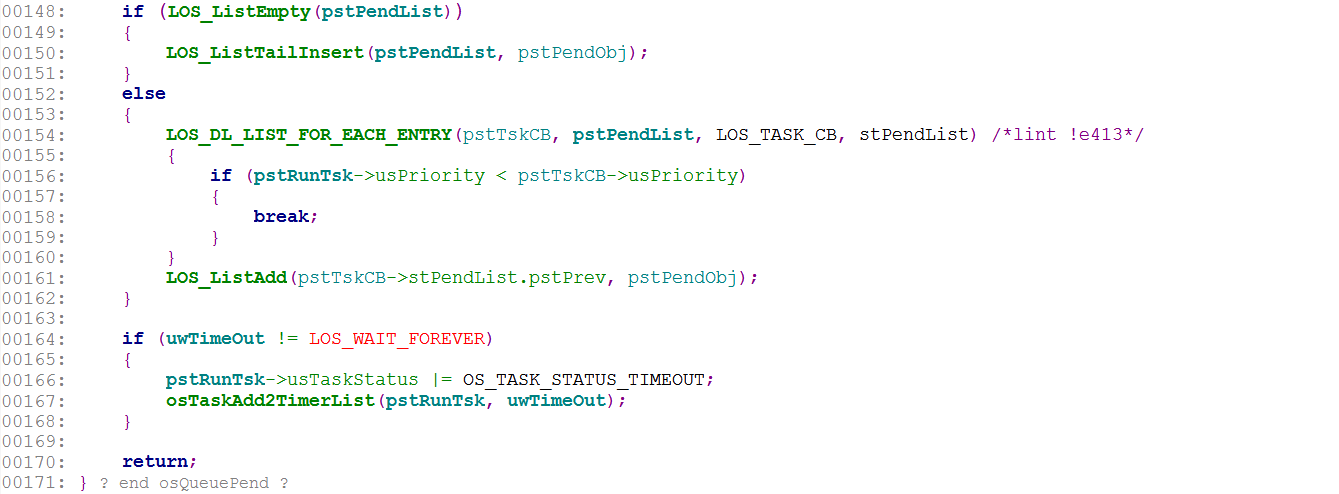
\*pstRunTsk：表示要挂起的正在运行的TASK。

\*pstPendList：挂起队列，表示将TASK挂起到这个队列中。

uwTimeOut：超时时间，若该TASK在超时时间内未被唤醒，则由超时唤醒挂起的TASK。



Line 144 - 147：将要挂起的正在运行的TASK从优先级队列取下，这样挂起以后就不会再参与系统调度，并清除OS\_TASK\_STATUS\_READY状态，然后设置挂起TASK的状态为OS\_TASK\_STATUS\_PEND\_QUEUE，即该TASK处于挂起到队列状态。



Line 148 - 151：如果挂起队列为NULL，则直接将要挂起的TASK插入到挂起队列的尾部。

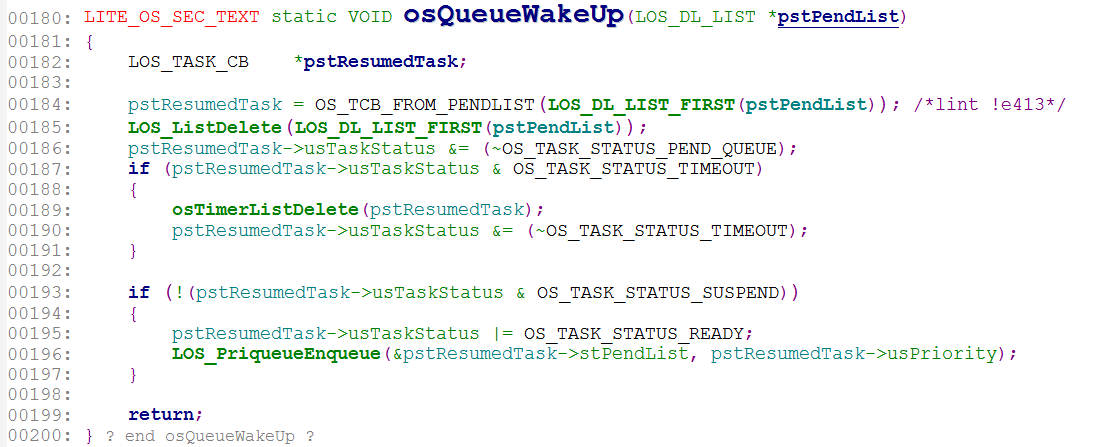
Line 154 - 161：若挂起队列不为空，则需要按照优先级由高到低的顺序（值越小，优先级越高）插入到挂起队列中。

Line 164 - 168：若有挂起超时时间限制，则需要同时将该TASK加入到超时等待列表中，并置状态为OS\_TASK\_STATUS\_TIMEOUT，表示该TASK处于超时队列中。这样的话，该TASK同时拥有OS\_TASK\_STATUS\_PEND\_QUEUE状态和OS\_TASK\_STATUS\_TIMEOUT状态，其中任何一个都能唤醒该TASK，唤醒以后需要清除另外的阻塞标志。

### 3.1.7 osQueueWakeUp( )函数

因为与队列相关的原因唤醒挂起队列中的第一个TASK（一次只能唤醒一个TASK，而且是优先级最高的那一个TASK）。

传入参数：\*pstPendList是挂起队列，表示从该队列中唤醒优先级最高的那个TASK。



Line 185 - 186：唤醒时将挂起队列的第一个TASK从挂起队列删除，并清除状态PEND\_QUEUE。

Line 187 - 191：如果挂起时该TASK同时被加入到了超时队列，则需要将该TASK从超时队列删除，并清除TIMEOUT状态。

Line 193 - 197：唤醒后如果该TASK不处于SUSPEND状态，则需要将该TASK重新加入到优先级队列的队尾参与系统调度。此处只是唤醒TASK，并不立即产生系统任务切换。

### 3.1.8 LOS\_QueueRead( )函数

从指定ID号的消息队列中读取一个消息，该函数为不带拷贝读取，即没有信息被复制到该消息体内。

传入参数：

uwQueueID：消息的ID号，对应于指定的消息队列。

\*pBufferAddr：用于保存消息数据，实际只能读到一个UINT32类型的数据或指针，这个数据或指针的具体含义由通信双方自行定义。

uwBufferSize：要读取的消息数据的大小，在函数内部未做处理，故该参数用意不明。

uwTimeOut：读取消息的超时时间，同时也是读取消息的阻塞模式，包括无阻塞模式、定时阻塞模式、永久阻塞模式。



Line 276：创建消息时ID加1，所以此处需要减1作为真正的消息ID号。

Line 293 - 299：在中断中只能以无阻塞的方式读取消息，因为中断中不允许因阻塞而挂起TASK，这样读不到消息以后会立即返回，不会影响中断的执行。

**备注：**中断中可以读取消息，但是只能以无阻塞模式读取。TASK中读取消息无此限制。



Line 309：pstQueueCB->usReadableCnt = 0表示消息队列中没有可读消息：

Line 311：uwTimeOut = 0说明用户不希望读消息等待，则立即返回消息队列为空的状态。

Line 316：用户请求读消息等待时，因为没有消息可读，所以需要将当前TASK挂起到读阻塞队列等待唤醒，挂起后需要切换任务调度，但是如果任务调度被锁，则无法切换任务，所以直接返回在任务调度被锁的情况下不允许挂起TASK到阻塞队列的错误。

Line 322：调用函数osQueuePend( )将该TASK挂起到该ID号对应的消息队列的读阻塞队列中。

Line 324：挂起任务后启动一次任务切换调度新的TASK执行。

调用函数osQueuePend( )执行挂起以后，当前的TASK已经从优先级队列中取出并进入了挂起队列，不再具备参与调度的条件。调用函数LOS\_Schedule( )执行任务切换以后，CPU调度新的任务执行，当前的这个TASK的执行位置被停在了LOS\_Schedule( )函数的后面一条语句上（Line 326），（因为每一个TASK都有自己的栈空间，所以多个TASK调用同一个函数时会有多个该函数的现场，各个TASK对该函数的调用互不影响）。

该TASK被唤醒以后（通过消息队列的写条件唤醒或者超时唤醒）继续向下执行：

当调用函数osQueueWakeUp( )唤醒时会清除PEND\_QUEUE状态和TIMEOUT状态并从两个阻塞队列中将该TASK删除，此时表示有消息可读，向下执行去读取消息。

Line 328 - 332：当TIMEOUT唤醒时会将该TASK从两个阻塞队列中删除，但是只清除了PEND\_QUEUE状态，而没有清除TIMEOUT状态，具体参见osTaskScan( )函数。所以当被唤醒后若该TASK处于TIMEOUT状态，说明该TASK被超时唤醒，没有消息可读取，所以需要在此处清除TIMEOUT状态并返回错误，错误码为超时未读取到消息。

Line 336：如果消息队列中已经有可读消息，即可读计数器不等于0，则将可读计数减1，去读取消息。

注意：在被唤醒后去读取消息时没有执行pstQueueCB->usReadableCnt-- 这条语句，是因为在LOS\_QueueWrite( )函数中写消息时若已经有TASK处于阻塞状态，则直接唤醒，不执行pstQueueCB->usReadableCnt++ 操作，所以此处不能有减的操作。



Line 339：消息节点队列是一个循环队列，所以需要通过usQueueHead \* usQueueSize找到消息队列的头节点的位置。

Line340：找到后将头节点中的有效数据取出。

Line 341：通过++pstQueueCB->usQueueHead将头节点上的消息删除。

注意：读取消息时只读取消息节点最开始的一个UINT32数据，存放于pBufferAddr中，这个数据也就是写消息TASK在写消息时唯一写入的消息内容，这个UINT32数据的具体含义由收发消息的双方自行定义，比如可以是一个缓冲区的地址，也可以是一个UINT32的实际数据等。

Line 346 - 348：：取出消息后消息队列又多了一个空出的位置，如果写阻塞队列不为空，说明在读取消息之前已经有TASK因为消息队列满而无法写入进而被挂起，所以需要调用osQueueWakeUp( )函数唤醒写阻塞队列中的一个TASK去写入消息。

Line 350：唤醒TASK后启动一次任务切换，以保证高优先级任务的实时响应。

注意：此处唤醒时未执行pstQueueCB->usWritableCnt++ 操作，所以在写任务写消息之前也不需要执行pstQueueCB->usWritableCnt-- 操作。

Line 356：读取消息后如果没有写阻塞的TASK存在，则直接执行pstQueueCB->usWritableCnt++ 操作表示可写位置又多了一个。

### 3.1.9 LOS\_QueueWrite( )函数

向指定ID号的消息队列中写入一个消息，该函数实现不带拷贝写入。

传入参数：

uwQueueID：消息的ID号。

\*pBufferAddr：保存消息的有效数据，写消息函数会将pBufferAddr指针以UINT32的形式写入消息中，对方会得到这个UINT32的数据，至于这个数据代表的具体含义由双方自行定义。

uwBufferSize：要写入的消息信息的大小。

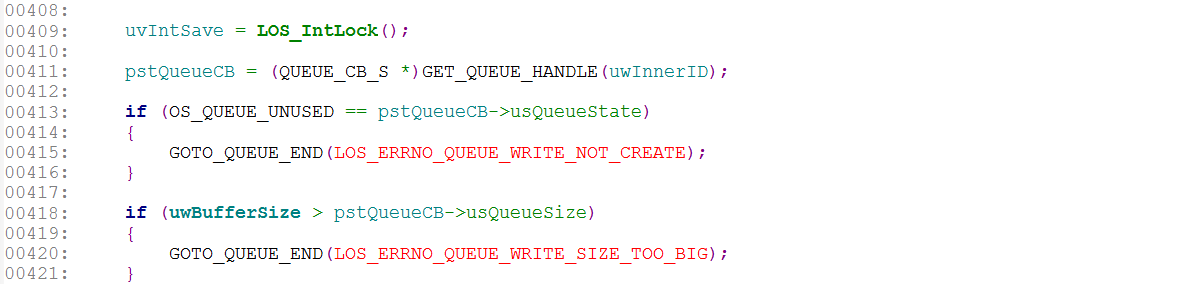
uwTimeOut：写入超时时间，同时也是写消息队列的阻塞模式，包括无阻塞模式、定时阻塞模式和永久阻塞模式。



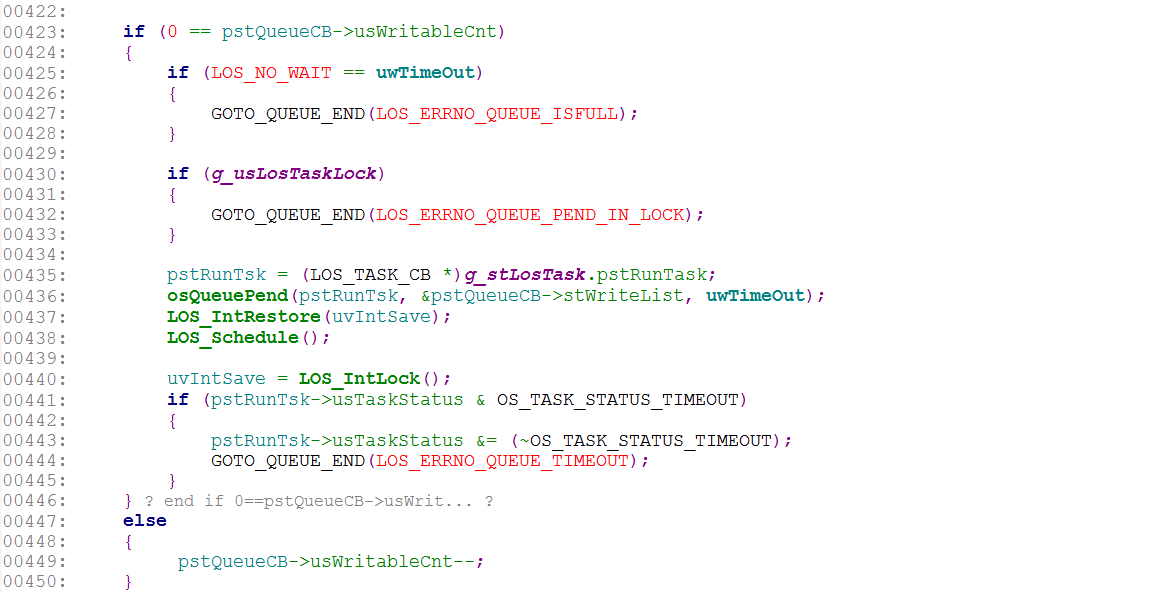
Line 393：若pBufferAddr == NULL，则会将地址0以UINT32的形式写入到消息节点中，对方会得到0，至于0的含义由通信双方自行定义。故此处不能返回。

Line401 - 407：在中断中只能以无阻塞的方式写入消息，因为中断中不允许因阻塞而挂起TASK，这样无法写入消息时会立即返回，不会影响中断的执行。

**备注：**中断中可以发送消息，但是只能以无阻塞模式发送，TASK中发送消息无此限制。



Line 418：要写入的数据的大小不能超过消息节点的大小。



Line 423：pstQueueCB->usWritableCnt = 0说明该ID号对应的消息队列已满，无法写入，此时：

Line 425：若用户不需要等待写入，则直接返回消息队列满的状态。

Line 430：若用户需要等待写入，则需要将该TASK挂起到写阻塞队列中等待唤醒，挂起后需要切入新的TASK执行，如果调度被锁，则不能挂起。

Line 436：调用函数osQueuePend( )将正在执行的TASK挂起到该ID号对应的消息的写阻塞队列中。

Line 438：挂起当前正在运行的TASK后启动一次任务切换调度新的TASK执行。

调用函数osQueuePend( )执行挂起以后，当前的TASK已经从优先级队列中取出并进入了挂起队列，不再具备参与调度的条件。调用函数LOS\_Schedule( )执行任务切换以后，CPU调度新的任务执行，当前的这个TASK的执行位置被停在了LOS\_Schedule( )函数的后面一条语句上（Line 440），（因为每一个TASK都有自己的栈空间，所以多个TASK调用同一个函数时会有多个该函数的现场，各个TASK对该函数的调用互不影响）。

当该TASK被唤醒后（通过消息队列的读条件唤醒或者超时唤醒）继续向下执行：

Line 441 - 445：当TIMEOUT唤醒时会将该TASK从两个阻塞队列中删除，但是只清除了PEND\_QUEUE状态，而没有清除TIMEOUT状态，具体参见osTaskScan( )函数。所以当被唤醒后若该TASK处于TIMEOUT状态，说明该TASK被超时唤醒，消息队列为满而无法写入，所以需要在此处清除TIMEOUT状态并返回错误，错误码为超时未正确写入消息。

当调用函数osQueueWakeUp( )唤醒时会清除PEND\_QUEUE状态和TIMEOUT状态并从两个阻塞队列中将该TASK删除，此时表示有空位，可以继续写入消息，向下执行去写入消息。

Line 449：如果消息队列不为满，即可写计数器不等于0，则将可写计数减1，去写入消息。

注意：在被唤醒后去写入消息时没有执行pstQueueCB->usWriteableCnt-- 这条语句，是因为在LOS\_QueueRead( )函数中读消息后若已经有TASK处于阻塞状态，则直接唤醒，不执行pstQueueCB->usWriteableCnt++ 操作，所以此处不能有减的操作。



Line 452：消息节点队列是一个循环队列，所以需要通过usQueueTail \* usQueueSize找到消息节点的尾部位置。

Line 453：找到尾部后将要写入的数据写入到usQueueTail所指向的节点的数据区域，实际上是将一个指针信息以UINT32的形式写入。

Line455：并通过++pstQueueCB->usQueueTail将队列的尾部位置向后移一个节点。

注意：写消息时只在消息节点的开始位置将pBufferAddr代表的指针以UINT32数据的方式写入，至于这个数据具体代表什么含义，由收发消息的双方自行定义，比如可以代表一个缓冲区的起始地址，也可以代表一个UINT32的实际数据等。

Line460 - 462：写入消息后消息队列又多了一个可读的消息，如果读阻塞队列不为空，说明在写入消息之前已经有TASK因为消息队列空而无法读取进而被挂起，所以需要调用osQueueWakeUp( )函数唤醒读阻塞队列中的一个TASK去读取消息。

Line 464：启动一次任务切换，以保证高优先级任务的实时性。

注意：此处唤醒时未执行pstQueueCB->usReadableCnt++ 操作，所以在读任务读消息之前也不需要执行pstQueueCB->usReadableCnt-- 操作。

Line 469：写入消息后如果没有读阻塞的TASK存在，则直接执行pstQueueCB->usReadableCnt++ 操作表示消息队列中可读的消息又多了一个。

### 3.1.10 osQueueMailAlloc( )函数

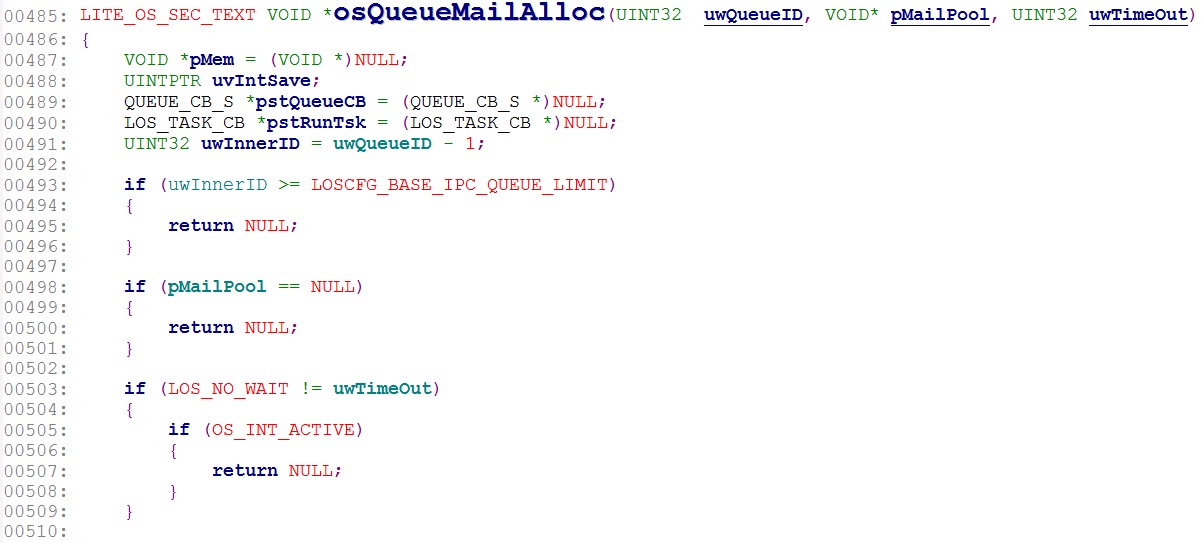
功能：从邮件中分配一个内存块。从\*pMailPool指示的静态内存池中申请一块内存块，若无法申请需要阻塞，则将该TASK阻塞在ID号为uwQueueID的消息队列的stMemList链表上，也就是阻塞在邮件队列上。

传入参数：

uwQueueID：消息队列的ID号。

\*pMailPool：静态内存池首地址。

uwTimeout：获取超时，同时表示获取模式：无阻塞模式获取、定时阻塞模式获取和永久阻塞模式获取。



Line 505：中断中只能以无阻塞模式获取内存块，获取不到立即返回。



Line 513：从pMailPool指针指示的静态内存池中获取一块内存块。

Line 516：无阻塞模式下获取不到立即返回。

Line 522：将该TASK阻塞到uwQueueID指示的消息队列的stMemList链表上，并设置超时状态。表示该TASK因未获取到内存块而被阻塞在消息队列的stMemList链表上。

Line 524：切换任务，CPU调度新的TASK执行，当前TASK的执行被停在了Line 526。当该TASK被唤醒后，从Line 526开始继续向下执行。

Line 527：该TASK被超时唤醒，表示超时时间之内未获取到内存块，清除TIMEOUT状态并返回NULL。

Line 532：该TASK被osQueueMailFree( )函数唤醒。osQueueMailFree( )函数唤醒该TASK时会从内存池中申请内存块，并将申请结果填入该TASK的puwMsg处。

Line 534：osQueueMailFree( )函数未申请到内存块给要唤醒的这个TASK。

Line 538 - 539：将osQueueMailFree( )函数的申请结果返回给用户。

### 3.1.11 osQueueMailFree( )函数

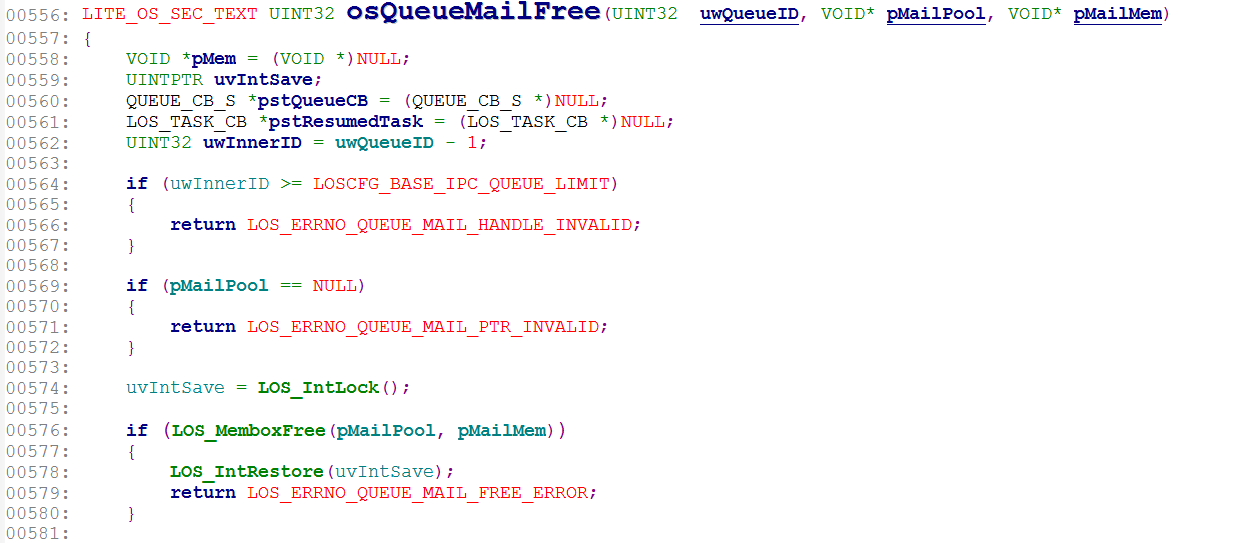
功能：释放一个邮件中的内存块。向内存池中释放一块内存块，并根据条件唤醒因未获取到该内存池中的内存块而阻塞在邮件队列上的TASK。

传入参数：

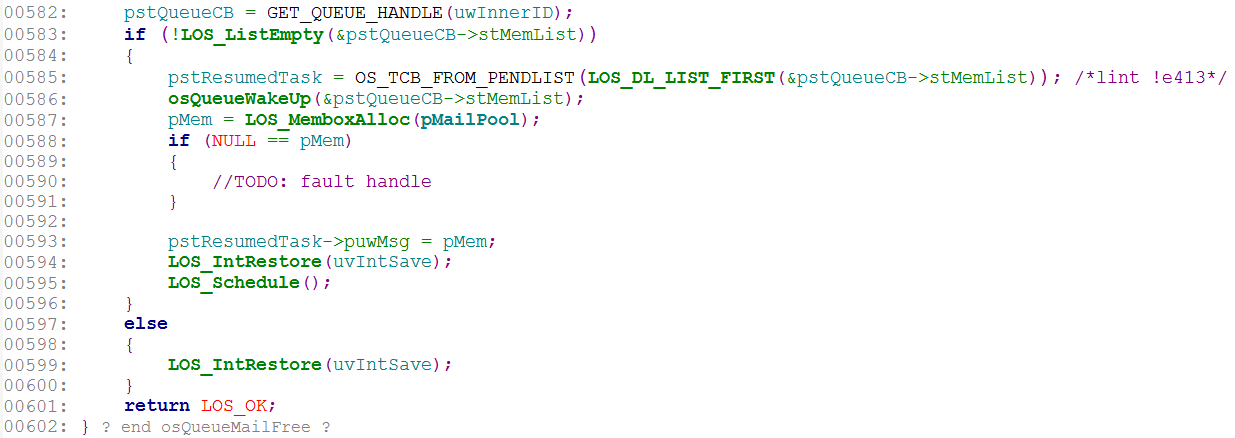
uwQueueID：消息队列ID号。

\*pMailPool：指示内存块对应的静态内存池。

\*pMailMen：指示要释放的内存块。



Line 576：将pMailMem指示的内存块释放回pMailPool指示的静态内存池中。



Line 583：stMemList阻塞链表非空，表示有因获取内存块而阻塞的TASK存在，需要唤醒它们。

Line 585 - 586：唤醒处于阻塞队列队头的TASK，每次释放只能唤醒一个TASK。

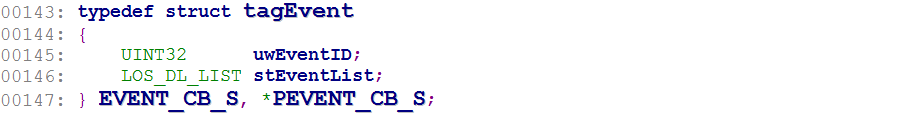
Line 587：为要唤醒的这个TASK申请内存块。

Line 593：将申请结果填入要唤醒的这个TASK的puwMsg处，这样被唤醒的TASK继续执行以后从TASK对应的TCB的puwMsg处去获取内存块。

Line 595：切换一次任务，CPU调度当前优先级队列中优先级最高的TASK执行。

## 3.2 事件

### 3.2.1 EVENT\_CB\_S结构解析

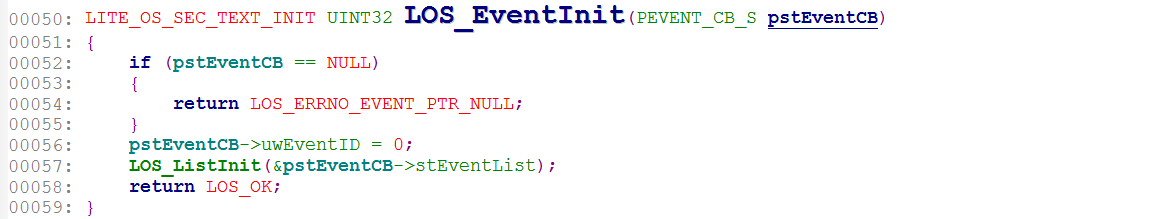


uwEventID：标识发生的事件类型位，每一位代表一种事件类型，第25位保留，一共31种事件类型。初始化为0表示没有事件发生，当有事件发生时，对应的事件标志位置1。

stEventList：读取事件任务链表，也就是等待事件阻塞队列，当有TASK需要等待事件发生时会被阻塞进入该队列。

### 3.2.2 LOS\_EventInit( )函数

事件初始化，传入事件控制块，每一个事件控制块最多可管理31种类型的事件，可以创建多个事件控制块。

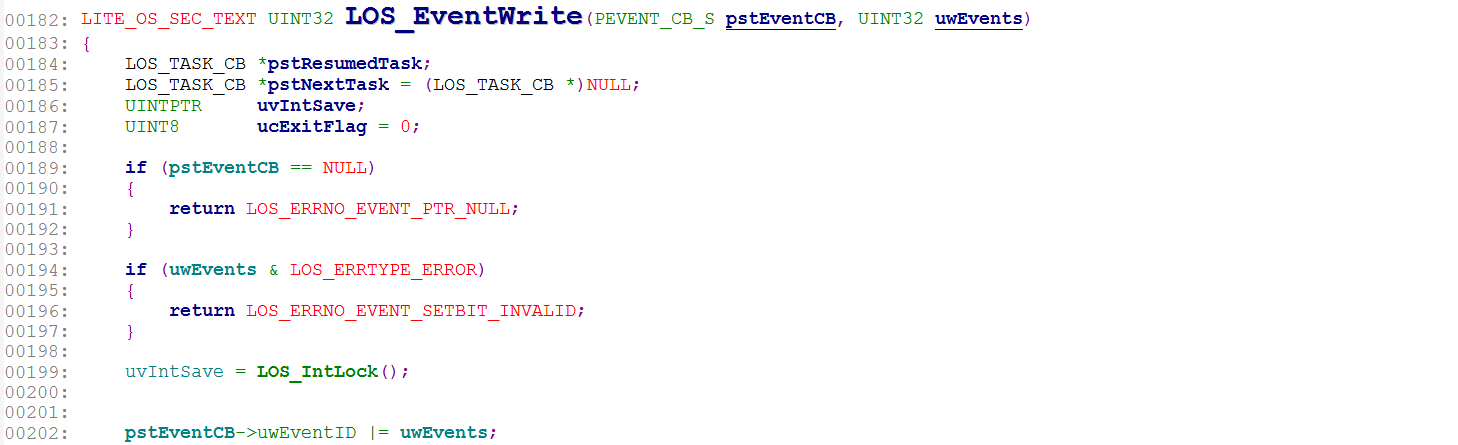


### 3.2.3 LOS\_EventWrite( )函数

写一个事件到对应的事件控制块，表示该事件发生。

传入参数：pstEventCB是事件控制块。

uwEvents是具体事件。



Line 194：LOS\_ERRTYPE\_ERROR被定义为((UINT32)0x02 << 24)，是第25位，为系统保留位。

Line 202：某一类型的事件发生时，将该事件发生的标识填入相应的标识位即可，多次填入同一个标识位相当于只填入了一次，即该事件在没有被清除的状况下多次发生相当于只发生了一次。





Line 203：当有事件发生时，如果读取事件任务链表不为空，则说明有TASK因等待某一或某些事件的发生而进入阻塞状态，需要查询并根据条件唤醒阻塞的TASK。

Line 205 - 206：从事件任务链表（也就是等待事件阻塞队列）的头开始依次扫描列表中的所有TASK，直到扫描到尾部为止。

Line 210 - 211：如果当前TASK的事件等待模式是OR的关系，则只需要等到其中任意一个事件发生就可以唤醒该TASK，如果当前TASK的事件等待模式是AND的关系，则需要等到所有的事件发生以后才能唤醒该TASK。

Line 215 - 221：唤醒TASK时将要唤醒的TASK从事件任务链表中删除，并清除PEND（阻塞）状态，若该TASK设置了等待超时，则从超时队列删除并清除TIMEOUT状态。

Line 223 - 227：最后若TASK不处于SUSPEND状态，则置READY状态并将该TASK加入到优先级队列等待调度。

Line 235：满足条件时启动一次任务调度。

Line 242：如果读取事件任务链表为空，则置事件发生标志位后直接返回。

### 3.2.4 LOS\_EventRead( )函数

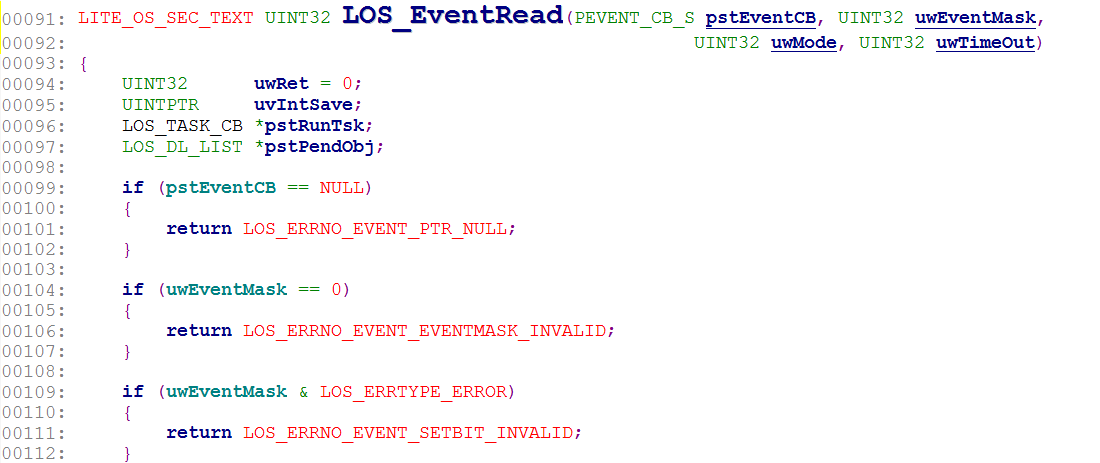
事件读取，TASK等待特定的事件发生。

传入参数：pstEventCB：事件对应的事件控制块。

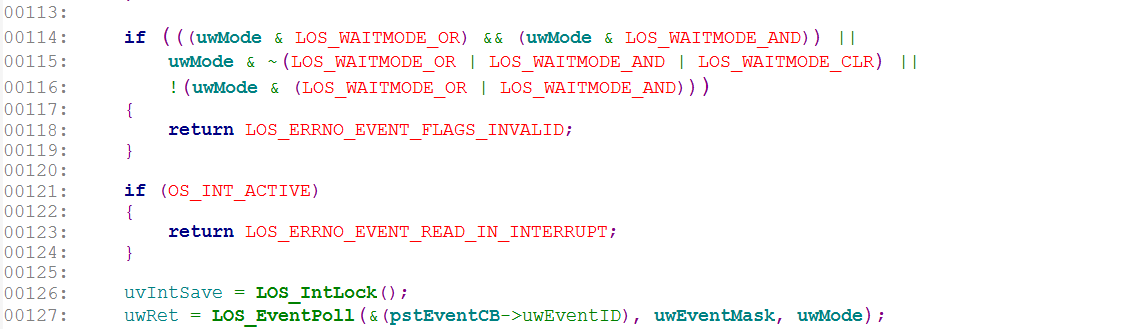
uwEventMask：用户要读取的事件号。

uwMode：读取模式。

uwTimeOut：读取超时时间。



Line 109：第25位系统保留，不允许读取。



Line 114 - 116：读取模式的有效性检测，OR和AND模式不能并存，OR、AND、CLR以外的模式无效，OR和AND两个都没有为无效。

Line 121：中断中不能读取事件。

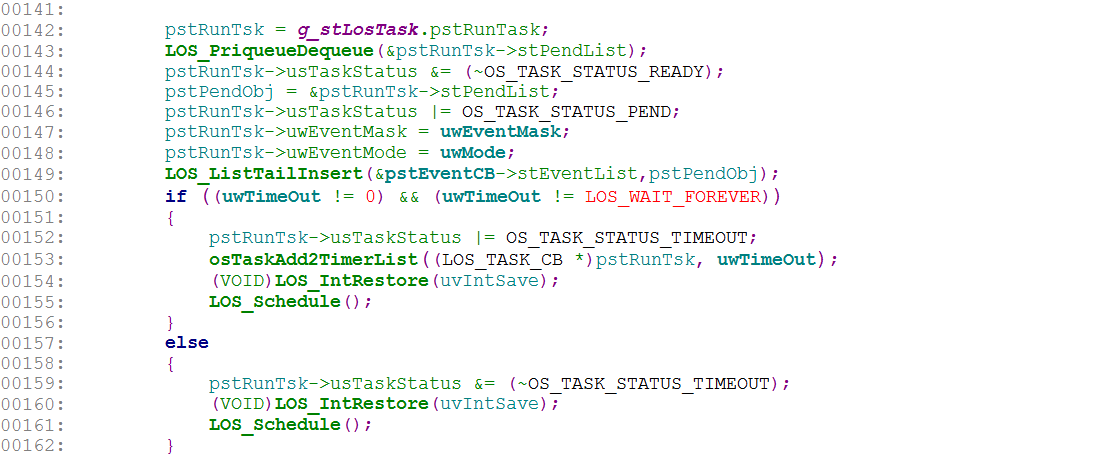
Line 127：调用函数LOS\_EventPoll( )判断是否已经有符合用户要求的事件发生，若返回非0，则说明用户等待的事件已经发生了，也满足用户等待的条件，故读取事件成功，直接返回，这样用户拿到返回值以后就知道是什么事件发生了。



Line 129：uwRet == 0说明用户读取事件的条件不满足，需要进入阻塞队列等待事件发生。

Line 131：用户不希望读取事件延时，则直接返回未读取到事件的状态码（0表示没有任何事件发生）。

Line 136：系统调度被锁状态下不允许等待读取事件。



延时等待读取事件：

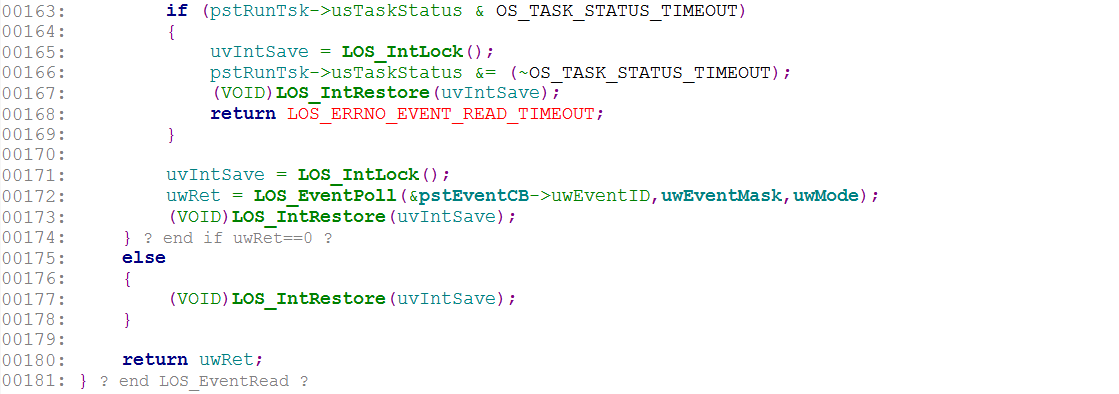
Line 143 - 146：将当前正在运行的TASK从优先级队列取下，这样就不会再参与调度，清除READY状态，设置PEND（阻塞）状态。

Line 147 - 148：并设置该TASK要等待的事件掩码位和模式位（这两个信息在写事件函数中会被判断到）。

Line 149：将该TASK加入到等待事件阻塞队列的尾部，进而等待被写事件函数唤醒。

Line 150 - 162：若用户设置了读取超时事件，则需要同时将该TASK加入到超时队列中，并设置TIMEOUT状态，然后启动一次任务切换。若用户设置超时为LOS\_WAIT\_FOREVER，则清除TIMEOUT状态（此处是否有必要写这条语句）并启动一次任务切换。

Line 155 or 161：调用LOS\_Schedule( )切换任务后，CPU调度新的TASK执行，当前这个TASK因读事件阻塞而被停在了LOS\_Schedule( )后面的一条语句上（Line 163），（因为每一个TASK都有自己的栈空间，所以多个TASK调用同一个函数时会有多个该函数的现场，各个TASK对该函数的调用互不影响）。



当这个TASK被唤醒后（写事件函数唤醒或超时唤醒），继续向下执行：

如果是被超时唤醒，则在超时处理时将该TASK从超时队列中移除，清除PEND状态并从阻塞队列中移除，然后加入到优先级队列参与调度，但并未清除TIMEOUT状态。具体参见osTaskScan( )函数。

Line 163 - 168：被唤醒后若该TASK处于TIMEOUT状态，则说明超时未等到想要的事件发生，清除TIMEOUT状态后返回超时未读到事件的错误。

Line 172：该函数被写事件唤醒，调用函数LOS\_EventPoll( )来获取到底是哪个或哪几个事件发生了。并将获取到的结果返回给用户。

### 3.2.5 LOS\_EventPoll( )函数

功能：根据用户传入的事件值、事件掩码及校验模式，返回用户传入的事件是否符合预期，即判断是否已经有符合用户要求的事件发生，若无，返回0，若有，返回具体已经发生的事件的位号，用户得到该位号以后就知道哪个或哪些事件发生了。

传入参数：

\*uwEventID：表示已经发生的事件的位号，某一事件发生则对应位被置1，否则置0。

uwEventMask：表示用户要读取事件的位号，用户等待哪些事件发生就将该位置1。

uwMode：校验模式，有OR、AND、CLR三种，OR和AND只能二选其一。



Line 67 - 80：分别按照OR和AND的规则检查用户希望的事件是否已经发生及是否满足用户的判断条件，并给出返回值，返回值表示哪些事件发生了。

Line 82：若uwMode设置了CLR，则说明用户希望在读取事件以后清除这些事件的发生标志位，以使下一次事件的发生能被正确检测。若用户没有设置CLR位，则最后需要用户自己显示清除事件标志位，否则下一次事件的发生无法正确检测。

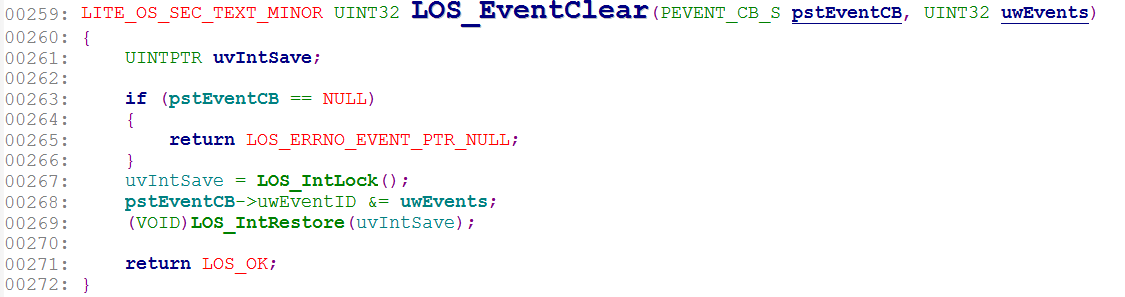
### 3.2.6 LOS\_EventClear( )函数

功能：用户调用该函数显示清除事件发生标志位，隐式清除在LOS\_EventPoll( )函数中完成。

传入参数：

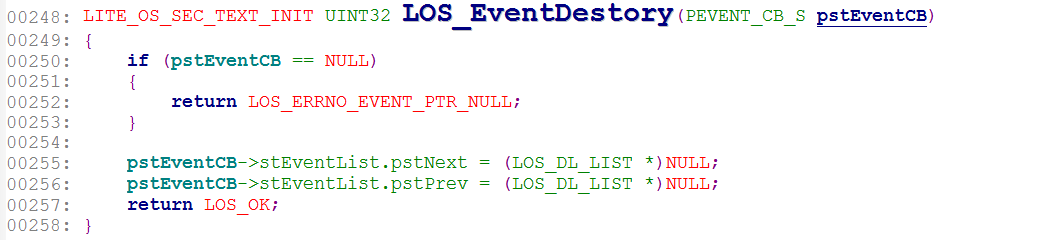
pstEventCB：事件控制块。

uwEvents：要清除的事件标志位号，注意清除函数的写法（pstEventCB->uwEventID &= uwEvents），故需要在传入uwEvents时将需要清除的位置0，其它不清除的位置1。



### 3.2.7 LOS\_EventDestory( )函数

销毁一个事件控制块，这个函数的实现不完整，缺少一些必要的判断条件（比如事件阻塞队列是否为空的判断等），故不能直接使用。



# IPC之任务同步(互斥锁和信号量)

## 4.1 优先级翻转问题概述

优先级翻转是当一个高优先级任务通过信号量机制访问共享资源时，该信号量已被一低优先级任务占有，因此造成高优先级任务被许多具有较低优先级任务阻塞，实时性难以得到保证。

例如：有优先级为A、B和C三个任务，优先级A > B > C，任务A，B处于挂起状态，等待某一事件发生，任务C正在运行，此时任务C开始使用某一共享资源S。在使用中，任务A等待事件到来，任务A转为就绪态，因为它比任务C优先级高，所以立即执行。当任务A要使用共享资源S时，由于其正在被任务C使用，因此任务A被挂起，任务C开始运行。如果此时任务B等待事件到来，则任务B转为就绪态。由于任务B优先级比任务C高，因此任务B开始运行，直到其运行完毕，任务C才开始运行。直到任务C释放共享资源S后，任务A才得以执行。在这种情况下，优先级发生了翻转，任务B先于任务A运行，任务执行顺序为C -> A -> C -> B -> C -> A，如果类似B这样的任务有很多的话，A的执行就会被无限推迟。

解决优先级翻转问题有优先级天花板(priority ceiling)和优先级继承(priority inheritance)两种算法。

优先级天花板是当任务申请某资源时，把该任务的优先级提升到可访问这个资源的所有任务中的最高优先级，这个优先级称为该资源的优先级天花板。这种方法简单易行，不必进行复杂的判断，不管任务是否阻塞了高优先级任务的运行，只要任务访问共享资源都会提升任务的优先级。

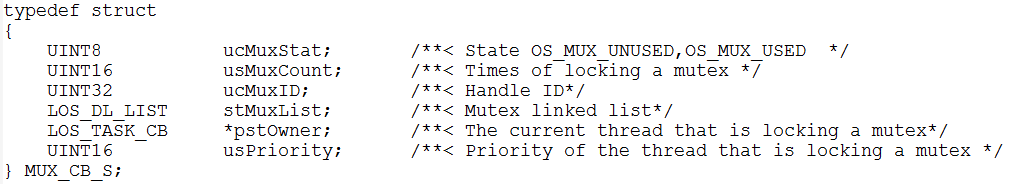
用优先级天花板算法解决上述问题：资源S的优先级天花板与访问S的最高任务的优先级相同，即与A的优先级相同，任务C在申请共享资源S时优先级被直接提到与A相同，此时即便A处于就绪态，按照同优先级轮转的调度算法，C的执行也不会被打断，直到C释放S后回到原有的优先级，所以最后的执行顺序是C -> A ->B。

优先级继承是当任务A申请共享资源S时，如果S正在被任务C使用，通过比较任务C 与自身的优先级，如发现任务C的优先级小于自身的优先级，则将任务C的优先级提升到自身的优先级，这样C的优先级暂时高于B，即便B进入就绪状态，但因优先级暂时低于C而不能运行，任务C继续执行，当任务C释放资源S后，再恢复任务C的原优先级。此时因为就绪队列中A的优先级高于B，所以先执行A，再执行B，使得原有的执行顺序变成了C -> A -> C -> A -> B，使得高优先级的任务A能够得到最及时的响应。即便类似B的任务有很多，A也因为B没有阻挡C而能够很快执行。这种方法只在占有资源的低优先级任务阻塞了高优先级任务时才动态的改变任务的优先级。

LiteOS采用优先级继承算法解决优先级翻转的问题。

## 4.2 互斥锁

### 4.2.1 MUX\_CB\_S结构解析



ucMuxStat：互斥锁控制块状态，标明该互斥锁是否已使用。

usMuxCount：互斥锁计数值，0表示释放状态，非0表示被获取状态，同一个TASK可多次获取同一个互斥锁，每获取一次，计数值加1。TASK每释放一次计数值减1，减到0表示互斥锁真正释放。

ucMuxID：互斥锁ID号，初始化时为每一个互斥锁控制块分配ID号，创建时返回给用户，用户通过ID号操作互斥锁。

stMuxList：链接指针，两个用途：一是用于在UNUSED状态下将MUX\_CB\_S结构链在未使用的链表中。二是在USED状态下用作互斥锁阻塞队列，链接因获取互斥锁失败需要进入阻塞状态的TASK，当互斥锁释放时从该队列依次唤醒被阻塞的TASK，每次释放只唤醒处于队头的那一个TASK。阻塞队列中有多个TASK时，由上一次被唤醒的TASK唤醒队头的TASK，直到阻塞队列为空时为止。

\*pstOwner：指示获得该互斥锁的TASK，指向TASK的TCB。

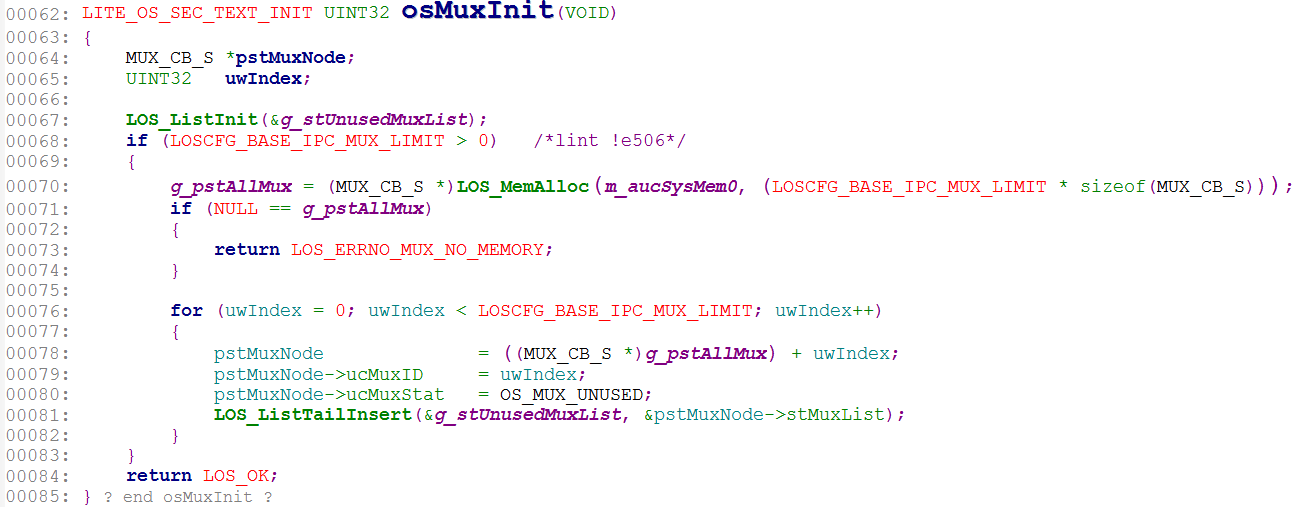
usPriority：保存\*pstOwner指向的TASK的原始优先级，该TASK的优先级在解决优先级翻转问题时有可能被更改，此处保存下来主要用于在释放互斥锁时恢复该TASK的原始优先级。

### 4.2.2 osMuxInit( )函数

mutex

g\_pstAllMux存储所有的互斥锁，所有互斥锁控制块顺序存储，g\_stUnusedMuxList指示目前未被分配的互斥锁链表，是双循环链表。

控制块被分配以后将从g\_stUnusedMuxList中删除，只存在于g\_pstAllMux数组中，不存在于任何链表中，当控制块被回收后重新进入g\_stUnusedMuxList中。



LOSCFG\_BASE\_IPC\_MUX\_LIMIT定义系统允许的最大互斥锁的个数，每个互斥锁由一个互斥锁控制块MUX\_CB\_S管理。

初始时申请所有互斥锁控制块所需的空间，并为每个MUX\_CB\_S设置ID号、状态为UNUSED，并依次将所有控制块加入到未使用互斥锁链表g\_stUnusedMuxList中（由成员stMuxList指示）。

### 4.2.3 LOS\_MuxCreate( )函数

用户创建一个互斥锁，参数\*puwMuxHandle用于接收创建成功的互斥锁的操作句柄，实际是互斥锁的ID号。



Line 112 - 113：从g\_stUnusedMuxList的头部取下一个MUX\_CB\_S，将其从未使用链表删除。

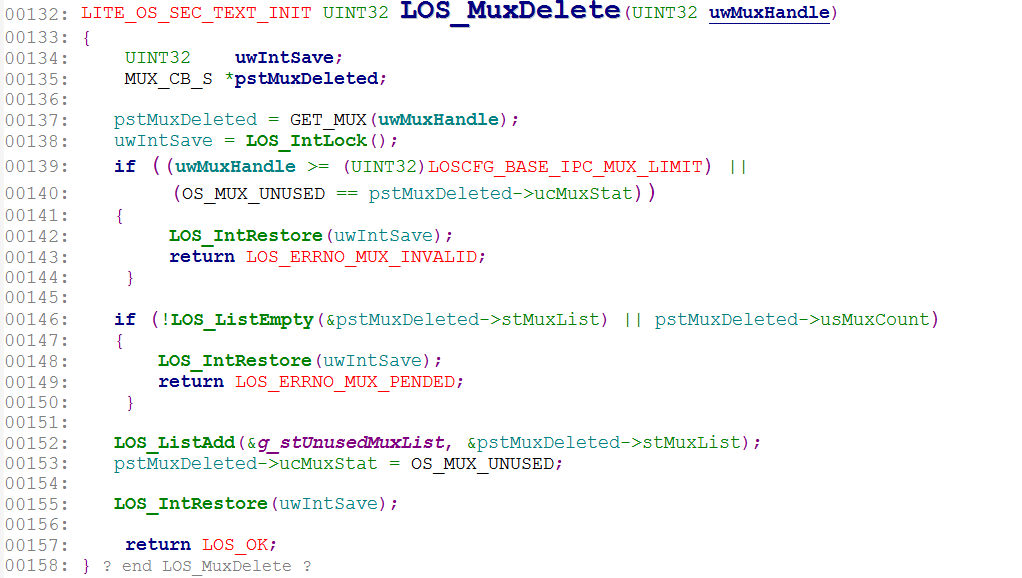
Line 114 - 118：初始化其成员变量。

Line 119：初始化stMuxList为空链表，后续该指针将用作指示互斥锁阻塞队列，链接因无法获取互斥锁而需要挂起的TASK。

Line 120：创建成功以后将互斥锁控制块的ID号作为操作句柄返回给用户。

### 4.2.4 LOS\_MuxDelete( )函数

用户删除一个互斥锁，参数uwMuxHandle是互斥锁ID号。



Line 137：GET\_MUX：由ID号查找对应的MUX\_CB\_S。

Line 146：stMuxList链表非空表示阻塞队列中还有等待该互斥锁的TASK存在，互斥锁不能删除。usMuxCount非0说明互斥锁还处于被获取状态，不允许删除。

Line 152：删除互斥锁时将该互斥锁对应的MUX\_CB\_S重新加入到g\_stUnusedMuxList的头部。

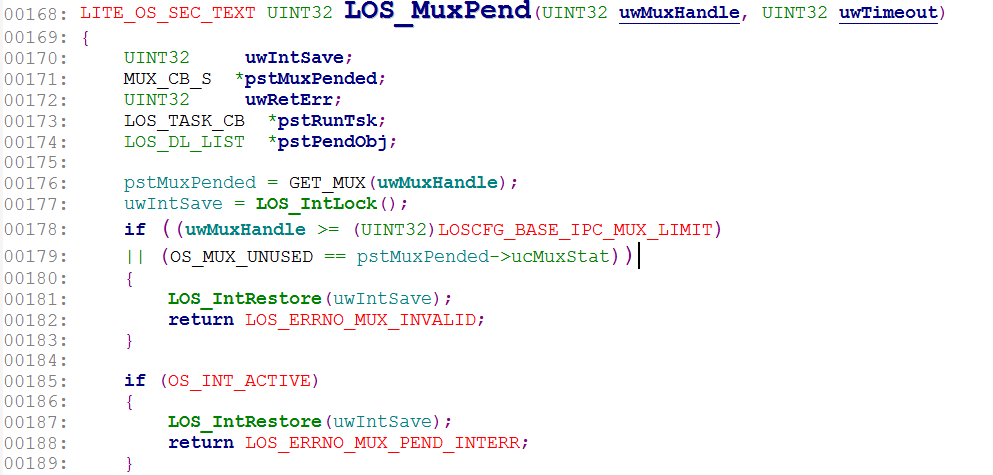
Line 153：置互斥锁控制块状态为UNUSED，其它变量在创建互斥锁时会被重新赋值。

### 4.2.5 LOS\_MuxPend( )函数

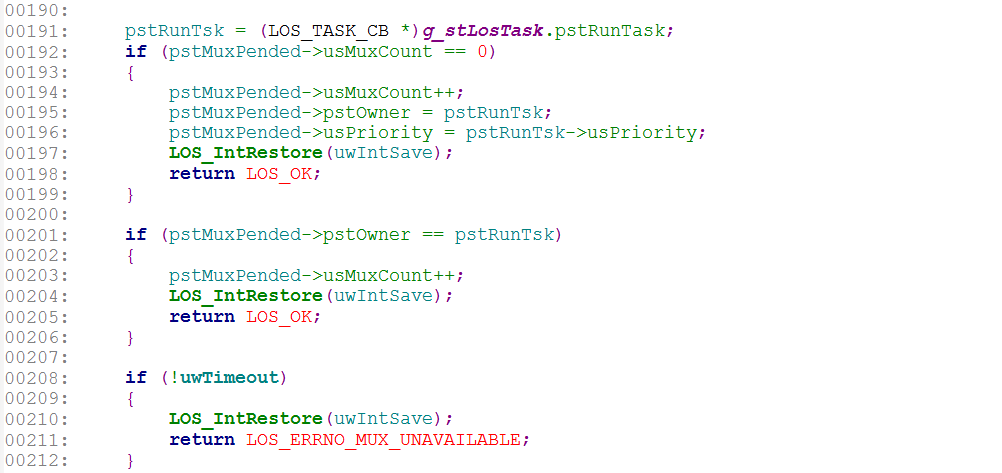
获取互斥锁，传入参数：

uwMuxHandle：互斥锁ID号。

uwTimeout：获取超时时间，同时也代表互斥锁的获取模式：0表示无阻塞模式，LOS\_WAIT\_FOREVER表示永久阻塞模式，其它值表示定时阻塞模式。



Line 185：中断中不能获取互斥锁，三种模式的互斥锁都不允许在中断中获取。（与信号量不同：无阻塞模式的信号量可以在中断中获取）



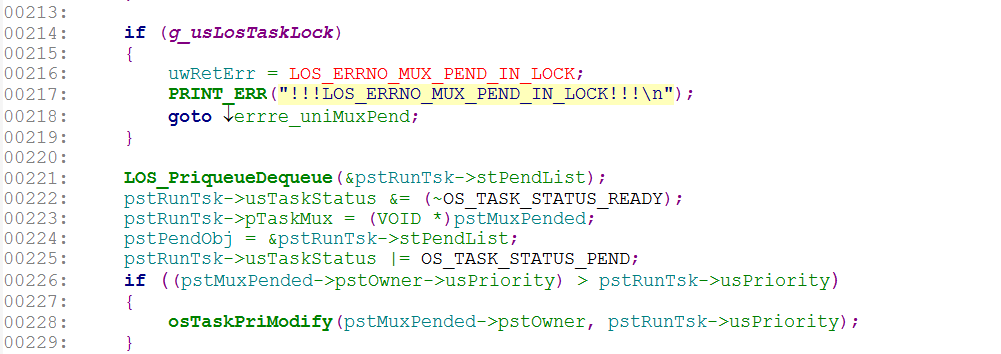
Line 192 - 196：usMuxCount == 0，说明该互斥锁处于释放状态，则直接获取该互斥锁，并设置相关变量，获取互斥锁成功。

Line 201 - 203：usMuxCount不为0，说明该互斥锁处于被获取的状态，若要获取互斥锁的TASK和互斥锁的拥有者一致，则表示获取成功，并使usMuxCount计数值增1，即同一个TASK可多次获取该互斥锁，每获取一次usMuxCount的值增1。

以上状态不满足表示互斥锁被其它TASK获取，当前TASK需要等待。

Line 208：若用户传入的获取模式是无阻塞模式，则直接返回未获取到互斥锁的错误状态。

否则，需要将该TASK挂起，以等待互斥锁释放唤醒或超时唤醒。



Line 214：系统调度被锁，则不允许挂起任务。

挂起因无法获取互斥锁而需要进入阻塞状态的任务：

Line 221 - 222：将当前任务从优先级队列中取下，并清除READY状态，不再参与系统调度。

Line 223：设置pstRunTsk->pTaskMux指向该TASK要获取的互斥锁的MUX\_CB\_S结构，表示该TASK正在等待获取该互斥锁。

Line 225：设置PEND（阻塞）状态。

Line 226 - 229：解决优先级翻转问题：如果互斥锁拥有者的优先级比当前正在运行的TASK的优先级低，则将互斥锁拥有者的优先级提升为与当前正在运行的TASK一致，这样可以通过优先级继承算法解决优先级翻转的问题，等到互斥锁拥有者释放互斥锁时再根据MUX\_CB\_S的成员usPriority来恢复其优先级。

不管互斥锁拥有者的优先级被提升多少次，被提升到多高，其最原始的优先级在MUX\_CB\_S的成员usPriority中保存，所以在互斥锁的拥有者释放互斥锁时，都能够恢复拥有者的优先级。



Line 231：将当前正在运行的TASK（由stPendList指示）加入到互斥锁的阻塞队列（由stMuxList指示）的尾部。

**问题：**阻塞队列采用FIFO模式，每次都唤醒处于队头的TASK，如果有多个不同优先级的TASK因获取同一个互斥锁而进入阻塞队列的话，如何保证阻塞队列中高优先级任务的实时性和剩余任务的优先级翻转的问题。

例如阻塞队列中有A、B、C三个TASK等待互斥锁，且排队顺序为A -> B -> C，优先级顺序A < B < C，互斥锁的拥有者为X，将A、B、C依次加入到阻塞队列后，X的优先级被提升到与C相同，当X释放互斥锁时唤醒A，X的优先级回到原始的优先级。此时互斥锁的拥有者变为A，互斥锁中保存的优先级为A原始的优先级，且A的优先级未被提升到与C相同。

这里有两个问题：一是更高优先级的C的实时性无法得到保证。二是运行A以后因A的优先级没有提升到与C相同，产生的优先级翻转问题如何解决。

有一个解决办法：进入阻塞队列时按照优先级顺序由高到低排队，而不是从队尾插入。这样做一是首先被唤醒的是C，高优先级任务的实时性得到保证。二是C的优先级在等待互斥锁的任务中目前处于最高，对B和A来讲不会产生优先级翻转的问题。

Line 232 - 244：设置阻塞超时模式：若为定时阻塞模式，则还需将该TASK加入到超时排序列表中等待超时唤醒，并设置TIMEOUT状态。永久阻塞模式只有等到互斥锁释放才能被唤醒。

Line 237 or 243：阻塞模式设置完成后，调用函数LOS\_Schedule( )启动一次任务切换。任务切换完成以后，CPU调度新的任务执行，当前的这个TASK的执行位置被停在了LOS\_Schedule( )函数的后面一条语句上（246行），（因为每一个TASK都有自己的栈空间，所以多个TASK调用同一个函数时会有多个该函数的现场，各个TASK对该函数的调用互不影响）。

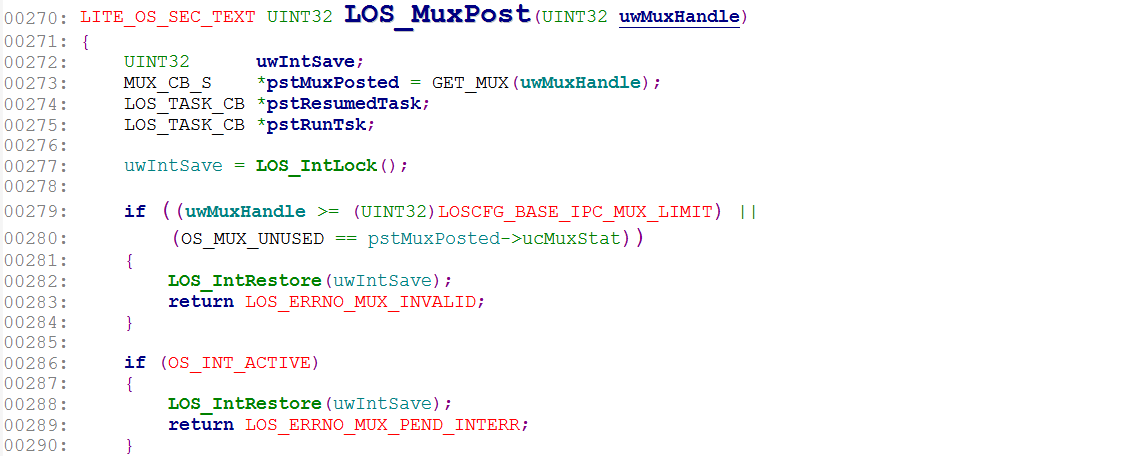
当这个TASK被唤醒后（通过互斥锁释放唤醒或者超时唤醒）继续向下执行。

Line 246 - 252：当TIMEOUT唤醒时会将该TASK从两个阻塞队列中删除，但是只清除了PEND状态，而没有清除TIMEOUT状态，具体参见osTaskScan( )函数。所以当被唤醒后若该TASK处于TIMEOUT状态，说明该TASK被超时唤醒，未获取到互斥锁，所以需要在此处清除TIMEOUT状态并返回错误，错误码为超时未获取到互斥锁。

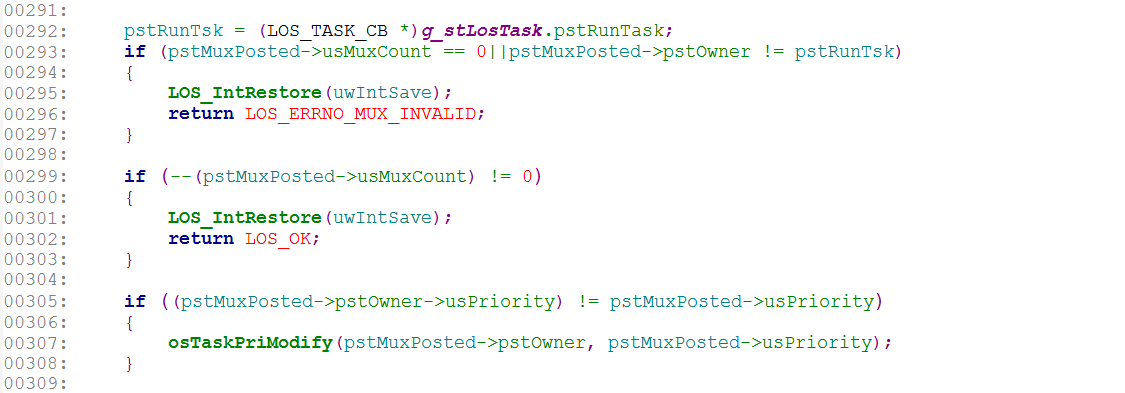
Line 255：互斥锁拥有者调用函数LOS\_MuxPost( )唤醒时会清除PEND状态和TIMEOUT状态并从两个阻塞队列中（超时排序队列和互斥锁阻塞队列）将该TASK删除，同时设置变量使互斥锁被这个TASK获取，所以这个TASK被唤醒以后已经获取到了想要的互斥锁，直接返回即可。

### 4.2.6 LOS\_MuxPost( )函数

释放互斥锁，参数uwMuxHandle是互斥锁的ID号。



中断处理程序中不允许释放互斥锁。

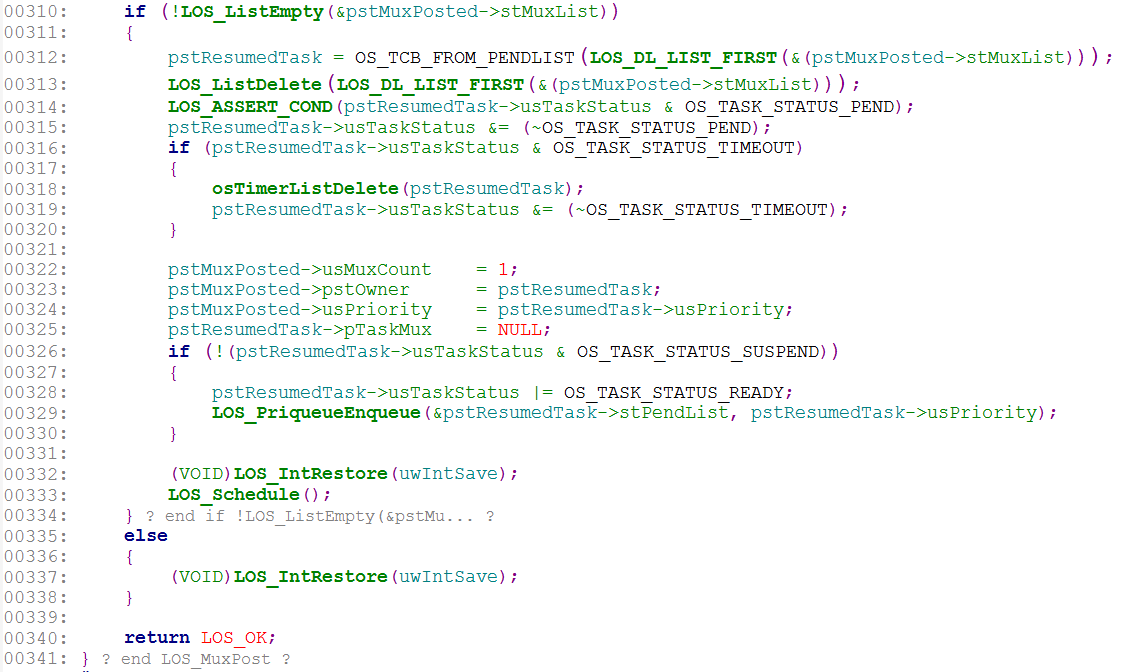


Line 293：usMuxCount == 0表示该互斥锁已经处于释放状态，不能再释放。若互斥锁的拥有者不是当前正在运行的TASK，则说明互斥锁被别的TASK获取，当前TASK无权释放。

Line 299：若usMuxCount减1后不为0，则说明该互斥锁被同一个TASK获取了多次，需要多次释放才能真正释放互斥锁，每一次释放都是释放成功状态。

若usMuxCount减1后等于0，则真正释放该互斥锁。

Line 305：若互斥锁拥有者的当前优先级与互斥锁控制块中保存下来的拥有者的原始优先级不一致，说明互斥锁拥有者因优先级翻转问题提升了其优先级，在释放互斥锁时需要恢复其原始的优先级。



Line 340：若互斥锁的阻塞队列为空，则说明没有因获取互斥锁而阻塞的TASK存在，且此时usMuxCount已经为0，表示互斥锁已经处于完全释放的状态，故直接返回。

Line 310：阻塞队列非空时唤醒处于阻塞队列中的TASK。

Line 312 - 315：将stMuxList指示的互斥锁阻塞队列中处于队头位置的TASK从阻塞队列删除，并清除PEND（阻塞）状态。

Line 316 - 320：若该TASK还有TIMEOUT状态，则说明用户设置了获取超时，该TASK同时被加入到了超时排序队列，需要将其从超时排序队列删除并清除TIMEOUT状态。

Line 322 - 325：将互斥锁通过设置互斥锁控制块MUX\_CB\_S的成员分配给要唤醒的这个TASK，pstResumedTask->pTaskMux = NULL表示该TASK已经等到了需要的互斥锁。

Line 326 - 330：若该TASK不处于SUSPEND状态，则设置READY状态并将其加入到优先级队列重新参与系统调度。否则要等到TASK被RESUME时再调度。

Line 333：启动一次任务切换，CPU调度优先级队列中最高优先级的任务执行，该函数返回后，释放互斥锁操作成功。

## 4.3 信号量

### 4.3.1 SEM\_CB\_S结构解析



usSemstat：信号量控制块的状态，OS\_SEM\_UNUSED或OS\_SEM\_USED。

uwSemCount：信号量对应的同一类型的互斥资源的总个数，初始时由用户传入。TASK每获取一个资源，uwSemCount减1，减到0意味着该信号量对应的所有的资源都被TASK获取了，系统目前无此类型的资源可用，同一个TASK可获取多个资源。

usSemID：信号量控制块的ID号，初始时分配，用户创建时返回给用户，用户通过ID号操作信号量。

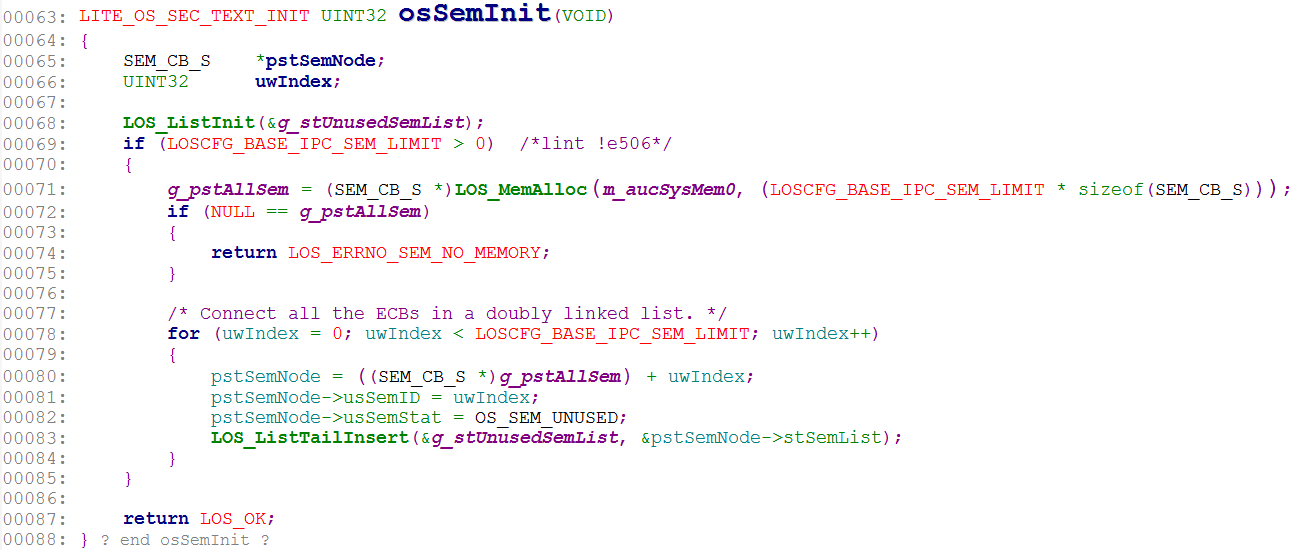
stSemList：链接指针，两个用途：一是用于在UNUSED状态下将SEM\_CB\_S结构链在未使用的链表中。二是在USED状态下用作信号量阻塞队列，链接因获取信号量失败需要进入阻塞状态的TASK，当信号量释放时从该队列依次唤醒被阻塞的TASK。每释放一个信号量，唤醒一个TASK。

### 4.3.2 osSemInit( )函数

sem

g\_pstAllSem存储所有的信号量控制块，所有信号量控制块顺序存储，g\_stUnusedSemList指示目前未被分配的信号量链表，是双循环链表。

控制块被分配以后将从g\_stUnusedSemList中删除，只存在于g\_pstAllSem数组中，不存在于任何链表中，当控制块被回收后重新进入g\_stUnusedSemList中。



LOSCFG\_BASE\_IPC\_SEM\_LIMIT定义系统允许的最大信号量的个数，每个信号量用一个信号量控制块SEM\_CB\_S管理。

初始时申请所有信号量控制块所需的空间，并为每个SEM\_CB\_S设置ID号、状态为UNUSED，并依次将所有控制块加入到未使用信号量链表g\_stUnusedSemList中（由成员stSemList指示）。

### 4.3.3 LOS\_SemCreate( )函数

用户创建一个信号量，传入参数：

usCount：指定信号量的初始计数值，对应于有效资源数，表示初始的可被占用的某一类型的互斥的资源个数。

\*puwSemHandle：用于接收创建成功的信号量的操作句柄，实际是信号量的ID号。



Line 108：OS\_SEM\_COUNT\_MAX被定义为0xFFFE，所以usCount的值不能是0xFFFF，0xFFFF在系统中表示无穷大。

创建信号量：

Line 121 - 122：从g\_stUnusedSemList的头部取下一个SEM\_CB\_S，将其从未使用链表删除。

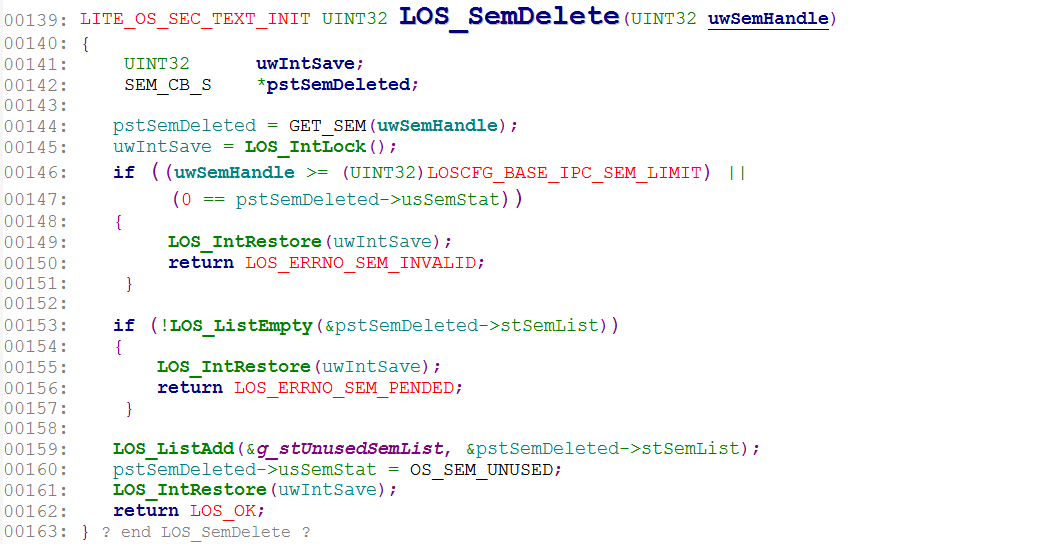
Line 123 - 125：初始化其成员变量。

Line 126：初始化stSemList，后续该指针将用作指示信号量阻塞队列，链接因无法获取信号量而需要挂起的TASK。

Line 127：创建成功以后将信号量控制块的ID号作为操作句柄返回给用户。

### 4.3.4 LOS\_SemDelete( )函数

用户删除一个信号量，参数uwMuxHandle是信号量ID号。



Line 144：GET\_SEM：由ID号查找对应的SEM\_CB\_S。

Line 153：stSemList链表非空表示阻塞队列中还有等待信号量的TASK存在，信号量删除以后这些TASK将无法被唤醒，故不能删除。

Line 159 - 160：删除信号量时将该信号量对应的SEM\_CB\_S重新加入到g\_stUnusedSemList的头部并置状态为UNUSED，其它变量在创建信号量时会被重新赋值。

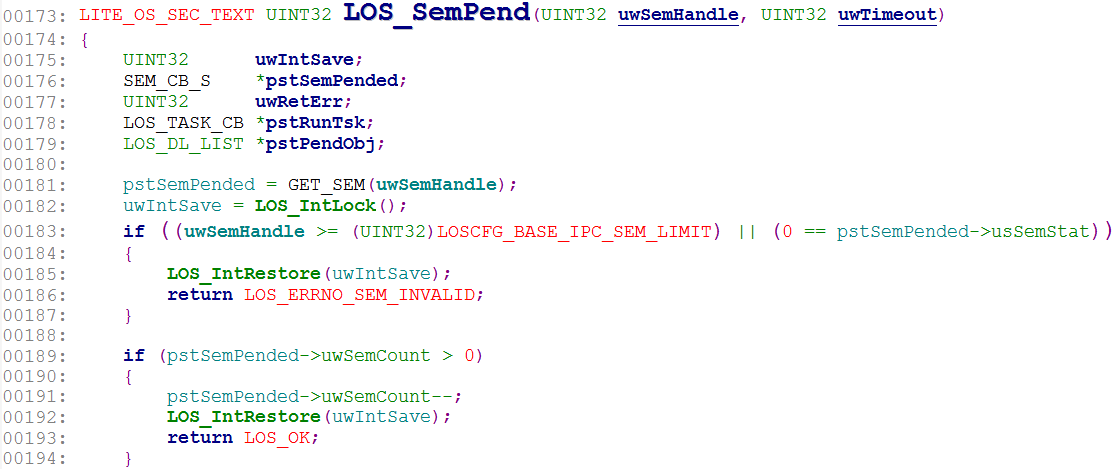
备注：只要阻塞队列为空，就可以删除信号量了，不需要考虑信号量被其它TASK获取的情况，删除后其它TASK在释放信号量时会因为UNUSED状态而直接返回错误。

### 4.3.5 LOS\_SemPend( )函数

获取信号量，传入参数：

uwSemHandle：信号量ID号。

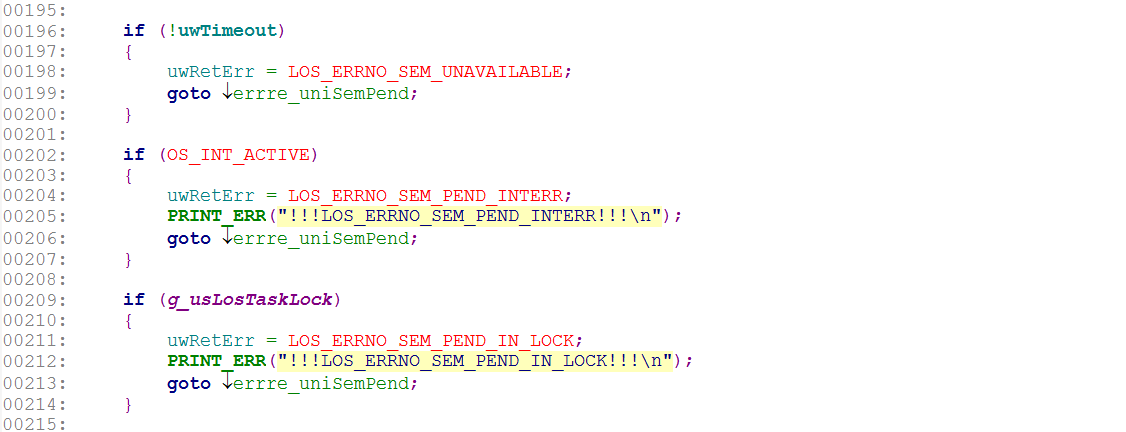
uwTimeout：获取超时时间，同时也代表信号量的获取模式：0表示无阻塞模式，LOS\_WAIT\_FOREVER表示永久阻塞模式，其它值表示定时阻塞模式。



Line 183：usSemStat为0代表UNUSED状态。

Line 189 - 194：若uwSemCount > 0，表示系统中同类型的互斥的资源还有空闲的，则直接减1返回，表示获取成功。

否则，需要阻塞获取。



Line 196：用户设置无阻塞模式获取，无法获取时直接返回没有可用信号量的错误状态，否则需要挂起TASK等待其它任务释放信号量。

Line 202：中断中不允许挂起TASK，但是中断中可以使用无阻塞模式获取信号量（与互斥锁不同：中断中不允许以任何方式获取互斥锁）。

Line 209：系统调度被锁时不允许挂起TASK，因为挂起TASK需要切换任务，而系统调度被锁时系统不能正常切换任务。



挂起因无法获取信号量而需要进入阻塞状态的TASK：

Line 216：挂起正在运行的TASK。

Line 217 - 218：将该TASK从优先级队列中取下，并清除READY状态，不再参与系统调度。

Line 219：设置pstRunTsk->pTaskSem指向该TASK要获取的信号量的SEM\_CB\_S结构，表示该TASK正在等待获取该信号量。

Line 221：设置PEND（阻塞）状态。

Line 222：将当前正在运行的TASK（由stPendList指示）加入到信号量的阻塞队列（由stSemList指示）的尾部。

**问题：**阻塞队列采用FIFO模式，每次都唤醒处于队头的TASK，如果有多个不同优先级的TASK因获取同一种类型的信号量而进入阻塞队列的话，如何保证阻塞队列中高优先级任务的实时性。

有一个解决办法：进入阻塞队列时按照优先级顺序由高到低排队，而不是从队尾插入。这样每次唤醒时都唤醒阻塞队列中优先级最高的TASK。

Line 223 - 231：设置阻塞超时模式。若为定时阻塞模式，则还需将该TASK加入到超时排序列表中等待超时唤醒，并设置TIMEOUT状态。永久阻塞模式只有等到有信号量释放才能被唤醒。

Line 234：调用函数LOS\_Schedule( )启动一次任务切换。任务切换完成以后，CPU调度新的任务执行，当前的这个TASK的执行位置被停在了LOS\_Schedule( )函数的后面一条语句上（236行），（因为每一个TASK都有自己的栈空间，所以多个TASK调用同一个函数时会有多个该函数的现场，各个TASK对该函数的调用互不影响）。

当这个TASK被唤醒后（通过信号量释放唤醒或者超时唤醒）继续向下执行。

Line 236 - 243：当TIMEOUT唤醒时会将该TASK从两个阻塞队列中删除，但是只清除了PEND状态，而没有清除TIMEOUT状态，具体参见osTaskScan( )函数。所以当被唤醒后若该TASK处于TIMEOUT状态，说明该TASK被超时唤醒，未获取到信号量，所以需要在此处清除TIMEOUT状态并返回错误，错误码为超时未获取到信号量。

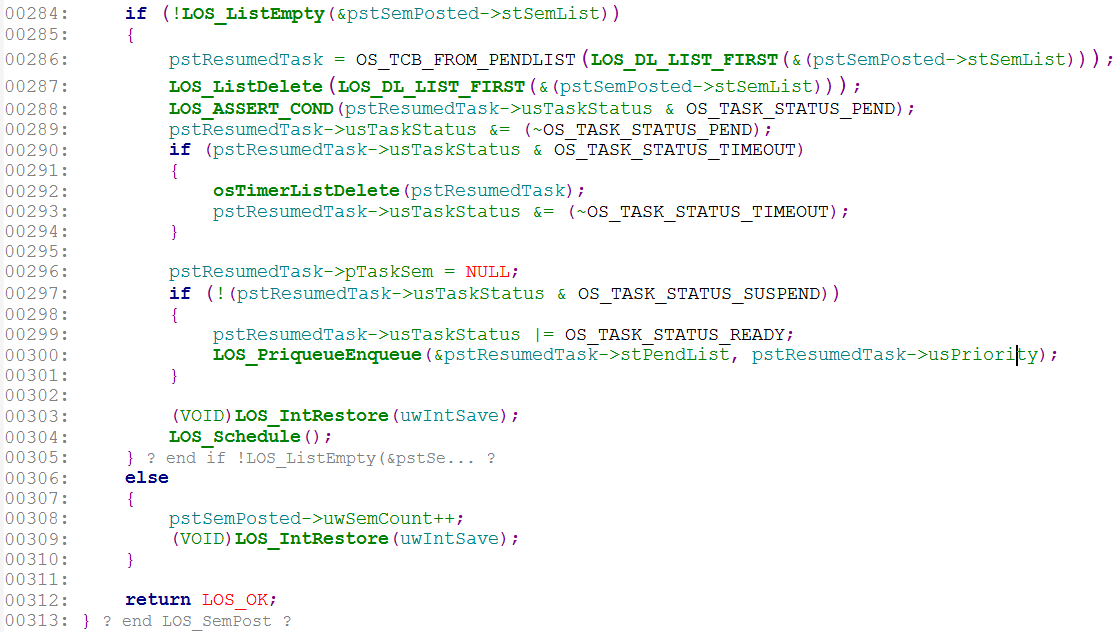
Line 245：其它获取到信号量的TASK调用函数LOS\_SemPost( )释放信号量唤醒时会清除PEND状态和TIMEOUT状态并从两个阻塞队列中（超时排序队列和信号量阻塞队列）将该TASK删除，直接唤醒该TASK表示该TASK获取信号量成功，LOS\_SemPost( )中不执行uwSemCount加1的操作，所以此处不需要执行uwSemCount减1的操作，直接返回表示获取信号量成功。

### 4.3.6 LOS\_SemPost( )函数

释放信号量，参数uwSemHandle是信号量的ID号。



Line 279：释放信号量时需要将uwSemCount加1，最大值是OS\_SEM\_COUNT\_MAX，加1变为0xFFFF，表示无穷大。



Line 308：若信号量阻塞队列为空，则说明没有因获取信号量而阻塞的TASK存在，则直接将可用信号量计数值加1后直接返回，有TASK需要获取信号量时直接减1获取。否则需要唤醒处于阻塞队列中的TASK。

Line 284：信号量阻塞队列非空，唤醒处于阻塞状态的TASK。

Line 287：将stSemList指示的信号量阻塞队列中处于队头位置的TASK从阻塞队列删除。

Line 289：清除PEND（阻塞）状态。

Line 290 - 294：若该TASK还有TIMEOUT状态，则说明用户设置了获取超时，该TASK同时被加入到了超时排序队列，需要将其从超时排序队列删除并清除TIMEOUT状态。

Line 296：pstResumedTask->pTaskSem = NULL表示该TASK等待的信号量已经获取到了。

Line 297 - 301：若该TASK不处于SUSPEND状态，则设置READY状态并将其加入到优先级队列重新参与系统调度。否则需要等到TASK被RESUME时才能参与调度。

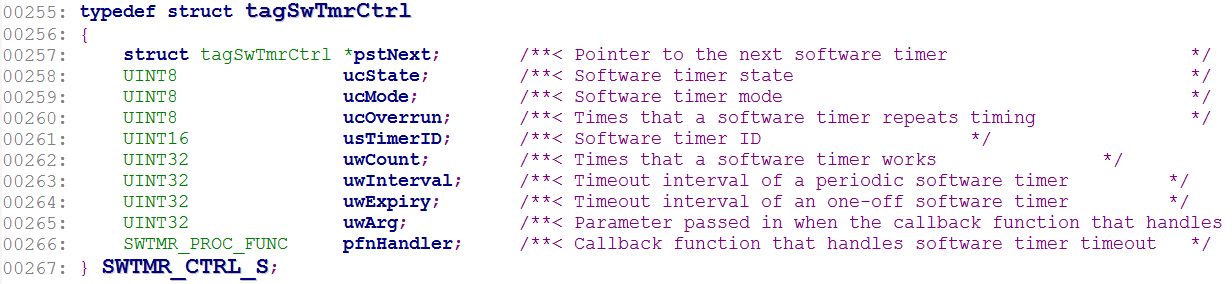
备注：唤醒TASK时不再执行uwSemCount加1的操作，直接唤醒队头的TASK，唤醒之后该TASK也不执行uwSemCount减1的操作。只要该TASK被唤醒，则表示已经获取到了信号量。

Line 304：启动一次任务切换，CPU调度优先级队列中最高优先级的任务执行，该函数返回后，释放信号量操作成功。

# 软件定时器

## 5.1 软件定时器任务函数

### 5.1.1 SWTMR\_CTRL\_S结构解析



\*pstNext：链接下一个SWTMR\_CTRL\_S结构的指针，单链表形式链接，主要用于将当前控制块链接到空闲链表或排序链表中。

ucState：swtmr状态，三种状态分别是：

OS\_SWTMR\_STATUS\_UNUSED：未使用状态，该控制块处于空闲链表中，初始化时或定时器被删除后均处于该状态。

OS\_SWTMR\_STATUS\_CREATED：创建未启动/停止状态，创建成功，已经从空闲链表中取出，但并未加入到排序链表中启动，或者定时器停止，从排序链表取下后均处于该状态。

OS\_SWTMR\_STATUS\_TICKING：计数状态，表示定时器被加入到排序链表，正在运行。

ucMode：swtmr的触发模式，三种触发模式分别是：

LOS\_SWTMR\_MODE\_ONCE：单次触发模式，启动后只触发一次定时器事件，执行一次回调函数，然后定时器自动删除，重新放回到空闲链表中。

LOS\_SWTMR\_MODE\_PERIOD：周期触发模式，周期性的触发定时器事件，直到用户手动停止定时器为止，否则将永远执行下去。

LOS\_SWTMR\_MODE\_OPP：表示在一次性计时器完成计时之后，启用周期性软件定时器，此模式目前不支持，为将来预留。

ucOverrun：周期触发模式下，该定时器被循环触发的次数。

usTimerID：swtmr的ID号，初始化时分配，创建swtmr成功后返回给用户，用户通过ID号操作对应的swtmr。

uwCount：定时器计数，以tick为单位，在启动定时器（加入到排序链表中）时会设置该计数值，在每个tick中断中减1，减到0意味着定时时间到。

uwInterval：周期性定时器的定时间隔。

uwExpiry：一次性定时器的定时时长。

uwArg：定时器回调函数的参数。

pfnHandler：定时器的回调函数，定时器时间到时在swtmr TASK中执行该函数。

### 5.1.2 osSwTmrInit( )函数

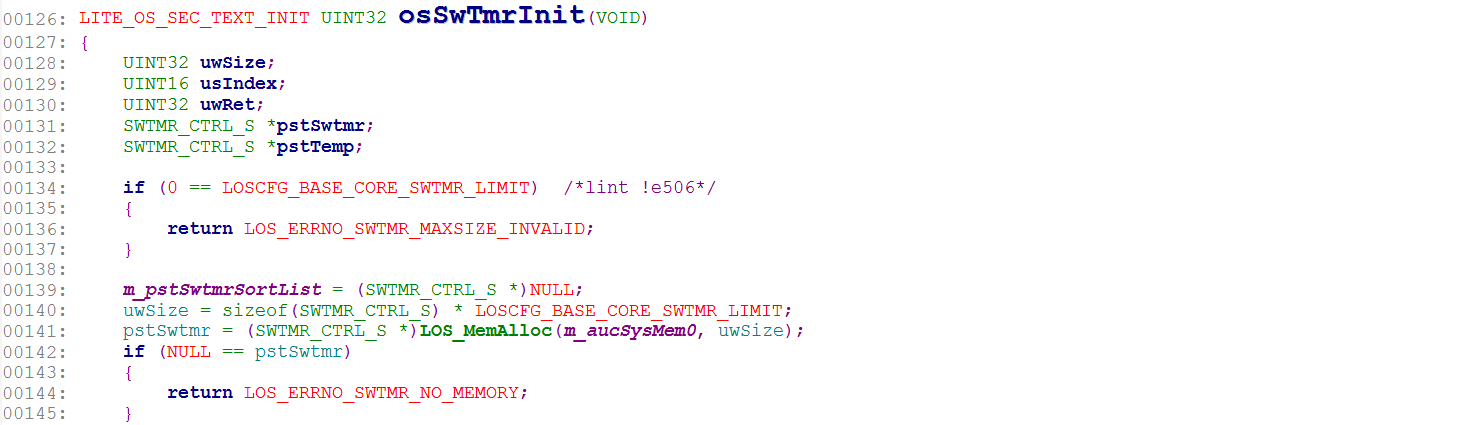
m\_pstSwtmrCBArray：指示swtmr控制块的存储数组，所有的控制块不管处于哪个链表，都同时存在于该数组中。

m\_pstSwtmrFreeList：指示空闲链表，处于该链表中的swtmr控制块都处于UNUSED状态。

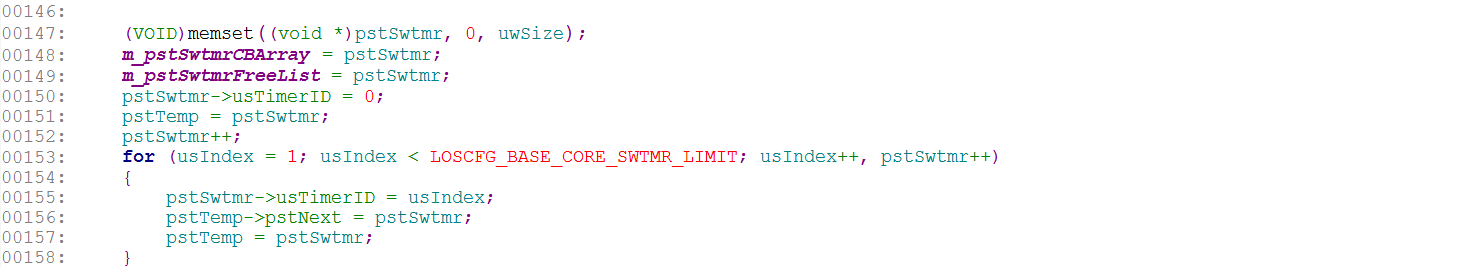
创建swtmr成功后，控制块离开m\_pstSwtmrFreeList，但不处于任何链表，只独立存在于m\_pstSwtmrCBArray中。

m\_pstSwtmrSortList：指示排序链表，处于该链表中的swtmr控制块都处于TICKING状态，表示正在运行。排序链表中的控制块按照定时时间由短到长的顺序加入。

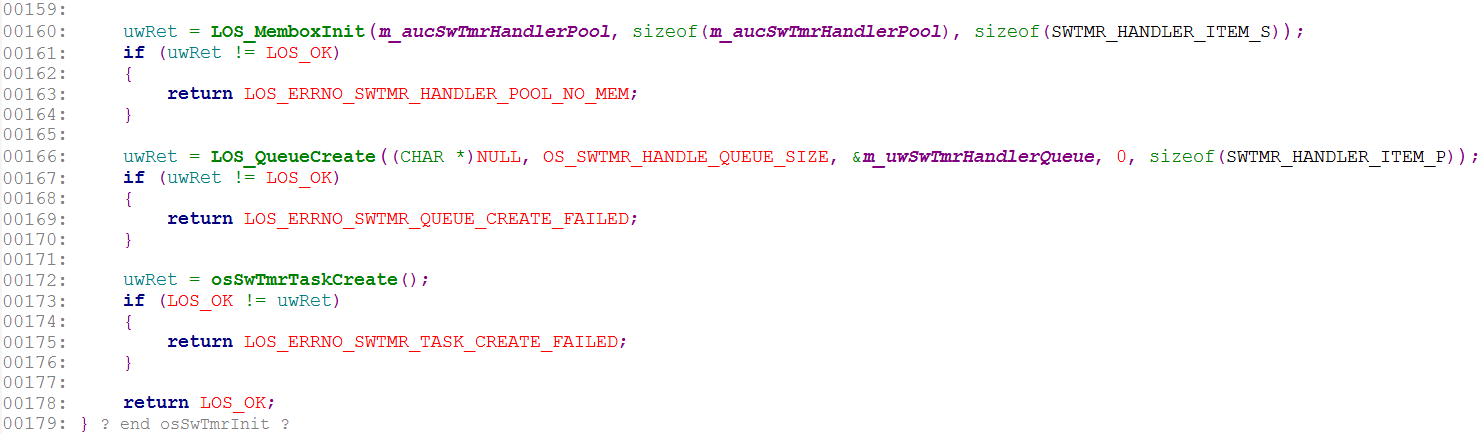
当启动一个处于CREATED状态的swtmr时，独立存在于m\_pstSwtmrCBArray中的控制块被加入到排序链表进行定时。当停止一个处于TICKING状态的swtmr时，控制块从排序链表取下，仍只存在于m\_pstSwtmrCBArray中。当删除一个处于CREATED状态的swtmr时，控制块被重新加入到空闲链表中。



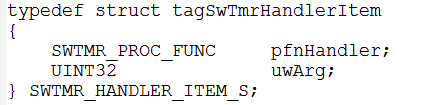
LOSCFG\_BASE\_CORE\_SWTMR\_LIMIT定义系统允许的软件定时器的最大个数。每一个软件定时器由唯一的控制块控制。初始时申请数量为LOSCFG\_BASE\_CORE\_SWTMR\_LIMIT的软件定时器控制块。



初始时为所有的swtmr控制块分配ID号并将它们依次链接在空闲链表中。



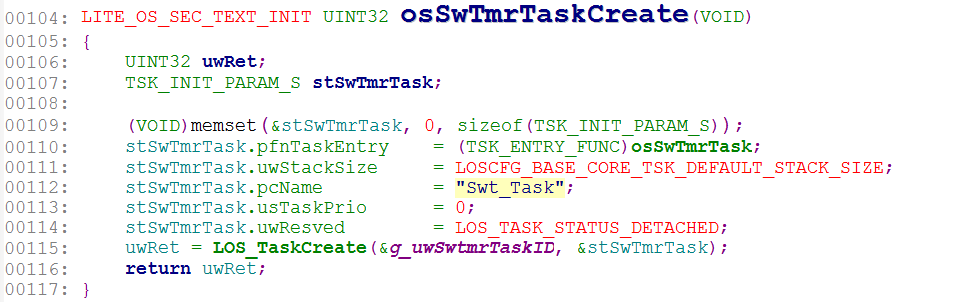
Line 160：将静态数组m\_aucSwTmrHandlerPool初始化为静态内存池，每个内存块的大小为SWTMR\_HANDLER\_ITEM\_S结构体的大小，主要用于存储swtmr的回调函数和回调函数的参数，结构体如下。



Line 166：创建消息队列，用于在swtmr定时时间到时向swtmr TASK发送消息。节点个数为软件定时器的最大个数，以保证在最极端情况下消息队列能容纳所有的软件定时器消息。每个节点的大小为一个指向m\_aucSwTmrHandlerPool标识的静态内存池中内存块的指针，发送消息时将存有回调函数和参数的内存块的地址传给swtmr TASK。

Line 172：创建swtmr TASK，swtmr以一个独立的任务的方式运行。

### 5.1.3 osSwTmrTaskCreate( )函数

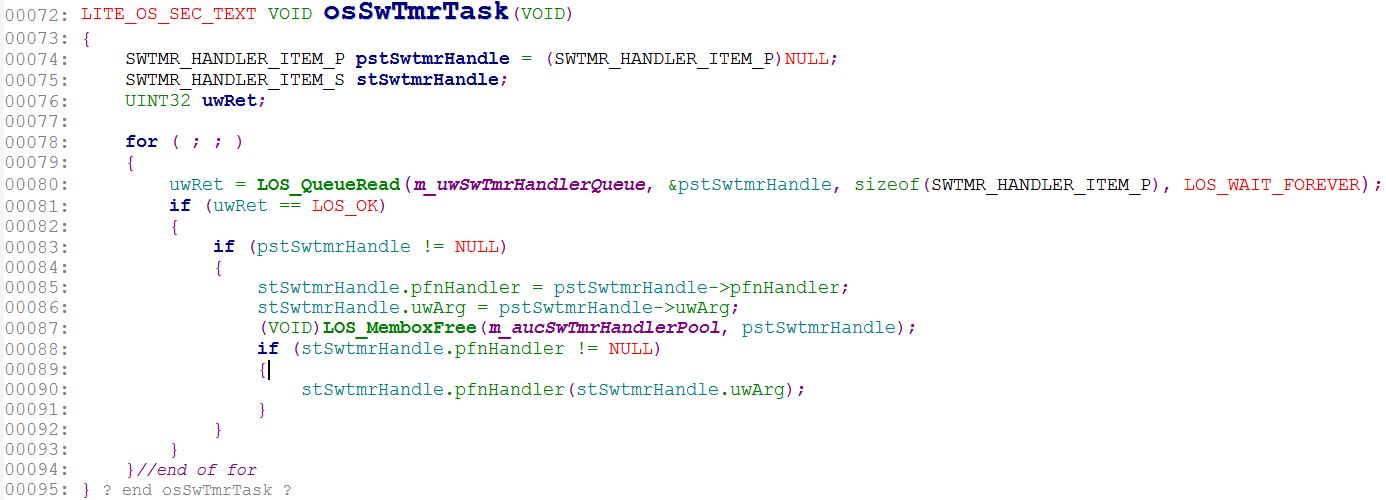


创建swtmr任务，优先级为最高且不允许更改，也不允许挂起swtmr任务。TASK状态为LOS\_TASK\_STATUS\_DETACHED，表示分离属性，即当该TASK的工作函数osSwTmrTask( )退出后需要自行删除该TASK。

osSwtmrTask( )是死循环函数，除非系统断电，否则不会退出，所以swtmr TASK不会被删除。

### 5.1.4 osSwTmrTask( )函数

Swtmr任务函数，优先级最高，所以被唤醒后立即执行，以保证定时器的定时准确性。平常该TASK会被阻塞在LOS\_QueueRead( )中，被挂起在阻塞队列中，当有定时器时间到时会发送消息，收到消息后被唤醒继续执行。



Line 80：读取消息时取得消息节点中的内容，\*pstSwtmrHandle中存放的是内存池中内存块的起始地址，该内存块中存放有swtmr的回调函数和参数。

Line 85 - 86：将内存块中存放的回调函数和参数取出。

Line 87：已经得到了需要的信息，故将内存块重新还回到内存池中去。

Line 90：执行软件定时器的回调函数。

**注意：**软件定时器的回调函数是在其它TASK中指定的，但是回调函数的执行却是在swtmr TASK中，回调函数将使用swtmr TASK的栈空间。

## 5.2 软件定时器基本函数

### 5.2.1 LOS\_SwtmrCreate( )函数

用户创建一个swtmr，创建好以后，swtmr对应的控制块将从空闲链表取下，不存在于任何链表中，只存在于m\_pstSwtmrCBArray中，swtmr处于CREATED状态。

传入参数：

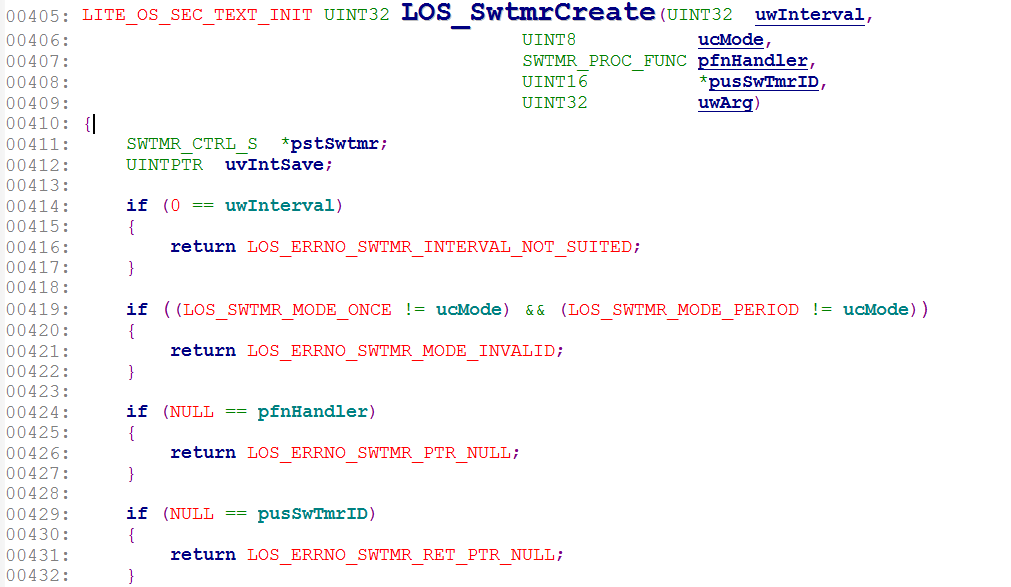
uwInterval：定时器时长，单位是tick，必须大于0。

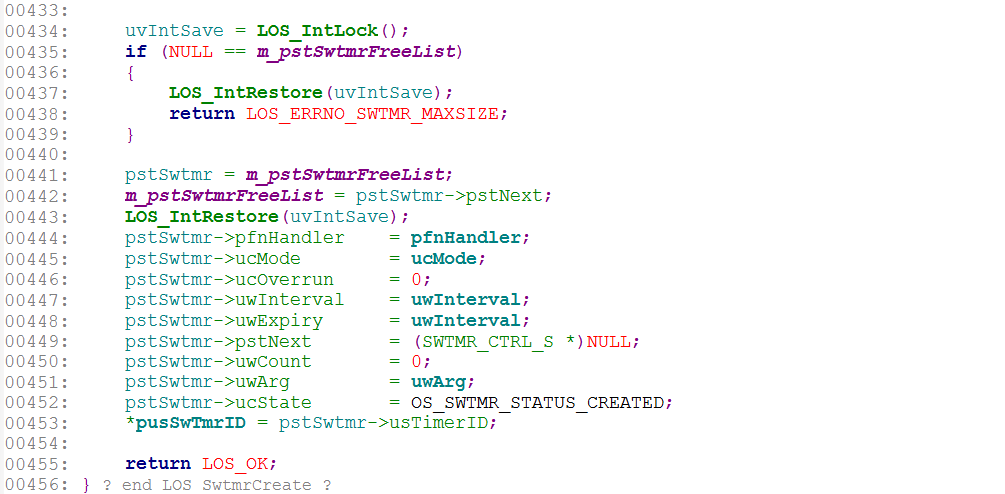
ucMode：定时器模式，目前只支持ONCE模式和PERIOD模式，OPP模式不支持。

pfnHandler：定时器回调函数，不能为NULL，当定时时间完成后执行该函数，这个函数的执行在软件定时器任务中，不在创建和使用定时器的任务中。

\*pusSwTmrID：用于接收创建完成的定时器的ID号，不能为NULL，后续用户通过该ID号操作该定时器。

uwArg：定时器回调函数的参数。





Line 435：定时器空闲链表为空，表示所有的定时器已经分配完了，系统再无可用的swtmr分配。

Line 441 - 442：分配定时器时，从空闲链表的头部取下一个定时器控制块分配给用户，并设置空闲链表的头指向当前位置的下一个位置。

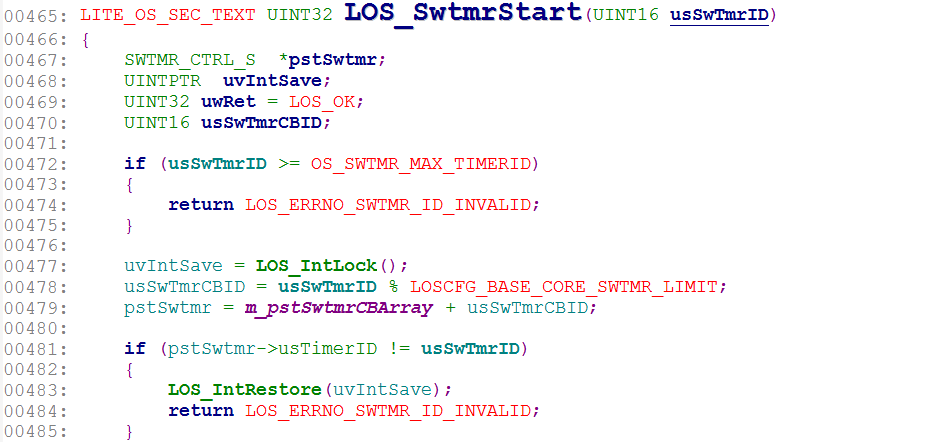
Line 444 - 452：填充定时器控制块，uwCount在启动定时器时填入。设置ucState为CREATED状态，表示创建未启动，uwInterval和uwExpiry被填入了相同的值。

Line 453：创建成功以后将定时器控制块的ID号返回给用户。

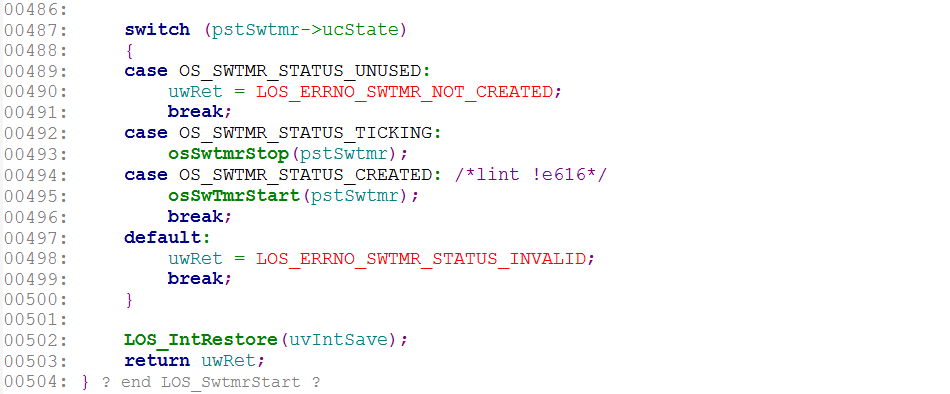
### 5.2.2 LOS\_SwtmrStart( )函数

启动一个处于CREATED状态的定时器，传入定时器ID。

启动成功以后该swtmr被加入到排序链表中做定时扫描，处于TICKING状态。



Line 478 - 481：由用户传入的swtmr的ID号找到对应的swtmr控制块，并判断控制块中的ID号和用户传入的ID号是否相等。



Line 494：处于CREATED状态的定时器直接启动（加入到排序链表中）。

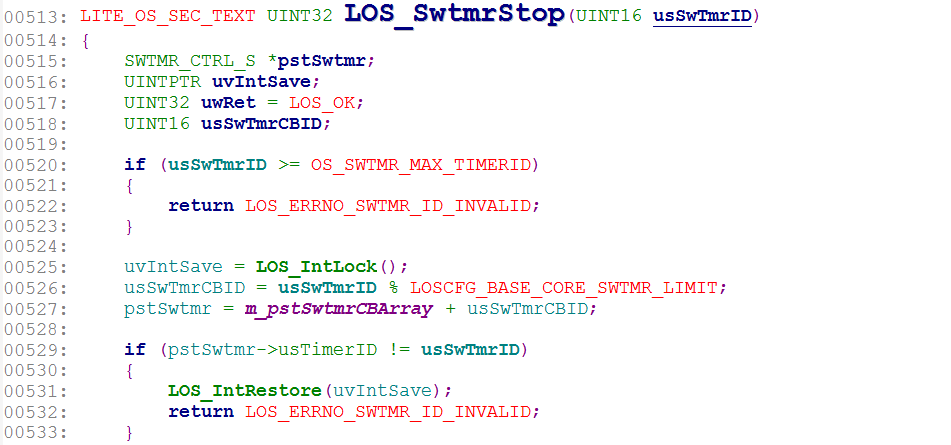
Line 492：处于TICKING状态的定时器已经在运行，需要先停止（从排序链表取下），然后再重新启动（再重新加入到排序链表重新开始定时）。

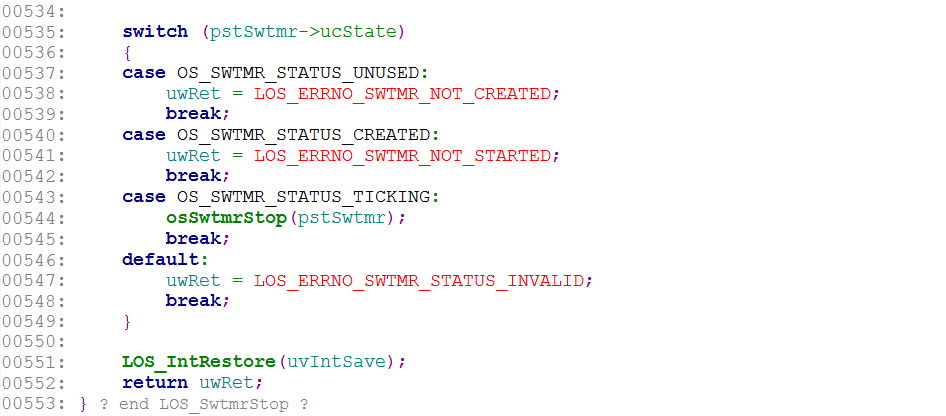
备注：case OS\_SWTMR\_STATUS\_TICKING: 后无break。

### 5.2.3 LOS\_SwtmrStop( )函数

停止一个处于TICKING状态的定时器，传入定时器ID。

停止成功以后swtmr从排序链表中取下，不再参与定时扫描，不处于任何链表中，只存在于m\_pstSwtmrCBArray中，状态变为CREATED（未启动）状态。

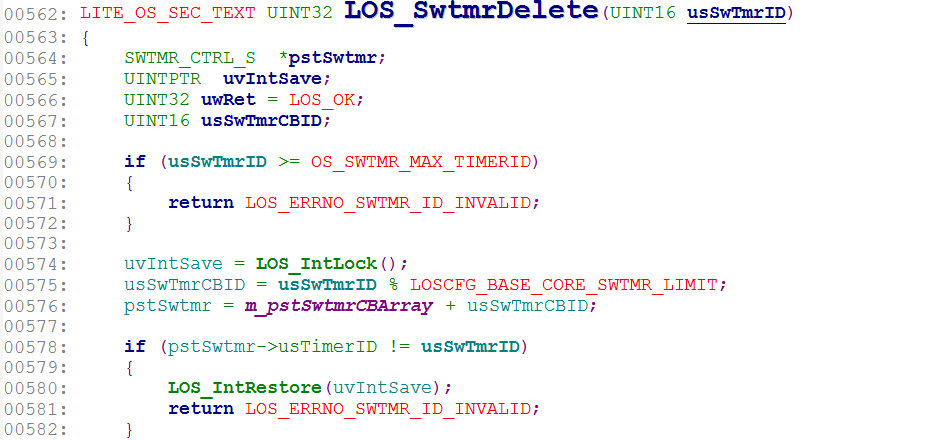


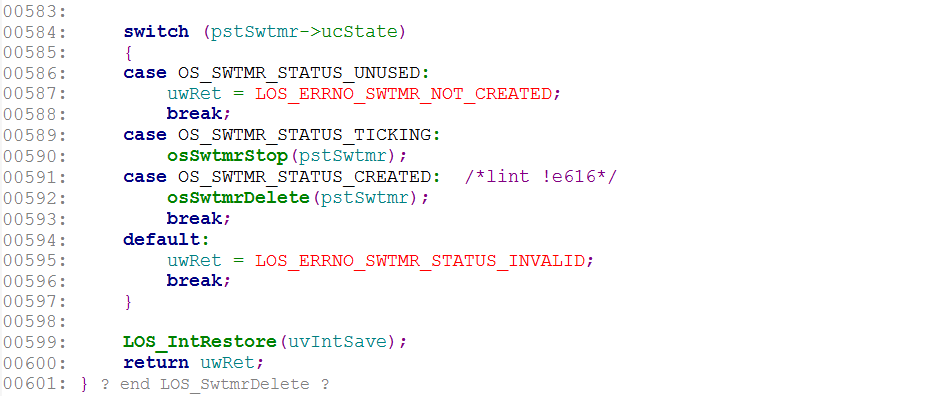


### 5.2.4 LOS\_SwtmrDelete( )函数

删除一个处于TICKING或CREATED状态的定时器，传入定时器ID。

删除成功后swtmr被重新加入到空闲链表中，定时器控制块变为UNUSED状态。





Line 591：因处于CREATED状态的定时器不在排序链表中，所以直接删除，将其加入到空闲链表中。

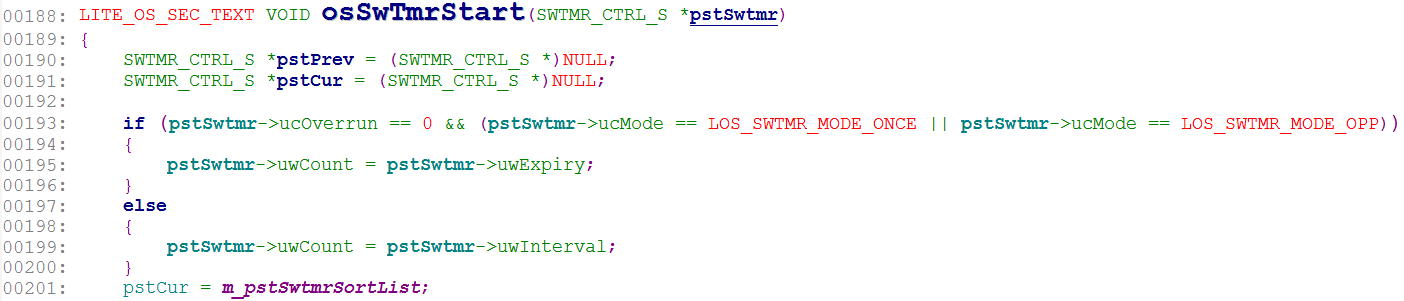
Line 589：处于TICKING状态的定时器在排序链表中正在定时，故需要先停止定时，将其从排序链表取下，然后再删除。

备注：case OS\_SWTMR\_STATUS\_TICKING: 后无break。

### 5.2.5 osSwTmrStart( )函数

启动一个处于OS\_SWTMR\_STATUS\_CREATED状态的定时器。传入该定时器对应的控制块。

启动定时器时将该定时器控制块按照时间由短到长的顺序加入到m\_pstSwtmrSortList指示的排序队列中，等待tick中断扫描。



Line 193：设置uwCount值，在创建的时候uwExpiry和uwInterval被赋了相同的值，故目前的判断没有用处。



Line 203 - 229：循环扫描排序链表，并将swtmr控制块按照时间计数值uwCount由小到大的顺序加入到排序链表的相应位置，且加入时的uwCount值是uwCount总值与前面所有控制块的uwCount总和的差值。

Line 231：启动成功后设置软件定时器的状态为TICKING。

例值1：现有A、B、C、D、E共5个swtmr，软件定时tick如下

Swtmr A B C D E

Tick 19 23 13 19 10

现将它们按照顺序加入到排序队列中，加入到排序队列后顺序和对应的uwCount如下：

m\_pstSwtmrSortList-> E C A D B

uwCount 10 3 6 0 4

排序队列的指示指针m\_pstSwtmrSortList指向队列头部的E，Tick中断扫描时先扫描E，当E定时时间结束时过了10个tick。再过3个tick后C也结束，共13个tick。再过6个tick后A结束，共19个tick。A与D时间相同，所以在同一个tick中断中A结束以后D也随之结束。最后再过4个tick后B结束，共23个tick。（每个tick中断扫描时只扫描队头的那个swtmr，若uwCount > 0，则减1后退出）

### 5.2.6 osSwtmrStop( )函数

停止一个处于OS\_SWTMR\_STATUS\_TICKING状态的定时器。传入该定时器对应的控制块。



Line 368 - 392：停止定时器时将定时器控制块从m\_pstSwtmrSortList指示的排序链表中删除，并更改后面一个定时器的定时计数（如果存在的话，Line 378），但不将其加入空闲队列（只是孤立的存在于m\_pstSwtmrCBArray数组中），并设置状态为CREATED。

以3.2.5中的排序链表为例，假设要删除C，则删除后的排序队列及对应的uwCount如下：

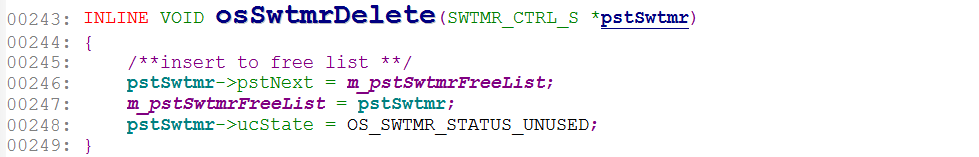
m\_pstSwtmrSortList-> E A D B

uwCount 10 9 0 4

删除C时将C的uwCount值加到C后面的A上，以保证删除C以后，后面的A、D、B的定时都能准确。

### 5.2.7 osSwtmrDelete( )函数

删除一个处于OS\_SWTMR\_STATUS\_CREATED状态下的swtmr，传入定时器控制块，其它状态下的swtmr不能调用该函数删除。

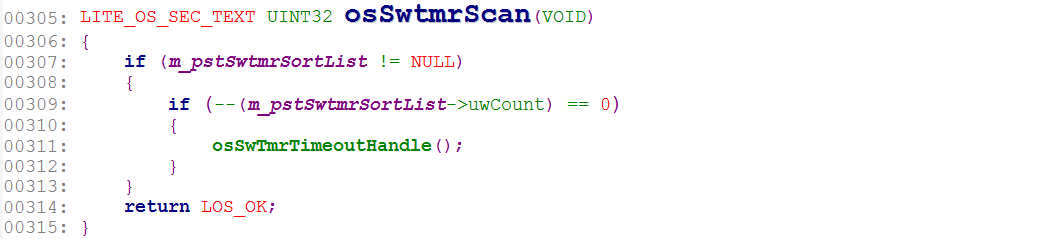


处于CREATED状态的swtmr，不存在于任何链表中，只是孤立的存在于m\_pstSwtmrCBArray中，所以只需要将其加入空闲链表（从头部插入）并设置该控制块的状态为UNUSED即可，其它变量不需要清除，在创建swtmr时会重新赋值。

## 5.3 软件定时器扫描及处理

### 5.3.1 osSwtmrScan( )函数

在tick中断处理函数osTickHandler( )中调用函数osSwtmrScan( )扫描软件定时器。



排序链表中的swtmr是按照定时时间由短到长的顺序排列的，故每次只需要扫描链表头位置的uwCount即可，当头位置的uwCount为0时表示定时器时间到，进行软件定时器超时处理。

### 5.3.2 osSwTmrTimeoutHandle( )函数



Line 263：在一个tick中断中，循环扫描排序队列中所有定时时间到的定时器，当扫描到一个满足条件的swtmr时，做以下操作：

Line 265：将该swtmr从软件定时器排序列表中删除。

Line 266：从m\_aucSwTmrHandlerPool所指示的静态内存池中申请一个内存块，用于填充相关信息。

Line 270 - 271：将swtmr的回调函数和回调函数的参数填入申请好的内存块的对应位置。

Line 273：向swtmr TASK发送消息，表示有定时器的定时时间到了，需要swtmr TASK执行定时器对应的回调函数。消息节点的内容是内存块的起始地址，该内存块中存储了要执行的回调函数和参数。当swtmr TASK收到这个消息后，从消息节点中取出内存块的首地址，再从该地址中取出回调函数和参数，然后再执行回调函数。

注意：中断中只允许以非阻塞模式发送消息。

Line 275：当消息发送失败时，需要将刚才申请的内存块还回内存池中。另外，如果消息发送失败，则会导致当前这一次的软件定时器失效。

Line 281：如果swtmr是单次触发模式，则定时时间到以后将该定时器自动删除，定时器控制块重新加入到空闲链表中。

Line 283 - 286：更改swtmr的ID号。因为是自动删除，用户并不知道这个删除过程，如果不更改ID号，那么当用户后续申请时会申请到两个同样ID号的swtmr，会对用户的判断造成影响。这种更改ID的方式仍然存在冲突的可能，但概率极小，可忽略不计。

Line 290 - 291：如果swtmr是循环触发模式，则将循环次数增1后重新启动定时器，将其再次加入到swtmr排序队列中等待扫描，直到用户手动停止该定时器为止。

Line 294：从新的排序队列头重新扫描swtmr，判断是否还有定时时间到的swtmr存在。