LiteOS内核任务调度机制分析

# 概述

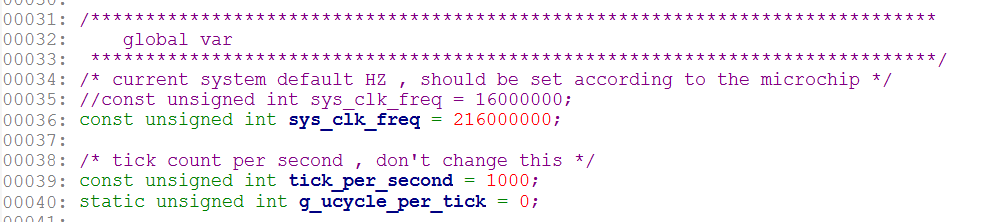
LiteOS内核任务调度基于高优先级抢占的时间片轮转调度算法，时间片基于系统tick产生。当有TASK需要被调度时，需将该TASK加入到对应的优先级队列中，并启动调度，每次调度都从优先级队列中选择优先级最高的TASK，同优先级之间严格按照时间片进行轮转调度，高优先级TASK强制抢占处理器，而不管正在执行的低优先级TASK的时间片是否结束，以达到系统实时响应的目的。

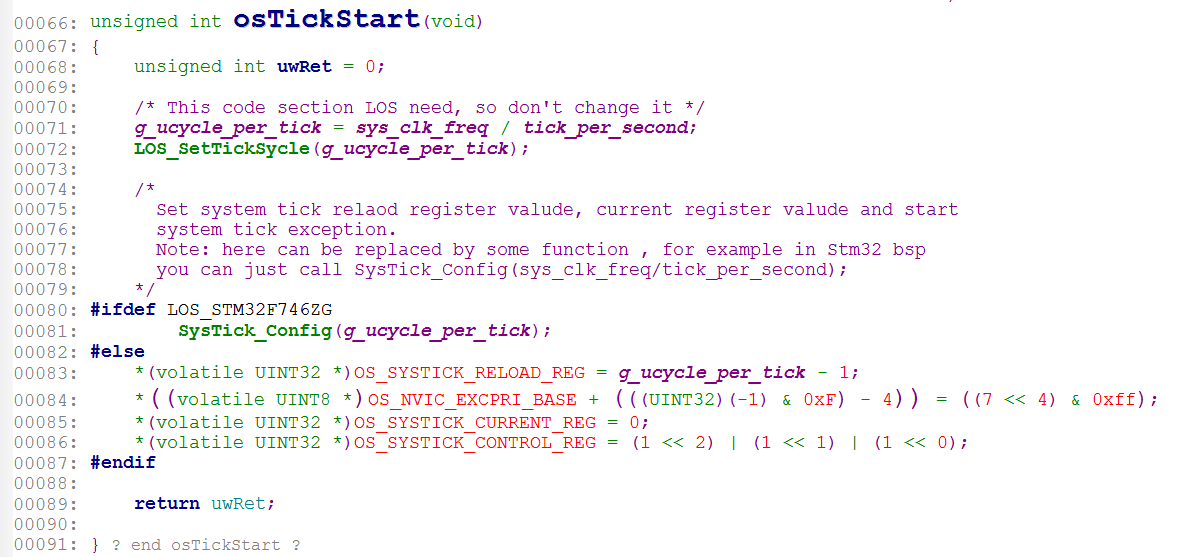
当正在运行的TASK因为某种原因需要进入阻塞状态时，该TASK会将自己从优先级队列中移除，移除后该TASK不再参与调度，并设置等待条件同时将该TASK加入到系统的排序队列中，每当处理系统tick中断时都会检测排序队列中的TASK的状态，如果有状态满足被调度条件的TASK，则将该TASK重新加入到优先级队列的尾部进而参与调度。

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。

# 系统tick

## 2.1 tick的配置及启动



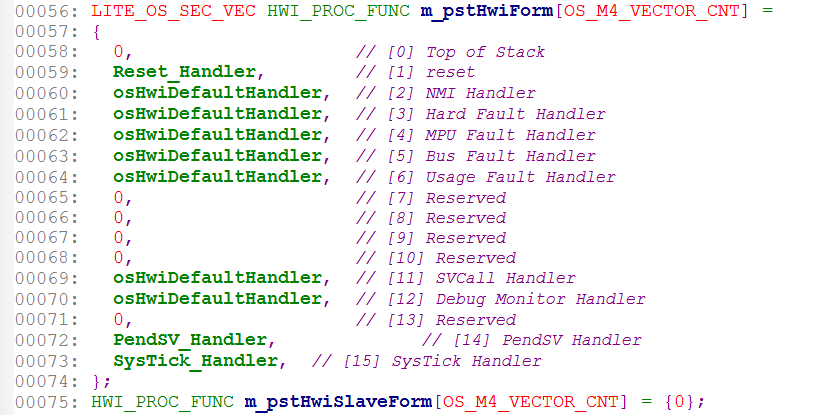


配置系统tick为1ms并启动系统tick。

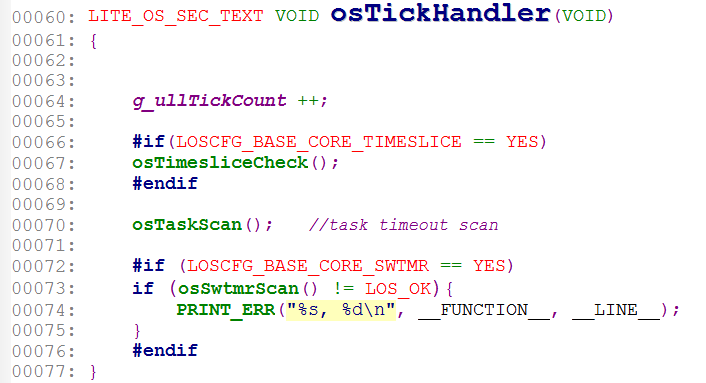
## 2.2 tick中断处理

系统tick每1ms产生一次中断，中断产生时CPU直接跳转至中断向量表中tick中断对应的中断向量处执行。

下图是中断向量表以及tick对应的中断处理函数。

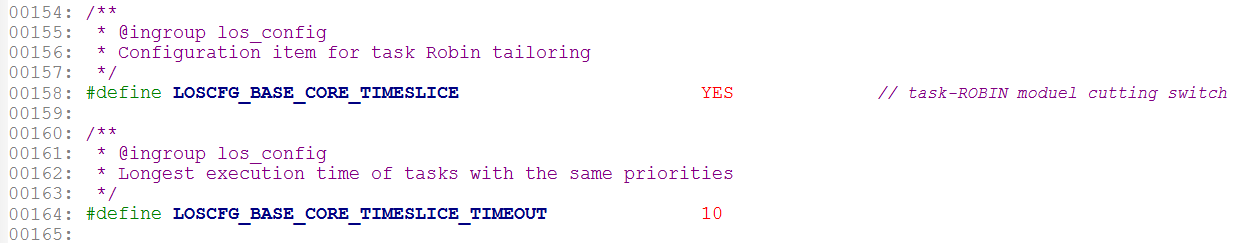


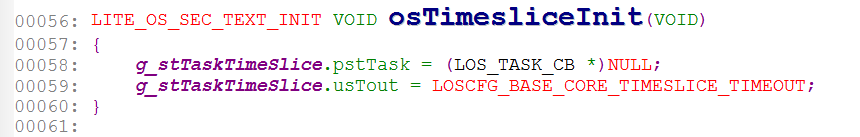
Tick的中断处理函数最后会调用到下图中的函数，该函数主要实现系统时间片的检测，系统排序队列中TASK的扫描以及重启调度，软件定时器的扫描功能。



# 时间片轮转

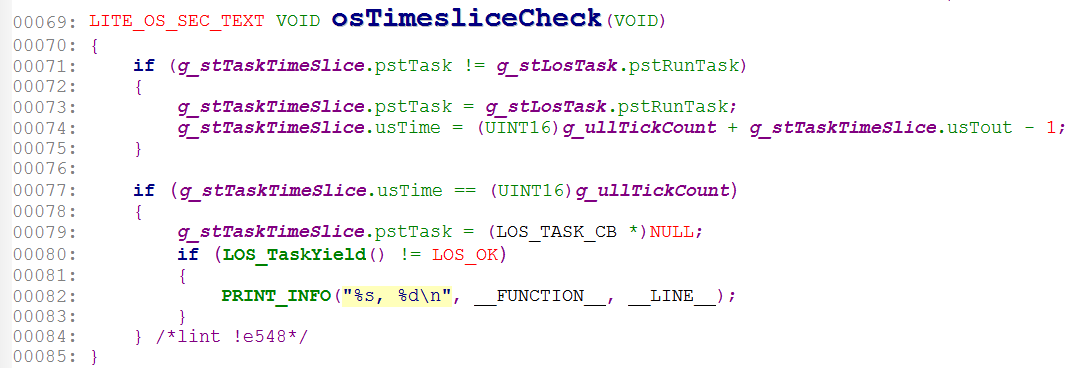
## 3.1 时间片的初始化





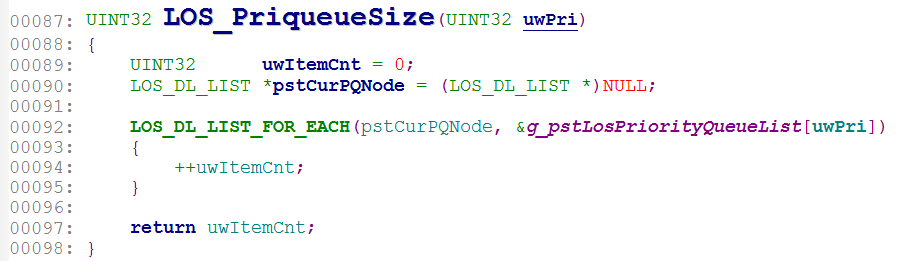
初始化使能时间片切换TASK功能及设置时间片为10个tick，就是10ms时间。

## 3.2 时间片的检测及时间片的轮转调度



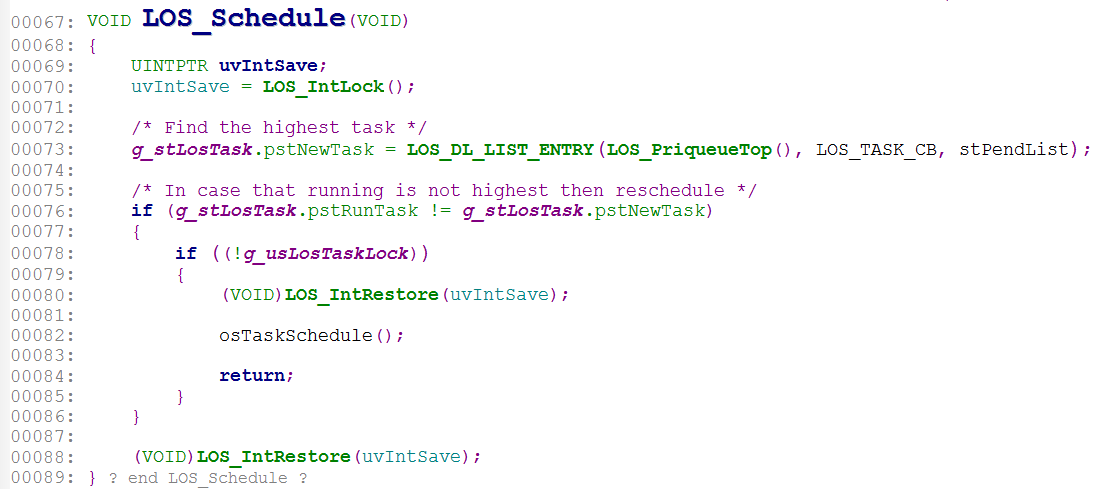
当前占用时间片的TASK（时间片用完以后为NULL）和当前正在执行的TASK不一致，则需要给当前正在执行的TASK分配时间片，时间片为10个tick时间。但因为当前正在执行的TASK是从上一个tick开始执行的（因系统调度发生在该判断的后面），等到此处判断时该TASK已经执行了一个tick的时间，所以正在执行的TASK的时间片就剩9个tick了，所以在设置当前正在执行的TASK的时间片时需要减去1个tick的时间。

当一次时间片结束时，需要将占用时间片的TASK置为NULL，以便为下一个TASK分配时间片，并启动下一个TASK的调度检测。





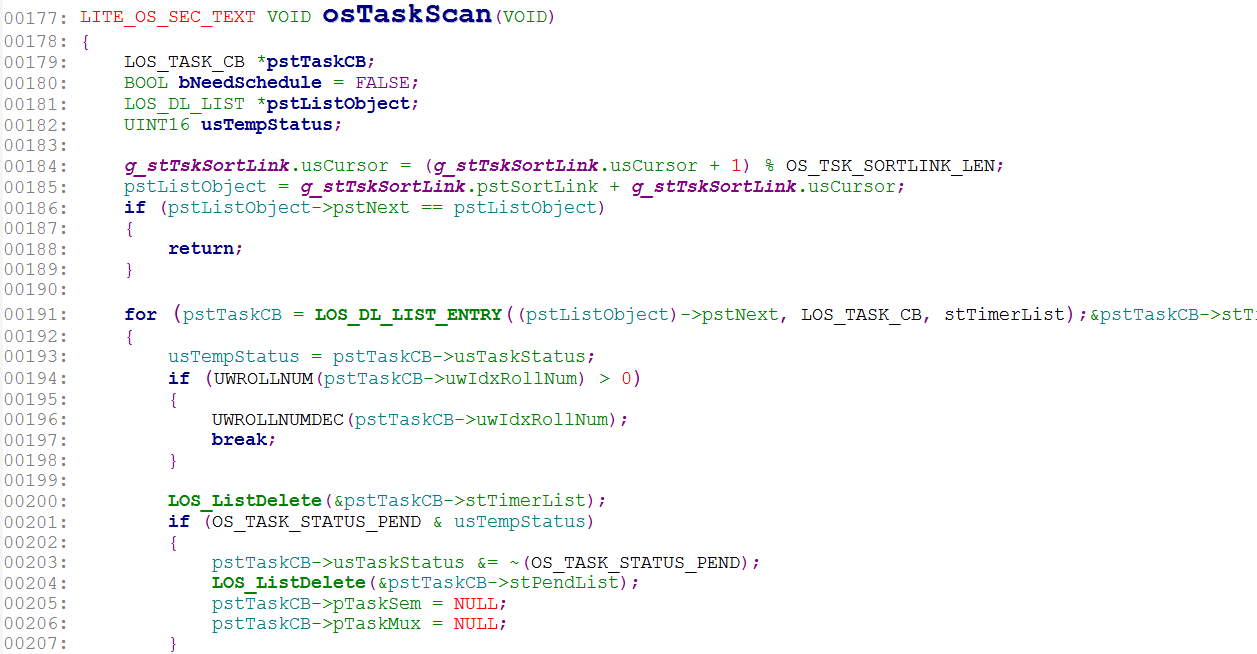
1. 调用函数LOS\_PriqueueSize( ) 检测当前正在运行的TASK对应的优先级队列(只包含同一优先级？？)中一共有多少个TASK在排队。
2. 如果得到的结果大于1，说明在当前的优先级队列中，除了正在运行的这个TASK以外还有其他的同优先级的TASK在排队等待调度，则需要严格按照时间片轮转的调度算法对同优先级的各个TASK依次进行调度。将当前正在执行的TASK从优先级队列的队头取下，然后重新插入到优先级队列的队尾等待下次调度，并调用函数LOS\_Schedule( ) 启动一次调度，调度执行当前处于队头的那个TASK，这样可保证同优先级的TASK执行的公平性。
3. 如果得到的结果是只有1个，说明在当前优先级的队列中除了正在运行的这个TASK以外没有其他TASK等待调度，所以直接返回，不启动调度切换，系统继续执行这个TASK，返回后下一次的时间片仍然分配给这个TASK，直到这个TASK因阻塞而被调出优先级队列或者有同优先级的TASK加入为保证公平让出时间片或者被更高优先级的TASK打断为止，否则这个TASK将一直执行下去，一直执行下去的原因在下面第4点中说明。

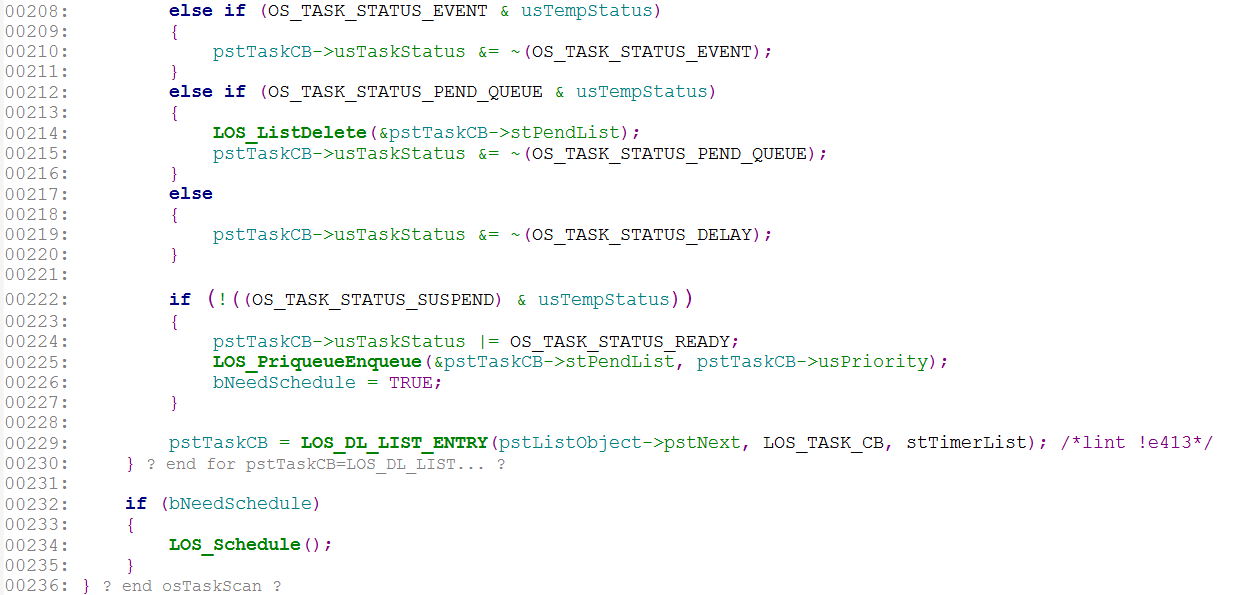


1. 上图中的LOS\_Schedule( ) 函数在后面说明，此处只说明优先级的问题。该函数中，系统每次都选择优先级队列中优先级最高的那一个TASK来执行，这就意味着：
2. : 当前正在执行的TASK与新得到的优先级最高的TASK不相等的情况只有三种，第一种情况是正在执行的TASK为NULL，需要新调入TASK来执行。第二种情况是同优先级的TASK被时间片切换调度。第三种情况是新得到的TASK的优先级比当前正在执行的TASK的优先级高，需要紧急处理。
3. : 即便有低优先级的TASK在排队等待，只要高优先级的TASK没有离开优先级队列，那么低优先级的TASK就没有被调度到的可能。
4. : 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。

# 排序队列中TASK的扫描及调度

## 4.1 TASK的扫描





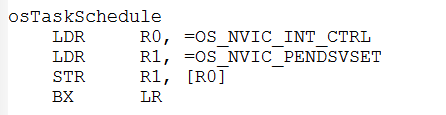
每当正在运行的TASK因某种原因需要进入阻塞状态时，该TASK将自己从优先级队列中移除，设置阻塞条件，并将自己加入到排序队列。每个系统tick中断都会扫描排序队列中的TASK，以判断是否有满足继续执行条件的TASK。当有满足条件的TASK产生时，将其从排序队列取下，重新加入到优先级队列等待调度，并重启一次调度切换以保证系统的实时响应，至于该TASK能否被调度到，则要依据该TASK的优先级和系统正在运行的TASK的优先级而定。

当排序队列为空时，表明系统当前没有处于阻塞状态的TASK需要被唤醒及调度，故直接返回，否则扫描排序队列。

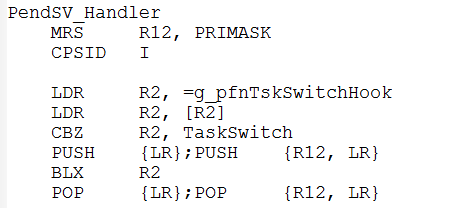
为了保证系统的实时响应，每个tick中断都会检测TASK的状态并尝试启动TASK调度切换，如果有更高优先级的任务满足重新调度的条件，则最迟会在一个tick之后被重新调度到。

## 4.2 系统调度

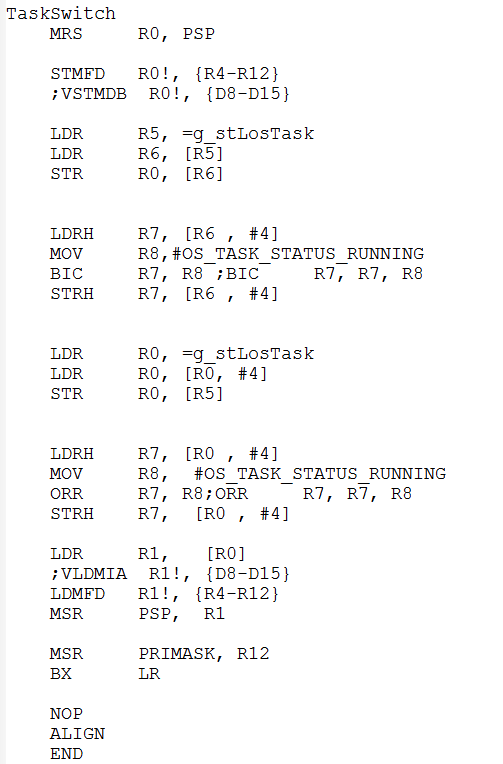
系统调度时每次选择优先级队列中优先级最高的那一个TASK，如果新选到的TASK与当前的TASK不一致，则说明有更高优先级的TASK或同优先级的TASK需要调度，此时需要切换任务，调用函数osTaskSchedule( ) 来完成切换TASK的功能。



切换TASK时通过产生更高优先级的PENDSV中断来完成。



在PENDSV中断处理中，通过调用TaskSwitch来完成TASK的切换工作。



TASK切换完成，当所有中断退出后，CPU调度新的TASK执行。