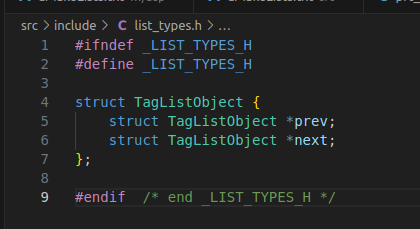
**实验六 任务调度**

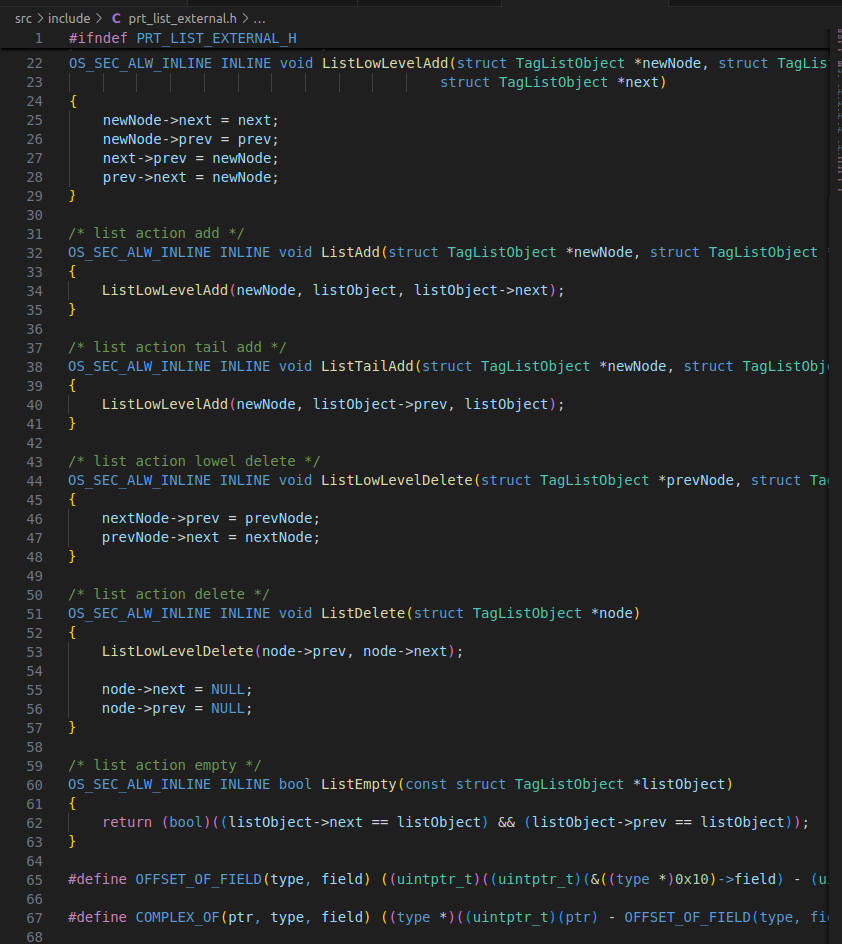
任务调度是操作系统的核心功能之一。 UniProton实现的是一个单进程支持多线程的操作系统。在UniProton中，一个任务表示一个线程。UniProton中的任务为抢占式调度机制，而非时间片轮转调度方式。高优先级的任务可打断低优先级任务，低优先级任务必须在高优先级任务挂起或阻塞后才能得到调度。

**基础数据结构：双向链表**

双向链表结构在 src/include/list\_types.h 中定义。



此外，在 src/include/prt\_list\_external.h 中定义了链表各种相关操作。



这里面最有意思的是 LIST\_COMPONENT 宏，其作用是根据成员地址得到控制块首地址, ptr成员地址, type控制块结构, field成员名。

LIST\_FOR\_EACH 和 LIST\_FOR\_EACH\_SAFE 用于遍历链表，主要是简化代码编写。

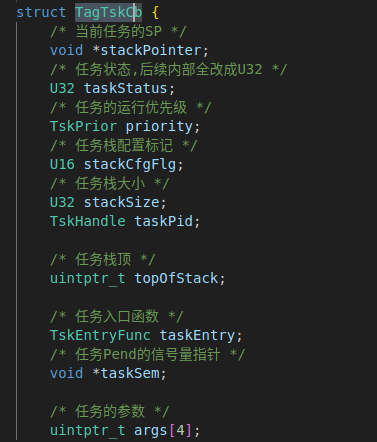
**任务控制块**

任务相关的头文件主要包括 src/include/prt\_task.h 和 src/include/prt\_task\_external.h两个头文件。此外还会用到 src/include/prt\_module.h 和 src/include/prt\_errno.h 两个头文件。 prt\_module.h中主要是一些模块ID的定义，而 prt\_errno.h 主要是错误类型的相关定义，引入这两个头文件主要是为了保持接口与原版 UniProton 相一致。

prt\_task.h 中除了一些相关宏定义外，还定义了任务创建时参数传递的结构体： struct TskInitParam。

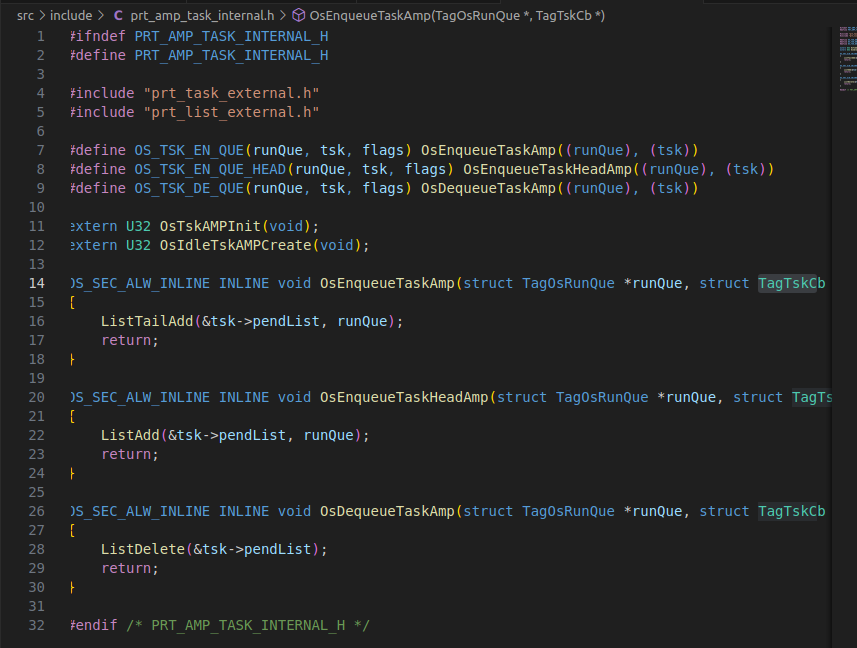


prt\_task\_external.h 中定义了任务调度中最重要的数据结构——任务控制块 struct TagTskCb。



简单起见，我们还将任务运行队列结构 TagOsRunQue 直接定义为双向链表 TagListObject（见上面代码）。

最后我们引入了 src/include/prt\_amp\_task\_internal.h



该头文件中主要是定义了三个内联函数，用于将任务控制块加入运行队列或从运行队列中移除任务控制块。

**任务创建**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#id8)

任务创建代码主要在 src/kernel/task/prt\_task\_init.c 中。 代码比较多，我们分几个部分分别介绍。

**相关变量与函数声明**

首先是引入必要的头文件。

然后声明了 1 个全局双向链表，并通过 LIST\_OBJECT\_INIT 宏进行初始化。 g\_tskCbFreeList 链表是空闲的任务控制块链表。

最后声明了3个外部函数。



其中头文件 src/include/prt\_asm\_cpu\_external.h包含内核相关的一些状态定义。

**极简内存空间管理**

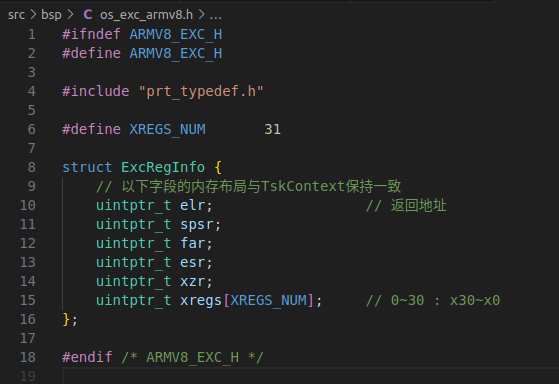
内核运行过程中需要动态分配内存。我们实现了一种极简的内存管理，该内存管理方法仅支持4K大小，最多256字节对齐空间的分配。

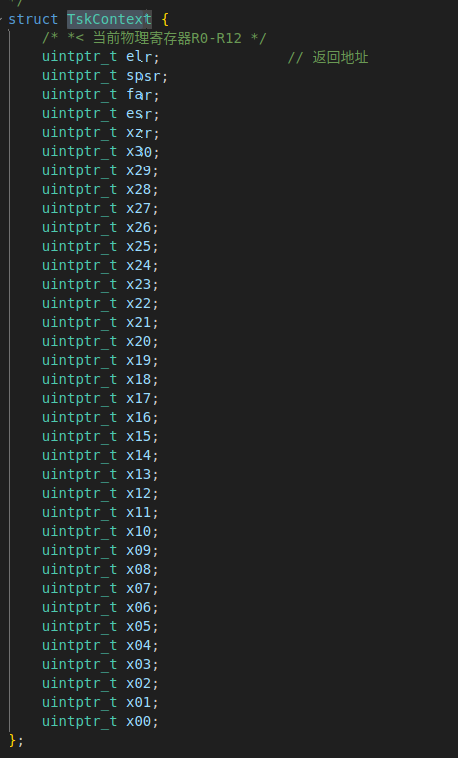
**任务栈初始化**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#id12)

在理论课程中，我们知道当发生任务切换时会首先保存前一个任务的上下文到栈里，然后从栈中恢复下一个将运行任务的上下文。可是当任务第一次运行的时候怎么恢复上下文，之前从来没有保存过上下文？

答案就是我们手工制造一个就可以了。下面代码中 stack->x01 到 stack->x29 被初始化成很有标志性意义的值，其他他们的值不重要。比较重要的是 stack->x30 和 stack->spsr 等处的值。

struct TskContext 表示任务上下文，放在 src/bsp/os\_cpu\_armv8.h 中定义。在我们的实现上它与中断上下文 struct ExcRegInfo (在 src/bsp/os\_exc\_armv8.h 中定义)没有区别。在UniProton中，它们的定义有一些差别。

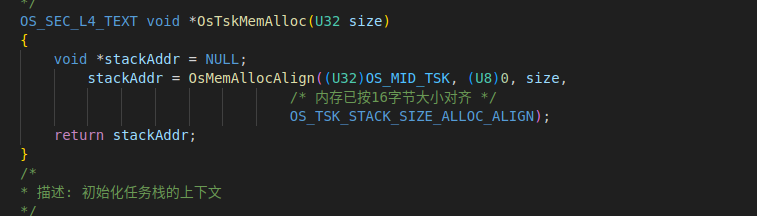
在 src/bsp/os\_cpu\_armv8.h 中加入 struct TskContext 定义。



**任务入口函数**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#id13)

这个函数有几个有趣的地方。（1）你找不到类似 OsTskEntry(taskId); 这样的对 OsTskEntry 的函数调用。这实际上是在通过 OsTskContextInit 函数进行栈初始化时传入的，也就意味着当任务第一次就绪运行时会进入 OsTskEntry 执行。（2）用户指定的 taskcb->taskEntry 不一定要求是 4 参数的，可以是 0~4 参数之间任意选定，这个需要你在汇编层面去理解。

采用 OsTskEntry 的好处是在用户提供的 taskCb->taskEntry 函数的基础上进行了一层封装，比如可以确保调用taskCb->taskEntry执行完后调用 OsTaskExit。

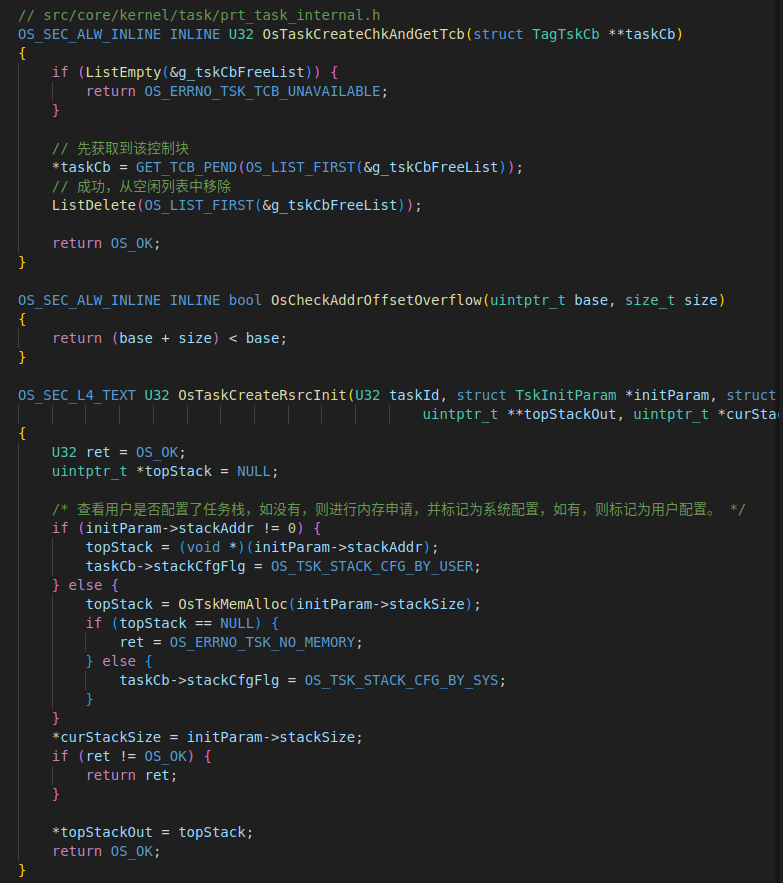


**创建任务**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#id14)

创建任务的代码看上去还是比较多，但已经不是很复杂了。我们从后面的代码往前面看，首先是接口函数 PRT\_TaskCreate 函数根据传入的 initParam 参数创建任务返回任务句柄 taskPid。

PRT\_TaskCreate 函数会直接调用 OsTaskCreateOnly 函数实际进行任务创建。OsTaskCreateOnly 函数将：

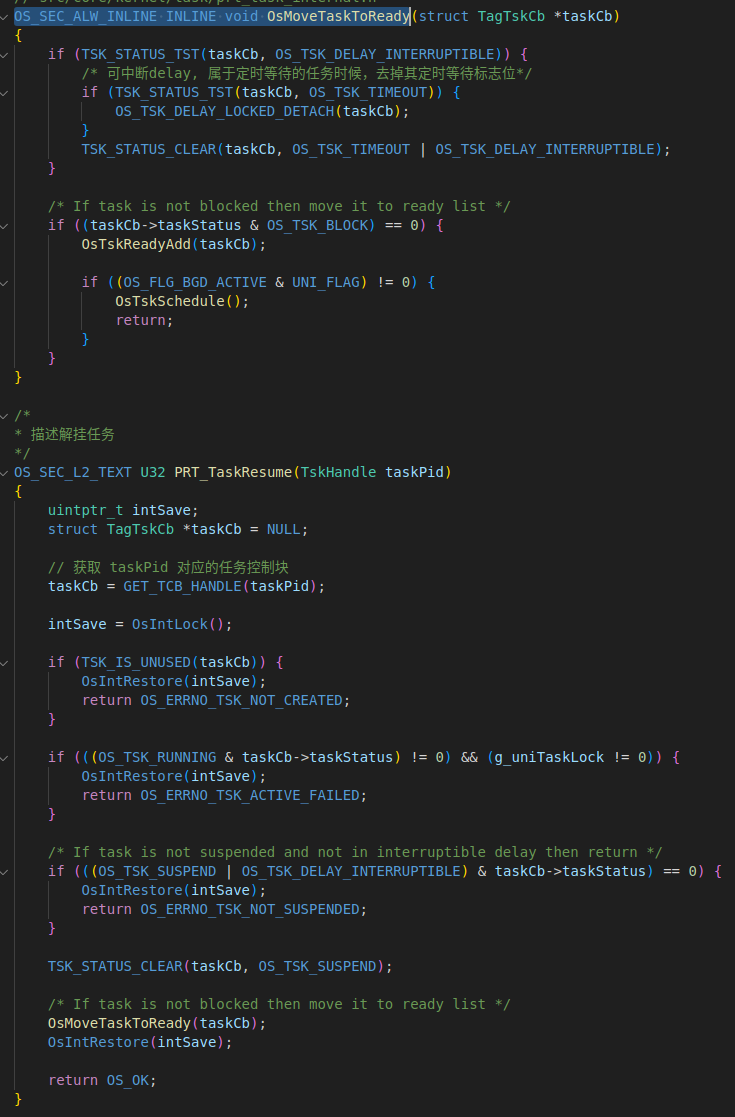
* 通过 OsTaskCreateChkAndGetTcb 函数从空闲链表 g\_tskCbFreeList 中取一个任务控制块；
* 在 OsTaskCreateRsrcInit 函数中，如果用户未提供堆栈空间，则通过 OsTskMemAlloc 为新建的任务分配堆栈空间；
* OsTskContextInit 函数负责将栈初始化成刚刚发生过中断一样；
* OsTskCreateTcbInit 函数负责用 initParam 参数等初始化任务控制块，包括栈指针、入口函数、优先级和参数等；
* 最后将任务的状态设置为挂起 Suspend 状态。这意味着 PRT\_TaskCreate 创建任务后处于 Suspend 状态，而不是就绪状态。



**解挂任务[ℑ](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html" \l "id15" \o "此标题的永久链接)**

PRT\_TaskResume 函数负责解挂任务，即将 Suspend 状态的任务转换到就绪状态。PRT\_TaskResume 首先检查当前任务是否已创建且处于 Suspend 状态，如果处于 Suspend 状态，则清除 Suspend 位，然后调用 OsMoveTaskToReady 将任务控制块移到就绪队列中。

OsMoveTaskToReady 函数将任务加入就绪队列 g\_runQueue，然后通过 OsTskSchedule 进行任务调度和切换（稍后描述）。 由于有新的任务就绪，所以需要通过OsTskSchedule 进行调度。这个位置一般称为调度点。对于优先级调度来说，找到所有的调度点并进行调度非常重要。



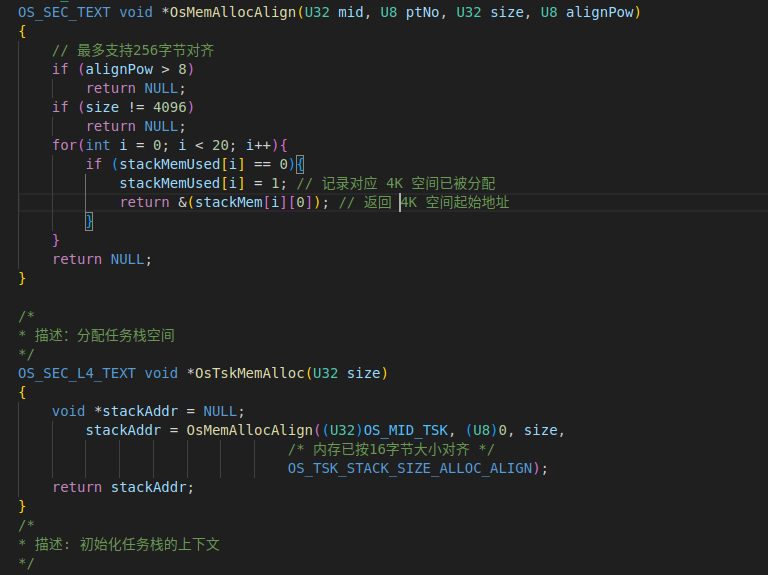
**任务管理系统初始化与启动**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#id16)

OsTskInit 函数通过调用 OsTskAMPInit 函数完成任务管理系统的初始化。主要包括：

* 为任务控制块分配空间，由于我们只实现了简单的内存分配算法，所以支持的任务控制块数目为：4096 / sizeof(struct TagTskCb) - 2; 减去2是因为预留了 1 个空闲任务， 1 个无效任务。
* 将所有分配的任务控制块加入空闲任务控制块链表 g\_tskCbFreeList， 并对所有控制块进行初始化。
* 任务就绪链表 g\_runQueue 通过 INIT\_LIST\_OBJECT 初始化为空。
* RUNNING\_TASK 目前指向无效任务。

OsActivate 启动多任务系统。

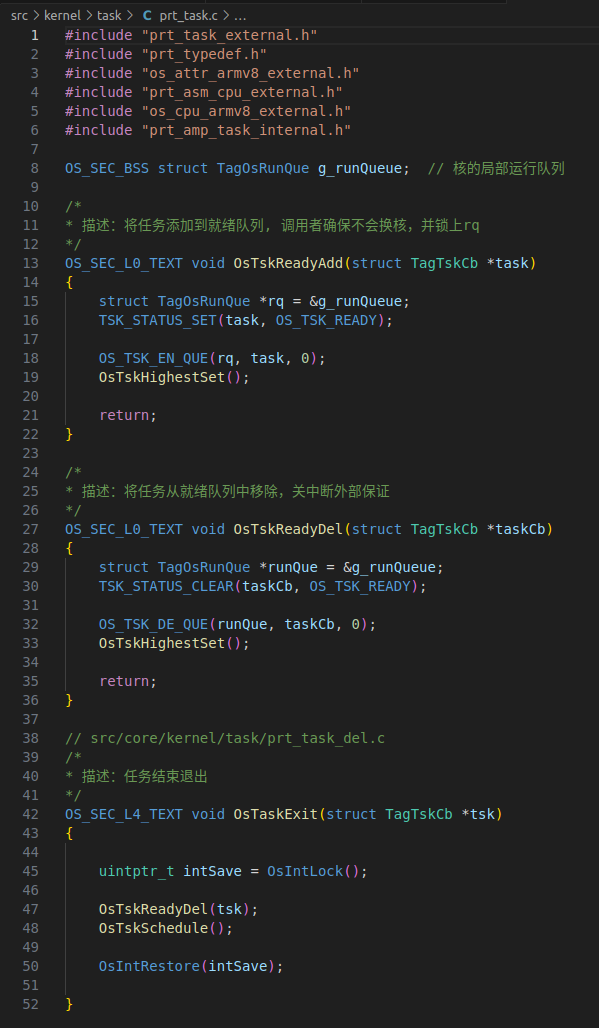
* 首先通过 OsIdleTskAMPCreate 函数创建空闲任务，这样当系统中没有其他任务就绪时就可以执行空闲任务了。
* OsTskHighestSet 函数在就绪队列中查找最高优先级任务并将 g\_highestTask 指针指向该任务。
* UNI\_FLAG 设置好内核状态
* OsFirstTimeSwitch 函数将会加载 g\_highestTask 的上下文后执行（稍后描述）。



**任务状态转换**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#id17)

在 src/kernel/task/prt\_task.c 中，

* 声明了运行队列 g\_runQueue， 注意我们之前已经将其定义为双向队列。
* **提供了将任务添加到就绪队列的 OsTskReadyAdd 函数和从就绪队列中移除就绪队列的 OsTskReadyDel 函数。**
  + OsTskReadyAdd 会设置任务为就绪态
  + OsTskReadyDel 会清除任务的就绪态
* 提供了任务结束退出 OsTaskExit 函数，注意 OsTskEntry 中会调用 OsTaskExit 函数。由于任务退出，因此需要进行调度，即存在调度点，所以调用 OsTskSchedule 函数。



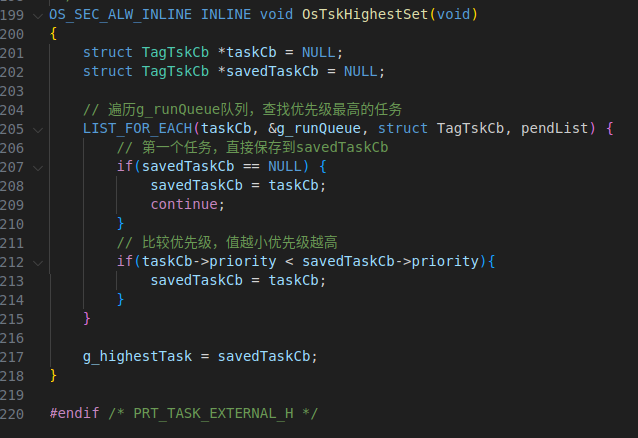
其中，OS\_TSK\_EN\_QUE 和 OS\_TSK\_DE\_QUE 宏在 src/include/prt\_amp\_task\_internal.h 定义。

**调度与切换**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#id18)

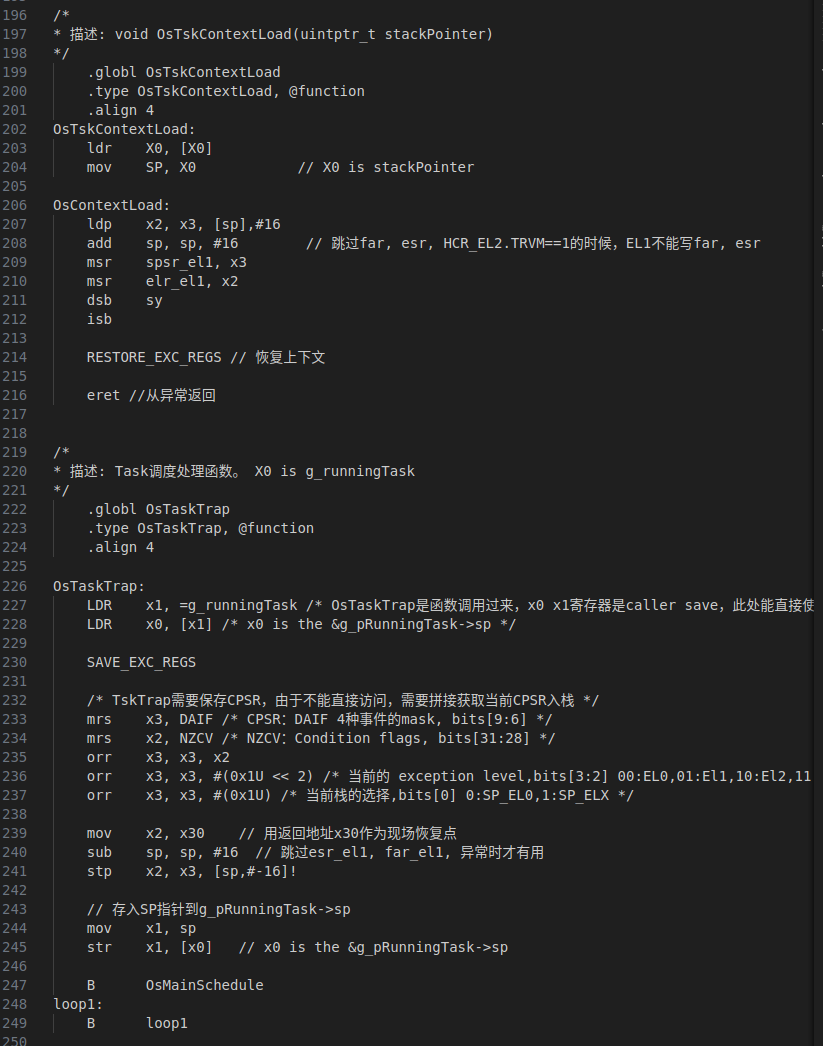
src/kernel/sched/prt\_sched\_single.c



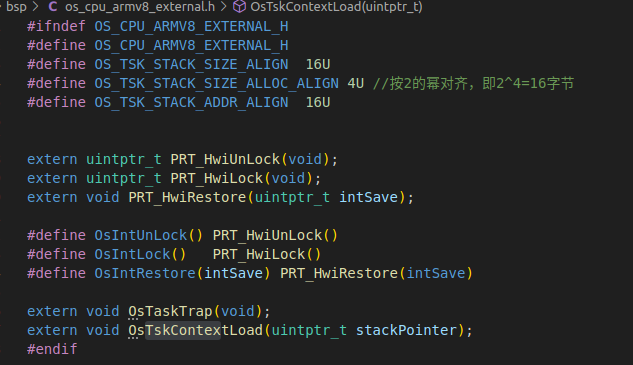
其中，OsTskHighestSet 函数在 src/include/prt\_task\_external.h 中被定义为内联函数，提高性能。



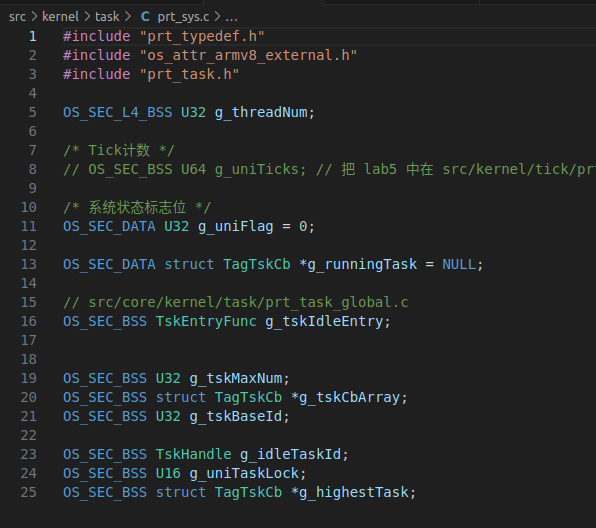
在 src/bsp/prt\_vector.S 实现 OsTskContextLoad，OsContextLoad 和 OsTaskTrap。



在 src/bsp/os\_cpu\_armv8\_external.h 加入 OsTaskTrap 和 OsTskContextLoad 的声明和关于栈地址和大小对齐宏。



最后在 src/kernel/task/prt\_sys.c 定义了内核的各种全局数据。

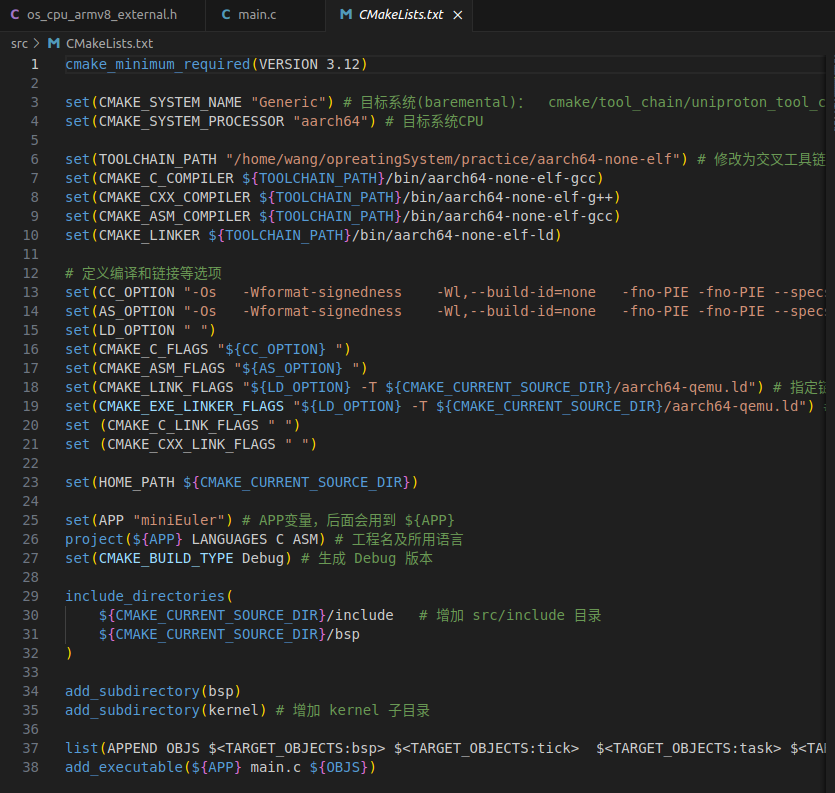


**任务调度测试**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#id19)



**提示**

将新建文件加入构建系统



**lab6 作业**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#lab6)

**作业1**[**ℑ**](https://os2024lab.readthedocs.io/zh-cn/latest/lab6/index.html#id20)

实现分时调度。

