目录

[1. 任务管理-信号量-消费者模拟 2](#_Toc198499355)

[【视频】任务管理 2](#_Toc198499356)

[【解析】信号量 源码 4](#_Toc198499357)

[【操作】生产者与消费者同步模型 15](#_Toc198499358)

[2. 定时-互斥锁-打印机模型 18](#_Toc198499359)

[【视频】 定时器的简单使用 18](#_Toc198499360)

[【视频】互斥 锁 19](#_Toc198499361)

[【解析】互斥锁 21](#_Toc198499362)

[【操作】打印机资源模型 23](#_Toc198499363)

本报告中包含源码解析，检查时可以按照页面直接跳转作业部分。

1. 任务管理-信号量-消费者模拟

【视频】任务管理

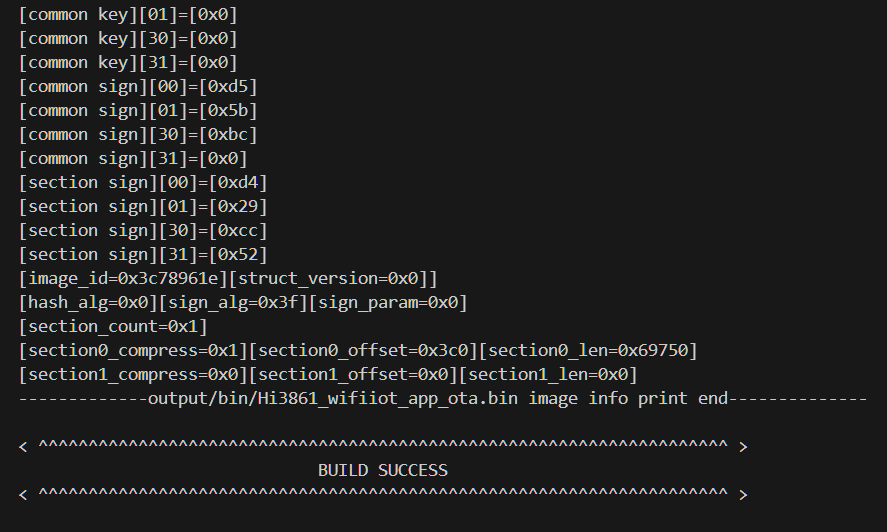
先将视频中使用的代码复现，选中线程示范1进行编译



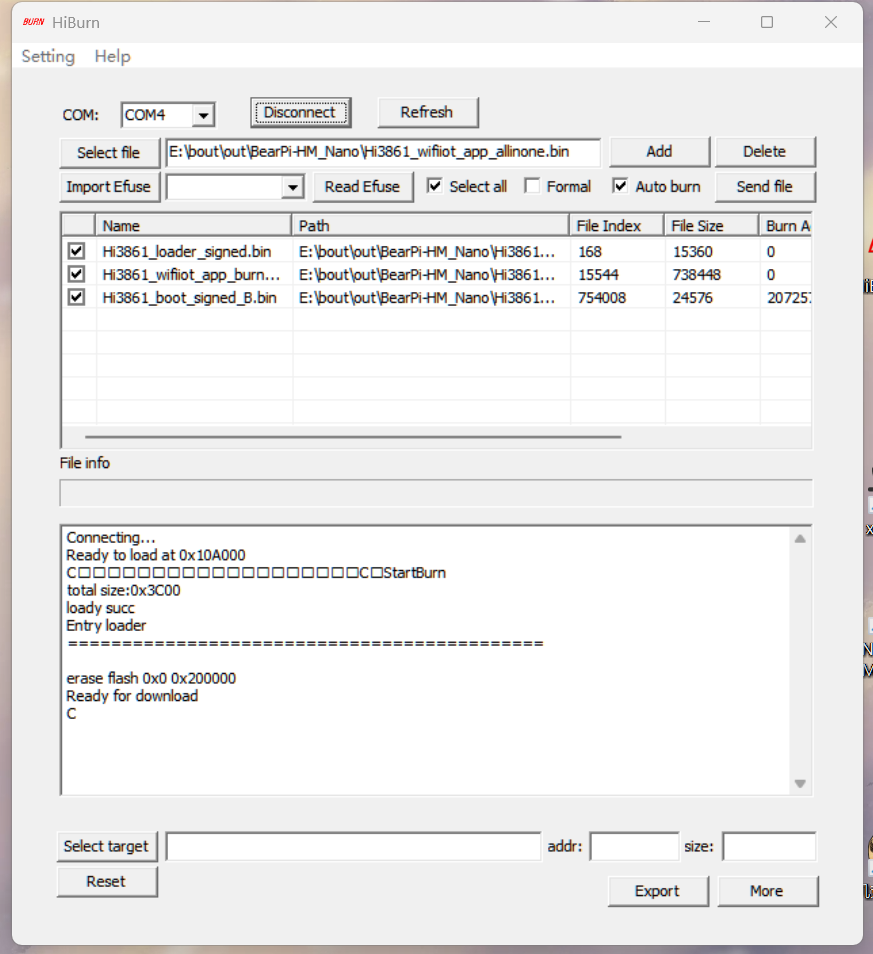
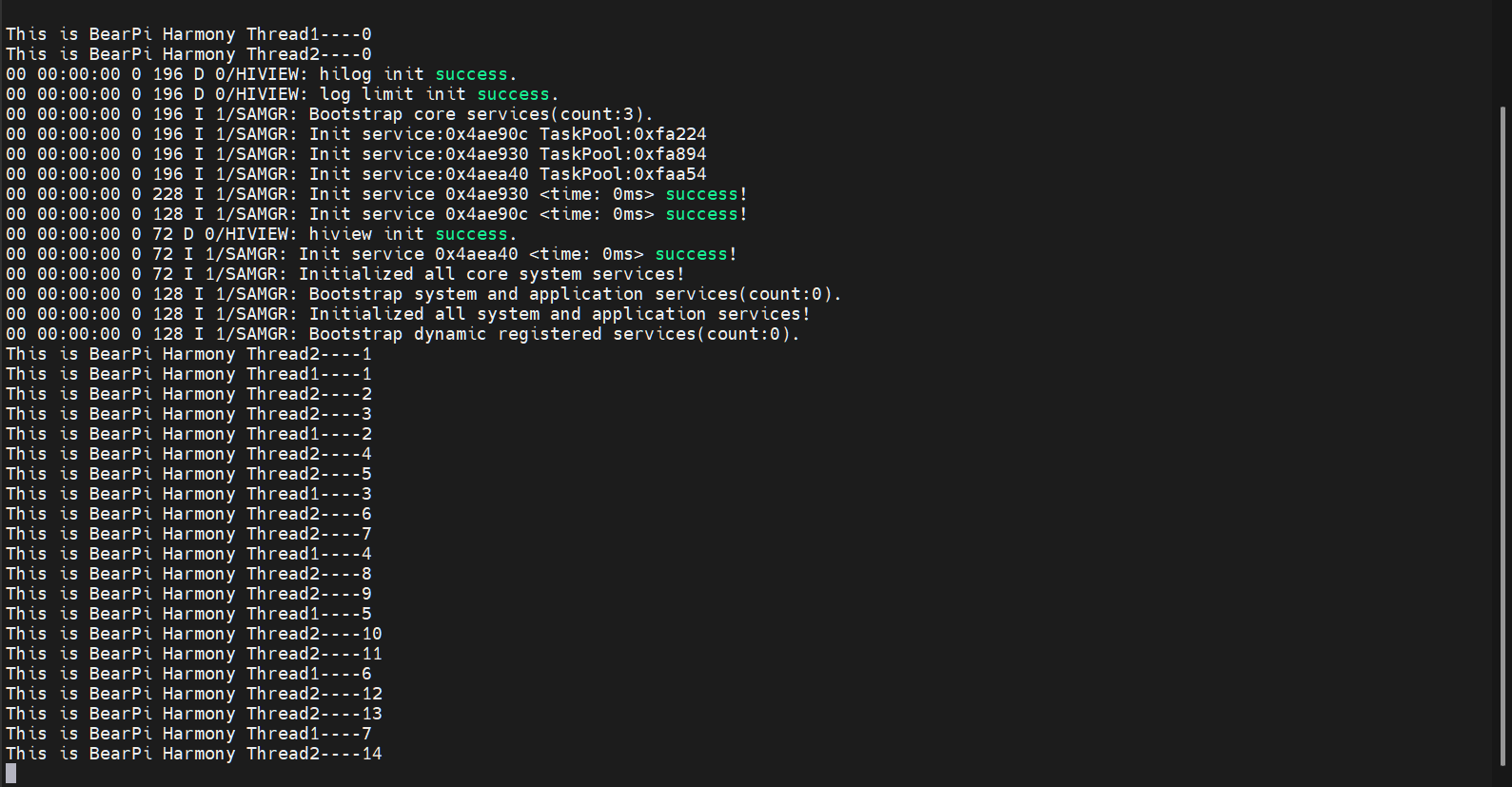
执行编译命令

* sh build.sh

等待编译成功

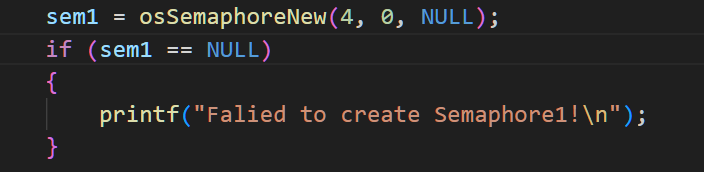


进行烧录，然后本地查看串口输出



可以发现成功打印

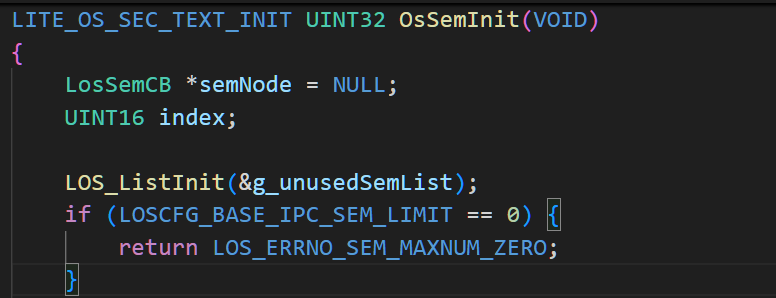
【解析】信号量 源码



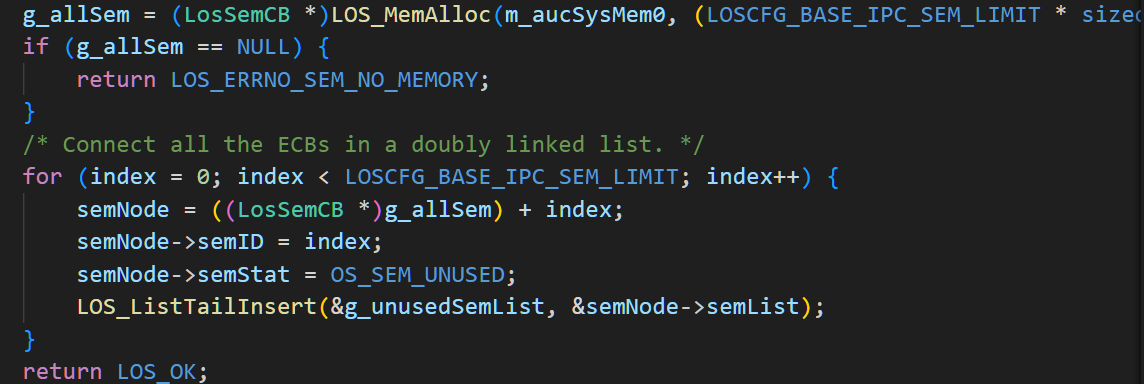
这是一个简单的信号量创建，我们查一下这个库函数有哪些操作。

1. 底层信号量初始化

信号量的底层在初始化时候有两个重要概念，一个是信号量池，是一块连续内存，存储了所有信号量，也可以理解为数组。第二个是空闲信号量队列

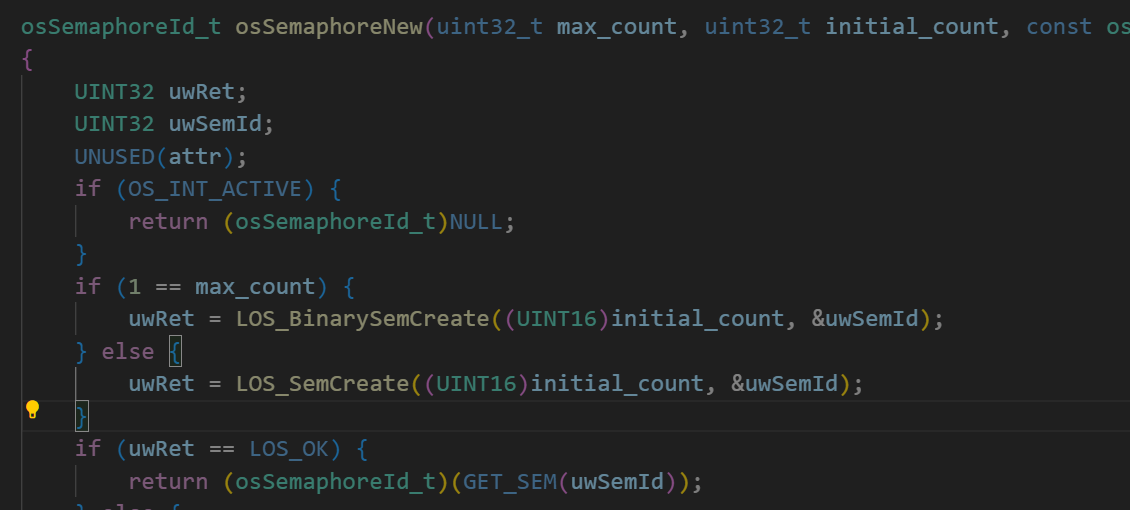


第一步先将队列进行初始化

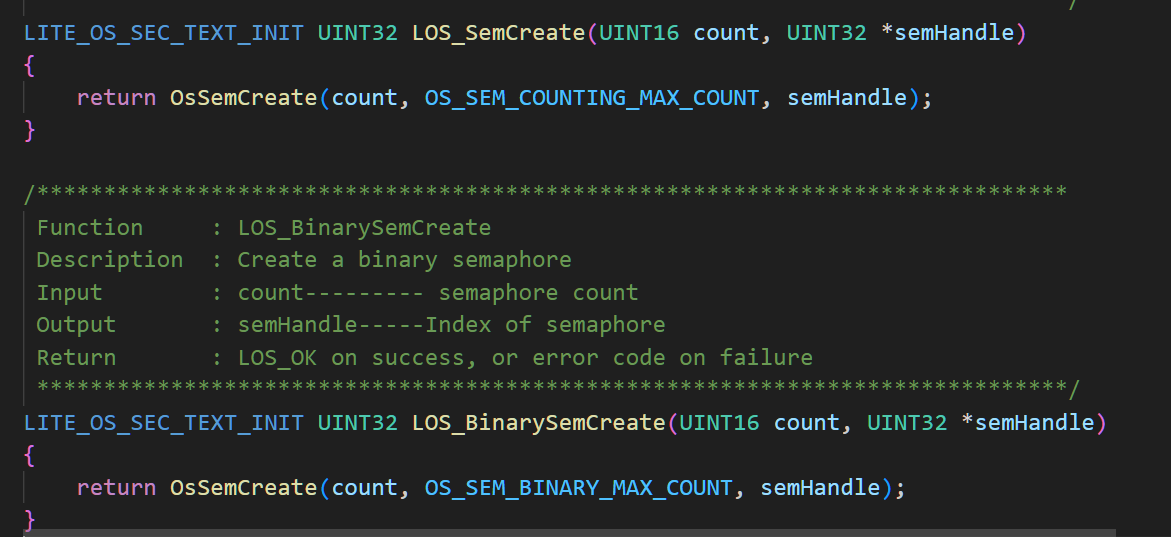


然后分配出一块连续内存，再将里面的双指针组成的节点标志位进行初始化，将这些标志位插入到未使用队列的尾部。

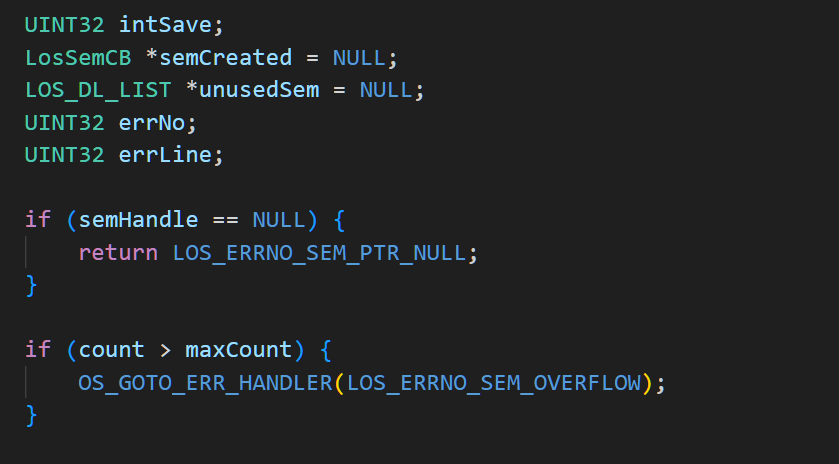
1. 信号量创建



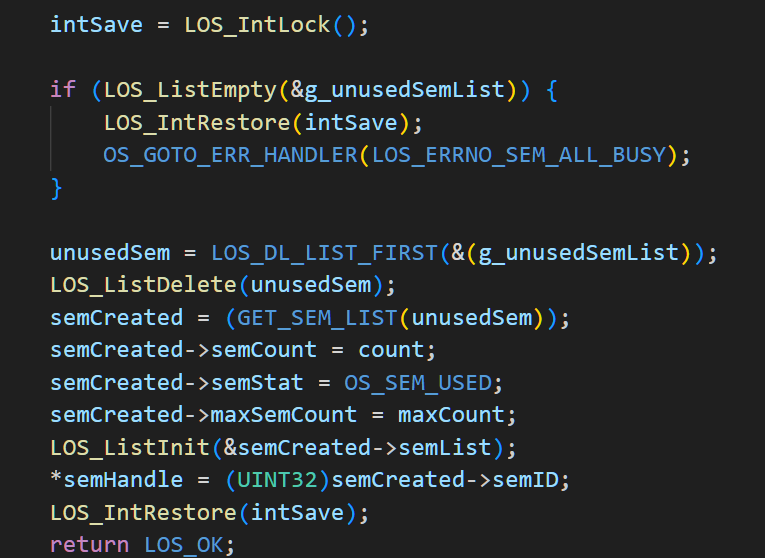
创建时会有选项，会区别创建为二进制信号量和变量类型信号量。



不过进一步解析后就会发现这两个底层调用是一个构造的函数只是中间对外的接口不同，然后二进制上限传参1。

首先观察Create源码，内部还有一个套娃，然后是真正的源码。

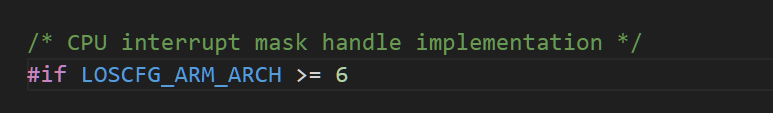
首先是初始化与错误处理，其中初始的信号量能设置最大值是65536



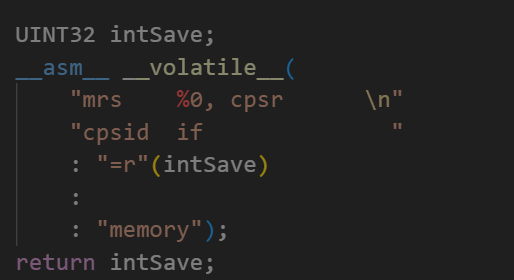
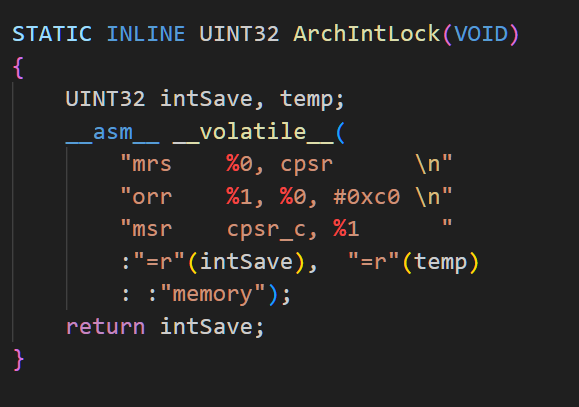
接下来观察这段核心代码，首先进行了一个原子操作，保证从信号量池中取出一个信号量的完整性。

如果不做限制，可能链表处理过程中发生竞态，比如读取了unusedSem但是还没来得及删除，就被抢占。导致两个信号量实际共用一个地址。

接下来深入内核层面，查看是怎么实现原子操作的，然后就会发现这是一个基于arm-arch的汇编，也就是说这个芯片是ARM Cortex-M4 CPU。



汇编会先读取当前arm的版本因为v6之后拥有更完备的指令集，**cpsid**（关闭中断）和 **cpsie**（开启中断）这两个，能直接操作寄存器中断标志。当寄存器中正处理的数据不被中断时候，目前处理的这个变量就不会被插手。

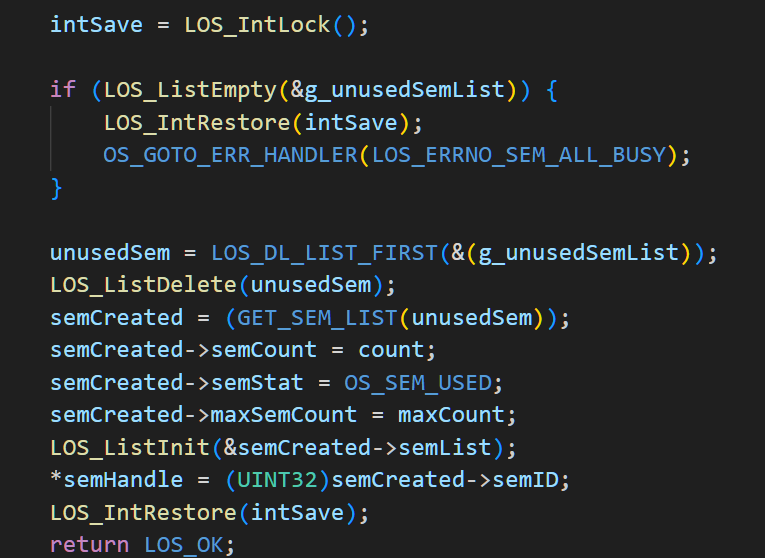


（左侧为v6+，右侧v6-）

当版本不够时候就有更加古法的操作😢，将cprs载入之后手动设置中断位，然后再放回去。

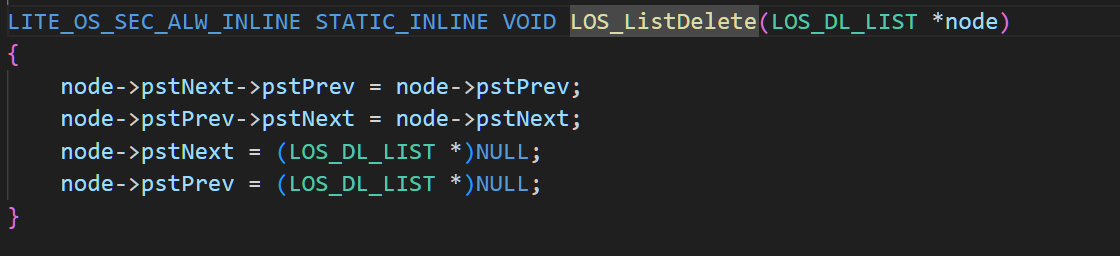
但是这两种修改都是基于寄存器中，中断位控制实现对变量的原子操作，也就是说这个代码适用于一个单核心的芯片，至于bearpi是不是单核心，我没了解。如果在多核心情况下，就会发生这边即便不被打断，别的核心还是能访问到这个链表。

经过了解在后面的armv7版本后，可以更轻松的在指令中编码自旋锁或者原子指令等新的指令集来达到真正的多核心使用公共资源😎。



回到之前的代码，可以发现就是比较正常的链表操作，将空信号量的一个节点标志从队列中取出🙌。这个模型好像有个专有名词叫，侵入式链表。

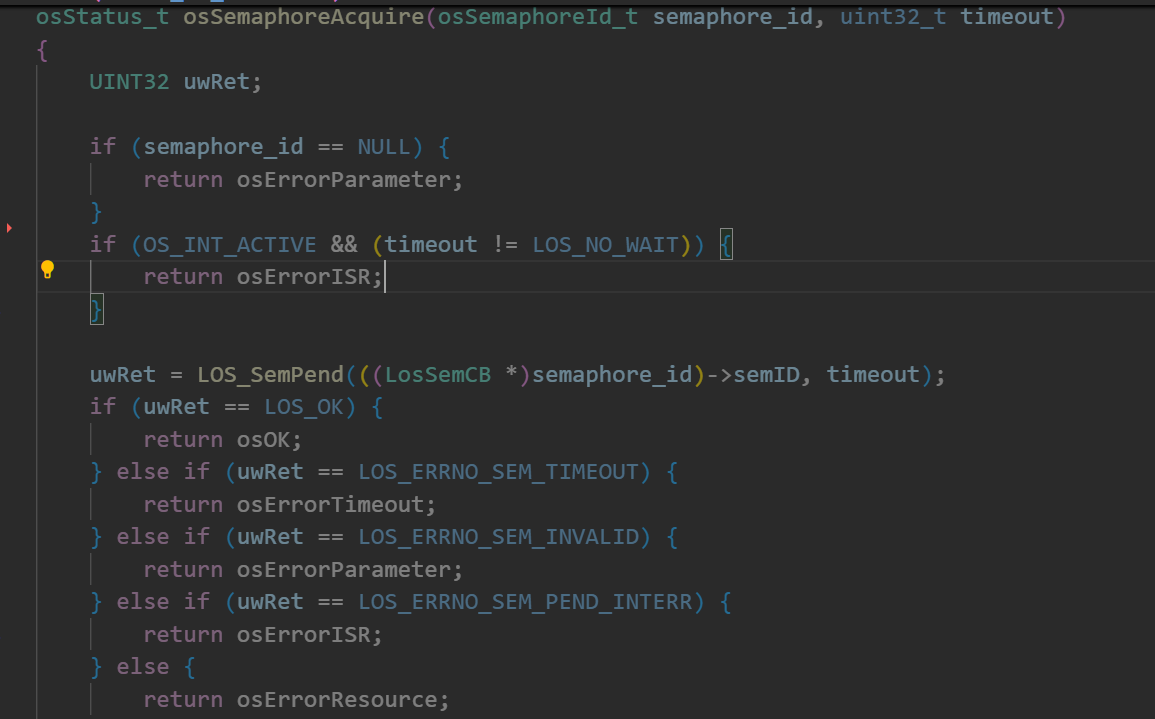
然后我们知道这个节点标志是一个完整信号量结构体中的一个属性，就可以根据完整结构体算出内存偏移，进而获取数组中完整的信号量变量地址然后返回id，也就是说最后到咱们手中的变量就是一个变量的id。



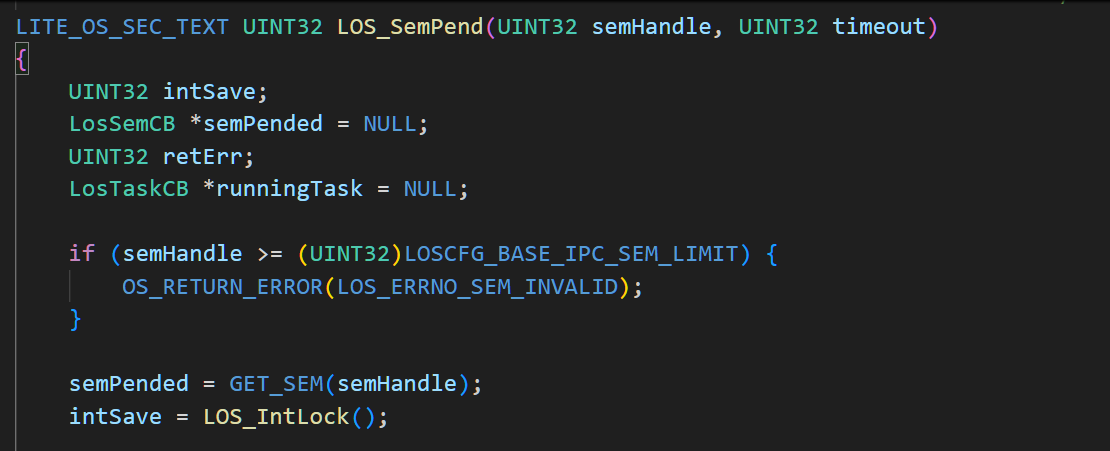
继续向下扒，就能看到，这个信号量关联一个任务队列，所有使用这个信号量的任务都会被放进队列中进行阻塞。

接下来看看是怎么实现PV操作的。

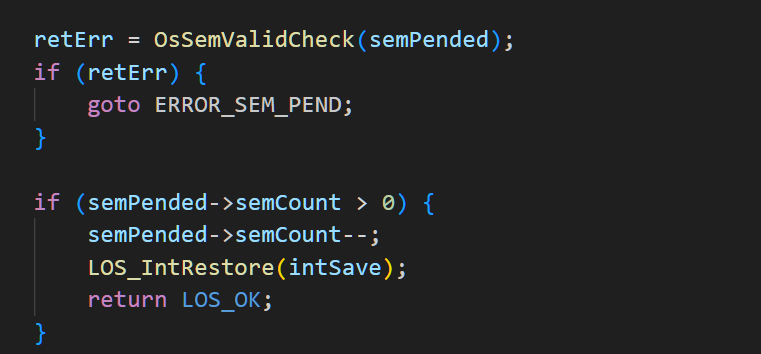
1. PV【信号量等待与释放】



P操作的最外部接口是一层完备的错误处理。想要了解底层就要进入 **os** 进行研究🧐sem\_pend。

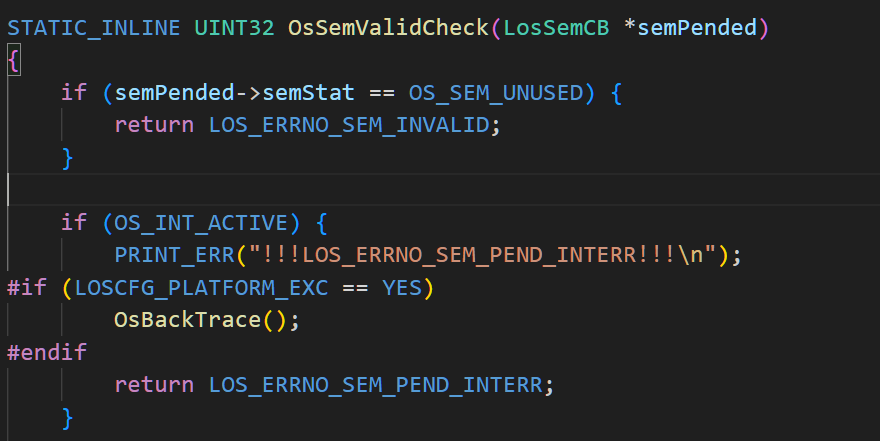


初始化，错误处理，句柄获取。这个Get\_sem的原理就是之前说的分配的一块完整的连续内存【数组】中找到指定偏移位【下标】的元素。然后IntLock开启原子操作保证接下来的完整性。😶‍🌫️

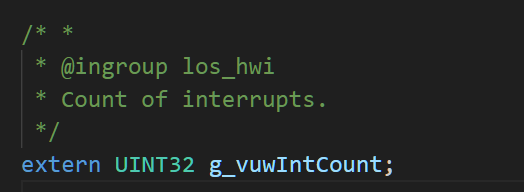


这个是错误处理部分。

错误处理中首先检查了Valid，就是一个检测有效性的错误排查。



进入初期检测传入的信号的使用标签【stat】是否是没被使用，如果没被使用就说明发生其他情况，这个没有被创建的信号量被使用了，可能发生了泄露😭。



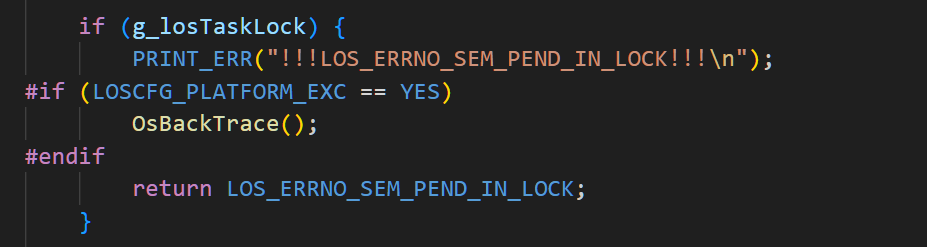
然后是 OS\_INT\_ACTIVE  中断上下文检测，#define OS\_INT\_ACTIVE (g\_vuwIntCount > 0)。这个g\_vuwIntCount是一个全局计数器，这个计数器中负责记录嵌套中断，会在中断任务出现时候++，结束时候--，零的时候就是任务上下文状态。

好接下来，我猜就会出现一个问题:什么是中断？🥺

* 中断是操作系统和硬件协同工作的核心机制之一。当某个硬件设备（如定时器、网络接口以及小熊派上的各个组件）需要 CPU 处理紧急事件时，会向 CPU 发送一个 **中断信号**，CPU 会立即暂停当前任务，跳转到预定义的 **中断服务程序（ISR）** 处理该事件，处理完成后恢复原任务。

以上是中断的定义，也就是说我们当前信号量的操作不能发生在中断程序运行期间，如果发生了就会报错。

接下来是最后一个有效性检测 **任务调度锁定检查**



这个是任务切换相关的逻辑，当这个Lock变量大于零时候表示：任务调度已经被锁定，被锁定时候是**不能**自然切换的。

调度普通情况有三种😗：

* **时间片耗尽**：通过定时器中断触发调度。
* **任务主动让出 CPU**：调用 **OS中的特定函数**。
* **资源等待**：任务因等待信号量、队列等挂起。

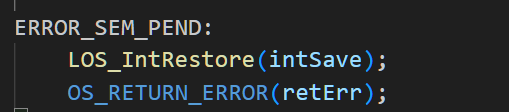
那么我们为什么要检查任务调度锁定呢？

我猜原因是为了有效防止意外情况，比如死锁🤔：

我们可能有一个信号量s定义在堆中，然后A线程中有一个P-s操作，B线程中有个V-s操作，但是A中如果添加了任务调度锁定就会发生这样：

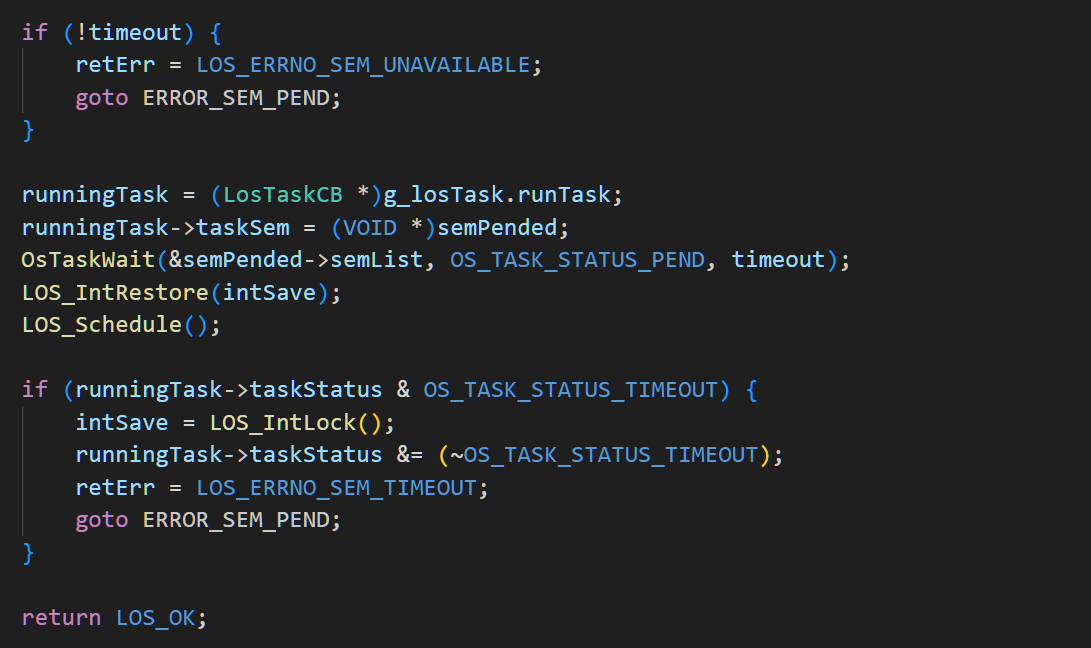
A -> P-s -need-> go B -> !go\_B(lock) -> ! V-s -> dead\_lock

尤其是在单核心操作系统中，如果被锁定了一个核心的任务切换，就无法切换别的线程下并发解锁信号量，而信号量大部分时候就是用于多线程进行同步的，这就导致一个信号量将引发死锁。



触发上面的情况时候，就会转到错误处理，解锁原子操作。

然后是逻辑部分：



超时检测，然后绑定任务与信号量。

runningTask = (LosTaskCB \*)g\_losTask.runTask; // 获取当前任务控制块

runningTask->taskSem = (VOID \*)semPended; // 记录任务等待的信号量

绑定逻辑是先获取当前任务控制块，然后将这个指针copy给runningTask下的Sem。

然后的然后，就会有个问题出来了，这个任务控制块哪来的😢。因为前边没讲过这一集，所以不知道也正常。

任务控制块（Task Control Block, TCB），夺莫高级的词汇🤨，但是实际上就是用来保存我们创建的某个任务的信息的，包括状态和上下文信息。

这个里面存储了很多信息，我简单举几个例子【因为现在主讲信号量】：

**stackPointer（栈指针 void\*）：**

这个指向任务当前的栈顶位置，任务切换时，CPU 寄存器状态保存在此处，保证下次寄存器再干这个活能记住自己干到哪了。

**taskStatus（任务状态 UINT16）：**

还记得上课教的 *就绪，运行，阻塞，挂起* 吗？🧐就是放在这个里面的。

**priority（优先级 UINT16）**

优先级高的先干

**topOfStack（栈顶地址 UINT32）**

任务栈的起始地址（高地址），初始化时将栈指针指向此处，然后向低处走。

**taskSem（持有的信号量 void\*）**

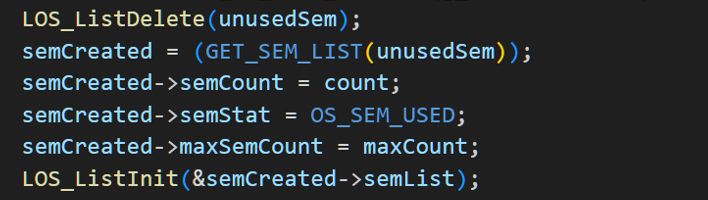
当前这个任务在等待或者持有的信号量

**taskID（任务ID UINT32）**

任务的身份证号。

**pendList（等待链表节点 LOS\_DL\_LIST）**

任务挂载到的某个资源的等待队列，还记得之前创建时候的semList吗？【下图】



当时没有讲为什么使用了Delete将信号量从未使用链表中开除之后又Init了一下，但是这里就可以看出原因了：用于复用给等待队列，需要等待的任务会放在后面。

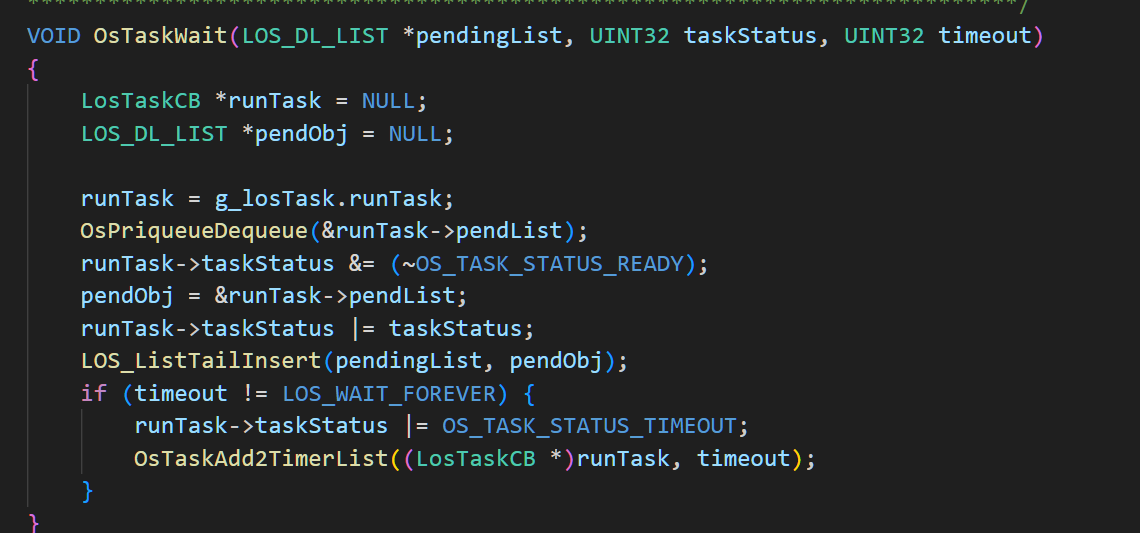
以上是任务块中的部分数据，对于解析信号量来说已经足够了。

下一句执行的是是：

OsTaskWait(&semPended->semList,OS\_TASK\_STATUS\_PEND,timeout);

这句话主要负责 **挂起任务到信号量等待队列**

来看一下函数内部是怎么操作的

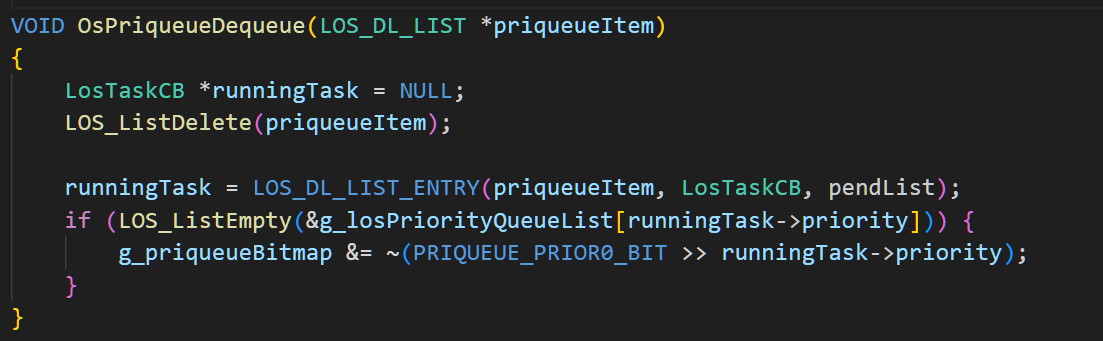


太好啦！是看起来就长得头疼的代码结构！写的不是很工整。

这段代码是先将任务控制块录入进来

* OsPriqueueDequeue(&runTask->pendList);

然后将这个任务从就绪队列中移除，就是正常的双向链表删除



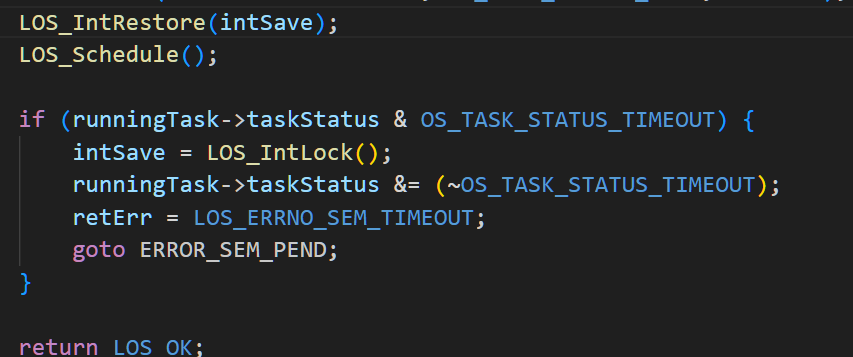
 runTask->taskStatus &= (~OS\_TASK\_STATUS\_READY); //清除状态

 pendObj = &runTask->pendList; //转移句柄

 runTask->taskStatus |= taskStatus; //设置新状态

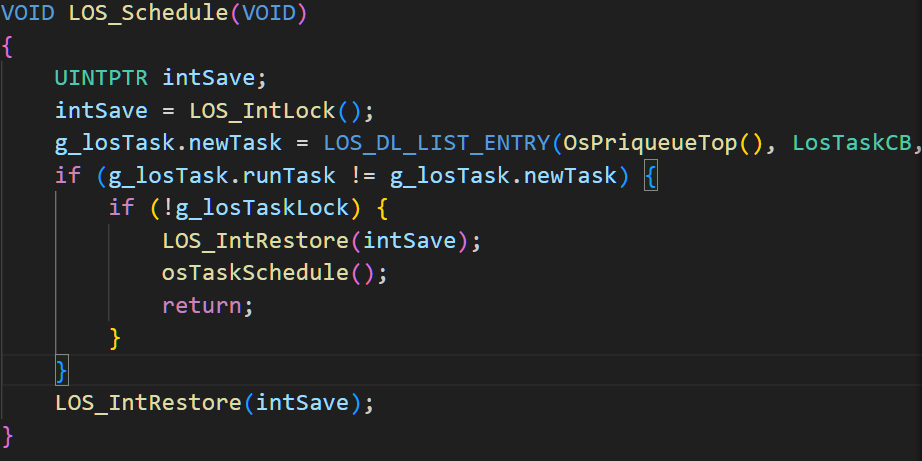
在配置完更新后的状态，将这个推入信号量不加入就绪队列队尾，而是选择加入当前信号量的等待队列，这样在就绪队列进行时间偏轮转时候就不会执行这个任务。

以上是oswait操作的介绍，我们还要继续往下看P操作的剩余部分是怎么执行的。



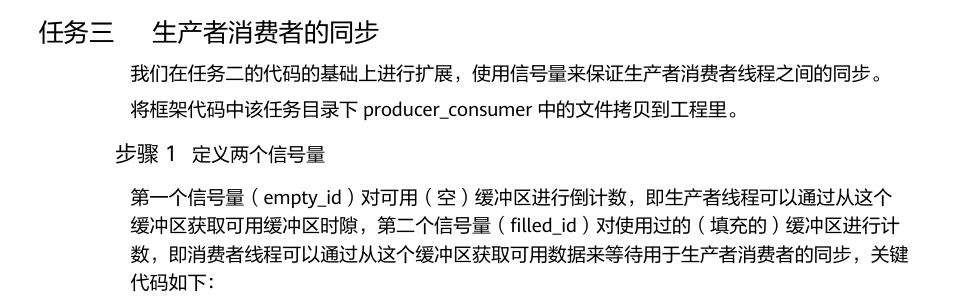
执行完成等待操作之后，先将原子操作结束，因为信号量还有任务转移已经做完了，即便中途切换也不会出现更多风险😋。

然后是LOS\_Schedule的调用，主动结束当前任务调用。



内部也是正常的先原子操作锁，然后将新任务从就绪队列中取出来。然后进行一个任务调度是否处于锁定状态的检测，如果没有被锁定就会触发先接触原子操作，然后使用os的一个系统函数。

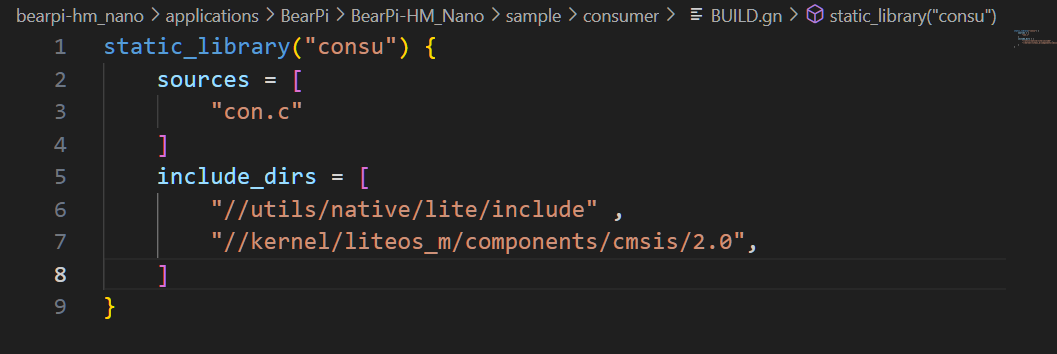
【操作】生产者与消费者同步模型



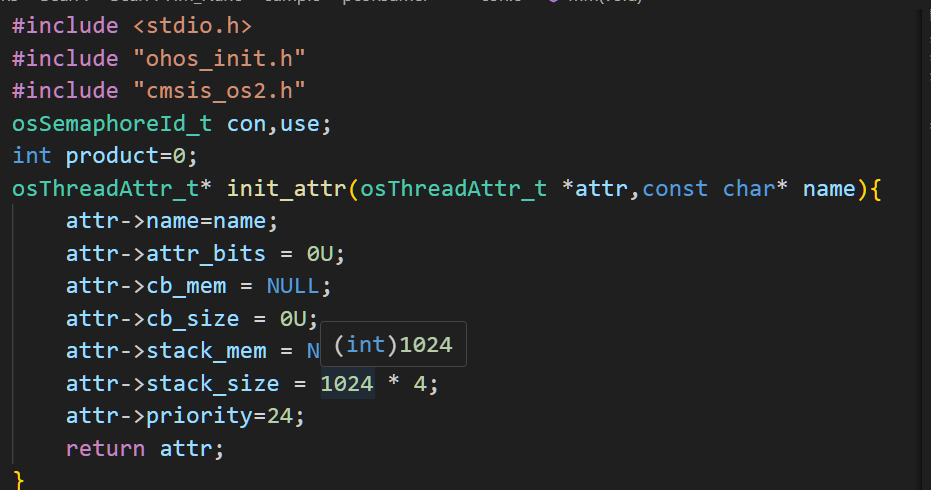
虽然有些比视频的节奏快，但是本来也不是什么复杂的东西，我们直接根据所学编写就好。**写到这时候我发现那个视频讲的一般，学生学完之后真的能回什么吗，我临时决定扒一下源代码，源码放在上面了。**

1. 创建新项目

创建一个新的编译对象，并引入编译需要的库函数

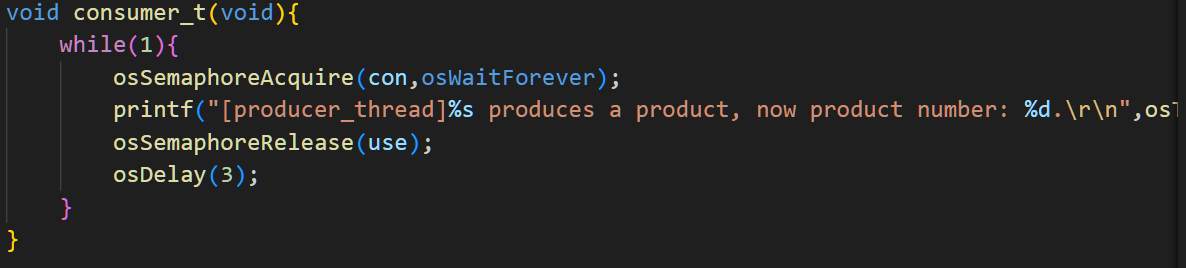


1. 创建初始化和全局变量

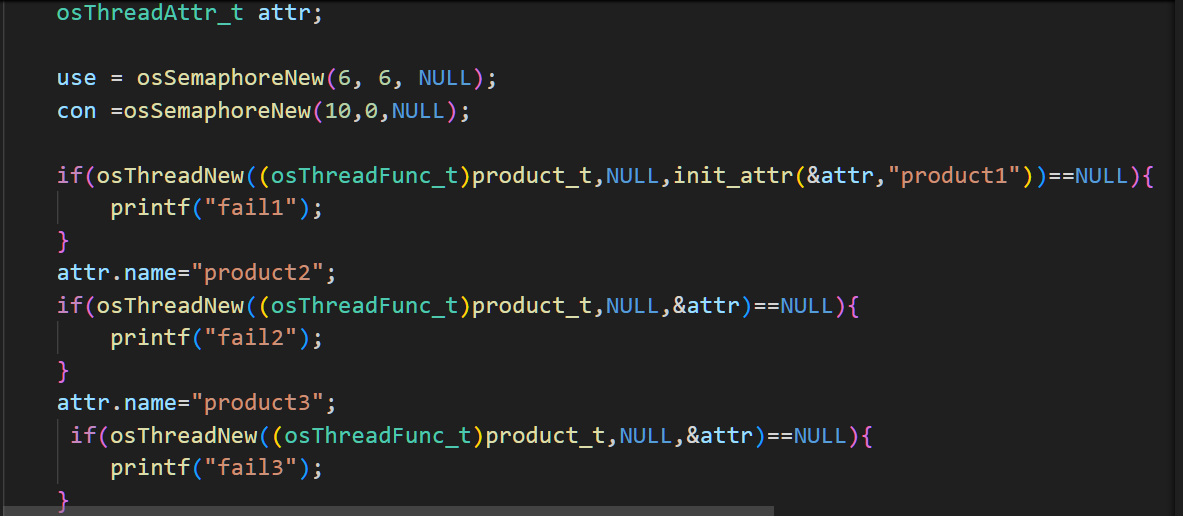




生产者线程，每次等待use可用，就是生产出来的产品有剩余空间可以放置，然后将产品数量++，讲当前线程进行delay才注意。



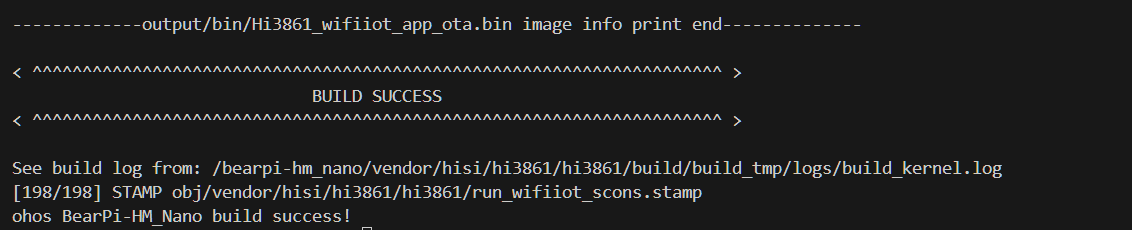
消费者线程，对生产者线程进行取反。

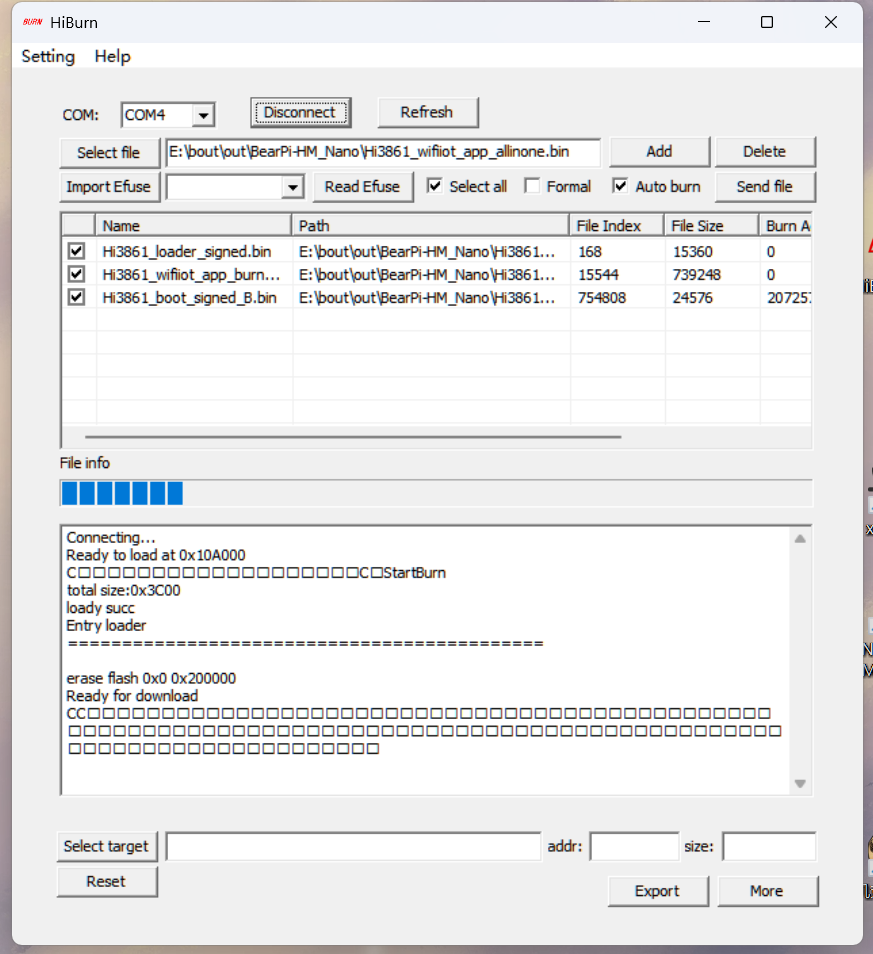


在主函数中进行信号量设定，仓库数量定为六个。

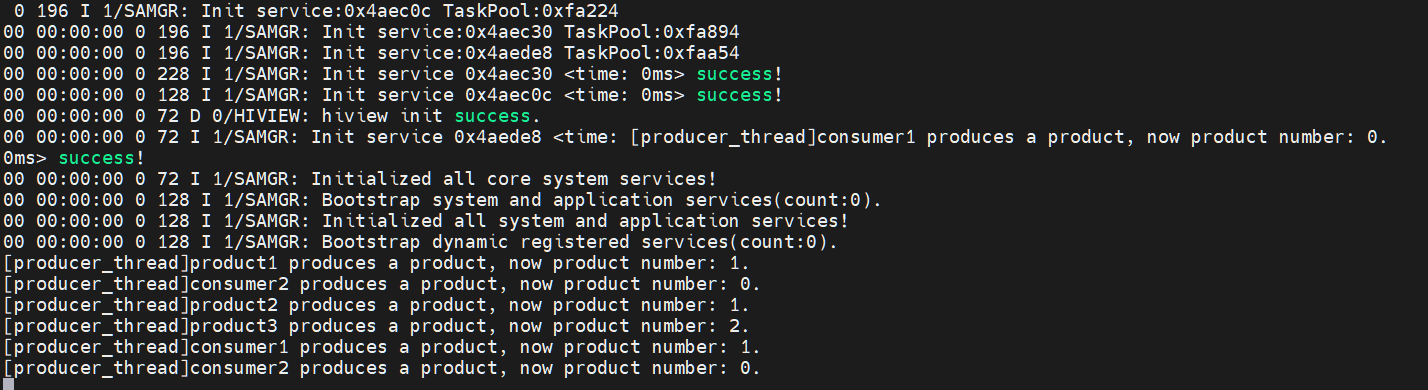
我们进行一下编译操作

* sh build.sh





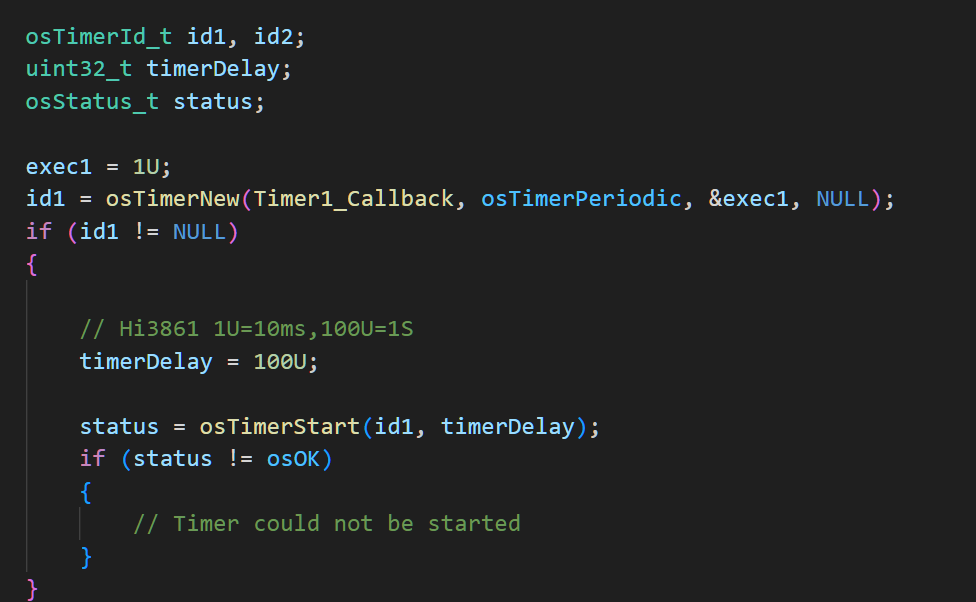
进行烧录，然后连接读取串口。



可以看到也是顺利同步了。但是我代码里面消费者输出忘记改成consumer\_thread了，但是无伤大雅。

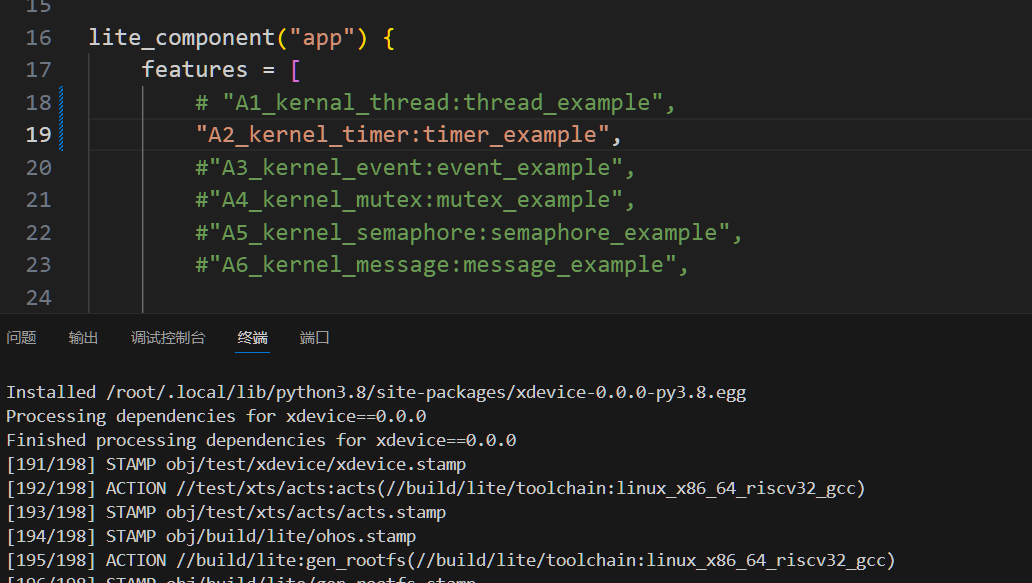
1. 定时-互斥锁-打印机模型

【视频】 定时器的简单使用

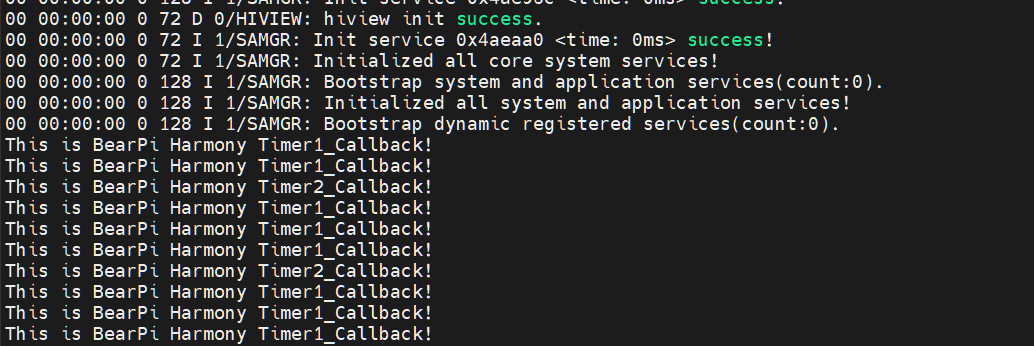


在主函数中创建定时任务，然后创建成功之后设定定时时间和开始时间。

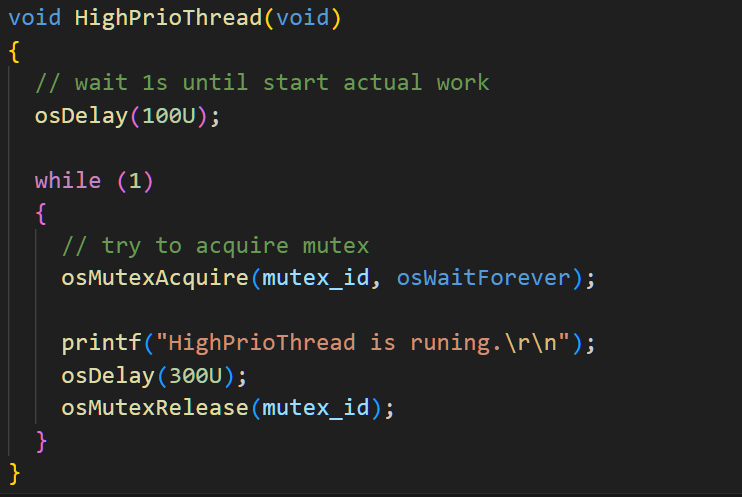
我们编译一下看看效果。



编译完成发现是一个完美的定时机制



【视频】互斥 锁

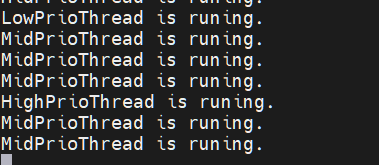


创建了三个线程，以低中高的优先级进行了区分，

高和中优先级会先挂起，这是就会由低优先级获取到锁，然后读取资源进行打印，打印之后会先进行挂起中优先级就会执行，最后是高优先级进行接手。

我们来烧录执行一下：

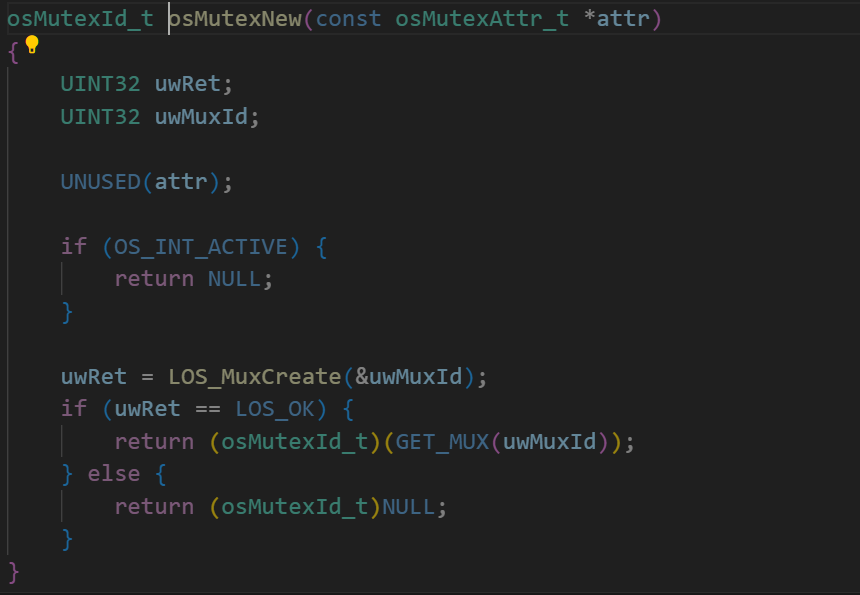




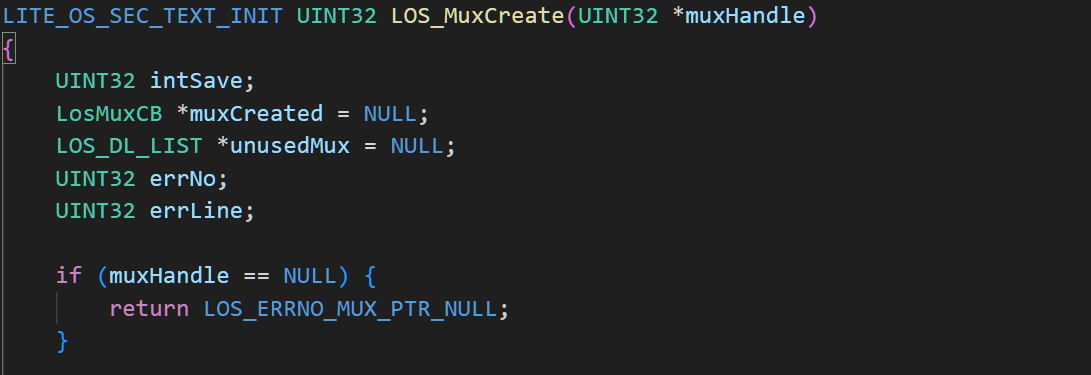
输出结果非常完善。

【解析】互斥锁

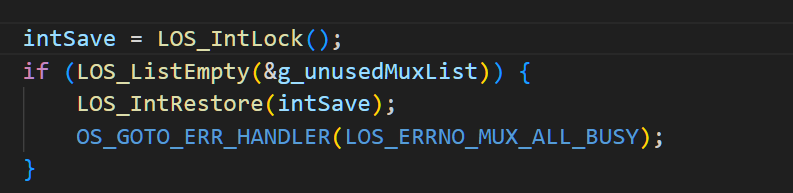
1. 创建



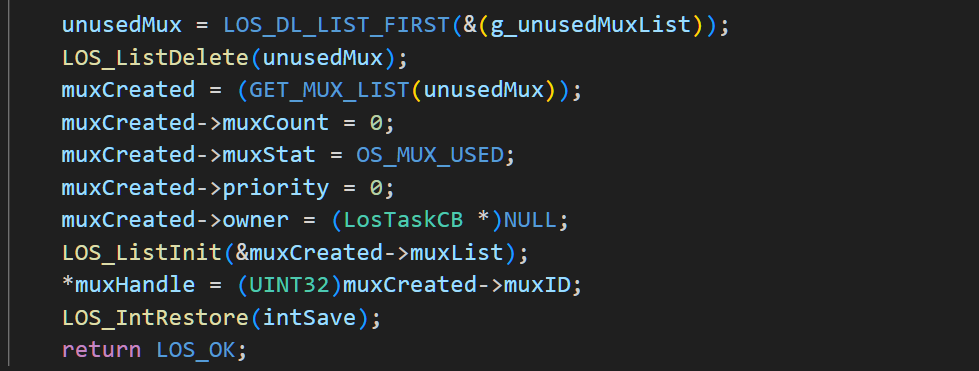
最外层的包装和错误处理。重点是那个os层的creat函数。🤨



进入内核看了，首先是最基础的初始化和防呆。



原子操作和检查锁空间中是否还有空闲锁，没有空闲的锁直接错误处理。

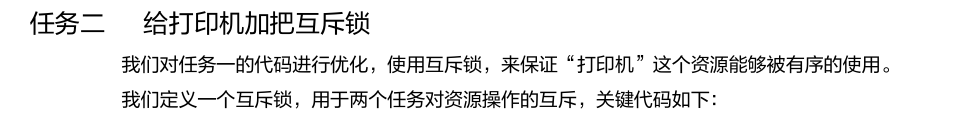


如果有的话取锁出来，我们会发现之后做的事情和信号量几乎如出一辙，可能是单核心就是好写🧐。大抵意思就是初始化一下，然后创建一个锁阻塞任务队列。

2,3. 上锁和解锁

经过我的检查发现了一个惨痛的教训，这两个操作几乎和信号量就是一窝出来的😨。这锁跟个二进制信号量似的，这人怎么这么坏啊。不解析这个，我们往下走。

【操作】打印机资源模型

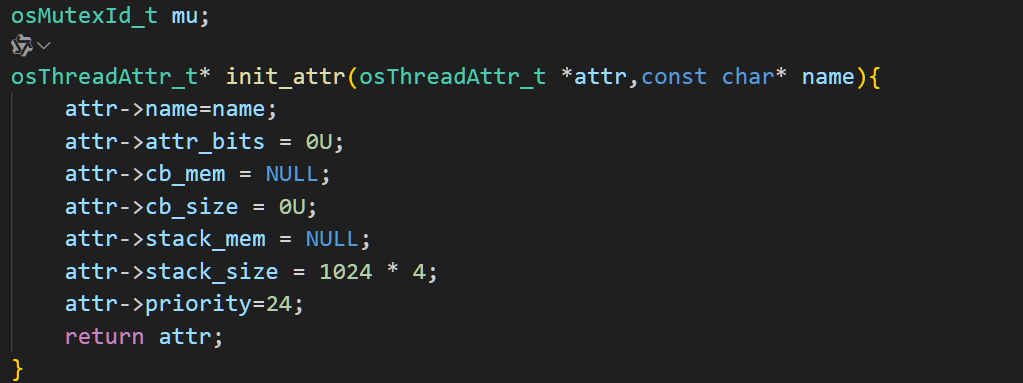


题目要求我们连续打印机要被占用连续打印五次这个消息，然后再连续五次打印那个消息。

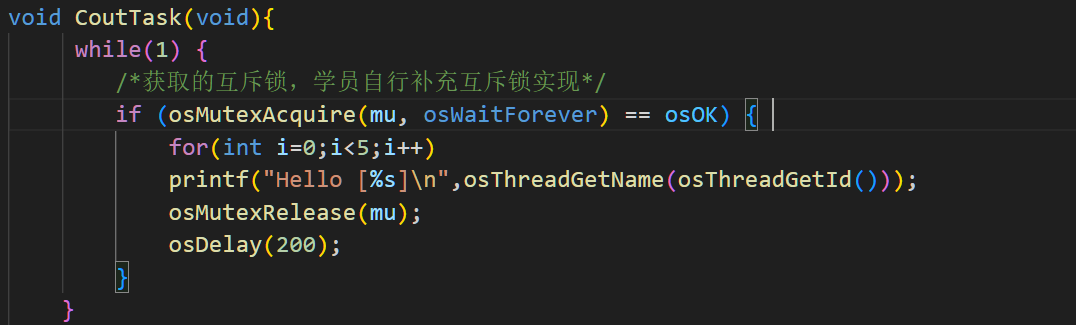
1. 创建gn编译文件



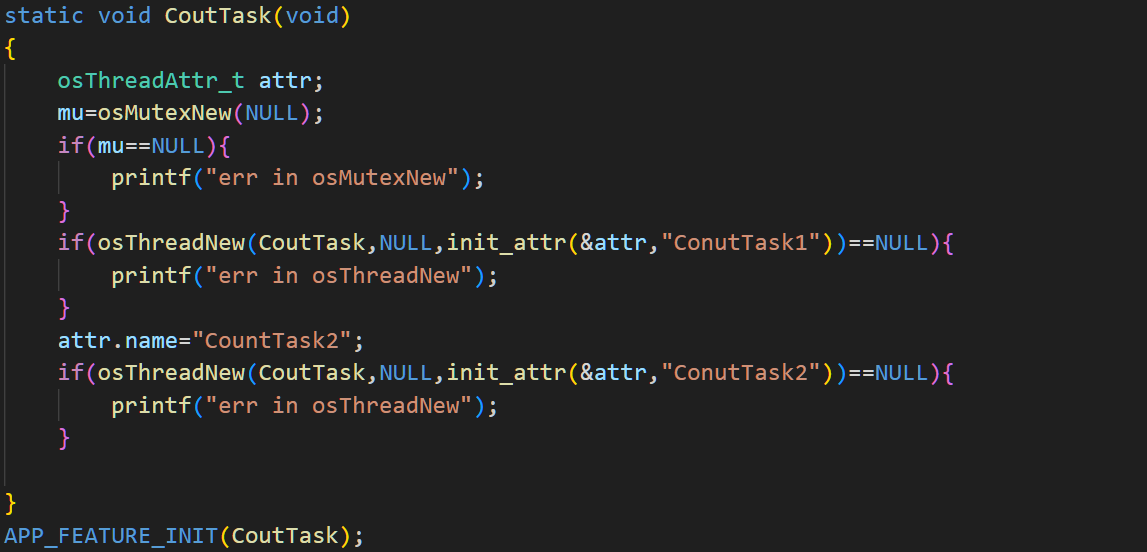
1. 初始化



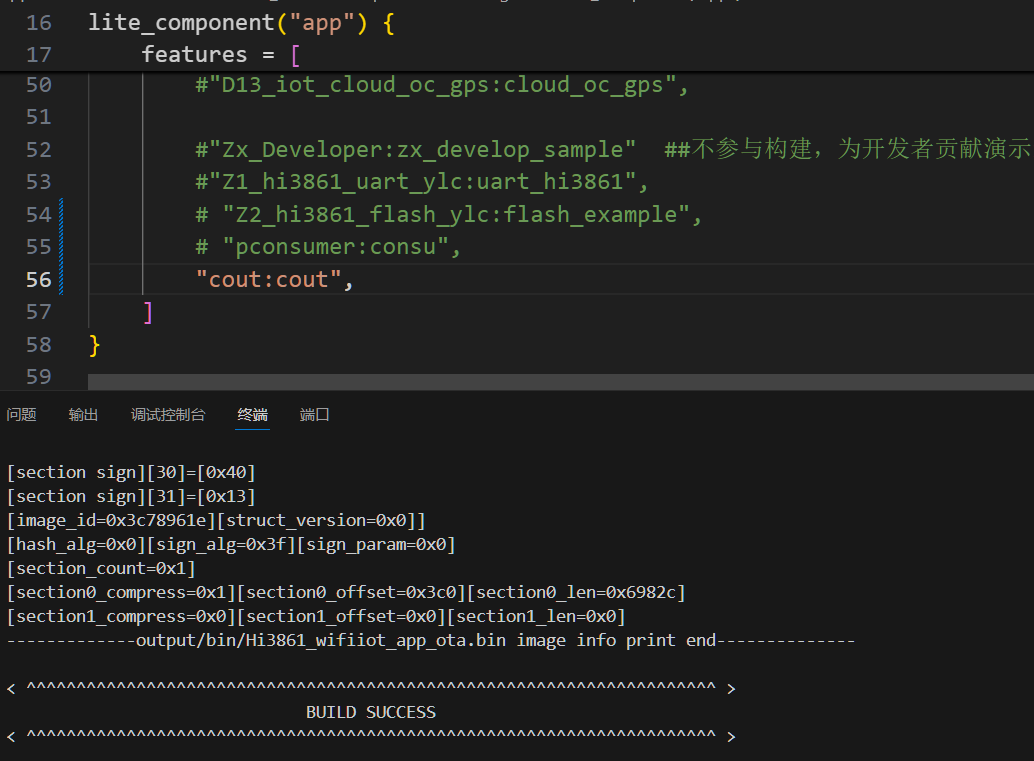
1. 打印进程

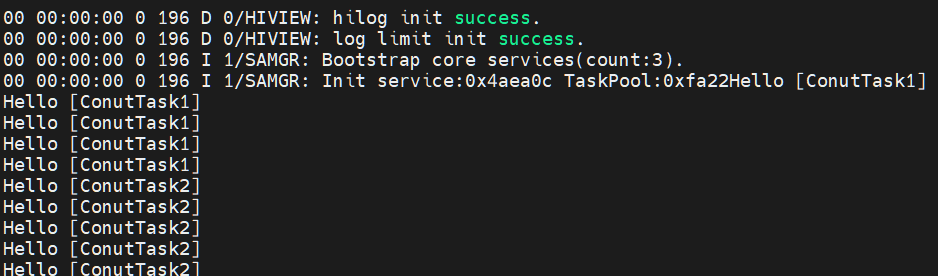


1. 主函数功能



编译





成功！！！！