文档介绍了子卡上电的大概逻辑，包括主控侧如何感知子卡状态，与子卡进行通信，以及子卡侧的软件启动和ibc消息的接收处理。中间涉及到的ibc板间通信和ros平台的socket收发属于公共基础模块，其详细实现没有进行说明。

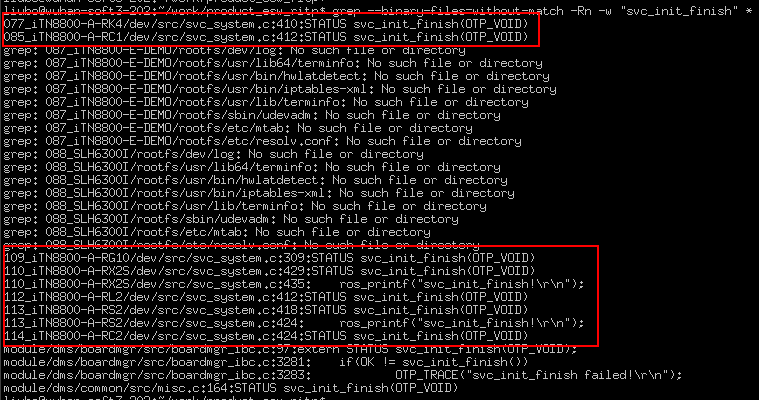
**子卡部分**

**板卡上电完成的通知函数svc\_init\_finish**

以RG10板卡的上电过程为例，板卡初始化完成后需要与NXU握手，通知NXU板卡上线初始化完成，子卡最终通过函数svc\_init\_finish将子卡的中断和ready标志位置位来给出硬件信号，然后NXU感知到子卡状态ready：



函数svc\_init\_finish的作用是用来响应NXU发送的ibc握手消息，一般需要跑应用软件的板卡都有这个函数的定义实现。在产品目录下可以搜索该函数：

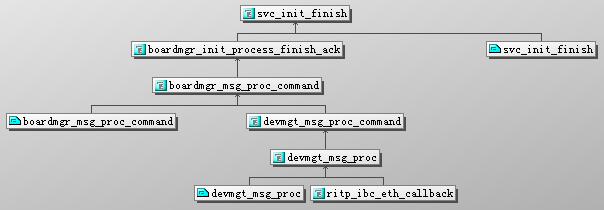


与product\_esw\_ritp目录下的各产品目录对比，只要是板卡类型的产品都有定义函数svc\_init\_finish：



所以，可以从svc\_init\_finish函数入手，梳理板卡的启动和通知过程。

往上跟踪函数的调用过程，可以跟踪到一个回调函数ritp\_ibc\_eth\_callback，整个调用过程中没有其他调用分支，就是回调函数ritp\_ibc\_eth\_callback被触发，然后一直调用到最终的svc\_init\_finish函数结束：



逐个调用查看，可以看到函数boardmgr\_msg\_proc\_command的实现内容，有一个命令类型BOARDMGR\_INIT\_NOTIFY：



接着查看函数boardmgr\_msg\_proc\_command的上两级调用函数devmgt\_msg\_proc，会发现触发回调函数ritp\_ibc\_eth\_callback的命令操作类型是IBC\_OPCODE\_COMMAND：



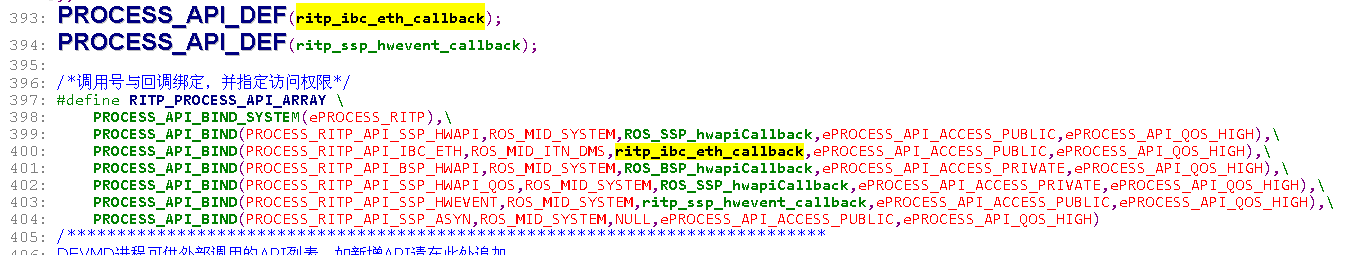
BOARDMGR\_INIT\_NOTIFY和IBC\_OPCODE\_COMMAND可以帮助梳理回调函数的触发过程，后面再介绍。

**回调函数ritp\_ibc\_eth\_callback**

先看回调函数ritp\_ibc\_eth\_callback的详细定义：

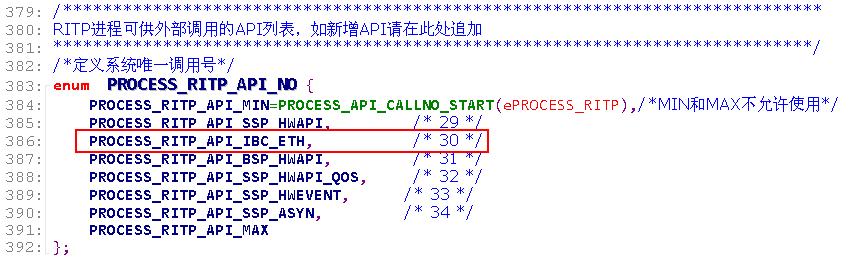


这里首先会想到搜索回调函数的所有使用到的地方，但是在product\_esw\_ritp产品目录下没有搜索到其他使用的地方，只有devmgt\_ibc\_core.c文件里回调函数的定义。然后，到ros平台代码中查找回调函数的使用：



在rosprocessapi.h文件中找到ritp\_ibc\_eth\_callback的使用，从宏定义的名字上看是将ritp\_ibc\_eth\_callback函数注册到了一个api消息的调用下，应该是api消息接收或发送时会触发ritp\_ibc\_eth\_callback回调。

继续查看PROCESS\_API\_BIND的第一个参数PROCESS\_RITP\_API\_IBC\_ETH的定义：

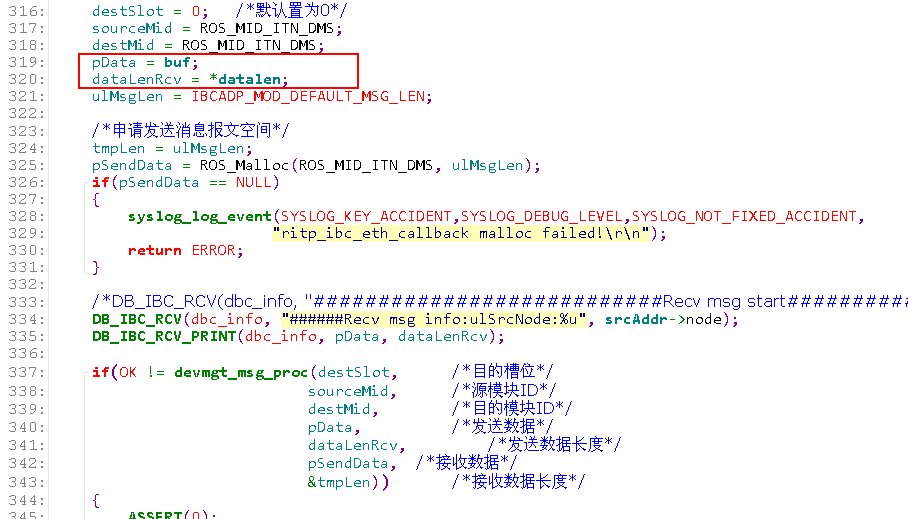


从注释上可以知道这个调用号应该就是用来处理回调函数的类似消息码的符号，而上一个截图中每一个调用号都在PROCESS\_API\_BIND里面有一个对应的回调函数，所以基本可以确定ritp\_ibc\_eth\_callback回调函数的触发就是由调用号PROCESS\_RITP\_API\_IBC\_ETH实现，板卡和主控的板间通信也由这个调用号来标识。

注意，这里我们需要梳理的是板卡和NXU主控对板卡上电的处理过程，不需要继续跟踪PROCESS\_API\_BIND以及RITP\_PROCESS\_API\_ARRAY的被使用情况，继续跟踪下去会发现是一整套进程间通信以及板间通信的实现逻辑，包括各种ros平台命令行向驱动ritp下发配置的接口实现，以及通过socket的通信实现过程，暂时没有必要详细了解。所以，只需要跟踪调用号PROCESS\_RITP\_API\_IBC\_ETH即可。

**子卡接收ibc消息**

前面介绍了子卡在ros平台注册了一个回调函数ritp\_ibc\_eth\_callback，当子卡的ros平台的socket通信模块收到主控发送的ibc消息后，经过对应的调用号PROCESS\_RITP\_API\_IBC\_ETH触发注册的回调函数，然后解析收到的主控消息，进行下一步处理：



其中，pData就是收到的主控发送的消息，申请的堆内存空间pSendData就是子卡准备返回给主控的消息结构。调用的设备管理消息处理函数devmgt\_msg\_proc开始解析收到的主控消息：



得到操作类型opcode，剩余的消息内容pDataLocal，以及准备返回给主控的消息内容指针pAckLocal，然后根据opcode进行分发处理：



主控发送的查询消息通过devmgt\_msg\_proc\_query函数处理，发送的握手消息通过devmgt\_msg\_proc\_command函数处理，最终触发子卡调用svc\_init\_finish。

不同opcode的不同处理函数的详细内容不做详细说明，因为这些处理函数都是定制功能，需要什么处理直接在这个函数里修改就行。比如，查询消息可以在devmgt\_msg\_proc\_query函数中增加不同的查询结果，原来只返回了设备类型、硬件版本、端口数量等信息的，可以增加返回其他如子卡状态等，只要返回的ibc消息长度和格式可以容纳。

**主控部分**

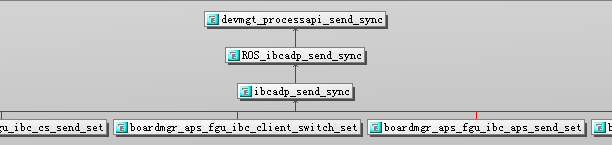
**主控触发板卡回调的ibc消息接口**

在产品目录中查找调用号PROCESS\_RITP\_API\_IBC\_ETH的使用情况：

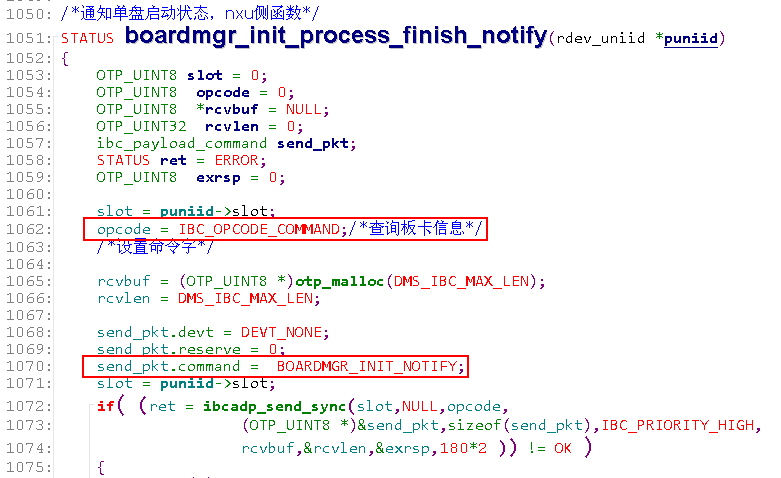
然后，发现该调用号使用的地方与回调函数ritp\_ibc\_eth\_callback的定义在同一个文件devmgt\_ibc\_core.c：



这里就基本确定NXU主控的ibc消息发送接口devmgt\_processapi\_send\_sync，以及板卡接收处理ibc消息的回调函数ritp\_ibc\_eth\_callback了。跟踪这个接口的调用关系：

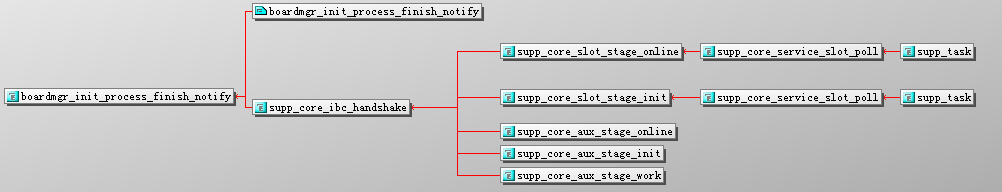


上上一级的函数ibcadp\_send\_sync的调用函数很多，这里前面提到的板卡上电完成的通知函数svc\_init\_finish的触发条件BOARDMGR\_INIT\_NOTIFY和IBC\_OPCODE\_COMMAND就起作用了，根据这两个条件来确定ibcadp\_send\_sync的上一级调用函数：



其实，在跟踪板卡上电完成的通知函数svc\_init\_finish时就已经通过BOARDMGR\_INIT\_NOTIFY找到了这个NXU侧的消息发送函数，也知道这个函数与调用svc\_init\_finish的boardmgr\_init\_process\_finish\_ack函数是一对收发接口，但是前面是想找到回调函数ritp\_ibc\_eth\_callback的触发逻辑，陷入ros平台的进程间通信和板间通信逻辑。

跟踪这个函数的调用关系：

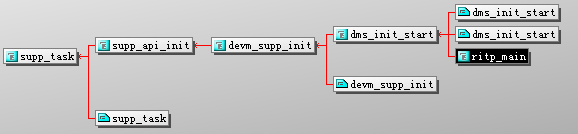


可以看到最终的调用函数supp\_core\_service\_slot\_poll，从函数名以及上一级的supp\_task调用函数来看是一个槽位轮询过程，每轮询到一个槽位最后都会向子卡发送一个supp\_core\_ibc\_handshake消息。

**主控轮询槽位状态**

**轮询任务主函数supp\_task**

前面跟踪到supp\_task函数轮询槽位时，最终会给子卡发送ibc消息，所以主控对子卡上电过程的处理应该就在supp\_task函数里面，supp\_task的调用关系：



在supp\_api\_init初始化函数中创建supp\_task任务，然后任务启动时先初始化设备管理需要的一些全局变量：



这for循环主要是初始化全局变量queueinfo和tmpInst。这两个变量的数据结构不同，一个是消息通信过程中使用的消息格式（queueinfo），一个是设备管理保存各个板卡状态和信息用的设备实例（tmpInst）：



这里跳过了主备两块NXU主控和电源风扇板卡，只初始化了业务板卡的全局变量，这是因为主控和电源风扇的初始化和轮询过程在另外的任务线程supp\_sam\_task和supp\_aux\_task函数中实现。

全局变量初始化后开始进入supp\_task线程的主要工作，轮询板卡：



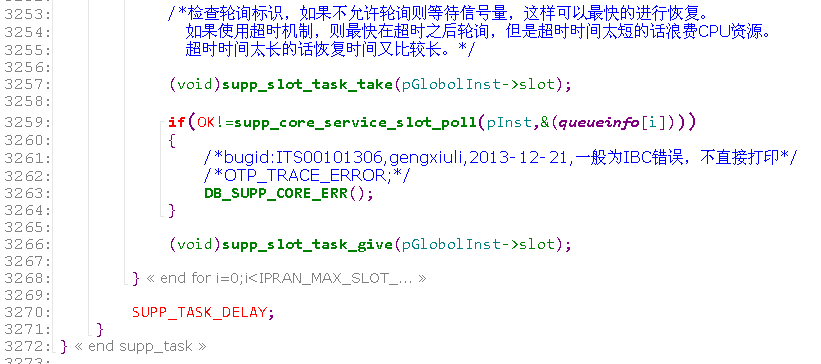
同样的，轮询任务主备两块NXU主控和电源风扇板卡，只轮询业务板卡。然后将supp\_get\_devInstance获取到的板卡实例信息（槽位号和管理总线类型）更新到全局变量queueinfo。另外取出全局变量tmpInst对象到局部变量pInst，方便后面使用：



中间主备同步部分暂时略过。当刚从supp\_get\_devInstance获取到的实例pGlobalInst与从全局变量tmpInst取到的实例pInst不一致时，更新临时实例信息tmpInst到全局实例对象pGlobalInst（通过supp\_core\_set\_instance\_info函数实现）：



实际上，任务刚启动第一次运行到这个地方时，临时实例tmpInst和全局实例pGlobalInst的内容都是刚初始化的，内容相同，需要执行下面的内容，向子卡获取信息后，这个地方的实例是否有更新的条件才会有效：



这里调用supp\_core\_service\_slot\_poll向子卡发送查询ibc消息时，第二个参数使用前面提到的全局变量queueinfo，用作ibc消息格式，子卡返回消息内容时直接填入全局变量queueinfo。

**轮询函数supp\_core\_service\_slot\_poll**

函数首先查询子卡在线状态和单盘上电状态，pmsg申请的堆内存用来保存子卡响应的消息内容：

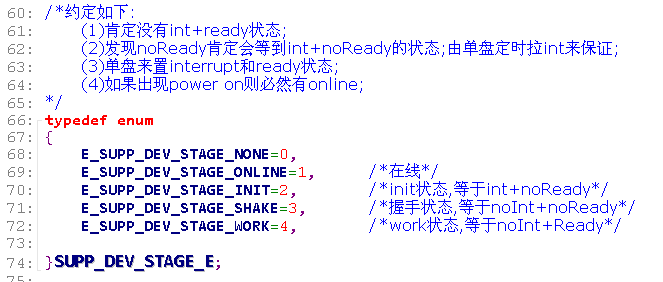


在线状态和上电状态通过主控的fpga获取，直接读取fpga定义的对应寄存器就行。为了尽量保证读取到的在线状态寄存器和上电状态寄存器准确，这两个函数在一定时间内采取反复读取，比对读值来校验最终状态值。

板卡状态机维护：



设备管理给出板卡的状态有五种：

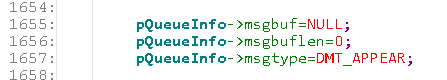


状态机按启动顺序依次为：刚上电NONE、上电后还没查到初始化ONLINE、有中断产生正在初始化INIT、初始化完成开始握手SHAKE和最终启动完成处于工作状态WORK。

**状态机维护**

**STAGE\_NONE**

板卡刚上电时状态肯定是NONE状态，如果之前读到板卡已经在线，那么函数supp\_core\_slot\_stage\_none更新全局变量queueinfo：



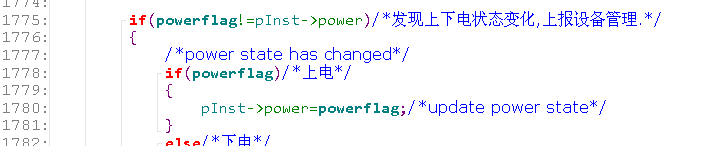
然后通知devm模块板卡插入事件，并更新pInst实例的stage状态：



**STAGE\_ONLINE**

轮询任务第一次跑完后，由supp\_core\_slot\_stage\_none函数处理，将pInst->stage状态更新到ONLINE状态，所以当主任务supp\_task再一次轮询到这个板卡时，轮询函数supp\_core\_service\_slot\_poll传入的实例状态pInst->stage已经是ONLINE状态。

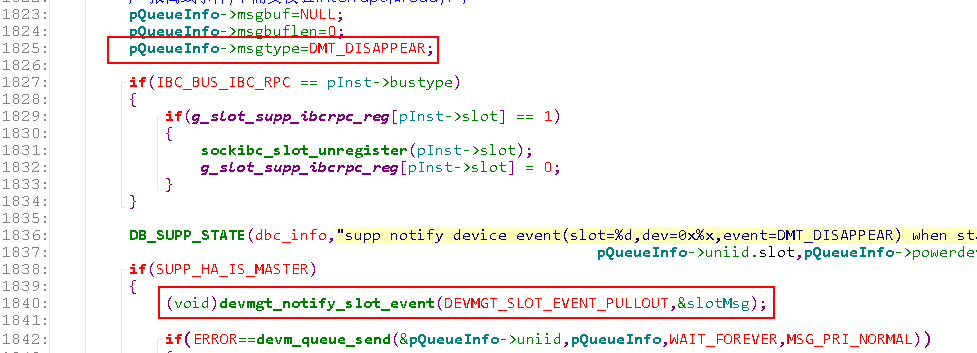
如果板卡在线，且处于上电状态，那么更新pInst的上电状态pInst->power，前面supp\_core\_slot\_stage\_none没有更新power标志位，这里更新：



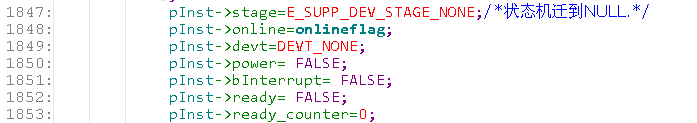
如果板卡在线，并且是由上电状态变为下电状态，那么更新全局变量queueinfo，向该板卡的消息队列发送下电消息，同时更新实例pInst的上电状态power，但是stage状态不用更新，仍然保持ONLINE：



如果板卡不在线，那么需要发送板卡不在位消息，并上报板卡拔出事件到devm模块：



同时，需要更新实例pInst的online和stage标志位到NONE：



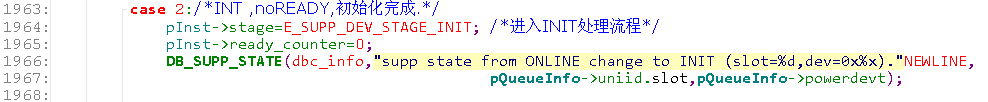
如果板卡在线，并且是上电状态的，继续进行后续的查询：



通过fpga提供的中断信号和ready标志来获取板卡当前状态，如果没有中断，有ready标志，那说明板卡处于STATE\_WORK状态，那么需要向板卡发送一个ibc握手消息，并将pInst->stage状态迁移到STAGE\_WORK：



如果是有中断，但是没有ready，那么说明板卡在初始化，更新实例pInst的stage状态到STAGE\_INIT即可：



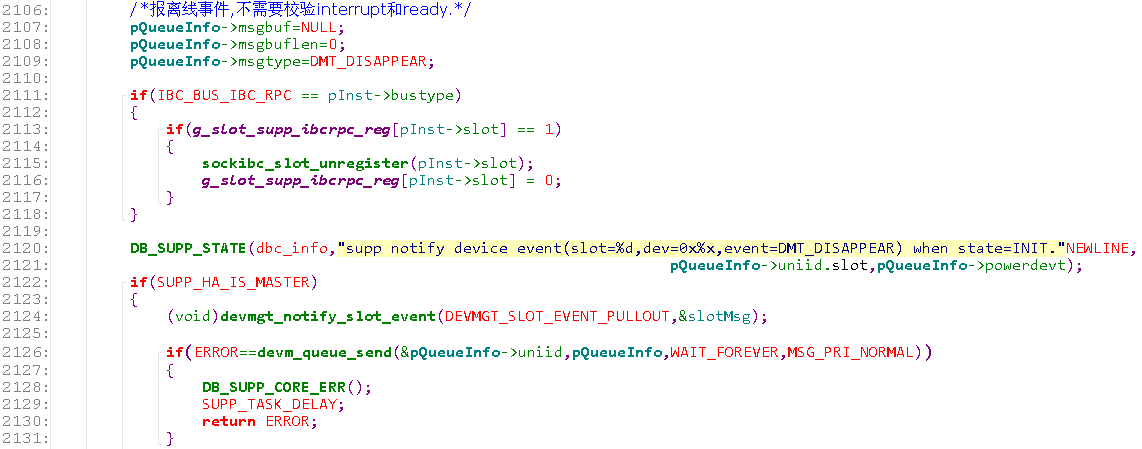
**STAGE\_INIT**

如果板卡在线，上电状态发生变化时，更新queueinfo，发送掉电消息到该板卡的消息队列，同时更新实例pInst的stage到STAGE\_ONLINE，下一次轮询会在supp\_core\_slot\_stage\_online函数中继续更新stage状态到STAGE\_NONE：

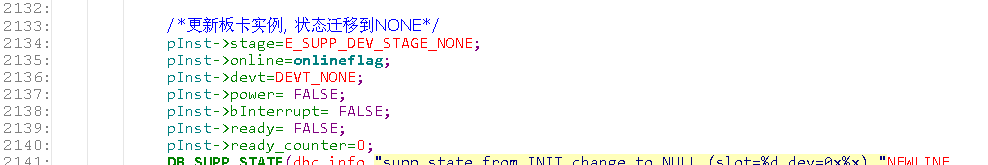


程序能进到这个地方说明上电状态之前应该是powerup的，不会出现之前是powerdown，然后这里变成powerup，一定是前面的STAGE\_ONLINE已经更新过上电状态的。

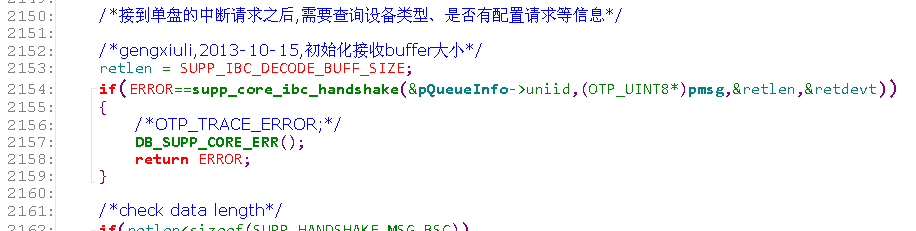
如果板卡不在线，与supp\_core\_slot\_stage\_online函数中的处理一样，需要发送板卡不在位消息，并上报板卡拔出事件到devm模块：



同时，需要更新实例pInst的online和stage标志位到NONE：



如果前面上电状态正常，这里直接给子卡发送ibc握手消息，主要目的是查询子卡的设备类型，而不是通知子卡，触发子卡的回调：

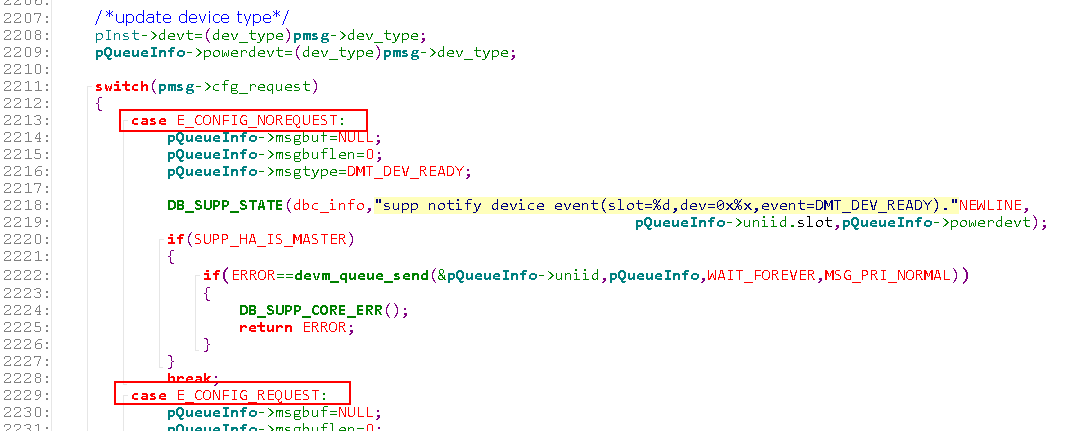


如果子卡不在线，处理类似supp\_core\_slot\_stage\_online，这里不再说明。

获取到子卡的设备类型后，如果发现设备类型与实例pInst保存的类型不一致，则判断子卡发生重启，那么更新子卡的queueinfo，发送消息到该子卡的消息队列，并发送子卡拔出的事件消息到devm模块，同时更新实例pInst的stage状态到STAGE\_NONE：



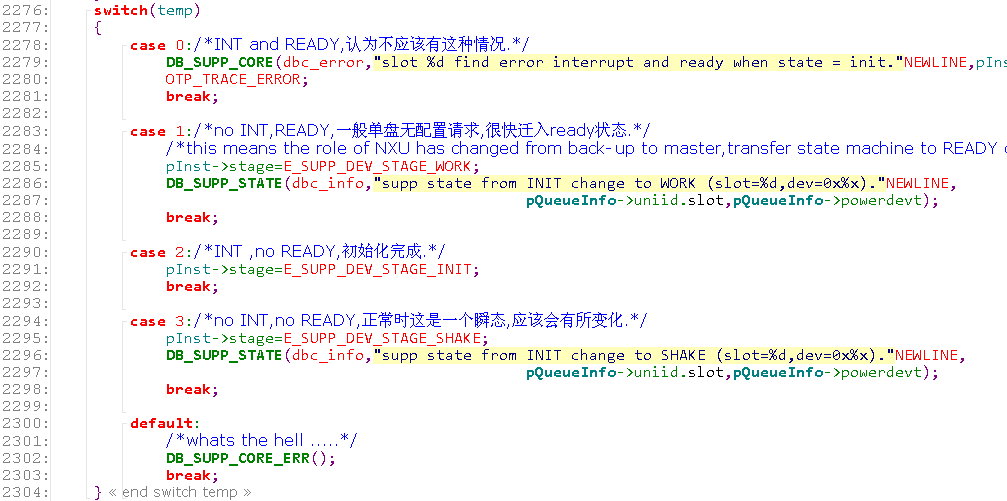
子卡状态正常的话，更新获取到的子卡类型到实例pInst和queueinfo，并且处理子卡返回的cfg\_request标志（当前子卡默认info\_get消息返回norequest）：



前面处理完成后，与supp\_core\_slot\_stage\_online中一样，检查中断状态和ready信号是否给出，这样来判断子卡当前的状态：



获取到中断和ready信号后，直接更新实例pInst的stage状态即可：



**STAGE\_SHAKE**

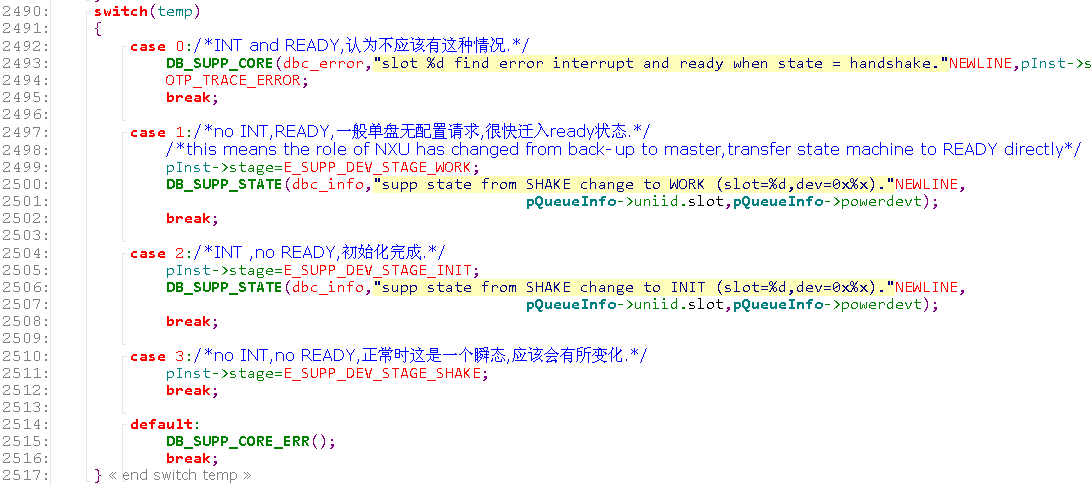
握手状态，通过中断和ready信号来判断的话，就是没有中断，但是也没有ready，说明子卡初始化完成了，但是还差一点点步骤，是子卡启动过程中的一个瞬态。

子卡在线和不在线的处理类似supp\_core\_slot\_stage\_online，这里不再说明。

子卡在线且上电正常时，直接读取中断和ready信号，与supp\_core\_slot\_stage\_online一样，不像supp\_core\_slot\_stage\_init还要先发送ibc消息获取子卡设备类型判断子卡是否重启：



与supp\_core\_slot\_stage\_init一样直接更新实例pInst的stage状态即可：



**STAGE\_WORK**

work状态，通过中断和ready信号来判断的话，就是没有中断，ready信号正常，说明子卡初始化完成了，所有配置流程都完成，进入工作状态了。

supp\_core\_slot\_stage\_work函数实现中，开始部分的子卡在线状态和上电状态的处理与supp\_core\_slot\_stage\_shake一样，中间直接读取中断和ready信号也一样，不用先从子卡获取设备类型。但是读取中断和ready信号后的处理与前面状态的函数不一样：



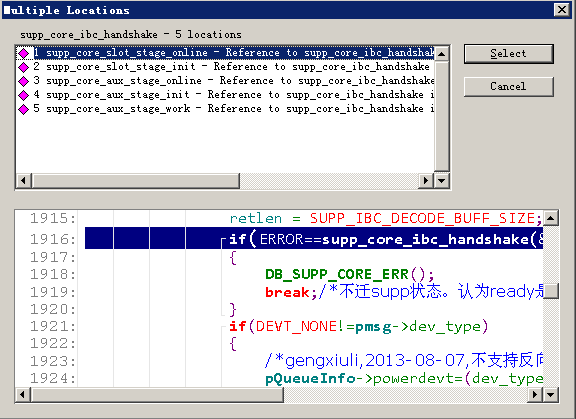
对没有中断，也没有ready信号的状态，这里是判断子卡发生过插拔或者复位，所以要更新queueinfo，发送消息到子卡的消息队列，以及发送子卡拔出事件消息到devm模块，同时更新实例pInst的stage状态到STAGE\_NONE。然后下一次轮询到这个子卡时，再由supp\_core\_service\_slot\_poll的状态机维护逻辑迁移stage状态。

**主控发送ibc消息**

主控在supp\_task和supp\_aux\_task轮询任务中会根据子卡状态选择发送ibc消息到子卡：



查看这个接口的调用关系，可以看到有slot和aux都有调用到，aux是多主控之间的处理，可以先不用看，只看slot的调用就行，也就是supp\_task轮询所有业务槽位的部分：



在板卡状态机处理函数supp\_core\_slot\_stage\_online和supp\_core\_slot\_stage\_init中会调用到这个ibc消息接口。

第二个finish\_notify通知函数的实现：



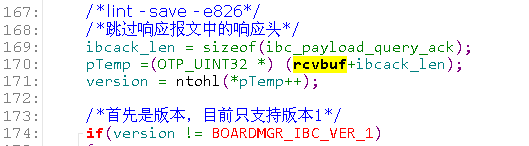
这里看到发送的消息内容send\_pkt，以及opcode。其中，opcode是IBC\_OPCODE\_COMMAND，消息内容send\_pkt的命令类型是BOARDMGR\_INIT\_NOTIFY。然后通过ibc的同步消息发送接口发送到指定槽位的子卡，同步等待子卡的返回消息rcvbuf。

根据opcode和send\_pkt的command类型，可以在子卡侧接收ibc的回调函数中找到对应的执行分支。

supp\_core\_hand\_shake函数的第一个调用是向子卡发送板卡查询消息：

其中，opcode是IBC\_OPCODE\_QUERY，发送的消息内容send\_pkt的命令类型command是BOARDMGR\_INFO\_GET，根据opcode和send\_pkt的command类型，可以在子卡侧接收ibc的回调函数中找到对应的执行分支。

同步消息正常发送后，子卡会响应查询消息，返回相应的查询内容，然后boardmgr\_dev\_basic\_info\_request函数后面的逻辑开始解析子卡返回的消息：



本函数目前只获取了子卡的设备类型，也就是消息中的dev\_type字段，但是整个返回的消息内容还是通过pmsg指给了函数的传入指针参数prcvbuf，后续有需要获取别的字段可以进行扩展解析：

