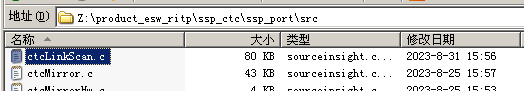
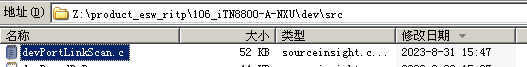
**link状态的入口文件**

当前，多数设备的接口link状态处理代码入口都在ctcLinkScan.c文件中实现：



但是，iTN8800-A设备的接口link状态除外。iTN8800-A设备的接口link状态在devPortLinkScan.c文件中轮询处理：

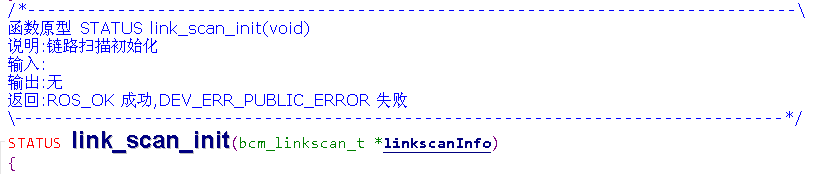


驱动层的接口震荡抑制逻辑也在ctcLinkScan.c文件中实现，但是震荡抑制功能有单独的文件做了模块化：

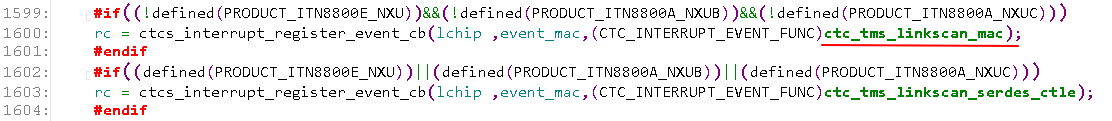


**link处理文件ctcLinkScan.c**

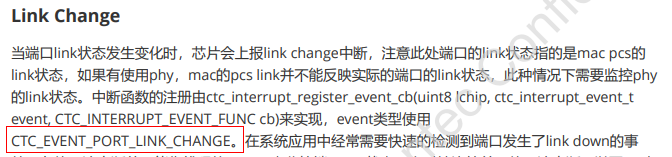
在查找对应设备的link状态处理接口时从初始化函数开始查找：



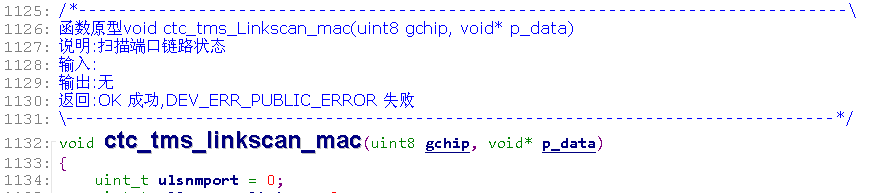
该函数根据不同设备初始化了接口link状态的处理方式。比如除了iTN8800之外的其他设备都是在sdk注册一个link中断函数，通过芯片上报中断的方式来处理接口的link状态变化：



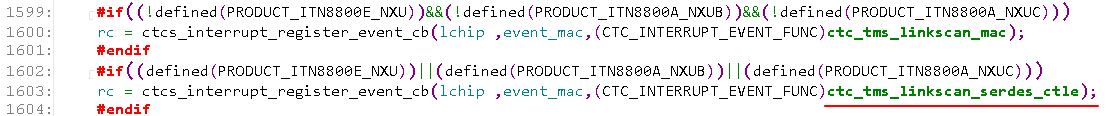
其中，注册的中断事件类型为CTC\_EVENT\_PORT\_LINK\_CHANGE：



当sdk有link中断产生时，中断通过sdk上报，进入注册的回调处理函数：



然后，在这个中断处理函数中对接口link状态进行处理。下图中，下面的iTN8800设备注册的中断处理函数与上面的不同：

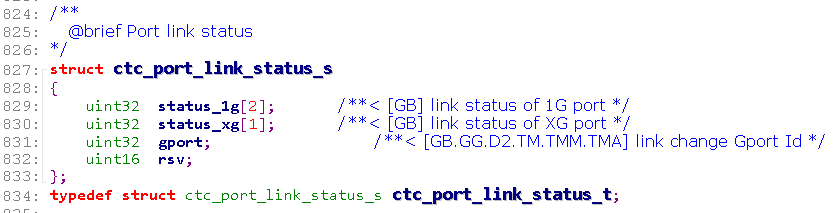


iTN8800设备这里注册的函数作用只是在接口产生link变化后，对接口的部分属性进行设置，对link状态进行处理的逻辑是在devPortLinkScan.c文件。

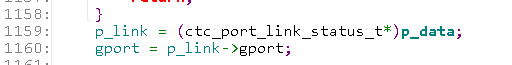
**向芯片注册的中断处理函数ctc\_tms\_linkscan\_mac**

sdk检测到接口状态变化，上报中断，然后注册的中断处理函数ctc\_tms\_linkscan\_mac被触发。下面按函数内顺序处理步骤依次截图说明。

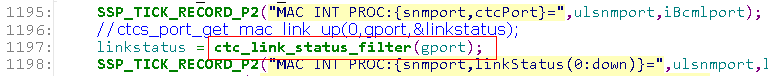
sdk触发该回调函数时产生中断的接口通过下面的数据结构传入：



回调函数里面现在是只用到了传入的端口号gport：



对于接口当前的link状态通过mac-link属性在回调函数里获取：



读取接口link状态的函数如下：



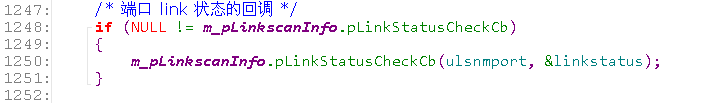
获取接口link状态时，为了避免接口震荡，导致读到的状态不稳定，这个函数在8ms内多次读取，最终返回一个稳定的link状态。

获取到接口的link状态后，开始处理相关操作，如果当前接口是linkdown的，那么需要快速通知涉及到保护倒换的功能模块，直接将接口linkdown事件通知到各个模块，尽量缩短倒换时间：

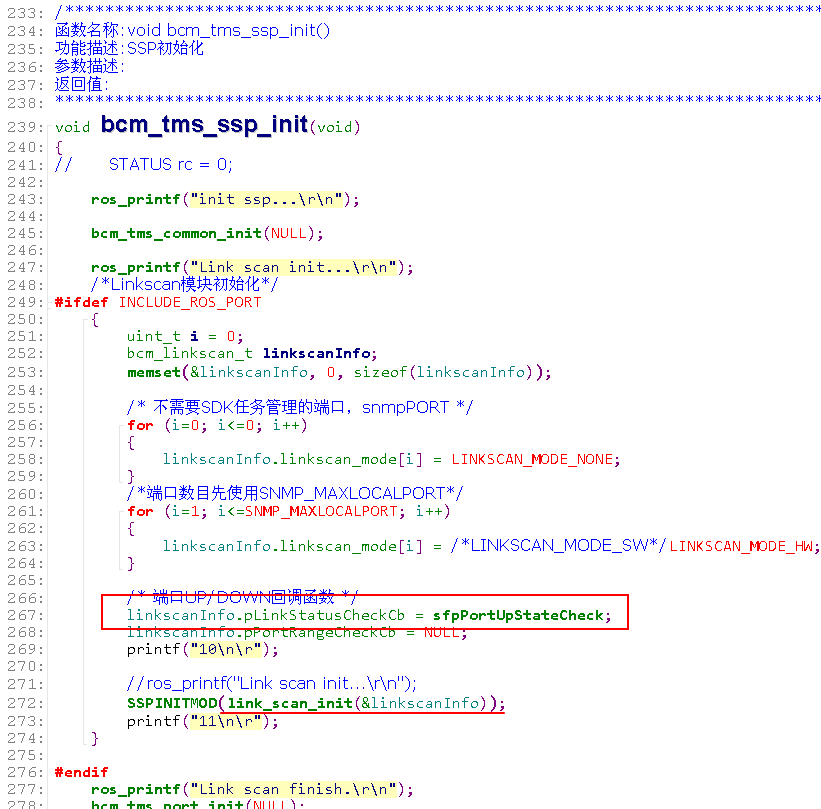


目前涉及到需要快速通知来进行倒换的模块如上，有聚合组的倒换、8031线性保护、8032环网保护、端口备份，以及mpls的保护倒换等功能。这些模块只需要linkdown事件的快速通知来保证倒换时间满足电信50ms要求，down->up的linkup事件正常由驱动上报到平台接口管理模块，然后由接口管理模块通知到各个功能模块，然后进行回切动作。

下一步骤是获取接口cpld/fpga的寄存器los状态：



这个回调函数在接口link扫描初始化函数之前注册，在各个产品的dev/src/sspInit.c文件中初始化：



该寄存器los状态获取函数在各个产品的dev/src/mdev.c函数中实现。

在已经通过sdk的mac-link接口获取接口的link状态之后，再一次获取接口的los状态，目的是避免一些sfp模块插上后无光纤连接也up的情况，由cpld或者fpga等硬件状态来校验从sdk获取的接口状态是否正确。当然，这个函数也是根据不同设备区分，比如iTN8800-A设备中的定义：

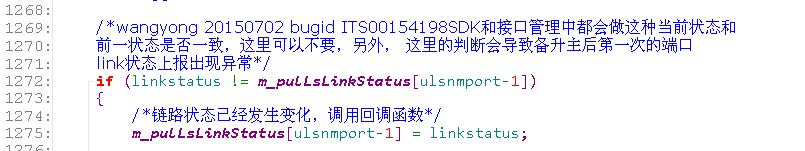


函数内容很简单，通过mdev\_sfp\_get\_sd函数获取接口的los状态，如果接口正常没有los，并且从sdk获取的也是linkup状态，那么说明接口的状态为linkup，如果两个状态不一致，那么接口状态为linkdown。

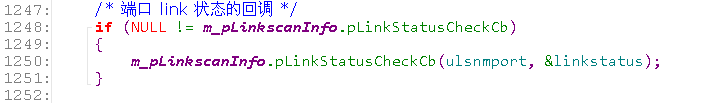
需要说明的是，有的设备插入电模块后，即使不插网线，sdk的mac-link返回的接口状态是linkup的，对这种情况也需要进行处理，比如721C设备的los状态函数：



通过sdk的mac-link和接口的los状态一起校验得到接口的link状态后，与软件上一次保存的接口link状态进行比较，如果不一致，则更新接口状态到全局变量*m\_pulLsLinkStatus*；如果一致，则不做其他处理：



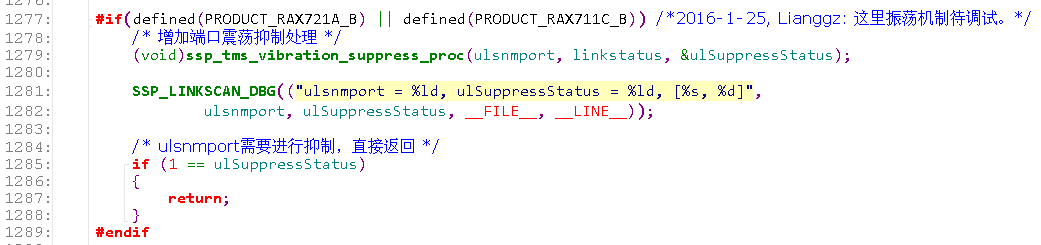
这里的**linkstatus**就是上面介绍过的回调函数，通过sdk的mac-link状态和接口los状态一起判断得到的接口状态：



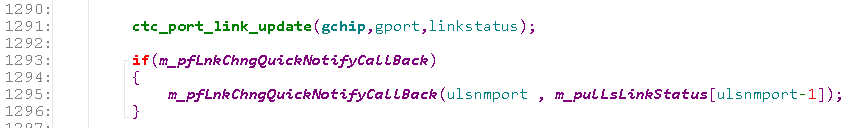
而全局变量*m\_pulLsLinkStatus*就是软件保存的各个接口的link状态，在link\_scan\_init中初始化：



更新软件保存link状态的全局变量后，对721A\_B和711C\_B设备添加接口震荡抑制步骤：



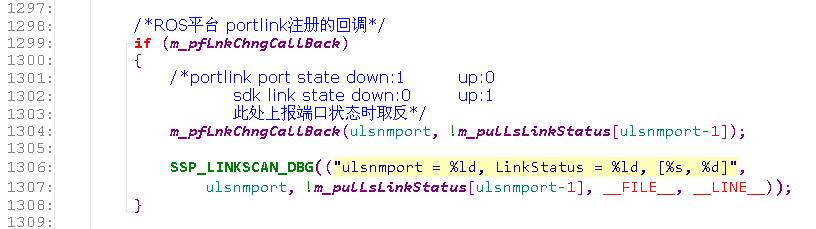
其他设备跳过继续后续操作，设置接口属性：



如果接口有phy芯片，那么接口link变化后要设置双工模式和速率，如果不是则在linkdown时设置port-en非使能，在linkup时设置port-en使能。

紧跟着的快速通知的回调函数调用，目前代码里面是没有注册相关回调的，可以忽略。

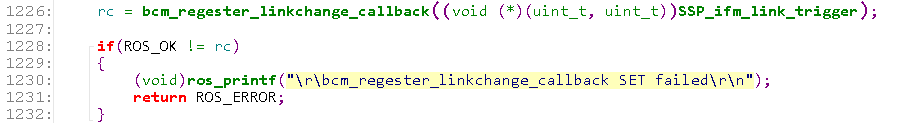
快速通知的回调没有地方使用，而向ROS平台上报link变化的回调函数是初始化注册过的：



这个回调函数**m\_pfLnkChngCallBack**将link状态变化上报给ROS平台，然后会在串口打印一行类似下面的通知，显示端口状态发生变化：



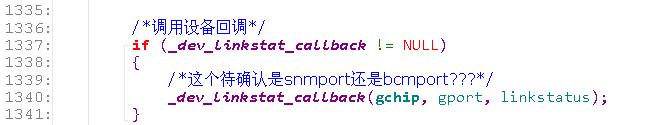
回调函数**m\_pfLnkChngCallBack**在sspIfmEvent.c文件的SSP\_Ifm\_event\_init函数中注册：



对于这个回调如何处理，最终上报平台后面介绍，回到ctc\_tms\_linkscan\_mac函数的处理过程。将接口的状态变化上送ROS平台后，如果设备支持单纤倒换功能，则触发单纤倒换的回调：



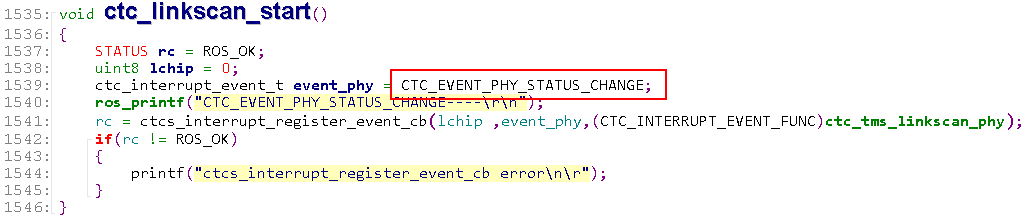
但是目前这个回调步骤没有注册使用，可以略过不看。同样的，最后的一个设备回调也暂时没有被使用到：



简单总结SDK接口link事件的中断处理函数ctc\_tms\_linkscan\_mac的主要步骤是，获取当前接口的状态，如果是linkdown事件，则首先将事件通过快速通知方法通知到各个linkdown事件敏感的功能模块，然后继续获取los状态一同决定接口当前的状态是linkup，还是linkdown，然后与软件保存各个接口状态的全局变量进行比较，如果不一致则说明接口状态发生变化，更新软件保存各个接口状态的全局变量，将变化的接口状态通过回调函数上送ROS平台。

**向芯片注册的中断处理函数ctc\_tms\_linkscan\_phy**

与ctc\_tms\_linkscan\_mac函数一样，也是通过sdk的中断注册接口注册的一个中断处理函数：

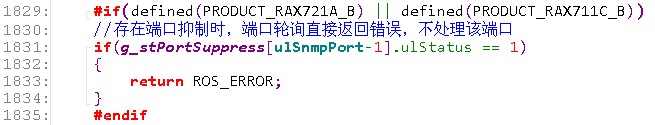


与ctc\_tms\_linkscan\_mac注册的CTC\_EVENT\_PORT\_LINK\_CHANGE类型不同，这里的类型是phy状态变化才会触发中断，然后上报这个中断处理函数ctc\_tms\_linkscan\_phy。当前的721C和iTN8800-A设备没有使用这种phy中断方式，可以暂时略过。

**bcm\_tms\_link\_status\_get**

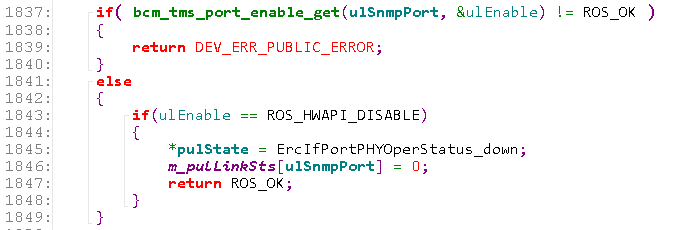
函数作用是获取接口当前的状态，根据状态来设置接口是否使能。

首先，判断接口是否是有震荡抑制：



当然，这个判断只针对部分设备，当前的721C和iTN8800-A设备都没有涉及。

然后，获取接口的使能状态：



这个函数在ssp\_ctc/ssp\_port/src/ctcPort.c中定义，区分接口是否有phy接入，如果有phy，那么需要phy的状态和芯片mac-en的状态一起来判断接口是否是使能的，如果没有phy，则直接通过芯片mac-en来判断。

接着获取接口当前的link状态：



bcm\_link\_status\_software\_get这个函数也是需要区分接口是否有phy，如果有phy，则返回接口的phy状态，如果没有，则返回芯片的mac-link状态。

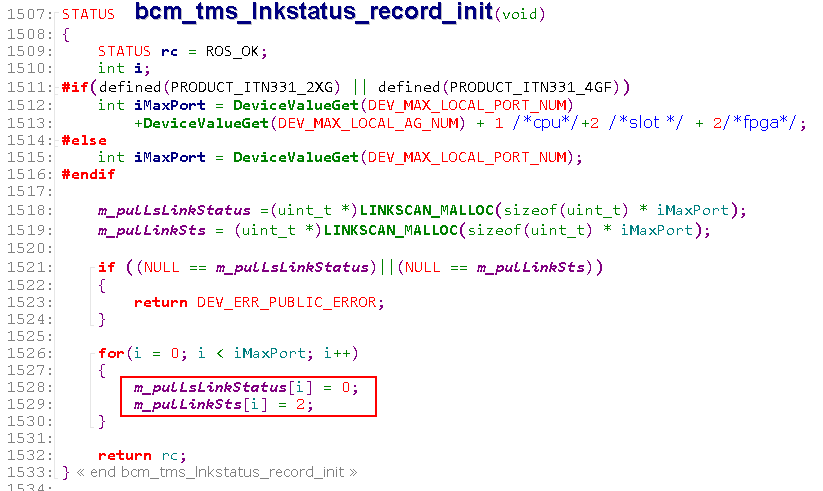
后面接着调用pLinkStatusCheckCb回调，这个回调函数在中断处理函数ctc\_tmc\_linkscan\_mac中介绍过，是读取接口los状态的，这里先通过函数bcm\_link\_status\_software\_get获取接口的一个link状态iUP，然后又通过los状态来更新iUP，说明在注册了接口los回调时更倾向于使用los状态来判断接口当前的link状态。

得到接口当前link状态后，与软件保存的接口状态比较：



如果不一致，则更新这个软件接口状态的全局变量*m\_pulLinkSts*，然后再获取接口当前的芯片port-en使能状态，如果跟link状态不一致，则刷新一下port-en配置。然后设置接口的点灯模式。

这个软件接口状态的全局变量*m\_pulLinkSts*，是在link\_scan\_init中通过函数bcm\_tms\_lnkstatus\_record\_init与全局变量*m\_pulLsLinkStatus*一起初始化的：



前面介绍过全局变量*m\_pulLsLinkStatus*是在中断处理函数中使用的，而这个*m\_pulLinkSts*只在bcm\_tms\_link\_status\_get中使用，可能是避免有设备没有注册芯片的中断处理函数，导致全局变量*m\_pulLsLinkStatus*没有实际使用，所以这里不直接使用*m\_pulLsLinkStatus*，而是用一个没有限制的新全局变量*m\_pulLinkSts*来保存接口link状态。但是，没有注册中断处理函数，*m\_pulLsLinkStatus*应该也可以在这里使用，不清楚为什么。

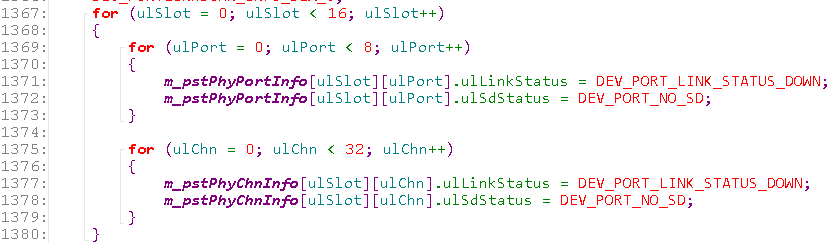
**link处理文件devPortLinkScan.c**

文件是iTN8800-A设备使用的接口状态处理功能，跟前面介绍的其他设备使用的ctcLinkScan.c文件主要依靠芯片中断上报来感知link事件是不同的两个处理逻辑。因为iTN8800-A是框式设备，除了有以太接口外，还有支持OTN功能带OUT接口的板卡，所以本文件主要是轮询加快速开销总线上报中断来感知接口link事件。

**初始化函数dev\_linkscan\_init**



函数初始化保存以太口link状态的全局变量m\_pstSnmpPortInfo。然后，初始化保存OTN板卡接口SF/SD状态的全局变量m\_pstPhyPortInfo，以及保存OTN板卡上各通道SF/SD状态的全局变量m\_pstPhyChnInfo：



目前保存OTN板卡各通道SF/SD状态的全局变量m\_pstPhyChnInfo暂时没有用到。然后，初始化创建消息队列，以及用来处理接口link事件的主任务：



其中，创建的主任务dev\_linkscan\_task的目的就是从创建的消息队列m\_ulLinkscanMsgId中接收接口的link事件消息。之后再创建一个定时器，用来定时扫描接口状态，发现有接口link发生变化，但是消息队列没有收到相关消息时，更新接口状态信息：



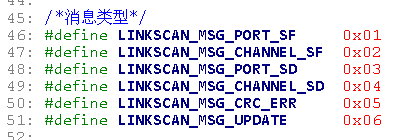
创建完定时器和定时器的处理函数后，使能快速开销功能，并打开定时器开始运行，定时器默认周期为20秒：



**消息队列接收函数dev\_linkscan\_task**



该任务创建后开始监听消息队列m\_ulLinkscanMsgId，然后根据接收到的消息的类型做后续处理。目前用到的消息类型有：



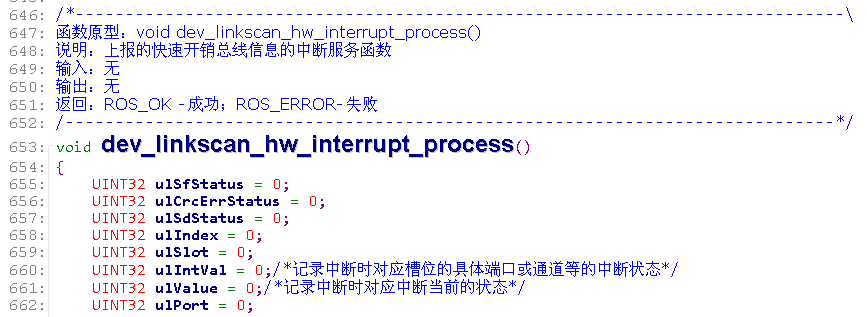
普通以太接口的link事件都是使用LINKSCAN\_MSG\_UPDATE。

**消息队列发送函数dev\_linkscan\_msg\_send**

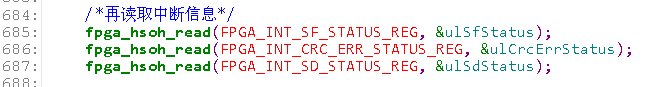


快速开销有中断产生后，最终会经由本函数将产生中断的槽位、端口和中断类型封装后送到消息队列m\_ulLinkscanMsgId，然后函数dev\_linkscan\_task从消息队列中取出该消息，根据消息类型做下一步处理。

**快速开销中断处理函数dev\_linkscan\_hw\_interrupt\_process**

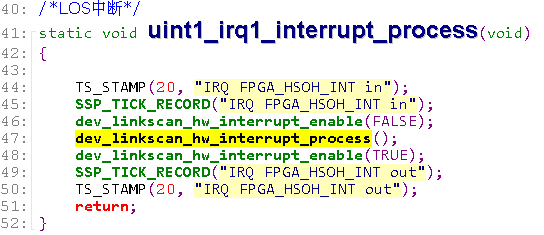


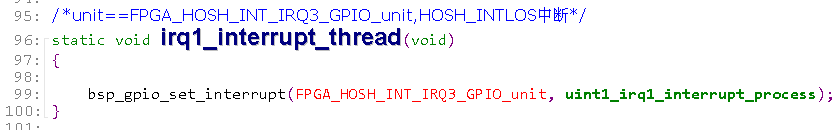
该函数作用就是快速开销总线的中断处理函数，当有link的los中断产生时，由该函数对中断进行处理，至于是SF，还是SD类型的中断，或者是CRC类型的中断，通过fpga给出的相应的寄存器获取：



然后在获取到的中断信息中轮询，查找标志位置位的接口，通过消息队列发送函数dev\_linkscan\_msg\_send将消息送到消息队列。

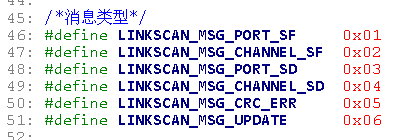
快速开销中断处理函数的注册：



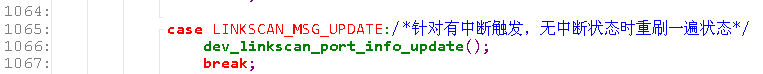


**link状态轮询函数dev\_linkscan\_port\_info\_update**

在快速开销支持的中断信息，下面几种中断消息类型：

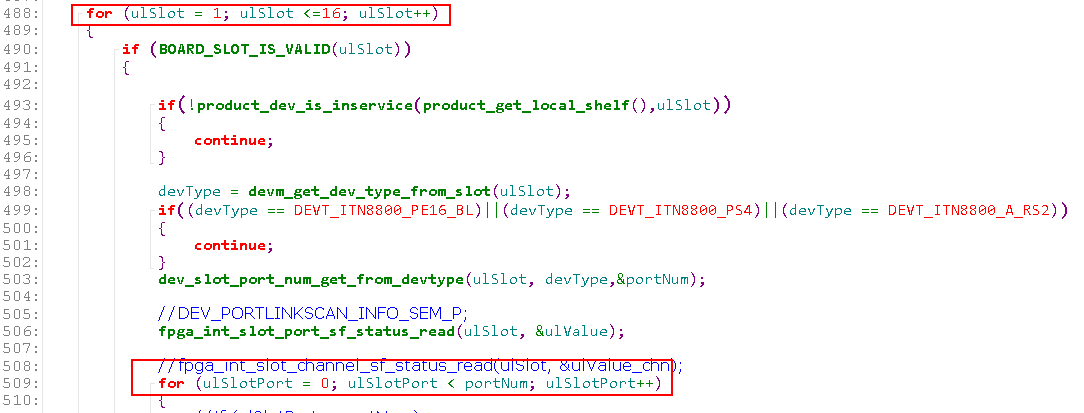


需要触发接口link状态轮询和更新的只有LINKSCAN\_MSG\_PORT\_SF和LINKSCAN\_MSG\_UPDATE：

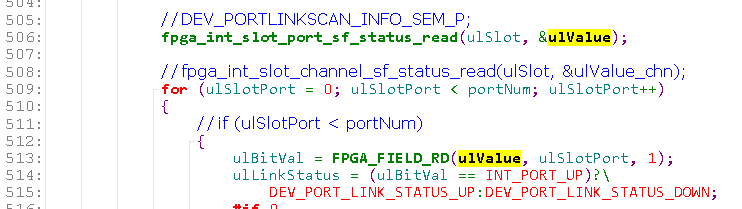


如果是LINKSCAN\_MSG\_PORT\_SF类型，那不会触发link事件，而是OTN接口的los更新上报。

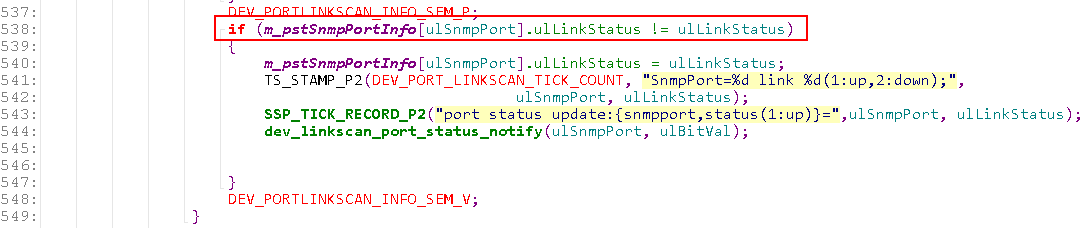
本函数轮询所有接口，通过fpga读取一个槽位的所有接口状态：



然后依次检查是否有接口的标志位置位：



如果置位，则说明该接口down，反之则说明该接口是up的。然后将这个从fpga获取的接口当前状态与软件保存的接口状态进行比较：



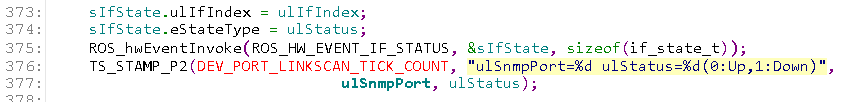
如果不一致，说明接口link状态发生变化，通过上图中的函数将接口link事件通知到相关模块，包括上报ROS平台的接口管理模块。

**link事件通知函数dev\_linkscan\_port\_status\_notify**

与ctcLinkScan.c中的中断处理函数ctc\_tms\_linkscan\_mac中的步骤一样，如果当前接口是linkdown的，那么需要快速通知涉及到保护倒换的功能模块，直接将接口linkdown事件通知到各个模块，尽量缩短倒换时间：



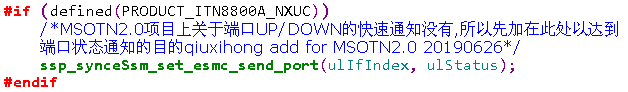
与ctcLinkScan.c中的中断处理函数ctc\_tms\_linkscan\_mac还需要读取sfp的los状态，以及与软件保存的接口状态比较不同，在将linkdown快速通知到各个模块后，直接通过hwapi将接口link事件上送ROS平台：



函数最后还有设置接口的port-en状态，以及点灯模式：



其他两个通知：



和



涉及到其他具体的功能模块，暂时不用继续深入。

**link事件上送平台**

在中断处理函数ctc\_tms\_linkscan\_mac中使用的回调函数**m\_pfLnkChngCallBack**，其注册和实现都是在sspIfmEvent.c文件。

先看文件中的接口事件处理的初始化函数：



第一个红色方框中的是创建的消息队列g\_ulPortLinkQueue，用于接口link消息的收发。第二个红色方框是创建的消息队列接收任务，从link消息队列g\_ulPortLinkQueue中收取消息。第三个红色方框是前面已经介绍过的，在中断处理函数ctc\_tms\_linkscan\_mac中使用的回调函数m\_pfLnkChngCallBack，其具体的回调实现就是方框里的函数SSP\_ifm\_link\_trigger，当接口状态发生变化时，触发这个回调函数，往link消息队列g\_ulPortLinkQueue中发送一个消息。

**消息队列发送函数SSP\_ifm\_link\_trigger**

函数实现只有一个步骤，就是将link状态发生变化的接口和当前状态等消息封装到固定的数据结构，然后送到link消息队列。



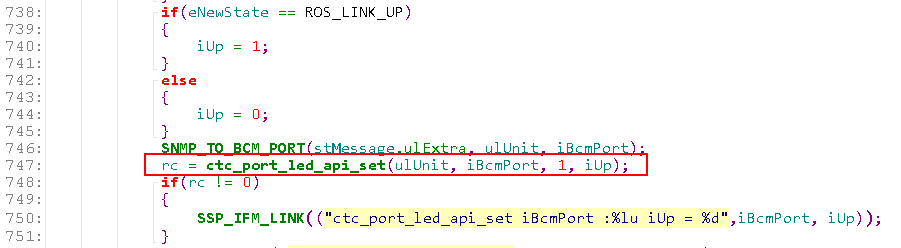
**消息队列接收函数SSP\_ifm\_link\_recv\_task**

这个函数是初始化创建的一个专门用来接收处理link消息队列g\_ulPortLinkQueue消息的线程。

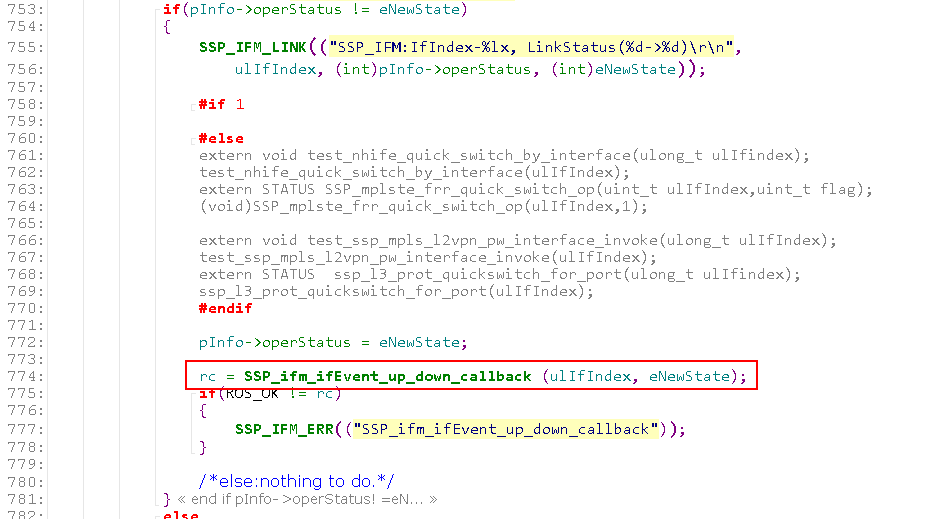


如果消息队列接收超时或者错误，则通过SSP\_ifm\_link\_poll函数轮询所有接口的状态，有当前状态与保存到软件的状态不一致的接口时上送ROS平台。

正常从消息队列中收到消息后，首先校验收到的消息内容，检查接口号是否正确，以及link状态是up，还是down。然后根据消息类型设置接口led点灯模式，是点亮还是熄灭：



然后与软件保存的接口的上一次状态进行比较，如果状态不一致，则调用事件通知函数SSP\_ifm\_ifEvent\_up\_down\_callback，将link状态上报ROS平台：



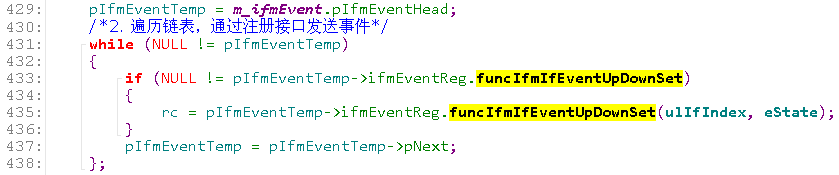
简单总结，接口的这个link消息是通过消息队列传送，link状态上报平台时就是通过一个线程从link消息队列中读取接口状态变化产生的消息，如果有消息，则调用事件通知函数SSP\_ifm\_ifEvent\_up\_down\_callback，将link状态上报ROS平台。

**事件通知函数SSP\_ifm\_ifEvent\_up\_down\_callback**

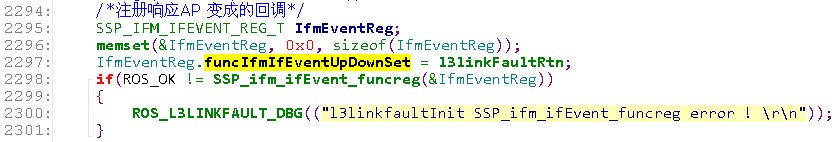
函数是驱动上报给ROS平台的最终调用接口，通过注册的hwapi事件将接口的link事件上报的平台的接口管理模块。

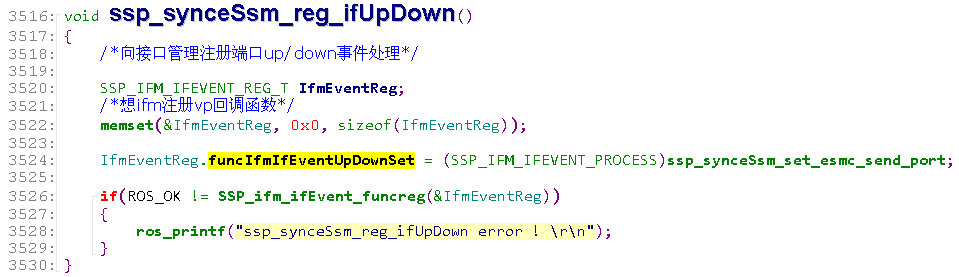


从函数内的注释来看，第一步是惯例的检查接口是否有效，第二步则是通过注册的回调函数，将接口状态变化通知到注册该事件消息的功能模块：



往上跟踪注册的回调函数，驱动层有两个模块注册了接口的link事件通知函数，分别如下：



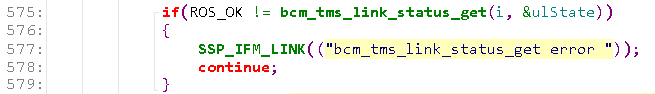


需要了解这两个功能模块时可以继续看这两个回调函数的具体实现。

函数最后一步就是通过hwapi注册的ROS\_HW\_EVENT\_IF\_STATUS事件类型，将接口link状态通知到ROS平台的接口管理模块。（hwapi的工作流程需要另外梳理）

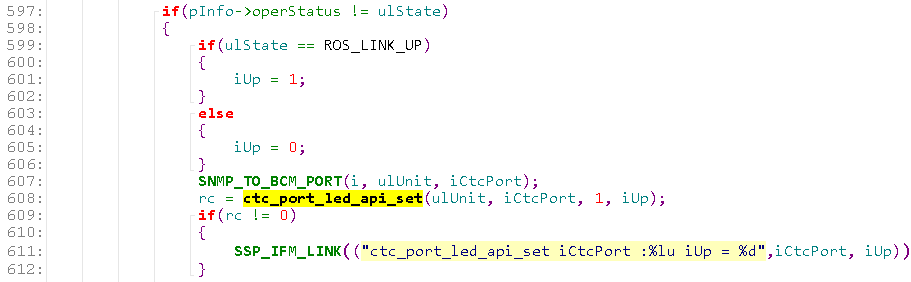
**状态轮询函数SSP\_ifm\_link\_poll**

函数需要轮询设备所有接口，对于部分设备要判断是否是插的电模块。轮询时先获取接口当前的link状态：



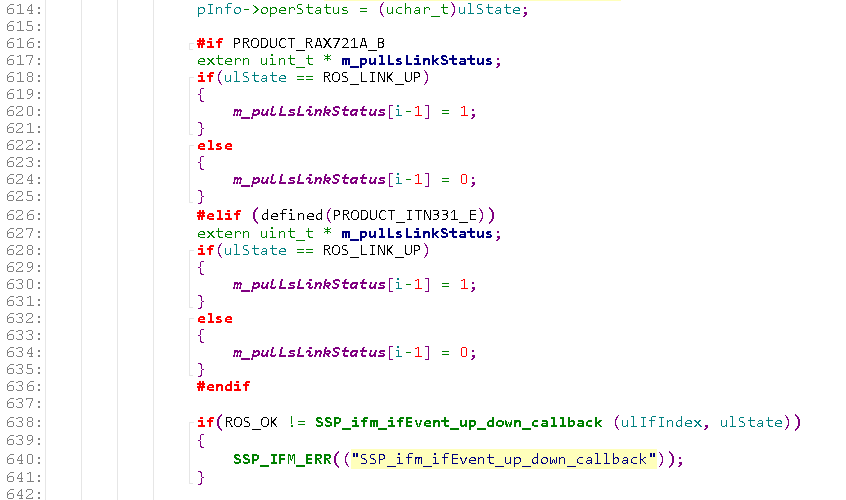
这个函数在ctcLinkScan.c文件里介绍。

得到接口当前状态后，与软件保存的接口的上一次状态进行比较，如果状态不一致，根据消息类型设置接口led点灯模式，是点亮还是熄灭：



这个与消息队列接收函数SSP\_ifm\_link\_recv\_task的处理有一点不同，是在当前接口状态与软件保存的状态不一致时去设置点灯，而SSP\_ifm\_link\_recv\_task是比较之前就设置点灯模式，原因是能进入SSP\_ifm\_link\_recv\_task函数，说明接口状态是发生了变化的，而后面比较当前接口状态与软件保存的状态是确保不会将同一状态反复上报ROS平台。

设置完点灯模式后，更新软件保存的接口状态，然后调用事件通知函数SSP\_ifm\_ifEvent\_up\_down\_callback将变化的接口link状态通知到ROS平台的接口管理模块：

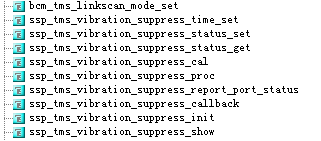


这里还更新了保存接口状态的全局变量*m\_pulLsLinkStatus*，这个变量前面介绍过，主要是在ctcLinkScan.c文件里的中断处理函数中使用，这里不清楚为什么要在721A\_B和iTN331\_E设备上轮询时更新这个全局变量（为了保证这个全局变量的准确性，避免ctcLinkScan.c文件里的中断处理函数出现问题，导致更新异常？）。

**驱动接口震荡抑制（基本无效）**

目前721C和iTN8800-A设备的接口震荡抑制是ROS平台的接口管理模块实现的，驱动层的震荡抑制没有用到，这里主要介绍驱动层的震荡抑制功能。

在ctcLinkScan.c文件中的接口震荡函数与ssp\_ctc/ssp\_ifindex/src/sspIfmVibrationSuppress.c中的接口函数在命名上相同，具体的函数实现上有细微区别，后者是将震荡抑制功能模块化之后以单独的文件归档的，目前这个文件没有使用，编译时也没有编译该文件。



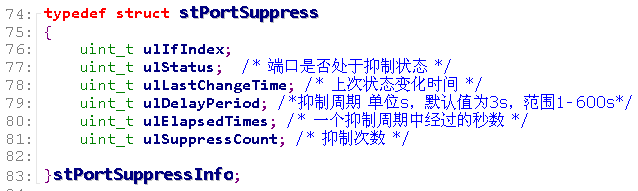
**ctcLinkScan.c文件的震荡抑制**

首先看震荡抑制功能的初始化：

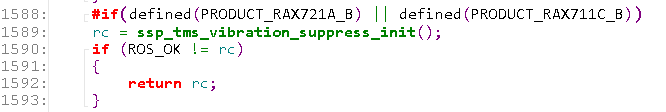


初始主要有三个步骤，初始化全局变量g\_stPortSuppress、创建震荡抑制任务和创建保护全局变量的信号量。

全局变量g\_stPortSuppress的数据结构类型：



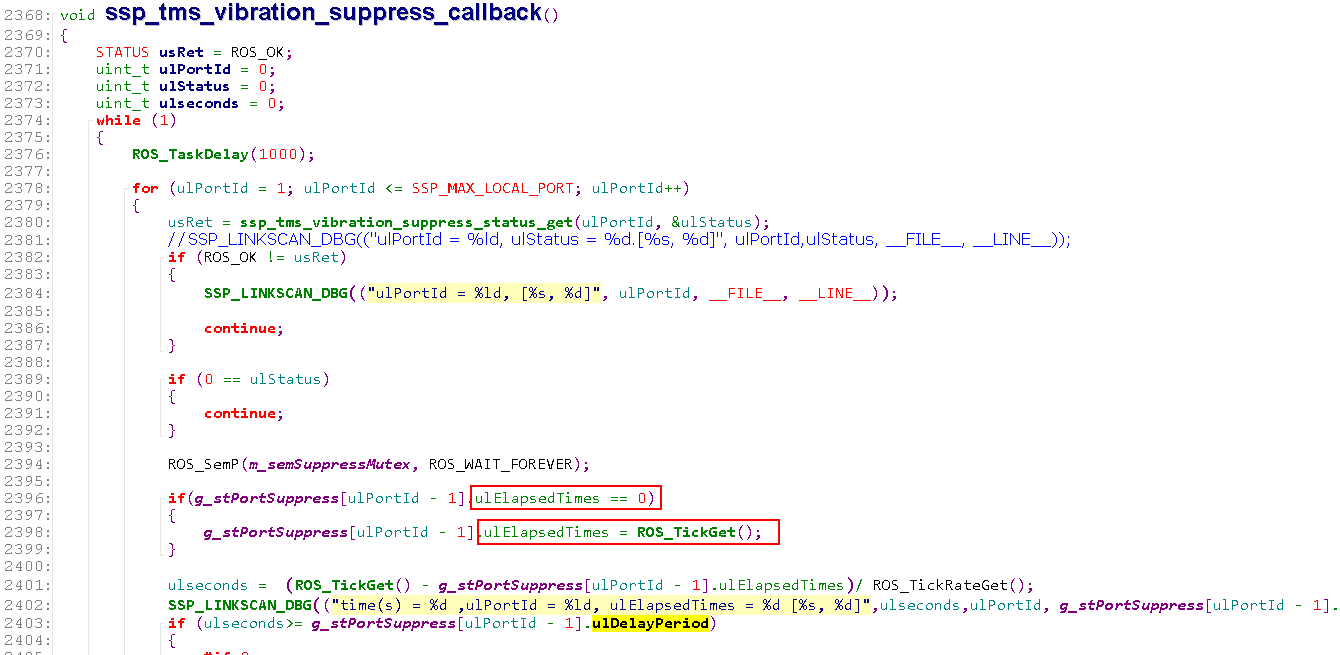
然后查看调用初始化函数的地方：





当前初始化的地方有两个地方使用，一个是ctcLinkScan.c中的link\_scan\_init函数，只在721A\_B和711C\_B两款设备上初始化调用，另一个则是sspIfm.c中的SSP\_Ifm\_init函数，没有区分设备类型，所有设备都会初始化接口震荡抑制功能。这个应该是有冲突，SSP\_Ifm\_init初始震荡抑制会让link\_scan\_init中的宏隔离失效，不知道什么原因要这么写。

初始化创建的震荡抑制任务：



这个任务创建的主要目的就是在接口产生震荡时，延缓状态上报，在一个抑制周期后再将接口link状态上报平台。

但是仔细分析这个函数，全局变量g\_stPortSuppress中的抑制周期ulDelayPeriod成员初始化是给的0，也就是说没有抑制作用（平台有配置命令）。另外，震荡时间的计时成员ulElapsedTimes在这个任务里面没有实时更新，而在其他触发震荡抑制的接口函数ssp\_tm\_vibration\_suppress\_cal中又注释掉了ulElapsedTimes变量的使用，导致这个主任务里面的时间比较没有意义。所以，从这里可以看出ctcLinkScan.c文件中的这一段震荡抑制功能是不完整的，没有抑制作用。

**sspIfmVibrationSuppress.c的震荡抑制**

震荡抑制功能的初始化：



函数功能与ctcLinkScan.c中的震荡抑制初始化一样，但是这个文件里的初始化是给定了抑制周期ulDelayPeriod的，默认是3秒，也就是3秒内的接口状态变化不上报ROS平台。

再看创建的震荡抑制主任务：



中间是有更新抑制持续时间ulElapsedTimes，然后与设置的抑制周期比较的。当抑制持续时间超过抑制周期时，将驱动当前的接口link状态上报ROS平台：



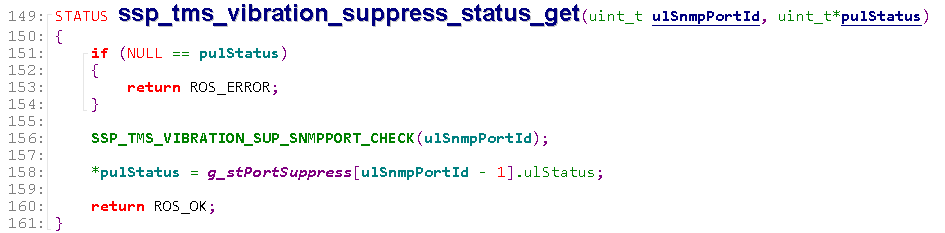
这个函数暂时没有放开功能，但是注释掉的这个接口函数是实现了hwapi通知ROS平台的：



再看对外提供触发震荡抑制功能的函数接口：



先检测接口是否已经处于震荡抑制状态：

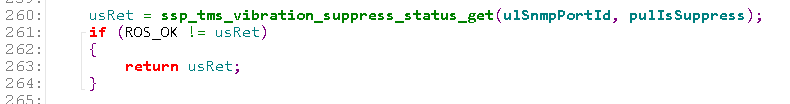


如果不是则判断是否需要进行震荡抑制：



如果接口状态在1秒内发生变化，函数判定需要进行震荡抑制，并将震荡时间ulElapsedTimes置0，这个时间由主任务函数ssp\_tms\_vibration\_suppress\_callback刷新。

震荡抑制判断完成后再次获取接口当前是否处于震荡抑制状态，然后函数返回。



外部模块如何触发震荡抑制功能只需要看对外提供触发震荡抑制功能的函数如何被调用即可：

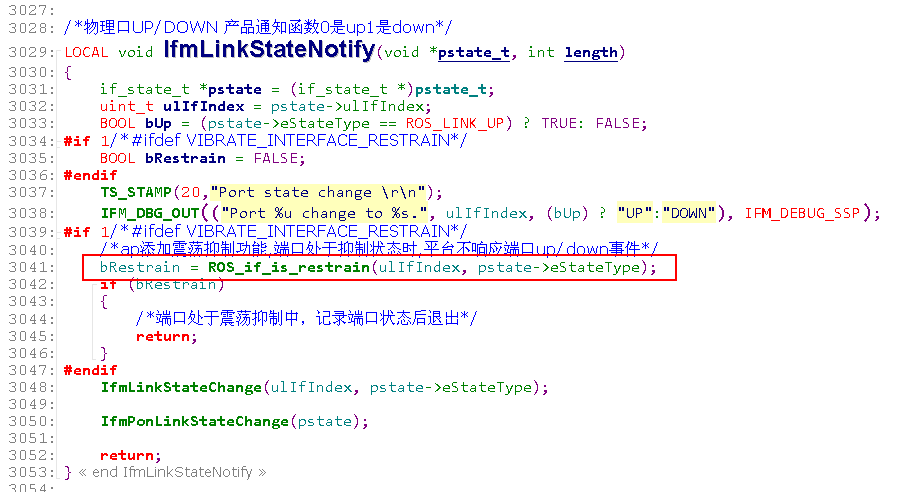


比如在芯片注册的中断处理函数ctc\_tms\_linkscan\_mac中，当接口link状态变化的中断上报到中断处理函数，软件判断接口当前状态发生变化时，通过震荡抑制功能的接口函数蓝判断是否继续后续处理。

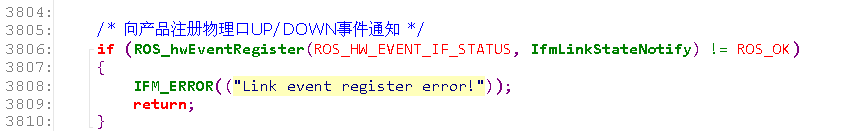
**平台接口震荡抑制**

目前默认接口状态由up->down一次时计数加一（down->up不计入），每30s计数减半，计数超过10次后，接口状态不在对外通知，计数小于4次后恢复对外通知。

平台接口管理接收驱动发送的接口状态变化通知的函数：



这个函数在ifm/ifm\_dev.c的if\_init函数中注册：



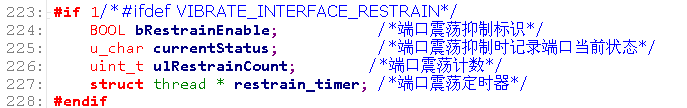
当驱动通过ROS\_hwEventInvoke通知接口状态事件时，ROS平台会进入注册的这个IfmLinkStateNotify函数处理。

**抑制判断函数ROS\_if\_is\_restrain**

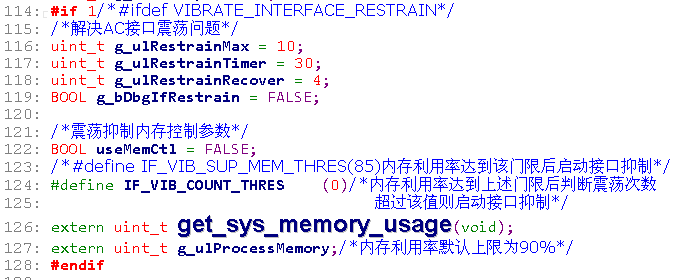
接口收到link状态变化通知时先由该函数判断是否处于震荡抑制状态：



接口管理的主要数据结构rosIf\_t中定义震荡抑制相关的成员，其中bRestrainEnable表示接口当前是否处于震荡抑制状态：



当平台收到的状态是linkdown时，震荡抑制的计数ulRestrainCount加1，默认的震荡计数最大10次：



**定时器处理函数**

平台接口震荡抑制的定时器处理函数：

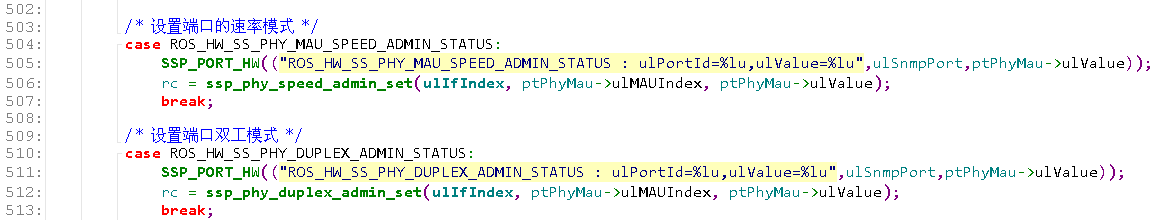


在平台添加一个接口时会默认创建一个定时器任务，传入参数就是接口的索引，定时器周期gulRestrainTimer默认为30秒。如上图定时器任务函数的实现，定时器到期后对震荡抑制的计数减半操作，然后再判断抑制计数是否小于最低的计数g\_ulRestrainRecover（默认4次），如果小于就恢复接口管理对外通知接口状态。

**接口配置**

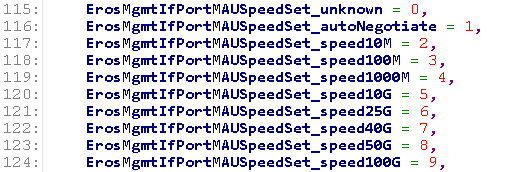
**接口速率、双工模式配置**

接口速率和双工模式的命令行配置消息下发后，驱动层接收hwapi的消息在ctcPhyHw.c中处理：



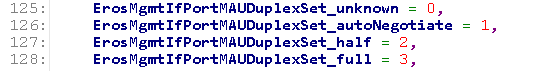
**速率、自协商（ROS\_HW\_SS\_PHY\_MAU\_SPEED\_ADMIN\_STATUS）**

接口速率根据接口不同，支持不同的速率配置，比如千兆接口支持十兆和百兆速率配置，万兆接口可以支持千兆速率配置。接口速率默认情况下是自协商模式。



**双工模式（ROS\_HW\_SS\_PHY\_MAU\_DUPLEX\_ADMIN\_STATUS）**

接口双工模式默认是自协商



目前，速率和双工模式都区分iTN8800和其他设备类型，如果是非iTN8800设备，那么检查一下平台下发的速率类型和双工模式是否合法，然后直接下发配置到芯片。如果是iTN8800设备，那么在配置到芯片前，还需要板间通信消息下发到单盘，然后主控再下发配置到芯片。

在非iTN8800设备的速率配置函数ssp\_phy\_speed\_admin\_set中，只需要通过函数\_ssp\_sfp\_speed\_check检查平台下发的速率类型是不是合法的：



而在iTN8800设备上还需要IBC消息通知到单盘进行配置：



另外，iTN8800设备上不同板卡支持不同的sfp，所以还需要检查sfp类型：

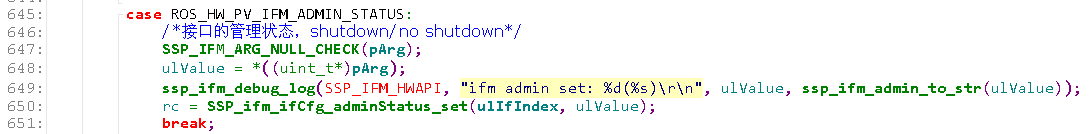


同样的，双工模式的配置也区分iTN8800和其他设备，也要检查sfp类型和ibc消息通知单盘进行配置。

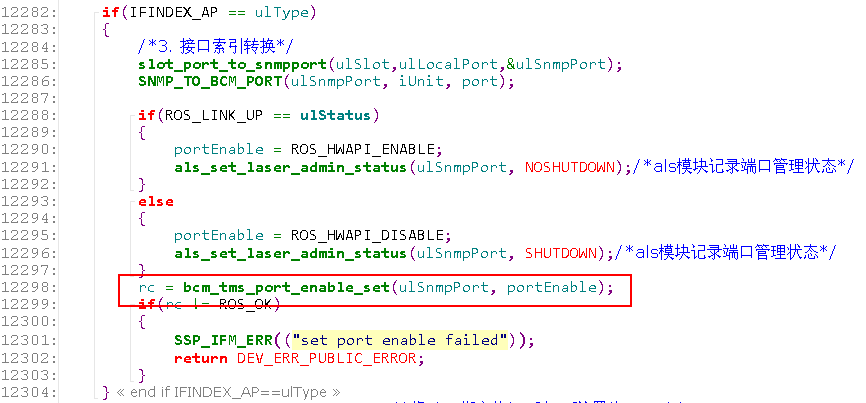
另外，iTN8800-A-NXUC设备在驱动层配置完速率后要再次获取实际的速率，然后上报到ROS平台，这个不知道目的是什么，看配置是否生效？？？

**shutdown/no shutdown**

接口状态的命令行配置消息下发后，驱动层接收hwapi的消息在sspIfmHw.c（ROS\_HW\_PV\_IFM\_ADMIN\_STATUS）中处理：



在函数SSP\_ifm\_ifCfg\_adminStatus\_set中对不同类型接口的shutdown/no shutdown配置需要处理的内容是不同的，比如物理口的shutdown/no shutdown：



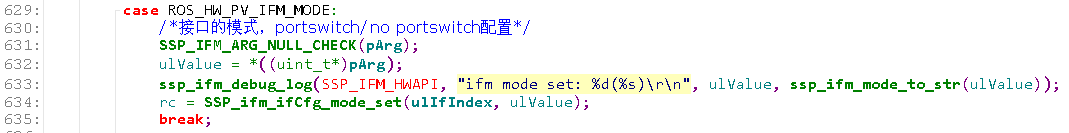
只需要设置接口的mac-en使能和非使能，以及接口的发光使能和非使能即可。而对于子接口或者vlanif这种逻辑口来说，shutdown/no shutdown不用配置激光器关断或者mac-en，需要设置的是芯片上的一些默认配置：



比如vlanif口的配置，需要更新创建vlanif口时芯片上对应的l3if接口的配置。这个没有继续查看vrf的配置过程。

**portswitch**

接口模式的命令行配置消息下发后，驱动层接收hwapi的消息在sspIfmHw.c（ROS\_HW\_PV\_IFM\_MODE）中处理：



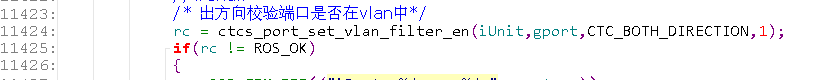
接口默认模式为三层模式，如果需要配置为二层接口模式，接口下配置portswitch命令即可，如果要恢复三层模式，接口下配置no portswitch命令即可，下发的两种模式类型：



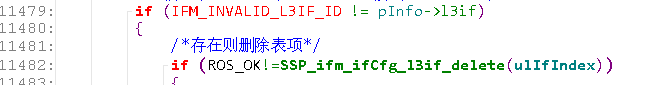
接口从默认的三层模式配置成二层模式时，需要删除三层接口相关的配置：



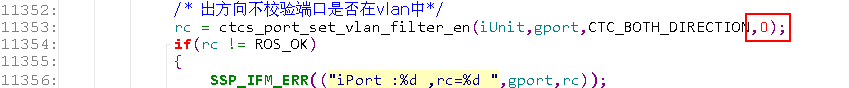
然后要使能二层的vlan检查功能：



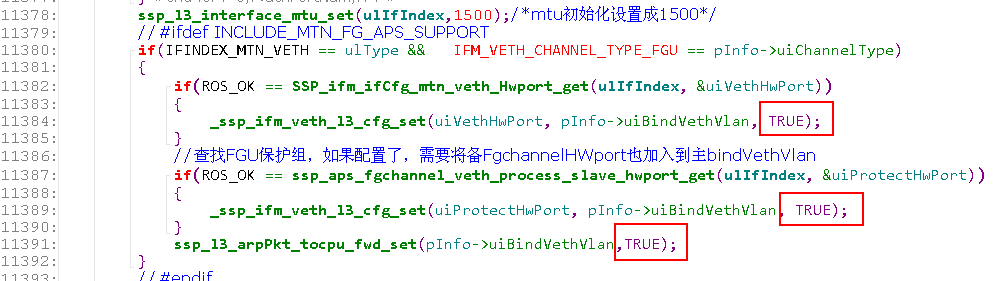
如果这个物理口下有创建l3if三层接口，也需要删除：



相反的，如果接口从二层模式恢复成三层模式时，需要关闭接口的vlan检查过滤功能：



重新使能三层接口相关的配置：



接口切换模式时平台不止下发一个ROS\_HW\_PV\_IFM\_MODE消息，比如二层模式切回到三层模式时，还涉及到三层接口默认的IP地址，接口下默认的子接口创建等，打开驱动层的debug调试信息查看配置流程涉及到的操作：

