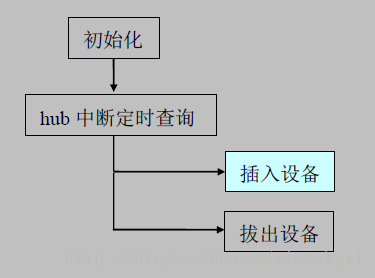
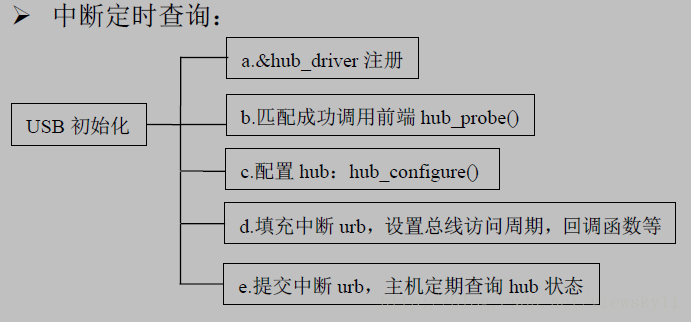
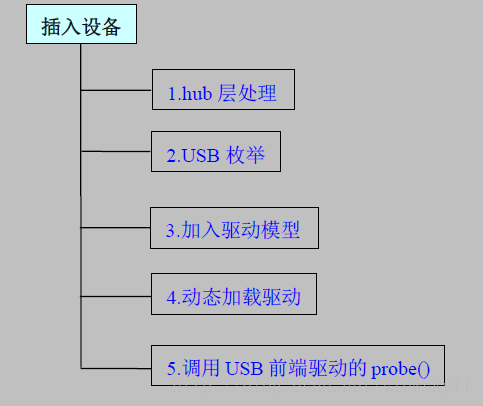
USB接口上有4个管脚，OTG功能有5个。原来4个分别是电 D+ D- 地。 现在增加了一个ID。这个ID线就决定了自己做主设备还是从设备。如果ID线是高则自己是从设备，反之是主设备。

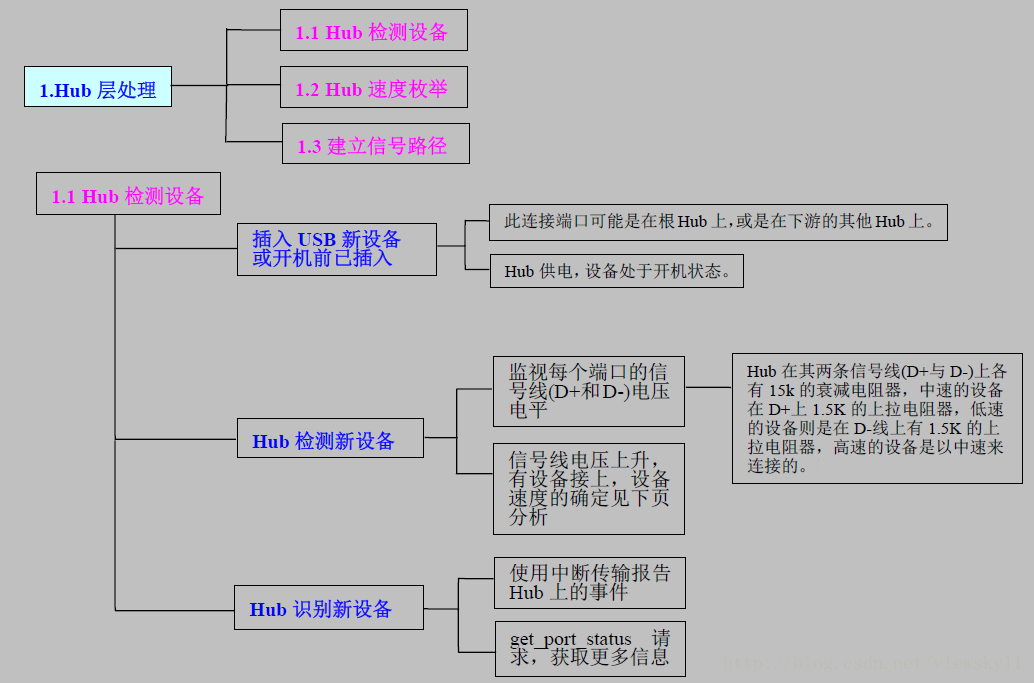
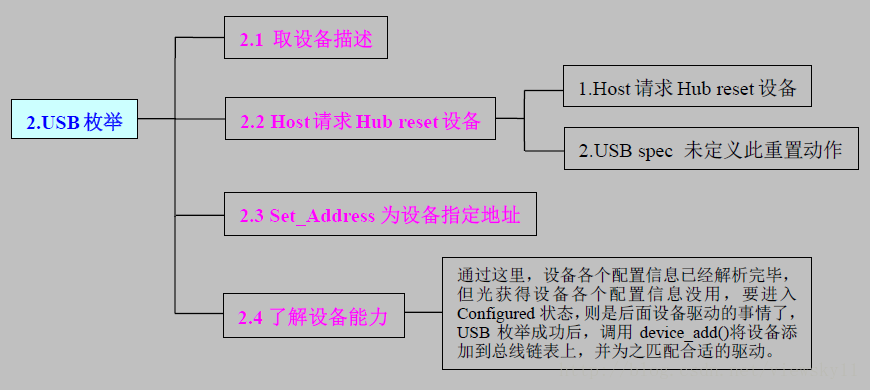
这个模块有4个USB 2.0控制器core，对于OTG Core和HOST1 Core均是HS/FS/LS UTMI compliant interface；而HOST2 Core和HOST3 Core则是High Speed Inter-Chip USB compliant interface (HSIC)。这两个接口有何异同，OTG Core和HOST1 Core是内部集成PHY的。

HSIC接口是用于chip-to-chip连接的，简单的说就是省去了外部USB线缆，是对于HOST1 Core（with phy）的外部接口说的。UTMI或者ULPI指的是HOST1 CORE 内部的host controller与PHY之间的连接方式。

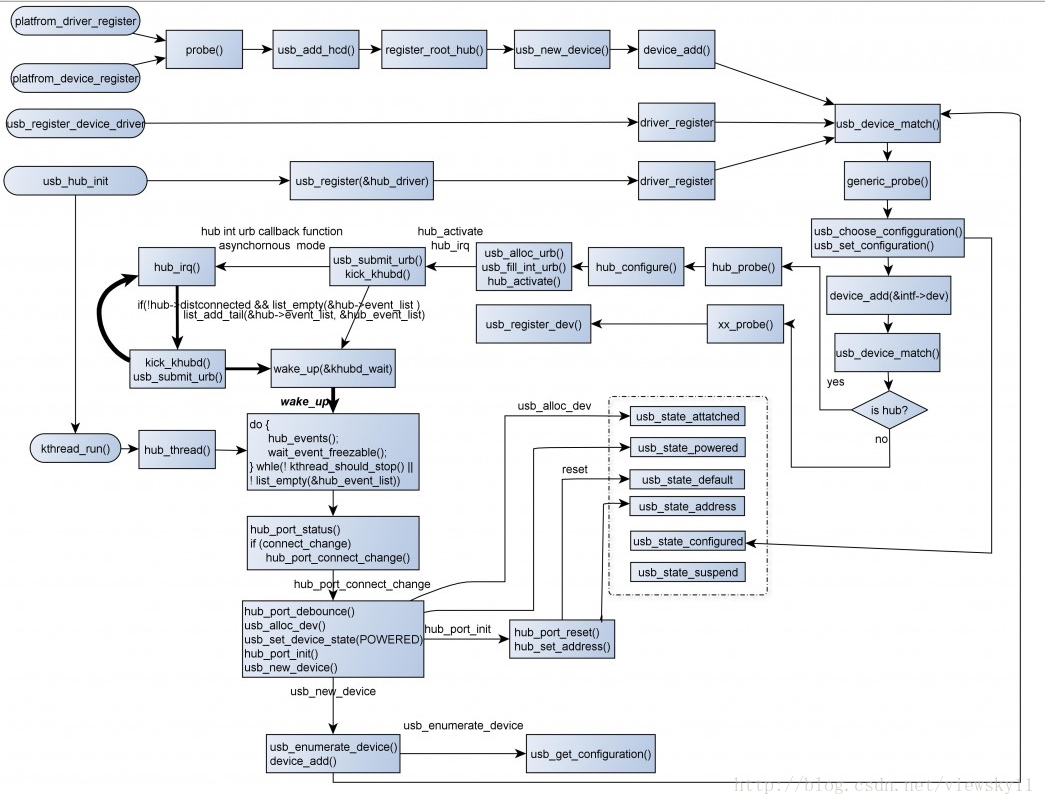
1. 中断定时查询：



中断定时查询：   
  
2. 总体架构设计：   
   
3. 解析各个部分：

Hub层处理   
  
usb枚举   


当守护程序第一次运行或usb port上状态发生变化，守护进程被唤醒都会运行hub\_events函数，这个函数在usb系统中处理核心位置，usb的枚举过程就是由它完成。

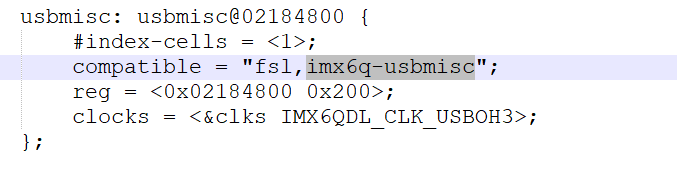
usb具体的枚举流程：   




USB模块中有两种寄存器：USB核心寄存器和USB非核心寄存器。  
USB核心寄存器用于控制USB核心功能，USB非核心寄存器是附加功能，i.MX系列产品的非核心寄存器有所不同。

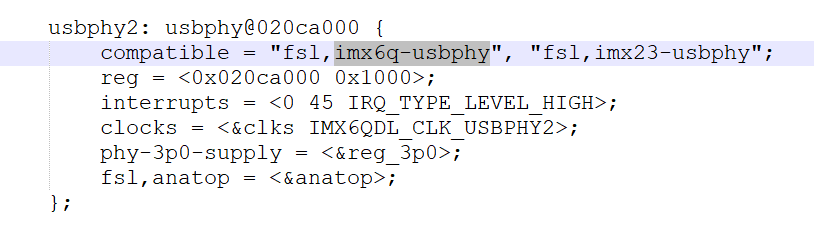
USB Non-Core Memory节点：  
X:\zhengfengjun\i.MX6-kernel-4.9.88\drivers\usb\chipidea\usbmisc\_imx.c

usbmisc\_imx\_probe对于usbmisc操作节点的操作函数是imx6q\_usbmisc\_ops，包括set\_wakeup、init 、charger\_primary\_detection 、charger\_secondary\_detection、hsic\_set\_connect、hsic\_set\_clk



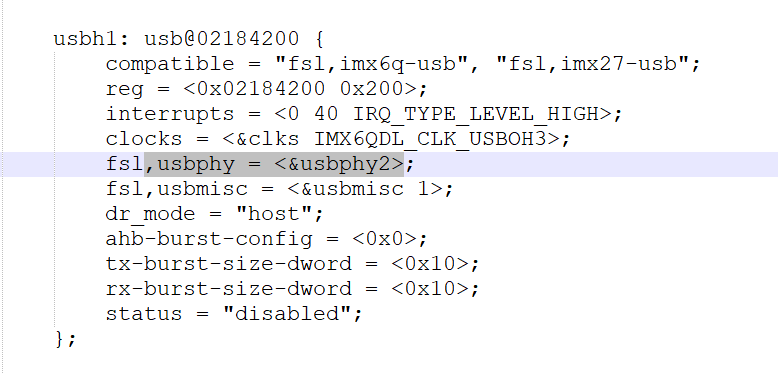
X:\zhengfengjun\i.MX6-kernel-4.9.88\drivers\usb\phy\phy-mxs-usb.c

mxs\_phy\_probe对于phy节点的驱动的注册和初始化。驱动中包括对phy的操作函数，如Init /shutdown /set\_suspend/ notify\_connect /notify\_disconnect/set\_wakeup/ set\_mode /notify\_suspend /notify\_resume



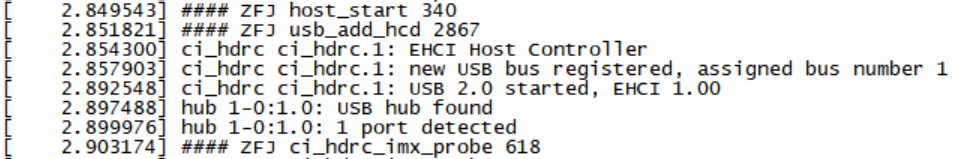
USB Core Memory节点：  
X:\zhengfengjun\i.MX6-kernel-4.9.88\drivers\usb\chipidea\ci\_hdrc\_imx.c

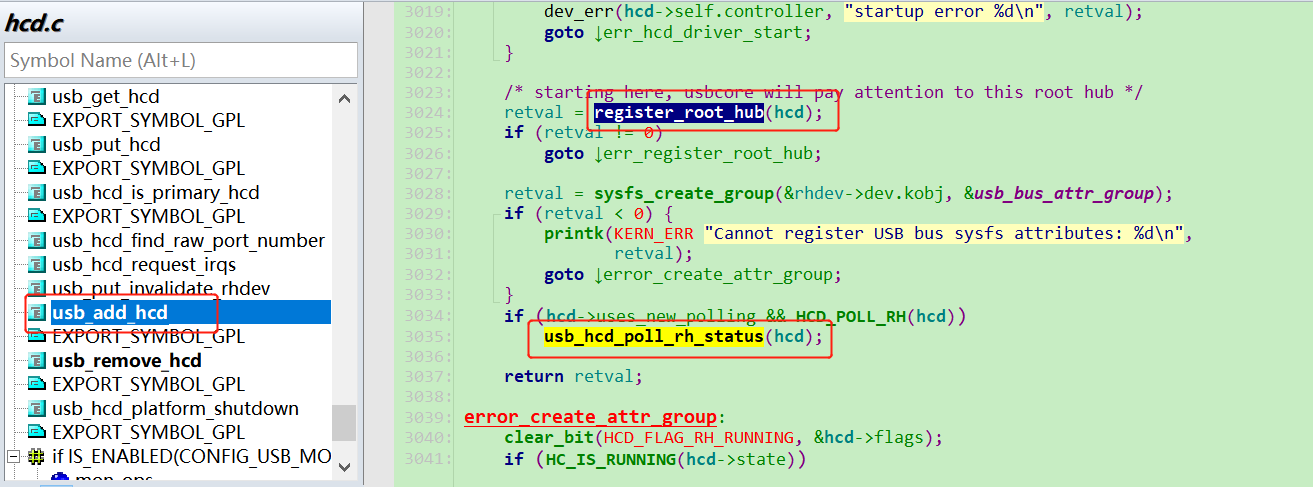
ci\_hdrc\_imx\_probe对于USB控制器USB Core Memory节点驱动的注册和设备的初始化



**ci\_hdrc\_imx\_probe** -> ci\_hdrc\_imx\_probe -> ci\_hdrc\_add\_device   -> platform\_device\_add -> **device\_add到此注册了一个"ci\_hdrc"设备，设备的驱动ci\_hdrc\_driver在X:\zhengfengjun\i.MX6-kernel-4.9.88\drivers\usb\chipidea\core.c**

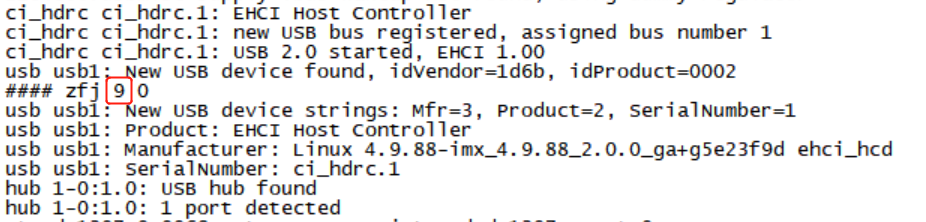
**驱动初始化函数ci\_hdrc\_probe->ci\_hdrc\_host\_init->ci\_role\_start->host\_start->usb\_add\_hcd**

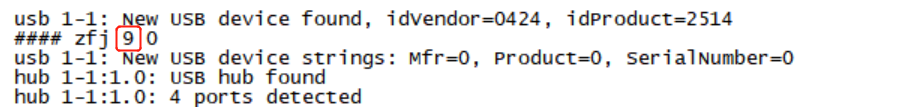




register\_root\_hub注册root hub设备，执行初始化函数hub\_probe。

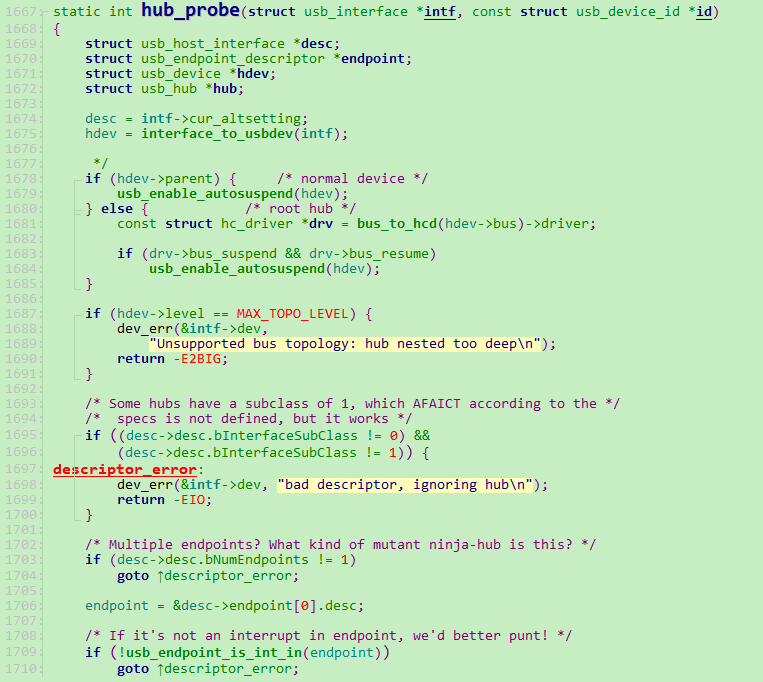
register\_root\_hub->usb\_new\_device->usb\_enumerate\_device主是要获取usb设备的配置信息，然后将usb设备添加到系统中，执行完device\_add函数后，如果设备是root hub，匹配的可能不是名字，可能是hub\_id\_table结构体中的bDeviceClass。如果从设备获取的描述符bDeviceClass=9，那么代表设备是hub，那么执行hub\_probe。





在添加主机控制器驱动时，创建了一个定时器hcd->rh\_timer，定时器的执行函数是rh\_timer\_func，这个定时器在register\_root\_hub完成后启动了。usb\_hcd\_poll\_rh\_status则是启动函数。在最后会详细解释。

# IMG_256





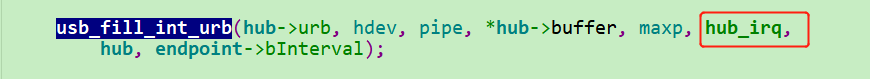
初始化hub，最终调用hub\_configure():

该函数内部所做的工作比较多：

a> 获取hub描述符 get\_hub\_descriptor

b> 获取hub状态 hub\_hub\_status

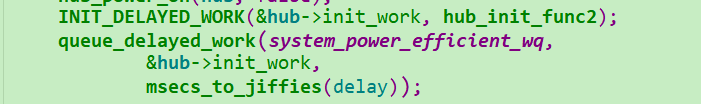
c> 填充一个hub中断 hub\_irq()

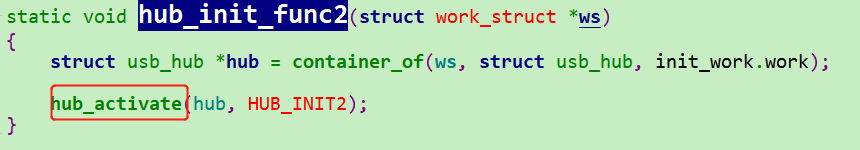


最终调用hub\_activate()激活hub。

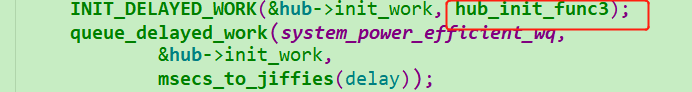


hub\_activate：创建工作队列，同时绑定对应的回调函数hub\_init\_func2

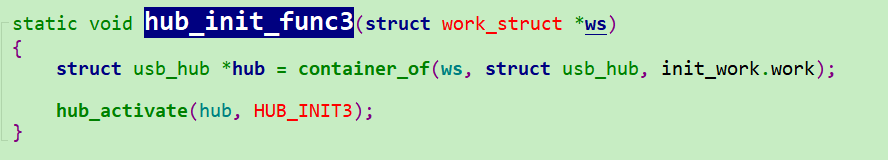




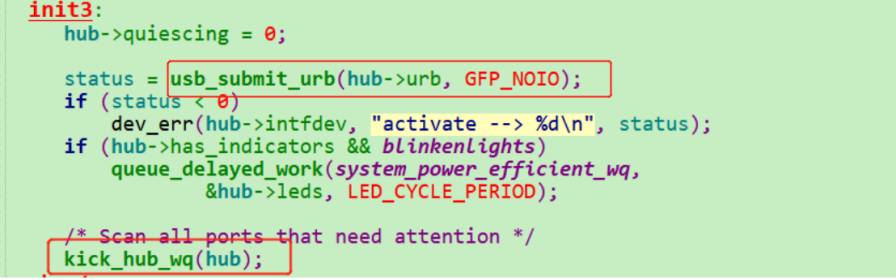
hub\_activate(hub, HUB\_INIT2); //激活HUB\_INIT2

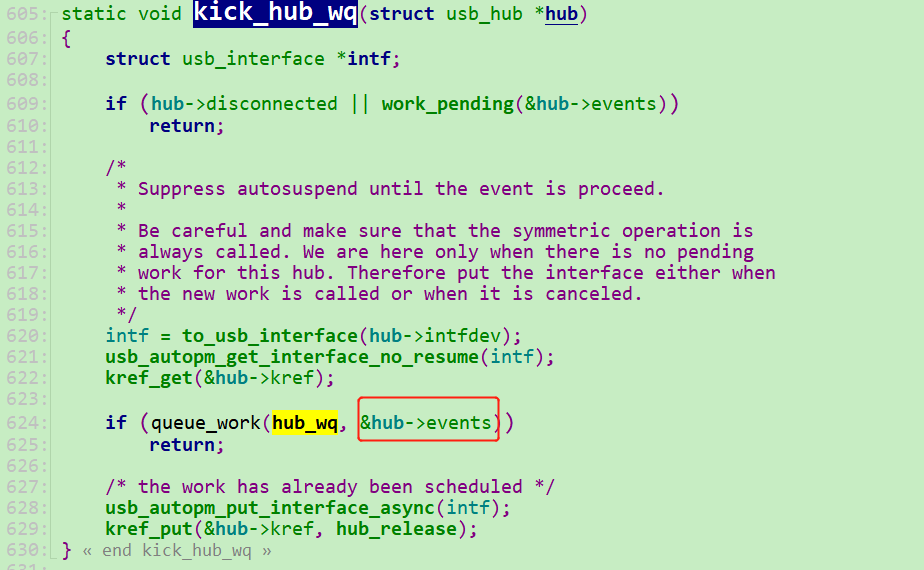


初始化工作队列，同时绑定回调函数hub\_init\_func3

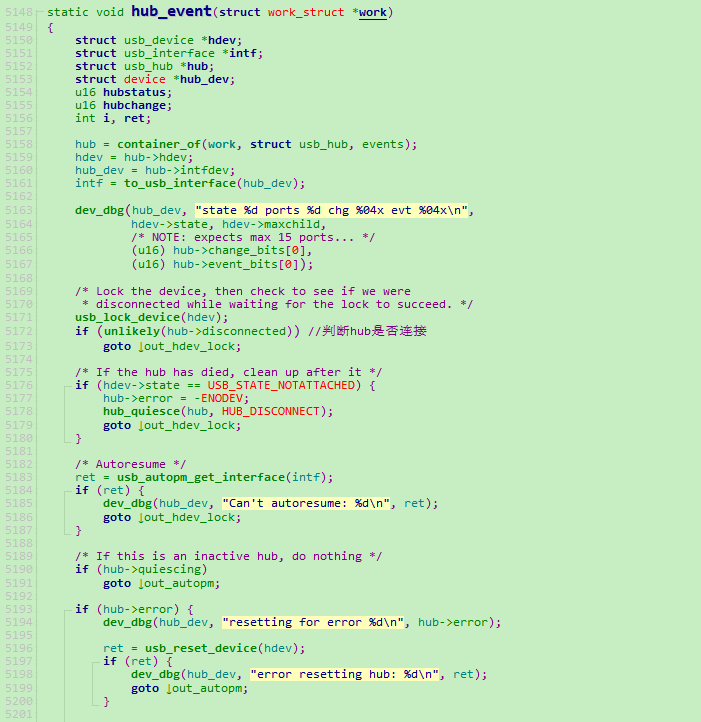


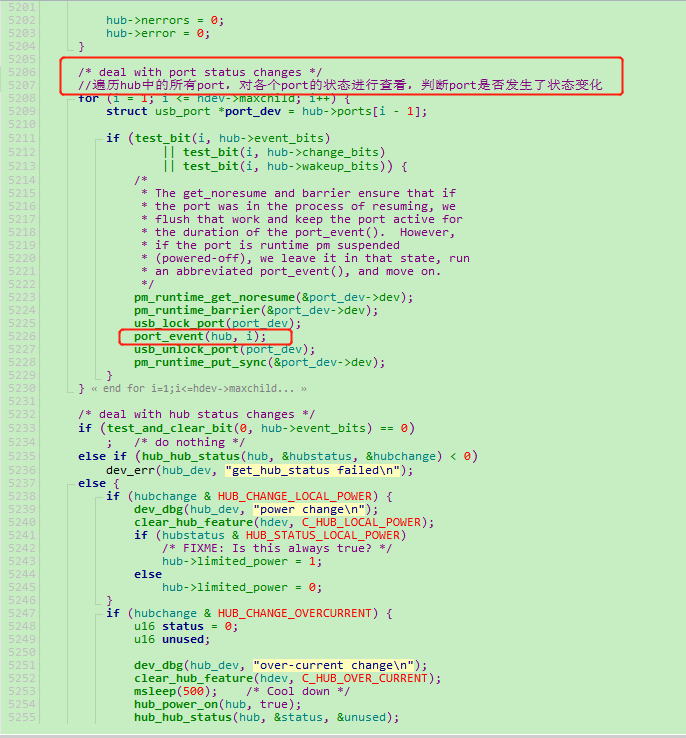
最终调用kick\_hub\_wq()



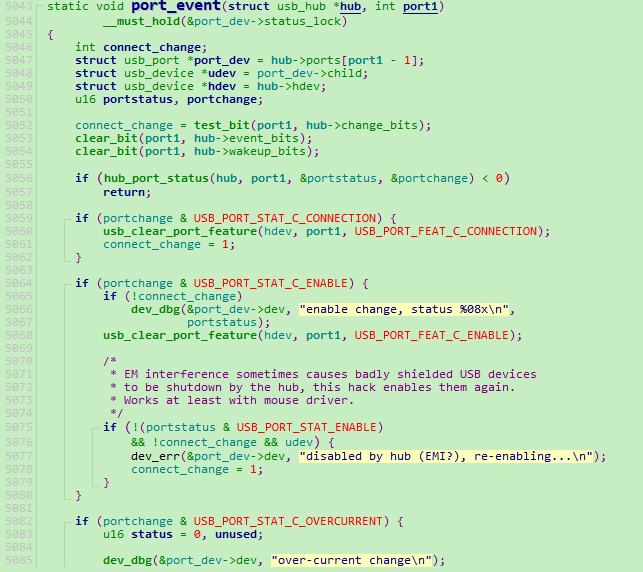


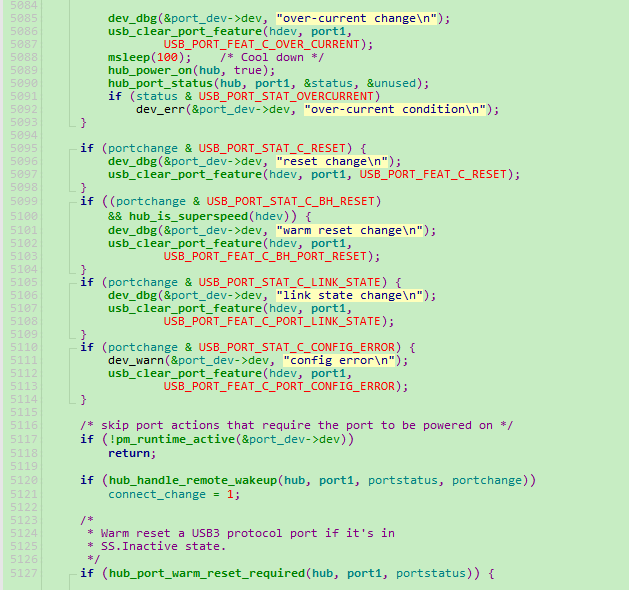
hub->events其实就是调用函数hub\_event。

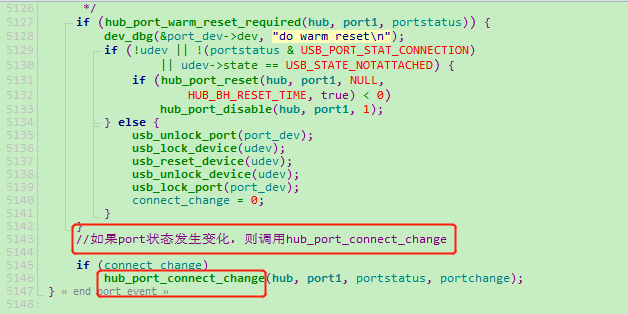




做了一些逻辑判断，判断hub是否连接，hub上是否有错误，如果有错误就重启hub，如果没有错误，接下来就对hub 上的每个port进行扫描，判断各个port是否发生状态变化；





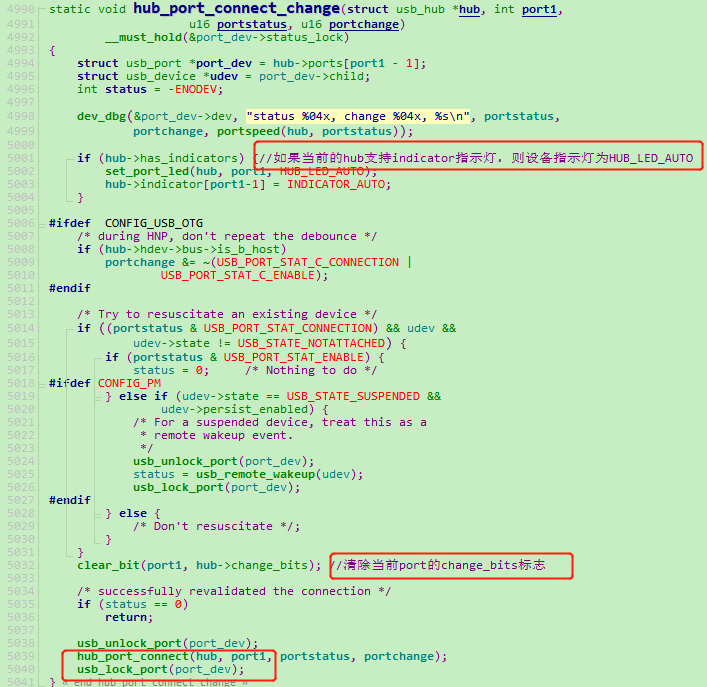


接着遍历hub中的所有port，对各个port的状态进行查看，判断port是否发生了状态变化，如果产生了变化则调用hub\_port\_connect\_change进行处理；

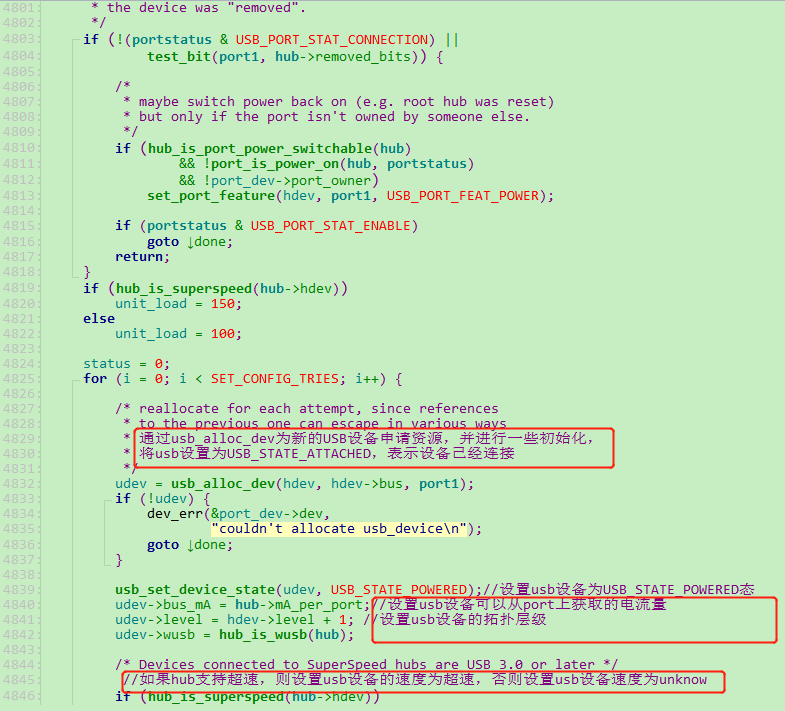
在hub结构中存在多个标志位，用来表示port一些状态，port应该是hub上一分四的四个端口：

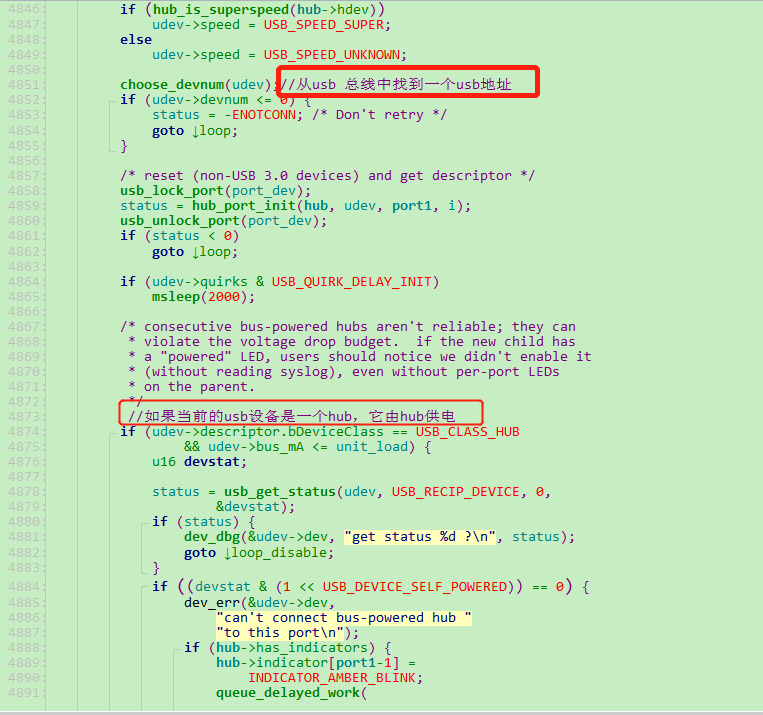
1. event\_bits中第0位用来表示hub本身是否产生local power status change和是否产生过流，其它的各位用来表示hub下各个port状态是否发生变化，这些状态包括： USB\_PORT\_STAT\_C\_CONNECTION连接状态发生变化，USB\_PORT\_STAT\_C\_ENABLE端口使能状态发生变化，USB\_PORT\_STAT\_C\_OVERCURRENT端口过流状态发生变化，USB\_PORT\_STAT\_C\_RESET端口reset10ms置位；
2. change\_bits用来表示hub下各个端口逻辑状态是否发生变化，它是在hub初始化的时候赋值的，这些状态主要有：1.原先port上没有设备，现在检测到有设备连接；2.原先port上有设备，由于控制器不正常关闭或禁止usb port等原图使得该设备处于NOATTACHED态，则需要断开设备；
3. wakeup\_bits

在遍历port时，首先对这些标志进行判断，如果没有产生变化则跳过当前port，否则读取当前port的status，判断出产生状态变化的原因；   
如果发生变化port的连接状态发生变化，清除相应的端口状态，设置connect\_change变量为1；   
如果port的使能状态发生变化，清除相应的状态标志，如果是由于EMI电磁干扰引起的状态变化，则设置connect\_change变量为1；   
经过一系列列port判断，如果发现port的连接状态发生变化或由于EMI导致连接使能发生变化，即connect\_change=1，则调用hub\_port\_connect\_change.











hub\_port\_connect\_change分为两部分：

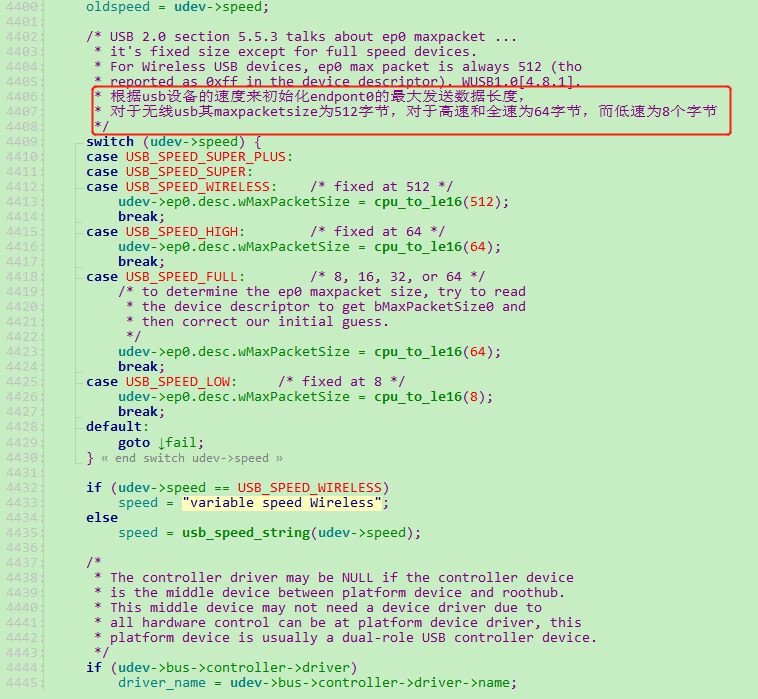
第一部分主要是：确认是否有新设备插入；   
第二部分主要是：确认port口上有新设备插入后，分配设备资源，并进行枚举操作；

对于一些已经连接的设备，进行一些类似防抖动处理，处理机制：每隔25ms去读取port的status和change状态，查看port连接状态是否稳定，如果port处于稳定状态大于100ms，则认为当前port上的设备已经稳定，这种处理机制最长时间为1.5s;如果port处于稳定状态的时间小于100ms，则认为连接是不稳定的，则跳出当前port，去执行下一个port;

对于一个新插入的设备，并已经确定它的存在，则接下来会为这个usb设备进行枚举

一条usb总线下总共可以有128个设备，usb总线通过位图的形式来管理usb设备的地址，choose\_devnum是从usb 总线中找到一个usb地址，usb地址在1和128之间；   
通过hub\_port\_init对usb设备进行reset，并设置usb设备的地址，获取usb设备的设备描述符，这个函数相对比较重要，所以对它进行了进一步分析：









r = usb\_control\_msg(udev, usb\_rcvaddr0pipe(),USB\_REQ\_GET\_DESCRIPTOR, USB\_DIR\_IN,USB\_DT\_DEVICE << 8, 0,buf, GET\_DESCRIPTOR\_BUFSIZE,initial\_descriptor\_timeout); 这里USB控制消息，就是用USB总线驱动提供的一些函数发一些控制消息

我们前面说有四种传输，这就是控制传输。他有什么命令SB\_REQ\_GET\_DESCRIPTOR 这里 获得描述符，当我们没有到获得描述符的那部分，现在还只是告诉地址部分，在这个函数后面有个hub\_set\_address这个函数 设置地址，把地址告诉USB设备，然后马上retval = usb\_get\_device\_descriptor(udev, 8);调用这个获得文件描述符。

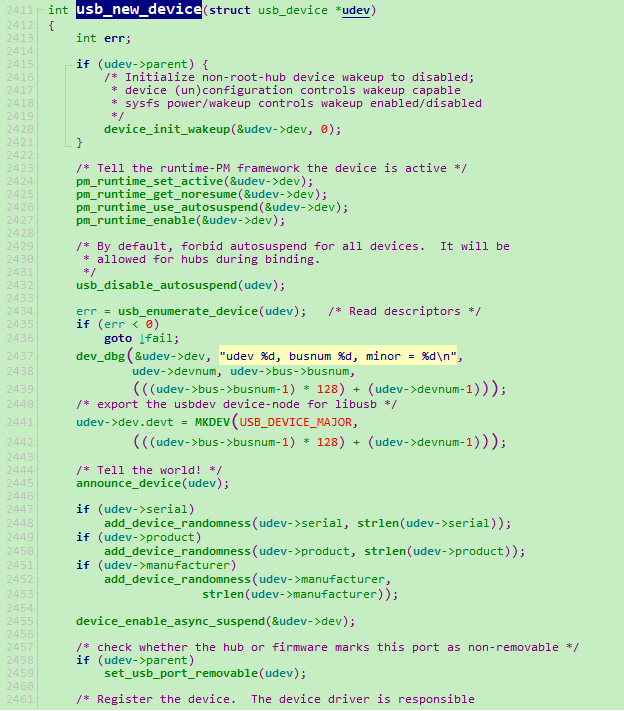


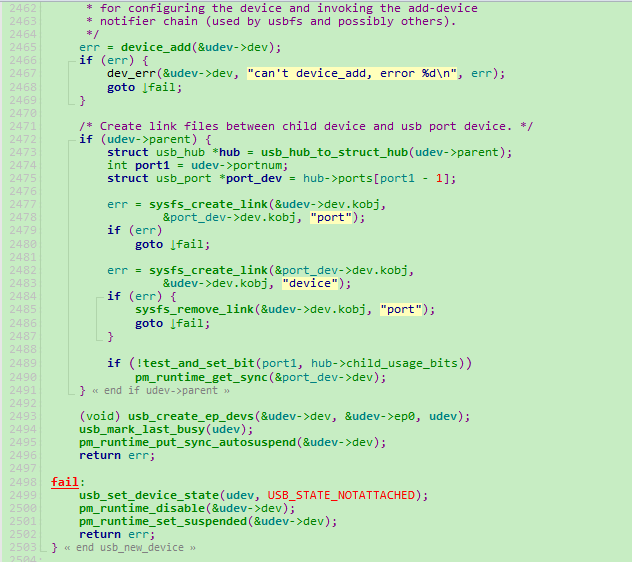




hub\_port\_init首先对port进行reset,复位成功后，设置usb设备的速度和端口0最大发送数据长度，设置usb设备的地址，并获取usb设备的设备描述符；   
通过hub\_port\_reset来reset设备，reset机制为：通过set\_port\_feature来传输USB\_PORT\_FEAT\_RESET指令，设置成功后循环延时由之前确定的时间间隔后去读取port的status和change状态，要确保usb设备在reset后还能正常存在，如reset还能正常，则通过port的status状态位来确定usb设备的速度，循环延时总时间为500ms,而usb设备reset次数为5次，   
根据usb设备的速度来初始化endpont0的最大发送数据长度，对于无线usb其maxpacketsize为512字节，对于高速和全速为64字节，而低速为8个字节；   
如果hub为高速，而usb设备为低速或全速，则在它们之间需要有一个速度转换设备tt；

通过获取usb设备的设备描述符来得到usb设备的endpoint0的最大发送数据长度，设备描述符实际上有18个字节，而endpoint0的maxpacketsize在设备描述符里的第8个字节位，所以只要获取设备描述符的前8个字节数据就可以了，但有些USB设备它并不支持只获取8个字节这样不完整的usb请求，为解决这种问题usb驱动里采用了两种策略去获取maxpacketsize，一种是windows作法，它直接向usb设备请求64字节数据，对于一些maxpacketsize为32或64的，它可以直接一次性把18个字节的设备描述符传输回来，而对于像low speed的设备它的maxpacketsize只有8个字节，就需要进行多次发送; 而Linux作法是先设置usb设备的地址，然后再发送8个字节数据请求，从返回的8个字节里获取endpoint0的maxpacketsize;

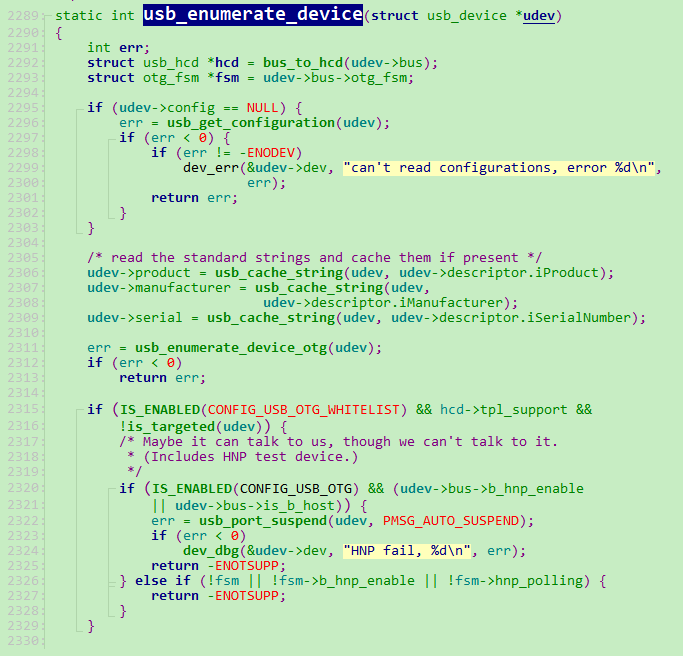




announce\_device会打印usb设备的"Product""Manufacturer""SerialNumber"

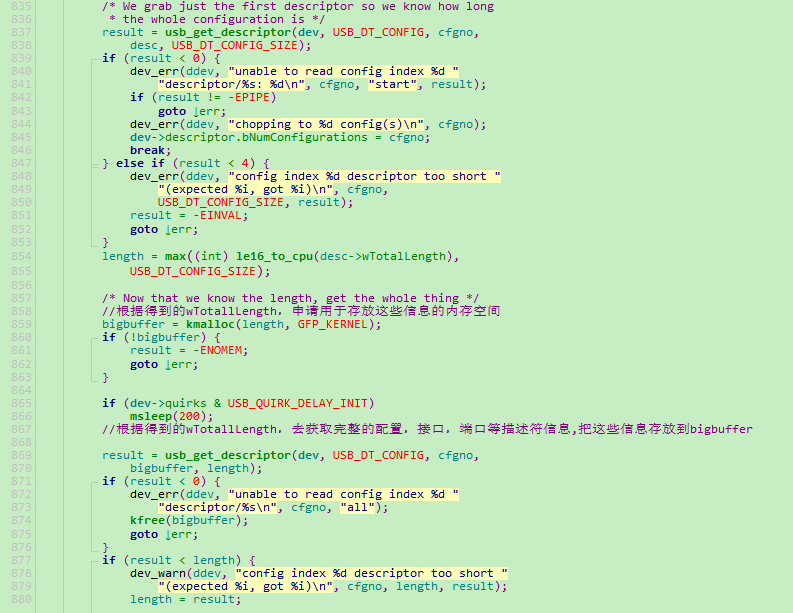
usb\_new\_device主是要获取usb设备的配置信息，然后将usb设备添加到系统中，最终执行相应驱动的probe函数。

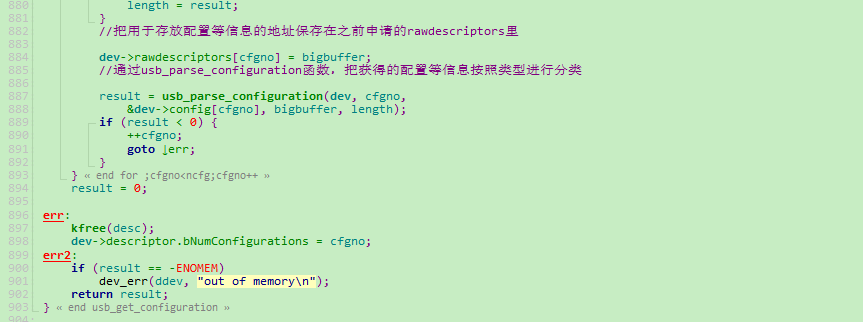
这里usb\_enumerate\_device相对比较重要，就是它完成了配置信息获取及分配；



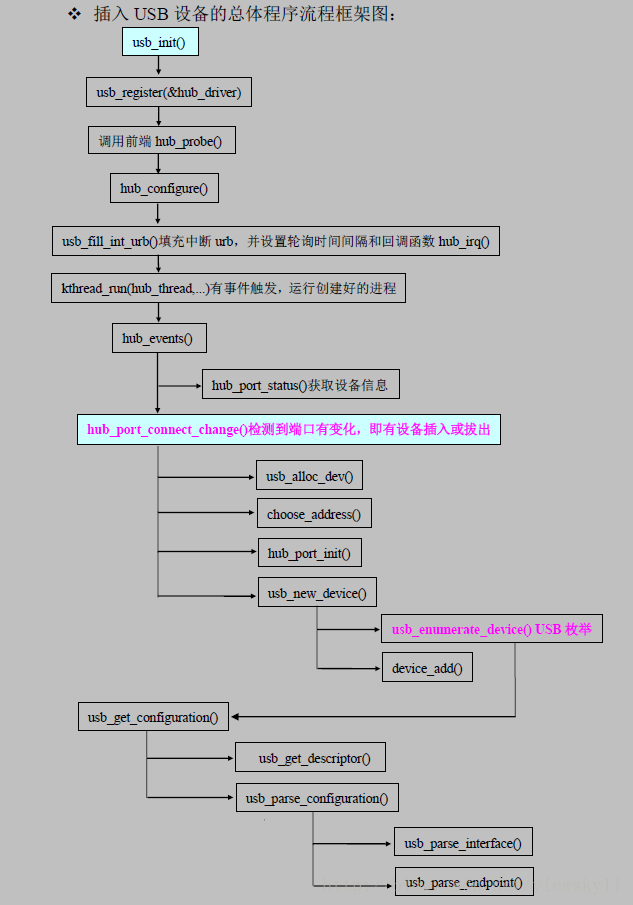
通过usb\_get\_configuration获取和分配usb设备的配置，接口，端口信息





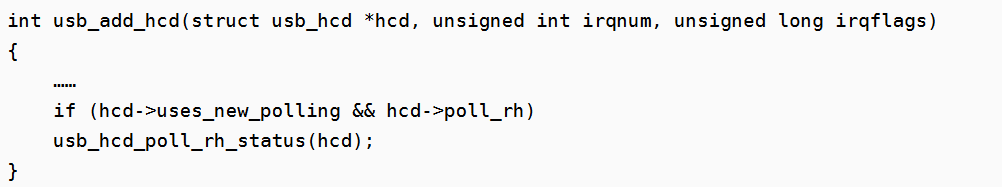


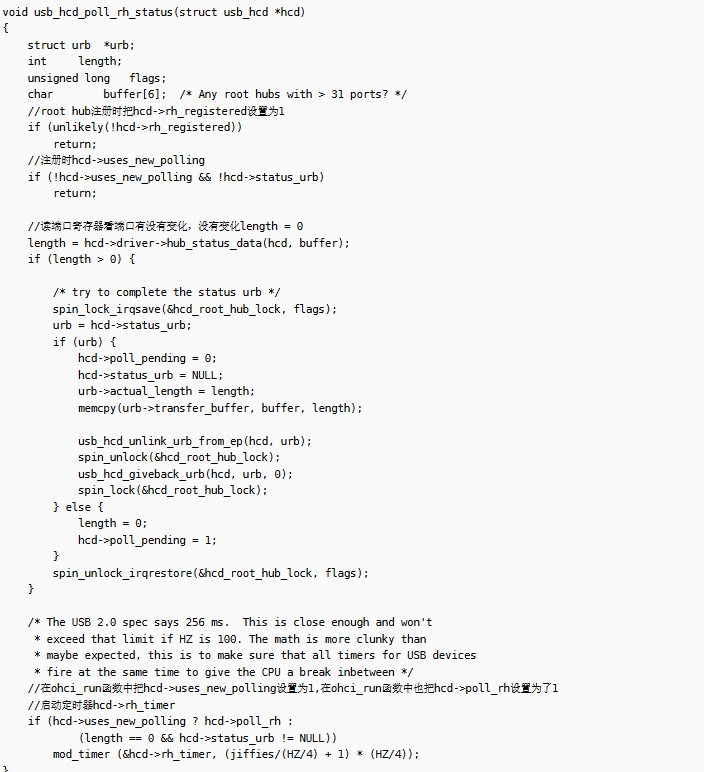
这个函数主要分成二部分，第一部分是向usb device侧获取设备的配置，接口，端口信息。首先，它为这些信息申请存放空间，然后像之前获取设备描述符一样，先发送一个9 个字节的请求，用来获取配置，接口，端口等描述符的总长度，最后根据得到的总长度去得到完整的设备配置，接口，端口信息；第二部分是把获取到的这个信息按照类别分别进行分类，并存到相应的结构中；通过usb\_parse\_configuration函数，把获得的配置等信息按照类型进行分类。



此外，如果有中断产生，调用hub\_irq。最终调用kick\_hub\_wq()。下面介绍中断。

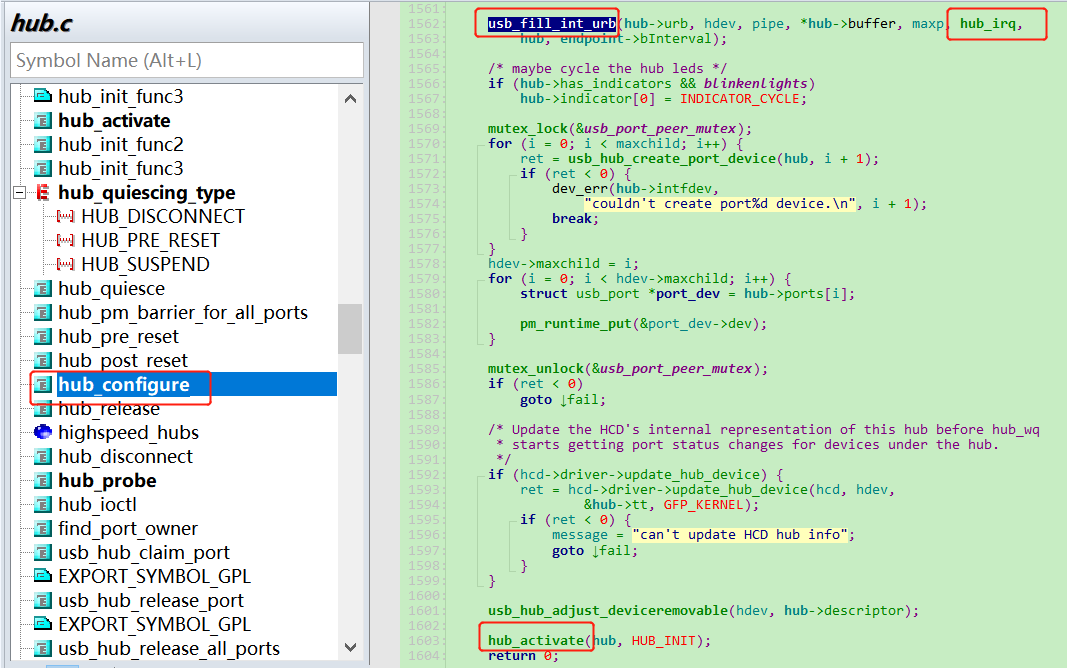
在添加主机控制器驱动时，创建了一个定时器hcd->rh\_timer，定时器的执行函数是rh\_timer\_func，这个定时器在register\_root\_hub完成后启动了。





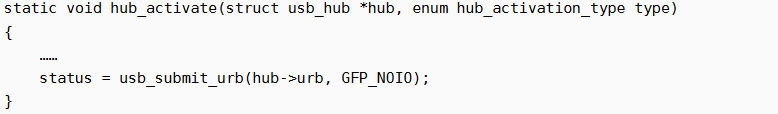
这个定时器一直读端口寄存器的状态，如果一直没有变化，这个定时器相当于空转，没做事。  
如果检测到端口寄存器状态有变化，还需要 hcd->status\_urb不等于NULL，才会执行usb\_hcd\_giveback\_urb。

在hub初始化时，在hub\_configure函数中创建了一个中断urb请求。

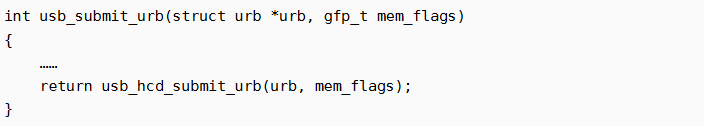


hub\_irq函数赋值给了urb->complete函数。

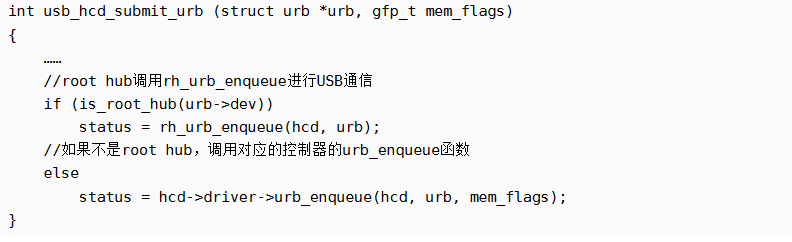
在hub\_activate函数中提交了urb请求，init3。



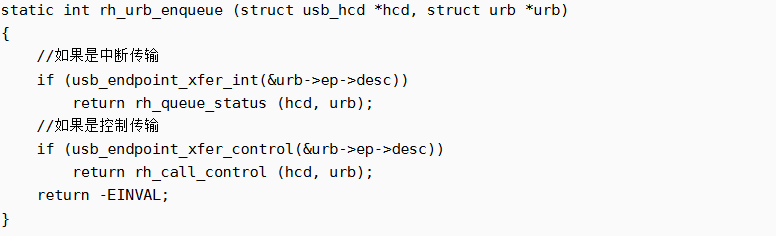
usb\_submit\_urb中调用usb\_hcd\_submit\_urb函数，将urb给HCD处理



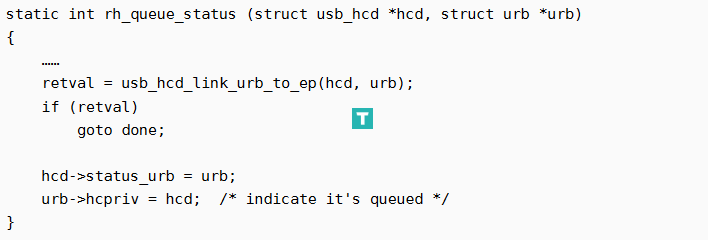
usb\_hcd\_submit\_urb函数里面，如果进来的设备是root hub，那么urb请求由rh\_urb\_enqueue函数处理，如果不是，调用主机控制器的urb\_enqueue函数。



我们这里还是root hub设备，所以调用rh\_urb\_enqueue函数。

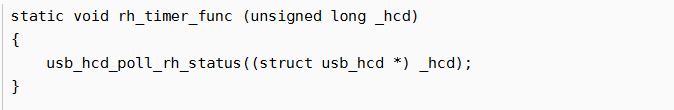


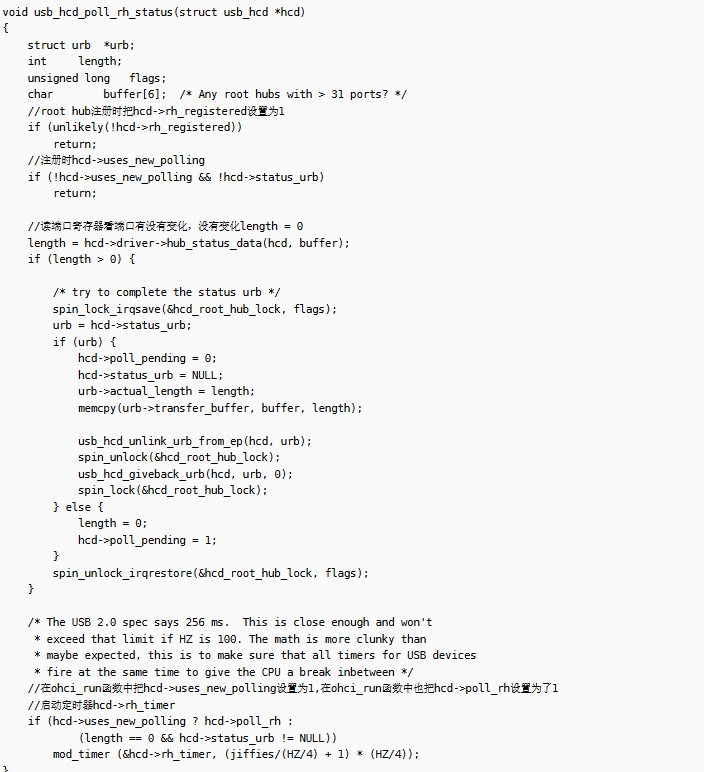
我们这里是中断传输，所以调用rh\_queue\_status函数。



上面usb\_hcd\_link\_urb\_to\_ep把urb放入urb端口的链表中。  
把urb赋值给hcd->status\_urb保存起来。这样timer的中断处理函数中hcd->status\_urb就有值了。

前面说到有一个定时器hcd->rh\_timer，执行函数是rh\_timer\_func





usb\_hcd\_poll\_rh\_status函数中，调用主机控制器的hub\_status\_data函数获取端口状态。如果端口的状态有变化，那么length > 0,把获取到的端口状态的数组拷贝到urb->transfer\_buffer中，就是前面的hub->buffer中，同时调用usb\_hcd\_giveback\_urb函数。

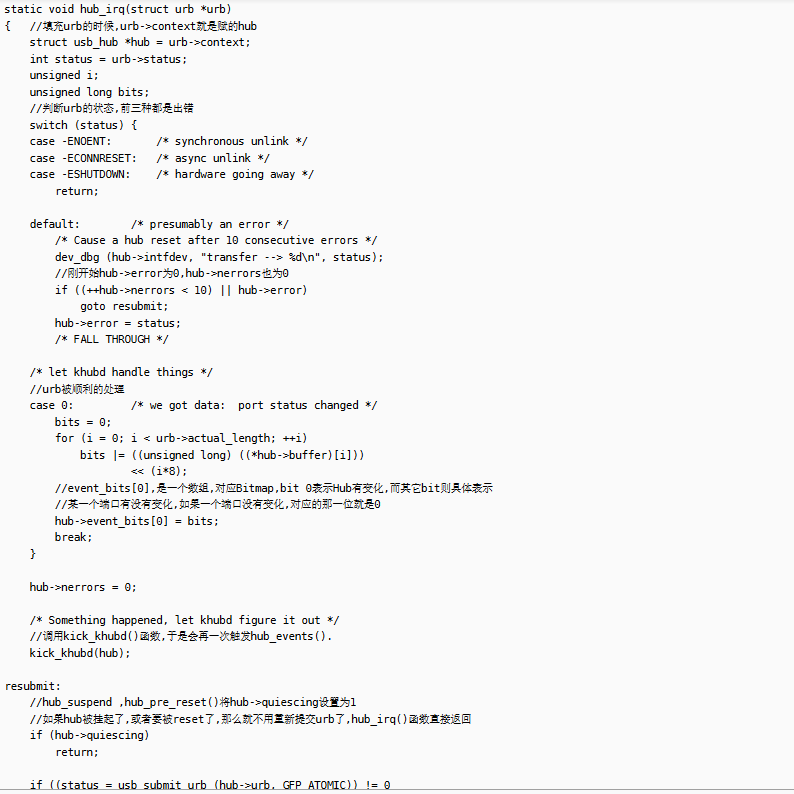
usb\_hcd\_giveback\_urb函数中调用urb->complete (urb)，而urb->complete = hub\_irq，这样就返回到了hub中。

返回之前，urb->status = 0，表示urb处理成功了。

urb->actual\_length = hcd->driver->hub\_status\_data返回的length，这个长度根据主机控制器的说明，等于1或2.

进入hub\_irq函数。

将获得的端口状态的数组存入一个long型的整数hub->event\_bits[0]中，它对应一个Bitmap，bit 0表示Hub有变化，而其它bit则具体表示某一个端口有没有变化，如果一个端口没有变化，对应的那一位就是0。我们通过按位与的方式可以知道哪一个端口发生了改变。



上图和源码有区别，其实调用了kick\_hub\_wq函数。去调度工作队列。

**hub正常工作后，主控制器就会定时询问hub是否有中断产生，当hub端口上有一个设备插入或拔除，hub就向主控制器发送urb请求，即把hub端口的变化状况告诉主控制器，这是通过urb请求来完成的，主机在处理完了这个urb后，调用该urb的complete函数,对于hub来说,这个函数就是hub\_irq().**

Root Hub interrupt transfers are polled using a timer if the driver requests it;  
也就是讲，roothub的中断传输，其实用个定时器，去访问HC的register

现在回头看看，hub驱动的探测过程中hub\_probe()中启动hub\_configure(hub, endpoint)也就是唤醒timer；