[Linux USB 3.0驱动分析（三）—— 分析USB 储存驱动程序](https://www.cnblogs.com/wen123456/p/14212660.html)

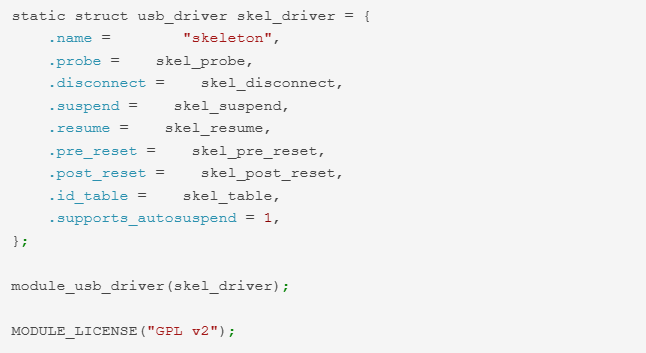
前面学习了USB驱动的一些基础概念与重要的数据结构，那么究竟如何编写一个USB 驱动程序呢？编写与一个USB设备驱动程序的方法和其他总线驱动方式类似，驱动程序把驱动程序对象注册到USB子系统中，稍后再使用制造商和设备标识来判断是否安装了硬件。当然，这些制造商和设备标识需要我们编写进USB 驱动程序中。

  USB 驱动程序依然遵循设备模型 —— 总线、设备、驱动。和I2C 总线设备驱动编写一样，所有的USB驱动程序都必须创建的主要结构体是 struct usb\_driver，它们向USB 核心代码描述了USB 驱动程序。

**"./drivers/usb/usb-skeleton.c**"是内核提供给usb设备驱动开发者的海量存储usb设备的模板程序, 程序不长， 通用性却很强，十分经典， 深入理解这个文件可以帮助我们更好的理解usb子系统以及usb设备驱动框架, 写出更好的usb海量存储设备驱动。

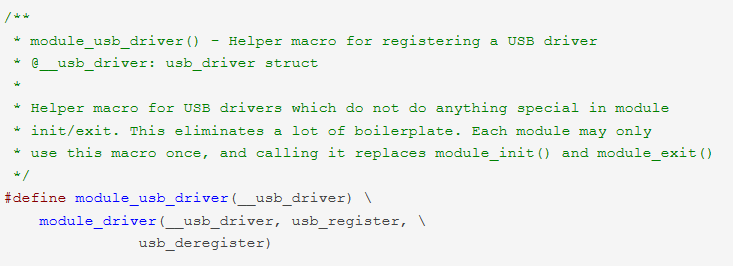
**一、注册USB驱动程序**

  Linux的设备驱动，特别是这种hotplug的USB设备驱动，会被编译成模块，然后在需要时挂在到内核。所以USB驱动和注册与正常的模块注册、卸载是一样的，下面是USB驱动的注册与卸载：



**1.module\_usb\_driver**

其中module\_usb\_driver是对usb\_register和usb\_deregister的封装的宏，用于注册和卸载usb驱动。



      USB设备驱动的模块加载函数通用的方法是在I2C设备驱动的模块加载函数中使用**usb\_register（struct \*usb\_driver）**函数添加usb\_driver的工作,而在模块卸载函数中利用**usb\_deregister（struct \*usb\_driver）**做相反的工作。 对比I2C设备驱动中的**i2c\_add\_driver(&i2c\_driver)**与**i2c\_del\_driver(&i2c\_driver)**。

**2.struct usb\_driver**

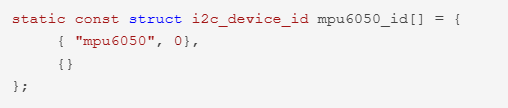
从代码看来，usb\_driver需要初始化几个字段：

模块的名字  skeleton  
probe函数   skel\_probe  
disconnect函数skel\_disconnect

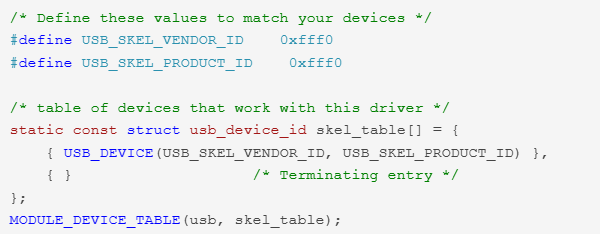
**3.id\_table**

    最重要的当然是probe函数与disconnect函数,这个在后面详细介绍，先谈一下id\_table：

    id\_table 是**struct usb\_device\_id**类型，包含了一列该驱动程序可以支持的所有不同类型的USB设备。**如果没有设置该变量，USB驱动程序中的探测回调该函数将不会被调用。**对比I2C中s**truct i2c\_device\_id \*id\_table**，一个驱动程序可以对应多个设备，i2c 示例：



usb子系统通过设备的production ID和vendor ID的组合或者设备的class、subclass跟protocol的组合来识别设备，并调用相关的驱动程序作处理。我们可以看看这个id\_table到底是什么东西：



 MODULE\_DEVICE\_TABLE的第一个参数是 **设备的类型**，如果是USB设备，那自然是usb。后面一个参数是 **设备表**， 这个设备表的最后一个元素是空的，用于标识结束。代码定义了USB\_SKEL\_VENDOR\_ID是0xfff0，USB\_SKEL\_PRODUCT\_ID是0xfff0，也就是说，当有一个设备接到集线器时，usb子系统就会检查这个设备的vendor ID和product ID，如果它们的值是0xfff0时，那么子系统就会调用这个skeleton模块作为设备的驱动。

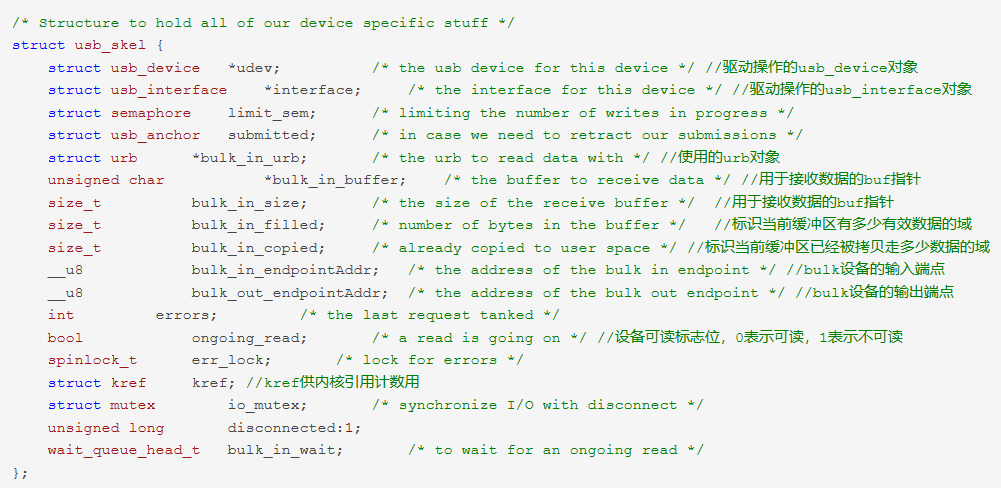
   当USB设备接到USB控制器接口时，usb\_core就检测该设备的一些信息，例如生产厂商ID和产品的ID，或者是设备所属的class、subclass跟protocol，以便确定应该调用哪一个驱动处理该设备。

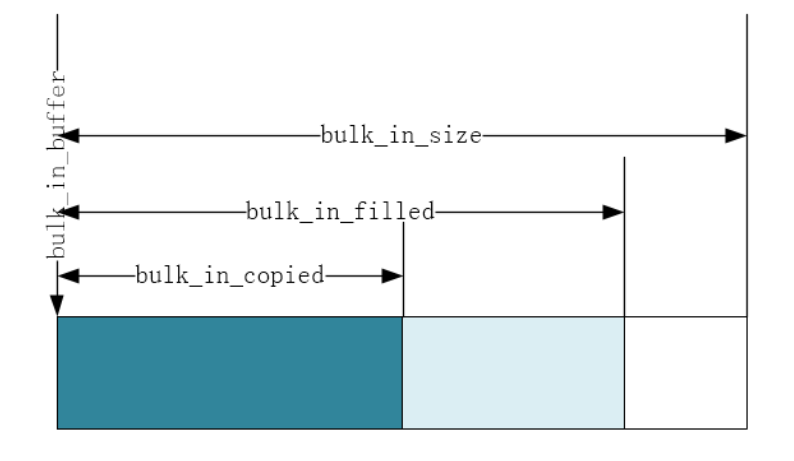
**二、USB驱动程序中重要数据结构**

**我们下面所要做的就是对probe函数与disconnect函数的填充了，但是在对probe函数与disconnect函数填充之前，有必要先学习三个重要的数据结构，这在我们后面probe函数与disconnect函数中有很大的作用：**

**1、usb-skeleton**

       usb-skeleton 是一个局部结构体，用于与端点进行通信。下面先看一下Linux内核源码中的一个usb-skeleton（就是usb驱动的骨架咯），其定义的设备结构体就叫做usb-skel：





**三.probe函数分析**

当platform driver与device platform匹配成功的时候，就会调用probe。这里调用skel\_probe，下面流程简化了



**三、探测和断开函数分析**

        USB驱动程序指定了两个USB核心在适当时间调用的函数。

**1、探测函数usb\_find\_common\_endpoints**

        当一个设备被安装而USB核心认为该驱动程序应该处理时，探测函数被调用；

**探测函数应该检查传递给他的设备信息**，**确定驱动程序是否真的适合该设备**。当驱动程序因为某种原因不应控制设备时，断开函数被调用，它可以做一些清洁的工作。

      系统会传递给探测函数的信息是什么呢？一个**usb\_interface \***跟一个**struct usb\_device\_id \***作为参数。他们分别是该**USB设备的接口描述**（一般会是该设备的第0号接口，该接口的默认设置也是第0号设置）跟它的**设备ID**描述（包括Vendor ID、Production ID等）。

**USB驱动程序应该初始化任何可能用于控制USB设备的局部结构体，它还应该把所需的任何设备相关信息保存到局部结构体中。例如，USB驱动程序通常需要探测设备对的端点地址和缓冲区大小，因为需要他们才能和端点通信**。

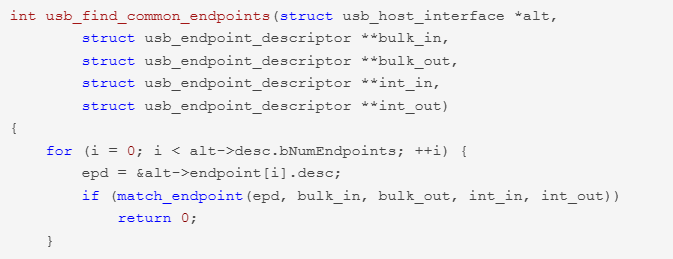
       下面具体分析探测函数做了哪些事情：

**a -- 探测设备的端点地址、缓冲区大小，初始化任何可能用于控制USB设备的数据结构**

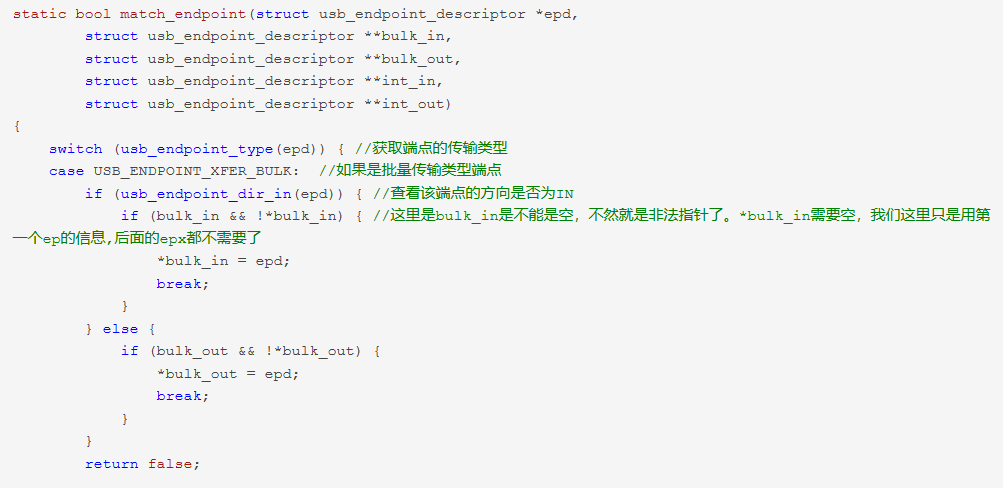
       下面是一个实例代码，他们探测批量类型的IN和OUT端点：

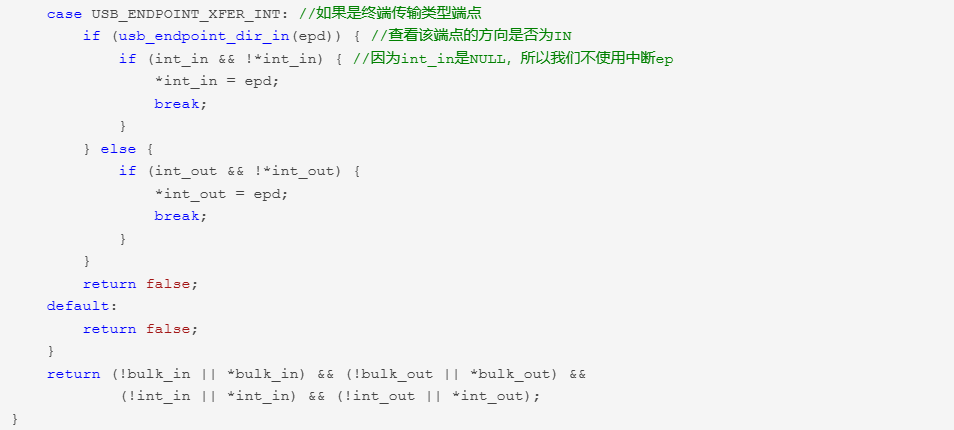
具体流程如下：

     该代码块首先循环访问该接口中存在的每一个端点，赋予该端点结构体的局部指针以使稍后的访问更加容易



       我们有了一个端点，我们测定该端点类型是否批量，这首先通过USB\_ENDPOINT\_XFERTYPE\_MASK 位掩码来取bmAttributes变量的值，然后检查它是否和USB\_ENDPOINT\_XFER\_BULK 的值匹配来完成。  然后查看该端点的方向是否为IN。这可以通过检查位掩码 USB\_DIR\_IN 是否包含在bEndpointAddress 端点变量中来确定。





        如果这些都通过了，驱动程序就知道它已经发现了正确的端点类型，可以把该端点相关的信息保存到一个局部结构体中，就是我们前面的usb\_skel ,以便稍后使用它和端点进行通信.

**b -- 把已经初始化数据结构的指针保存到接口设备中**

      接下来的工作是向系统注册一些以后会用的的信息。首先我们来说明一下**usb\_set\_intfdata()**，**他向内核注册一个data，这个data的结构可以是任意的，这段程序向内核注册了一个usb\_skel结构，就是我们刚刚看到的被初始化的那个，这个data可以在以后用usb\_get\_intfdata来得到**

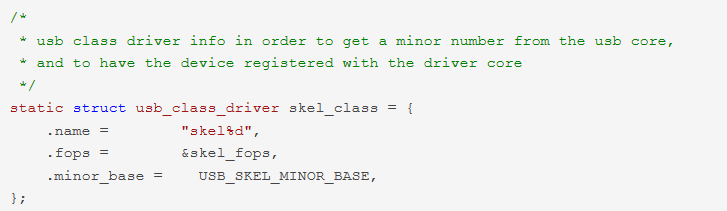
**usb\_set\_intfdata(interface, dev);**

**c -- 注册USB设备**

如果USB驱动程序没有和处理设备与用户交互（例如输入、tty、视频等）的另一种类型的子系统相关联，驱动程序可以使用USB主设备号，以便在用户空间使用传统的字符驱动程序接口。如果要这样做，USB驱动程序必须在探测函数中调用 usb\_resgister\_dev 函数来把设备注册到USB核心。只要该函数被调用，就要确保设备和驱动程序都处于可以处理用户访问设备的要求的恰当状态

**retval = usb\_register\_dev(interface, &skel\_class);**

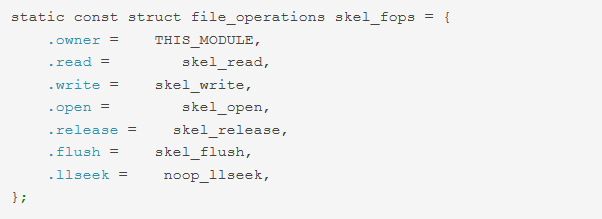
skel\_class结构。这个结构又是什么？我们就来看看这到底是个什么东西：



        它其实是一个系统定义的结构，里面包含了一名字、一个文件操作结构体还有一个次设备号的基准值。事实上它才是定义真正完成对设备IO操作的函数。所以他的核心内容应该是skel\_fops。

   因为usb设备可以有多个interface，每个interface所定义的IO操作可能不一样，所以向系统注册的usb\_class\_driver要求注册到某一个interface，而不是device，因此，usb\_register\_dev的第一个参数才是interface，而第二个参数就是某一个usb\_class\_driver。

   通常情况下，linux系统用主设备号来识别某类设备的驱动程序，用次设备号管理识别具体的设备，驱动程序可以依照次设备号来区分不同的设备，所以，这里的次设备号其实是用来管理不同的interface的，但由于这个范例只有一个interface，在代码上无法求证这个猜想。



**2、断开函数**

      当设备被拔出集线器时，usb子系统会自动地调用disconnect，他做的事情不多，最重要的是注销class\_driver（交还次设备号）和interface的data，取消:



**四、USB请求块**

       USB 设备驱动代码通过urb和所有的 USB 设备通讯。urb用 struct urb 结构描述（include/linux/usb.h ）。

       urb 以一种异步的方式同一个特定USB设备的特定端点发送或接受数据。一个 USB 设备驱动可根据驱动的需要，分配多个 urb 给一个端点或重用单个 urb 给多个不同的端点。设备中的每个端点都处理一个 urb 队列, 所以多个 urb 可在队列清空之前被发送到相同的端点。

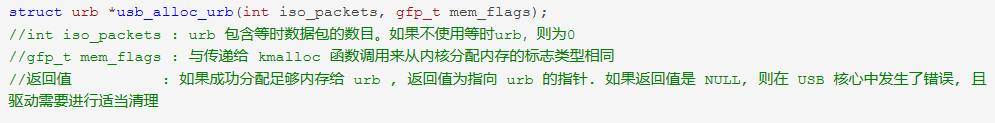
 一个 urb 的典型生命循环如下:

 （1）被创建；  
 （2）被分配给一个特定 USB 设备的特定端点；  
 （3）被提交给 USB 核心；  
 （4）被 USB 核心提交给特定设备的特定 USB 主机控制器驱动；  
 （5）被 USB 主机控制器驱动处理, 并传送到设备；  
 （6）以上操作完成后，USB主机控制器驱动通知 USB 设备驱动。  
   
   urb 也可被提交它的驱动在任何时间取消；如果设备被移除，urb 可以被USB核心取消。urb 被动态创建并包含一个内部引用计数，使它们可以在最后一个用户释放它们时被自动释放。

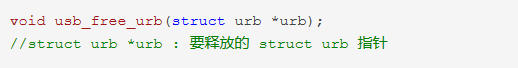


**1、创建和注销 urb**

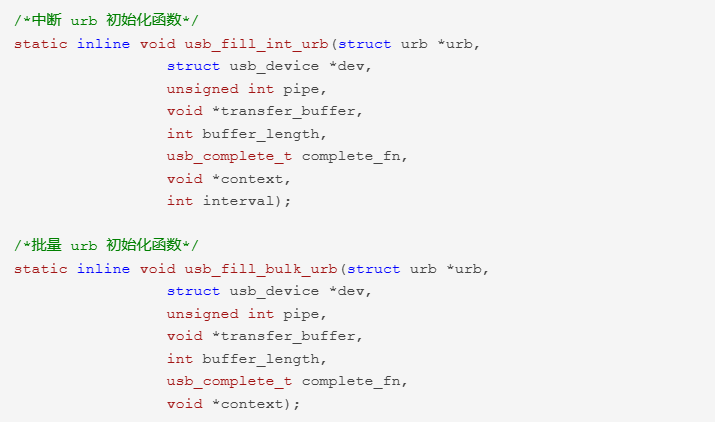
struct urb 结构不能静态创建,必须使用 usb\_alloc\_urb 函数创建. 函数原型:

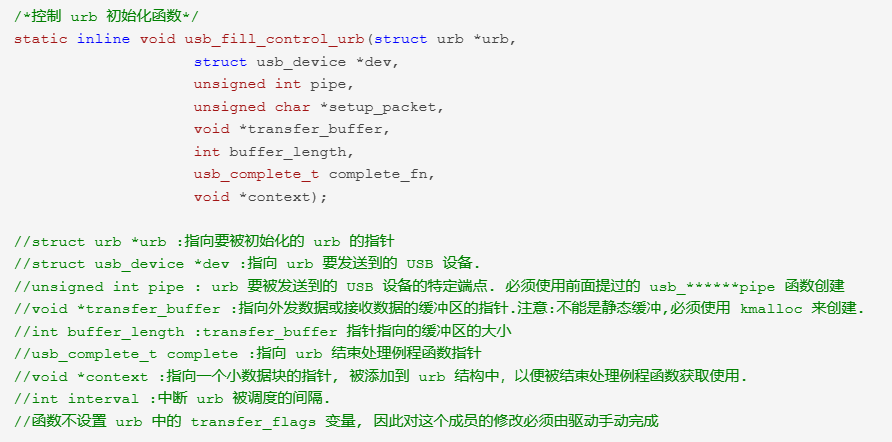


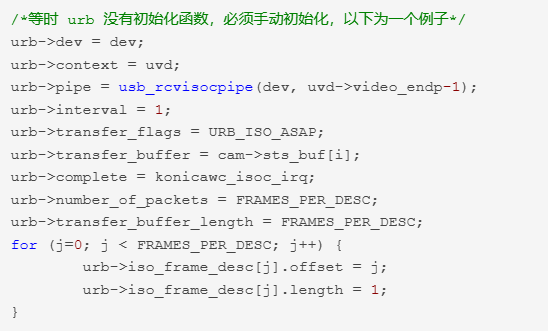
如果驱动已经对 urb 使用完毕, 必须调用 usb\_free\_urb 函数，释放urb。函数原型:



**2、初始化 urb**

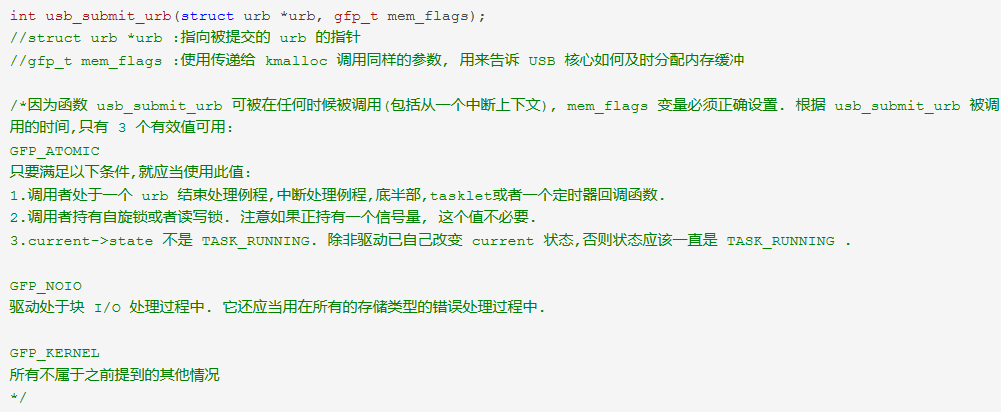






**3、提交 urb**

    一旦 urb 被正确地创建并初始化, 它就可以提交给 USB 核心以发送出到 USB 设备. 这通过调用函数 usb\_submit\_urb 实现:



在 urb 被成功提交给 USB 核心之后, 直到结束处理例程函数被调用前,都不能访问 urb 结构的任何成员.

**4、urb结束处理例程**

      如果 usb\_submit\_urb 被成功调用, 并把对 urb 的控制权传递给 USB 核心, 函数返回 0; 否则返回一个负的错误代码. 如果函数调用成功, 当 urb 被结束的时候结束处理例程会被调用一次.当这个函数被调用时, USB 核心就完成了这个urb, 并将它的控制权返回给设备驱动.

只有 3 种结束urb并调用结束处理例程的情况:

(1)urb 被成功发送给设备, 且设备返回正确的确认.如果这样, urb 中的status变量被设置为 0.  
(2)发生错误, 错误值记录在 urb 结构中的 status 变量.  
(3)urb 从 USB 核心unlink. 这发生在要么当驱动通过调用 usb\_unlink\_urb 或者 usb\_kill\_urb告知 USB 核心取消一个已提交的 urb,或者在一个 urb 已经被提交给它时设备从系统中去除.

**5、取消 urb**

使用以下函数停止一个已经提交给 USB 核心的 urb:

void usb\_kill\_urb(struct urb \*urb)  
int usb\_unlink\_urb(struct urb \*urb);

如果调用usb\_kill\_urb函数,则 urb 的生命周期将被终止. 这通常在设备从系统移除时,在断开回调函数(disconnect callback)中调用.

对一些驱动, 应当调用 usb\_unlink\_urb 函数来使 USB 核心停止 urb. 这个函数不会等待 urb 完全停止才返回. 这对于在中断处理例程中或者持有一个自旋锁时去停止 urb 是很有用的, 因为等待一个 urb 完全停止需要 USB 核心有使调用进程休眠的能力(wait\_event()函数).

**五.usb读取**

在probe函数中注册了file\_operations结构体skel\_fops用于应用访问驱动。应用open之后就可以调用read和write对设备进行读写了。

**1.skel\_read**

首先是skel\_read()，这个函数是应用层读设备时回调的函数，它试图实现这样一个功能: 如果内核缓冲区有数据就将适当的数据拷贝给应用层, 如果没有就调用skel\_do\_read\_io来向设备请求数据。

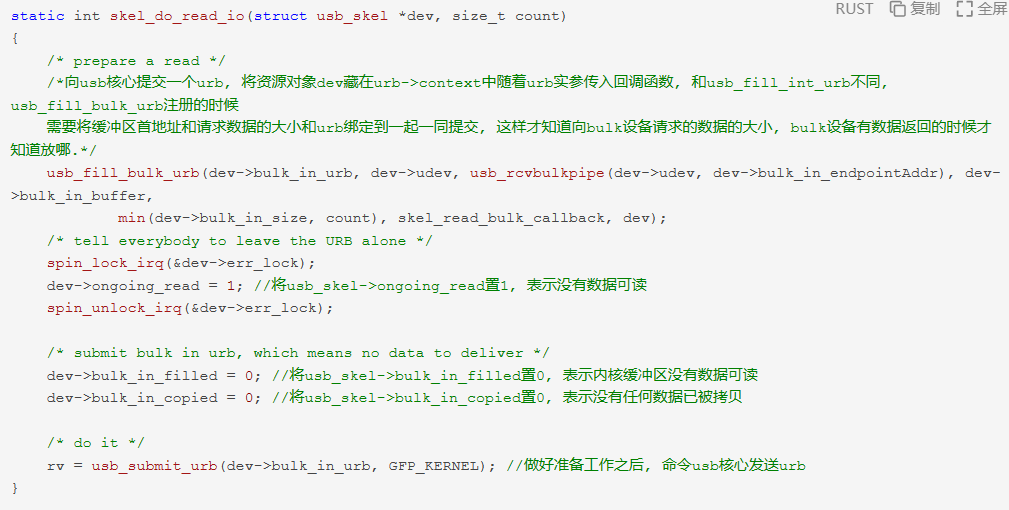






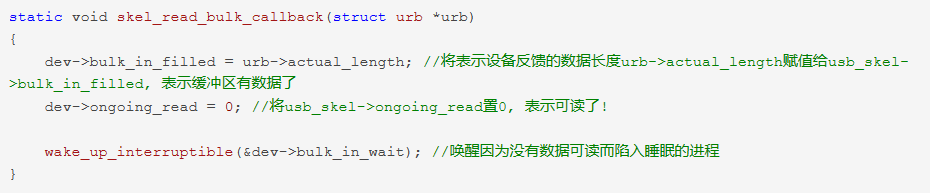
**2.skel\_do\_read\_io**

刚才也说了, 如果缓冲区不能满足应用层需求的时候, 就会调用下面这个函数向bulk usb设备请求数据, 得到数据后将数据放到缓冲区并将相应的标志位置1/置0



**3.skel\_read\_bulk\_callback**

请求被发出后, usb总线就会静待设备的反馈, 设备有反馈后就会回调urb的注册函数, 我们看看这个回调函数都做了什么



**六.USB写入**

**写入数据的思路是一样的, 我这里就不罗嗦了.**