

USB core

USB tree

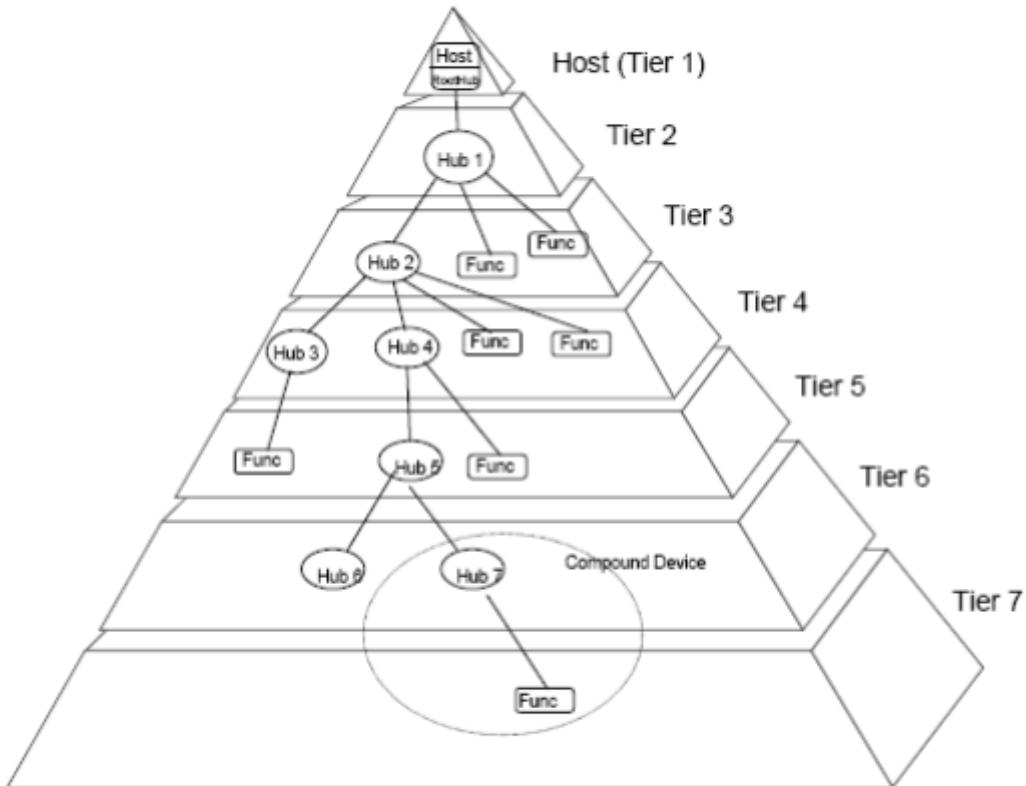


Figure 4-1. Bus Topology

USB device 还包括了 hub 和功能设备，也就是上图里的 Func

USB controller

在一个 USB 系统中只能有一个 host，其实说白了就是咱们的主机，而 USB 和主机的接口就是 host controller，一个主机可以支持多个 host controller，比如分别属于不同厂商的。那么 USB host controller 本身是做什么的？controller，控制器，顾名思义，用于控制，控制什么，控制所有的 usb 设备的通信。通常计算机的 cpu 并不是直接和 usb 设备打交道，而是和控制器打交道，他要对设备做什么，他会告诉控制器，而不是直接把指令发给设备，然后控制器再去负责处理这件事情，他会去指挥设备执行命令，而 cpu 就不用管剩下的事情，他还是该干嘛干嘛去，控制器替他去完成剩下的事情，事情办完了再通知 cpu。

hub

在大学里，有的宿舍里网口有限，但是我们这一代人上大学基本上是每人一台电脑，所以网口不够，于是有人会使用 hub，让多人共用一个网口，这是以太网上的 hub，而 usb 的世界里同样有 hub，其实原理是一样的，任何支持 usb 的电脑不会说只允许你只能一个时刻使用一个 usb 设备，比如你插入了 u 盘，你同样还可以插入 usb 键盘，还可以再插一个 usb 鼠标，因为你会发现你的电脑里不只是一个 usb 接口。这些口实际上就是所谓的 hub 口。而现实中经常是让一个 usb 控制器和一个 hub 绑定在一起，专业一点说叫集成，而这个 hub 也被称作 **root hub**，换言之，和 usb 控制器绑定在一起的 hub 就是系统中最根本的 hub，其它的 hub 可以连接到她这里，然后可以延伸出去，外接别的设备，当然也可以不用别的 hub，让 usb 设备直接接到 root hub 上。

USB 连接

USB 连接指的就是连接 device 和 host (或 hub) 的四线电缆。电缆中包括 VBUS (电源线)、GND (地线) 还有两根信号线。USB 系统就是通过 VBUS 和 GND 向设备提供电源的。USB 设备可以使用主机提供的电源，也可以使用外接电源供电。



Figure 4-2. USB Cable

USB 通信

USB 通信最基本的形式是通过 USB 设备里一个叫 endpoint 的东东，而主机和 endpoint 之间的数据传输是通过 pipe。endpoint 就是通信的发送或者接收点，你要发送数据，那你只要把数据发送到正确的端点那里就可以了。端点也是有方向的，从 usb 主机到设备称为 out 端点，从设备到主机称为 in 端点。而管道，实际上只是让我们能够找到端点，就相当于我们日常说的邮编地址，比如一个国家，为了通信，我们必须给各个地方取名，完了 给各条大大小小的路取名，比如你要揭发你们那里的官员腐败，你要去上访，你从某偏僻的小县城出发，要到北京来上访，那你的这个端点(endpoint)就是北京，而你知道你来北京的路线，那这个从你们县城到北京的路线就算一条管道。有人好奇的问了，管道应该有两端吧，一个端点是北京，那另一个端点呢?答案是，这条管道有些特殊，就比如上访，我们只需要知道一端是北京，而另一端是哪里无所谓，因为不管你在哪你都得到北京来上访。没听说过你在山西你可以上访，你要在宁夏还不能上访了，没这事对吧。严格来说，管道的另一端应该是 usb 主机，即前面说的那个 host，usb 协议里边也是这么说的，协议里边说 pipes 代表着一种能力，怎样一种能力呢，在主机和设备上的端点之间移动数据，听上去挺玄的。

端点

端点不但是有方向的，而且这个方向还是确定的，或者 in，或者 out，没有又是 in 又是 out 的，鱼与熊掌是不可兼得的，脚踩两只船虽然是每个男人的美好愿望，但不具可操作性，也不提倡。所以你到北京就叫上访，北京的下来就叫慰问，这都是生来就注定的。有没有特殊的那，看你怎么去理解 0 号端点了，协议里规定了，所有的 USB 设备必须具有端点 0，它可以作为 in 端点，也可以作为 out 端点，USB 系统软件利用它来实现缺省的控制管道，从而控制设备。端点也是限量供应的，不是想有多少就有多少的，除了端点 0，低速设备最多只能拥有 2 个端点，高速设备也最多只能拥有 15 个 in 端点和 15 个 out 端点。这些端点在设备内部都有唯一的端点号，这个端点号是在设备设计时就已经指定的。

0号端点

为什么端点 0 就非要那么的个性那？这还是有内在原因的。管道的通信方式其实有两种，一种是 stream 的，一种是 message 的，message 管道要求从它那儿过的数据必须具有一定的格式，不是随便传的，因为它主要就是用于主机向设备请求信息的，必须得让设备明白请求的是什么。而 stream 管道就没这么苛刻，要随和多了，它对数据没有特殊的要求。协议里说，message 管道必须对应两个相同号码的端点，一个用来 in，一个用来 out，咱们的缺省管道就是 message 管道，当然，与缺省管道对应的端点 0 就必须是两个具有同样端点号 0 的端点。

端点类型

USB endpoint 有四种类型，也就分别对应了四种不同的数据传输方式。它们是控制传输 Control Transfers，中断传输 Interrupt Data Transfers，批量传输 Bulk Data Transfers，等时传输 Isochronous Data Transfers。

控制传输

控制传输用来控制对 USB 设备不同部分的访问，通常用于配置设备，获取设备信息，发送命令到设备，或者获取设备的状态报告。总之就是用来传送控制信息的，每个 USB 设备都会有一个 endpoint 0 的控制端点，内核里的 USB core 使用它在设备插入时进行设备的配置。这么说吧，君士坦丁旁边有非常信赖的这么一个人，往往通过他来对其他人做些监控迫害什么的，虽然他最后被判了君士坦丁，但我们的 endpoint 0 不会，它会一直在那里等待着 USB core 发送控制命令。最不忠诚的往往是人心，不是么。

中断传输

中断传输用来以一个固定的速率传送少量的数据，USB 键盘和 USB 鼠标使用的就是这种方式，USB 的触摸屏也是，传输的数据包含了坐标信息。

批量传输

批量传输用来传输大量的数据，确保没有数据丢失，并不保证在特定的时间内完成。U 盘使用的就是批量传输，咱们用它备份数据时需要确保数据不能丢，而且也不能指望它能在一个固定的时间内拷贝完。

等时传输

等时传输同样用来传输大量的数据，但并不保证数据是否到达，以稳定的速率发送和接收实时的信息，对传送延迟非常敏感。显然是用于音频和视频一类的设备，这类设备期望能够有个比较稳定的数据流，比如你在网上 QQ 视频聊天，肯定希望每分钟传输的图像/声音速率是比较稳定的，不能说这一分钟对方看到你在向她向你深情表白，可是下一分钟却看见画面停滞在那里，只能看到你那傻样一动不动，你说这不浪费感情嘛。

USB 系统实现

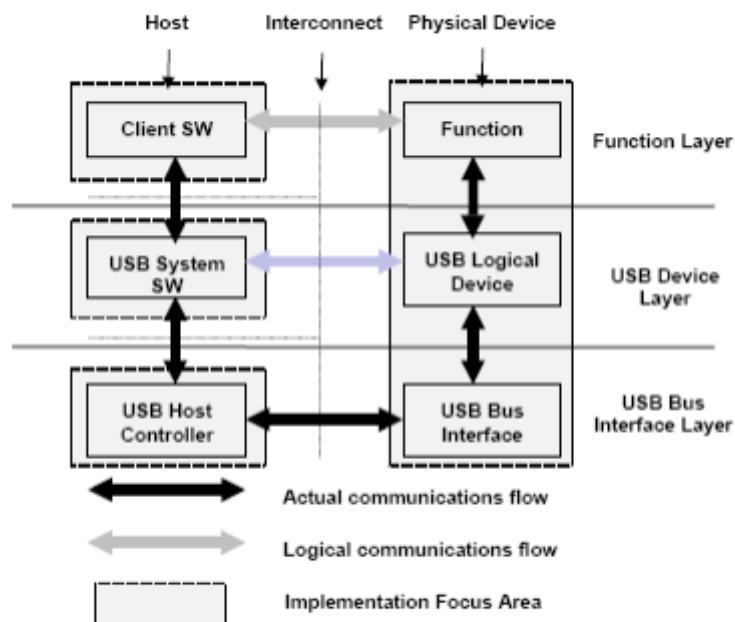


Figure 5-2. USB Implementation Areas

一个完整的 USB 系统应该实现上面图里的各个部分，USB 协议如是说。图里主要显示了四个层次，USB 物理设备（USB Physical Device）、客户软件（Client SW）、USB 系统软件（USB System SW）以及 USB Host Controller。Host Controller 已经说过了，系统软件就是操作系统里用来支持 USB 的部分，像咱们的 usb core，还有各种设备驱动等等，客户软件么，就是上层应用了，只有设备和驱动程序，我们仍然什么都做不了，现实生活中一个很浅显的道理就是只靠摄像头和驱动是不可能和 mm 视频的，不是么，这是个应用为王的时代。

实际上

上面的**系统软件**，只表示了系统里支持 USB 的部分，也就是系统相关设备无关的部分，相对于咱们的 linux 来说，就是 usb core，并不包括各种设备驱动。而**客户软件**则指设备相关，也就是对应于特定设备的部分，你的 USB 键盘驱动、触摸屏驱动什么的都在这儿。这里的名字太迷惑人了，一直觉得写驱动是系统级的编程，原来搞协议的这些同志觉得不是这么回事，我羞愧的低下了无知的脑袋。

主机这边就分这三层，**Host Controller** 看似在最低层，却掌控着整个 USB 的通信，你的 USB 设备要想发挥作用，首先得获得它的批准，此路是它开，要想从此过，留下买路财。我们也在最底层，不同的是被掌握，不同的角色决定了不同的命运。

USB 物理设备这边看着好像也分了三层，其实我们可以把它们看成一样的东东，只是为了对应了主机这边的不同层次，Host Controller 看到的是一个个 hub 还有 hub 上的 USB device，而在系统软件的眼里没那么多道道儿，hub 还有各种设备什么的都是一个个的 usb 逻辑设备，客户软件看到的是设备提供的功能。接下来还会有说到。站在不同的高度看到不同的风景，不然为什么买房子时高一层要加多少钱那。

已经被计算机网络中的七层协议洗过脑的我们应该很容易的就看出，**真实的数据流只发生在 Host Controller 和设备的 Bus Interface 之间**，其它的都是**逻辑上的**，也就是说是虚的，如果谁对我们说什么什么是逻辑上存在的，那它肯定就是虚的，比如说任小强逻辑上给你了一套北京的房子，你相信么，给是给，得掏钱，而且还得掏的多。

各种 USB 设备提供的功能是不同的，但是面向主机的**Bus Interface** 却是一致的，**主机也不是神仙**，掐指一算就可以知道哪个是哪个，所以，那些**设备本身还必须要提供用来确认自己身份的信息**，这些信息里有些是共有的，有些是个别设备特有的，我们都是光荣的中国公民，但是有的人是卖房子，有的人买房子。

compound device

compound device 是那些将 hub 和连在 hub 上的设备封装在一起所组成的设备

composite device

是包含彼此独立的多个接口的设备

从主机的角度看，一个 compound device 和单独的一个 hub 然后连接了多个 USB 设备是一样的，它里面包含的 hub 和各个设备都会有自己的独立的地址，而一个 composite device 里不管具有多少接口，它都只有一个地址。

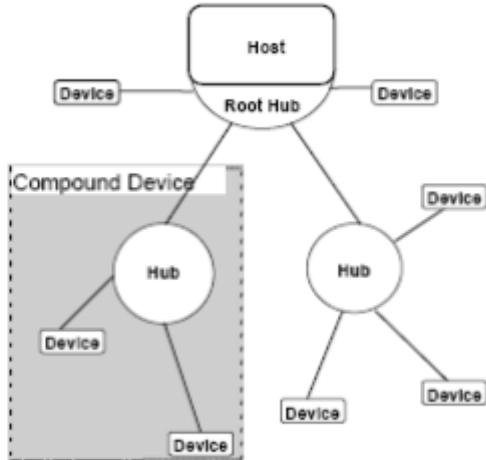


Figure 5-5. USB Physical Bus Topology

上面都是站在 host controller 的层次上，说的是实实在在的物理拓扑，对于系统软件来说，没有这么复杂，所有的 hub 和设备都被看作是一个个的逻辑设备，好像它们本来就直接连接在 root hub 上一样。站的越高，看的越远，快乐如此简单，可以做售楼广告了。

USB逻辑设备

一个 USB 逻辑设备就是一群端点 (endpoint) 的集合，它与主机之间的通信发生在主机上的一个缓冲区和设备上的一个端点之间，通过管道来传输数据。意思就是管道的一端是主机上的一个缓冲区，一端是设备上的端点。

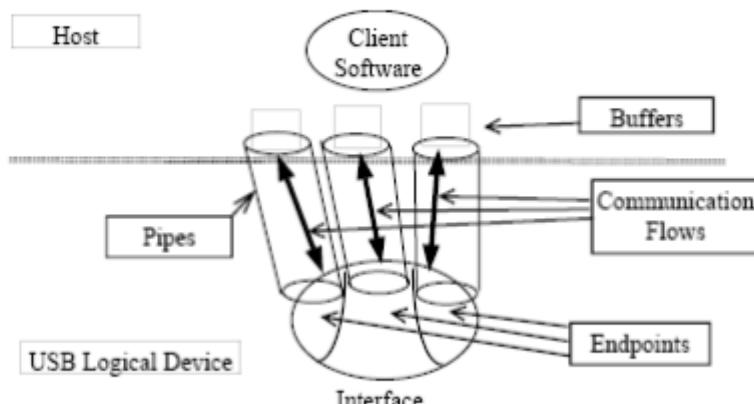


Figure 5-10. USB Communication Flow

Interface

图里的 Interface 是怎么回事？这里先简单说说吧，反正代码里会不停的遇到再遇到。USB 端点被捆绑为接口 (Interface)，一个接口代表一个基本功能。有的设备具有多个接口，像 USB 扬声器就包括一个键盘接口和一个音频流接口。在内核里一个接口要对应一个驱动程序，USB 扬声器在 linux 里就需要两个不同的驱动程序。到目前为止，可以这么说，一个设备可以包括多个接口，一个接口可以具有多个端点，当然以后我们会发现并不仅仅止于此。不过先说这么多吧，省得说得慷慨激昂，看的昏昏欲睡。

usb core

Linux 内核开发者们，专门写了一些代码，负责实现一些核心的功能，为别的设备驱动程序提供服务，比如申请内存，比如实现一些所有的设备都会需要的公共的函数，并美其名曰 usb core。

在 drivers/usb/目录下面出来了一个 core 目录，就专门放一些核心的代码，比如初始化整个 usb 系统，初始化 root hub，初始化 host controller 的代码，再后来甚至把 host controller 相关的代码也单独建了一个目录，叫 host 目录，这是因为 usb host controller 随着时代的发展，也开始有了好几种，不再像刚开始那样只有一种，所以呢，设计者们把一些 host controller 公共的代码仍然留在 core 目录下，而一些各 host controller 单独的代码则移到 host 目录下面让负责各种 host controller 的人去维护。

usb core 负责实现一些核心的功能，为别的设备驱动程序提供服务，提供一个用于访问和 20 控制 USB 硬件的接口，而不用去考虑系统当前存在哪种 host controller。

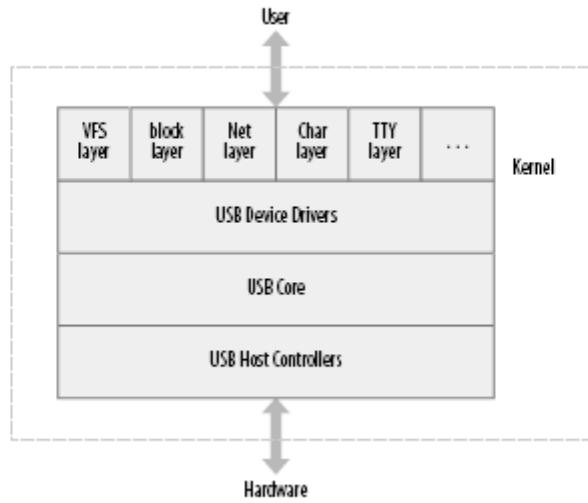
usb gadget

gadget 白了说就是配件的意思，主要就是一些内部运行 linux 的嵌入式设备，如 PDA，设备本身有 USB 设备控制器（usb device controller），可以将 PC，也就是我们的 host 作为 master 端，将这样的设备作为 slave 端和 PC 通过 USB 进行通信。从 host 的观点来看，主机系统的 USB 驱动程序控制插入其中的 USB 设备，而 usb gadget 的驱动程序控制外围设备如何作为一个 USB 设备和主机通信。比如，我们的嵌入式板子上支持 SD 卡，如果我们希望在将板子通过 USB 连接到 PC 之后，这个 SD 卡被模拟成 U 盘，那么就要通过 usb gadget 架构的驱动。

gadget 目录下大概能够分为两个模块，一个是 udc 驱动，这个驱动是针对具体 cpu 平台的，如果找不到现成的，就要自己实现。另外一个就是 gadget 驱动，主要有 file_storage、ether、serial 等。另外还提供了 USB gadget API，即 USB 设备控制器硬件和 gadget 驱动通信的接口。PC 及服务器只有 USB 主机控制器硬件，它们并不能作为 USB gadget 存在，而对于嵌入式设备，USB 设备控制器常被集成到处理器中，设备的各种功能，如 U 盘、网卡等，常依赖这种 USB 设备控制器来与主机连接，并且设备的各种功能之间可以切换，比如可以根据选择作为 U 盘或网卡等。

其他目录

剩下的几个目录分门别类的放了各种 USB 设备的驱动，U 盘的驱动在 storage 目录下，触摸屏和 USB 键盘鼠标的驱动在 input 目录下，等等。多说一下的是，Usb 协议中，除了通用的软硬件电气接口规范等，还包含了各种各样的 Class 协议，用来为不同的功能定义各自的标准接口和具体的总线上的数据交互格式和内容。这些 Class 协议的数量非常多，最常见的比如支持 U 盘功能的 Mass Storage Class，以及通用的数据交换协议：CDC class。此外还包括 Audio Class, Print Class 等等。理论上说，即使没有这些 Class，通过专用驱动也能实现各种各样的应用功能。但是，正如 Mass StorageClass 的使用，使得各 19 个厂商生产的 U 盘都能通过操作系统自带的统一的驱动程序来使用，对 U 盘的普及使用起了极大的推动作用，制定其它这些 Class 也是为了同样的目的。



driver 和 host controller 像不像 core 的两个保镖？没办法，这可是 core 啊。协议里也说了，host controller 的驱动（HCD）必须位于 USB 软件的最下一层，任小强们也说了，咱们必须位于房地产这个链子的最下一层。HCD 提供 host controller 硬件的抽象，隐藏硬件的细节，在 host controller 之下是物理的 USB 及所有与之连接的 USB 设备。而 HCD 只有一个客户，对一个人负责，就是咱们的 USB core，USB core 将用户的请求映射到相关的 HCD，用户不能直接访问 HCD。咱们写 USB 驱动的时候，只能调用 core 的接口，core 会将咱们的请求发送给相应的 HCD，用得着咱们操心的只有这么一亩三分地，core 为咱们完成了大部分的工作，linux 的哲学 是不是和咱们生活中不太一样那？

Kconfig

usbfs

usbfs 文件系统挂载在`/proc/bus/usb` 上(`mount -t usbfs none /proc/bus/usb`)，显示了当前连接的 USB 设备及总线的各种信息，每个连接的 USB 设备在其中都会有一个文件进行描述。比如文件`/proc/bus/usb/xxx/yyy`，xxx 表示总线的序号，yyy 表示设备在总线的地址，不过不能够依赖它们来稳定地访问设备，因为同一设备两次连接对应的描述文件可能会不同，比如，第一次连接一个设备时，它可能是 002/027，一段时间后再次连接，它可能就已经改变为 002/048。就好比好不容易你暗恋的 mm 今天见你的时候对你抛了个媚眼，你心花怒放，赶快去买了 100 块彩票庆祝，到第二天再见到她的时候，她对你说你是谁啊，你悲痛欲绝的刮开那 100 块彩票，上面清一色的谢谢你，谢谢你送钱。usbfs 可以开个专题来讨论了，以后的日子里并不会花更多的口舌 在它上面。

USB_SUSPEND

这一项是有关 usb 设备的挂起和恢复。开发 USB 的人都是节电节能的好孩子，所以协议里就规定了，所有的设备都必须支持挂起状态，就是说为了达到节电的目的，当设备在指定的时间内，3ms 吧，如果没有发生总线传输，就要进入挂起状态。当它收到一个 non-idle 的信号时，就会被唤醒。在这里呼吁一下，多利用利用有太阳的时候，少熬夜，又费电对身体又不好，不过，我应该明天才说这句话，因为半夜还有米兰的冠军杯那。节约用电从 USB 做起。不过目前来说内核对挂起休眠的支持普遍都不太好，而且许多的 USB 设备也没有支持它，还是暂且不表了。

Makefile

Makefile 可比 Kconfig 简略多了，所以看起来也更亲切点，咱们总是拿的 money 越多越好，看的代码越少越好。这里之所以会出现 CONFIG_PCI，是因为通常 USB 的 root hub 包含在一个 PCI 设备中，前面也已经聊过了。hcd-pci 和 hcd 顾名思义就知道是说 host controller 的，它们实现了 host controller 公共部分，按协议里的说法它们就是 HCDI (HCD 的公共 接口)，host 目录下则实现了各种不同的 host controller，咱们这里不怎么会聊到具体 host controller 的实现。CONFIG_USB_DEVICEFS 前面的

Kconfig文件里也见到了，关于 usbfs的，已经说了这里不打算过多关注它，所以 inode.c和 devices.c两个文件也可以不用管了。这么看来，好像 core 下面的代码几乎都需要关注的样子，并没有减轻多少压力，不过要知道，这里本身就是 usb 的 core 部分，是要做很多的事为咱们分忧的，所以多点也是可以理解的。

drivers/usb/core/usb.c

```
938 subsys_initcall(usb_init);
939 module_exit(usb_exit)
```

我们看到一个 subsys_initcall，它是一个宏，我们可以把它理解为module_init，只不过因为这部分代码比较核心，开发者们把它看作一个子系统，而不仅仅是一个模块，这也很好理解，usbcore这个模块它代表的不是某一个设备，而是所有usb设备赖以生存的模块，Linux中，像这样一个类别的设备驱动被归结为一个子系统。比如pci子系统，比如scsi子系统，基本上，drivers/目录下面第一层的每个目录都算一个子系统，因为它们代表了一类设备。subsys_initcall(usb_init)的意思就是告诉我们usb_init是我们真正的初始化函数，而usb_exit()将是整个usb子系统的结束时的清理函数，于是我们就从usb_init开始看起。至于子系统在内核里具体的描述，牵涉到linux设备模型了，可以去看l3d3，或者更详细的。目前来说，我们只需要知道子系统通常显示在sysfs分层结构中的顶层，比如块设备子系统 24 对应/sys/block，当然也不一定，usb子系统对应的就是/sys/bus/usb。

```
860 /*
861 * Init
862 */
863 static int __init usb_init(void)
864 {
865     int retval;
866     if (nousb) {
867         pr_info("%s: USB support disabled\n", usbcore_name); // 867 行, pr_info只是一个打印信息的宏, printk的变体, 在 include/linux/kernel.h 里定义
868     return 0;
869 }
```

866 行，知道 C 语言的人都会知道 nousb 是一个标志，只是不同的标志有不一样的精彩，这里的 nousb 是用来让我们在启动内核的时候通过内核参数去掉 USB 子系统的，linux 社会是一个很人性化的世界，它不会去逼迫我们接受 USB，一切都只关乎我们自己的需要。不过我想我们一般来说是不会去指定 nousb 的吧，毕竟它那么的讨人可爱。如果你真的指定了 nousb，那它就只会幽怨的说一句“USB support disabled”，然后退出 usb_init。

```
870
871     retval = ksuspend_usb_init(); // 871 行, 电源管理方面的。如果在编译内核时没有打开电源管理, 也就是说没有定义 CONFIG_PM, 它就什么也不做。
872     if (retval)
873         goto out;
874     retval = bus_register(&usb_bus_type); // 874 行, 注册 USB 总线, 只有成功的将 USB 总线子系统注册到系统中, 我们才可以向这个总线添加 USB 设备。基于它显要的江湖地位, 就拿它做为日后突破的方向了, 擒贼先擒王, 这个越老越青春的道理在 linux 中也是同样适用的。
875     if (retval)
876         goto bus_register_failed;
877     retval = usb_host_init(); // 执行 host controller 相关的初始化。
878     if (retval)
```

```

879 goto host_init_failed;
880 retval = usb_major_init();//一个实际的总线也是一个设备，必须单独注册，因为 USB 是通过快速串行通信
来
读写数据，这里把它当作了字符设备来注册。
881 if (retval)
882 goto major_init_failed;
883 retval = usb_register(&usbfs_driver);
884 if (retval)
885 goto driver_register_failed;
886 retval = usb_devio_init();
887 if (retval)
888 goto usb_devio_init_failed;
889 retval = usbfs_init();
890 if (retval)
891 goto fs_init_failed;
//883~891 行，都是 usbfs 相关的初始化。
892 retval = usb_hub_init();//hub 的初始化，这个某人讲了。
893 if (retval)
894 goto hub_init_failed;
895 retval = usb_register_device_driver(&usb_generic_driver,
THIS_MODULE);//注册 USB device driver，戴好眼镜看清楚了，是 USB device driver 而不是
USB driver，前面说过，一个设备可以有多个接口，每个接口对应不同的驱动程序，这里
所谓的 device driver 对应的是整个设备，而不是某个接口。内核里结构到处有，只是 USB
这儿格外多。
896 if (!retval)
897 goto out;
898
899 usb_hub_cleanup();
900 hub_init_failed:
901 usbfs_cleanup();
902 fs_init_failed:
903 usb_devio_cleanup();
904 usb_devio_init_failed:
905 usb_deregister(&usbfs_driver);
906 driver_register_failed:
907 usb_major_cleanup();
908 major_init_failed:
909 usb_host_cleanup();
910 host_init_failed:
911 bus_unregister(&usb_bus_type);
912 bus_register_failed:
913 ksuspend_usb_cleanup();
914 out:
915 return retval;
916 }

```

_init

看到上面定义里的_init 标记没，写过驱动的应该不会陌生，它对内核来说就是一种暗示，表明这个函数仅在初始化期间使用，在模块被装载之后，它占用的资源就会释放掉用作它处。它的暗示你懂，可你的暗示，她却不懂或者懂装不懂，多么让人感伤。它在自己短暂的一生 中一直从事繁重的工作，吃的是草吐出的是牛奶，留下的是整个 USB 子系统的繁荣。

对_init 的定义在 include/linux/init.h 里

```
43 #define __init __attribute__((__section__(".init.text")))
```

attribute

而 **attribute** 就是这些扩展中的一个，它主要被用来声明一些特殊的属性，这些属性主要被用来指示编译器进行特定方面的优化和更仔细的代码检查。GNU C 支持十几个属性，**section** 是其中的一个，我们查看 gcc 的手册可以看到下面的描述

通常编译器将函数放在.text 节，变量放在.data 或 .bss 节，使用 section 属性，可以让编译器将函数或变量放在指定的节中。那么前面对 **init** 的定义便表示将它修饰的代码放在.init.text 节。连接器可以把相同节的代码或数据安排在一起，比如 init 修饰的所有代码都会被放在.init.text 节里，初始化结束后就可以释放这部分内存。

那内核又是如何调用到这些_init 修饰的初始化函数那？

要回答这个问题，还需要回顾一下上面 938 行的代码，那里已经提到 subsys_initcall 也是一个宏，它也在 include/linux/init.h 里定义 125 #define subsys_initcall(fn) **define_initcall("4",fn,4)** 这里又出现了一个宏 **define_initcall**，它是用来将指定的函数指针 fn 放到 initcall.init 节里，也在 include/linux/init.h 文件里定义，这里就不多说了，有那点意思就可以了。而对于具体的 subsys_initcall 宏，则是把 fn 放到 initcall.init 的子节 initcall4.init 里。要弄清楚 initcall.init、.init.text 和 initcall4.init 这样的东东，我们还需要了解一点内核可执行文件相关的概念。内核可执行文件由许多链接在一起的对象文件组成。对象文件有许多节，如文本、数据、init 数据、bass 等等。这些对象文件都是由一个称为链接器脚本的文件链接并装入的。这个链接器脚本的功能是将输入对象文件的各节映射到输出文件中；换句话说，它将所有输入对象文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。

vmlinux.lds 是存在于 arch// 目录中的内核链接器脚本，它负责链接内核的各个节并将它们装入内存中特定偏移量处。涉及到的东西越来越多了是吧，先深呼吸，平静一下，坚定而又勇敢的打开 arch/i386/kernel/vmlinux.lds 文件，你就会见到前所未见的景象。我可以负责任的说，要看懂这个文件是需要一番功夫的，不过大家都是聪明人，聪明人做聪明事，所以你需要做的只是搜索 initcall.init，然后便会看到似曾相识的内容

```
_inicall_start = ;  
.initcall.init : AT(ADDR(.initcall.init) - 0xC0000000) {  
    *(.initcall1.init)  
    27  
    *(.initcall2.init)  
    *(.initcall3.init)  
    *(.initcall4.init)  
    *(.initcall5.init)  
    *(.initcall6.init)  
    *(.initcall7.init)  
}
```

这里的 **inicall_start** 指向 **.initcall.init** 节的开始，**inicall_end** 指向它的结尾。而 **.initcall.init** 节又被分为了 7 个子节，分别是

```
.initcall1.init  
.initcall2.init  
.initcall3.init  
.initcall4.init  
.initcall5.init  
.initcall6.init  
.initcall7.init
```

我们的 **subsys_initcall** 宏便是将指定的函数指针放在了 **.initcall4.init** 子节。其它的比如 **core_initcall** 将函数指针放在 **.initcall1.init** 子节，**device_initcall** 将函数指针放在了 **.initcall6.init** 子节等等，都可以从 include/linux/init.h 文件找到它们的定义。各个子节的顺序是确定的，即先调用 **.initcall1.init** 中的函数指针再调用 **.initcall2.init** 中的函数指针，等等。**init** 修饰的初始化函数在内核初始化过程中调用的顺序

和.initcall.init节里函数指针的顺序有关，不同的初始化函数被放在不同的子节中，因此也就决定了它们的调用顺序。至于实际执行函数调用的地方，就在/init/main.c文件里，内核的初始化么，不在那里还能在哪里，里面的do_initcalls函数会直接用到这里的initcall_start、_initcall_end来进行判断，不多说了。我的思念已经入滔滔江水泛滥成灾了，还是回到久违的usb_init函数吧。

linux 的设备模型

总线是如何发现设备的，设备又是如何和驱动对应起来的，它们经过怎样的艰辛才找到命里注定的那个他，它们的关系如何，白头偕老型的还是朝三暮四型的，这些问题就不是他们关心的了，是咱们需要关心的。经历过高考千锤百炼的咱们还能够惊喜的发现，这些疑问的中心思想中心词汇就是总线、设备和驱动，没错，它们都是咱们这里要聊的linux设备模型的名角。

总线：struct bus_type

当然，struct bus_type中的drivers和devices分别表示了这个总线拥有哪些设备和哪些驱动

```
52 struct bus_type {  
53     const char * name;  
54     struct module * owner;  
55  
56     struct kset subsys;  
57     struct kset drivers;  
58     struct kset devices;  
59     struct klist klist_devices;  
60     struct klist klist_drivers;  
61  
62     struct blocking_notifier_head bus_notifier;  
63  
64     struct bus_attribute * bus_attrs;  
65     struct device_attribute * dev_attrs;  
66     struct driver_attribute * drvAttrs;  
67     struct bus_attribute drivers_autoprobe_attr;  
68     struct bus_attribute drivers_probe_attr;  
69  
70     int (*match)(struct device * dev, struct device_driver *  
drv);  
71     int (*uevent)(struct device * dev, char **envp,  
72     int num_envp, char *buffer, int  
buffer_size);  
73     int (*probe)(struct device * dev);  
74     int (*remove)(struct device * dev);  
75     void (*shutdown)(struct device * dev);  
76  
77     int (*suspend)(struct device * dev, pm_message_t state);  
78     int (*suspend_late)(struct device * dev, pm_message_t state);  
79     int (*resume_early)(struct device * dev);  
80     int (*resume)(struct device * dev);  
81  
82     unsigned int drivers_autoprobe:1;  
83 };
```

设备：struct device

struct device中的bus表示这个设备连到哪个总线上， driver表示这个设备的驱动是什么

```
410 struct device {  
411     struct klist klist_children;  
412     struct klist_node knode_parent; /* node in sibling list */  
413     struct klist_node knode_driver;  
414     struct klist_node knode_bus;  
415     struct device *parent;  
416  
417     struct kobject kobj;  
418     char bus_id[BUS_ID_SIZE]; /* position on parent bus */  
419     struct device_type *type;  
420     unsigned is_registered:1;  
421     unsigned uevent_suppress:1;  
422     struct device_attribute uevent_attr;  
423     struct device_attribute *devt_attr;  
424  
425     struct semaphore sem; /* semaphore to synchronize calls to  
426     * its driver.  
427 */  
428  
429     struct bus_type * bus; /* type of bus device is on */  
430     struct device_driver *driver; /* which driver has allocated this  
431     * device */  
432     void *driver_data; /* data private to the driver */  
433     void *platform_data; /* Platform specific data, device  
434     * core doesn't touch it */  
435     struct dev_pm_info power;  
436  
437 #ifdef CONFIG_NUMA  
438     int numa_node; /* NUMA node this device is close to */  
439 #endif  
440     u64 *dma_mask; /* dma mask (if dma'ble device) */  
441     u64 coherent_dma_mask; /* Like dma_mask, but for  
442     * alloc_coherent mappings as  
443     * not all hardware supports  
444     * 64 bit addresses for consistent  
445     * allocations such descriptors. */  
446  
447     struct list_head dma_pools; /* dma pools (if dma'ble) */  
448  
449     struct dma_coherent_mem *dma_mem; /* internal for coherent mem  
450     * override */  
451     /* arch specific additions */  
452     struct dev_archdata archdata;  
453  
454     spinlock_t devres_lock;  
455     struct list_head devres_head;  
456  
457     /* class_device migration path */  
458     struct list_head node;  
459     struct class *class;  
460     dev_t devt; /* dev_t, creates the sysfs "dev" */  
461     struct attribute_group **groups; /* optional groups */
```

```
462
463 void (*release)(struct device * dev);
464 };
```

驱动： struct device_driver

struct device_driver中的bus表示这个驱动属于哪个 34 总线， klist_devices表示这个驱动都支持哪些设备，因为这里device是复数，又是list， 因为一个驱动可以支持多个设备，而一个设备只能绑定一个驱动。

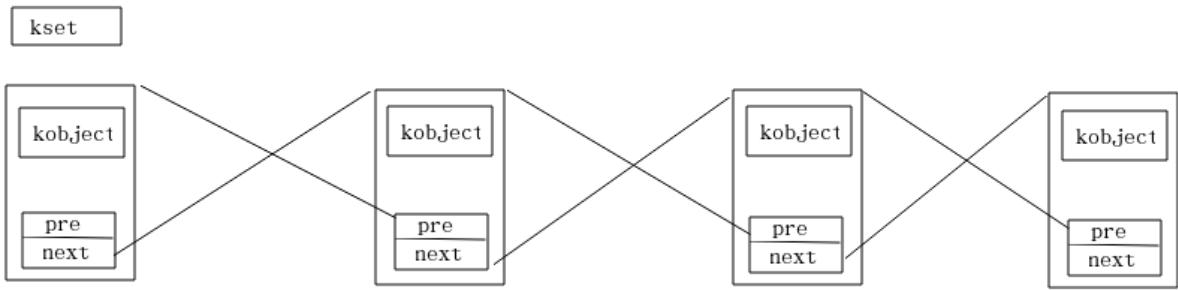
```
124 struct device_driver {
125     const char * name;
126     struct bus_type * bus;
127
128     struct kobject kobj;
129     struct klist klist_devices;
130     struct klist_node knode_bus;
131
132     struct module * owner;
133     const char * mod_name; /* used for built-in modules */
134     struct module_kobject * mkobj;
135
136     int (*probe) (struct device * dev);
137     int (*remove) (struct device * dev);
138     void (*shutdown) (struct device * dev);
139     int (*suspend) (struct device * dev, pm_message_t state);
140     int (*resume) (struct device * dev);
141 };
```

kobject

我可以肯定的告诉你，kobject和kset都是linux设备模型中最基本的元素，总线、设备、驱动是西瓜，**kobjjet、 klist** 是种瓜的人，没有幕后种瓜人的汗水不会有清爽解渴的西瓜，我们不能光知道西瓜的甜，还要知道种瓜人的辛苦。kobject 和 kset 不会在意自己自己的得失，它们存在的意义在于**把总线、设备和驱动这样的对象连接到设备模型上**。种瓜的人也不会在意自己的汗水，在意的只是能不能送出甜蜜的西瓜。

一般来说应该这么理解，整个 linux 的设备模型是一个 OO 的体系结构，总线、设备和驱动都是其中鲜活存在的对象，**kobject 是它们的基类**，所实现的只是一些公共的接口，**kset 是同种类型 kobject 对象的集合**，也可以说是对象的容器。只是因为 C 里不可能会有 C++ 里类的 class 继承、组合等的概念，只有通过 **kobject 嵌入到对象结构里来实现**。这样，**内核使用 kobject 将各个对象连接起来组成了一个分层的结构体系**，就好像通过马列主义 将我们 13 亿人也连接成了一个分层的社会体系一样。**kobject 结构里包含了 parent 成员，指向了另一个 kobject 结构，也就是这个分层结构的上一层结点**。而 kset 是通过链表来实现的，这样就可以明白，struct bus_type 结构中的成员 drivers 和 devices 表示了一条总线拥有两条链表，一条是设备链表，一条是驱动链表。我们知道了总线对应的数据结构，就可以找到这条总线关联了多少设备，又有哪些驱动来支持这类设备。

那么**klist**那？其实它就包含了一个链表和一个自旋锁，我们暂且把它看成链表也无妨，本来在早先的内核版本里，struct device_driver结构的devices成员就是一个链表类型。

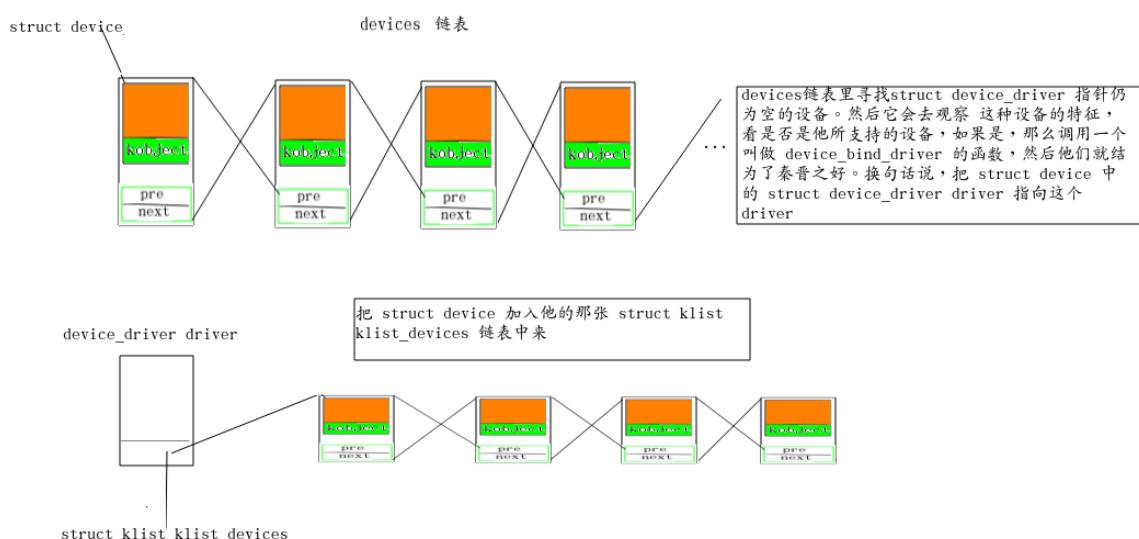


总线中的两条链表的形成

这要求每次出现一个设备就要向总线汇报，或者说注册，每次出现一个驱动，也要向总线汇报，或者说注册。比如系统初始化的时候，会扫描连接了哪些设备，并为每一个设备建立起一个 `struct device` 的变量，每一次有一个驱动程序，就要准备一个 `struct device_driver` 结构的变量。把这些变量统统加入相应的链表，`device` 插入 `devices` 链表，`driver` 插入 `drivers` 链表。这样通过总线就能找到每一个设备，每一个驱动。然而，假如计算机里只有设备却没有对应的驱动，那么设备无法工作。反过来，倘若只有驱动却没有设备，驱动也起不了任何作用。在他们遇见彼此之前，双方都如同路梗的野草，一个飘啊飘，一个摇啊摇，谁也不知道未来在哪里，只能在生命的风里飘摇。于是总线上的两张表里就慢慢的就挂上了那许多孤单的灵魂。`devices` 开始多了，`drivers` 开始多了，他们像是两个来自世界，`devices` 们彼此取暖，`drivers` 们一起狂欢，但他们有一点是相同的，都只是在等待属于自己的那个另一半。

设备与驱动的绑定

现在，总线上的两条链表已经有了，这个三角关系三个边已经有了两个，剩下的那个那？链表里的 `device` 和 `driver` 又是如何联系那？先有 `device` 还是先有 `driver`？很久很久以前，在那激情燃烧的岁月里，先有的是 `device`，每一个要用的 `device` 在计算机启动之前就已经插好了，插放在它应该在的位置上，然后计算机启动，然后操作系统开始初始化，总线开始扫描设备，每找到一个设备，就为其申请一个 `struct device` 结构，并且挂入总线中的 `devices` 链表中来，然后每一个驱动程序开始初始化，开始注册其 `struct device_driver` 结构，然后它去总线的 `devices` 链表中去寻找(遍历)，去寻找每一个还没有绑定 `driver` 的设备，即 `struct device` 中的 `struct device_driver` 指针仍为空的设备，然后它会去观察这种设备的特征，看是否是他所支持的设备，如果是，那么调用一个叫做 `device_bind_driver` 的函数，然后他们就结为了秦晋之好。换句话说，把 `struct device` 中的 `struct device_driver` 指向这个 `driver`，而 `struct device_driver` 把 `struct device` 加入他的那张 `struct klist klist_devices` 链表中来。就这样，`bus`、`device` 和 `driver`，这三者之间或者说他们中的两两之间，就给联系上了。知道其中之一，就能找到另外两个。一荣俱荣，一损俱损。



但现在情况变了，在这红莲绽放的日子里，在这樱花伤逝的日子里，出现了一种新的名词，叫热插拔。device 可以在计算机启动以后在插入或者拔出计算机了。因此，很难再说是先有 device 还是先有 driver 了。因为都有可能。device 可以在任何时刻出现，而 driver 也可以在任何时刻被加载，所以，出现的情况就是，**每当一个 struct device 诞生，它就会去 bus 的 drivers 链表中寻找自己的另一半，反之，每当一个一个 struct device_driver 诞生，它就去 bus 的 devices 链表中寻找它的那些设备。**如果找到了合适的，那么 ok，和之前那种情况一下，调用 device_bind_driver 绑定好。如果找不到，没关系，等待吧，等到昙花再开，等到风景看透，心中相信，这世界上总有一人是你所等的，只是还没有遇到而已。

Linux 设备模型中的总线

Linux 设备模型中的总线落实在 USB 子系统里就是 usb_bus_type，它在 usb_init 函数的 874 行注册，在 driver.c 文件里定义

```
1523 struct bus_type usb_bus_type = {  
1524     .name = "usb",  
1525     .match = usb_device_match,//match这个函数指针  
就比较有意思了，它充当了一个红娘的角色，在总线的设备和驱动之间牵线搭桥  
1526     .uevent = usb_uevent,  
1527     .suspend = usb_suspend,  
1528     .resume = usb_resume,  
1529 };
```

usb_device_match

540参数我们都已经很熟悉了，对应的就是总线两条
链表里的设备和驱动，也可以说是鹊桥版上的挂牌的和摘牌的。总线上有新设备或新的驱动
添加时，这个函数总是会被调用，如果指定的驱动能够处理指定的设备，也就是匹配成功，
函数返回 0。梦想是美好的，现实是残酷的，匹配是未必成功的，红娘再努力，双方对不上
眼也是实在没办法的事。

```
540 static int usb_device_match(struct device *dev, struct device_driver *drv)  
541 {  
542 /* devices and interfaces are handled separately */
```

```
543 if (is_usb_device(dev)) {  
544  
545 /* interface drivers never match devices */  
546 if (!is_usb_device_driver(drv))  
547 return 0;  
548  
549 /* TODO: Add real matching code */  
550 return 1;  
551  
552 } else {  
553     struct usb_interface *intf;  
554     struct usb_driver *usb_drv;  
555     const struct usb_device_id *id;  
556  
557 /* device drivers never match interfaces */  
558 if (is_usb_device_driver(drv))  
559 return 0;  
560  
561     intf = to_usb_interface(dev);  
562     usb_drv = to_usb_driver(drv);  
563 }
```

```
564 id = usb_match_id(intf, usb_drv->id_table);
565 if (id)
566 return 1;
567
568 id = usb_match_dynamic_id(intf, usb_drv);
569 if (id)
570 return 1;
571 }
572
573 return 0;
574 }
```

接口

前面的前面已经说了，接口是设备的接口。设备可以有多个接口，每个接口代表一个功能，每个接口对应着一个驱动。

Linux 设备模型的 device 落实在 USB 子系统，成了两个结构，一个是 struct usb_device，一个是 struct usb_interface，一个石头砸了两个坑，一支箭射下来两只麻雀，你说怪不怪。怪不怪还是听听复旦人甲怎么说，一个 usb 设备，两种功能，一个键盘，上面带一个扬声器，两个接口，那这样肯定得要两个驱动程序，一个是键盘驱动程序，一个是音频流驱动程序。道上的兄弟喜欢把这样两个整合在一起的东西叫做一个设备，那好，让他们去叫吧，我们用 interface 来区分这两者行了吧。于是有了这里提到的那个数据结构，struct usb_interface。

struct usb_interface

```
140 struct usb_interface {
141 /* array of alternate settings for this interface,
142 * stored in no particular order */
143 struct usb_host_interface *altsetting;//143 行，这里有个 altsetting 成员，只用耗费一个脑细胞就可以明白它的意思就是
alternate setting，可选的设置。
144
145 struct usb_host_interface *cur_altsetting; /* the currently
146 * active alternate setting */那么再耗费一个脑细胞就可以知道 145 行的
cur_altsetting 表示当前正在使用的设置，
147 unsigned num_altsetting; /* number of alternate settings */47 行的 num_altsetting 表示这个接口具有
可选设置的数量。
148
149 int minor; /* minor number this interface is
150 * bound to */149 行，minor，分配给接口的次设备号。地球人都知道，linux 下所有的硬件设备都是用
文件来表示的，俗称设备文件，在/dev 目录下边儿，为了显示自己并不是普通的文件，它
们都会有一个主设备号和次设备号，比如
151 enum usb_interface_condition condition; /* state of binding */151 行，condition 字段表示接口和驱
动的
绑定状态
152 unsigned is_active:1; /* the interface is not suspended */152 行，153 行与 157 行都是关于挂起和唤醒
的。协议里规定，所有的 usb 设备都必须支
持挂起状态，就是说为了达到节电的目的，当设备在指定的时间内，3ms 吧，如果没有发
生总线传输，就要进入挂起状态。当它收到一个 non-idle 的信号时，就会被唤醒。
152 表示接口是不是处于挂起状态。
153 unsigned needs_remote_wakeup:1; /* driver requires remote wakeup */表示是否需要
打开远程唤醒功能。远程唤醒允许挂起的设备给主机发信号，通知主机它将从挂起状态恢复，
注意如果此时主机处于挂起状态，就会唤醒主机，不然主机仍然在睡着，设备自个醒过来干
吗用。协议里并没有要求 USB 设备一定要实现远程唤醒的功能，即使实现了，从主机这边
```

儿也可以打开或关闭它。

154

```
155 struct device dev; /* interface specific device info */  
156 struct device *usb_dev; /* pointer to the usb class's device,  
if any */
```

接下来就剩下 155 行的 struct device dev 和 156 行的 struct device *usb_dev，看到 struct device 没，它们就是 linux 设备模型里的 device 嵌在这儿的对象，我们的心中要时时有个模型。不过这么想当然是不正确的，两个里面只有 dev 才是模型里的 device 嵌在这儿的，usb_dev 则不是。当接口使用 USB_MAJOR 作为主设备号时，usb_dev 才会用到，你找遍整个内核，也只在 usb_register_dev 和 usb_deregister_dev 两个函数里能够看到它，usb_dev 指向的就是 usb_register_dev 函数里创建的 usb class device。

157 int pm_usage_cnt; /* usage counter for autosuspend */

//pm 就是电源管理，usage_cnt 就是使

用计数，当它为 0 时，接口允许 autosuspend。什么叫 autosuspend？用过笔记本吧，曾经拥有过一台 DELL，现在正在拥有另一台 DELL，好无奈啊，为什么不是小黑或小白？有时合上笔记本后，它会自动进入休眠，这就叫 autosuspend。但不是每次都是这样的，就像这里只有当 pm_usage_cnt 为 0 时才会允许接口 autosuspend。至于这个计数在哪里统计，暂时还是飘过吧。

158};

配置与设置

咱们是难得糊涂几千年了，不会去区分配置还有设置有什么区别，起码我平时即使是再无聊也不会去想这个，但老外不一样，他们不知道老子也不知道郑板桥，所以说他们挺较真儿这个，还分了两个词，配置是 configuration，设置是 setting。先说配置，一个手机可以有 40 多种配置，比如可以摄像，可以接在电脑里当做一个 U 盘，那么这两种情况就属于不同的 配置，在手机里面有相应的选择菜单，你选择了哪种它就按哪种配置进行工作，供你选择的 这个就叫做配置。很显然，当你摄像的时候你不可以访问这块 U 盘，当你访问这块 U 盘的时候你不可以摄像，因为你做了选择。第二，既然一个配置代表一种不同的功能，那么很显然，不同的配置可能需要的接口就不一样，我假设你的手机里从硬件上来说一共有 5 个接口，那么可能当你配置成 U 盘的时候它只需要用到某一个接口，当你配置成摄像的时候，它可能只需要用到另外两个接口，可能你还有别的配置，然后你可能就会用到剩下那两个接口。再说说设置，一个手机可能各种配置都确定了，是振动还是铃声已经确定了，各种功能都确定了，但是声音的大小还可以变吧，通常手机的音量是一格一格的变动，大概也就 5、6 格，那么这个可以算一个 setting 吧。不过你还是不明白啥是配置啥是设置的话，就直接用大小关系来理解好了，毕竟大家对互相之间的大小关系都更敏感一些，不要说不是，一群 mm 走过来的时候，你的眼神已经背叛了你。这么说吧，**设备大于配置，配置大于接口，接口大于设置**，更准确的说是设备可以有多个配置，配置里可以包含一个或更多的接口，而接口通常又具有一个或更多的设置。

设备号

```
zhaoyong@zhaoyong:/dev$ ls -l | sed -n '1,5p'  
总用量 0  
crw-r--r-- 1 root      root      10,   235 4月    3 12:24 autofs  
drwxr-xr-x  2 root      root          560 4月    3 12:25 block  
drwxr-xr-x  2 root      root          100 4月    3 12:24 bsg  
crw-----  1 root      root      10,   234 4月    3 12:24 btrfs-control
```

主设备号表明了设备的种类，也表明了设备对应着哪个驱动程序，而次设备号则是因为一个驱动程序要支持多个设备而为了让驱动程序区分它们而设置的。

主设备号用来帮你 找到对应的驱动程序，次设备号给你的驱动用来决定对哪个设备进行操作。

设备要想在 linux 里分得一个主设备号，有个立足之地，也并不是那么容易的，主设备号虽说不是什么特别稀缺的资源，但还是需要设备先在驱动里提出申请，获得系统的批准才能拥有一个。因为一部分的主设备号已经被静态的预先指定给了许多常见的设备，你申请的时候 要避开它们，选择一个里面没有列出来的，也就是名花还没有主的，很严肃的说，挖墙角是很不道德的。这些已经被分配掉的主设备号都

列在 Documentation/devices.txt 文件里。当然，如果你是用动态分配的形式，就可以不去理会这些，直接让系统为你作主，替你选择一个即可。

很显然，任何一个有理智有感情的人都会认为 USB 设备是很常见的，linux 理应为它预留 了一个主设备号。看看 include/linux/usb.h 文件

```
7 #define USB_MAJOR 180
8 #define USB_DEVICE_MAJOR 189
```

苏格拉底说过，学的越多，知道的越多，知道的越多，发现需要知道更多。当我们知道了主设备号，满怀激情与向往的来寻找 USB 的主设备号时，我们却发现这里在上演真假李逵。这两个哪个才是我们苦苦追寻的她？你可以在内核里搜索它们都曾经出现什么地方，或者就跟随我回到 usb_init 函数。

```
880 retval = usb_major_init();
881 if (retval)
882     goto major_init_failed;
883 retval = usb_register(&usbfs_driver);
884 if (retval)
885     goto driver_register_failed;
886 retval = usb_devio_init();
887 if (retval)
888     goto usb_devio_init_failed;
889 retval = usbfs_init();
890 if (retval)
891     goto fs_init_failed;
```

前面只提了句 883~891 是与usbfs相关的就简单的飘过了，这里略微说的多一点。usbfs 为咱们提供了在用户空间直接访问usb硬件设备的接口，但是世界上没有免费的午餐，它需要内核的大力支持，usbfs_driver 就是用来完成这个光荣任务的。咱们可以去 usb_devio_init函数里看一看，它在devio.c文件里定义

```
retval = register_chrdev_region(USB_DEVICE_DEV, USB_DEVICE_MAX,
"usb_device");
if (retval) {
err("unable to register minors for usb_device");
goto out;
}
```

register_chrdev_region 函数获得了设备 usb_device 对应的设备编号，设备 usb_device 对应的驱动当然就是 usbfs_driver，参数 USB_DEVICE_DEV 也在同一个文件里有定义 #define USB_DEVICE_DEV MKDEV(USB_DEVICE_MAJOR, 0) 终于再次见到了 USB_DEVICE_MAJOR，也终于明白它是为了 usbfs 而生，为了让广大人民群众能够在用户空间直接和 usb 设备通信而生。因此，它并不是我们所要寻找的。

那么答案很明显了，USB_MAJOR就是咱们苦苦追寻的那个她，就是linux为USB设备预留 的主设备号。事实上，前面usb_init函数的 880 行，usb_major_init函数已经使用 USB_MAJOR注册了一个字符设备，名字就叫usb。我们可以在文件/proc/devices里看到 它们。

```
zhaoyong@zhaoyong:/$ cat /proc/devices | grep usb
180 usb
189 usb_device
```

/proc/devices 文件里显示了所有当前系统里已经分配出去的主设备号，当然上面只是列出了字符设备，Block devices 被有意的飘过了。很明显，咱们前面提到的 usb_device 和 usb 都在里面。

不过到这里还没完，USB设备有很多种，并不是都会用到这个预留的主设备号。比如俺的移动硬盘显示出来的主设备号就是8，你的摄像头在linux显示的主设备号也绝对不会是这里的USB_MAJOR。坦白的说，咱们经常遇到的大多数usb设备都会与input、video等子系统关联，并不单单只是作为usb设备而存在。如果usb设备没有与其它任何子系统关联，就需要在对应驱动的probe函数里使用usb_register_dev函数来注册并获得主设备号USB_MAJOR，你可以在drivers/usb/misc目录下看到一些例子，drivers/usb/usb-skeleton.c文件也属于这种。如果usb设备关联了其它子系统，则需要在对应驱动的probe函数里使用相应的注册函数，USB_MAJOR也就该干吗干吗去，用不着它了。比如，usb键盘关联了input子系统，驱动对应drivers/hid/usbhid目录下的usbkbd.c文件，在它的probe函数里可以看到使用了input_register_device来注册一个输入设备。43准确的说，这里的USB设备应该说成USB接口，当USB接口关联有其它子系统，也就是说不使用USB_MAJOR作为主设备号时，struct usb_interface的字段minor可以简单的忽略。**minor只在USB_MAJOR起作用时起作用。**

接口和驱动的绑定状态

```
83 enum usb_interface_condition {  
84     USB_INTERFACE_UNBOUND = 0,  
85     USB_INTERFACE_BINDING,  
86     USB_INTERFACE_BOUND,  
87     USB_INTERFACE_UNBINDING,  
88 };  
孤苦、期待、幸福、分开
```

p44