Linux 那些事儿

系列丛书

之

我是USB Core[1]

摘要

承上启下继往开来，承的是 U 盘/HUB，启的是 UHCI/EHCI，抓住设备和驱动两条生命线，讲述 usb core 的故事。

---

[1] 原文为 blog.csdn.net/fudan_abc 上的《linux 那些事儿之戏说 USB》，有闲情逸致的或者有批评建议的可以到上面做客，也可以 email 到 illtv.cn@gmail.com。

说在前面

说在前面，在这里耗费二八青春码这些，并不是因为喜欢它，相反，对它是毫无感觉可言，虽然每天都必须和它相依为伴，不离不弃，不过那可是丝毫没有办法的事情，非我所愿。听着是不是说到心坎里去了，感情都是很无奈是吧。再说谁让现在几乎所有电脑相关好玩有用的东东都往它身上靠，好像沾上它就跟上了时尚。时尚是什么，时尚就是标准，没看很久很久以前，咱们的郭天王留着个小分头的时候，大街上看到的听到的都是标准的小分头么，喔，这个当然是听不到的了。当然，时尚也需要发展改变，动不动还要来个复古什么的，现在好像很少见郭氏小分头了吧。不要说不知道我说的是什么吧，看俺的题目，无可奈何 USB 啊，什么，它还不够时尚？总不能来个奥运会指定总线才叫时尚吧。

君要问，既然如此无奈，还在这里罗唆它干吗。我对它是没感觉，因为俺已经心有所属了，但是没感觉并不代表不需要。那些找小蜜的也大都不会是因为感觉，这个问题有点敏感了，还是留给了解的人探讨吧，咱们还是探讨 USB，它无关政治民生，关乎的只是咱们的需要。

不过要有充足的动力和冲动罗唆它，还是需要点由头的。我是理科生，比较讲究逻辑性，有因有果才合理。多少年以前碰过 USB 摄像头驱动，现在又刚写了 USB 触摸屏的驱动，希望能够对 USB 协议在 linux 内核的实现有个通透的了解，将来与它怎么亲密接触将来才知道。过去，现在，将来，多象咱们每个人的人生三部曲，过去的已经过去，再为大学几年的吃喝玩乐不学无术懊悔也找不回失去的那几年，将来的无从可知，能把握的只有现在，所以不能再吃喝玩乐了，要抑制消费，努力的赚钱做房奴，努力的炒股做股东。我还要去努力的去了解它，努力的培养和它之间同志般的友谊，来争取在将来可能的多少次亲密接触中没有那种熟悉而又陌生的感觉，熟悉的陌生人，多伤感的词汇。

它从哪里来

你从哪里来，我的朋友，好像一只蝴蝶，飞进我的窗口。

在毛阿姨的嘹亮歌声中，USB 好像一只蝴蝶飞进了千家万户。它从哪里来，它从 intel 来。Intel 不养蝴蝶，做 CPU，它只是在蝴蝶的翅膀上烙上 intel inside，蝴蝶让咱们的同胞去养了，然后带着 intel 飞进了千万家。没办法，别人的核心竞争力是技术，咱们的核心竞争力是房子，老外的技术占领了咱们的房子。别人留下的是各种各样乱花渐欲迷人眼的标准，咱们留下的是拆拆建建的大厦千万间，还有任小强们的钞票。

不过，与 PCI、AGP 属于 intel 单独提出的硬件标准不同，Compaq、IBM、Microsoft 等也一起参与了这个游戏，它们一起在 94 年 11 月提出了 USB，于 95 年 11 月制定了 0.9 版本，96 年制定了 1.0 版本，不过它并没有因为有这些大佬的支持立即迎来它的春天，谁

让它诞生在冬天那，生不逢时啊。因为缺乏操作平台的良好支持和大量支持它的产品，这些标准都成了空谈。然后是 98 年 USB1.1 的出现，忽如一夜春风来，它就象春天里的一朵油菜花，终于涂上了浓重的一抹黄色。就像现在有些一炮走红的星们，谁又知道她们之前付出了多少努力，经历了多少艰辛，做了多少的铺垫那。

为什么要开发 USB。就好像我们问为什么房价这么高，任小强们的解释是地价太高，成本太高，造的少买的多，一面要满足广大人民群众的可观需求，一面要与国际价格体系接轨，压力多大啊，ZF 的解释是 KFS 牟取暴力，囤积居奇，咱们的解释是＃￥％×……。不过咱们这里的问题没有那么复杂，同样无关政治民生，关乎的只是咱们的需要。USB 出现以前，电脑的接口处于春秋战国时代，串口并口等多方割据，键盘、鼠标、Modem、打印机、扫描仪等都要连接在这些不同种的接口上，实行的是一夫一妻制，一个接口只能连接一个设备，不过咱们的电脑不可能有那么多这种接口，所以扩展能力不足，而且它们的速度也确实很有限。还有关键的一点是，热插拔对它们来说也是比较危险的操作，不想用了都成黄脸婆了还不能立即换掉，岂不是很不能满足很多 man 们内心的潜在需要。USB 正是为了解决速度、扩展能力、易用性应景而生的。

PK

2006 最火的是超级女生，最流行的是 PK。“她的一生充满了 PK”——从湖南卫视在《大长今》预告片中铿锵地说出了这句旁白时起，PK 已经不仅仅是 PK。

USB 的一生也充满了 PK，不过 USB 还不够老，说一生太早了些，发哥说的好，“我才刚上路呢”。

USB 最初的设计目标就是替代串行、并行等各种低速总线，以达到以一种单一类型的总线连接各种不同的设备。它现在几乎可以支持所有连接到 PC 上的设备，99 年提出的 USB2.0 理论上可以达到 480Mbps 的速度。它与串口、并口等的这场 PK 从一开始就是不平等的，这样的开始注定了以什么样的结果结束，只能说命运选择了 USB。我们很多人都说命运掌握在自己手里，从 USB 充满 PK 的一生，可以知道，只有变得比别人更强命运才能掌握在自己手里，所以说，还是赶紧吃点秋天的菠菜，去做强做大吧。

有了 USB 在这场 PK 中的大获全胜，才有了 USB 键盘、USB 鼠标、USB 打印机、USB 摄像头、USB 扫描仪、USB 音箱等。有了李宇春在超女 PK 中的胜利，才有了李宇春的蒙牛绿色心情。至于将来，“PK 自己的，让别人去说吧”，USB 如是说。

漫漫辛酸路

USB 的一生充满了 PK，并在 PK 中发展，1.0、1.1、2.0，……，漫漫辛酸路，一把辛酸泪。我们又何尝不是，上学碰到实行自费，毕业碰到 IT 崩溃，工作碰到房价见鬼，现在又碰到股市泡沫，与房价 PK，与庄家 PK，从来的结局都只有失败一个，USB 在 PK 中发展，我们在 PK 中只有变老。

PERFORMANCE	APPLICATIONS	ATTRIBUTES
LOW-SPEED • Interactive Devices • 10 – 100 kb/s	Keyboard, Mouse Stylus Game Peripherals Virtual Reality Peripherals	Lowest Cost Ease-of-Use Dynamic Attach-Detach Multiple Peripherals
FULL-SPEED • Phone, Audio, Compressed Video • 500 kb/s – 10 Mb/s	POTS Broadband Audio Microphone	Lower Cost Ease-of-Use Dynamic Attach-Detach Multiple Peripherals Guaranteed Bandwidth Guaranteed Latency
HIGH-SPEED • Video, Storage • 25 – 400 Mb/s	Video Storage Imaging Broadband	Low Cost Ease-of-Use Dynamic Attach-Detach Multiple Peripherals Guaranteed Bandwidth Guaranteed Latency High Bandwidth

Figure 3-1. Application Space Taxonomy

这张表是从 USB2.0 spec 里的，可以看出，它的高速模式最高已经达到了 480Mbps，即 60MBps，这是个什么概念，也就是说，照这个速度，你将自己从网上下的小短片备份到自己的移动硬盘上不用一秒钟，而按照 USB1.1 最高 12Mbps 的速度，你需要将近 1 分钟。2.0 比 1.1 的最高速度足足提高了几十倍，日后任小强们说房价还很合理，增加不多的时候可以不用和美国比和日本比了，和 USB 传输速度的发展比好了，更加能够显示房价的低廉。USB 走过的这段辛酸路，对咱们来说最直观的结果也就是传输速度提高了，过程很艰辛，结果很简单，是不。

USB 的各个版本是兼容的。每个 USB2.0 控制器带有 3 个芯片，根据设备的识别方式将信号发送到正确的控制芯片。我们可以将 1.1 设备连接到 2.0 的控制器上使用，不过它只能达到 1.1 的速度。同时也可以将 2.0 的设备连接到 1.1 的控制器上，不过不能指望它能以 2.0 的速度运行。毕竟走过的路太辛酸了，没有那么快就忘掉，好像我们不时的要去交大门口的老赵烤肉店忆苦思甜一样，我们不能忘本，USB 也不能。

显然，Linux 对 USB1.1 和 2.0 都是支持的，通过浏览 drivers/usb/host 目录下的 Kconfig 文件，我们可以知道内核里支持的控制器情况。

```
7 config USB_EHCI_HCD
8         tristate "EHCI HCD (USB 2.0) support"
9         depends on USB && USB_ARCH_HAS_EHCI
10         ---help---
11           The Enhanced Host Controller Interface (EHCI) is standard for USB 2.0
12           "high speed" (480 Mbit/sec, 60 Mbyte/sec) host controller hardware.
13           If your USB host controller supports USB 2.0, you will likely want to
14           configure this Host Controller Driver.  At the time of this writing,
15           the primary implementation of EHCI is a chip from NEC, widely available
16           in add-on PCI cards, but implementations are in the works from other
17           vendors including Intel and Philips.  Motherboard support is
appearing.
18
19           EHCI controllers are packaged with "companion" host controllers (OHCI
20           or UHCI) to handle USB 1.1 devices connected to root hub ports.  Ports
21           will connect to EHCI if the device is high speed, otherwise they
22           connect to a companion controller.  If you configure EHCI, you should
23           probably configure the OHCI (for NEC and some other vendors) USB Host
24           Controller Driver or UHCI (for Via motherboards) Host Controller
25           Driver too.
26
27           You may want to read <file:Documentation/usb/ehci.txt>.
28
29           To compile this driver as a module, choose M here: the
30           module will be called ehci-hcd.
```

这里使用的是 2.6.22 版本的内核。所有与 USB 相关的代码都在 drivers/usb 目录下面。上面的 Kconfig 文件说的很清楚，ehci-hcd 模块支持的是 USB2.0 控制器的高速模式，它本身并不支持全速或低速模式，对连接上的 USB1.1 设备的支持，是通过 ohci-hcd 或 uhci-hcd 模块。如果我们只配置了 EHCI，就没有办法使用 usb 的鼠标键盘。如果你碰到了 usb 键盘或鼠标不能用的情况，很可能就是因为配置 EHCI 的同时没有去配置 OHCI 或 UHCI。多少年以前我还是个青涩少年的时候就遇到过这个问题。

```
87 config USB_OHCI_HCD
88         tristate "OHCI HCD support"
89         depends on USB && USB_ARCH_HAS_OHCI
90         select ISP1301_OMAP if MACH_OMAP_H2 || MACH_OMAP_H3
91         select I2C if ARCH_PNX4008
92         ---help---
93           The Open Host Controller Interface (OHCI) is a standard for accessing
94           USB 1.1 host controller hardware.  It does more in hardware than
Intel's
```

```
95            UHCI specification.  If your USB host controller follows the OHCI
spec,
96            say Y.  On most non-x86 systems, and on x86 hardware that's not using
a
97            USB controller from Intel or VIA, this is appropriate.  If your host
98            controller doesn't use PCI, this is probably appropriate.  For a PCI
99            based system where you're not sure, the "lspci -v" entry will list the
100           right "prog-if" for your USB controller(s):  EHCI, OHCI, or UHCI.
101
102           To compile this driver as a module, choose M here: the
103           module will be called ohci-hcd.
160
161 config USB_UHCI_HCD
162         tristate "UHCI HCD (most Intel and VIA) support"
163         depends on USB && PCI
164         ---help---
165           The Universal Host Controller Interface is a standard by Intel for
166           accessing the USB hardware in the PC (which is also called the USB
167           host controller). If your USB host controller conforms to this
168           standard, you may want to say Y, but see below. All recent boards
169           with Intel PCI chipsets (like intel 430TX, 440FX, 440LX, 440BX,
170           i810, i820) conform to this standard. Also all VIA PCI chipsets
171           (like VIA VP2, VP3, MVP3, Apollo Pro, Apollo Pro II or Apollo Pro
172           133). If unsure, say Y.
173
174           To compile this driver as a module, choose M here: the
175           module will be called uhci-hcd.
```

OHCI 和 UHCI 虽然支持的都是 1.1 的控制器，但是支持的硬件范围不一样，房子、股票我们需要关心的太多了，就不用去管它们了，如果你在编译内核，直接选上它们就是了。

我型我秀

在 2006 这个选秀年里，超级女生也并不是一枝独秀，有个成语怎么说来着？好像是雨后春笋吧，现在的选秀节目就像这个笋，很多很多，不过有非常蔫的，有被雨水泡烂的，有刚发芽很嫩很嫩的，也有很成熟快枯萎的。我型我秀算是发育的比较好的一个了，虽然我愣是都没看过，超级女生我也是在叶一茜被淘汰后就再也不看了，不过俺从 10 进几来着才开始看。当然俺可绝对不是什么 QQ 糖，俺只是一个正常的 man 而已。如果你问什么是选秀节目的话，那只能说明两个问题，一是你很有幽默感，二就是说明你好象有那么一点点落后了，咱可以不喜欢但不能落后，是不。

USB 既然能一路 PK 走过来，也算是一个挺能秀的角色了，不然也不会有那么多的拥护者那。现在都提倡要善于推销自己，不是也有那么多 mm 拿性感写真做简历么，可惜俺是一大老爷们，所以去不了伊莱克什么的地方做助理，只能老老实实在这里码字。

既然这里说的就是 USB，也挑一些大家可能感兴趣的帮它秀一下。USB 为所有的 USB 外设都提供了单一的标准的连接类型，这就简化了外设的设计，也让我们不用再去想哪个设备对应哪个插槽的问题，就象种萝卜，一个萝卜一个坑，但是哪个萝卜种到哪个坑里我们是不用想的吧。它比我们大多数人都讨巧的多是不。

USB 支持热插拔，热插拔不会不知道吧，除非你耍我。其它的如 SCSI 设备等必须在关掉主机的情况下才能增加或移走外围设备。所以说，USB 的一生不仅仅是 PK 的一生，也是丰富多彩的一生，不用实行一夫一妻制，可以不用关机就能更换不同种类的外设。

USB 在设备供电方面提供了灵活性。USB 设备可以通过 USB 电缆供电，不然咱们的移动硬盘、ipod 什么的也用不了了。相对应，有的 USB 设备也可以使用普通的电源供电。

USB 能够支持从几十 k 到几十 m 的传输速率，来适应不同种类的外设，这点前面那张表说得很详细了。它可以支持多个设备同时操作，也支持多功能的设备。多功能的设备当然指的就是一个设备同时有多个功能，大白话，比如 USB 扬声器。这通过在一个设备里包含多个接口来支持，一个接口支持一个功能，这是后话了。

USB 可以支持多达 127 个设备，很多吧，不过比起皇帝的后宫佳丽三千来说还不是一个数量级的。

USB 可以保证固定的带宽，这个对视频音频设备是利好。

USB spec 上还有很多可以秀的，就不多说了。没有人可以了解对方的全部，不然恋爱就谈不下去了，是不。

应该给 USB 的这场秀什么评价那？我型我秀造福了少数人还有电信移动联通，USB 造福了全人类。

我是一棵树（一）

我是一棵树，静静的站在田野里，风儿吹过，我不知它的去向，人儿走过，我不知谁会为我停留。

我多少多少年以前刚刚情窦初开的时候，在本本上留下过一篇我是一棵树，这是其中一句。当然经过了这些年的漫漫辛酸路，当时的心情早就被腐蚀掉了，现在只觉得自己居然也有过

颇有才情的一段日子。USB 子系统也是一颗树，比我幸运的是它不用再等待谁为它停留了，我会在这里深情款款的陪伴它的。

从拓扑上来看，USB 子系统并不以总线的方式来部署，它是一颗由多个点对点的连接构成的树。

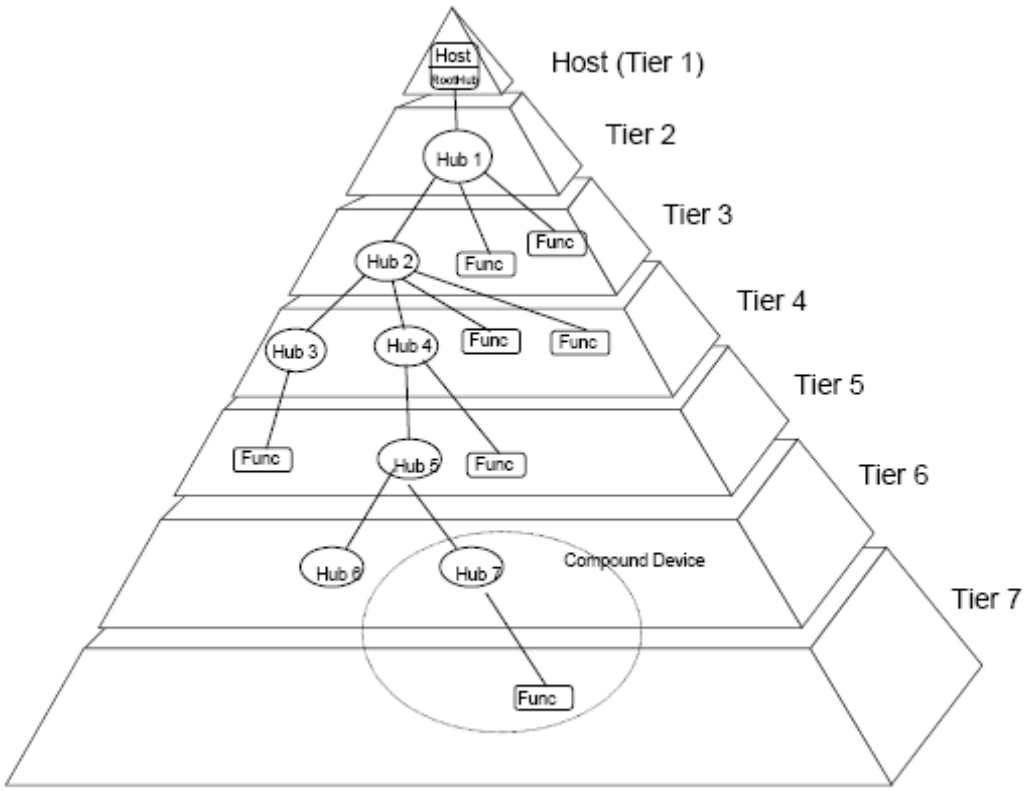


Figure 4-1. Bus Topology

我指着路边一颗老的奇形怪状的树问朋友：这是什么树？朋友的回答很简短：大树。那上面图里的是什么树？答案就是用头发梢想想也应该知道了吧，当然是大树了，不过是 USB 的大树。答案简短，蕴含的道理可不简短，它主要包括了 USB 连接、USB host controller 和 USB device 三个部分。而 USB device 还包括了 hub 和功能设备，也就是上图里的 Func。下面还是逐个说说吧。

什么是 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。否则让 cpu 去盯着每一个设备做每一件事情，那是不现实的，那就好比让一个学院的院长去盯着我们每一个本科生上课，去管理我们的出勤，只能说，不现实。

所以我们就被分成了几个系，通常院长有什么指示直接跟各系领导说就可以了，如果他要和三个系主任说事情，他即使不把三个人都召集起来开个会，也可以给三个人各打一个电话，打完电话他就忙他自己的事情去了，比如去和他带的女硕士风花雪月。而三个系主任就会去安排下面的人去执行具体的任务，完了之后他们就会像院长汇报。

那么 hub 是什么？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，让多个人共用一个网口，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并不只是一个 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。

而 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。

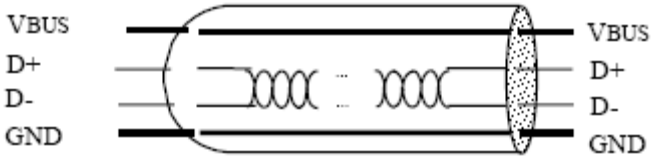


Figure 4-2. USB Cable

我是一棵树（二）

公元 312 年一天夜里，罗马附近的米尔维亚桥，忧思满腹的君士坦丁正在对第二天即将到来的大战感到发愁，当他眺望星空之际，突然看到苍茫的天空中突然出现了四个硕大无朋的火红色的十字架，伴随着这样的字样：依靠此，你将大获全胜。

前几天看碟看到欧洲历史部分的君士坦丁，就 google 到了这么一段。当然这太玄乎了，介于宗教信仰自由，我还是持保留意见吧。不过真有这样的好事，我希望是这样情景：在公元 2007 年 9 月的一天夜里，上海的徐家汇，忧心忡忡的我正在为股市被套而发愁，当我眺望夜空之际，插句题外话，上海几乎没有星空，所有只好将就用夜空代替了，突然看到苍茫的天空中出现了 6 个硕大无朋的数字，伴随着这样的字样：依靠此，你将有 43 个涨停。

为什么说 43 个涨停，这不是刚好比那个金泰多一个么，好不容易碰到这种好事，怎么地也

得来个吉尼斯吧，咱别的不多就是吉尼斯多，谁让人多那，找几千个人一块弹弹琴，甩甩头什么的不就有了。

君士坦丁的统治离不开基督教。咱们的 USB 大树要想茁长成长也离不开 USB 协议。USB 总线是一种轮询方式的总线。协议规定所有的数据传输都必须由主机发起，host controller 初始化所有的数据传输，各种设备紧紧围绕在主机周围。基督教说了，君士坦丁是上帝的第多少个儿子，所有罗马人都要围绕在君士坦丁周围，他不说话，谁都不能出声。

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，实际上只是为了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，而你得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，因为不管你在哪里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。

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。

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，往往通过他来对其它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比较快的时间内拷贝完。

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。

最终奥义

奥义是什么？看过圣斗士不？人是人他妈生的，妖是妖他妈生的，大家都是讨口饭吃，不用这么耍我吧。

那么打败圣斗士的最终奥义是什么？因为对圣斗士用过的招式第二次就不管用了，所以，你必须练 9999999……种拳法，让星矢们搞不清哪个才是你的绝招。如果全部用完了，他们还没有死的话，千万不要慌张，请重复一次。因为那条定律已经被破解了。谁还记得你打过什么招式啊？？

这么强的话不是我说的，我也不是教大家怎么打败圣斗士，我是他们忠实的 fans，爱护还来不及。

前面说了 USB 是一棵树，不过树也是有生命的，也是有内心世界的，我们不能只看到它枝枝丫丫的表面，需要从更深层次的去关怀它。我们每个人不也都希望能够有那么一个人真正的理解自己的内心么？

那么 USB 的内心世界是什么，我们都需要理解些什么，这就是上面所说的最终的奥义么？其实，是称不上最终的，只是会换个角度冷静的剖析一番而已，因为我们每个人都不可能完全的了解另外一个人的内心世界，虽然有时你可能以为自己已经做到了，但实际上，最终还是会发现，这还差的很远，现实都是残酷的，不是么。

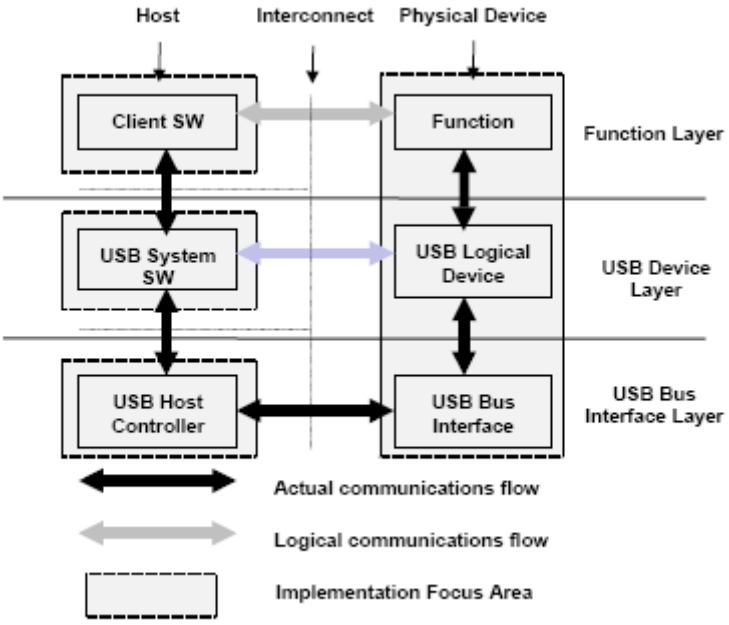


Figure 5-2. USB Implementation Areas

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。自从人类社会出现了阶级，我们的生活就再也离不开各种各样的层次了。孔子说，读史使人显得更明智，分层使人看的更明晰。ZF、任小强们、中介们、我们，多完美的层次，如果没有了层次，谁又能透彻的明白之间都是什么关系啊。所以，我们应该感谢有了阶级，感谢有了层次，能够让去更好的理解这些技术，更好的混口饭吃，拿更多的 money 去买房子，让更多的任小强们先富起来……。

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。

真的是这样么？多少年以前，作为一个普通人，我会坚定的说是。只是现在，作为一个读了协议的普通人，我要说并不完全是这样。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，并不包括所谓的各种设备驱动。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。

为什么会对这几个概念这么较真儿那，不是因为它们有多可爱，只是它们在协议里无处不在。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。

各种设备和主机是怎么连接在一起的那？前面的那颗树已经描绘的比较形象了，不过那棵树里的 compound device 被有意的飘过了。那么，刀是什么样的刀，剑是什么样的剑，compound 设备又是什么样的设备？其实，在 USB 的世界里，不仅仅有 compound device，还有 composite device，简单的中文名字已经无法形象的表达它们的区别，就还是使用它们的英文原名了。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，而 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。

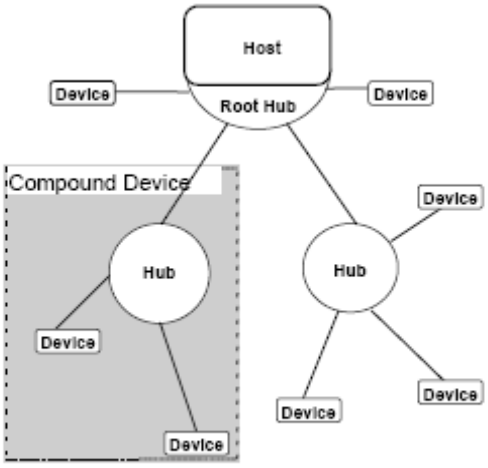


Figure 5-5. USB Physical Bus Topology

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。

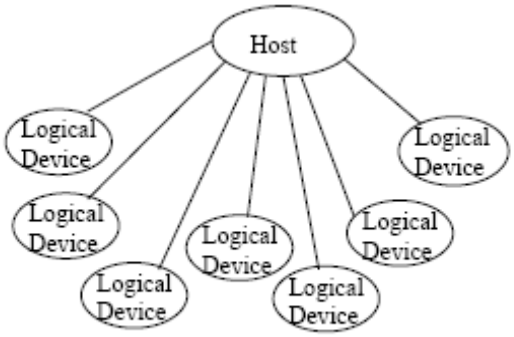


Figure 5-7. USB Logical Bus Topology

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。

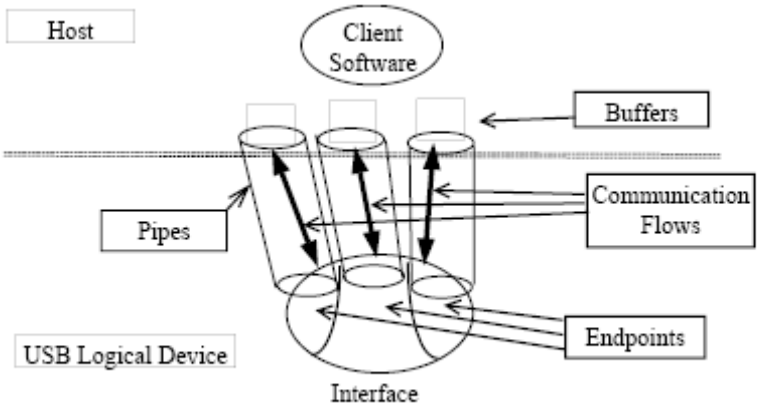


Figure 5-10. USB Communication Flow

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。

现在已经说了 USB 的历史发展体系结构等比较基本的东东，也就是协议的前几章，更多的东东还是下面和代码一块说吧。为了纪念这个历史性的时刻，学学电影里的酷哥警察说句：好戏开始了。

好戏开始了

上海的房价又在疯涨了，央行又加息了，邻居老大妈前几天丢的小狗居然自己跑回来了。多姿多彩的九月。

每一天的太阳都会不同，每一天的股市也一样。昨天绿油油的，今天红彤彤的，终于可歌可泣的迎来了人生中的第一次涨停。心情舒畅的一天。

有了这么好的大环境做铺垫，咱们的好戏也该开始了。

这以后的岁月里，主要就是结合代码去聊了，当然，新气象要用新代码，就使用最新的 2.6.22 版本的内核了。新的总会比旧的好，是么，总是只见新人笑，不见旧人哭。

在这什么都在失去理智飞涨的历史时刻，我会尽力的保持头脑的清醒，不会忘记自己要聊的是关于什么，所以，首先要去 drivers/usb 目录下走一走看一看。

atm class core gadget host image misc mon serial storage Kconfig Makefile README usb-skeleton.c

ls 命令的结果就是上面的 10 个目录和 4 个文件。usb-skeleton.c 是一个简单的 usb driver 的框架，感兴趣的可以去看看，目前来说，它还吸引不了我的眼球。那么首先应该关注什么？如果迎面走来一个 ppmm，你会首先看脸、脚还是其它？当然答案依据每个人的癖好会有所不同。不过这里的问题应该只有一个答案，那就是 Kconfig、Makefile、README。

README 里有关于这个目录下内容的一般性描述，它不是关键，只是帮助你了解。再说了，面对读我吧读我吧这么热情奔放的呼唤，善良的我们是不可能无动于衷的，所以先来看看里面都有些什么内容。

```
23 Here is a list of what each subdirectory here is, and what is contained in
24 them.
25
26 core/           - This is for the core USB host code, including the
27                   usbfs files and the hub class driver ("khubd").
28
29 host/           - This is for USB host controller drivers.  This
30                   includes UHCI, OHCI, EHCI, and others that might
31                   be used with more specialized "embedded" systems.
32
33 gadget/         - This is for USB peripheral controller drivers and
34                   the various gadget drivers which talk to them.
35
36
37 Individual USB driver directories.  A new driver should be added to the
38 first subdirectory in the list below that it fits into.
39
40 image/          - This is for still image drivers, like scanners or
41                   digital cameras.
42 input/          - This is for any driver that uses the input subsystem,
43                   like keyboard, mice, touchscreens, tablets, etc.
44 media/          - This is for multimedia drivers, like video cameras,
45                   radios, and any other drivers that talk to the v4l
46                   subsystem.
47 net/            - This is for network drivers.
48 serial/         - This is for USB to serial drivers.
49 storage/        - This is for USB mass-storage drivers.
50 class/          - This is for all USB device drivers that do not fit
51                   into any of the above categories, and work for a range
```

```
52                      of USB Class specified devices.
53 misc/             - This is for all USB device drivers that do not fit
54                      into any of the above categories.
```

drivers/usb/README 文件就描述了前面 ls 列出的那 10 个文件夹的用途。那么什么是 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。时代总在发展，当年胖杨贵妃照样迷死唐明皇，而如今人们欣赏的则是林志玲这样的魔鬼身材。同样，早期的 Linux 内核，其结构并不是如今天这般有层次感，远不像今天这般错落有致，那时候 drivers/usb/这个目录下边放了很多很多文件，usb core 与其他各种设备的驱动程序的代码都堆砌在这里，后来，怎奈世间万千的变幻，总爱把有情的人分两端。于是在 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。

那么 usb gadget 那？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。

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，通过专用驱动也能够实现各种各样的应用功能。但是，正如 Mass StorageClass 的使用，使得各

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。

我们响应了 README 的呼唤，它便给予了我们想要的，通过它我们了解了 usb 目录里的那些文件夹都有着什么样的角色。到现在为止，就只剩下 Kconfig、Makefile 两个文件了，它们又扮演着什么样的角色那？就好像我吃东西总是喜欢把好吃的留在最后享受一样，我也习惯于将重要的留在最后去描述。对于一个希望能够在 Linux 内核的汪洋代码里看到一丝曙光的人来说，将它们放在怎么重要的地位都不过分。我们去香港，通过海关的时候，总会有免费的地图啊各种指南拿，有了它们在手里我们才不至于无头苍蝇般迷惘的行走在陌生的街道上。即使在境内出去旅游的时候一般来说也总是会首先找份地图，当然了，这时就是要去买了，拿是拿不到的，不同的地方有不同的特色，不是么，别人的特色是服务，咱们的特色是花钱，有的地方特色是一块石头，那这块石头收你 70 也不能嫌贵，有的山上特色是那些多少年的洋房子，那就从进山收起吧，一个一个房子的来，口号就是不能让一分钱下山。Kconfig、Makefile 就是 linux kernel 迷宫里的地图，我们每次浏览 kernel 寻找属于自己的那一段代码时，都应该首先看看目录下的这两个文件。

不过，这里很明显，从俺的题目上就可以看出来，usb core 就是接下来需要关注的对象，就不表 Kconfig、Makefile 文件的内容了。

不一样的 core

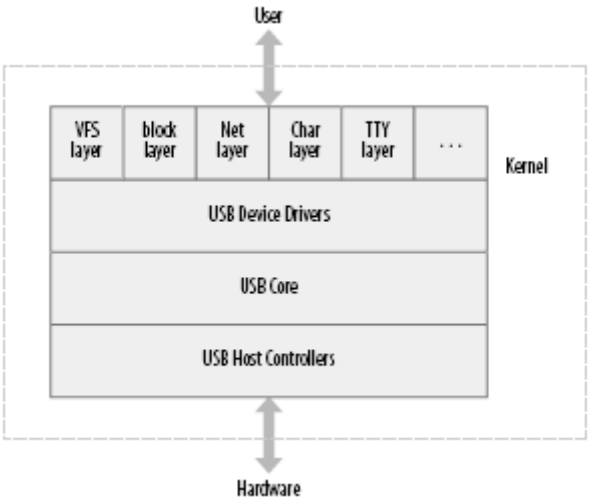
“生活中肯定还有比睡觉更好玩的事情！”大小卡梅拉们一直都抱有这样的信念。他们执著地追求那些种群中认为不可想象的事情。去看大海、去摘星星、去追回逃逸的太阳……，一路上处处坎坷、历经艰难，但总是逢凶化吉、化险为夷。最后还能收获超乎想象的回报和异乎寻常的果实。卡梅拉已经被当作一种象征，一种成长路上必不可少的“伴侣”。读不一样的卡梅拉，成就与众不同的你！读不一样的 core，成就不一样的你！

是不是像广告，嗯，确实是广告，是《不一样的卡梅拉》的广告。什么？不知道啥是卡梅拉？唉，有代沟了，就是说一只小鸡不想只是下蛋睡觉去冒险的故事，小时候看的书了。那 core 那？嗯，这句是我加的，因为它也是咱们接下来的漫漫辛酸路上必不可少的伴侣。

使用命令 lsmod，看看它的输出，然后找这么个模块 usbcore，不要说你找不到，我不会相信的。它是什么？它就是咱们这里要说的 usb 系统的核心，如果要在 linux 里使用 usb，这个模块是必不可少的，另外，你应该在 usbcore 那一行的后面看到 ehci_hcd 或 uhci_hcd 这样的东东，它们就是前面说的 usb host controller 的驱动模块，你的 usb 设备要工作，合适的 usb host controller 模块也是必不可少的。不过，咱们这里的主角还是 usbcore。

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

控制 USB 硬件的接口，而不用去考虑系统当前存在哪种 host controller。至于 core、host controller 和 driver 三者之间的关系，还是用 ldd3 里的图来说明吧。



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？

走到 drivers/usb/core 里去，使用 ls 瞧一瞧看一看，

Kconfig Makefile buffer.c config.c devices.c devio.c driver.c endpoint.c file.c generic.c hcd-pci.c hcd.c hcd.h hub.c hub.h inode.c message.c notify.c otg_whitelist.h quirks.c sysfs.c urb.c usb.c usb.h

使用 wc –l 命令统计一下，将近两万行的代码，core 不愧是 core，为大家默默的做这么多事，人民的好公仆鞠躬尽瘁，我会用一颗感恩的心去深刻理解你的内心回报你的付出的。Linux 背后的哲学是不是又和我们生活中不一样？

不过这么多文件里不一定都是我们所需要关注的，好钢要用在刃上，先拿咱们的地图来看看接下来该怎么走。先看看 Kconfig 文件

```
4 config USB_DEBUG
5         bool "USB verbose debug messages"
```

```
6          depends on USB
7          help
8            Say Y here if you want the USB core & hub drivers to produce a bunch
9            of debug messages to the system log. Select this if you are having a
10            problem with USB support and want to see more of what is going on.
```

这是 USB 的调试 tag，如果你在写 USB 设备驱动的话，最好还是打开它吧，不过这里它就不是我们关注的重点了。

```
15 config USB_DEVICEFS
16          bool "USB device filesystem"
17          depends on USB
18          ---help---
19            If you say Y here (and to "/proc file system support" in the "File
20            systems" section, above), you will get a file /proc/bus/usb/devices
21            which lists the devices currently connected to your USB bus or
22            busses, and for every connected device a file named
23            "/proc/bus/usb/xxx/yyy", where xxx is the bus number and yyy the
24            device number; the latter files can be used by user space programs
25            to talk directly to the device. These files are "virtual", meaning
26            they are generated on the fly and not stored on the hard drive.
27
28            You may need to mount the usbfs file system to see the files, use
29            mount -t usbfs none /proc/bus/usb
30
31            For the format of the various /proc/bus/usb/ files, please read
32            <file:Documentation/usb/proc_usb_info.txt>.
33
34            Usbfs files can't handle Access Control Lists (ACL), which are the
35            default way to grant access to USB devices for untrusted users of a
36            desktop system. The usbfs functionality is replaced by real
37            device-nodes managed by udev. These nodes live in /dev/bus/usb and
38            are used by libusb.
```

这个选项是关于 usbfs 文件系统的。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。

```
74 config USB_SUSPEND
75         bool "USB selective suspend/resume and wakeup (EXPERIMENTAL)"
76         depends on USB && PM && EXPERIMENTAL
77         help
78           If you say Y here, you can use driver calls or the sysfs
79           "power/state" file to suspend or resume individual USB
80           peripherals.
81
82           Also, USB "remote wakeup" signaling is supported, whereby some
83           USB devices (like keyboards and network adapters) can wake up
84           their parent hub.  That wakeup cascades up the USB tree, and
85           could wake the system from states like suspend-to-RAM.
86
87           If you are unsure about this, say N here.
```

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。

剩下的还有几项，不过似乎与咱们关系也不大，碰到再去说它，还是看看 Makefile。惭愧，差点打成 Makelove，看某人的文章看的了，这样的词汇总是很有感染力的，是不。

```
5 usbcore-objs    := usb.o hub.o hcd.o urb.o message.o driver.o \
6                        config.o file.o buffer.o sysfs.o endpoint.o \
7                        devio.o notify.o generic.o quirks.o
8
9 ifeq ($(CONFIG_PCI),y)
10         usbcore-objs    += hcd-pci.o
11 endif
12
13 ifeq ($(CONFIG_USB_DEVICEFS),y)
14         usbcore-objs    += inode.o devices.o
15 endif
16
17 obj-$(CONFIG_USB)       += usbcore.o
18
19 ifeq ($(CONFIG_USB_DEBUG),y)
20 EXTRA_CFLAGS += -DDEBUG
21 endif
```

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顾名而思义就知道是说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。

从这里开始

任小强们说房价高涨从现在开始，股评家们说牛市从 5000 点开始。他们的开始需要我们的钱袋，我的开始只需要一台电脑，最好再有一杯茶，伴着几支小曲儿，不盯着钱总是会比较惬意的。生容易，活容易，生活不容易，因为要盯着钱。

USB core 从 USB 子系统的初始化开始，我们也需要从那里开始，其实前面说的那么多只是一些开胃菜，先搞点协议里的东西垫垫底儿，正餐也就是说咱们 core 的故事现在才正式拉开了序幕。

usb 子系统的初始化在文件 drivers/usb/core/usb.c 里，因为咱们这里聊的主题就是 usb core，所以如果日后牵涉到 core 下面的哪个文件，就不再指明 drivers/usb/core/这么一长串目录名了。

```
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ldd3，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

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);
868                 return 0;
869         }
870
871         retval = ksuspend_usb_init();
872         if (retval)
873                 goto out;
874         retval = bus_register(&usb_bus_type);
875         if (retval)
876                 goto bus_register_failed;
877         retval = usb_host_init();
878         if (retval)
879                 goto host_init_failed;
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;
892         retval = usb_hub_init();
893         if (retval)
894                 goto hub_init_failed;
895         retval = usb_register_device_driver(&usb_generic_driver,
THIS_MODULE);
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 }
```

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。

受这种精神所感染，我觉得还是有必要为它说的更多些，21 世纪多的是任小强，缺的是知恩图报的人。对__init 的定义在 include/linux/init.h 里

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

好像这里的疑问要更多，不过与__init相比，这点辛苦算什么，我会在它强大的精神支持下尽量说清楚的。那么 __attribute__是什么？Linux内核代码使用了大量的GNU C扩展，以至于GNU C成为能够编译内核的唯一编译器，GNU C的这些扩展对代码优化、目标代码布局、安全检查等方面也提供了很强的支持。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

‘section ("section-name")’

Normally, the compiler places the code it generates in the `text’ section. Sometimes, however, you need additional sections, or you need certain particular functions to appear in special sections. The `section’ attribute specifies that a function lives in a particular section. For example, the declaration:

```
extern void foobar (void) __attribute__ ((section ("bar")));
```

puts the function ‘foobar’ in the ‘bar’ section.

```
Some file formats do not support arbitrary sections so the
‘section’ attribute is not available on all platforms.  If you
need to map the entire contents of a module to a particular
section, consider using the facilities of the linker instead.
```

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？好奇心是科学的原动力，茶叶蛋就是这么煮出来的，原子弹也是这么造出来的，__init 背后的哲学总不会比它们还难，芙蓉姐姐说了，我挑战，我喜欢。好像越聊越远了，不过想想多少年前当自己还是青涩少年的时候就对它极度的好奇过，这里还是尽量将它说一下，也顺便积攒下 rp，稍后的冠军杯里俺的米兰也好有个开门红。

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/<target>/ 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)
```

```
*(.initcall2.init)
*(.initcall3.init)
*(.initcall4.init)
*(.initcall5.init)
*(.initcall6.init)
*(.initcall7.init)
}
__initcall_end = .;
```

这里的__initcall_start 指向.initcall.init 节的开始，__initcall_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。

面纱

在爱情、背叛与死亡的漩涡中挣扎的凯蒂，亲历了幻想破灭与生死离别之后，终将生活的面纱从她的眼前渐渐揭去，从此踏上了不悔的精神成长之路。

向大家推荐这部片子《面纱》，没有那些小电影精彩，但是绝对值得一看。为什么会想到它，只在乎于现在的心情。前面说了那么多，才接触到 usb_init，有点一窥 usb 面纱下神秘容颜的味道。当然，我们并不需要去经历爱情、被判与死亡，所需要经历的只是忍受前面大段

大段的唠叨。

人往往可以被高尚感动，但始终不能因为高尚而爱上。因为被__init 给盯上，usb_init 在做牛做马的辛勤劳作之后便不得不灰飞烟灭，不可谓不高尚，但它始终只能是我们了解 usb 子系统这个面纱后面的内容的跳板，是起点，却不是终点，我们不会为它停留太久，有太多的精彩和苦恼在等着我们。

```
865         int retval;
866         if (nousb) {
867                 pr_info("%s: USB support disabled\n", usbcore_name);
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。

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：

```
242 #define pr_info(fmt,arg...) \
243         printk(KERN_INFO fmt,##arg)
```

这个可变参数宏要不要说一下？地球人都知道了，不过还是聊一下吧，我有多话症。99 年的 ISO C 标准里规定了可变参数宏，和函数语法类似，给个例子

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

里面的“...”就表示可变参数，调用时，它们就会替代宏体里的__VA_ARGS__。GCC 总是会显得特立独行一些，它支持更复杂的形式，可以给可变参数取个名字，再给个这种形式的例子

```
#define debug(format, args...) fprintf (stderr, format, args)
```

是不是显得更容易读了些？有了名字总是会容易交流一些。是不是与咱们的 pr_info 比较接近了？除了‘##’，它主要是针对空参数的情况。既然说是可变参数，那传递空参数也总是可以的，空即是多，多即是空，股市里的哲理这里同样也是适合的。如果没有‘##’，传递空参数的时候，比如

```
debug ("A message");
```

宏展开后，里面的字符串后面会多个多余的逗号。这个逗号你应该不会喜欢，它这是表错情了，而‘##’则会使预处理器去掉这个多余的逗号。

```
871         retval = ksuspend_usb_init();
872         if (retval)
873                 goto out;
874         retval = bus_register(&usb_bus_type);
875         if (retval)
876                 goto bus_register_failed;
877         retval = usb_host_init();
878         if (retval)
879                 goto host_init_failed;
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;
892         retval = usb_hub_init();
893         if (retval)
894                 goto hub_init_failed;
895         retval = usb_register_device_driver(&usb_generic_driver,
THIS_MODULE);
896         if (!retval)
897                 goto out;
```

871 到 897 这些行是代码里的排比句，相似的 init 不相似的内容，很显然都是在完成一些初始化，也是 usb_init 任劳任怨所付出的全部。这里先简单的说一下。

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。

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。

877 行，执行 host controller 相关的初始化。

880 行，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。

883~891 行，都是 usbfs 相关的初始化。

892 行，hub 的初始化，这个某人讲了。

895 行，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。

剩下的几行代码都是有关资源清除的，usb_init 这个短短的函数在承载着我们的希望的时候嘎然而止了，你的感觉是什么？我的感觉是：这哪是我能说的清楚的啊。它的每个分叉都更像是一个陷阱，黑黝黝看不到底，但是已经没有回头的路。

模型，又见模型

百晓生说，世界上有两样东西最让人捉摸不透，一个是小李飞刀，一个就是 linux 的设备模型。

进入 21 世纪，小李飞刀已经在电视上又见过无数遍，早就没了那种神秘感，可 linux 的设备模型仍然偏居一偶，让人端端的生起许多好奇来。

上文说 usb_init 给我们留下了一些岔路口，每条都像是不归路，让人不知道从何处开始，也看不到路的尽头。趁着徘徊彷徨的档儿，咱们还是先聊一下 linux 的设备模型。各位看官听好了，这可不是任小强们摆的房子模型，不存在忽悠你们的可能。

顾名而思义就知道设备模型是关于设备的模型，对咱们写驱动的和不写驱动的人来说，设备的概念就是总线和与其相连的各种设备了。电脑城的 IT 工作者都会知道设备是通过总线连到计算机上的，而且还需要对应的驱动才能用，可是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、设备、驱动，也就是 bus、device、driver，既然是名角，在内核里都会有它们自己专属的结构，在 include/linux/device.h 里定义。

```
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 * drv_attrs;
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 };
```

```
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'able 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 };
```

```
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 };
```

没有人会监督我节省纸张，所以就都贴出来了，有没有发现它们的共性是什么？对，都很复杂很长，那是因为还没有见到更复杂更长的。不妨把它们看成艺术品，linux 整个内核都是艺术品，既然是艺术，当然不会让你那么容易的就看懂了，不然怎么称大师称名家。这么想想咱们就会比较的宽慰了，阿 Q 是鲁迅对咱们 80 后最大的贡献。

我知道进入了 21 世纪，最缺的就是耐性，房价股价都让咱们没有耐性，内核的代码也让人没有耐性。不过做为最没有耐性的一代人，因为都被压扁了，还是要平心净气的扫一下上面的结构，我们会发现，struct bus_type 结构中有成员struct kset drivers 和struct kset devices，同时struct device结构中有两个成员struct bus_type * bus和struct device_driver *driver，struct device_driver结构中有两个成员struct bus_type * bus和struct klist klist_devices。先不说什么是klist、kset，光从成员的名字看，它们就是一个完美的三角关系。我们每个人心中是不是都有两个她？一个梦中的她，一个现实中的她。

凭一个男人的直觉，我们可以知道，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，struct device_driver中的bus表示这个驱动属于哪个

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。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。

单凭直觉，张钰出不了名。我们还需要看看什么是 klist、kset。还有上面 device 和 driver 结构里出现的 kobject 结构是什么？作为一个五星红旗下长大的孩子，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，kobjcet、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。这么一说，咱们上面的直觉都是正确的，如果咱们买股票，摸彩票时直觉都这么管用，那现在哪还有任小强们牛气哄哄的份儿。

是个 21 世纪的人都知道，三角关系很难处，不要说自己没搞过三角关系，没吃过猪肉还没见过猪跑啊。那么总线、设备和驱动它们只见是如何和谐共处那？还是先说说总线中的那两条链表是怎么形成的吧。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，双方都如同路埂的野草，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。

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 driver 指向这个 driver，而 struct device_driver 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，没有关系，等待吧，等到昙花再开，等到风景看透，心中相信，这世界上总有一个人是你所等的，只是还没有遇到而已。

繁华落尽

台湾作家林清玄在接受记者采访时，如此评价自己的 30 多年写作生涯：“第一个十年我才华横溢，‘贼光闪现’，令周边黯然失色；第二个十年，我终于‘宝光现形’，不再去抢风头，反而与身边的美丽相得益彰；进入第三个十年，繁华落尽见真醇，我进入了‘醇光初现’的阶段，真正体味到了境界之美。”

很久很久以前，在自己还是个文学小青年儿比较喜欢散文的时候，林清玄是我仅次于余秋雨的第二偶像。长夜有穷，真水无香。看过了 Linux 设备模型的繁华似锦，该是体味境界之美的时候了。

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,
1526         .uevent =       usb_uevent,
1527         .suspend =      usb_suspend,
1528         .resume =       usb_resume,
1529 };
```

看来是要向这个分叉走了，既然没有回头的路，就放平心情，欣赏沿路美景吧。name自然就是usb总线的绰号了，与芙蓉姐姐一般无二，人在江湖，身不由己。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，类似于交大BBS上的鹊桥版，虽然它们上面的条件都琳琅满目的，但明显这里match的条件不是那么的苛刻，要实际些。match指向了函数 usb_device_match

```
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 }
```

540 行，经历了 linux 设备模型的繁华，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。

543 行，一遇到 if 和 else，我们就知道又处在两难境地了，代码里我们可选择的太多，生活里我们可选择的太少，出生，长大，死亡，好像一直身不由己的随着命运在走。这里的岔路口只有两条路，一条给 USB 设备走，一条给 USB 接口走，各走各的路，分开了，就不再相见。

接口

宋正说天盛是球迷的天盛

张玉说张玉是导演的张玉

任小强说房子是人民的房子

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。

```
90 /**
91  * struct usb_interface - what usb device drivers talk to
92  * @altsetting: array of interface structures, one for each alternate
93  *      setting that may be selected.  Each one includes a set of
94  *      endpoint configurations.  They will be in no particular order.
95  * @num_altsetting: number of altsettings defined.
```

```
 * @cur_altsetting: the current altsetting.
 * @driver: the USB driver that is bound to this interface.
 * @minor: the minor number assigned to this interface, if this
 *      interface is bound to a driver that uses the USB major number.
 *      If this interface does not use the USB major, this field should
 *      be unused.  The driver should set this value in the probe()
 *      function of the driver, after it has been assigned a minor
 *      number from the USB core by calling usb_register_dev().
 * @condition: binding state of the interface: not bound, binding
 *      (in probe()), bound to a driver, or unbinding (in disconnect())
 * @is_active: flag set when the interface is bound and not suspended.
 * @needs_remote_wakeup: flag set when the driver requires remote-wakeup
 *      capability during autosuspend.
 * @dev: driver model's view of this device
 * @usb_dev: if an interface is bound to the USB major, this will point
 *      to the sysfs representation for that device.
 * @pm_usage_cnt: PM usage counter for this interface; autosuspend is not
 *      allowed unless the counter is 0.
 *
 * USB device drivers attach to interfaces on a physical device.  Each
 * interface encapsulates a single high level function, such as feeding
 * an audio stream to a speaker or reporting a change in a volume control.
 * Many USB devices only have one interface.  The protocol used to talk to
 * an interface's endpoints can be defined in a usb "class" specification,
 * or by a product's vendor.  The (default) control endpoint is part of
 * every interface, but is never listed among the interface's descriptors.
 *
 * The driver that is bound to the interface can use standard driver model
 * calls such as dev_get_drvdata() on the dev member of this structure.
 *
 * Each interface may have alternate settings.  The initial configuration
 * of a device sets altsetting 0, but the device driver can change
 * that setting using usb_set_interface().  Alternate settings are often
 * used to control the use of periodic endpoints, such as by having
 * different endpoints use different amounts of reserved USB bandwidth.
 * All standards-conformant USB devices that use isochronous endpoints
 * will use them in non-default settings.
 *
 * The USB specification says that alternate setting numbers must run from
 * 0 to one less than the total number of alternate settings.  But some
 * devices manage to mess this up, and the structures aren't necessarily
 * stored in numerical order anyhow.  Use usb_altnum_to_altsetting() to
 * look up an alternate setting in the altsetting array based on its number.
 */
```

```
140 struct usb_interface {
141         /* array of alternate settings for this interface,
142          * stored in no particular order */
143         struct usb_host_interface *altsetting;
144
145         struct usb_host_interface *cur_altsetting;      /* the currently
146                                          * active alternate setting */
147         unsigned num_altsetting;        /* number of alternate settings */
148
149         int minor;                      /* minor number this interface is
150                                          * bound to */
151         enum usb_interface_condition condition;         /* state of binding */
152         unsigned is_active:1;           /* the interface is not suspended */
153         unsigned needs_remote_wakeup:1; /* driver requires remote wakeup */
154
155         struct device dev;              /* interface specific device info */
156         struct device *usb_dev;         /* pointer to the usb class's device,
if any */
157         int pm_usage_cnt;               /* usage counter for autosuspend */
158 };
```

生在红旗下，长在 21 世纪，我们的责任是重大的，没看那些专家都在慈眉善目忧国忧民的说，房子降价了我方经济要倒退多少年，我辈都是热血青年，好不容易有个报国的窗户，怎么着也得一起扶着房地产这个柱子，上头都说了，这是支柱产业，倒不得。不过，一屋不扫何以扫天下，精忠报国要从看代码开始，那个柱子先让专家们扶着，看代码也并不是容易的事，理解是万岁他爹，各有各的难处。

咱们这里瞅瞅，这么长的结构，猛的一看都要让人华丽丽的摔倒了，不过还好大都是注释，三分之一是海水，三分之二是火焰。话说回来，应该感到庆幸在看的是 USB 这块的代码，写代码的都比较的勤奋，不厌其烦的写了那么多的注释，当然不是说其它块的开发者不勤奋，因为时间和注释实在是一对不可调和的矛盾。

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。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 行的 num_altsetting 表示这个接口具有可选设置的数量。前面提过过 USB 设备的配置，那这里的设置是嘛意思？这可不是耗费一两个脑细胞就可以明白的了，不过不要怕，虽然说脑细胞是不可再生资源，但并不是多宝贵的东东，不然咋黄金煤炭的股票都在疯长，就不见人的工资长。

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

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。

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

```
brw-r----- 1 root disk 8,  0 Sep  26 09:17 /dev/sda
brw-r----- 1 root disk 8,  1 Sep  26 09:17 /dev/sda1
crw-r----- 1 root tty 4,  1 Sep  26 09:17 /dev/tty1
```

这是在我的系统里使用 ls –l 命令查看的，当然只是显示了其中的几个来表示表示而已。作为改革开放春风沐浴下的新一代年轻人，咱们对数字都是比较敏感的，一眼就能看到，在每一行的日期前面有两个逗号隔开的数字，对于普通文件而言这个位置显示的是文件的长度，而对于设备文件，这里显示的两个数字表示了该设备的主设备号和次设备号。一般来说，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。也就是说，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。上面就显示了俺的移动硬盘主设备号为 8，系统里 tty 设备的主设备为 4。

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。

```
localhost:/usr/src/linux/drivers/usb/core # cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 14 sound
 29 fb
116 alsa
128 ptm
136 pts
162 raw
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。

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。

说完了设备号，回到 struct usb_interface 的 151 行，condition 字段表示接口和驱动的绑定状态，enum usb_interface_condition 类型，在 include/linux/usb.h 里定义

```
83 enum usb_interface_condition {
84         USB_INTERFACE_UNBOUND = 0,
85         USB_INTERFACE_BINDING,
86         USB_INTERFACE_BOUND,
87         USB_INTERFACE_UNBINDING,
88 };
```

前面说linux设备模型的时候说了，设备和驱动是相生相依的关系，总线上的每个设备和驱动都在等待着命中的那个她，找到了，执子之手与子偕老，找不到，孤苦伶仃北冰洋。enum usb_interface_condition形象的描绘了这个过程中接口的个中心情，孤苦、期待、幸福、分开，人生又何尝不是如此？

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 行 is_active 表示接口是不是处于挂起状态。153 行 needs_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。157 行 pm_usage_cnt，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。

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。

设置

最近看了一些韩国的反转剧，什么是反转剧？就是影片儿演到一半时如果让你猜最后的结局，十有八九都会出乎你的预料。虽然每片只有短短 20 多分钟，但论故事情节，比咱们的帘子幽梦之类的梦了几十集还不知道做啥的精彩的多。

咱们在生活中是不会有这般戏剧性的反转和悬念的，有的只是吃饭的人生，上班的人生，睡觉的人生。思源湖边做了多少次的白日梦毕业时就被扔到了湖水里，然后我们从平淡走向平庸。

不过这里还是有点小悬念的，前面 struct usb_interface 里表示接口设置的 struct usb_host_interface 就被有意无意的飘过了，咱们在这里看看它的真面目，同样在 include/linux/usb.h 文件里定义。

```
69 /* host-side wrapper for one interface setting's parsed descriptors */
70 struct usb_host_interface {
71         struct usb_interface_descriptor desc;
72
73         /* array of desc.bNumEndpoint endpoints associated with this
74          * interface setting.  these will be in no particular order.
75          */
76         struct usb_host_endpoint *endpoint;
77
78         char *string;           /* iInterface string, if present */
79         unsigned char *extra;   /* Extra descriptors */
80         int extralen;
81 };
```

71 行，desc，接口的描述符。什么叫描述符？我们的生活就是一个不断的遇到人认识人的过程，有些人注定只是擦肩而过，有些人却深深的留在我们的内心里，比如 USB 的描述符。实际上，usb 的描述符是一个带有预定义格式的数据结构，里面保存了 usb 设备的各种属性还有相关信息，姓甚名谁啊，哪儿生产的啊等等，我们可以通过向设备请求获得它们的内容来深刻的了解感知一个 usb 设备。主要有四种 usb 描述符，设备描述符，配置描述符，接口描述符和端点描述符，协议里规定一个 usb 设备是必须支持这四大描述符的，当然也有其它一些描述符来让设备可以显得个性些，但这四大描述符是一个都不能少的。

这些描述符放哪儿？当然是在设备里。就好像你要把身份证放自己身上以免在哪里心情舒畅的散步时被新时代最可爱的人警察叔叔查到一样，你不会直接放他们那儿吧，然后在他们亲切慈祥的向你招手时，告诉他们说不就在你那儿么，那样的话等待你又是个什么样的结果，我不知道，我想你也不会想知道。咱们的描述符就在设备里，等着主机去拿。具体在哪儿？usb 设备里都会有一个叫 EEPROM 的东东，没错，就是放在它那儿，它就是用来存储设备本身信息的。如果你的脑海里还残存着一些大学里的美好时光的话，应该还会记得 EEPROM

就是电可擦写的可编程ROM，它与Flash虽说都是要电擦除的，但它可以按字节擦除，Flash只能一次擦除一个block，所以如果要改动比较少的数据的话，使用它还是比较合适的，但是世界上没有完美的东西，此物成本相对Flash比较高，所以一般来说usb设备里只拿它来存储一些本身特有的信息，要想存储数据，还是用Flash吧。

具体到接口描述符，它当然就是描述接口本身的信息的。一个接口可以有多个设置，使用不同的设置，描述接口的信息会有些不同，所以接口描述符并没有放在struct usb_interface结构里，而是放在表示接口设置的struct usb_host_interface结构里。定义在include/linux/usb/ch9.h文件里

```
294 /* USB_DT_INTERFACE: Interface descriptor */
295 struct usb_interface_descriptor {
296         __u8  bLength;
297         __u8  bDescriptorType;
298
299         __u8  bInterfaceNumber;
300         __u8  bAlternateSetting;
301         __u8  bNumEndpoints;
302         __u8  bInterfaceClass;
303         __u8  bInterfaceSubClass;
304         __u8  bInterfaceProtocol;
305         __u8  iInterface;
306 } __attribute__ ((packed));
```

又看到了__attribute__，不过这里改头换面成了__attribute__ ((packed))，意思就是告诉编译器，这个结构的元素都是1字节对齐的，不要再添加填充位了。因为这个结构和spec里的Table 9.12是完全一致的，包括字段的长度，如果不给编译器这么个暗示，编译器就会依据你平台的类型在结构的每个元素之间添加一定的填充位，如果你拿这个添加了填充位的结构去向设备请求描述符，你想想会是什么结果。

296行，bLength，描述符的字节长度。协议里规定，每个描述符必须以一个字节打头来表明描述符的长度。那可以扳着指头数一下，接口描述符的bLength应该是9，两个巴掌就数完了，没错，ch9.h文件里紧挨着接口描述符的定义就定义了这个长度

```
308 #define USB_DT_INTERFACE_SIZE           9
```

297行，bDescriptorType，描述符的类型。各种描述符的类型都在ch9.h文件里有定义，对应spec中的Table 9.5。对于接口描述符来说，值为USB_DT_INTERFACE，也就是0x04。

Table 9-5. Descriptor Types

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER[1]	8

299 行，bInterfaceNumber，接口号。每个配置可以包含多个接口，这个值就是它们的索引值。

300 行，bAlternateSetting，接口使用的是哪个可选设置。协议里规定，接口默认使用的设置总为 0 号设置。

301 行，bNumEndpoints，接口拥有的端点数量。这里并不包括端点 0，端点 0 是所有的设备都必须提供的，所以这里就没必要多此一举的包括它了。

302 行，bInterfaceClass，303 行 bInterfaceSubClass，304 行 bInterfaceProtocol。这个世界上有许许多多的 usb 设备，它们各有各的特点，为了区分它们，usb 规范，或者说 usb 协议，把 usb 设备分成了很多类，然而每个类又分成子类，这很好理解，我们一个大学也是如此，先是分成很多个学院，然后每个学院又被分为很多个系，然后可能每个系下边又分了各个专业，usb 协议也是这样干的，首先每个 Device 或 Interface 属于一个 Class，然后 Class 下面又分了 SubClass，完了 SubClass 下面又按各种设备所遵循的不同的通信协议继续细分。usb 协议里边为每一种 Class，每一种 SubClass，每一种 Protocol 定义一个数值，比如 mass storage 的 Class 就是 0x08，hub 的 Class 就是 0x09。

305 行，iInterface，接口对应的字符串描述符的索引值。疑？这里怎么又跳出来一个叫字符串描述符的东东？你没看错我也没说错，除了前面提到的四大描述符，是还有字符串描述符，不过那四大描述符是每个设备必须支持的，这个字符串描述符却是可有可无的，有了你欢喜我也欢喜，没有也不是什么问题。使用 lsusb 命令看一下

```
localhost:/usr/src/linux/drivers/usb/core # lsusb
Bus 001 Device 013: ID 04b4:1081 Cypress Semiconductor Corp.
Bus 001 Device 001: ID 0000:0000
```

第一行里显示的是我手边儿的 Cypress USB 开发板，看里面的 Cypress Semiconductor Corp.，这么一长串的东东从哪里来？是不是应该从设备里来？设备的那几个标准描述符，整个描述符的大小也不一定放得下这么一长串，所以，一些设备专门准备了一些字符串描述符（string descriptor），就用来记这些长串的东西。字符串描述符主要就是提供一些设备接口相关的描述性信息，比如厂商的名字，产品序列号等等。字符串描述符当然可以有多个，这里的索引值就是用来区分它们的。

说过了接口描述符，回到struct usb_host_interface的 76 行，endpoint，一个数组，表示这个设置所使用到端点。至于端点的结构struct usb_host_endpoint，天这么热，让它先一边儿凉快凉快吧，不怕春光乍泄的话，可以去思源湖冲个凉，咱们先看完struct usb_host_interface再去说它。

78 行，string，用来保存从设备里取出来的字符串描述符信息的，既然字符串描述符可有可无，那这里的指针也有可能为空了。

79 行，extra，80 行 extralen，有关额外的描述符的。除了前面提到的四大描述符还有字符串描述符外，还有为一组设备也就是一类设备定义的描述符，和厂商为设备特别定义的描述符，extra 指的就是它们，extralen 表示它们的长度。比如上海规定了，社保必须得交多少多少，公积金多少多少，有个最低的比例，有地儿觉得太少，给你多交些，叫补充什么金的，还有些地儿，觉得补充都不过瘾，像公务员这种特殊行业特别劳心劳力的，再加些特殊行业补贴等什么的，既规定了必须实现的，也给你特殊行业发挥的空间，当然怎么发挥就不是咱说了算了，不操那份儿心了。

端点

于丹说，生与死，是人生起始的两个端点。

于丹？不会不知道这个人吧，我哭。百家讲坛继易中天之后放的第二颗大卫星，最开始讲论语一下子就红了的那个，接着又讲了庄子。咱可以不喜欢但不可以落后，是不。她在《庄子心得》的谈笑论生死里说了个寓言，我这儿给转一下。

兄弟两个人，他们家住在一座摩天大楼的第 80 层。这天，两个人深夜回家，恰好忘记了看通知，电梯停了。

兄弟俩背着沉重的大背包，在楼底下商量一下，决定一鼓作气，爬楼梯回家。两人抖擞精神，开始爬楼。爬到 20 楼的时候，开始觉得背包很重了。两人商量，决定把背包存在 20 楼，到时候再回过头来取。卸下了背包，两个人觉得很轻松，说说笑笑地继续往上爬。

爬到 40 楼的时候，两人已经很累了，就开始互相抱怨指责。哥哥说：你为什么不看通知啊？弟弟说：我忘了看通知这件事，你怎么不提醒我呢？两个人就这样吵吵闹闹，一路吵到 60 层。

到了这时候，两人实在疲惫不堪，终于懒得吵了，觉得还是应该安安静静地继续爬楼。当他们终于爬完了最后 20 层，来到了家门口的时候，两个人互相一看，不约而同想起了一件事：钥匙忘在 20 楼了，在背包里。

其实，这说的就是人的一生。

这是于丹比的人生，咱们的不是，她和咱们隔了老远去了，咱们是前二十年无忧无虑，谈谈情说说爱没啥包袱，说说笑笑的爬了 20 年后，发现人生是需要找工作，需要赚钱，需要买房子的，于是自己捡的，或者被和谐过来的，一个接一个的包袱往身上丢，越来越沉重的向上爬，爬到 40 年、60 年怎么样？不用去想它，也就一个字，累，两个字，很累，三个字，非常累，四个字，累死人了。

折腾 USB spec 的同志应该不会读过庄子，也不会知道于丹这个人物，可别人也知道端点，于是端点成了 USB 数据传输的终点。看看它在内核里的定义

```
46 /**
47  * struct usb_host_endpoint - host-side endpoint descriptor and queue
48  * @desc: descriptor for this endpoint, wMaxPacketSize in native byteorder
49  * @urb_list: urbs queued to this endpoint; maintained by usbcore
50  * @hcpriv: for use by HCD; typically holds hardware dma queue head (QH)
51  *      with one or more transfer descriptors (TDs) per urb
52  * @ep_dev: ep_device for sysfs info
53  * @extra: descriptors following this endpoint in the configuration
54  * @extralen: how many bytes of "extra" are valid
55  *
56  * USB requests are always queued to a given endpoint, identified by a
57  * descriptor within an active interface in a given USB configuration.
58  */
59 struct usb_host_endpoint {
60         struct usb_endpoint_descriptor  desc;
61         struct list_head                urb_list;
62         void                            *hcpriv;
63         struct ep_device                *ep_dev;        /* For sysfs info */
64
65         unsigned char *extra;   /* Extra descriptors */
66         int extralen;
67 };
```

60 行，desc，端点描述符，四大描述符的第二个隆重登场了。它也在

include/linux/usb/ch9.h里定义

```
312 /* USB_DT_ENDPOINT: Endpoint descriptor */
313 struct usb_endpoint_descriptor {
314         __u8  bLength;
315         __u8  bDescriptorType;
316
317         __u8  bEndpointAddress;
318         __u8  bmAttributes;
319         __le16 wMaxPacketSize;
320         __u8  bInterval;
321
322         /* NOTE:  these two are _only_ in audio endpoints. */
323         /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
324         __u8  bRefresh;
325         __u8  bSynchAddress;
326 } __attribute__ ((packed));
327
328 #define USB_DT_ENDPOINT_SIZE            7
329 #define USB_DT_ENDPOINT_AUDIO_SIZE      9       /* Audio extension */
```

这个结构与 spec 中的 Table 9.13 是一一对应的，0 号端点仍然保持着它特殊的地位，它没有自己的端点描述符。

314 行，bLength，描述符的字节长度，数一下，前边儿有 7 个，后边儿又多了两个字节，那是针对音频设备扩展的，不用管它，紧接着 struct usb_host_endpoint 定义的就是两个长度值的定义。

315 行，bDescriptorType，描述符类型，这里对于端点就是 USB_DT_ENDPOINT，0x05。

317 行，bEndpointAddress，这个字段描述的信息挺多的，比如这个端点是输入端点还是输出端点，这个端点的地址，以及这个端点的端点号。它的bits 0~3 表示的就是端点号，你使用 0x0f和它相与就可以得到端点号。不过，开发内核的同志想的都很周到，定义好了一个掩码USB_ENDPOINT_NUMBER_MASK，它的值就是 0x0f，当然，这是为了让咱们更容易去读他们的代码，也为了以后的扩展。另外，它的bit 8 表示方向，输入还是输出，同样有掩码 USB_ENDPOINT_DIR_MASK，值为 0x80，将它和bEndpointAddress相与，并结合USB_DIR_IN和USB_DIR_OUT作判断就可以得到端点的方向。

```
42 /*
43  * USB directions
44  *
45  * This bit flag is used in endpoint descriptors' bEndpointAddress field.
46  * It's also one of three fields in control requests bRequestType.
47  */
```

```
48 #define USB_DIR_OUT                     0               /* to device */
49 #define USB_DIR_IN                      0x80            /* to host */
```

318 行，bmAttributes，属性，总共 8 位，其中bit1 和bit0 共同称为Transfer Type,即传输类型， 00 表示控制，01 表示等时，10 表示批量，11 表示中断。前面的端点号还有端点方向都有配对儿的掩码，这里当然也有，就在struct usb_endpoint_descriptor定义的下面

```
338 #define USB_ENDPOINT_XFERTYPE_MASK      0x03    /* in bmAttributes */
339 #define USB_ENDPOINT_XFER_CONTROL       0
340 #define USB_ENDPOINT_XFER_ISOC          1
341 #define USB_ENDPOINT_XFER_BULK          2
342 #define USB_ENDPOINT_XFER_INT           3
```

319 行，wMaxPacketSize，端点一次可以处理的最大字节数。比如你老板比较看重你，一次给你交代了几个任务，于是你大声的疾呼，神啊，我一次只能做一个，当然神是听不到的，怎么办那，加班加点儿，一个一个的分开做呗。端点也是，如果你发送的数据量大于端点的这个值，也会分成多次一次一次来传输。友情提醒一下，这个字段还是有点门道的，对不同的传输类型也有不同的要求，日后碰到了再说。

320 行，bInterval， USB 是轮询式的总线，这个值表达了端点一种美好的期待，希望主机轮询自己的时间间隔，但实际上批准不批准就是 host 的事了。不同的传输类型 bInterval 也有不同的意义，暂时就提这么一下，碰到各个实际的传输类型了再去说它。不是俺捂盘惜售，而是初次照面儿就对人家寻根问底的不大礼貌，这里先留个印象，有缘总会再相见的。

回到struct usb_host_endpoint，61 行，urb_list，端点要处理的urb队列。urb是什么？这年头儿钱不多就是新名词儿多，是个新名词爆炸的时代，不过urb可是usb通信的主角，它包含了执行urb传输所需要的所有信息，你要想和你的usb通信，就得创建一个urb，并且为它赋好值，交给咱们的usb core，它会找到合适的host controller，从而进行具体的数据传输。设备中的每个端点都可以处理一个urb队列，当然，urb是内核里对usb传输数据的封装也叫抽象吧，协议里可不这么叫。基于urb特殊的江湖地位，接下来的哪一个黄道吉日里，我会对它大书特书的。

62 行，hcpriv，这是提供给 HCD（host controller driver）用的。比如等时端点会在里边儿放一个 ehci_iso_stream，什么意思？郑板桥告诉我们要难得糊涂。

63 行，ep_dev，这个字段是供 sysfs 用的。好奇的话可以去/sys 下看一看

```
localhost:/usr/src/linux/drivers/usb/core # ls /sys/bus/usb/devices/usb1/ep_00/
bEndpointAddress  bmAtributes   direction   subsystem    wMaxpacketSize
bInterval         dev           interval    type
bLength           device        power       uevent
```

ep_00 端点目录下的这些文件从哪儿来的？就是在 usb_create_ep_files 函数里使用 ep_dev 创建的。

65 行，extra，66 行，extralen，有关一些额外扩展的描述符的，和struct usb_host_interface里差不多，只是这里的是针对端点的，如果你请求从设备里获得描述符信息，它们会跟在标准的端点描述符后面返回给你。

设备

第一眼看到struct usb_device这个结构，我仿佛置身于衡山路的酒吧里，盯着舞池里扭动的符号，眼神迷离。

交大里苟了几年，毕业了又是住在学校附近的徐虹北路上，沿着虹桥路走过去，到徐家汇不过 10 多分钟，再溜达几步就可以到衡山路。学校里睡的就比较晚，毕业了仍然一样，其它好习惯还是坏习惯扔掉了不少，就这个保持的还不错，于是经常穷极无聊的晚上，只好和同学沿着虹桥路也好，沿着番愚路再走到广元路或淮海路也好，慢慢的向东走。走着走着就会到衡山路，有时遇到许多老年人在衡山电影院对面的小广场上跳各种各样的舞，就会在旁边一曲一曲的看，感慨咱们爬到 60 岁，爬到那般年纪是否会有那样的乐趣。有时想疯了，就会到旁边的酒吧坐坐，麻醉一下别人眼里的自己。有时只是在旁边儿安静的巷子里四处走走。

衡山路的夜晚杂乱而又冗长，struct usb_device结构冗长而又杂乱。衡山路还是会去，struct usb_device还是得说。

```
328 /*
329  * struct usb_device - kernel's representation of a USB device
330  *
331  * FIXME: Write the kerneldoc!
332  *
333  * Usbcore drivers should not set usbdev->state directly.  Instead use
334  * usb_set_device_state().
335  */
336 struct usb_device {
337         int             devnum;         /* Address on USB bus */
338         char            devpath [16];   /* Use in messages: /port/port/... */
339         enum usb_device_state   state;  /* configured, not attached, etc */
340         enum usb_device_speed   speed;  /* high/full/low (or error) */
341
342         struct usb_tt   *tt;            /* low/full speed dev, highspeed hub
*/
343         int             ttport;         /* device port on that tt hub */
```

```
344
345         unsigned int toggle[2];         /* one bit for each endpoint
346                                          * ([0] = IN, [1] = OUT) */
347
348         struct usb_device *parent;      /* our hub, unless we're the root */
349         struct usb_bus *bus;            /* Bus we're part of */
350         struct usb_host_endpoint ep0;
351
352         struct device dev;              /* Generic device interface */
353
354         struct usb_device_descriptor descriptor;/* Descriptor */
355         struct usb_host_config *config; /* All of the configs */
356
357         struct usb_host_config *actconfig;/* the active configuration */
358         struct usb_host_endpoint *ep_in[16];
359         struct usb_host_endpoint *ep_out[16];
360
361         char **rawdescriptors;          /* Raw descriptors for each config */
362
363         unsigned short bus_mA;          /* Current available from the bus */
364         u8 portnum;                     /* Parent port number (origin 1) */
365         u8 level;                       /* Number of USB hub ancestors */
366
367         unsigned discon_suspended:1;    /* Disconnected while suspended */
368         unsigned have_langid:1;         /* whether string_langid is valid */
369         int string_langid;              /* language ID for strings */
370
371         /* static strings from the device */
372         char *product;                  /* iProduct string, if present */
373         char *manufacturer;             /* iManufacturer string, if present */
374         char *serial;                   /* iSerialNumber string, if present */
375
376         struct list_head filelist;
377 #ifdef CONFIG_USB_DEVICE_CLASS
378         struct device *usb_classdev;
379 #endif
380 #ifdef CONFIG_USB_DEVICEFS
381         struct dentry *usbfs_dentry;    /* usbfs dentry entry for the device
*/
382 #endif
383         /*
384          * Child devices - these can be either new devices
385          * (if this is a hub device), or different instances
386          * of this same device.
```

```
387          *
388          * Each instance needs its own set of data structures.
389          */
390
391         int maxchild;                   /* Number of ports if hub */
392         struct usb_device *children[USB_MAXCHILDREN];
393
394         int pm_usage_cnt;               /* usage counter for autosuspend */
395         u32 quirks;                     /* quirks of the whole device */
396
397 #ifdef CONFIG_PM
398         struct delayed_work autosuspend; /* for delayed autosuspends */
399         struct mutex pm_mutex;          /* protects PM operations */
400
401         unsigned long last_busy;        /* time of last use */
402         int autosuspend_delay;          /* in jiffies */
403
404         unsigned auto_pm:1;             /* autosuspend/resume in progress */
405         unsigned do_remote_wakeup:1;    /* remote wakeup should be enabled */
406         unsigned autosuspend_disabled:1; /* autosuspend and autoresume */
407         unsigned autoresume_disabled:1;  /*  disabled by the user */
408 #endif
409 };
```

337 行，devnum，设备的地址。此地址非彼地址，和咱们写程序时说的地址是不一样的，devnum 只是 usb 设备在一条 usb 总线上的编号。你的 usb 设备插到 hub 上时，hub 观察到这个变化，于是来了精神，会在一个漫长而又曲折的处理过程中调用一个名叫 choose_address 的函数，为你的设备选择一个地址。就像在那个浪漫的季节的一个温馨的下午，你去吃港汇下边儿的那个必胜客，同样会领取一个属于自己的编号陪伴自己度过一个漫长的过程。有人说我没有用 hub，我的 usb 设备直接插到主机的 usb 接口上了。我哭，即使你没有用 hub，也总要明白主机里还会有个叫 root hub 的东东吧，不管是一般的 hub 还是 root hub，你的 usb 设备总要通过一个 hub 才能在 usb 的世界里生活。

现在来认识一下 usb 子系统里面关于地址的游戏规则。在 usb 世界里，一条总线就是大树一棵，一个设备就是叶子一片。为了记录这棵树上的每一个叶子节点，每条总线设有一个地址映射表，即 struct usb_bus 结构体里有一个成员 struct usb_devmap devmap，

```
268 /* USB device number allocation bitmap */
269 struct usb_devmap {
270         unsigned long devicemap[128 / (8*sizeof(unsigned long))];
271 };
```

什么是usb_bus？前面不是已经有了一个struct bus_type类型的 usb_bus_type了么？没错，在usb子系统的初始化函数usb_init里已经注册了 usb_bus_type，不过那是让系统

知道有这么一个类型的总线。而一个总线类型和一条总线是两码子事儿。从硬件上来讲，一个host controller就会连出一条usb总线，而从软件上来讲，不管你有多少个host controller，或者说有多少条总线，它们通通属于usb_bus_type这么一个类型，只是每一条总线对应一个struct usb_bus结构体变量，这个变量在host controller的驱动程序中去申请。

上面的 devmap 地址映射表就表示了一条总线上设备连接的情况，假设 unsigned long=4bytes，那么 unsigned long devicemap[128/(8*sizeof(unsigned long))]就等价于 unsigned long devicemap[128/(8*4)]，进而等价于 unsigned long devicemap[4]，而 4bytes 就是 32 个 bits，因此这个数组最终表示的就是 128 个 bits。而这也对应于一条总线可以连接 128 个 usb 设备。之所以这里使用 sizeof(unsigned long)，就是为了跨平台应用，不管 unsigned long 到底是几，总之这个 devicemap 数组最终可以表示 128 位，也就是说每条总线上最多可以连上 128 个设备。

338 行，devpath [16]，它显然是用来记录一个字符串的，这个字符串啥意思？给你看个直观的东西

```
localhost:~ # ls /sys/bus/usb/devices/
1-0:1.0  2-1      2-1:1.1  4-0:1.0  4-5:1.0  usb2  usb4
2-0:1.0  2-1:1.0  3-0:1.0  4-5      usb1     usb3
```

Sysfs 文件系统下，我们看到这些乱七八糟的东西，它们都是啥？usb1/usb2/usb3/usb4 表示哥们的计算机上接了 4 条 usb 总线，即 4 个 usb 主机控制器，事物多了自然就要编号，就跟我们中学或大学里面的学号一样，就是用于区分多个个体，而 4-0:1.0 表示什么？4 表示是 4 号总线，或者说 4 号 Root Hub，0 就是这里我们说的 devpath，1 表示配置为 1 号，0 表示接口号为 0。也即是说，4 号总线的 0 号端口的设备，使用的是 1 号配置，接口号为 0。那么 devpath 是否就是端口号呢？显然不是，这里我列出来的这个例子是只有 Root Hub 没有级联 Hub 的情况，如果在 Root Hub 上又接了别的 Hub，然后一级一级连下去，子又生孙，孙又生子，子又有子，子又有孙。子子孙孙，无穷匮也。那么如何在 sysfs 里面来表征这整个大家族呢？这就是 devpath 的作用，顶级的设备其 devpath 就是其连在 Root Hub 上的端口号，而次级的设备就是其父 Hub 的 devpath 后面加上其端口号，即如果 4-0:1.0 如果是一个 Hub，那么它下面的 1 号端口的设备就可以是 4-0.1:1.0，2 号端口的设备就可以是 4-0.2:1.0，3 号端口就可以是 4-0.3:1.0。总的来说，就是端口号一级一级往下加。这个思想是很简单的，也是很朴实的。

339 行，state，设备的状态。这是个枚举类型

```
557 enum usb_device_state {
558         /* NOTATTACHED isn't in the USB spec, and this state acts
559          * the same as ATTACHED ... but it's clearer this way.
560          */
561         USB_STATE_NOTATTACHED = 0,
562
```

```
563         /* chapter 9 and authentication (wireless) device states */
564         USB_STATE_ATTACHED,
565         USB_STATE_POWERED,                      /* wired */
566         USB_STATE_UNAUTHENTICATED,              /* auth */
567         USB_STATE_RECONNECTING,                 /* auth */
568         USB_STATE_DEFAULT,                      /* limited function */
569         USB_STATE_ADDRESS,
570         USB_STATE_CONFIGURED,                   /* most functions */
571
572         USB_STATE_SUSPENDED
573
574         /* NOTE:  there are actually four different SUSPENDED
575          * states, returning to POWERED, DEFAULT, ADDRESS, or
576          * CONFIGURED respectively when SOF tokens flow again.
577          */
578 };
```

上面定义了 9 种状态，spec 里只定义了 6 种，Attached，Powered，Default，Address，Configured，Suspended，对应于 Table 9.1。

Attached 表示设备已经连接到 usb 接口上了，是 hub 检测到设备时的初始状态。那么这里所谓的 USB_STATE_NOTATTACHED 就是表示设备并没有 Attached。

Powered 是加电状态。USB 设备的电源可以来自外部电源，协议里叫做 self-powered，也可以来自 hub，叫 bus-powered。尽管 self-powered 的 USB 设备可能在连接上 USB 接口以前已经上电，但它们直到连上 USB 接口后才能被看作是 Powered 的，你觉得它已经上电了那是站在你的角度看，可是现在你看的是 usbcore，所以要放弃个人的成见，团结在 core 的周围。

Default 缺省状态，在 Powered 之后，设备必须在收到一个复位（reset）信号并成功复位后，才能使用缺省地址回应主机发过来的设备和配置描述符的请求。

Address 状态表示主机分配了一个唯一的地址给设备，此时设备可以使用缺省管道响应主机的请求。真羡慕这些 usb 设备，住的地方都是包分配的，哪像咱们辛辛苦苦一路小跑着也不一定能达到 Address 状态。

Configured 状态表示设备已经被主机配置过了，也就是协议里说的处理了一个带有非 0 值的 SetConfiguration()请求，此时主机可以使用设备提供的所有功能。

Suspended 挂起状态，为了省电，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。此时，usb 设备要自己维护包括地址、配置在内的信息。

USB 设备从生到死都要按照这么几个状态，遵循这么一个过程。它不可能像咱们的房价，林志伶的胸部一样跳跃式的发展。

340 行，speed，设备的速度，这也是个枚举变量

```
548 /* USB 2.0 defines three speeds, here's how Linux identifies them */
549
550 enum usb_device_speed {
551         USB_SPEED_UNKNOWN = 0,                  /* enumerating */
552         USB_SPEED_LOW, USB_SPEED_FULL,          /* usb 1.1 */
553         USB_SPEED_HIGH,                         /* usb 2.0 */
554         USB_SPEED_VARIABLE,                     /* wireless (usb 2.5) */
555 };
```

地球人都知道，USB 设备有三种速度，低速，全速，高速。USB1.1 那会儿只有低速，全速，后来才出现了高速，就是所谓的 480Mbps/s。这里还有个 USB_SPEED_VARIABLE，是无线 USB 的，号称 usb 2.5，还在发展中，据说小黑的 T61 已经支持了，向往 ing。USB_SPEED_UNKNOWN 只是表示现阶段还不知道这个设备究竟什么速度。

342 行，tt，343 行，ttport。知道 tt 干嘛的吗？tt 叫做 transaction translator。你可以把它想成一块特殊的电路，是 hub 里面的电路，确切的说是高速 hub 中的电路，我们知道 usb 设备有三种速度的，分别是 low speed，full speed，high speed。即所谓的低速/全速/高速，抗日战争那会儿，这个世界上只有 low speed/full speed 的设备，没有 high speed 的设备，后来解放后，国民生产力的大幅度提升催生了一种 high speed 的设备，包括主机控制器，以前只有两种接口的，OHCI/UHCI，这都是在 usb spec 1.0 的时候，后来 2.0 推出了 EHCI，高速设备应运而生。Hub 也有高速 hub 和过去的 hub，但是这里就有一个兼容性问题了，高速的 hub 是否能够支持低速/全速的设备呢？一般来说是不支持的，于是有了一个叫做 TT 的电路，它就负责高速和低速/全速的数据转换，于是，如果一个高速设备里有这么一个 TT，那么就可以连接低速/全速设备，如不然，那低速/全速设备没法用，只能连接到 OHCI/UHCI 那边出来的 hub 口里。

345 行，toggle[2]，这个数组只有两个元素，分别对应 IN 和 OUT 端点，每一个端点占一位。似乎这么说仍是在雾中看花，黑格尔告诉我们，存在就是有价值的，那么这个数组存在的价值是什么？一言难尽，说来话长，那就长话长说好了。

咱们前边儿说，你要想和你的 usb 通信，创建一个 urb，为它赋好值，交给咱们的 usb core 就可以了。这个 urb 是站在咱们的角度，实际上在 usb cable 里流淌的根本就不是那么回事儿，咱们提交的是 urb，usb cable 里流淌的是一个一个的数据包（packet），就像咱们吃的是社会主义的粮，身体里流淌的是无产阶级的鲜血。咱们无产阶级的鲜血里，虽说不包括房产财产什么的，但是还是有许多的成分一定的结构的，usb 底层传输的 packets 也一样。

咱们凄苦的人生是从第一声哭开始，所有的 packets 都从一个 SYNC 同步字段开始，SYNC 是一个 8 位长的二进制串，只是用来同步用的，它的最后两位标志了 SYNC 的结束和 PID（Packet Identifer）的开始，就像咱们的大四标志了梦想的结束和现实的开始。PID 也是一个 8 位的二进制串，前四位用来区分不同的 packet 类型，后面四位只是前四位的反码，校验用的。packet 的类型主要有四种，在 spec 中的 Table 8-1 里有说明

Table 8-1. PID Types

PID Type	PID Name	PID<3:0>*	Description
Token	OUT	0001B	Address + endpoint number in host-to-function transaction
	IN	1001B	Address + endpoint number in function-to-host transaction
	SOF	0101B	Start-of-Frame marker and frame number
	SETUP	1101B	Address + endpoint number in host-to-function transaction for SETUP to a control pipe
Data	DATA0	0011B	Data packet PID even
	DATA1	1011B	Data packet PID odd
	DATA2	0111B	Data packet PID high-speed, high bandwidth isochronous transaction in a microframe (see Section 5.9.2 for more information)
	MDATA	1111B	Data packet PID high-speed for split and high bandwidth isochronous transactions (see Sections 5.9.2, 11.20, and 11.21 for more information)
Handshake	ACK	0010B	Receiver accepts error-free data packet
	NAK	1010B	Receiving device cannot accept data or transmitting device cannot send data
	STALL	1110B	Endpoint is halted or a control pipe request is not supported
	NYET	0110B	No response yet from receiver (see Sections 8.5.1 and 11.17-11.21)
Special	PRE	1100B	(Token) Host-issued preamble. Enables downstream bus traffic to low-speed devices.
	ERR	1100B	(Handshake) Split Transaction Error Handshake (reuses PRE value)
	SPLIT	1000B	(Token) High-speed Split Transaction Token (see Section 8.4.2)
	PING	0100B	(Token) High-speed flow control probe for a bulk/control endpoint (see Section 8.5.1)
	Reserved	0000B	Reserved PID

主机和设备都是纯理性的东东，完全通过 PID 来判断送过来的 packet 是不是自己所需要的，不像咱们，往往缺乏这么一个用来判断的标准，不知道自己究竟需要的是什么。PID 之后紧跟着的是地址字段，每个 packet 都需要知道自己要往哪里去，它们是一个一个目的明确的精灵，行走在 usb cable 里，而我们的前方在哪里？这个地址实际上包括两部分，7 位表示了总线上连接的设备或接口的地址，4 位表示端点的地址，这就是为什么前面说每条

usb 总线最多只能有 128 个设备，即使是高速设备除了 0 号端点也最多只能有 15 个 in 端点和 15 个 out 端点。地址字段再往后是 11 位的帧号（frame number），值达到 7FFH 时归零，像一个个无聊的夜晚一样循环往复。这个帧号并不是每一个 packet 都会有，它只在每帧或微帧（Mircoframe）开始的 SOF Token 包里发送。帧是对于低速和全速模式来说的，一帧就是 1ms，对于高速模式的称呼是微帧，一个微帧为 125 微妙，每帧或微帧当然不会只能传一个 packet。帧号再往后就是千呼万唤始出来的 Data 字段了，它可以有 0 到 1024 个字节不等。最后还有 CRC 校验字段来做扫尾工作。

咱们要学习 packet，做一个有理想有目标的人，所以这里只看看 Data 类型的 packet。前面的 Table 8-1 里显示，有四种类型的 Data 包，DATA0，DATA1，DATA2 和 MDATA。存在就是有价值的，这里分成 4 种数据包自然有里面的道理，其中 DATA0 和 DATA1 就可以用来实现 data toggle 同步，看到 toggle，好像有点接近不久之前留下的疑问了。

对于批量传输、控制传输和中断传输来说，数据包最开始都是被初始化为 DATA0 的，然后为了传输的正确性，就一次传 DATA0，一次传 DATA1，一旦哪次打破了这种平衡，主机就可以认为传输出错了。对于等时传输来说，data toggle 并不被支持。USB 就是在使用这种简单的哲学来判断对于错，而我们的生活中有的只是复杂，即使一个馒头都能引起一个两亿多的血案。

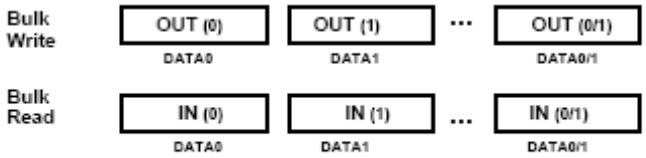


Figure 8-35. Bulk Reads and Writes

我们的 struct usb_device 中的数组 unsigned int toggle[2]就是为了支持这种简单的哲学而生的，它里面的每一位表示的就是每个端点当前发送或接收的数据包是 DATA0 还是 DATA1。

348 行，parent，struct usb_device 结构体的 parent 自然也是一个 struct usb_device 指针。对于 Root Hub，前面说过，它是和 Host Controller 是绑定在一起的，它的 parent 指针在 Host Controller 的驱动程序中就已经赋了值，这个值就是 NULL，换句话说，对于 Root Hub，它不需要再有父指针了，这个父指针就是给从 Root Hub 连出来的节点用的。USB 设备是从 Root Hub 开始，一个一个往外面连的，比如 Root Hub 有 4 个口，每个口连一个 USB 设备，比如其中有一个是 Hub，那么这个 Hub 有可以继续有多个口，于是一级一级的往下连，最终连成了一棵树。

349 行，bus，没什么说的，设备所在的那条总线。

350 行，ep0，端点 0 的特殊地位决定了她必将受到特殊的待遇，在 struct usb_device 对象产生的时候它就要初始化。

353 行，dev，嵌入到 struct usb_device 结构里的 struct device 结构。

354 行，desc，设备描述符，四大描述符的第三个姗姗而来。它在 include/linux/usb/ch9.h 里定义

```
203 /* USB_DT_DEVICE: Device descriptor */
204 struct usb_device_descriptor {
205         __u8  bLength;
206         __u8  bDescriptorType;
207
208         __le16 bcdUSB;
209         __u8  bDeviceClass;
210         __u8  bDeviceSubClass;
211         __u8  bDeviceProtocol;
212         __u8  bMaxPacketSize0;
213         __le16 idVendor;
214         __le16 idProduct;
215         __le16 bcdDevice;
216         __u8  iManufacturer;
217         __u8  iProduct;
218         __u8  iSerialNumber;
219         __u8  bNumConfigurations;
220 } __attribute__ ((packed));
221
222 #define USB_DT_DEVICE_SIZE              18
```

205 行，bLength，描述符的长度，可以自己数数，或者看紧接着的定义 USB_DT_DEVICE_SIZE。

206 行，bDescriptorType，这里对于设备描述符应该是 USB_DT_DEVICE，0x01。

208 行，bcdUSB，USB spec 的版本号，一个设备如果能够进行高速传输，那么它设备描述符里的 bcdUSB 这一项就应该为 0200H。

209 行，bDeviceClass，210 行，bDeviceSubClass，211 行，bDeviceProtocol，和接口描述符的意义差不多，前面说了这里就不再罗唆了。

212 行，bMaxPacketSize0，端点 0 一次可以处理的最大字节数，端点 0 的属性却放到设备描述符里去了，更加彰显了它突出的江湖地位，它和机器人公敌里的机器人 Sonny，交大南门外的老赵烤肉一样特别一样独一无二。

前面说端点的时候说了端点 0 并没有一个专门的端点描述符，因为不需要，基本上它所有的特性都在 spec 里规定好了的，然而，别忘了这里说的是“基本上”，有一个特性则是不

一样的，这叫做 maximum packet size，每个端点都有这么一个特性，即告诉你该端点能够发送或者接收的包的最大值。对于通常的端点来说，这个值被保存在该端点描述符中的 wMaxPacketSize 这一个 field，而对于端点 0 就不一样了，由于它自己没有一个描述符，而每个设备又都有这么一个端点，所以这个信息被保存在了设备描述符里，所以我们在设备描述符里可以看到这么一项，bMaxPacketSize0。而且 spec 还规定了，这个值只能是 8，16，32 或者 64 这四者之一，如果一个设备工作在高速模式，这个值还只能是 64，如果是工作在低速模式，则只能是 8，取别的值都不行。

213 行，idVendor，214 行，idProduct，分别是厂商和产品的 ID。

215 行，bcdDevice，设备的版本号。

216 行，iManufacturer，217 行，iProduct，218 行，iSerialNumber，分别是厂商，产品和序列号对应的字符串描述符的索引值。

219 行，bNumConfigurations，设备当前速度模式下支持的配置数量。有的设备可以在多个速度模式下操作，这里包括的只是当前速度模式下的配置数目，不是总的配置数目。

这就是设备描述符，它和spec Table 9-8 是一一对应的。咱们回到struct usb_device的 355 行，config，357 行，actconfig，分别表示设备拥有的所有配置和当前激活的，也就是正在使用的配置。usb设备的配置用struct usb_host_config结构来表示，下节再说。

358 行，ep_in[16]，359 行，ep_out[16]，除了端点 0，一个设备即使在高速模式下也最多只能再有 15 个 IN 端点和 15 个 OUT 端点，端点 0 太特殊了，对应的管道是 Message 管道，又能进又能出特能屈能伸的那种，所以这里的 ep_in 和 ep_out 数组都有 16 个值。

361 行，rawdescriptors，这是个字符指针数组，数组里的每一项都指向一个使用 GET_DESCRIPTOR 请求去获得配置描述符时所得到的结果。考虑下，为什么我只说得到的结果，而不直接说得到的配置描述符？不是请求的就是配置描述符么？这是因为你使用 GET_DESCRIPTOR 去请求配置描述符时，设备返回给你的不仅仅只有配置描述符，它把该配置所包括的所有接口的接口描述符，还有接口里端点的端点描述符一股脑的都塞给你了。第一个接口的接口描述符紧跟着这个配置描述符，然后是这个接口下面端点的端点描述符，如果有还有其它接口，它们的接口描述符和端点描述符也跟在后面，这里面，专门为一类设备定义的描述符和厂商定义的描述符跟在它们对应的标准描述符后面。这和我们去买水果，买了 5 斤苹果却只有 5 个真实天壤之别，现实生活中的愤懑在 USB 世界里得到了发泄。

这里提到了 GET_DESCRIPTOR 请求，就顺便简单提一下 USB 的设备请求（device request）。协议里说了，所有的设备通过缺省的控制管道来响应主机的请求，既然使用的是控制管道，那当然就是控制传输了，这些请求的底层 packet 属于 Setup 类型，前面的那张表里也可以看到它，在 Setup 包里包括了请求的各种参数。协议里同时也定义了一些标准的设备请求，并规定所有的设备必须响应它们，即使它们还处于 Default 或 Address

状态。这些标准的设备请求里，GET_DESCRIPTOR 就赫然在列。

363 行，bus_mA，这个值是在 host controller 的驱动程序中设置的，通常来讲，计算机的 usb 端口可以提供 500mA 的电流。

364 行，portnum，不管是 root hub 还是一般的 hub，你的 USB 设备总归要插在一个 hub 的端口上才能用，portnum 就是那个端口号。当然，对于 root hub 这个 usb 设备来说它本身没有 portnum 这么一个概念，因为它不插在别的 Hub 的任何一个口上。所以对于 Root Hub 来说，它的 portnum 在 Host Controller 的驱动程序里给设置成了 0。

365 行，level，层次，也可以说是级别，表征 usb 设备树的级连关系。Root Hub 的 level 当然就是 0，其下面一层就是 level 1，再下面一层就是 level 2，依此类推。

366 行，discon_suspended，Disconnected while suspended。

368 行，have_langid，369 行，string_langid，usb 设备里的字符串描述符使用的是 UNICODE 编码，可以支持多种语言，string_langid 就是用来指定使用哪种语言的，have_langid 用来判断 string_langid 是否有效。

372 行，product，373 行，manufacturer，374 行，serial，分别用来保存产品、厂商和序列号对应的字符串描述符信息。

376~382 行，usbfs 相关的，不可知的未来说 usbfs 的时候再聊它们。

391 行，maxchild，hub 的端口数，注意可不包括上行端口。

392 行，children[USB_MAXCHILDREN]，USB_MAXCHILDREN 是 include/linux/usb.h 里定义的一个宏，值为 31

```
315 /* This is arbitrary.
316  * From USB 2.0 spec Table 11-13, offset 7, a hub can
317  * have up to 255 ports. The most yet reported is 10.
318  *
319  * Current Wireless USB host hardware (Intel i1480 for example) allows
320  * up to 22 devices to connect. Upcoming hardware might raise that
321  * limit. Because the arrays need to add a bit for hub status data, we
322  * do 31, so plus one evens out to four bytes.
323  */
324 #define USB_MAXCHILDREN         (31)
```

其实 hub 可以接一共 255 个端口，不过实际上遇到的 usb hub 最多的也就是说自己支持 10 个端口的，所以 31 基本上够用了。

394 行，pm_usage_cnt，struct usb_interface 结构里也有，想知道吗？想知道回那儿看吧。

396 行，quirks，祭起我们法宝金山词霸看看，怪僻的意思，白了说就是大家的常用语“毛病”。本来指定 usb spec 就是让大家团结一致好办事，但总是有些厂商不太守规矩，拿出一些有点毛病的产品给我们用，你说它大毛病吧，也不是，就像俺这儿的厦 X 彩电一样，绝对能看，只是动不动就罢次工。不说远了，总之这里的 quirk 就是用来判断这些有毛病的产品啥毛病的。谁去判断？不像咱们的中国足协，把中国足球折腾成这样子，也就是出来声明一下完事儿，咱们 usb 这儿实行的可是责任制，你的设备接哪儿哪儿负责，也就是说 hub 去判断，就不用咱费心了。

397 行，看到#ifdef CONFIG_PM 这个标志，我们就知道从这里直到最后的那个#endif 都是关于电源管理的。让我们先大胆的忽略它们，struct usb_device 这个结构已经够让我们疲惫了，还是换换口味吧。

配置

越来越觉得 USB 的世界是一个理想化的世界，在那里，一个设备可以有多种配置，做不同的事，过不一样的生活，而我们的配置永远只有一个，从生到死，没得选择。所谓的爱情的选择，职业的选择，在你选择过后再看，似乎冥冥中都有一种神秘的力量在支配着，而我们在其中只不过是上紧的发条，每个人自出生始就被配置了不同的路，你的彷徨，你的叛逆，你的伤心，你的快乐，都只不过是许多看似的偶然所组成的必然。

我不是宿命论者，我也经常在夜深人静不能入寐的时候回忆过去的悲欢与离合，惆怅现在的无助与无奈，思索将来要走的路。不过在爱成往事，烟花散尽后，过往的情形似乎都是不可逆转的必然。这就是所谓的人类一思考上帝就发笑吧，因为他知道你的努力都是徒劳，但是你还不得不去努力，这就是所谓的无奈的人生。

上帝又发笑了，还是接着看 usb 设备的配置吧，在 include/linux/usb.h 里定义

```
206 /**
207  * struct usb_host_config - representation of a device's configuration
208  * @desc: the device's configuration descriptor.
209  * @string: pointer to the cached version of the iConfiguration string, if
210  *      present for this configuration.
211  * @interface: array of pointers to usb_interface structures, one for each
212  *      interface in the configuration.  The number of interfaces is stored
213  *      in desc.bNumInterfaces.  These pointers are valid only while the
214  *      the configuration is active.
215  * @intf_cache: array of pointers to usb_interface_cache structures, one
```

```
 *      for each interface in the configuration.  These structures exist
 *      for the entire life of the device.
 * @extra: pointer to buffer containing all extra descriptors associated
 *      with this configuration (those preceding the first interface
 *      descriptor).
 * @extralen: length of the extra descriptors buffer.
 *
 * USB devices may have multiple configurations, but only one can be active
 * at any time.  Each encapsulates a different operational environment;
 * for example, a dual-speed device would have separate configurations for
 * full-speed and high-speed operation.  The number of configurations
 * available is stored in the device descriptor as bNumConfigurations.
 *
 * A configuration can contain multiple interfaces.  Each corresponds to
 * a different function of the USB device, and all are available whenever
 * the configuration is active.  The USB standard says that interfaces
 * are supposed to be numbered from 0 to desc.bNumInterfaces-1, but a lot
 * of devices get this wrong.  In addition, the interface array is not
 * guaranteed to be sorted in numerical order.  Use usb_ifnum_to_if() to
 * look up an interface entry based on its number.
 *
 * Device drivers should not attempt to activate configurations.  The choice
 * of which configuration to install is a policy decision based on such
 * considerations as available power, functionality provided, and the user's
 * desires (expressed through userspace tools).  However, drivers can call
 * usb_reset_configuration() to reinitialize the current configuration and
 * all its interfaces.
 */
struct usb_host_config {
        struct usb_config_descriptor    desc;

        char *string;           /* iConfiguration string, if present */
        /* the interfaces associated with this configuration,
         * stored in no particular order */
        struct usb_interface *interface[USB_MAXINTERFACES];

        /* Interface information available even when this is not the
         * active configuration */
        struct usb_interface_cache *intf_cache[USB_MAXINTERFACES];

        unsigned char *extra;   /* Extra descriptors */
        int extralen;
};
```

245 行，desc，四大描述符里最后的一个终于出现了，同样是在它们的老巢 include/linux/usb/ch9.h 里定义

```
250 /* USB_DT_CONFIG: Configuration descriptor information.
251  *
252  * USB_DT_OTHER_SPEED_CONFIG is the same descriptor, except that the
253  * descriptor type is different.  Highspeed-capable devices can look
254  * different depending on what speed they're currently running.  Only
255  * devices with a USB_DT_DEVICE_QUALIFIER have any OTHER_SPEED_CONFIG
256  * descriptors.
257  */
258 struct usb_config_descriptor {
259         __u8  bLength;
260         __u8  bDescriptorType;
261
262         __le16 wTotalLength;
263         __u8  bNumInterfaces;
264         __u8  bConfigurationValue;
265         __u8  iConfiguration;
266         __u8  bmAttributes;
267         __u8  bMaxPower;
268 } __attribute__ ((packed));
269
270 #define USB_DT_CONFIG_SIZE              9
```

259 行，bLength，描述符的长度，值为 USB_DT_CONFIG_SIZE。

260 行，bDescriptorType，描述符的类型，值为 USB_DT_CONFIG，0x02。这么说对不对？按照前面接口描述符、端点描述符和设备描述符的习惯来说，应该是没问题。但是，生活总是会在我们已经习惯它的时候来个转折，这里的值却并不仅仅可以为 USB_DT_CONFIG，还可以为USB_DT_OTHER_SPEED_CONFIG，0x07。这里说的 OTHER_SPEED_CONFIG描述符描述的是高速设备操作在低速或全速模式时的配置信息，和配置描述符的结构完全相同，区别只是描述符的类型不同，是只有名字不同的孪生兄弟。

262 行，wTotalLength，使用 GET_DESCRIPTOR 请求从设备里获得配置描述符信息时，返回的数据长度，也就是说对包括配置描述符、接口描述符、端点描述符，class-或 vendor-specific 描述符在内的所有描述符算了个总帐。

263 行，bNumInterfaces，这个配置包含的接口数目。

263 行，bConfigurationValue，对于拥有多个配置的幸运设备来说，可以拿这个值为参数，使用 SET_CONFIGURATION 请求来改变正在被使用的 USB 配置，bConfigurationValue 就指明了将要激活哪个配置。咱们的设备虽然可以有多个配置，但

同一时间却也只能有一个配置被激活。捎带着提一下，SET_CONFIGURATION 请求也是标准的设备请求之一，专门用来设置设备的配置。

265 行，iConfiguration，描述配置信息的字符串描述符的索引值。

266 行，bmAttributes，这个字段表征了配置的一些特点，比如 bit 6 为 1 表示 self-powered，bit 5 为 1 表示这个配置支持远程唤醒。另外，它的 bit 7 必须为 1，为什么？协议里就这么说的，我也不知道，这个世界上并不是什么事情都找得到原因的。ch9.h 里有几个相关的定义

```
272 /* from config descriptor bmAttributes */
273 #define USB_CONFIG_ATT_ONE              (1 << 7)        /* must be set */
274 #define USB_CONFIG_ATT_SELFPOWER        (1 << 6)        /* self powered */
275 #define USB_CONFIG_ATT_WAKEUP           (1 << 5)        /* can wakeup */
276 #define USB_CONFIG_ATT_BATTERY          (1 << 4)        /* battery powered */
```

267 行，bMaxPower，设备正常运转时，从总线那里分得的最大电流值，以 2mA 为单位。设备可以使用这个字段向 hub 表明自己需要的的电流，但如果设备需求过于旺盛，请求的超出了 hub 所能给予的，hub 就会直接拒绝，不会心软。你去请求她给你多一点点爱，可她心系天下人，没有多的分到你身上，于是怎么办？拒绝你呗，不要说爱情是多么残酷，这个世界就是很无奈。还记得 struct usb_device 结构里的 bus_mA 吗？它就表示 hub 所能够给予的。Alan Stern 大侠告诉我们

(c->desc.bMaxPower * 2) is what the device requests and udev->bus_mA is what the hub makes available.

到此为止，四大标准描述符已经全部登场亮相了，还是回到struct usb_host_config结构的 247 行，string，这个字符串保存了配置描述符iConfiguration字段对应的字符串描述符信息。

250 行，interface[USB_MAXINTERFACES]，配置所包含的接口。注释里说的很明确，这个数组的顺序未必是按照配置里接口号的顺序,所以你要想得到某个接口号对应的struct usb_interface 结构对象，就必须使用 usb.c 里定义的 usb_ifnum_to_if 函数。

```
65 /**
66  * usb_ifnum_to_if - get the interface object with a given interface number
67  * @dev: the device whose current configuration is considered
68  * @ifnum: the desired interface
69  *
70  * This walks the device descriptor for the currently active configuration
71  * and returns a pointer to the interface with that particular interface
72  * number, or null.
73  *
```

```
74  * Note that configuration descriptors are not required to assign interface
75  * numbers sequentially, so that it would be incorrect to assume that
76  * the first interface in that descriptor corresponds to interface zero.
77  * This routine helps device drivers avoid such mistakes.
78  * However, you should make sure that you do the right thing with any
79  * alternate settings available for this interfaces.
80  *
81  * Don't call this function unless you are bound to one of the interfaces
82  * on this device or you have locked the device!
83  */
84 struct usb_interface *usb_ifnum_to_if(const struct usb_device *dev,
85                                       unsigned ifnum)
86 {
87         struct usb_host_config *config = dev->actconfig;
88         int i;
89
90         if (!config)
91                 return NULL;
92         for (i = 0; i < config->desc.bNumInterfaces; i++)
93                 if (config->interface[i]->altsetting[0]
94                                 .desc.bInterfaceNumber == ifnum)
95                         return config->interface[i];
96
97         return NULL;
98 }
```

这个函数的道理很简单，就是拿你指定的接口号，和当前配置的每一个接口可选设置 0 里的接口描述符的 bInterfaceNumber 字段做比较，相等了，那个接口就是你要寻找的，都不相等，那对不起，不能满足你的要求，虽然它已经尽力了。

如果你看了协议，可能会在 9.6.5 里看到，请求配置描述符时，配置里的所有接口描述符是按照顺序一个一个返回的。那为什么这里又明确说明，让咱们不要期待它就会是接口号的顺序那？其实很久很久以前这里并不是这么说地，它就说这个数组是按照0..desc.bNumInterfaces的顺序，但同时又说需要通过 usb_ifnum_to_if函数来获得指定接口号的接口对象，Alan Stern大侠质疑了这种有些矛盾的说法，于是David Brownell大侠就把它改成现在这个样子了，为什么改？因为协议归协议，厂商归厂商，有些厂商就是有不遵守协议的癖好，它非要先返回接口 1 再返回接口 0，你也没辙，所以就不得不增加usb_ifnum_to_if函数。

USB_MAXINTERFACES 是 include/linux/usb.h 里定义的一个宏，值为 32，不要说不够用，谁见过有很多接口的设备？

```
176 /* this maximum is arbitrary */
177 #define USB_MAXINTERFACES       32
```

254 行，intf_cache[USB_MAXINTERFACES]，cache 是什么？缓存。答对了，不然大学四年岂不是光去吃老赵烤肉了。这是个 struct usb_interface_cache 对象的结构数组，usb_interface，usb 接口，cache，缓存，所以 usb_interface_cache 就是 usb 接口的缓存。缓存些什么？看看 include/linux/usb/usb.h 里的定义

```
179 /**
180  * struct usb_interface_cache - long-term representation of a device interface
181  * @num_altsetting: number of altsettings defined.
182  * @ref: reference counter.
183  * @altsetting: variable-length array of interface structures, one for
184  *      each alternate setting that may be selected.  Each one includes a
185  *      set of endpoint configurations.  They will be in no particular order.
186  *
187  * These structures persist for the lifetime of a usb_device, unlike
188  * struct usb_interface (which persists only as long as its configuration
189  * is installed).  The altsetting arrays can be accessed through these
190  * structures at any time, permitting comparison of configurations and
191  * providing support for the /proc/bus/usb/devices pseudo-file.
192  */
193 struct usb_interface_cache {
194         unsigned num_altsetting;        /* number of alternate settings */
195         struct kref ref;                /* reference counter */
196
197         /* variable-length array of alternate settings for this interface,
198          * stored in no particular order */
199         struct usb_host_interface altsetting[0];
200 };
```

199 行的 altsetting[0]是一个可变长数组，按需分配的那种，你对设备说 GET_DESCRIPTOR的时候，内核就根据返回的每个接口可选设置的数目分配给 intf_cache数组相应的空间，有多少需要多少分配多少，在咱们还在为拥有一套房而奋斗终生的时候，这里已经提前步入了共产主义。

为什么要缓存这些东东？房价在变，物价在变，设备的配置也在变，此时这个配置可能还在欢快的被宠幸着，彼时它可能就躲在冷宫里写《后宫回忆录》，漫长的等待之后，哪个导演慧眼识剧本发现了它，她就又迎来了自己的第二春。这就叫此一时彼一时。为了在配置被取代之后仍然能够获取它的一些信息，就把日后可能会需要的一些东东放在了intf_cache数组的struct usb_interface_cache对象里。谁会需要？这么说吧，你通过sysfs这个窗口只能看到设备当前配置的一些信息，即使是这个配置下面的接口，也只能看到接口正在使用的那个可选设置的信息，可是你希望能够看到更多的，怎么办，窗户太小了，可以趴门口看，usbfs就是这个门，里面显示有你的系统中所有usb设备的可选配置和端点信息，它就是利用intf_cache这个数组里缓存的东东实现的。

256 行， extra， 257 行， extralen，有关额外扩展的描述符的，和 struct usb_host_interface 里的差不多，只是这里的是针对配置的，如果你使用 GET_DESCRIPTOR请求从设备里获得配置描述符信息，它们会紧跟在标准的配置描述符后面返回给你。

向左走，向右走

他们彼此深信，是瞬间迸发的热情，让他们相遇；

这样的确定是美丽的，但变幻无常更为美丽；

他们素未谋面，所以他们确定，彼此并无任何瓜葛，

但是自街道、楼梯、大堂传来的话语，

他们也许擦肩而过一百万次了吧？

我想问他们是否记得，

在旋转门面对面那一刹，

或是在人群中喃喃道出的“对不起”，

或是在电话的另一端道出的“打错了”，

但是，我早知道答案——

是的，他们并不记得。

他们会很讶异，原来缘份已经戏弄他们多年，

时机尚未成熟，变成他们的命运。

缘份，将他们推进、驱离，阻挡他们的去路，

忍住笑声，

然后闪到一旁。

……

我大学四年的历史是南门外老赵烤肉发展的历史，大一时，只是一间破败的小门面，大二时，地方大了些，大三时，有了金字招牌，装的也像了点样子，大四时，老板娘戴上了傻粗傻粗的金项链，我也在和同学翻了多少次的南门，吃了多少只烤鱼后，灌了多少瓶啤酒后看到了向左走向右走，于是知道了几米，于是少了些烤鱼少了些啤酒，于是看了更多的几米。随后

不知道过了多久，看到徐家汇太平洋那里有了几米的小工艺品，于是买了个小记事本做纪念，不是没有其它的卖，而是都太昂贵，几米的漫画人人可以看，几米的工艺品不是人人都可以狠得下心去买。随后不知道又过了多久，有了杜琪峰和韦家辉的向左走向右走，对影片本身并没有多少特别的感觉，但还是知道了梁咏琪翻译的上面的那首诗。

伴着这首诗，让我们回到很久很久以前提到的函数 usb_device_match，之前做的所有铺垫，只是为了与它再次相见。过去很久了么？“我们以前都失散过，十三年以后，还不是再遇见？我知道十三年时间很长，但如果十三年后能再见到也很好啊。如果十三年之后见不到，那再过十三年都会见得到也好啊，最重要是见得到。”

```
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 }
```

USB的世界里，设备和驱动不会有向左走向右走那般的遗憾，缘分对于我们来说是不可言传的玄机，对于它们来说只是 usb_device_match函数的两端。usb_device_match函数为它们指明向左走还是向右走，为它们指明哪个才是它们命中注定的缘，而我们的生活里却不会有这样的一个角色，所以只能有无数的“如果你尝试换个方向，你可能发现，一切原来都不一样。”这样的假设。

543 行，第一次遇到这个函数的时候，我说了这里有两条路，一条给 USB 设备走，一条给 USB 接口走。前面站在这个路口上，将两条路上可能遇到的主要角色大都介绍了下，它们类似于影片刚开幕的演员列表里的领衔主演，总是会谋杀比较多的菲林。先来看看设备走的这条路，上面儿只有两个函数，好像要平坦的多，可以一下走到底儿的样子。

```
85 static inline int is_usb_device(const struct device *dev)
86 {
87         return dev->type == &usb_device_type;
88 }
```

usb.h 里定义的这个函数看长相就是个把门儿的角色，好像那些所谓高尚小区的门卫，看到你和你同事分别开一辆宝来和奥拓要进去，气势汹汹的过来告你，宝来才能进，奥拓一边儿去。这个函数就是要告诉你，是 usb_device 才能打这儿过，否则一边儿该干吗干吗去。但关键问题不是它让不让你进，而是它怎么知道你是不是 usb_device，那些门卫起码也得认识啥是宝来啥是奥拓啊，否则直接拿奥拓充宝来不就可以开进去了，现在可是 21 世纪，假的比真的还流行的时代，塞几块硅胶就成性感女神流行代名词儿了。所以咱们要了解这里是怎么做到真的假不了的，关键就在于这个 dev->type，设备的类型，看它是不是等于咱们定义的 usb_device_type，也就是说 usb 设备类型，相等的话，那好说，他躬送你进去，不相等，那就此路不是为你开的，你找别的路吧。usb_device_type 在 usb.c 里定义

```
195 struct device_type usb_device_type = {
196         .name =         "usb_device",
197         .release =      usb_release_dev,
198 };
```

它和咱们前面看到的那个 usb_bus_type差不多，一个表示总线的类型，一个表示设备的类型，总线有总线的类型，设备有设备的类型，就好像在那个红色中国的时期里，人有很多种成分儿，地主老财就不能混到贫下中农的革命队伍里，当时俺爷爷有几亩薄田，身子骨儿薄，找了几个人帮着种下，结果就成了剥削劳动人民血汗的地主老财，俺姥爷挑着个剃头担子走街串巷，帮人剃头谋几个小钱，结果被打到划成小手工业劳动者，他们顶着成分不好的帽子含辛茹苦好多年，做梦都想与贫下中农靠的近点儿再近点儿，可现在就不同了，谁还愿意去做贫下中农。

假设现在过来一个设备，经过判断，它要走的是设备这条路，可问题是，这个设备的 type 字段啥时候被初始化成 usb_device_type 了。嗯，这到是个问题，不过先按下它不表，继

续向前走，带着疑问上路。

546 行，又见到一个 if，一个把门儿的，这年头儿，就是关关卡卡的多，走个人民的公路还有许多人民的收费站来收人民的钱。先看看它是干吗的，收的合理不合理。它就跟在上面的 is_usb_device 函数后面在 usb.h 文件里定义

```
90 /* Do the same for device drivers and interface drivers. */
91
92 static inline int is_usb_device_driver(struct device_driver *drv)
93 {
94         return container_of(drv, struct usbdrv_wrap, driver)->
95                         for_devices;
96 }
```

这个函数的脸上就写着我是用来判断是不是 usb device driver 的，那咱们就要问问什么是 usb device driver？前面不是一直都是说一个 usb 接口对应一个 usb 驱动么？作为一个吃社会主义粮长大的人，我可以负责任的告诉你前面说的一点错都没有，一个接口就是要对应一个 usb 驱动，可是我们不能只钻到接口的那个口里边儿，我们应该眼光放的更加开阔些，要知道接口在 usb 的世界里并不是老大，它上边儿还有配置，还有设备，都比它大。每个接口对应了一个独立的功能，是需要专门的驱动来和它交流，但是接口毕竟整体是作为一个 usb 设备而存在的，设备还可以有不同的配置，我们还可以为设备指定特定的配置，那谁来做这个事情？接口驱动么？它还不够级别，它的级别只够和接口会谈会谈。这个和整个 usb 设备进行对等交流的光荣任务就交给了 struct usb_device _driver，即 usb 设备驱动，它和 usb 的接口驱动 struct usb_driver 都定义在 include/linux/usb.h 文件里

```
784 /**
785  * struct usb_driver - identifies USB interface driver to usbcore
786  * @name: The driver name should be unique among USB drivers,
787  *      and should normally be the same as the module name.
788  * @probe: Called to see if the driver is willing to manage a particular
789  *      interface on a device.  If it is, probe returns zero and uses
790  *      dev_set_drvdata() to associate driver-specific data with the
791  *      interface.  It may also use usb_set_interface() to specify the
792  *      appropriate altsetting.  If unwilling to manage the interface,
793  *      return a negative errno value.
794  * @disconnect: Called when the interface is no longer accessible, usually
795  *      because its device has been (or is being) disconnected or the
796  *      driver module is being unloaded.
797  * @ioctl: Used for drivers that want to talk to userspace through
798  *      the "usbfs" filesystem.  This lets devices provide ways to
799  *      expose information to user space regardless of where they
800  *      do (or don't) show up otherwise in the filesystem.
801  * @suspend: Called when the device is going to be suspended by the system.
802  * @resume: Called when the device is being resumed by the system.
```

```
803  * @pre_reset: Called by usb_reset_composite_device() when the device
804  *      is about to be reset.
805  * @post_reset: Called by usb_reset_composite_device() after the device
806  *      has been reset.
807  * @id_table: USB drivers use ID table to support hotplugging.
808  *      Export this with MODULE_DEVICE_TABLE(usb,...).  This must be set
809  *      or your driver's probe function will never get called.
810  * @dynids: used internally to hold the list of dynamically added device
811  *      ids for this driver.
812  * @drvwrap: Driver-model core structure wrapper.
813  * @no_dynamic_id: if set to 1, the USB core will not allow dynamic ids to be
814  *      added to this driver by preventing the sysfs file from being created.
815  * @supports_autosuspend: if set to 0, the USB core will not allow autosuspend
816  *      for interfaces bound to this driver.
817  *
818  * USB interface drivers must provide a name, probe() and disconnect()
819  * methods, and an id_table.  Other driver fields are optional.
820  *
821  * The id_table is used in hotplugging.  It holds a set of descriptors,
822  * and specialized data may be associated with each entry.  That table
823  * is used by both user and kernel mode hotplugging support.
824  *
825  * The probe() and disconnect() methods are called in a context where
826  * they can sleep, but they should avoid abusing the privilege.  Most
827  * work to connect to a device should be done when the device is opened,
828  * and undone at the last close.  The disconnect code needs to address
829  * concurrency issues with respect to open() and close() methods, as
830  * well as forcing all pending I/O requests to complete (by unlinking
831  * them as necessary, and blocking until the unlinks complete).
832  */
833 struct usb_driver {
834         const char *name;
835
836         int (*probe) (struct usb_interface *intf,
837                       const struct usb_device_id *id);
838
839         void (*disconnect) (struct usb_interface *intf);
840
841         int (*ioctl) (struct usb_interface *intf, unsigned int code,
842                         void *buf);
843
844         int (*suspend) (struct usb_interface *intf, pm_message_t message);
845         int (*resume) (struct usb_interface *intf);
846
```

```
847         void (*pre_reset) (struct usb_interface *intf);
848         void (*post_reset) (struct usb_interface *intf);
849
850         const struct usb_device_id *id_table;
851
852         struct usb_dynids dynids;
853         struct usbdrv_wrap drvwrap;
854         unsigned int no_dynamic_id:1;
855         unsigned int supports_autosuspend:1;
856 };
```

蒲松龄曰，每个男人的心中都有一个狐狸精，每个写usb驱动的人心中都有一个usb_driver。一般来说，我们平时所谓的编写usb驱动指的也就是写usb接口的驱动，需要以一个struct usb_driver结构的对象为中心，以设备的接口提供的功能为基础，开展usb驱动的建设。

834 行，name，驱动程序的名字，对应了在/sys/bus/usb/drivers/下面的子目录名称。和我们每个人一样，它只是彼此区别的一个代号，不同的是我们可以有很多人叫张三或者李四，但这里的名字在所有的 usb 驱动中必须是唯一的。

836 行，probe，用来看看这个 usb 驱动是否愿意接受某个接口的函数。每个驱动自诞生起，它的另一半就已经确定了，这个函数就是来判断哪个才是她苦苦等待的那个他。当然，这个他应该是他们，因为一个驱动往往可以支持多个接口。

839 行，disconnect，当接口失去联系，或使用rmmod卸载驱动将它和接口强行分开时这个函数就会被调用。

841 行，ioctl，当你的驱动有通过usbfs和用户空间交流的需要的话，就使用它吧。

844 行，suspend，845 行，resume，分别在设备被挂起和唤醒时使用。

847 行，pre_reset，848 行，post_reset，分别在设备将要复位（reset）和已经复位后使用。

850 行，id_table，驱动支持的所有设备的花名册，所有的三宫六院要想受到宠幸都要在这里登记。驱动就靠这张表儿来识别是不是支持哪个设备接口的，如果不属于这张表，那就躲一边儿去吧。

852 行，dynids，支持动态 id 的。什么是动态 id？本来前面刚说每个驱动诞生时她的另一半在 id_table 里就已经确定了，可是谁规定了女同胞就一定要从一而终了，那是封建旧思想要打到的，听听她们内心的呼声“谁说我白，瘦，漂亮~我就跟他做好朋友”，Greg 大侠显然也听到了，于是在一年多前的一个寒风萧萧的日子里平地一声吼，加入了动态 id 的机

制。即使驱动已经加载了，也可以添加新的 id 给她，只要新 id 代表的设备存在，对她说“你又白又瘦又漂亮”，她就会和他绑定起来。

怎么添加新的 id？到驱动所在的地方瞅瞅，也就是/sys/bus/usb/drivers 目录下边儿，那里列出的每个目录就代表了一个 usb 驱动，随便选一个进去，能够看到一个 new_id 文件吧，使用 echo 将厂商和产品 id 写进去就可以了。看看 Greg 举的一个例子

```
echo 0557 2008 > /sys/bus/usb/drivers/foo_driver/new_id
```

就可以将 16 进制值 0557 和 2008 写到 foo_driver 驱动的设备 id 表里取。

853 行，drvwrap，这个字段有点意思，struct usbdrv_wrap 结构的，也在 include/linux/usb.h 里定义

```
774 /**
775  * struct usbdrv_wrap - wrapper for driver-model structure
776  * @driver: The driver-model core driver structure.
777  * @for_devices: Non-zero for device drivers, 0 for interface drivers.
778  */
779 struct usbdrv_wrap {
780         struct device_driver driver;
781         int for_devices;
782 };
```

近距离观察一下这个结构，它里面内容是比较的贫乏的，只有一个struct device_driver结构的对象和一个for_devices的整型字段。回想一下linux的设备模型，我们的心头就会产生这样的疑问，这里的struct device_driver对象不是应该嵌入到struct usb_driver结构里么，怎么这里又包装了一层？再包装这么一层当然不是为了美观，写代码的哥们儿没这么媚俗，这主要还是因为本来挺单纯的驱动在usb的世界里不得已分成了设备驱动和接口驱动两种，为了区分这两种驱动，就中间加了这么一层，添了个for_devices标志来判断是哪种。大家发现没，之前见识过的结构里，很多不是 1 就是 0 的标志使用的是位字段，特别是几个这样的标志放一块儿的时候，而这里的for_devices虽然也只能有两个值，但却没有使用位字段，为什么？简单的说就是这里没那个必要，那些使用位字段的是几个在一块儿，可以节省点儿存储空间，而这里只有这么一个，就是使用位字段也节省不了，就不用多此一举了，这个大家都知道哈，我有点多说了，还是言归整传。其实就这么说为了加个判断标志就硬生生的塞这么一层，还是会有点模糊的，不过，其它字段不敢说，这个drvwrap咱们以后肯定还会遇到它，这里先有个概念，混个面熟，等到再次相遇的那一刻，我保证你会明白它的用心。

854 行，no_dynamic_id，可以用来禁止动态 id 的，设置了之后，驱动就从一而终吧，别七想八想了。

855 行，supports_autosuspend，对 autosuspend 的支持，如果设置为 0 的话，就不再允许绑定到这个驱动的接口 autosuspend。

struct usb_driver 结构就暂时了解到这里，咱们再来看看所谓的 usb 设备驱动与接口驱动到底都有多大的不同。

```
859 /**
860  * struct usb_device_driver - identifies USB device driver to usbcore
861  * @name: The driver name should be unique among USB drivers,
862  *      and should normally be the same as the module name.
863  * @probe: Called to see if the driver is willing to manage a particular
864  *      device.  If it is, probe returns zero and uses dev_set_drvdata()
865  *      to associate driver-specific data with the device.  If unwilling
866  *      to manage the device, return a negative errno value.
867  * @disconnect: Called when the device is no longer accessible, usually
868  *      because it has been (or is being) disconnected or the driver's
869  *      module is being unloaded.
870  * @suspend: Called when the device is going to be suspended by the system.
871  * @resume: Called when the device is being resumed by the system.
872  * @drvwrap: Driver-model core structure wrapper.
873  * @supports_autosuspend: if set to 0, the USB core will not allow autosuspend
874  *      for devices bound to this driver.
875  *
876  * USB drivers must provide all the fields listed above except drvwrap.
877  */
878 struct usb_device_driver {
879         const char *name;
880
881         int (*probe) (struct usb_device *udev);
882         void (*disconnect) (struct usb_device *udev);
883
884         int (*suspend) (struct usb_device *udev, pm_message_t message);
885         int (*resume) (struct usb_device *udev);
886         struct usbdrv_wrap drvwrap;
887         unsigned int supports_autosuspend:1;
888 };
```

就当一个ppmm在走近你，你先用放光的眼睛上上下下的扫一遍，这个usb设备驱动比前面的接口驱动要苗条多了，除了少了很多东西外，剩下的将参数里的struct usb_interface换成struct usb_device后就几乎一摸一样了。友情提醒一下，这里说的是几乎，而不是完全，这是因为probe，它的参数里与接口驱动里的probe相比少了那个设备的花名册，也就是说它不用再去根据花名册来判断是不是愿意接受一个usb设备。那么这意味着什么？是它来者不拒，接受所有的usb设备？还是独锁深闺垂影自恋决绝所有人？当然只会是前者，不然这个usb设备驱动就完全毫无疑义了，这里不像咱们的政府大院儿，容不得毫无作用的东

东存在。而且我们在内核里找来找去，也就只能找得着它在generic.c文件里定义了usb_generic_driver这么一个对象

```
210 struct usb_device_driver usb_generic_driver = {
211         .name = "usb",
212         .probe = generic_probe,
213         .disconnect = generic_disconnect,
214 #ifdef  CONFIG_PM
215         .suspend = generic_suspend,
216         .resume = generic_resume,
217 #endif
218         .supports_autosuspend = 1,
219 };
```

即使这么一个独苗儿，也早在 usb_init 的 895 行就已经注册给 usb 子系统了。那么我们该用什么样的言语表达自己的感受？所谓的 usb 设备驱动完全就是一个博爱的主儿，我们的 core 用它来与所有的 usb 设备进行交流，它们都交流些什么？现在人人都有一颗八卦的心，不过这是后话，我会告诉你的。

不管怎么说，总算是把 usb 接口的驱动和设备的驱动给过了一下，还是回到这节开头儿的 usb_device_match 函数，目前为止，设备这条路已经比较清晰了。就是如果设备过来了，走到了设备这条路，然后要判断下驱动是不是设备驱动，是不是针对整个设备的，如果不是的话，对不起，虽然这条路走对了，可是沿这条路，设备找不到对应的驱动，匹配不成功，就直接返回了，那如果驱动也确实是设备驱动那？代码里是直接返回 1，表示匹配成功了，没有再花费哪怕多一点儿的精力。

本来这么说应该是可以了，可是我还是忍不住想告诉你，在某年某月某日之前的内核版本里，是没有很明确的struct usb_device_driver这样一个表示usb设备驱动的结构的，而是直接定义了struct device_driver结构类型的一个对象 usb_generic_driver来处理与整个设备相关的事情，相对应的，usb_device_match这个匹配函数也只有简简单单的一条路，在接口和对应的驱动之间做匹配，不用去理会整个usb设备的是是非非。但是在 2006 年的那个夏天，在我们为卡卡，为舍普琴科，为皮尔洛们疯狂的时候，黄建翔不是一个人在战斗，Alan Stern大侠也不是一个人在战斗，于是内核里多了struct usb_device_driver结构，usb_device_match这里也多了一条给设备走的路。

为什么？杨钰莹唱，我不想说我很困惑。是时候也有必要对 usb 设备在 usb 世界里的整个人生旅程做个介绍了。与 usb_device_match 短暂相遇便要再次分开，不过，最重要的是见得到。

设备的生命线（一）

李安告诉我们，每个人的心中都有一座断背山，每个人的手里都有一条生命线。

Google 一下，找到这么一句：通常生命线都起自姆指和食指的中央，如果比这个位置还要上方的人，主进取心和克已心都很强，只要奋斗努力，事业终有可成。再仔细看看自己的手，果然位于姆指与食指的中间靠上 08 微米，也算是搭了个 08 年奥运的好兆头，继海南那个万里长跑迎奥运的小女孩儿之后，俺也终于有了奥运题材，有了乐观的理由。什么叫乐观派的人？这个就像茶壶一样，屁股都烧的红红的，他还有心情吹口哨。俺可不想做茶壶，还是赶紧的言归正传吧。

BH 的人生有 BH 的活法，设备的人生有设备的过法。设备也有它自己的生命线，自你把它插到 hub 上始，自你把它从 hub 上拔下来终，它的一生是勤勉努力、朴实无华的一生，它的一生是埋头苦干、默默奉献的一生。BH 的人生不需要解释，设备的人生值得我们去分析。

我相信科学不相信迷信，港剧里俺最喜欢的罗嘉良在迷离档案里说，要用科学来研究迷信。咱们现在就沿着设备的生命线走一走，看看其中都发生了什么。

首先当然是你将 usb 设备连接在 hub 的某个端口上，hub 检测到有设备连接了进来，它也知道有朋自远方来不亦乐乎，于是精神头儿就上来了，就觉得有必要为设备做点什么。它会为设备分配一个 struct usb_device 结构的对象并初始化，并调用设备模型提供的接口将设备添加到 usb 总线的设备列表里，然后 usb 总线会遍历驱动列表里的每个驱动，调用自己的 match 函数看它们和你的设备或接口是否匹配。这不，又走到 match 函数了，那接下来那，先看看前面的，等真正遇到它的时候再说。

这么说是不是很不过瘾？本来满怀期待的耗费一个匹萨的钱去上海影城看大片儿，结果只看到了一个小馒头和一个圆环套圆环娱乐城。为了看清楚那一个馒头背后的故事，接下来只要你不嫌烦，咱就往细里去说，说到你烦为止。

hub 检测到自己的某个端口有设备连接了进来后，它会调用 core 里的 usb_alloc_dev 函数为 struct usb_device 结构的对象申请内存，这个函数在 usb.c 文件里定义

```
226 /**
227  * usb_alloc_dev - usb device constructor (usbcore-internal)
228  * @parent: hub to which device is connected; null to allocate a root hub
229  * @bus: bus used to access the device
230  * @port1: one-based index of port; ignored for root hubs
231  * Context: !in_interrupt()
232  *
233  * Only hub drivers (including virtual root hub drivers for host
234  * controllers) should ever call this.
235  *
```

```
236  * This call may not be used in a non-sleeping context.
237  */
238 struct usb_device *
239 usb_alloc_dev(struct usb_device *parent, struct usb_bus *bus, unsigned port1)
240 {
241         struct usb_device *dev;
242
243         dev = kzalloc(sizeof(*dev), GFP_KERNEL);
244         if (!dev)
245                 return NULL;
246
247         if (!usb_get_hcd(bus_to_hcd(bus))) {
248                 kfree(dev);
249                 return NULL;
250         }
251
252         device_initialize(&dev->dev);
253         dev->dev.bus = &usb_bus_type;
254         dev->dev.type = &usb_device_type;
255         dev->dev.dma_mask = bus->controller->dma_mask;
256         dev->state = USB_STATE_ATTACHED;
257
258         INIT_LIST_HEAD(&dev->ep0.urb_list);
259         dev->ep0.desc.bLength = USB_DT_ENDPOINT_SIZE;
260         dev->ep0.desc.bDescriptorType = USB_DT_ENDPOINT;
261         /* ep0 maxpacket comes later, from device descriptor */
262         dev->ep_in[0] = dev->ep_out[0] = &dev->ep0;
263
264         /* Save readable and stable topology id, distinguishing devices
265          * by location for diagnostics, tools, driver model, etc.  The
266          * string is a path along hub ports, from the root.  Each device's
267          * dev->devpath will be stable until USB is re-cabled, and hubs
268          * are often labeled with these port numbers.  The bus_id isn't
269          * as stable:  bus->busnum changes easily from modprobe order,
270          * cardbus or pci hotplugging, and so on.
271          */
272         if (unlikely(!parent)) {
273                 dev->devpath[0] = '';
274
275                 dev->dev.parent = bus->controller;
276                 sprintf(&dev->dev.bus_id[0], "usb%d", bus->busnum);
277         } else {
278                 /* match any labeling on the hubs; it's one-based */
279                 if (parent->devpath[0] == '')
```

```
280                         snprintf(dev->devpath, sizeof dev->devpath,
281                                 "%d", port1);
282                 else
283                         snprintf(dev->devpath, sizeof dev->devpath,
284                                 "%s.%d", parent->devpath, port1);
285
286                 dev->dev.parent = &parent->dev;
287                 sprintf(&dev->dev.bus_id[0], "%d-%s",
288                         bus->busnum, dev->devpath);
289
290                 /* hub driver sets up TT records */
291         }
292
293         dev->portnum = port1;
294         dev->bus = bus;
295         dev->parent = parent;
296         INIT_LIST_HEAD(&dev->filelist);
297
298 #ifdef  CONFIG_PM
299         mutex_init(&dev->pm_mutex);
300         INIT_DELAYED_WORK(&dev->autosuspend, usb_autosuspend_work);
301         dev->autosuspend_delay = usb_autosuspend_delay * HZ;
302 #endif
303         return dev;
304 }
```

usb_alloc_dev 函数就相当于 usb 设备的构造函数，参数里边儿，parent 是设备连接的那个 hub，bus 是设备连接的那条总线，port1 就是设备连接在 hub 上的那个端口。

243 行，为一个 struct usb_device 结构的对象申请内存并初始化为 0。直到在看到这一行的前一天，我还仍在使用 kmalloc 加 memset 这对儿最佳拍档来申请内存和初始化，但是在看到 kzalloc 之后，我知道了江山代代有人出，眉先生，须后生，先生不及后生长的道理，没看那些 90 后的都在嘲笑咱们 80 后的不知道 kzalloc 了么。kzalloc 直接取代了 kmalloc/memset，一个函数起到了两个函数的作用。这种角色当然是最讨人喜欢的了，所以说现在那些简历上搞什么艺术照片儿性感写真什么的完全毫无必要，只要写上个能够做牛做马，白天做牛晚上做马，那些老板就乐翻天了，哪还用玩儿那些虚头。

然后是判断内存申请成功了没，不成功就不用往下走了。那么通过这么几行，咱们应该记住两个凡是，凡是你想用 kmalloc/memset 组合申请内存的时候，就使用 kzalloc 代替吧，凡是申请内存的，不要忘了判断是不是申请成功了。我们的心里应该把这两个凡是提高到和四项基本原则一样的地位。

247 行，这里的两个函数是hcd，主机控制器驱动里的，具体咱不讲，只要知道usb的世界

里一个主机控制器对应着一条usb总线，主机控制器驱动用struct usb_hcd结构表示，一条总线用struct usb_bus结构表示，函数bus_to_hcd是为了获得总线对应的主机控制器驱动，也就是struct usb_hcd结构对象，函数usb_get_hcd只是将得到的这个usb_hcd结构对象的引用计数加 1，为什么？因为总线上多了一个设备，设备在主机控制器的数据结构就得在，当然得为它增加引用计数。如果这俩函数没有很好的完成自己的任务，那整个usb_alloc_dev函数也就没有继续执行下去的必要了，将之前为struct usb_device结构对象申请的内存释放掉就稍息去吧。

252 行，device_initialize 是设备模型里的函数，这里就是将 struct usb_device 结构里嵌入的那个 struct device 结构体初始化掉，以后好方便用，初始化些什么？以前资本家宁可把牛奶倒掉也不给穷人喝，现在房产商宁可把房子空着也不给百姓住，而我宁可自己不知道也不告诉你。

253 行，将设备所在的总线类型设置为 usb_bus_type。usb_bus_type 咱们很早很早就遇到它了，usb 子系统的初始化函数 usb_init 里就把它给注册掉了，还记得聊到模型的时候说过的那个著名的三角关系不，这里就是把设备和总线这条边儿给搭上了。如果您忘了，那就回头看看吧。

254 行，将设备的设备类型初始化为usb_device_type，这是咱们上节第二次遇到usb_device_match函数，走设备那条路，使用 is_usb_device判断是不是usb设备时留下的疑问，就是在这儿把设备的类型给初始化成usb_device_type了。

255 行，这个就是与 DMA 传输相关的了，设备能不能进行 dma 传输，得看主机控制器的脸色，主机控制器不支持的话设备自作多情那也没有用。所以这里 dma_mask 被设置为 host controller 的 dma_mask。

256 行，将 usb 设备的状态设置为 Attached，表示设备已经连接到 usb 接口上了，是 hub 检测到设备时的初始状态。咱们前面说了，USB 设备从生到死都要按照那么几个状态，这里随着设备生命线的逐渐深入，咱们会看到设备的状态也在逐渐的变化。

258 行，端点 0 实在是太太太特殊了，这个咱们是一而再再而三的感叹，struct usb_device 里直接就有这么一个成员 ep0，这行就将 ep0 的 urb_list 给初始化掉。因为接下来遇到的那些主要角色的成员前面集中都说过了，咱们就不再说它们是嘛意思了，忘了的话可以到前面看看。

259 行，260 行，分别初始化了端点 0 的描述符长度和描述符类型。

260 行，使 struct usb_device 结构里的 ep_in 和 ep_out 指针数组的第一个成员指向 ep0，ep_in[0]和 ep_out[0]本来表示的就是端点 0。

272 行，这里平白无故的多出了个 unlikely，不知道什么意思？先看看它们在

include/linux/compiler.h 的定义

```
54 /*
55  * Generic compiler-dependent macros required for kernel
56  * build go below this comment. Actual compiler/compiler version
57  * specific implementations come from the above header files
58  */
59
60 #define likely(x)       __builtin_expect(!!(x), 1)
61 #define unlikely(x)     __builtin_expect(!!(x), 0)
```

unlikely 不是一个人在奋斗，还有个 likely。定义里那个怪怪的__builtin_expect 是 GCC 里内建的一个函数，具体是做嘛用的可以看看 GCC 的手册

```
long __builtin_expect (long exp, long c)
You may use __builtin_expect to provide the compiler with branch prediction
information. In general, you should prefer to use actual profile feedback for this
( ‘-fprofile-arcs’ ), as programmers are notoriously bad at predicting how their
programs actually perform. However, there are applications in which this data is
hard to collect.
The return value is the value of exp, which should be an integral expression. The
value of c must be a compile-time constant. The semantics of the built-in are that
it is expected that exp == c. For example:
if (__builtin_expect (x, 0))
        foo ();
would indicate that we do not expect to call foo, since we expect x to be zero. Since
you are limited to integral expressions for exp, you should use constructions such
as
if (__builtin_expect (ptr != NULL, 1))
        error ();
when testing pointer or floating-point values.
```

大致意思就是由于大部分写代码的在分支预测方面做的比较的糟糕，所以 GCC 提供了这个内建的函数来帮助处理分支预测，优化程序，它的第一个参数 exp 为一个整型的表达式，返回值也是这个 exp，它的第二个参数 c 的值必须是一个编译期的常量，那这个内建函数的意思就是 exp 的预期值为 c，编译器可以根据这个信息适当的重排条件语句块的顺序，将符合这个条件的分支放在合适的地方。具体到 unlikely(x)就是告诉编译器条件 x 发生的可能性不大，那么这个条件块儿里语句的目标码可能就会被放在一个比较远的为止，以保证经常执行的目标码更紧凑。likely 则相反。在一个月高风黑的夜晚，你向你从穿开裆裤就开始暗恋的 mm 表白，她回复你

if(unlikely(你以后会有房子，不是按揭的，会有车子，不是奥拓))

咱们明天可以去领证儿

你觉得你是应该高兴还是不应该高兴？她这明摆着就是告你别做梦了，你有房子车子的可能性太小了，往远处排吧，她要先照顾有房有车的。如果没看过内核，你说不定还兴高采烈觉得成了那，你会觉得以后房子车子还不是小 case。所以，这里要感叹一下，读内核是多么重要啊。

使用的时候还是很简单的，就是，if 语句你照用，只是如果你觉得 if 条件为 1 的可能性非常大的时候，可以在条件表达式外面包装一个 likely()，如果可能性非常小，则用 unlikely()包装。那么这里 272 行的意思就很明显了，就是说写内核的哥们儿觉得你的 usb 设备直接连接到 root hub 上的可能性比较小，因为 parent 指的就是你的设备连接的那个 hub。

272~291 行整个的代码就是首先判断你的设备是不是直接连到 root hub 上的，如果是，将 dev->devpath[0]赋值为‘0’，以示特殊，然后父设备设为 controller，同时把 dev->bus_id[]设置为像 usb1/usb2/usb3/usb4 这样的字符串。如果你的设备不是直接连到 root hub 上的，有两种情况，如果你的设备连接的那个 hub 是直接连到 root hub 上的，则 dev->devpath 就等于端口号，否则 dev->devpath 就等于在父 hub 的 devpath 基础上加一个‘.’再加一个端口号，最后把 bus_id[]设置成 1-/2-/3-/4-这样的字符串后面连接上 devpath。

293~295 行，没什么说的，轻轻的飘过。

296 行，初始化一个队列，usbfs 用的。

298~302 行，电源管理的，仍然飘过。

设备的生命线（二）

现在设备的 struct usb_device 结构体已经准备好了，只是还不怎么饱满，hub 接下来就会给它做做整容手术，往里边儿塞点什么，充实一些内容，比如：将设备的状态设置为 Powered，也就是加电状态；因为此时还不知道设备支持的速度，于是将设备的 speed 成员暂时先设置为 USB_SPEED_UNKNOWN；设备的级别 level 当然会被设置为 hub 的 level 加上 1 了；还有为设备能够从 hub 那里获得的电流赋值；为了保证通信畅通，hub 还会为设备在总上选择一个独一无二的地址。

经历了几千里路的云和月，属于设备的那个 struct usb_device 结构体现在又有了多少的功名尘与土？给张表吧，集中列了下到目前为止，设备结构体里成员的状况，这里不用藏着掖着，整就整了，不都是为了生活么。里面的 taken 只是表示赋过值了，好像期末考试前你去突击自习，这时一个 ppmm 走到你旁边，用一个美妙的声音问你“Is this seat taken?”，你怎么回答？当然是“No, No, please.”，那如果星爷片子里面的如花过来问你

那？你回答什么？对头，taken 就是这个意思。那赋了什么值？没必要知道的那么详细，给别人留点自尊，歌里唱的好，人人都需要一点隐私。

devnum	taken
devpath[16]	taken
state	USB_STATE_POWERED
speed	USB_SPEED_UNKNOWN
parent	设备连接的那个 hub
bus	设备连接的那条总线
ep0	ep0.urb_list，描述符长度/类型
dev	dev.bus，dev.type，dev.dma_mask,，dev.parent，dev.bus_id
ep_in[16]	ep_in[0]
ep_out[16]	ep_out[0]
bus_mA	hub->mA_per_port
portnum	设备连接在 hub 上的那个端口
level	hdev->level + 1
filelist	taken
pm_mutex	taken
autosuspend	taken
autosuspend_delay	2 * HZ

还记不记得无间道里的那个傻强？“我是傻强，我是傻的。”这种经典的台词儿也不是什么时候都有的。那即使是傻强过来瞅瞅，也知道你的设备现在已经处在了 Powered 状态。前面讲过的，设备要想从 Powered 状态发展到下一个状态 Default，必须收到一个复位信号并成功复位。就好像你要想和你 mm 从暧昧关系发展到你梦寐以求的以身相许的地步，不是随随便便就可以的，情调、money 一个都不能少。那 hub 接下来的动作就很明显了，复位设备，复位成功后，设备就会进入 Default 状态。

白天停水，晚上停电，发不出工资，买不起面，打开邓选找到答案，原来是社会主义初级阶段，翻到最后，我靠，一百年不变。设备复位不需要一百年那么久，咱们也等不起，顺利的话也就那么几十毫秒的功夫，都不值得秒针动一下。不顺利的话，它会多尝试几次，难道你追 mm 时，表白一次不成功就撤退了？那也忒……了些，白受了这么多年教育。不过如果试了几次都复位不成，那就不用试了，这条设备的生命线就算提前玩完儿了。同样的道理，如果你表白了很多次都不成，那还是换个目标吧，虽说失败是成功他妈，挫折本身并不可怕，但可怕的是成功他妈也太多了些了。

现在就算设备成功复位了，大步迈进了 Default 状态，同时，hub 也会获得设备真正的速度，低速、全速也好，高速也罢，总算是浮出水面了，speed 也终于知道了自己的真正身份，不用再是 UNKNOWN 了。那根据这个速度，咱们能知道些什么？起码能够知道端点 0 一次能够处理的最大数据长度啊，协议里说，对于高速设备，这个值为为 64 字节，对于低速设备为 8 字节，而对于全速设备可能为 8，16，32，64 其中的一个。遇到这种模棱两

可的答案，写代码的哥们儿不会满足，咱们也不会满足，所以 hub 还要通过一个蜿蜒曲折的过程去获得这个确定的值，至于怎么个曲折法儿，此处省略 2008 字。

hub 也辛苦的蛮久了，设备也该进入 Address 状态了，也要知恩图报啊。好像任小强们辛苦了那么久盖了那么多的房子之后，给咱们说，你们快进入 Address 吧，不然兄弟们撑不下去了，那你怎么办，上下三四代，左右五六家的凑呗，于是 21 世纪的奇观房奴大军形成了。

咱们的设备要想进入 Address 状态没那么费劲儿，只要 hub 使用 core 里定义的一个函数 usb_control_msg，发送 SET_ADDRESS 请求给设备，设备就兴高采烈的迈进 Address 了。那么设备的这个 address 是什么，就是上面的 devnum 啊，说过的。不用羡慕它们太久，经过了 10 年，百年，千年，咱们也会 Address 的。

那现在咱就来说说这个 usb_control_msg 函数，它在 message.c 里定义

```
94 /**
95  *      usb_control_msg - Builds a control urb, sends it off and waits for
completion
96  *      @dev: pointer to the usb device to send the message to
97  *      @pipe: endpoint "pipe" to send the message to
98  *      @request: USB message request value
99  *      @requesttype: USB message request type value
100  *      @value: USB message value
101  *      @index: USB message index value
102  *      @data: pointer to the data to send
103  *      @size: length in bytes of the data to send
104  *      @timeout: time in msecs to wait for the message to complete before
105  *              timing out (if 0 the wait is forever)
106  *      Context: !in_interrupt ()
107  *
108  *      This function sends a simple control message to a specified endpoint
109  *      and waits for the message to complete, or timeout.
110  *
111  *      If successful, it returns the number of bytes transferred, otherwise
a negative error number.
112  *
113  *      Don't use this function from within an interrupt context, like a
114  *      bottom half handler.  If you need an asynchronous message, or need to
send
115  *      a message from within interrupt context, use usb_submit_urb()
116  *      If a thread in your driver uses this call, make sure your disconnect()
117  *      method can wait for it to complete.  Since you don't have a handle on
118  *      the URB used, you can't cancel the request.
119  */
```

```
120 int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
__u8 requesttype,
121                           __u16 value, __u16 index, void *data, __u16 size, int
timeout)
122 {
123         struct usb_ctrlrequest *dr = kmalloc(sizeof(struct usb_ctrlrequest),
GFP_NOIO);
124         int ret;
125
126         if (!dr)
127                 return -ENOMEM;
128
129         dr->bRequestType= requesttype;
130         dr->bRequest = request;
131         dr->wValue = cpu_to_le16p(&value);
132         dr->wIndex = cpu_to_le16p(&index);
133         dr->wLength = cpu_to_le16p(&size);
134
135         //dbg("usb_control_msg");
136
137         ret = usb_internal_control_msg(dev, pipe, dr, data, size, timeout);
138
139         kfree(dr);
140
141         return ret;
142 }
```

这个函数主要目的是创建一个控制 urb，并把它发送给 usb 设备，然后等待它完成。urb 是什么？忘了么，前面提到过的，你要想和你的 usb 通信，就得创建一个 urb，并且为它赋好值，交给 usb core，它会找到合适的 host controller，从而进行具体的数据传输。而且我还说会挑选一个黄道吉日，对它大书特书。那现在是不是那个黄道吉日？甭着急，现在还不是，一步一步来，到了我会说的，俺可是一诺千斤石头的，对自己的 rp 有信心。题外话，你知道最差的 rp 是什么？最差的 rp 莫过于痴痴地盯着一个丑女看半晌，然后叹口气说：“靠，这恐龙做得太像真的了……”

123 行，为一个 struct usb_ctrlrequest 结构体申请了内存，这里又出现了一个新生事物，郭天王早提醒我们了，这里的结构说不完。它在 include/linux/usb/ch9.h 文件里定义

```
123 /**
124  * struct usb_ctrlrequest - SETUP data for a USB device control request
125  * @bRequestType: matches the USB bmRequestType field
126  * @bRequest: matches the USB bRequest field
127  * @wValue: matches the USB wValue field (le16 byte order)
128  * @wIndex: matches the USB wIndex field (le16 byte order)
```

```
129  * @wLength: matches the USB wLength field (le16 byte order)
130  *
131  * This structure is used to send control requests to a USB device.  It matches
132  * the different fields of the USB 2.0 Spec section 9.3, table 9-2.  See the
133  * USB spec for a fuller description of the different fields, and what they are
134  * used for.
135  *
136  * Note that the driver for any interface can issue control requests.
137  * For most devices, interfaces don't coordinate with each other, so
138  * such requests may be made at any time.
139  */
140 struct usb_ctrlrequest {
141         __u8 bRequestType;
142         __u8 bRequest;
143         __le16 wValue;
144         __le16 wIndex;
145         __le16 wLength;
146 } __attribute__ ((packed));
```

这个结构完全对应于 spec 里的 Table 9-2，描述了主机通过控制传输发送给设备的请求（Device Requests）。一直都在羡慕它们，这会儿到体现出咱们的优越了，主机向设备请求些信息必须得按照协议里规定好的格式，不然设备就会不明白主机是嘛意思，而咱们就不一样了，不用填这个值那个值，你只要唱一句你是风儿我是沙，你 mm 就明明白白你的心了。不过还是要感慨一下，要是爱情也有固定的格式该多好，这样世界上就少了许多无谓的误会和争吵，把这种精力转化到生产力上，国民生产总值又要提高多少个百分点，绝对比房地产这根柱子的贡献大多了。

这个结构描述的 request 都在 Setup 包里发送，Setup 包是前面某处说到的 Token PID 类型中的一种，为了你好理解，这里细说一下控制传输底层的 packet 情况。控制传输最少要有两个阶段的 transaction，SETUP 和 STATUS，SETUP 和 STATUS 中间的那个 DATA 阶段是可有可无的。Transaction 这个词儿在很多地方都有，也算是个跨地区跨学科的热门词汇了，在这里你称它为事务也好，会话也罢，我还是直呼它的原名 transaction，可以理解为主机和设备之间形成的一次完整的交流，比如 2004 中华小姐环球大赛总决赛上评委蔡澜和陕西选手姚佳雯之间的对话：

要老公还是要钱？要钱。

这就可以算是一次 transaction，还有

要父母还是要钱？要父母。

要国家还是要钱？要钱。

这些也都算是 transactions，usb 的 transaction 要比上面的对话复杂，起码要过过脑子，它可以包括一个 Token 包、一个 Data 包和一个 Handshake 包。

Token、Data 和 Handshake 都属于四种 PID 类型中的，前面说到时提到的一个包里的那些部分，如 SYNC、PID、地址域、DATA、CRC，并不是所有 PID 类型的包都会全部包括的。Token 包只包括 SYNC、PID、地址域、CRC，并没有 DATA 字段，它的名字起的很形象，就是用来标记所在 transaction 里接下来动作的，对于 Out 和 Setup Token 包，里面的地址域指明了接下来要接收 Data 包的端点，对于 In Token 包，地址域指明了接下来哪个端点要发送 Data 包。还有，只有主机才有权利发送 Token 包，协议里就这么规定的。别嫌 spec 规定太多，又管这儿又管那儿的，没有规矩不成方圆，连咱们首都机场的的边检民警们都规定了，微笑服务必须露出 8 颗牙齿，上边儿四颗下边儿四颗，多一颗少一颗都不合格直接 kick，那嘴小的看不到 8 颗咋办？嘴巴剪大点，嘴大的露的太多咋办？那就缝住，那虎牙咋办？这可是技术难题，不好解决啊，不是中科院、自然科学基金等这几家正在征募 10000 科学难题么，你可以拿它去申报一下，不定还能拿点资助什么的。

与 Token 包相比，Data 包里没了地址域，多了 Data 字段，这个 Data 字段对于低速设备最大为 8 字节，对于全速设备最大为 1023 字节，对于高速设备最大为 1024 字节。里里外外看过去，它就是躲在 Token 后边儿用来传输数据的。Handshake 包的成分就非常的简单了，简直和那位姚佳雯的回答一样简单，除了 SYNC，它就只包含了一个 PID，通过 PID 取不同的值来报告一个 transaction 的状态，比如数据已经成功接收了等。

俗话说，兔有三窟，人有三急，控制传输的 SETUP transaction 一般来说也有三个阶段，就是主机向设备发送 Setup Token 包、然后发送 Data0 包，如果一切顺利，设备回应 ACK Handshake 包表示 OK，为什么加上一般？如果中间的那个 Data0 包由于某种不可知因素被损坏了，设备就什么都不会回应，这时就成俩阶段了。SETUP transaction 之后，接下来如果控制传输有 DATA transaction 的话，那就 Data0、Data1 这样交叉的发送数据包，前面说过这是为了实现 data toggle。最后是 STATUS transaction，向主机汇报前面 SETUP 和 DATA 阶段的结果，比如表示主机下达的命令已经完成了，或者主机下达的命令没有完成，或者设备正忙着那没功夫去理会主机的那些命令。

这样经过 SETUP、DATA、STATUS 这三个 transaction 阶段，一个完整的控制传输完成了。主机接下来可以规划下一次的控制传输。

现在对隐藏在控制传输背后的是是非非摸了个底儿，群众的眼睛是雪亮的，咱们现在应该可以看出之前说 requests 都在 Setup 包里发送是有问题的，因为 Setup 包本身并没有数据字段，严格来说它们应该都是在 SETUP transaction 阶段里 Setup 包后的 Data0 包里发送的。还有点糊涂？唉，女人之美，在于蠢得无怨无悔；男人之美，在于说得白日见鬼；诗歌之美，在于煽动男女出轨；学问之美，在于使人一头雾水。那就带着雾水向下看吧，可能会有那么云开雾明的一天。

141 行，bRequestType，这个字段别看就一个字节，内容很丰富的，大道理往往都包含

这种在小地方。它的 bit7 就表示了控制传输中 DATA transaction 阶段的方向，当然，如果有 DATA 阶段的话。bit5~6 表示 request 的类型，是标准的，class-specific 的还是 vendor-specific 的。bit0~4 表示了这个请求针对的是设备，接口，还是端点。内核为它们专门量身定做了一批掩码，也在 ch9.h 文件里，

```
42 /*
43  * USB directions
44  *
45  * This bit flag is used in endpoint descriptors' bEndpointAddress field.
46  * It's also one of three fields in control requests bRequestType.
47  */
48 #define USB_DIR_OUT                     0               /* to device */
49 #define USB_DIR_IN                      0x80            /* to host */
50
51 /*
52  * USB types, the second of three bRequestType fields
53  */
54 #define USB_TYPE_MASK                   (0x03 << 5)
55 #define USB_TYPE_STANDARD               (0x00 << 5)
56 #define USB_TYPE_CLASS                  (0x01 << 5)
57 #define USB_TYPE_VENDOR                 (0x02 << 5)
58 #define USB_TYPE_RESERVED               (0x03 << 5)
59
60 /*
61  * USB recipients, the third of three bRequestType fields
62  */
63 #define USB_RECIP_MASK                  0x1f
64 #define USB_RECIP_DEVICE                0x00
65 #define USB_RECIP_INTERFACE             0x01
66 #define USB_RECIP_ENDPOINT              0x02
67 #define USB_RECIP_OTHER                 0x03
68 /* From Wireless USB 1.0 */
69 #define USB_RECIP_PORT                  0x04
70 #define USB_RECIP_RPIPE                 0x05
```

142 行，bRequest，表示具体是哪个 request。

143 行，wValue，这个字段是 request 的参数，request 不同，wValue 就不同。

144 行，wIndex，也是 request 的参数，bRequestType 指明 request 针对的是设备上的某个接口或端点的时候，wIndex 就用来指明是哪个接口或端点。

145 行，wLength，控制传输中 DATA transaction 阶段的长度，方向已经在 bRequestType 那儿指明了。如果这个值为 0，就表示没有 DATA transaction 阶段，

bRequestType 的方向位也就无效了。

和 struct usb_ctrlrequest 的约会暂时就到这里，回到 usb_control_msg 函数里。很明显要进行控制传输，得首先创建一个 struct usb_ctrlrequest 结构体，填上请求的内容。129 到 133 行就是来使用传递过来的参数初始化这个结构体的。对于刚开始提到的 SET_ADDRESS 来说，bRequest 的值就是 USB_REQ_SET_ADDRESS，标准请求之一，ch9.h 里定义有。因为 SET_ADDRESS 请求并不需要 DATA 阶段，所以 wLength 为 0，而且这个请求是针对设备的，所以 wIndex 也为 0。这么一来，bRequestType 的值也只能为 0 了。因为是设置设备地址的，总得把要设置的地址发给设备，不然设备会比咱们还一头雾水不知道主机是嘛个意思，所以请求的参数 wValue 就是之前 hub 已经你的设备指定好的 devnum。其实 SET_ADDRESS 请求各个部分的值 spec 9.4.6 里都有规定，就和我这里说的一样，不信你去看看。

接下来先看 139 行，走到这儿就表示成也好败也好，总之这次通信已经完成了，那么 struct usb_ctrlrequest 结构体也就没用了，没用的东西要好不犹豫的精简掉。

回头看 137 行，这是引领咱们往深处走了，不过不怕，路有多远，咱们看下去的决心就有多远。

设备的生命线（三）

函数usb_control_msg调用了 usb_internal_control_msg之后就去一边儿睡大觉了，脏活儿累活儿，全部留给 usb_internal_control_msg去做了，这才叫骨干啊，俺一华为的哥们儿如是说。那么咱们接下来就给这个骨干多点儿关注，了解一下它背后的真实生活，现在是焦点访谈时间，要用事实说话。

```
70 // returns status (negative) or length (positive)
71 static int usb_internal_control_msg(struct usb_device *usb_dev,
72                                     unsigned int pipe,
73                                     struct usb_ctrlrequest *cmd,
74                                     void *data, int len, int timeout)
75 {
76         struct urb *urb;
77         int retv;
78         int length;
79
80         urb = usb_alloc_urb(0, GFP_NOIO);
81         if (!urb)
82                 return -ENOMEM;
83
```

```
84          usb_fill_control_urb(urb, usb_dev, pipe, (unsigned char *)cmd, data,
85                               len, usb_api_blocking_completion, NULL);
86
87          retv = usb_start_wait_urb(urb, timeout, &length);
88          if (retv < 0)
89                  return retv;
90          else
91                  return length;
92 }
```

这个函数粗看过去，可以概括为一个中心，三个基本点，以一个struct urb结构体为中心，以 usb_alloc_urb、usb_fill_control_urb、usb_start_wait_urb三个函数为基本点。

一个中心：struct urb结构体，就是咱们前面多次提到又多次飘过，只闻其名不见其形的传说中的urb，全称usb request block，站在咱们的角度看，usb通信靠的就是它这张脸。

第一个基本点：usb_alloc_urb函数，创建一个urb，struct urb结构体只能使用它来创建，它是urb在usb世界里的独家代理，和天盛一样的角色。

第二个基本点：usb_fill_control_urb函数，初始化一个控制urb，urb被创建之后，使用之前必须要正确的初始化。

第三个基本点：usb_start_wait_urb函数，将urb提交给咱们的usb core，以便分配给特定的主机控制器驱动进行处理，然后默默的等待处理结果，或者超时。

```
963 /**
964  * struct urb - USB Request Block
965  * @urb_list: For use by current owner of the URB.
966  * @pipe: Holds endpoint number, direction, type, and more.
967  *      Create these values with the eight macros available;
968  *      usb_{snd,rcv}TYPEpipe(dev,endpoint), where the TYPE is "ctrl"
969  *      (control), "bulk", "int" (interrupt), or "iso" (isochronous).
970  *      For example usb_sndbulkpipe() or usb_rcvintpipe().  Endpoint
971  *      numbers range from zero to fifteen.  Note that "in" endpoint two
972  *      is a different endpoint (and pipe) from "out" endpoint two.
973  *      The current configuration controls the existence, type, and
974  *      maximum packet size of any given endpoint.
975  * @dev: Identifies the USB device to perform the request.
976  * @status: This is read in non-iso completion functions to get the
977  *      status of the particular request.  ISO requests only use it
978  *      to tell whether the URB was unlinked; detailed status for
979  *      each frame is in the fields of the iso_frame-desc.
980  * @transfer_flags: A variety of flags may be used to affect how URB
981  *      submission, unlinking, or operation are handled.  Different
```

```
982  *      kinds of URB can use different flags.
983  * @transfer_buffer:  This identifies the buffer to (or from) which
984  *      the I/O request will be performed (unless URB_NO_TRANSFER_DMA_MAP
985  *      is set).  This buffer must be suitable for DMA; allocate it with
986  *      kmalloc() or equivalent.  For transfers to "in" endpoints, contents
987  *      of this buffer will be modified.  This buffer is used for the data
988  *      stage of control transfers.
989  * @transfer_dma: When transfer_flags includes URB_NO_TRANSFER_DMA_MAP,
990  *      the device driver is saying that it provided this DMA address,
991  *      which the host controller driver should use in preference to the
992  *      transfer_buffer.
993  * @transfer_buffer_length: How big is transfer_buffer.  The transfer may
994  *      be broken up into chunks according to the current maximum packet
995  *      size for the endpoint, which is a function of the configuration
996  *      and is encoded in the pipe.  When the length is zero, neither
997  *      transfer_buffer nor transfer_dma is used.
998  * @actual_length: This is read in non-iso completion functions, and
999  *      it tells how many bytes (out of transfer_buffer_length) were
1000 *      transferred.  It will normally be the same as requested, unless
1001 *      either an error was reported or a short read was performed.
1002 *      The URB_SHORT_NOT_OK transfer flag may be used to make such
1003 *      short reads be reported as errors.
1004 * @setup_packet: Only used for control transfers, this points to eight bytes
1005 *      of setup data.  Control transfers always start by sending this data
1006 *      to the device.  Then transfer_buffer is read or written, if needed.
1007 * @setup_dma: For control transfers with URB_NO_SETUP_DMA_MAP set, the
1008 *      device driver has provided this DMA address for the setup packet.
1009 *      The host controller driver should use this in preference to
1010 *      setup_packet.
1011 * @start_frame: Returns the initial frame for isochronous transfers.
1012 * @number_of_packets: Lists the number of ISO transfer buffers.
1013 * @interval: Specifies the polling interval for interrupt or isochronous
1014 *      transfers.  The units are frames (milliseconds) for for full and low
1015 *      speed devices, and microframes (1/8 millisecond) for highspeed ones.
1016 * @error_count: Returns the number of ISO transfers that reported errors.
1017 * @context: For use in completion functions.  This normally points to
1018 *      request-specific driver context.
1019 * @complete: Completion handler. This URB is passed as the parameter to the
1020 *      completion function.  The completion function may then do what
1021 *      it likes with the URB, including resubmitting or freeing it.
1022 * @iso_frame_desc: Used to provide arrays of ISO transfer buffers and to
1023 *      collect the transfer status for each buffer.
1024 *
1025 * This structure identifies USB transfer requests. URBs must be allocated by
```

```
1026  * calling usb_alloc_urb() and freed with a call to usb_free_urb().
1027  * Initialization may be done using various usb_fill_*_urb() functions.  URBs
1028  * are submitted using usb_submit_urb(), and pending requests may be canceled
1029  * using usb_unlink_urb() or usb_kill_urb().
1030  *
1031  * Data Transfer Buffers:
1032  *
1033  * Normally drivers provide I/O buffers allocated with kmalloc() or otherwise
1034  * taken from the general page pool.  That is provided by transfer_buffer
1035  * (control requests also use setup_packet), and host controller drivers
1036  * perform a dma mapping (and unmapping) for each buffer transferred.  Those
1037  * mapping operations can be expensive on some platforms (perhaps using a dma
1038  * bounce buffer or talking to an IOMMU),
1039  * although they're cheap on commodity x86 and ppc hardware.
1040  *
1041  * Alternatively, drivers may pass the URB_NO_xxx_DMA_MAP transfer flags,
1042  * which tell the host controller driver that no such mapping is needed since
1043  * the device driver is DMA-aware.  For example, a device driver might
1044  * allocate a DMA buffer with usb_buffer_alloc() or call usb_buffer_map().
1045  * When these transfer flags are provided, host controller drivers will
1046  * attempt to use the dma addresses found in the transfer_dma and/or
1047  * setup_dma fields rather than determining a dma address themselves.  (Note
1048  * that transfer_buffer and setup_packet must still be set because not all
1049  * host controllers use DMA, nor do virtual root hubs).
1050  *
1051  * Initialization:
1052  *
1053  * All URBs submitted must initialize the dev, pipe, transfer_flags (may be
1054  * zero), and complete fields.  All URBs must also initialize
1055  * transfer_buffer and transfer_buffer_length.  They may provide the
1056  * URB_SHORT_NOT_OK transfer flag, indicating that short reads are
1057  * to be treated as errors; that flag is invalid for write requests.
1058  *
1059  * Bulk URBs may
1060  * use the URB_ZERO_PACKET transfer flag, indicating that bulk OUT transfers
1061  * should always terminate with a short packet, even if it means adding an
1062  * extra zero length packet.
1063  *
1064  * Control URBs must provide a setup_packet.  The setup_packet and
1065  * transfer_buffer may each be mapped for DMA or not, independently of
1066  * the other.  The transfer_flags bits URB_NO_TRANSFER_DMA_MAP and
1067  * URB_NO_SETUP_DMA_MAP indicate which buffers have already been mapped.
1068  * URB_NO_SETUP_DMA_MAP is ignored for non-control URBs.
1069  *
```

```
1070  * Interrupt URBs must provide an interval, saying how often (in milliseconds
1071  * or, for highspeed devices, 125 microsecond units)
1072  * to poll for transfers.  After the URB has been submitted, the interval
1073  * field reflects how the transfer was actually scheduled.
1074  * The polling interval may be more frequent than requested.
1075  * For example, some controllers have a maximum interval of 32 milliseconds,
1076  * while others support intervals of up to 1024 milliseconds.
1077  * Isochronous URBs also have transfer intervals.  (Note that for isochronous
1078  * endpoints, as well as high speed interrupt endpoints, the encoding of
1079  * the transfer interval in the endpoint descriptor is logarithmic.
1080  * Device drivers must convert that value to linear units themselves.)
1081  *
1082  * Isochronous URBs normally use the URB_ISO_ASAP transfer flag, telling
1083  * the host controller to schedule the transfer as soon as bandwidth
1084  * utilization allows, and then set start_frame to reflect the actual frame
1085  * selected during submission.  Otherwise drivers must specify the start_frame
1086  * and handle the case where the transfer can't begin then.  However, drivers
1087  * won't know how bandwidth is currently allocated, and while they can
1088  * find the current frame using usb_get_current_frame_number () they can't
1089  * know the range for that frame number.  (Ranges for frame counter values
1090  * are HC-specific, and can go from 256 to 65536 frames from "now".)
1091  *
1092  * Isochronous URBs have a different data transfer model, in part because
1093  * the quality of service is only "best effort".  Callers provide specially
1094  * allocated URBs, with number_of_packets worth of iso_frame_desc structures
1095  * at the end.  Each such packet is an individual ISO transfer.  Isochronous
1096  * URBs are normally queued, submitted by drivers to arrange that
1097  * transfers are at least double buffered, and then explicitly resubmitted
1098  * in completion handlers, so
1099  * that data (such as audio or video) streams at as constant a rate as the
1100  * host controller scheduler can support.
1101  *
1102  * Completion Callbacks:
1103  *
1104  * The completion callback is made in_interrupt(), and one of the first
1105  * things that a completion handler should do is check the status field.
1106  * The status field is provided for all URBs.  It is used to report
1107  * unlinked URBs, and status for all non-ISO transfers.  It should not
1108  * be examined before the URB is returned to the completion handler.
1109  *
1110  * The context field is normally used to link URBs back to the relevant
1111  * driver or request state.
1112  *
1113  * When the completion callback is invoked for non-isochronous URBs, the
```

```
1114  * actual_length field tells how many bytes were transferred.  This field
1115  * is updated even when the URB terminated with an error or was unlinked.
1116  *
1117  * ISO transfer status is reported in the status and actual_length fields
1118  * of the iso_frame_desc array, and the number of errors is reported in
1119  * error_count.  Completion callbacks for ISO transfers will normally
1120  * (re)submit URBs to ensure a constant transfer rate.
1121  *
1122  * Note that even fields marked "public" should not be touched by the driver
1123  * when the urb is owned by the hcd, that is, since the call to
1124  * usb_submit_urb() till the entry into the completion routine.
1125  */
1126 struct urb
1127 {
1128         /* private: usb core and host controller only fields in the urb */
1129         struct kref kref;               /* reference count of the URB */
1130         spinlock_t lock;                /* lock for the URB */
1131         void *hcpriv;                   /* private data for host controller
1132 */
1132         atomic_t use_count;             /* concurrent submissions counter */
1133         u8 reject;                      /* submissions will fail */
1134
1135        /* public: documented fields in the urb that can be used by drivers */
1136         struct list_head urb_list;      /* list head for use by the urb's
1137                                          * current owner */
1138         struct usb_device *dev;        /* (in) pointer to associated device */
1139         unsigned int pipe;              /* (in) pipe information */
1140         int status;                     /* (return) non-ISO status */
1141         unsigned int transfer_flags;    /* (in) URB_SHORT_NOT_OK | ...*/
1142         void *transfer_buffer;          /* (in) associated data buffer */
1143         dma_addr_t transfer_dma;       /* (in) dma addr for transfer_buffer */
1144         int transfer_buffer_length;     /* (in) data buffer length */
1145         int actual_length;              /* (return) actual transfer length */
1146         unsigned char *setup_packet;    /* (in) setup packet (control only) */
1147         dma_addr_t setup_dma;           /* (in) dma addr for setup_packet */
1148         int start_frame;                /* (modify) start frame (ISO) */
1149         int number_of_packets;          /* (in) number of ISO packets */
1150         int interval;                   /* (modify) transfer interval
1151                                          * (INT/ISO) */
1152         int error_count;                /* (return) number of ISO errors */
1153         void *context;                  /* (in) context for completion */
1154         usb_complete_t complete;        /* (in) completion routine */
1155         struct usb_iso_packet_descriptor iso_frame_desc[0];
1156                                         /* (in) ISO ONLY */
```

```
1157 };
```

“年度最大赃物”——俄罗斯一神秘男子偷走跨度达 5 米的整座钢桥切割后当废铁卖，俄罗斯警方破案后称。

“年度最长结构”——经过了惊情四百年，看过了上下数百行，我们对 struct urb 说。

1129 行，kref，urb 的引用计数。甭看它是隐藏在 urb 内部的一个不起眼的小角色，但小角色做大事情，它决定了一个 urb 的生死存亡。一个 urb 有用没用，是继续委以重任还是无情销毁都要看它的脸色。那第一个问题就来了，为什么 urb 的生死要掌握在这个小小的引用计数手里边儿？

很早很早很早很早以前就说过，主机与设备之间通过管道来传输数据，管道的一端是主机上的一个缓冲区，另一端是设备上的端点。管道之中流动的数据，在主机控制器和设备看来是一个个 packets，在咱们看来就是 urb。因而，端点之中就有那么一个队列，叫 urb 队列。不过，这并不代表一个 urb 只能发配给一个端点，它可能通过不同的管道发配给不同的端点，那么这样一来，我们如何知道这个 urb 正在被多少个端点使用，如何判断这个 urb 的生命已经 over？如果没有任何一个端点在使用它，而我们又无法判断这种情况，它就会永远的飘荡在 usb 的世界里，犹如飘荡在人冥两届的冤魂。我们需要寻求某种办法在这种情况下给它们一个好的归宿，这就是引用计数。每多一个使用者，它的这个引用计数就加 1，每减少一个使用者，引用计数就减一，如果连最后一个使用者都释放了这个 urb，宣称不再使用它了，那它的生命周期就走到了尽头，会自动的销毁。

接下来就是第二个问题，如何来表示这个神奇的引用计数？其实它是一个 struct kref 结构体，在 include/linux/kref.h 里定义

```
23 struct kref {
24         atomic_t refcount;
25 };
```

这个结构与 struct urb 相比简约到极致了，简直就是迎着咱们的口味来的。不过别看它简单，内核里就是使用它来判断一个对象还有没有用的。它里边儿只包括了一个原子变量，为什么是原子变量？既然都使用引用计数了，那就说明可能同时有多个地方在使用这个对象，总要考虑一下它们同时修改这个计数的可能性吧，也就是俗称的并发访问，那怎么办？加个锁？就这么一个整数值专门加个锁未免也忒大材小用了些，所以就使用了原子变量。围绕这个结构，内核里还定义了几个专门操作引用计数的函数，它们在 lib/kref.c 里定义

```
17 /**
18  * kref_init - initialize object.
19  * @kref: object in question.
20  */
21 void kref_init(struct kref *kref)
22 {
```

```
23          atomic_set(&kref->refcount,1);
24          smp_mb();
25 }
26
27 /**
28  * kref_get - increment refcount for object.
29  * @kref: object.
30  */
31 void kref_get(struct kref *kref)
32 {
33          WARN_ON(!atomic_read(&kref->refcount));
34          atomic_inc(&kref->refcount);
35          smp_mb__after_atomic_inc();
36 }
37
38 /**
39  * kref_put - decrement refcount for object.
40  * @kref: object.
41  * @release: pointer to the function that will clean up the object when the
42  *           last reference to the object is released.
43  *           This pointer is required, and it is not acceptable to pass kfree
44  *           in as this function.
45  *
46  * Decrement the refcount, and if 0, call release().
47  * Return 1 if the object was removed, otherwise return 0.  Beware, if this
48  * function returns 0, you still can not count on the kref from remaining in
49  * memory.  Only use the return value if you want to see if the kref is now
50  * gone, not present.
51  */
52 int kref_put(struct kref *kref, void (*release)(struct kref *kref))
53 {
54          WARN_ON(release == NULL);
55          WARN_ON(release == (void (*)(struct kref *))kfree);
56
57          if (atomic_dec_and_test(&kref->refcount)) {
58                  release(kref);
59                  return 1;
60          }
61          return 0;
62 }
```

整个kref.c文件就定义了这么三个函数，kref_init初始化，kref_get将引用计数加 1，kref_put将引用计数减一并判断是不是为 0，为 0 的话就调用参数里release函数指针指向的函数把对象销毁掉。它们对独苗儿refcount的操作都是通过原子变量特有的操作函数，

其实这句话可以当选当日最大废话，原子变量当然要使用专门的操作函数了，编译器还能做些优化，否则直接使用一般的变量就可以了干吗还要使用原子变量，不是没事找事儿么，再说如果你直接像对待一般整型值一样对待它，编译器也会看不过去你的行为，直接给你个error的。友情提醒一下，kref_init初始化时，是把refcount的值初始化为 1 了的，不是 0。还有一点要说的是 kref_put参数里的那个函数指针，你不能传递一个NULL过去，否则这个引用计数就只是计数，而背离了最初的目的，要记住我们需要在这个计数减为 0 的时候将嵌入这个引用计数struct kref结构体的对象给销毁掉，所以这个函数指针也不能为kfree，因为这样的话就只是把这个struct kref结构体给销毁了，而不是整个对象。

第三个问题，如何使用 struct kref 结构来为我们的对象计数？当然我们需要把这样一个结构嵌入到你希望计数的对象里边，不然你根本就无法对对象在它整个生命周期里的使用情况作出判断，难道还真能以为 Linus 是 linux 里的太阳，像太阳神阿波罗一样掐指一算就知道谁谁在哪儿待过，现在在哪儿。但是我们应该是几乎见不到内核里边儿直接使用上面那几个函数来给对象计数的，而是每种对象又定义了自己专用的引用计数函数，比如咱们的 urb，在 urb.c 里定义

```
17 /**
18  * usb_init_urb - initializes a urb so that it can be used by a USB driver
19  * @urb: pointer to the urb to initialize
20  *
21  * Initializes a urb so that the USB subsystem can use it properly.
22  *
23  * If a urb is created with a call to usb_alloc_urb() it is not
24  * necessary to call this function.  Only use this if you allocate the
25  * space for a struct urb on your own.  If you call this function, be
26  * careful when freeing the memory for your urb that it is no longer in
27  * use by the USB core.
28  *
29  * Only use this function if you _really_ understand what you are doing.
30  */
31 void usb_init_urb(struct urb *urb)
32 {
33         if (urb) {
34                 memset(urb, 0, sizeof(*urb));
35                 kref_init(&urb->kref);
36                 spin_lock_init(&urb->lock);
37         }
38 }
39
71 /**
72  * usb_free_urb - frees the memory used by a urb when all users of it are finished
73  * @urb: pointer to the urb to free, may be NULL
74  *
```

```
75  * Must be called when a user of a urb is finished with it.  When the last user
76  * of the urb calls this function, the memory of the urb is freed.
77  *
78  * Note: The transfer buffer associated with the urb is not freed, that must be
79  * done elsewhere.
80  */
81 void usb_free_urb(struct urb *urb)
82 {
83         if (urb)
84                 kref_put(&urb->kref, urb_destroy);
85 }
86
87 /**
88  * usb_get_urb - increments the reference count of the urb
89  * @urb: pointer to the urb to modify, may be NULL
90  *
91  * This must be  called whenever a urb is transferred from a device driver to
a
92  * host controller driver.  This allows proper reference counting to happen
93  * for urbs.
94  *
95  * A pointer to the urb with the incremented reference counter is returned.
96  */
97 struct urb * usb_get_urb(struct urb *urb)
98 {
99         if (urb)
100                 kref_get(&urb->kref);
101         return urb;
102 }
```

usb_init_urb、usb_get_urb、usb_free_urb 这三个函数分别调用了前面看到的 struct kref 结构的三个操作函数来进行引用计数的初始化、加 1、减一。什么叫封装？这就叫封装。usb_init_urb 和 usb_get_urb 都没什么好说的，比较感兴趣的是 usb_free_urb 里给 kref_put 传递的那个函数 urb_destroy，它也在 urb.c 里定义

```
9 #define to_urb(d) container_of(d, struct urb, kref)
10
11 static void urb_destroy(struct kref *kref)
12 {
13         struct urb *urb = to_urb(kref);
14         kfree(urb);
15 }
```

这个 urb_destroy 首先调用了 to_urb，实际上就是一个 container_of 来获得引用计数关

联的那个 urb，然后使用 kfree 将它销毁。

到此，世界如此美丽，引用计数如此简单，不是么？

1130 行，lock，一把自旋锁。韦唯早就唱了，每个 urb 都有一把自旋锁。

1131 行，hcpriv，走到今天，你应该明白这个 urb 最终还是要提交给主机控制器驱动的，这个字段就是 urb 里主机控制器驱动的自留地，咱们就不插手了。

1132 行，use_count，这里又是一个使用计数，不过此计数非彼计数，它与上面那个用来追踪 urb 生命周期的 kref 一点儿血缘关系也没有，连远亲都不是。那它是用来做什么的，凭什么在臃肿的 struct urb 不断喊着要瘦身的时候还仍有一席之地？

先了解下使用 urb 来完成一次完整的 usb 通信都要经历哪些阶段，首先，驱动程序发现自己有与 usb 设备通信的需要，于是创建一个 urb，并指定它的目的地是设备上的哪个端点，然后提交给 usb core，usb core 将它修修补补的做些美化之后再移交给主机控制器的驱动程序 HCD，HCD 会去解析这个 urb，了解它的目的是什么，并与 usb 设备进行相应的交流，在交流结束，urb 的目的达到之后，HCD 再把这个 urb 的所有权移交回驱动程序。

这里的 use_count 就是在 usb core 将 urb 移交给 HCD，办理移交手续的时候，插上了那么一脚，每当走到这一步，它的值就会加 1。什么时候减 1？在 HCD 重新将 urb 的所有权移交回驱动程序的时候。这样说吧，只要 HCD 拥有这个 urb 的所有权，那么该 urb 的 use_count 就不会为 0。这么一说，似乎 use_count 也有点追踪 urb 生命周期的味道了，当它的值大于 0 时，就表示当前有 HCD 正在处理它，和上面的 kref 概念上有部分的重叠，不过，显然它们之间是有区别的，没区别的话这儿干吗要用两个计数，不是没事儿找抽么。上面的那个 kref 实现方式是内核里统一的引用计数机制，当计数减为 0 时，urb 对象就被 urb_destroy 给销毁了。这里的 use_count 只是用来统计当前这个 urb 是不是正在被哪个 HCD 处理，即使它的值为 0，也只是说明没有 HCD 在使用它而已，并不代表就得把它给销毁掉。比方说，HCD 利用完了 urb，把它还给了驱动，这时驱动还可以对这个 urb 检修检修，再提交给哪个 HCD 去使用。

下面的问题就是既然它不会平白无故的多出来，那它究竟是用来干啥的？还要从刚提到的那几个阶段说起。urb 驱动也创建了，提交也提交了，HCD 正处理着那，可驱动反悔了，它不想再继续这次通信了，想将这个 urb 给终止掉，善解任意的 usb core 当然会给驱动提供这样的接口来满足这样的需要。不过这个需要还被写代码的哥们儿细分为两种，一种是驱动只想通过 usb core 告诉 HCD 一声，说这个 urb 我想终止掉，您就别费心再处理了，然后它不想在那里等着 HCD 的处理，想忙别的事去，这就是俗称的异步，对应的是 usb_unlink_urb 函数。当然对应的还有种同步的，驱动会在那里苦苦等候着 HCD 的处理结果，等待着 urb 被终止，对应的是 usb_kill_urb 函数。而 HCD 将这次通信终止后，同样会将 urb 的所有权移交回驱动。那么驱动通过什么判断 HCD 已经终止了这次通信？就是通过这里的 use_count，驱动会在 usb_kill_urb 里面一直等待着这个值变为 0。

1133 行，reject，拒绝，拒绝什么？不是邀请 ppmm 共进晚餐被拒绝，也不是你要求老板多给点薪水被拒绝，那又是被谁拒绝？

在目前版本的内核里，只有 usb_kill_urb 函数有特权对它进行修改，那么，显然 reject 就与上面说的 urb 终止有关了。那就看看 urb.c 里定义的这个函数

```
444 /**
445  * usb_kill_urb - cancel a transfer request and wait for it to finish
446  * @urb: pointer to URB describing a previously submitted request,
447  *      may be NULL
448  *
449  * This routine cancels an in-progress request.  It is guaranteed that
450  * upon return all completion handlers will have finished and the URB
451  * will be totally idle and available for reuse.  These features make
452  * this an ideal way to stop I/O in a disconnect() callback or close()
453  * function.  If the request has not already finished or been unlinked
454  * the completion handler will see urb->status == -ENOENT.
455  *
456  * While the routine is running, attempts to resubmit the URB will fail
457  * with error -EPERM.  Thus even if the URB's completion handler always
458  * tries to resubmit, it will not succeed and the URB will become idle.
459  *
460  * This routine may not be used in an interrupt context (such as a bottom
461  * half or a completion handler), or when holding a spinlock, or in other
462  * situations where the caller can't schedule().
463  */
464 void usb_kill_urb(struct urb *urb)
465 {
466         might_sleep();
467         if (!(urb && urb->dev && urb->dev->bus))
468                 return;
469         spin_lock_irq(&urb->lock);
470         ++urb->reject;
471         spin_unlock_irq(&urb->lock);
472
473         usb_hcd_unlink_urb(urb, -ENOENT);
474         wait_event(usb_kill_urb_queue, atomic_read(&urb->use_count) == 0);
475
476         spin_lock_irq(&urb->lock);
477         --urb->reject;
478         spin_unlock_irq(&urb->lock);
479 }
```

466 行，因为 usb_kill_urb函数要一直等候着HCD将urb终止掉，它必须是可以休眠的，

不然就太可恶了，就像那些脚踩多只船的，占着本来就稀有的ppmm资源，让大批男同志们找不到另一半来关爱。而历史上，当大批男性无法结婚时，他们就会聚到一起，要么成为和尚，要么结为匪帮，所以说这可是一个严重的社会问题、治安问题。所以说 usb_kill_urb 不能用在中断上下文，必须能够休眠将自己占的资源给让出来。

写代码的哥们儿也都是忧国忧民的主儿，也深刻体会到广大男同胞们的无奈，于是提供了might_sleep函数，用它来判断一下这个函数是不是处在能够休眠的情况，如果不是，就会打印出一大堆的堆栈信息，比如你在中断上下文调用了这个函数时。不过，它也就是基于调试的目的用一用，方便日后找错，并不能强制哪个函数改变自己的上下文。法律也规定了要一夫一妻制，但也就是用来警示警示，如果以为实际上真能禁止些什么，纯粹就是扯淡。

467 行，这里就是判断一下 urb，urb 要去的那个设备，还有那个设备在的总线有没有，如果哪个不存在，就还是返回吧。

469 行，去获得每个urb都有的那把锁，然后将 reject加 1。加 1 有什么用？其实目前版本的内核里只有两个地方用到了这个值进行判断。第一个地方是在usb core将urb提交给HCD，正在办移交手续的时候，如果reject大于 0，就不再接着移交了，也就是说这个urb被HCD给拒绝了。这是为了防止这边儿正在终止这个urb，那边儿某个地方却又妄想将这个urb重新提交给HCD。

473 行，这里告诉HCD驱动要终止这个urb了，usb_hcd_unlink_urb函数也只是告诉HCD一声，然后不管HCD怎么处理就返回了。

474 行，上面的 usb_hcd_unlink_urb是返回了，但并不代表HCD已经将urb给终止了，HCD可能没那么快，所以这里 usb_kill_urb要休息休息，等人通知它。这里使用了wait_event宏来实现休眠，usb_kill_urb_queue是在hcd.h里定义的一个等待队列，专门给 usb_kill_urb休息用的。需要注意的是这里的唤醒条件，use_count必须等于 0，终于看到 use_count实战的地方了。

那在哪里唤醒正在睡大觉的 usb_kill_urb？这牵扯到了第二个使用reject来做判断的地方。在HCD将urb的所有权返还给驱动的时候，会对reject进行判断，如果reject大于 0，就调用wake_up唤醒在 usb_kill_urb_queue上休息的 usb_kill_urb。也好理解，HCD都要将urb的所有权返回给驱动了，那当然就是已经处理完了，放在这里就是已经将这个urb终止了，usb_kill_urb等的就是这一天的到来，当然就要醒过来继续往下走了。

476 行，再次获得 urb 的那把锁，将 reject 刚才增加的那个 1 给减掉。urb 都已经终止了，也没人再会去拒绝它了，reject 还是开始什么样儿结束的时候就什么样吧。

索性将 usb_unlink_urb函数也贴出来看看它们之间有什么区别吧

```
435 int usb_unlink_urb(struct urb *urb)
436 {
```

```
437         if (!urb)
438                 return -EINVAL;
439         if (!(urb->dev && urb->dev->bus))
440                 return -ENODEV;
441         return usb_hcd_unlink_urb(urb, -ECONNRESET);
442 }
```

看 usb_unlink_urb这儿就简单多了，只是把自己的意愿告诉HCD，然后就非常洒脱的返回了。

struct urb 结构里的前边儿这几个，只是 usb core 和主机控制器驱动需要关心的，实际的驱动里根本用不着也管不着，它们就是 usb 和 HCD 的后花园，想种点什么不种点什么都由写这块儿代码的哥们儿决定,他们在里面怎么为所欲为都不关你写驱动的啥事。usb 在 linux 里起起伏伏这么多年，前边儿的这些内容早就变过多少次，说不定你今天还看到谁谁，到接下来的哪天就看不到了，不过，变化的是形式，不变的是道理。驱动要做的只是创建一个 urb，然后初始化，再把它提交给 usb core 就可以了，使用不使用引用计数，加不加锁之类的一点都不用去操心。感谢 David Brownell，感谢 Alan Stern，感谢……，没有他们就没有 usb 在 linux 里的今天。

设备的生命线（四）

洗澡是屁股享福，脑袋吃苦；看电影是脑袋享福，屁股吃苦；看内核代码是脑袋、屁股都吃苦。

下边儿这些，就是每个写 usb 驱动的都需要关心的了，坐这儿看了老半天，struct urb 才露出来这么一个角儿。继续看之前，猜个谜语，放松一下，还记得蜡笔小新里的小白不？那么两只小白是什么？

1136 行，urb_list，还记得每个端点都会有的那个 urb 队列么？那个队列就是由这里的 urb_list 一个一个的链接起来的。HCD 每收到一个 urb，就会将它添加到这个 urb 指定的那个端点的 urb 队列里去。这个链表的头儿在哪儿？当然是在端点里，就是端点里的那个 struct list_head 结构体成员。

1138 行，dev，傻强都知道，它表示的是 urb 要去的那个 usb 设备。

1139 行，pipe，urb 到达端点之前，需要经过一个通往端点的管道，就是这个 pipe。那第一个问题，怎么表示一个 pipe？人生有两极，管道有两端，一端是主机上的缓冲区，一

端是设备上的端点，既然有两端，总要有个方向吧，不然 urb 要待在管道里无所适从的仰天长叹，我从哪里来，又该往哪里去？而且早先说过，端点有四种类型，那么与端点相生相依的管道也应该不只一种吧。这么说来，确定一条管道至少要知道两端的地址、方向和类型了，不过这两端里主机是确定的，需要确定的只是另一端设备的地址和端点的地址。那怎么将这些内容揉合起来表示成一个管道？一个包含了各种成员属性的结构再加上一些操作函数？多么完美的封装，但是不需要这么搞，写代码的哥们儿和俺的人生信条差不多，复杂简单化，一个整型值再加上一些宏就够了。

先看看管道，也就是这个整型值的构成，bit7 用来表示方向，bit8~14 表示设备地址，bit15~18 表示端点号，早先说过，设备地址用 7 位来表示，端点号用 4 位来表示，剩下来的 bit30~31 表示管道类型。再看看围绕管道的一些宏，在 include/linux/usb.h 里定义

```
1388 /*
1389  * For various legacy reasons, Linux has a small cookie that's paired with
1390  * a struct usb_device to identify an endpoint queue.  Queue characteristics
1391  * are defined by the endpoint's descriptor.  This cookie is called a "pipe",
1392  * an unsigned int encoded as:
1393  *
1394  *  - direction:        bit 7           (0 = Host-to-Device [Out],
1395  *                                       1 = Device-to-Host [In] ...
1396  *                                      like endpoint bEndpointAddress)
1397  *  - device address:   bits 8-14       ... bit positions known to uhci-hcd
1398  *  - endpoint:         bits 15-18      ... bit positions known to uhci-hcd
1399  *  - pipe type:        bits 30-31      (00 = isochronous, 01 = interrupt,
1400  *                                       10 = control, 11 = bulk)
1401  *
1402  * Given the device address and endpoint descriptor, pipes are redundant.
1403  */
1404
1405 /* NOTE:  these are not the standard USB_ENDPOINT_XFER_* values!! */
1406 /* (yet ... they're the values used by usbfs) */
1407 #define PIPE_ISOCHRONOUS                0
1408 #define PIPE_INTERRUPT                  1
1409 #define PIPE_CONTROL                    2
1410 #define PIPE_BULK                       3
1411
1412 #define usb_pipein(pipe)        ((pipe) & USB_DIR_IN)
1413 #define usb_pipeout(pipe)       (!usb_pipein(pipe))
1414
1415 #define usb_pipedevice(pipe)    (((pipe) >> 8) & 0x7f)
1416 #define usb_pipeendpoint(pipe)  (((pipe) >> 15) & 0xf)
1417
1418 #define usb_pipetype(pipe)      (((pipe) >> 30) & 3)
```

```
1419 #define usb_pipeisoc(pipe)        (usb_pipetype((pipe)) == PIPE_ISOCHRONOUS)
1420 #define usb_pipeint(pipe)         (usb_pipetype((pipe)) == PIPE_INTERRUPT)
1421 #define usb_pipecontrol(pipe)     (usb_pipetype((pipe)) == PIPE_CONTROL)
1422 #define usb_pipebulk(pipe)        (usb_pipetype((pipe)) == PIPE_BULK)
```

这些宏什么意思，就不用我说了，不知道的都可以去跳浩然高科了。我可是亲眼目睹过这样的惨状，触目惊心啊，在我交大的历史记忆里抹上了浓重的一笔。

现在看第二个问题，如何创建一个管道？主机和设备不是练家子，没练过千里传音什么的绝世神功，要交流必须通过管道，你必须得创建一个管道给 urb，它才知道路怎么走。不过在说怎么创建一个管道前，先说个有关管道的故事，咱们也知道知道管道这么热门儿的词汇是咋来的。

1801 年，意大利中部的小山村，有两个名叫柏波罗和布鲁诺的年轻人，和我现在最大的梦想是看懂内核代码一样，他们的最大梦想是称为村子里最富有的人，有一天，喜鹊在枝头唧唧喳喳的叫，他们的好运也就随着来了。村里决定雇两个人把附近河里的水运到村广场的水缸里去，他们砸对了幸运 52 里的金蛋，得到了这个机会。“我提一桶水，只收他一分钱，我提 10 桶水赚多少钱，恩？老子发点狠!! 我一天提他一百桶水！一百桶水！算一下多少钱先，一七得七……呀，二七四十八……呀，三八……妇女节，五一……劳动节，六一……我爹过节，七一……” 布鲁诺激动的盘算着。但柏波罗却想着一天才几分钱的报酬，还要这样来回提水，干脆修一条管道将水从河里引到村里去得了。于是布鲁诺每天辛勤的提着水，很快买了新衣服，买了驴，虽然仍然买不起车也买不起房，但已经被看作是中产阶级了，而柏波罗每天还要抽出一部分提水的时间去挖管道，收入是入不敷出啊，挖管道的同时还要接收很多人对他管道男的嘲笑。两年后，柏波罗的管道完工了，水哗哗的直往村里流，钱哗哗的直往口袋里钻，而此时布鲁诺已经被长时间的提桶生涯给压榨的腰弯背驼。管道男柏波罗成了奇迹的创造者，他没有被赞扬冲昏头脑，他想的是创建更多的管道，他邀请提桶男布鲁诺加入了他的管道事业，从此他们的管道事业芝麻开花节节高，遍布了全球。

这个故事告诉我们，管道很重要啊同志们。显然，写代码的哥们儿也深刻的认识到了这个道理，于是内核的 include/linux/usb.h 文件里多了很多专门用来创建不同管道的宏。

```
1432 static inline unsigned int __create_pipe(struct usb_device *dev,
1433                 unsigned int endpoint)
1434 {
1435         return (dev->devnum << 8) | (endpoint << 15);
1436 }
1437
1438 /* Create various pipes... */
1439 #define usb_sndctrlpipe(dev,endpoint)    \
1440         ((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint))
1441 #define usb_rcvctrlpipe(dev,endpoint)    \
1442         ((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1443 #define usb_sndisocpipe(dev,endpoint)    \
```

```
1444         ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev,endpoint))
1445 #define usb_rcvisocpipe(dev,endpoint)   \
1446         ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1447 #define usb_sndbulkpipe(dev,endpoint)   \
1448         ((PIPE_BULK << 30) | __create_pipe(dev,endpoint))
1449 #define usb_rcvbulkpipe(dev,endpoint)   \
1450         ((PIPE_BULK << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1451 #define usb_sndintpipe(dev,endpoint)    \
1452         ((PIPE_INTERRUPT << 30) | __create_pipe(dev,endpoint))
1453 #define usb_rcvintpipe(dev,endpoint)    \
1454         ((PIPE_INTERRUPT << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
```

端点是有四种的，对应着管道也就有四种，同时端点是有IN也有OUT的，相应的管道也就有两个方向，于是二四得八，上面就出现了八个创建管道的宏。有了struct usb_device结构体，也就是说知道了设备地址，再加上端点号，你就可以需要什么管道就创建什么管道。__create_pipe宏只是一个幕后的角色，用来将设备地址和端点号放在管道正确的位置上。

1140 行，status，urb 的当前状态。urb 当然是可以有多种状态的，像咱们这种除了三点一线就是三点一线的小角色都被码上了各种各样的状态，何况 usb 通信的顶梁柱 urb。咱们的状态是别人了解自己的窗口，同时也是意味着自己要尽的责任，你已经有老婆了就不要再去追求人家小 mm 了，不然你从未婚状态转到已婚状态只是户口档案里改了一个字？不过如果人家小 mm 乐意那也就只好祝福你了，人的事情就是这么的复杂，还是咱们的 urb 简单，它当前什么状态就是让咱们了解出了什么事情。至于各种具体的状态代表了什么意思，碰到了再说。

1141 行，transfer_flags，一些标记，可用的值都在 include/linux/usb.h 里有定义

```
939 /*
940  * urb->transfer_flags:
941  */
942 #define URB_SHORT_NOT_OK        0x0001  /* report short reads as errors */
943 #define URB_ISO_ASAP            0x0002  /* iso-only, urb->start_frame
944                                          * ignored */
945 #define URB_NO_TRANSFER_DMA_MAP 0x0004  /* urb->transfer_dma valid on submit
*/
946 #define URB_NO_SETUP_DMA_MAP    0x0008  /* urb->setup_dma valid on submit */
947 #define URB_NO_FSBR             0x0020  /* UHCI-specific */
948 #define URB_ZERO_PACKET         0x0040  /* Finish bulk OUT with short packet
*/
949 #define URB_NO_INTERRUPT        0x0080  /* HINT: no non-error interrupt
950                                          * needed */
```

URB_SHORT_NOT_OK，这个标记只对用来从 IN 端点读取数据的 urb 有效，意思就是说

如果从一个 IN 端点那里读取了一个比较短的数据包，就可以认为是错误的。那么这里的 short 究竟 short 到什么程度？

之前说到端点的时候，就知道端点描述符里有一个叫 wMaxPacketSize 这样的东东，指明了端点一次能够处理的最大字节数。然后，在另外某个地方也提了，在 usb 的世界里是有很多种包的，四种 PID 类型，每种 PID 下边儿还有一些细分的品种。这四种 PID 里面，有一个叫 Data 的，也只有它里边儿有个数据字段，像其它的，Token、Handshake 之类的 PID 类型都是没有这个字段的，所以里里外外看过去，还只有 Data PID 类型的包最实在，就是用来传输数据的，但是它里面并不是只有一个数据字段，还有 SYNC、PID、地址域、CRC 等陪伴在数据字段的左右。镜头向前回退了这么多，那现在一个问题出来了，每个端点描述符里的 wMaxPacketSize 所表示的最大字节数都包括了哪些部分？是整个 packet 的长度么？我可以负责任的告诉你，它只包括了 Data 包里面数据字段，俗称 data payload，其它那些七大姑八大姨什么的都是协议本身需要的信息，和 TCP/IP 里的报头差不多。

wMaxPacketSize 与 short 有什么关系？关系还不小，short 不 short 就是与 wMaxPacketSize 相比的，如果从 IN 端点那儿收到了一个比 wMaxPacketSize 要短的包，同时也设置了 URB_SHORT_NOT_OK 这个标志，那么就可以认为传输出错了。本来如果收到一个比较短的包是意味着这次传输到此为止就结束了，你想想 data payload 的长度最大必须为 wMaxPacketSize 这个规定是不可违背的了，但是如果端点想给你的数据不止那么多，怎么办？就需要分成多个 wMaxPacketSize 大小的 data payload 来传输，事情有时不会那么凑巧，刚好能平分成多个整份，这时，最后一个 data payload 的长度就会比 wMaxPacketSize 要小，这种情况本来意味着端点已经传完了它想传的，释放完了自己的需求，这次传输就该结束了，不过如果你设置了 URB_SHORT_NOT_OK 标志，HCD 这边就会认为错误发生了。

URB_ISO_ASAP，这个标志只是为了方便等时传输用的。等时传输和中断传输在 spec 里都被认为是 periodic transfers，也就是周期传输，咱们都知道在 usb 的世界里都是主机占主导地位，设备是没多少发言权的，但是对于等时传输和中断传输，端点可以对主机表达自己一种美好的期望，希望主机能够隔多长时间访问自己一次，这个期望的时间就是这里说的周期。当然，期望与现实是有一段距离的，如果期望的都能成为现实，咱们还研究 usb 干吗，中国足球也不用整天喊冲出亚洲了，阎世铎在国青赴德时放的那句 “中国足球缺的是一块金牌，希望北京奥运会能够实现这个目标！”就够了。端点的这个期望能不能得到满足，要看主机控制器答应不答应。对于等时传输，一般来说也就一帧（微帧）一次，主机那儿也很忙，再多也抽不出空儿来。那么如果你有个用于等时传输的 urb，你提交给 HCD 的时候，就得告诉 HCD 它应该从哪一帧开始的，就要对下面要说的那个 start_frame 赋值，也就是说告诉 HCD 等时传输开始的那一帧（微帧）的帧号，如果你留心，应该还会记得前面说过在每帧或微帧（Mircoframe）的开始都会有个 SOF Token 包，这个包里就含有个帧号字段，记录了那一帧的编号。这样的话，一是比较烦，作为一个男人，烦心的事儿已经够多了，钞票一天比一天难赚，前方怎么也看不到岸，还要去设置这个 start_frame，你说烦不烦，二是到你设置的那一帧的时候，如果主机控制器没空开始等时传输，你说怎么办，

要知道 usb 的世界里它可是老大。于是，就出现了 URB_ISO_ASAP，它的意思就是告诉 HCD 啥时候不忙就啥时候开始，就不用指定什么开始的帧号了，是不是感觉特轻松？所以说，你如果想进行等时传输，又不想标新立异的话，就还是把它给设置了吧。

URB_NO_TRANSFER_DMA_MAP，还有 URB_NO_SETUP_DMA_MAP，这两个标志都是有关 DMA 的，什么是 DMA？就是外设，比如咱们的 usb 摄像头，和内存之间直接进行数据交换，把 CPU 给撇一边儿了，本来，在咱们的电脑里，CPU 自认为是老大，什么事都要去插一脚，都要经过它去协调处理，和地球对面的那个美利坚合众国差不多。可是这样的话就影响了数据传输的速度，就像革命青年上山下乡那会儿，谁对哪个 mm 有意思了，要先向自己的老爸老妈汇报思想动态，说想和哪家姑娘处对象，等着老爸老妈经过高层协商说可以交往了，然后才能和人家姑娘见面，这多慢啊，哪像现在，老爸老妈都不知道那，下一代可能都已经培育出来了，有 DMA 和没有 DMA 区别就是这么大。

usb 的世界里也是要与时俱进，要创建和谐社会的，所以 dma 也是少不了的。一般来说，都是驱动里提供了 kmalloc 等分配的缓冲区，HCD 做一定的 DMA 映射处理，DMA 映射是干吗的？外设和内存之间进行数据交换，总要互相认识吧，难不成是在衡山路的酒吧啊，会有不认识的 mm 过来和你答腔，外设是通过各种总线连到主机里边儿的，使用的是总线地址，而内存使用的是虚拟地址，它们之间本来就是两条互不相交的平行线，要让它们中间产生连接点，必须得将一个地址转化为另一个地址，这样才能找得到对方，才能互通有无，而 DMA 映射就是干这个的。这只是轻描淡写三言两语的粗略说法，实际上即使千言万语也道不完的。它可是高技术含量的活儿，不是看看报纸喝喝茶，开开会泡泡 mm 就能搞定的，所以在某些平台上非常的费时费力，为了分担点 HCD 的压力，于是就有了这里的两个标志，告诉 HCD 不要再自己做 DMA 映射了，驱动提供的 urb 里已经提供有 DMA 缓冲区地址，为领导分忧解难是咱们这些小百姓应该做的事情。具体提供了哪些 DMA 缓冲区？就涉及到下面的 transfer_buffer，transfer_dma，还有 setup_packet，setup_dma 这两对儿了。

URB_NO_FSBR，这是给 UHCI 用的。

URB_ZERO_PACKET，这个标志表示批量的 OUT 传输必须使用一个 short packet 来结束。批量传输的数据大于批量端点的 wMaxPacketSize 时，需要分成多个 Data 包来传输，最后一个 data payload 的长度可能等于 wMaxPacketSize，也可能小于，当等于 wMaxPacketSize 时，如果同时设置了 URB_ZERO_PACKET 标志，就需要再发送一个长度为 0 的数据包来结束这次传输，如果小于 wMaxPacketSize 就没必要多此一举了。你要问，当批量传输的数据小于 wMaxPacketSize 时那？也没必要再发送 0 长的数据包，因为此时发送的这个数据包本身就是一个 short packet。

URB_NO_INTERRUPT，这个标志用来告诉 HCD，在 URB 完成后，不要请求一个硬件中断，当然这就意味着你的结束处理函数可能不会在 urb 完成后立即被调用，而是在之后的某个时间被调用，咱们的 usb core 会保证为每个 urb 调用一次结束处理函数。

1142~1144 行，transfer_buffer，transfer_dma，transfer_buffer_length，前面说过管道的一端是主机上的缓冲区，一端是设备上的端点，这三个家伙就是描述主机上的那个缓冲区的。transfer_buffer 是使用 kmalloc 分配的缓冲区，transfer_dma 是使用 usb_buffer_alloc 分配的 dma 缓冲区，HCD 不会同时使用它们两个，如果你的 urb 自带了 transfer_dma，就要同时设置 URB_NO_TRANSFER_DMA_MAP 来告诉 HCD 一声，不用它再费心做 DMA 映射了。transfer_buffer 是必须要设置的，因为不是所有的主机控制器都能够使用 DMA 的，万一遇到这样的情况，也好有个备用。transfer_buffer_length 指的就是 transfer_buffer 或 transfer_dma 的长度。

1145 行，actual_length，urb 结束之后，会用这个字段告诉你实际上传输了多少数据。

1146~1147 行，setup_packet，setup_dma，同样是两个缓冲区，一个是kmalloc分配的，一个是用usb_buffer_alloc分配的，不过，这两个缓冲区是控制传输专用的，记得struct usb_ctrlrequest不？它们保存的就是一个struct usb_ctrlrequest结构体，如果你的urb设置了setup_dma，同样要设置URB_NO_SETUP_DMA_MAP标志来告诉HCD。如果进行的是控制传输，setup_packet是必须要设置的，也是为了防止出现主机控制器不能使用DMA的情况。

1148 行，start_frame，如果你没有指定 URB_ISO_ASAP 标志，就必须自己设置 start_frame，指定等时传输在哪帧或微帧开始。如果指定了 URB_ISO_ASAP，urb 结束时会使用这个值返回实际的开始帧号。

1150 行，interval，等时和中断传输专用。interval 间隔时间的意思，什么的间隔时间？就是上面说的端点希望主机轮询自己的时间间隔。这个值和端点描述符里的 bInterval 是一样的，你不能随便儿的指定一个，然后就去做春秋大梦，以为到时间了梦里的名车美女都会跑出来，协议里对你能指定的值是有范围限制的，对于中断传输，全速时，这个范围为 1~255ms，低速是为 10~255ms，高速时为 1~16，这个 1~16 只是 bInterval 可以取的值，实际的间隔时间需要计算一下，为 2 的（bInterval-1）次方乘以 125 微妙，也就是 2 的（bInterval-1）次方个微帧。对于等时传输，没有低速了，等时传输根本就不是低速端点负担得起的，有多大能耐就做多大事，人有多大胆地有多大产的时代早就已经过去了，对于全速和高速，这个范围也是为 1~16，间隔时间由 2 的（bInterval-1）次方算出来，单位为帧或微帧。这样看来，每一帧或微帧里，你最多只能期望有一次等时和中断传输，不能再多了，你只能期望房价涨的慢点，要是希望它跌下去，那要求就太过分了，它可是经济的柱子，要是倒了，那不是陷国人的生活于困境么，所以咱们要爱国啊，要送钱给 ZF 还有任小强们啊同志们。

不过即使完全按照上面的范围来取，你的期望也并不是就肯定可以实现的，因为对于高速来说，最多有 80%的总线时间给这两种传输用，对于低速和全速要多点儿，达到 90%，这个时间怎么分配，都由主机控制器掌握着，所以你的期望能不能实现还要看主机控制器的脸色，没办法，它就有这种权力。在咱们这个官本位的历史悠久的民族里，什么最重要？权力，而且它还无处不在，个中滋味大家自有体会了。

1153 行，context，驱动设置了给下面的结束处理函数用的。比如可以将自己驱动里描述自己设备的结构体放在里边儿，在结束处理函数里就可以取出来。

1154 行，complete，一个指向结束处理函数的指针，传输成功完成，或者中间发生错误的时候就会调用它，驱动可以在这里边儿检查 urb 的状态，并做一些处理，比如可以释放这个 urb，或者重新提交给 HCD。就说摄像头吧，你向 HCD 提交了个等时的 urb 从摄像头那里读取视频数据，传输完成的时候调用了你指定的这个结束处理函数，并在里面取出了 urb 里面获得的数据进行解码等处理，然后怎么着？总不会这一个 urb 读取的数据就够你向 mm 表白了吧，你的爱慕之情可是犹如滔滔江水连绵不绝，所以需要获得更多的数据，那你也总不会再去创建、初始化一个等时的 urb 吧，即使再穷极无聊的人也不会那么做，明显刚刚的那个可以继续用的，只要将它再次提交给 HCD 就可以了。这个函数指针的定义在 include/linux/usb.h

```
961 typedef void (*usb_complete_t)(struct urb *);
```

还有三个，都是等时传输专用的，等时传输与其它传输不一样，可以指定传输多少个packet，每个 packet 使用 struct usb_iso_packet_descriptor 结构来描述。1155 行的 iso_frame_desc就表示了一个变长的struct usb_iso_packet_descriptor结构体数组，而 1149 行的number_of_packets指定了要这个结构体数组的大小，也就是要传输多少个 packet。

要说明的是这里说的packet不是说你在一次等时传输里传输了多个Data packet，而是说你在一个urb里指定了多次的等时传输，每个struct usb_iso_packet_descriptor结构体都代表了一次等时传输。这里说一下等时传输底层的packet情况。不像控制传输最少要有SETUP和STATUS两个阶段的transaction，等时传输只有Isochronous transaction，即等时transaction一个阶段，一次等时传输就是一次等时transaction的过程。而等时transaction也只有两个阶段，就是主机向设备发送OUT Token包，然后发送一个Data包，或者是主机向设备发送IN Token包，然后设备向主机发送一个Data包，这个Data包里data payload 的长度只能小于或者等于等时端点的wMaxPacketSize 值。这里没有了Handshake包，因为不需要，等时传输是不保证数据完全正确无误的到达的，没有什么错误重传机制，也就不需要使用Handshake包来汇报OK不OK。对它来说实时要比正确性重要的多，你的摄像头一秒钟少给你一帧多给你一帧，没什么本质的区别，如果给你延迟个几秒，就明显的感觉不爽了。

所以对于等时传输来说，在完成了 number_of_packets 次传输之后，会去调用你的结束处理函数，在里面对数据做处理，而 1152 行的 error_count 记录了这么多次传输中发生错误的次数。

现在看看struct usb_iso_packet_descriptor结构的定义，在include/linux/usb.h里定义

```
952 struct usb_iso_packet_descriptor {
```

```
953         unsigned int offset;
954         unsigned int length;            /* expected length */
955         unsigned int actual_length;
956         int status;
957 };
```

offset表示transfer_buffer里的偏移位置，你不是指定了要进行number_of_packets次等时传输么，那么也要准备够这么多次传输用的缓冲区吧，当然不是说让你准备多个缓冲区，没必要，都放transfer_buffer或者transfer_dma里面好了，只要记着每次传输对应的数据偏移就可以。length是预期的这次等时传输Data包里数据的长度，注意这里说的是预期，因为实际传输时因为种种原因可能不会有那么多数据，urb结束时，每个struct usb_iso_packet_descriptor结构体的actual_length就表示了各次等时传输实际传输的数据长度，而 status分别记录了它们的状态。

不管你理解不理解，struct urb 都是暂且说到这里了，最后听听 David Brownell 大侠的呼声

> I'd rather see 'struct urb' start to shrink, not grow!

设备的生命线（五）

人的一生就象在拉屎，有时你已经很努力了可出来的只是一个屁。

看这内核代码也一个样，已经很努力了，俺的葱葱玉指都磨出茧子了，才勉勉强强把 struct urb 这个中心给说完，下面接着看那三个基本点。看之前，再猜个谜语，蜘蛛侠是什么颜色的？

第一个基本点，usb_alloc_urb函数，创建urb的专用函数，为一个urb申请内存并做初始化，在urb.c里定义。

```
40 /**
41  * usb_alloc_urb - creates a new urb for a USB driver to use
42  * @iso_packets: number of iso packets for this urb
43  * @mem_flags: the type of memory to allocate, see kmalloc() for a list of
44  *      valid options for this.
45  *
46  * Creates an urb for the USB driver to use, initializes a few internal
47  * structures, incrementes the usage counter, and returns a pointer to it.
```

```
48  *
49  * If no memory is available, NULL is returned.
50  *
51  * If the driver want to use this urb for interrupt, control, or bulk
52  * endpoints, pass '' as the number of iso packets.
53  *
54  * The driver must call usb_free_urb() when it is finished with the urb.
55  */
56 struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags)
57 {
58         struct urb *urb;
59
60         urb = kmalloc(sizeof(struct urb) +
61                 iso_packets * sizeof(struct usb_iso_packet_descriptor),
62                 mem_flags);
63         if (!urb) {
64                 err("alloc_urb: kmalloc failed");
65                 return NULL;
66         }
67         usb_init_urb(urb);
68         return urb;
69 }
```

这函数长的很让人提神，是个亲民的角色。它只做了两件事情，拿kmalloc来为urb申请内存，然后调用 usb_init_urb进行初始化。usb_init_urb函数在前面说struct urb中的引用计数的时候已经贴过了，它主要的作用就是初始化urb的引用计数，还用memset顺便把这里给urb申请的内存清零。

没什么说的了么？usb_alloc_urb说：别看我简单，我也是很有内涵的。先看第一个问题，它的第一个参数 iso_packets，表示的是struct urb结构最后那个变长数组iso_frame_desc的元素数目，也就是应该与number_of_packets的值相同，所以对于控制/中断/批量传输，这个参数都应该为 0。这也算是给咱们示范了下变长数组咋个用法，内核里到处都是C的示范工程。

第二个问题是参数mem_flags的类型 gfp_t，早几个版本的内核，这里还是int，当然这里变成 gfp_t是因为 kmalloc参数里的那个标志参数的类型从int变成 gfp_t了，你要用kmalloc来申请内存，就得遵守它的规则。不过这里要说的不是kmalloc，而是gfp_t，它在江湖上也没出现多久，名号还没打出来，很多人不了解，咱们来调查一下它的背景。它在include/linux/types.h里定义

```
193 typedef unsigned __bitwise__ gfp_t;
```

很显然，要了解 gfp_t，关键是要了解__bitwise__，它也在 types.h 里定义

```
170 #ifdef __CHECKER__
171 #define __bitwise__ __attribute__((bitwise))
172 #else
173 #define __bitwise__
174 #endif
```

__bitwise__ 的含义又取决于是否定义了 __CHECKER__，如果没有定义 __CHECKER__，那__bitwise__就啥也不是。哪里定义了__CHECKER__？穿别人的鞋，走自己的路，让他们去找吧，咱们不找，因为内核代码里就没有哪个地方定义了 __CHECKER__，它是有关 Sparse 工具的，内核编译时的参数决定了是不是使用 Sparse 工具来做类型检查。那 Sparse 又是什么？它是一种静态分析工具(static analysis tool)，用于在 linux 内核源代码中发现各种类型的漏洞，一直都是比较神秘的角色，最初由 Linus Torvalds 写的，后来 linus 没有继续维护，直到去年的冬天，它才又有了新的维护者 Josh Triplett。有关 Sparse 再多的东东，咱们还是各自去研究吧，这里不多说了。

可能还会有第三个问题，usb_alloc_urb也没做多少事啊，它做的那些咱们自己很容易就能做了，为什么还说驱动里一定要使用它来创建urb那？按照C++的说法，它就是urb的构造函数，构造函数是创建对象的唯一方式，你抬杠说C++里面儿使用位拷贝去复制一个简单对象给新对象就没使用构造函数，那是你不知道，C++的ARM里将这时的构造函数称为 trivial copy constructor。再说，现在它做这些事儿，以后还是做这些么？它将创建urb的工作给包装了，咱们只管调用就是了，孙子兵法里有，以不变应万变。

对应的，当然还会有个析构函数，销毁 urb 的，也在 urb.c 里定义

```
71 /**
72  * usb_free_urb - frees the memory used by a urb when all users of it are finished
73  * @urb: pointer to the urb to free, may be NULL
74  *
75  * Must be called when a user of a urb is finished with it.  When the last user
76  * of the urb calls this function, the memory of the urb is freed.
77  *
78  * Note: The transfer buffer associated with the urb is not freed, that must be
79  * done elsewhere.
80  */
81 void usb_free_urb(struct urb *urb)
82 {
83         if (urb)
84                 kref_put(&urb->kref, urb_destroy);
85 }
```

usb_free_urb更潇洒，只调用 kref_put将urb的引用计数减一，减了之后如果变为 0，也就是没人再用它了，就调用 urb_destroy将它销毁掉。

接着看第二个基本点，usb_fill_control_urb函数，初始化刚才创建的控制urb，你要想使用urb进行usb传输，不是光为它申请点内存就够的，你得为它初始化，充实点实实在在的内容，这个和女星要出名快，也要填充点内容的道理是一样的。它是在include/linux/usb.h里定义的内联函数

```
1161 /**
1162  * usb_fill_control_urb - initializes a control urb
1163  * @urb: pointer to the urb to initialize.
1164  * @dev: pointer to the struct usb_device for this urb.
1165  * @pipe: the endpoint pipe
1166  * @setup_packet: pointer to the setup_packet buffer
1167  * @transfer_buffer: pointer to the transfer buffer
1168  * @buffer_length: length of the transfer buffer
1169  * @complete_fn: pointer to the usb_complete_t function
1170  * @context: what to set the urb context to.
1171  *
1172  * Initializes a control urb with the proper information needed to submit
1173  * it to a device.
1174  */
1175 static inline void usb_fill_control_urb (struct urb *urb,
1176                                          struct usb_device *dev,
1177                                          unsigned int pipe,
1178                                          unsigned char *setup_packet,
1179                                          void *transfer_buffer,
1180                                          int buffer_length,
1181                                          usb_complete_t complete_fn,
1182                                          void *context)
1183 {
1184         spin_lock_init(&urb->lock);
1185         urb->dev = dev;
1186         urb->pipe = pipe;
1187         urb->setup_packet = setup_packet;
1188         urb->transfer_buffer = transfer_buffer;
1189         urb->transfer_buffer_length = buffer_length;
1190         urb->complete = complete_fn;
1191         urb->context = context;
1192 }
```

这个函数长的就让人兴奋，纯粹是来增长咱们自信的，自信多一分，成功就多十分，你就能搞懂内核，你就能成为一个成功的男人。这个函数基本上都是赋值语句，把你在参数里指定的值充实给刚刚创建的 urb，urb 的元素有很多，这里只是填充了一部分，剩下那些不是控制传输管不着的，就是自有安排可以不用去管的。

你想想，一个struct urb结构要应付四种传输类型，每种传输类型总会有点自己特别的要求，

总会有些元素专属于某种传输类型，而其它传输类型不用管的。如果按C++的做法，这称不上是一个好的设计思想，应该有个基类urb，里面放点儿四种传输类型公用的，比如pipe，transfer_buffer等，再搞几个子类，control_urb，bulk_urb等等，专门应付具体的传输类型，如果不用什么虚函数，实际的时间空间消耗也不会增加什么。但是实在没必要这么搞，这年头儿内核的结构已经够多了，你创建什么类型的urb，就填充相关的一些字段好了，况且写usb core的哥们儿已经给咱们提供了不同传输类型的初始化函数，就像上面的usb_fill_control_urb，对于批量传输有usb_fill_bulk_urb，对于中断传输有usb_fill_int_urb，一般来说这也就够了，下面就看看 usb_fill_control_urb函数的这俩孪生兄弟。

```
1194 /**
1195  * usb_fill_bulk_urb - macro to help initialize a bulk urb
1196  * @urb: pointer to the urb to initialize.
1197  * @dev: pointer to the struct usb_device for this urb.
1198  * @pipe: the endpoint pipe
1199  * @transfer_buffer: pointer to the transfer buffer
1200  * @buffer_length: length of the transfer buffer
1201  * @complete_fn: pointer to the usb_complete_t function
1202  * @context: what to set the urb context to.
1203  *
1204  * Initializes a bulk urb with the proper information needed to submit it
1205  * to a device.
1206  */
1207 static inline void usb_fill_bulk_urb (struct urb *urb,
1208                                       struct usb_device *dev,
1209                                       unsigned int pipe,
1210                                       void *transfer_buffer,
1211                                       int buffer_length,
1212                                       usb_complete_t complete_fn,
1213                                       void *context)
1214 {
1215         spin_lock_init(&urb->lock);
1216         urb->dev = dev;
1217         urb->pipe = pipe;
1218         urb->transfer_buffer = transfer_buffer;
1219         urb->transfer_buffer_length = buffer_length;
1220         urb->complete = complete_fn;
1221         urb->context = context;
1222 }
1223
1224 /**
1225  * usb_fill_int_urb - macro to help initialize a interrupt urb
1226  * @urb: pointer to the urb to initialize.
1227  * @dev: pointer to the struct usb_device for this urb.
```

```
1228  * @pipe: the endpoint pipe
1229  * @transfer_buffer: pointer to the transfer buffer
1230  * @buffer_length: length of the transfer buffer
1231  * @complete_fn: pointer to the usb_complete_t function
1232  * @context: what to set the urb context to.
1233  * @interval: what to set the urb interval to, encoded like
1234  *      the endpoint descriptor's bInterval value.
1235  *
1236  * Initializes a interrupt urb with the proper information needed to submit
1237  * it to a device.
1238  * Note that high speed interrupt endpoints use a logarithmic encoding of
1239  * the endpoint interval, and express polling intervals in microframes
1240  * (eight per millisecond) rather than in frames (one per millisecond).
1241  */
1242 static inline void usb_fill_int_urb (struct urb *urb,
1243                                      struct usb_device *dev,
1244                                      unsigned int pipe,
1245                                      void *transfer_buffer,
1246                                      int buffer_length,
1247                                      usb_complete_t complete_fn,
1248                                      void *context,
1249                                      int interval)
1250 {
1251         spin_lock_init(&urb->lock);
1252         urb->dev = dev;
1253         urb->pipe = pipe;
1254         urb->transfer_buffer = transfer_buffer;
1255         urb->transfer_buffer_length = buffer_length;
1256         urb->complete = complete_fn;
1257         urb->context = context;
1258         if (dev->speed == USB_SPEED_HIGH)
1259                 urb->interval = 1 << (interval - 1);
1260         else
1261                 urb->interval = interval;
1262         urb->start_frame = -1;
1263 }
```

负责批量传输的usb_fill_bulk_urb和负责控制传输的 usb_fill_control_urb的相比，只是少初始化了一个 setup_packet，因为批量传输里没有Setup包的概念，中断传输里也没有，所以usb_fill_int_urb里也没有初始化 setup_packet这一说。不过usb_fill_int_urb比那两个还是多了点儿内容的，因为它有个 interval，比控制传输和批量传输多了个表达自己期望的权利，1258 行还做了次判断，如果是高速就怎么怎么着，否则又怎么怎么着，主要是高速和低速/全速的间隔时间单位不一样，低速/全速的单位为帧，高速的单位为微帧，还要经过 2 的（bInterval-1）次方这么算一下。至于 1262 行start_frame，它是给等时传输

用的，这里自然就设置为-1，关于为什么既然start_frame是等时传输用的这里还要设置那么一下，你往后看吧，现在我也不知道。

作为一个共产主义接班人，我们很快就能发现 usb_fill_control_urb的孪生兄弟里，少了等时传输对应的那个初始化函数，三缺一啊，在哪里都会是个遗憾。不是不想有，而是没办法，对于等时传输，urb里是可以指定进行多次传输的，你必须一个一个的对那个变长数组iso_frame_desc里的内容进行初始化，没人帮得了你。难道你能想出一个办法搞出一个适用于各种情况等时传输的初始化函数？我是不能。如果想不出来，使用urb进行等时传输的时候，还是老老实实的对里面相关的字段一个一个的填内容吧。如果想找个例子旁观一下别人是咋初始化的，可以去找个摄像头驱动，或者其它usb音视频设备的驱动看看，内核里也有一些的。

现在，你应该还记得咱们是因为要设置设备的地址，让设备进入Address状态，调用了usb_control_msg，才走到这里遇到 usb_fill_control_urb的，参数里的 setup_packet就是之前创建和赋好值的struct usb_ctrlrequest结构体，设备的地址已经在struct usb_ctrlrequest结构体wValue字段里了，这次控制传输并没有DATA transaction阶段，也并不需要urb提供什么transfer_buffer缓冲区，所以transfer_buffer应该传递一个NULL，当然transfer_buffer_length也就为 0 了，有意思的是这时候传递进来的结束处理函数usb_api_blocking_completion，可以看一下当这次控制传输已经完成，设备地址已经设置好后，接着做了些什么，它的定义在message.c里

```
21 static void usb_api_blocking_completion(struct urb *urb)
22 {
23         complete((struct completion *)urb->context);
24 }
```

这个函数更简洁，就那么一句，没有前面说的释放 urb，也没有重新提交它，本来就想设置个地址就完事儿了，没必要再将它提交给 HCD，你就是再提交多少次，设置多少次，也只能有一个地址，usb 的世界里不提倡囤积居奇，不鼓励一人多个 Address 去炒。那在这仅仅一句里面都做了些什么？你接着往下看。

然后就是第三个基本点，usb_start_wait_urb函数，将前面历经千辛万苦创建和初始化的urb提交给咱们的usb core，让它移交给特定的主机控制器驱动进行处理，然后望眼欲穿的等待HCD回馈的结果，如果等待的时间超过了预期的限度，它不会再等，不会去变成望夫石。它在message.c里定义

```
27 /*
28  * Starts urb and waits for completion or timeout. Note that this call
29  * is NOT interruptible. Many device driver i/o requests should be
30  * interruptible and therefore these drivers should implement their
31  * own interruptible routines.
32  */
33 static int usb_start_wait_urb(struct urb *urb, int timeout, int *actual_length)
```

```
34 {
35         struct completion done;
36         unsigned long expire;
37         int status;
38
39         init_completion(&done);
40         urb->context = &done;
41         urb->actual_length = 0;
42         status = usb_submit_urb(urb, GFP_NOIO);
43         if (unlikely(status))
44                 goto out;
45
46         expire = timeout ? msecs_to_jiffies(timeout) : MAX_SCHEDULE_TIMEOUT;
47         if (!wait_for_completion_timeout(&done, expire)) {
48
49                 dev_dbg(&urb->dev->dev,
50                         "%s timed out on ep%d%s len=%d/%d\n",
51                         current->comm,
52                         usb_pipeendpoint(urb->pipe),
53                         usb_pipein(urb->pipe) ? "in" : "out",
54                         urb->actual_length,
55                         urb->transfer_buffer_length);
56
57                 usb_kill_urb(urb);
58                 status = urb->status == -ENOENT ? -ETIMEDOUT : urb->status;
59         } else
60                 status = urb->status;
61 out:
62         if (actual_length)
63                 *actual_length = urb->actual_length;
64
65         usb_free_urb(urb);
66         return status;
67 }
```

35 行，定义了一个struct completion结构体。completion是内核里一个比较简单的同步机制，一个线程可以通过它来通知另外一个线程某件事情已经做完了。你使用某个下载软件去下载A片，然后撇一边儿不管就去忙着聊QQ泡mm了，下载完了那个软件会通知你，然后你想怎么做就怎么做，自个看也成，不怕被扁和正在聊的mm一块看也成，没人会去管你。怎么？你的那个软件下载完了也没通知你？那就紧赶的换个别的吧，写那个软件的也太没职业道德了，该做的事情不做。completion机制也同样是这么回事儿，你的代码执行到某个地方，需要再忙点儿其它的，就新开个线程，让它去忙活，然后自个接着忙自己的，想知道那边儿忙活的结果了，就停在某个地方等着，那边儿忙活完了会通知一下已经有结果了，于

是你的代码又可以继续往下走。

completion机制围绕struct completion结构去实现，有两种使用方式，一种是通过DECLARE_COMPLETION宏在编译时就创建好struct completion的结构体，另外一种就是上面的形式，运行时才创建的，先在 35 行定义一个struct completion结构体，然后在 39 行使用 init_completion去初始化。光是创建struct completion的结构体没用，关键的是如何通知任务已经完成了，和怎么去等候完成的好消息。片子下载完了可能会用声音、对话框等多种方式来通知你，同样这里用来通知已经完成了的函数也不只一个，

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

complete 只通知一个等候的线程，complete_all 可以通知所有等候的线程，大家都一个宿舍的好兄弟，你总不好意思自己藏着好东西，不让大家欣赏吧，所以可能会有多个人来等着片子下完。

你不可能会毫无限度的等下去，21 世纪最缺的是什么？耐心，凡事都有个限度，即使片子再精彩，多会儿下不完也不等它了，当然会有比我还穷极无聊的人愿意一直在那里等着，毕竟林子大了什么鸟儿都有，或者说正等着那，一个 ppmm 过来打断你，你赶着花前月下去了，也不会去继续等了。所以针对不同的情况，等候的方式就有好几种，都在 kernel/sched.c 里定义

```
void wait_for_completion(struct completion *c);
unsigned long wait_for_completion_timeout(struct completion *x, unsigned long timeout);
int wait_for_completion_interruptible(struct completion *x);
unsigned long wait_for_completion_interruptible_timeout(struct completion *x, unsigned long timeout);
```

上面 47 行使用的就是wait_for_completion_timeout，设定一个时间限度，然后在那里候着，直到得到通知，或者超过时间。既然有等的一方，也总得有通知的一方吧，不然岂不是每次都超时？写代码的哥们儿没这么变态，记得上面刚出现过的那个结束处理函数usb_api_blocking_completion不？不是吧，被窝都热乎着就不认人了啊，它里面唯一的一句 complete((struct completion *)urb->context)就是用来通知这里的 47 行的。有疑问的话看 40 行，将刚刚初始过的struct completion结构体done的地址赋值给了urb->context，47 行等的就是这个done。再看 42 行，usb_submit_urb函数将这个控制urb提交给usb core，它是异步的，也就是说提交了之后不会等到传输完成了才返回。

现在就比较清晰了，usb_start_wait_urb函数将urb提交给usb core去处理后，就停在 47 行等候usb core和HCD的处理结果，而这个urb代表的控制传输完成之后会调用结束处理函数 usb_api_blocking_completion，从而调用 complete 来通知 usb_start_wait_urb 说不用再等了，传输已经完成了，当然还有种可能是usb_start_wait_urb已经等的超过了时间限度仍然没有接到通知，不管是哪种情况，

usb_start_wait_urb都可以不用再等，继续往下走了。

下边儿挨个说一下，42 行，提交 urb，并返回一个值表示是否提交成功了，显然，成功的可能性要远远大于失败的可能性，不然就是或者写代码的哥们儿有问题，或者就是我有问题，所以接下来的判断加上了 unlikely，如果你真的那么衰，遇上了小概率事件，那也就没必要在 47 行等通知了，直接到后边儿去吧。

46 行，计算超时值。超时值在参数里不是已经给了么，还计算什么？没错，你是在参数里是指定了自己能够忍受的最大时间限度，不过那是以 ms 为单位的，作为一个平头小百姓，咱们的时间概念里也只有分钟啊妙啊毫秒啊什么的，不过作为一个要在 linux 里混的平头小百姓，咱们的时间概念里必须得加上一个号称 jiffy 的东东，因为函数 wait_for_completion_timeout 里的超时参数是必须以 jiffy 为单位的。

jiffy，金山词霸告诉我们，是瞬间，短暂的时间跨度，短暂到什么程度？linux 里它表示的是两次时钟中断的间隔，时钟中断是由定时器周期性产生的，关于这个周期，内核里有个巨直白巨形象的变量来描述，就是 HZ，它是个体系结构相关的常数。内核里还提供了专门的计数器来记录从系统引导开始所度过的 jiffy 值，每次时钟中断发生时，这个计数器就增加 1。

既然你指定的时间和 wait_for_completion_timeout 需要的时间单位不一致，就需要转换一下，msecs_to_jiffies 函数可以完成这个工作，它将 ms 值转化为相应的 jiffy 值。这一行里还剩个 MAX_SCHEDULE_TIMEOUT 比较陌生，在 include/linux/sched.h 里它被定义为 LONG_MAX，最大的长整型值，我知道你会好奇这个 LONG_MAX 是怎么来的，好奇就说出来嘛，好奇又不会真的害死猫，我也很好奇，所以咱们到生它养它的 include/linux/kernel.h 里看看

```
23 #define INT_MAX         ((int)(~0U>>1))
24 #define INT_MIN         (-INT_MAX - 1)
25 #define UINT_MAX        (~0U)
26 #define LONG_MAX        ((long)(~0UL>>1))
27 #define LONG_MIN        (-LONG_MAX - 1)
28 #define ULONG_MAX       (~0UL)
29 #define LLONG_MAX       ((long long)(~0ULL>>1))
30 #define LLONG_MIN       (-LLONG_MAX - 1)
31 #define ULLONG_MAX      (~0ULL)
```

各种整型数的最大值最小值都在这里了，俺现在替谭浩强再普及点基础知识，‘~’是按位取反，‘UL’是无符号长整型，那么‘ULL’就是 64 位的无符号长整型，‘<<’左移运算的话就是直接一股脑的把所有位往左边儿移若干位，‘>>’右移运算比较容易搞混，主要是牵涉到怎么去补缺，有关空缺儿的事情在哪里都会比较的复杂，勾心斗角阶级斗争的根源，在 C 里主要就是无符号整数和有符号整数的之间的冲突，在你补管三七二十一一直往右移多少位之后，空出来的那些空缺，对于无符号整数得补 0，对于有符号的，得补上符号位。

还是拿 LONG_MAX 来说事儿，上边定义为((long)(~0UL>>1))，0UL 按位取反之后全为 1 的无符号长整型，向右移 1 位，左边儿空出来的那个补 0，这个数对于无符号的 long 来说不算什么，但是再经过 long 这么强制转化一下变为有符号的长整型，它就是老大了。每个老大的成长过程都是一部血泪史，都要历经很多曲折。

现在你可以很明白的知道写代码的哥们儿在 46 行都做了些什么，你指定的超时时间被转化为相应的 jiffy 值，或者直接被设定为最大的 long 值。

47 行，等待通知，我们需要知道的是怎么去判断等待的结果，也就是 wait_for_completion_timeout 的返回值代表什么意思？一般来说，一个函数返回了 0 代表了好消息，一切顺利，如果你这么想那可就错了。wait_for_completion_timeout 返回 0 恰恰表示的是坏消息，表示直到超过了自己的忍耐的极限仍没有接到任何的回馈，而返回了一个大于 0 的值则表示接到通知了，那边儿不管是完成了还是出错了总归是告诉这边儿不用再等了，这个值具体的含义就是距你设定的时限提前了多少时间。为什么会这样？你去看看 wait_for_completion_timeout 的定义就知道了，我就不贴了，它是通过 schedule_timeout 来实现超时的，schedule_timeout 的返回值就是这么个意思。

那么现在就很明显了，如果超时了，就打印一些调试信息提醒一下，然后调用 usb_kill_urb 终止这个 urb，再将返回值设定一下。如果收到了通知，就简单了，直接设定了返回值，就接着往下走。

62 行，actual_length 是用来记录实际传输的数据长度的，是上头儿的上头儿 usb_control_msg 需要的。不要给我说这个值 urb 里本来就有保存，何必再多次一举找个地儿去放，没看接下来的 65 行就用 usb_free_urb 把这个 urb 给销毁了啊，到那时花非花树非树，urb 也已经不是 urb，哪里还去找这个值。actual_length 是从上头儿那里传递过来的一个指针，写内核的哥们儿教导我们，遇到指针一定要小心再小心啊，同志们。所以这里要先判断一下 actual_length 是不是空的。

现在，只剩一个 usb_submit_urb 在刚才被有意无意的飘过了，咱们下面说。

设备的生命线（六）

等俺变拽了，手表买两块，左一块右一块，汽车买两辆，开一辆拖一辆

等俺变拽了，宝马买两辆，一辆开道一辆护架，俺在中间骑自行车

等俺变拽了，上市公司开两家,一家挤垮另一家

等俺变拽了，航空母舰造两艘，一艘打沉另一艘；原子弹造两颗，一颗引爆另一颗

等俺变拽了，通信公司开两家，一家叫不在服务区，一家叫暂时无法接通，我让你不在服务区你就不在服务区，我让你暂时无法接通你就暂时无法接通

等俺变拽了，变态函数写两个，一个让系统崩溃，一个让你崩溃

俺现在离拽还差两个筋斗云，所以只有 usb_submit_urb 函数让我给崩溃的份儿，现在就看看这个几百行的函数。

```
107 /**
108  * usb_submit_urb - issue an asynchronous transfer request for an endpoint
109  * @urb: pointer to the urb describing the request
110  * @mem_flags: the type of memory to allocate, see kmalloc() for a list
111  *      of valid options for this.
112  *
113  * This submits a transfer request, and transfers control of the URB
114  * describing that request to the USB subsystem.  Request completion will
115  * be indicated later, asynchronously, by calling the completion handler.
116  * The three types of completion are success, error, and unlink
117  * (a software-induced fault, also called "request cancellation").
118  *
119  * URBs may be submitted in interrupt context.
120  *
121  * The caller must have correctly initialized the URB before submitting
122  * it.  Functions such as usb_fill_bulk_urb() and usb_fill_control_urb() are
123  * available to ensure that most fields are correctly initialized, for
124  * the particular kind of transfer, although they will not initialize
125  * any transfer flags.
126  *
127  * Successful submissions return 0; otherwise this routine returns a
128  * negative error number.  If the submission is successful, the complete()
129  * callback from the URB will be called exactly once, when the USB core and
130  * Host Controller Driver (HCD) are finished with the URB.  When the completion
131  * function is called, control of the URB is returned to the device
132  * driver which issued the request.  The completion handler may then
133  * immediately free or reuse that URB.
134  *
135  * With few exceptions, USB device drivers should never access URB fields
136  * provided by usbcore or the HCD until its complete() is called.
137  * The exceptions relate to periodic transfer scheduling.  For both
138  * interrupt and isochronous urbs, as part of successful URB submission
139  * urb->interval is modified to reflect the actual transfer period used
140  * (normally some power of two units).  And for isochronous urbs,
141  * urb->start_frame is modified to reflect when the URB's transfers were
```

```
142  * scheduled to start.  Not all isochronous transfer scheduling policies
143  * will work, but most host controller drivers should easily handle ISO
144  * queues going from now until 10-200 msec into the future.
145  *
146  * For control endpoints, the synchronous usb_control_msg() call is
147  * often used (in non-interrupt context) instead of this call.
148  * That is often used through convenience wrappers, for the requests
149  * that are standardized in the USB 2.0 specification.  For bulk
150  * endpoints, a synchronous usb_bulk_msg() call is available.
151  *
152  * Request Queuing:
153  *
154  * URBs may be submitted to endpoints before previous ones complete, to
155  * minimize the impact of interrupt latencies and system overhead on data
156  * throughput.  With that queuing policy, an endpoint's queue would never
157  * be empty.  This is required for continuous isochronous data streams,
158  * and may also be required for some kinds of interrupt transfers. Such
159  * queuing also maximizes bandwidth utilization by letting USB controllers
160  * start work on later requests before driver software has finished the
161  * completion processing for earlier (successful) requests.
162  *
163  * As of Linux 2.6, all USB endpoint transfer queues support depths greater
164  * than one.  This was previously a HCD-specific behavior, except for ISO
165  * transfers.  Non-isochronous endpoint queues are inactive during cleanup
166  * after faults (transfer errors or cancellation).
167  *
168  * Reserved Bandwidth Transfers:
169  *
170  * Periodic transfers (interrupt or isochronous) are performed repeatedly,
171  * using the interval specified in the urb.  Submitting the first urb to
172  * the endpoint reserves the bandwidth necessary to make those transfers.
173  * If the USB subsystem can't allocate sufficient bandwidth to perform
174  * the periodic request, submitting such a periodic request should fail.
175  *
176  * Device drivers must explicitly request that repetition, by ensuring that
177  * some URB is always on the endpoint's queue (except possibly for short
178  * periods during completion callacks).  When there is no longer an urb
179  * queued, the endpoint's bandwidth reservation is canceled.  This means
180  * drivers can use their completion handlers to ensure they keep bandwidth
181  * they need, by reinitializing and resubmitting the just-completed urb
182  * until the driver longer needs that periodic bandwidth.
183  *
184  * Memory Flags:
185  *
```

```
186  * The general rules for how to decide which mem_flags to use
187  * are the same as for kmalloc.  There are four
188  * different possible values; GFP_KERNEL, GFP_NOFS, GFP_NOIO and
189  * GFP_ATOMIC.
190  *
191  * GFP_NOFS is not ever used, as it has not been implemented yet.
192  *
193  * GFP_ATOMIC is used when
194  *   (a) you are inside a completion handler, an interrupt, bottom half,
195  *       tasklet or timer, or
196  *   (b) you are holding a spinlock or rwlock (does not apply to
197  *       semaphores), or
198  *   (c) current->state != TASK_RUNNING, this is the case only after
199  *       you've changed it.
200  *
201  * GFP_NOIO is used in the block io path and error handling of storage
202  * devices.
203  *
204  * All other situations use GFP_KERNEL.
205  *
206  * Some more specific rules for mem_flags can be inferred, such as
207  *  (1) start_xmit, timeout, and receive methods of network drivers must
208  *      use GFP_ATOMIC (they are called with a spinlock held);
209  *  (2) queuecommand methods of scsi drivers must use GFP_ATOMIC (also
210  *      called with a spinlock held);
211  *  (3) If you use a kernel thread with a network driver you must use
212  *      GFP_NOIO, unless (b) or (c) apply;
213  *  (4) after you have done a down() you can use GFP_KERNEL, unless (b) or (c)
214  *      apply or your are in a storage driver's block io path;
215  *  (5) USB probe and disconnect can use GFP_KERNEL unless (b) or (c) apply;
and
216  *  (6) changing firmware on a running storage or net device uses
217  *      GFP_NOIO, unless b) or c) apply
218  *
219  */
220 int usb_submit_urb(struct urb *urb, gfp_t mem_flags)
221 {
222         int                     pipe, temp, max;
223         struct usb_device       *dev;
224         int                     is_out;
225
226         if (!urb || urb->hcpriv || !urb->complete)
227                 return -EINVAL;
228         if (!(dev = urb->dev) ||
```

```
229             (dev->state < USB_STATE_DEFAULT) ||
230             (!dev->bus) || (dev->devnum <= 0))
231                 return -ENODEV;
232         if (dev->bus->controller->power.power_state.event != PM_EVENT_ON
233                         || dev->state == USB_STATE_SUSPENDED)
234                 return -EHOSTUNREACH;
235
236         urb->status = -EINPROGRESS;
237         urb->actual_length = 0;
238
239         /* Lots of sanity checks, so HCDs can rely on clean data
240          * and don't need to duplicate tests
241          */
242         pipe = urb->pipe;
243         temp = usb_pipetype(pipe);
244         is_out = usb_pipeout(pipe);
245
246         if (!usb_pipecontrol(pipe) && dev->state < USB_STATE_CONFIGURED)
247                 return -ENODEV;
248
249         /* FIXME there should be a sharable lock protecting us against
250          * config/altsetting changes and disconnects, kicking in here.
251          * (here == before maxpacket, and eventually endpoint type,
252          * checks get made.)
253          */
254
255         max = usb_maxpacket(dev, pipe, is_out);
256         if (max <= 0) {
257                 dev_dbg(&dev->dev,
258                         "bogus endpoint ep%d%s in %s (bad maxpacket %d)\n",
259                         usb_pipeendpoint(pipe), is_out ? "out" : "in",
260                         __FUNCTION__, max);
261                 return -EMSGSIZE;
262         }
263
264         /* periodic transfers limit size per frame/uframe,
265          * but drivers only control those sizes for ISO.
266          * while we're checking, initialize return status.
267          */
268         if (temp == PIPE_ISOCHRONOUS) {
269                 int     n, len;
270
271                 /* "high bandwidth" mode, 1-3 packets/uframe? */
272                 if (dev->speed == USB_SPEED_HIGH) {
```

```
273                         int     mult = 1 + ((max >> 11) & 0x03);
274                         max &= 0x07ff;
275                         max *= mult;
276                 }
277
278                 if (urb->number_of_packets <= 0)
279                         return -EINVAL;
280                 for (n = 0; n < urb->number_of_packets; n++) {
281                         len = urb->iso_frame_desc[n].length;
282                         if (len < 0 || len > max)
283                                 return -EMSGSIZE;
284                         urb->iso_frame_desc[n].status = -EXDEV;
285                         urb->iso_frame_desc[n].actual_length = 0;
286                 }
287         }
288
289         /* the I/O buffer must be mapped/unmapped, except when length=0 */
290         if (urb->transfer_buffer_length < 0)
291                 return -EMSGSIZE;
292
293 #ifdef DEBUG
294         /* stuff that drivers shouldn't do, but which shouldn't
295          * cause problems in HCDs if they get it wrong.
296          */
297         {
298         unsigned int    orig_flags = urb->transfer_flags;
299         unsigned int    allowed;
300
301         /* enforce simple/standard policy */
302         allowed = (URB_NO_TRANSFER_DMA_MAP | URB_NO_SETUP_DMA_MAP |
303                         URB_NO_INTERRUPT);
304         switch (temp) {
305         case PIPE_BULK:
306                 if (is_out)
307                         allowed |= URB_ZERO_PACKET;
308                 /* FALLTHROUGH */
309         case PIPE_CONTROL:
310                 allowed |= URB_NO_FSBR; /* only affects UHCI */
311                 /* FALLTHROUGH */
312         default:                        /* all non-iso endpoints */
313                 if (!is_out)
314                         allowed |= URB_SHORT_NOT_OK;
315                 break;
316         case PIPE_ISOCHRONOUS:
```

```
317                 allowed |= URB_ISO_ASAP;
318                 break;
319         }
320         urb->transfer_flags &= allowed;
321
322         /* fail if submitter gave bogus flags */
323         if (urb->transfer_flags != orig_flags) {
324                 err("BOGUS urb flags, %x --> %x",
325                         orig_flags, urb->transfer_flags);
326                 return -EINVAL;
327         }
328         }
329 #endif
330         /*
331          * Force periodic transfer intervals to be legal values that are
332          * a power of two (so HCDs don't need to).
333          *
334          * FIXME want bus->{intr,iso}_sched_horizon values here.  Each HC
335          * supports different values... this uses EHCI/UHCI defaults (and
336          * EHCI can use smaller non-default values).
337          */
338         switch (temp) {
339         case PIPE_ISOCHRONOUS:
340         case PIPE_INTERRUPT:
341                 /* too small? */
342                 if (urb->interval <= 0)
343                         return -EINVAL;
344                 /* too big? */
345                 switch (dev->speed) {
346                 case USB_SPEED_HIGH:    /* units are microframes */
347                         // NOTE usb handles 2^15
348                         if (urb->interval > (1024 * 8))
349                                 urb->interval = 1024 * 8;
350                         temp = 1024 * 8;
351                         break;
352                 case USB_SPEED_FULL:    /* units are frames/msec */
353                 case USB_SPEED_LOW:
354                         if (temp == PIPE_INTERRUPT) {
355                                 if (urb->interval > 255)
356                                         return -EINVAL;
357                                 // NOTE ohci only handles up to 32
358                                 temp = 128;
359                         } else {
360                                 if (urb->interval > 1024)
```

```
361                                     urb->interval = 1024;
362                             // NOTE usb and ohci handle up to 2^15
363                             temp = 1024;
364                     }
365                     break;
366             default:
367                     return -EINVAL;
368             }
369             /* power of two? */
370             while (temp > urb->interval)
371                     temp >>= 1;
372             urb->interval = temp;
373     }
374
375     return usb_hcd_submit_urb(urb, mem_flags);
376 }
```

看到这个函数之后我深刻的体会到，世界上只有一个地方有真乐：乐土。不过这个函数长归长，目标却很简单，就是对 urb 做些前期处理后扔给 HCD。

226 行，一些有关存在性的判断，某主义的哲学告诉我们：存在是检验真理的唯一标准。所以这个函数的开头儿就要履行一下常规的检验，urb 为空，都没有初始化是不可以提交给 core 的，core 很生气，后果很严重，hcpriv 本来说好了留给 HCD 用的，你得保证送过去的时候它还是贞洁的，自己不能偷偷先用了，HCD 很生气，后果也会很严重，complete，每个 urb 结束了都必须的调用一次 complete 代表的函数，这是真理，你必须得让它存在。

228 行，上边儿是对 urb 本身的检验，这里是对 urb 的目的地 usb 设备的检验。设备所属于的那条总线不存在，或者设备本身不存在，你 urb 还嚷嚷着要过去要过去，也太苏维埃了吧？或者设备甚至还没进入 USB_STATE_DEFAULT 状态，管道的另一端还都是堵着的怎么过去，早先强调多少回了，要想让设备回应你，它起码得达到 Default 状态。设备编号 devnum 肯定是不能为负的了，那为 0 为什么也不行那？到现在，地球人都知道了，Token 包的地址域里有 7 位是表示设备地址的，也就是说总共可以有 128 个地址来分配给设备，但是其中 0 号地址是被保留作为缺省地址用的，任何一个设备处于 Default 状态还没有进入 Address 状态的时候都需要通过这个缺省地址来响应主机的请求，所以 0 号地址不能分配给任何一个设备，hub 为设备选择一个地址的时候，只有选择到一个大于 0 的地址，设备的生命线才会继续，才会走到这里，因此说这里的 devnum 是不可能也不应该为 0 的，如果为 0 的话，那就是中间哪里谁暗地里动了手脚，就没必要往下走了。

因为咱们看到这里是因为要设置设备的地址，让设备进入 Address 状态，所以针对 SET_ADDRESS 请求再看看这个 devnum。主机向设备发送 SET_ADDRESS 请求时，如果设备处于 Default 状态，就是它现在处的状态，指定一个非 0 值时，设备将进入 Adress 状态，指定 0 值时，设备仍然会处于 Default 状态，所以说即使从这个角度看，这里的

devnum 也是不能为 0 的，不然就是吃饱饭没事干故意找抽。如果设备已经处于 Adress 状态，指定一个非 0 值时，设备仍然会处于 Address 状态，只是将使用新分配的地址，一个设备只能占用一个地址，是包分配的，真正的居者有其屋，如果指定了一个 0 值，则设备将离开 Address 状态退回到 Default 状态。

232 行，power，power_state，event，还有 PM_EVENT_ON 都是电源管理核心里的东西，这里的目的是判断设备所在的那条总线的主机控制器有没有挂起，然后再判断设备本身是不是处于 Suspended 状态，如果挂起了都不欢迎你，还死皮赖脸去个什么劲儿，回去得了。

236 行，常规检查都做完了，core 和 HCD 已经认同了这个 urb，就将它的状态设置为 -EINPROGRESS，表示从现在开始 urb 的控制权就在 core 和 HCD 手里边儿了，驱动那里是看不到这个状态的。

237 行，这时还没开始传输，实际传输的数据长度当然为 0 了，这里初始化这么一下，也是为了防止以后哪里出错返回了，驱动里好检查。

242 行，这几行获得管道的类型还有方向。

246 行，在设备进入 Configured 状态之前，主机只能使用控制传输，通过缺省管道，也就是管道 0 来和设备进行交流。

255 行，获得端点的 wMaxPacketSize，看看 include/linux/usb.h 里定义的这个函数

```
1458 static inline __u16
1459 usb_maxpacket(struct usb_device *udev, int pipe, int is_out)
1460 {
1461         struct usb_host_endpoint        *ep;
1462         unsigned                        epnum = usb_pipeendpoint(pipe);
1463
1464         if (is_out) {
1465                 WARN_ON(usb_pipein(pipe));
1466                 ep = udev->ep_out[epnum];
1467         } else {
1468                 WARN_ON(usb_pipeout(pipe));
1469                 ep = udev->ep_in[epnum];
1470         }
1471         if (!ep)
1472                 return 0;
1473
1474         /* NOTE:  only 0x07ff bits are for packet size... */
1475         return le16_to_cpu(ep->desc.wMaxPacketSize);
1476 }
```

这个函数不管从理论上还是实际上都是很简单的。咱们可以先问下自己，根据现有的信息如何获得一个端点的wMaxPacketSize？当然是必须得获得该端点的描述符，我们知道每个struct usb_device结构体里都有两个数组ep_out和ep_in保存了各个端点对应的struct usb_host_endpoint结构体，只要知道该端点对应了这两个数组里的哪个元素就可以获得它的描述符了，这就需要知道该端点的端点号和方向，而端点的方向就管道的方向，端点号也在pipe里保存有。

你是不是会担心ep_out或ep_in数组都还空着，或者说没有保存对应的端点信息？倒不用担心它还是空的，即使是现在设备还刚从Powered走到Default，百废待兴，连自己的Address都没有，但是在使用usb_alloc_dev构造这个设备的struct usb_device的时候就把它里面端点 0 的struct usb_host_endpoint结构体ep0 指定给ep_out[0]和ep_in[0]了，而且后来还对ep0 的wMaxPacketSize指定了值。不过如果真的没有从它们里面找到想要的那个端点的信息，那肯定就是哪里出错了，指定了错误的端点号，或其它什么原因，也就不用再继续走了，还是打到回府吧。

268 行，如果是等时传输要做一些特别的处理。272 到 276 这几行涉及到高速、高带宽端点（high speed，high bandwidth endpoint）。前面提到 interval 的时候，说过每一帧或微帧最多只能有一次等时传输，完成一次等时 transaction，那时这么说主要是因为还没遇到高速高带宽的等时端点。高速高带宽等时端点每个微帧可以进行 2 到 3 次等时 transaction，它和一般等时端点的主要区别也就在这儿，没必要专门为它搞个描述符类型，端点描述符 wMaxPacketSize 字段的 bit 11~12 就是用来指定可以额外有几次等时 transaction 的，00 表示没有额外的 transaction，01 表示额外有 1 次，10 表示额外有两次，11 被保留。wMaxPacketSize 字段的前 10 位就是实际的端点每次能够处理的最大字节数。所以这几行意思就是如果是高速等时端点，获得它允许的额外的等时 transaction 次数，和每次能够处理的最大字节数，再将它们相乘就得出了该等时端点每个微帧的所能传输的最大字节数。

278 行，number_of_packets 不大于 0 就表示这个等时 urb 没有指定任何一次等时传输，这就怪哉了，咋混过来的，可以直接返回了。

280~286 行对等时 urb 里指定的各次等时传输分别做处理。如果它们预期传输的数据长度比上面算出来的 max 还要大，对不起，要求太过分了，返回吧。然后将它们实际传输的数据长度先置为 0，状态都先初始化为-EXDEV，表示这次等时传输仅仅部分完成了，因为走到这里时传输都还没开始那。

290 行，transfer_buffer_length 长度不能小于 0 吧，等于 0 倒是可以的，毕竟不是什么时候都是有数据要传的。

293 行，见到#ifdef DEBUG 我们都应该很高兴，这意味着直到下面对应的#endif，之间的代码都可以华丽丽的飘过了，给人调试时用的，说明对整体无关痛痒。

338 行，temp 是上面计算出来的管道类型，那下面的各个 case 肯定是针对四种传输类型的了。不过经过实地考察，我们可以发现，这里只 case 了等时传输和中断传输两种周期性的传输类型，因为是关于 interval 的处理，所以就没控制传输和批量传输什么事儿了。

342 行，这里保证等时和中断 urb 的 interval 必须是大于 0 的，不然主机那边儿看不懂你这是表示什么意思，究竟要不要去访问你，搞个负数和 0 含含糊糊的，日里万鸡的主机没功夫去猜你心思。

345 行，这里的 switch 根据目的设备的速度去 case，速度有三种，case 也有三个。我们前面已经说过，不同的速度，urb->interval 可以取不同的范围，不过你可能会发现那时说的最大值要比这里的限制要大一些，这是因为协议归协议，实现归实现，比如，对于 UHCI 来说，中断传输的 interval 不能比 128 更大，而协议规定的最大值为 255。那么现在的问题是，temp 又是做什么用的？要注意 urb->interval 的单位是帧或者微帧，temp 只是为了调整它的值为 2 的次幂，这点从 370 行就可以看出来。

375 行，将 urb 扔给 HCD，然后就进入 HCD 的片儿区了。

本来 usb_submit_urb 函数到此应该结束了，但是它对于写驱动的来说太重要了，驱动里做的所有铺垫就是为了使用 usb_submit_urb 提交一个合适的 urb 给设备，然后满怀期待的等待着设备回馈你需要的信息，再然后才有你接下来的处理，不然你的 usb 驱动只是一纸空谈毫无用处，就像长工一年的辛劳就是为了交够给地主家的租子，咱们一生的辛苦只是为了从房地产商那里讨得一套房子。所以有必要多说一些。

首先还是要再次强调一下，在调用usb_submit_urb提交你的urb之前，一定必须不得不要对它正确的初始化，对于控制/中断/批量传输，core都提供了 usb_fill_control_urb的几个孪生兄弟供你初始化使用，对于等时传输要自己手工一步一步小心翼翼的对urb的相关元素逐个赋值。下层基础决定上层建筑，你的urb决定了你的整个usb驱动能否顺利运转。

第二，对于驱动来说，usb_submit_urb 是异步的，也就是说不用等传输完全完成就返回了，只要 usb_submit_urb 的返回值表示为 0，就表示已经提交成功了，你的 urb 已经被 core 和 HCD 认可了，接下来 core 和 HCD 怎么处理就是它们的事了，驱动该干吗干吗去。比如你东拼西凑拿够了开发商要的款子，他给你个证明，俗称合同，然后你就就不用管也管不着他们怎么去盖，用什么材料去盖，你能做的只是等着收房子。

只要你提交成功了，不管是中间出了差错还是顺利完成，你指定的结束处理函数总是会调用，只有到这个时候，你才能够重新拿到 urb 的控制权，检查它是不是出错了，需要不需要释放或者是重新提交。你只要付了款子，房子盖好了不管怎样总是会有你的，你可以去测量，去检查，去验收，运气好了就 OK，运气不好就自己受着吧，或者去大马路边儿上挂横幅“黑心开发商还我血汗钱”。过程是不同的，道理是一样的。

那么，第三就是，什么时候需要在结束处理函数里重新提交这个 urb？其实，我更想问的是对于中断/等时传输，是怎么实现让主机按一定周期去访问端点的？端点的描述符里已经指定了这个间隔时间是没错儿，urb 里也有 interval 描述了这个间隔周期，更没错儿，可是咱们的 urb 一次只完成一次传输，即使等时传输也只完成有限次的传输，然后就在结束处理函数里返回了，urb 的控制权就完全属于驱动了，接下来的周期访问是怎么做到的？难道脱离 urb 主机自己就去智能化的自动的与端点通信了？OK，即使是这样了，那通信的数据又在哪里，你又怎么去得到这些数据？

事实上，你第一次提交一个中断或等时的 urb 的时候，HCD 会根据 interval 判断一下自己是否能够满足你的需要，如果不能够安排足够的带宽来完成这种周期性的传输，它是不可能会批准你的请求的，如果它估量一下觉得自己可以满足，就会为你保留足够的带宽。但是这并不是就表明万事大吉了，HCD 是为你保留带宽了，可是驱动得保证在对应端点要处理的那个 urb 队列里总是有 urb，不能是空的，否则这个保留的带宽就会被 cancel 掉。那么问题就变成，对于中断/等时传输，如何保证对应端点的 urb 队列里总是会有 urb？这就回到最开始的问题了，驱动需要在结束处理函数里重新初始化和提交刚刚完成的 urb，友情提醒一下，这个时候你是不能够修改 interval 的值的，否则等待你的只能是错误。中断传输的例子可以去看看触摸屏驱动，等时传输的可以去看看摄像头驱动，看看它们在结束处理函数里都做了些什么，你就悟道了。

注意是刚刚完成的！

第四个要说的是，对于控制/批量/中断传输，实际上很多时候你可以不用创建 urb，不用对它初始化，不用调用 usb_submit_urb 来提交，core 将这个过程分别封装在了 usb_control_msg、usb_bulk_msg 和 usb_interrupt_msg 这三个函数里，不同的是它们的实现是同步的，会去等待传输的完全结束。咱们就是从 usb_control_msg 走过来的，所以这里只看看另外两个，它们都定义在 message.c 里

```
145 /**
146  * usb_interrupt_msg - Builds an interrupt urb, sends it off and waits for completion
147  * @usb_dev: pointer to the usb device to send the message to
148  * @pipe: endpoint "pipe" to send the message to
149  * @data: pointer to the data to send
150  * @len: length in bytes of the data to send
151  * @actual_length: pointer to a location to put the actual length transferred in bytes
152  * @timeout: time in msecs to wait for the message to complete before
153  *      timing out (if 0 the wait is forever)
154  * Context: !in_interrupt ()
155  *
156  * This function sends a simple interrupt message to a specified endpoint and
157  * waits for the message to complete, or timeout.
158  *
159  * If successful, it returns 0, otherwise a negative error number.  The number
160  * of actual bytes transferred will be stored in the actual_length paramater.
```

```
161  *
162  * Don't use this function from within an interrupt context, like a bottom half
163  * handler.  If you need an asynchronous message, or need to send a message
164  * from within interrupt context, use usb_submit_urb() If a thread in your
165  * driver uses this call, make sure your disconnect() method can wait for it
to
166  * complete.  Since you don't have a handle on the URB used, you can't cancel
167  * the request.
168  */
169 int usb_interrupt_msg(struct usb_device *usb_dev, unsigned int pipe,
170                       void *data, int len, int *actual_length, int timeout)
171 {
172         return usb_bulk_msg(usb_dev, pipe, data, len, actual_length, timeout);
173 }
```

usb_interrupt_msg够酷，也够省事儿，全部都借助usb_bulk_msg去完成了。

```
176 /**
177  *      usb_bulk_msg - Builds a bulk urb, sends it off and waits for completion
178  *      @usb_dev: pointer to the usb device to send the message to
179  *      @pipe: endpoint "pipe" to send the message to
180  *      @data: pointer to the data to send
181  *      @len: length in bytes of the data to send
182  *      @actual_length: pointer to a location to put the actual length
transferred in bytes
183  *      @timeout: time in msecs to wait for the message to complete before
184  *              timing out (if 0 the wait is forever)
185  *      Context: !in_interrupt ()
186  *
187  *      This function sends a simple bulk message to a specified endpoint
188  *      and waits for the message to complete, or timeout.
189  *
190  *      If successful, it returns 0, otherwise a negative error number.
191  *      The number of actual bytes transferred will be stored in the
192  *      actual_length paramater.
193  *
194  *      Don't use this function from within an interrupt context, like a
195  *      bottom half handler.  If you need an asynchronous message, or need to
196  *      send a message from within interrupt context, use usb_submit_urb()
197  *      If a thread in your driver uses this call, make sure your disconnect()
198  *      method can wait for it to complete.  Since you don't have a handle on
199  *      the URB used, you can't cancel the request.
200  *
201  *      Because there is no usb_interrupt_msg() and no USBDEVFS_INTERRUPT
202  *      ioctl, users are forced to abuse this routine by using it to submit
```

```
203  *      URBs for interrupt endpoints.  We will take the liberty of creating
204  *      an interrupt URB (with the default interval) if the target is an
205  *      interrupt endpoint.
206  */
207 int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
208                         void *data, int len, int *actual_length, int timeout)
209 {
210         struct urb *urb;
211         struct usb_host_endpoint *ep;
212
213         ep = (usb_pipein(pipe) ? usb_dev->ep_in : usb_dev->ep_out)
214                         [usb_pipeendpoint(pipe)];
215         if (!ep || len < 0)
216                 return -EINVAL;
217
218         urb = usb_alloc_urb(0, GFP_KERNEL);
219         if (!urb)
220                 return -ENOMEM;
221
222         if ((ep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
223                         USB_ENDPOINT_XFER_INT) {
224                 pipe = (pipe & ~(3 << 30)) | (PIPE_INTERRUPT << 30);
225                 usb_fill_int_urb(urb, usb_dev, pipe, data, len,
226                                 usb_api_blocking_completion, NULL,
227                                 ep->desc.bInterval);
228         } else
229                 usb_fill_bulk_urb(urb, usb_dev, pipe, data, len,
230                                 usb_api_blocking_completion, NULL);
231
232         return usb_start_wait_urb(urb, timeout, actual_length);
233 }
```

都坚持走到这里了，看懂这个函数还是很easy的，首先根据指定的pipe获得端点的方向和端点号，然后从设备struct usb_device结构体的ep_in或ep_out数组里得道端点对应的struct usb_host_endpoint结构体，接着调用usb_alloc_urb创建urb。因为这个函数可能是从usb_interrupt_msg那里调用过来的，所以接下来要根据端点描述符的bmAttributes字段获取传输的类型，判断究竟是中断传输还是批量传输，是中断传输的话还要修改pipe的类型，防止万一谁谁直接调用usb_bulk_msg来完成中断传输，虽说很少人会穷极无聊到这种地步，预防一下总归是没错的。不管是中断传输还是批量传输，都要调用usb_fill_xxx_urb来初始化，最后，和usb_control_msg一样，调用usb_start_wait_urb函数。

设备的生命线（七）

今年过节不收礼啊，不收礼，收礼只收结构体。

从尖沙咀儿辛辛苦苦赶到铜锣湾，算是进入了 HCD 的片儿区，这里的老大不是帮派头目也不是巡逻片儿警，而是几个结构。C 里边是以结构为王的，随便到一个新地方，新环境，新山头儿，首先要去结识的就是几个占山为王，雄据一方的结构。在 HCD 这个片儿区，这个山头儿，王中之王就是 hcd.h 里定义的 struct usb_hcd。

```
49 /*
50  * USB Host Controller Driver (usb_hcd) framework
51  *
52  * Since "struct usb_bus" is so thin, you can't share much code in it.
53  * This framework is a layer over that, and should be more sharable.
54  */
55
56 /*-------------------------------------------------------------------------*/
57
58 struct usb_hcd {
59
60         /*
61          * housekeeping
62          */
63         struct usb_bus          self;           /* hcd is-a bus */
64         struct kref             kref;           /* reference counter */
65
66         const char              *product_desc;  /* product/vendor string */
67         char                    irq_descr[24];  /* driver + bus # */
68
69         struct timer_list       rh_timer;       /* drives root-hub polling */
70         struct urb              *status_urb;    /* the current status urb */
71 #ifdef CONFIG_PM
72         struct work_struct      wakeup_work;    /* for remote wakeup */
73 #endif
74
75         /*
76          * hardware info/state
77          */
78         const struct hc_driver  *driver;        /* hw-specific hooks */
79
80         /* Flags that need to be manipulated atomically */
81         unsigned long           flags;
82 #define HCD_FLAG_HW_ACCESSIBLE  0x00000001
```

```
83 #define HCD_FLAG_SAW_IRQ        0x00000002
84
85         unsigned                rh_registered:1;/* is root hub registered? */
86
87         /* The next flag is a stopgap, to be removed when all the HCDs
88          * support the new root-hub polling mechanism. */
89         unsigned                uses_new_polling:1;
90         unsigned                poll_rh:1;      /* poll for rh status? */
91         unsigned                poll_pending:1; /* status has changed? */
92         unsigned                wireless:1;     /* Wireless USB HCD */
93
94         int                     irq;            /* irq allocated */
95         void __iomem            *regs;          /* device memory/io */
96         u64                     rsrc_start;     /* memory/io resource start */
97         u64                     rsrc_len;       /* memory/io resource length */
98         unsigned                power_budget;   /* in mA, 0 = no limit */
99
100 #define HCD_BUFFER_POOLS        4
101         struct dma_pool         *pool [HCD_BUFFER_POOLS];
102
103         int                     state;
104 #       define  __ACTIVE                0x01
105 #       define  __SUSPEND               0x04
106 #       define  __TRANSIENT             0x80
107
108 #       define  HC_STATE_HALT           0
109 #       define  HC_STATE_RUNNING        (__ACTIVE)
110 #       define  HC_STATE_QUIESCING      (__SUSPEND|__TRANSIENT|__ACTIVE)
111 #       define  HC_STATE_RESUMING       (__SUSPEND|__TRANSIENT)
112 #       define  HC_STATE_SUSPENDED      (__SUSPEND)
113
114 #define HC_IS_RUNNING(state) ((state) & __ACTIVE)
115 #define HC_IS_SUSPENDED(state) ((state) & __SUSPEND)
116
117         /* more shared queuing code would be good; it should support
118          * smarter scheduling, handle transaction translators, etc;
119          * input size of periodic table to an interrupt scheduler.
120          * (ohci 32, uhci 1024, ehci 256/512/1024).
121          */
122
123         /* The HC driver's private data is stored at the end of
124          * this structure.
125          */
126         unsigned long hcd_priv[0]
```

```
127                             __attribute__ ((aligned (sizeof(unsigned long))));
128 };
```

经过了血与火，熊市与牛市的洗礼，我们都应该对这种变态结构习以为常了，男人么，图不了房子图不了车子图不了美女，能图的还有啥？不就是一颗平常心么。那就用一颗平常心去想想，换作你，会怎么用一个结构去描述主机控制器？毫无头绪吧，这就对了，要真是思如泉涌的话就和写代码的哥们儿一样变态了。

63 行，又一个结构体，struct usb_bus，还名曰self，struct usb_hcd里还有self，看来这家伙是双子座的，以为能再分裂出一个自己，和黄金十二宫里的双子一样。俺也是双子的，不过俺从来没想再分裂出一个，现在世道艰难，一个就已经存活不易了。

为什么这里会用这么一个戏剧性的词汇self？难道Greg他们都是具有乐观主义精神的无产阶级革命者？当然不是，他们都是资产阶级，咱们才是无产阶级。俺在前面的某处提到过那么一下，一个主机控制器就会连出一条usb总线，主机控制器驱动用struct usb_hcd结构表示，一条总线用struct usb_bus结构表示，它们是白天与黑夜般相生相依的关系，一个白天只能连着一个黑夜，一个黑夜只能引出一个白天，没听说过谁过了两个白天才到夜里的，如果谁说了，他不是疯子就是哲学家或经济学家。struct usb_bus在include/linux/usb.h里定义

```
273 /*
274  * Allocated per bus (tree of devices) we have:
275  */
276 struct usb_bus {
277         struct device *controller;      /* host/master side hardware */
278         int busnum;                     /* Bus number (in order of reg) */
279         char *bus_name;                 /* stable id (PCI slot_name etc) */
280         u8 uses_dma;                    /* Does the host controller use DMA? */
281         u8 otg_port;                    /* 0, or number of OTG/HNP port */
282         unsigned is_b_host:1;           /* true during some HNP roleswitches */
283         unsigned b_hnp_enable:1;        /* OTG: did A-Host enable HNP? */
284
285         int devnum_next;                /* Next open device number in
286                                          * round-robin allocation */
287
288         struct usb_devmap devmap;       /* device address allocation map */
289         struct usb_device *root_hub;    /* Root hub */
290         struct list_head bus_list;      /* list of busses */
291
292         int bandwidth_allocated;        /* on this bus: how much of the time
293                                          * reserved for periodic (intr/iso)
294                                          * requests is used, on average?
295                                          * Units: microseconds/frame.
```

```
296                                         * Limits: Full/low speed reserve 90%,
297                                          * while high speed reserves 80%.
298                                          */
299         int bandwidth_int_reqs;         /* number of Interrupt requests */
300         int bandwidth_isoc_reqs;        /* number of Isoc. requests */
301
302 #ifdef CONFIG_USB_DEVICEFS
303         struct dentry *usbfs_dentry;    /* usbfs dentry entry for the bus */
304 #endif
305         struct class_device *class_dev; /* class device for this bus */
306
307 #if defined(CONFIG_USB_MON)
308         struct mon_bus *mon_bus;        /* non-null when associated */
309         int monitored;                  /* non-zero when monitored */
310 #endif
311 };
```

277 行，controller，struct usb_hcd那里含了个usb_bus，这里就回应了个controller，你山西来个黑砖窑，我唐山就回应个黑军车，黑对黑，遥相呼应。那现在通过struct usb_hcd里的self和struct usb_bus里的controller这两个很有乐观主义精神的词儿，你能不能说下它们到底是什么关系？你当然可以说是一个对应主机控制器，一个描述一条总线，但其实对于写代码的来说一个主机控制器和一条总线差不多是一码事，不用分的那么清，可以简单的说它们都是用来描述主机控制器的，那为什么又分成了两个结构，难道Greg他们现在又不信奉简约主义了？

这个问题的答案我也很想知道，但知道了又能怎么样？知道了你就能明白为什么美女都不喜欢你，为什么她们身上衣服件数越多反而露得越多？你不能明白，所以也不用去知道了。不过思索了一杯茶的时间，还是有那么点儿线索。不要小看这杯茶，按日里万鸡来换算，这点儿时间都能理多少鸡了？

前面说过 linux 里和小李飞刀齐名的就是设备模型了，usb 主机控制器当然也是一个设备，而且更多的时候它还是一个 PCI 设备，那它就应该纳入这个设备模型范畴之内，struct usb_hcd 结构里就得嵌入类似 struct device 或 struct pci_dev 这样的一个结构体，但是你仔细瞅瞅，能不能在它里面发现这么一个成员？不能，对于一个设备来说，这可是大逆不道的。但是你再瞅瞅 struct usb_bus，第一个就是一个 struct device 结构体。好，第一条线索就先到这儿。

再利用这杯茶的时间挑个具体的主机控制器驱动程序快速的走一下，就 UHCI 吧，都在 host 目录下的 uhci-族文件里，首先它是个 pci 设备，要使用 pci_register_driver 注册一个 struct pci_driver 结构体 uhci_pci_driver，uhci_pci_driver 里又有个熟悉的 probe，在这个 probe 里，它调用 usb_create_hcd 来创建一个 usb_hcd，初始化里面的 self，还将这个 self 里的 controller 设定为描述主机控制器的那个 pci_dev 里的 struct device

结构体，从而将 usb_hcd、usb_bus 和 pci_dev，甚至设备模型都连接起来了。

这杯茶应该还没有这么快就喝的完，那就再接着巡视一下 uhci-文件里定义的那些函数，只用看它们的参数，你会发现参数里不是 struct usb_hcd 就是 struct uhci_hcd，如果你和我一样无聊愿意多看点的话，你会看到那些函数的前面几行常常会有 hcd_to_uhci 或者 uhci_to_hcd 这样的函数在 struct usb_hcd 和 struct uhci_hcd 之间做着转换。struct uhci_hcd 是什么？它是 uhci 自己私有的一个结构体，就像每个成功的男人背后都有一个女人一样，每个具体的主机控制器都有这么一个类似的结构体。如果你再无聊一下，顺便瞧了下 hcd_to_uhci 或者 uhci_to_hcd 的定义，你就会明白，每个主机控制器的这个私有结构体都藏在 struct usb_hcd 结构最后的那个 hcd_priv 变长数组里。

通过这杯茶，你能悟出什么？如果说镜头闪的太快，让你看的不太明白，那就只管听俺说好了。对于具体的主机控制器驱动来说，它们的眼里只有 struct usb_hcd，struct usb_hcd 结构之于主机控制器驱动，就如同 struct usb_device 或 struct usb_interface 之于 usb 驱动。没有 usb_create_hcd 去创建 usb_hcd，就不会有 usb_bus 的存在。而对于 linux 设备模型来说，struct usb_bus 无疑要更亲切一些。总之，你可以把 struct usb_bus 当作只是嵌入到 struct usb_hcd 里面的一个结构体，它将 struct usb_hcd 要完成的一部分工作进行了封装，因为要描述一个主机控制器太复杂太难，于是就开了 struct usb_bus 这么一个窗户去专门面对设备模型、sysfs 等等。这也就是俺开头儿就说这个片儿区，struct usb_hcd 才是王中之王的原因。

你知道 Greg 他们是怎么描述这种奇妙的关系么？他们把这个叫作 HCD bus-glue layer，并致力于 flatten out it. 这个关系早先是比较混沌的，现在要清晰些，以后只会更清晰，struct usb_hcd 越来越走上台前，struct usb_bus 越来越走向幕后。就好像我们一开始是天地混沌，然后是女娲造人，有了社会有了阶级，再然后有了 GCD 才有了新中国一样。

278 行，busnum，总线编号，你的机子里总可以有多个主机控制器吧，自然也就可以有多条 usb 总线了，既然可以有多条，就要编个号方便确认了。有关总线编号，可以看看定义在 drivesr/usb/core/hcd.c 里的这几行

```
88 /* used when allocating bus numbers */
89 #define USB_MAXBUS              64
90 struct usb_busmap {
91         unsigned long busmap [USB_MAXBUS / (8*sizeof (unsigned long))];
92 };
93 static struct usb_busmap busmap;
```

讲struct usb_device的devnum时候，说到过一个devicemap，这里又有个 busmap，当时分析说devicemap一共有 128 位，同理可知，这里的 busmap一共有 64 位，也就是说最多可以有 64 条usb总线，如果你还觉得不够，言一声，我可以躲你远远的。

279 行，bus_name，bus 总线，name 名字，bus_name 总线的名字，什么样的名字？

要知道大多数情况下主机控制器都是一个 PCI 设备，那么 bus_name 应该就是用来在 PCI 总线上标识 usb 主机控制器的名字，PCI 总线使用标准的 PCI ID 来标识 PCI 设备，所以 bus_name 里保存的应该就是主机控制器对应的 PCI ID。UHCI 等调用 usb_create_hcd 创建 usb_hcd 的时候确实是将它们对应 PCI ID 赋给了 bus_name。

现在简单说说这个 PCI ID。PCI spec 允许单个系统可以最多有 256 条 PCI 总线，对咱们当然是太多了，但是对于一些极变态，需求极为旺盛的系统，它可能还觉得这满足不了要求，于是所有的 PCI 总线又被划分为 domain，每个 PCI domain 又可以最多拥有 256 条总线，这下总该够了吧，而每条总线上又可以支持 32 个设备，这些设备里边儿还都可以是多功能板，它们还都可以最多支持 8 种功能。那系统怎么来区分每种功能？总要知道它在哪个 domain，哪条总线，哪个设备板上吧。这么说还是太笼统了，你可以用 lspci 命令看一下

```
00:00.0 Host bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX Host bridge (rev 01)
00:01.0 PCI bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX AGP bridge (rev 01)
00:07.0 ISA bridge: Intel Corporation 82371AB/EB/MB PIIX4 ISA (rev 08)
00:07.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:07.2 USB Controller: Intel Corporation 82371AB/EB/MB PIIX4 USB
00:07.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0f.0 VGA compatible controller: VMware Inc [VMware SVGA II] PCI Display Adapter
00:10.0 SCSI storage controller: LSI Logic / Symbios Logic 53c1030 PCI-X Fusion-MPT Dual Ultra320
SCSI (rev 01)
00:11.0 Ethernet controller: Advanced Micro Devices [AMD] 79c970 [PCnet32 LANCE] (rev 10)
00:12.0 Multimedia audio controller: Ensoniq ES1371 [AudioPCI-97] (rev 02)
```

每行前面的数字就是所谓的 PCI ID，每个 PCI ID 由 domain 号（16 位），总线编号（8 位），设备号（5 位），功能号（3 位）组成，不过这里 lspci 没有标明 domain 号，但对于一台普通 PC 而言，一般也就只有一个 domain，0x0000。

280 行，uses_dma，表明这个主机控制器支持不支持 DMA。主机控制器的一项重要工作就是在内存和 USB 总线之间传输数据，这个过程可以使用 DMA 或者不使用 DMA，不使用 DMA 的方式即所谓的 PIO 方式。DMA 代表着 Direct Memory Access，即直接内存访问，不需要 CPU 去干预。具体的去看看 PCI DMA 的东东吧，因为一般来说主机控制器都是 PCI 设备，uses_dma 都在它们自己的 probe 函数里设置了。

281~283 行，有关 otg 的，飘过。

285 行，devnum_next，288 行，devmap，早就说过 devmap 这张表了，devnum_next 中记录的就是这张表里下一个为 0 的位，里面为 1 的位都是已经被这条总线上的 usb 设备占据了的，名花有主的。

289 行，root_hub，就好像端点 0 在所有设备的端点里面那么的鹤立鸡群一样，root hub 在所有的 hub 里面也是那么的特殊，还记得 usb 的那颗树么，它就是那颗树的根，和 usb

主机控制器绑定在一起，其它的 hub 和设备都必须从它这儿延伸出去。正是因为这种特殊的关系，写代码的哥们儿也素有成人之心，就直接将它放在了 struct usb_bus 结构里，让他们永不分离。usb 主机控制器，usb 总线，root hub，1 比 1 比 1。

290 行，bus_list，在 hcd.c 中定义有一个全局队列 usb_bus_list

```
84 /* host controllers we manage */
85 LIST_HEAD (usb_bus_list);
86 EXPORT_SYMBOL_GPL (usb_bus_list);
```

它就是所有 usb 总线的组织。每次一条总线新添加进来，都要向这个组织靠拢，都要使用 bus_list 字段链接在这个队列上。

292 行，bandwidth_allocated，表明总线为中断传输和等时传输预留了多少带宽，协议里说了，对于高速来说，最多可以有 80%，对于低速和全速要多点儿，可以达到 90%。它的单位是微妙，表示一帧或微帧内有多少微妙可以留给中断/等时传输用。

299 行，bandwidth_int_reqs，300 行，bandwidth_isoc_reqs，分别表示当前中断传输和等时传输的数量。

302~304 行，是 usbfs 的，每条总线都对应于/proc/bus/usb 下的一个目录。你无聊的话可以去瞅瞅。

305 行，class_dev，这里又牵涉到设备模型中的一个概念，设备的 class，即设备的类。像前面提到的设备模型里的总线、设备、驱动三个核心概念，纯粹是从写驱动的角度看的，而这里的类则是面向于 linux 的广大用户的，它不管你是用什么接口，怎么去连接，它只管你对用户来说提供了什么功能，一个 SCSI 硬盘和一个 ATA 硬盘对驱动来说是八杆子打不着的两个东西，但是对于用户来说，它们都是硬盘，都是用来备份文件，备份各种小电影的，这也就是所谓的物以类聚人以群分。

设备模型与 sysfs 是分不开的，class 在 sysfs 里的体现就在/sys/class 下面，可以去看看

```
atm           dma         graphics hwmon           i2c-adapter      input
mem           misc        net      pci_bus         scsi_device      scsi_disk
scsi_host     sound       spi_host spi_master      spi_transport    tty
usb_device    usb_endpoint          usb_host        vc               vtconsole
```

看到里面的 usb_host 了吧，它就是所有 usb 主机控制器的类，这些目录都是怎么来的那？咱们还要追溯一下 usb 子系统的初始化函数 usb_init，它里面有这么一段

```
877         retval = usb_host_init();
878         if (retval)
879                 goto host_init_failed;
```

当时只是简单说这是用来初始化 host controller 的，现在鼓气勇气进去看看，在 hcd.c 里

```
671 static struct class *usb_host_class;
672
673 int usb_host_init(void)
674 {
675         int retval = 0;
676
677         usb_host_class = class_create(THIS_MODULE, "usb_host");
678         if (IS_ERR(usb_host_class))
679                 retval = PTR_ERR(usb_host_class);
680         return retval;
681 }
```

usb_host_init所作的一切就是调用 class_create创建了一个usb_host这样的类，你只要加载了usbcore模块就能在/sys/class下面看到有usb_host目录出现。既然usb_host目录表示的是usb主机控制器的类，那么它下面应该就对应各个具体的主机控制器了，你用ls 命令look一下就能看到usb_host1、usb_host2 等等这样的目录，它们每个都对应一个在你系统里实际存在的主机控制器，实际上在hcd.c里的usb_register_bus函数有这么几行

```
735         bus->class_dev = class_device_create(usb_host_class, NULL, MKDEV(0,0),
736                                              bus->controller, "usb_host%d", busnum);
```

这两行就是使用 class_device_create在/sys/class/usb_host下面为每条总线创建了一个目录，目录名里的数字代表的就是每条总线的编号，usb_register_bus函数是每个主机控制器驱动在probe里调用的，向usb core注册一条总线，也可以说是注册一个主机控制器。

307~310 行，CONFIG_USB_MON 是干吗用的？这要看看 drivers/usb/mon 目录下的 Kconfig

```
1 #
2 # USB Monitor configuration
3 #
4
5 config USB_MON
6         bool "USB Monitor"
7         depends on USB!=n
8         default y
9         help
10           If you say Y here, a component which captures the USB traffic
11           between peripheral-specific drivers and HC drivers will be built.
12           For more information, see <file:Documentation/usb/usbmon.txt>.
13
14           This is somewhat experimental at this time, but it should be safe.
```

```
15
16          If unsure, say Y.
```

文件里就这么多内容，从里面咱们可以知道，如果定义了 CONFIG_USB_MON，一个所谓的 usb Monitor，也就是 usb 监视器的东东就会编进内核。这个 Monitor 是用来监视 usb 总线上的底层通信流的，相关的文件都在 drivers/usb/mon 下面。2005 年的阳春三月，Greg 大侠春心思动，于是就孕育出了这个 usb Monitor。

设备的生命线（八）

这个世界上不需要努力就能得到的东西只有一样，那就是年龄。所以要不怕苦不怕累，回到 struct usb_hcd，继续努力的往下看。

64 行，又见 kref，usb 主机控制器的引用计数。struct usb_hcd 也有自己专用的引用计数函数，看 hcd.c 文件

```
1526 static void hcd_release (struct kref *kref)
1527 {
1528         struct usb_hcd *hcd = container_of (kref, struct usb_hcd, kref);
1529
1530         kfree(hcd);
1531 }
1532
1533 struct usb_hcd *usb_get_hcd (struct usb_hcd *hcd)
1534 {
1535         if (hcd)
1536                 kref_get (&hcd->kref);
1537         return hcd;
1538 }
1539 EXPORT_SYMBOL (usb_get_hcd);
1540
1541 void usb_put_hcd (struct usb_hcd *hcd)
1542 {
1543         if (hcd)
1544                 kref_put (&hcd->kref, hcd_release);
1545 }
1546 EXPORT_SYMBOL (usb_put_hcd);
```

和 struct urb 的那几个长的也忒像了，像的俺都不好意思多介绍它们了，如果不明白就回去看看聊 struct urb 的时候怎么说的吧。

66 行，product_desc，主机控制器的产品描述字符串，对于 UHCI，它为“UHCI Host Controller”，对于 EHCI，它为“EHCI Host Controller”。

67 行，irq_descr[24]，这里边儿保存的是“ehci-hcd:usb1”之类的字符串，也就是驱动的大名再加上总线编号。

71~73 行，电源管理的，飘过。

78 行，driver，每个男人心中都有一个狐狸精，每个女人心里都有一个洛丽塔，每个主机控制器驱动都有一个 struct hc_driver 结构体。看看它在 hcd.h 里的定义

```
149 struct hc_driver {
150         const char      *description;   /* "ehci-hcd" etc */
151         const char      *product_desc;  /* product/vendor string */
152         size_t          hcd_priv_size;  /* size of private data */
153
154         /* irq handler */
155         irqreturn_t     (*irq) (struct usb_hcd *hcd);
156
157         int     flags;
158 #define HCD_MEMORY      0x0001          /* HC regs use memory (else I/O) */
159 #define HCD_USB11       0x0010          /* USB 1.1 */
160 #define HCD_USB2        0x0020          /* USB 2.0 */
161
162         /* called to init HCD and root hub */
163         int     (*reset) (struct usb_hcd *hcd);
164         int     (*start) (struct usb_hcd *hcd);
165
166         /* NOTE:  these suspend/resume calls relate to the HC as
167          * a whole, not just the root hub; they're for PCI bus glue.
168          */
169         /* called after suspending the hub, before entering D3 etc */
170         int     (*suspend) (struct usb_hcd *hcd, pm_message_t message);
171
172         /* called after entering D0 (etc), before resuming the hub */
173         int     (*resume) (struct usb_hcd *hcd);
174
175         /* cleanly make HCD stop writing memory and doing I/O */
176         void    (*stop) (struct usb_hcd *hcd);
177
178         /* shutdown HCD */
179         void    (*shutdown) (struct usb_hcd *hcd);
180
181         /* return current frame number */
```

```
182         int     (*get_frame_number) (struct usb_hcd *hcd);
183
184         /* manage i/o requests, device state */
185         int     (*urb_enqueue) (struct usb_hcd *hcd,
186                                     struct usb_host_endpoint *ep,
187                                     struct urb *urb,
188                                     gfp_t mem_flags);
189         int     (*urb_dequeue) (struct usb_hcd *hcd, struct urb *urb);
190
191         /* hw synch, freeing endpoint resources that urb_dequeue can't */
192         void    (*endpoint_disable)(struct usb_hcd *hcd,
193                         struct usb_host_endpoint *ep);
194
195         /* root hub support */
196         int             (*hub_status_data) (struct usb_hcd *hcd, char *buf);
197         int             (*hub_control) (struct usb_hcd *hcd,
198                                 u16 typeReq, u16 wValue, u16 wIndex,
199                                 char *buf, u16 wLength);
200         int             (*bus_suspend)(struct usb_hcd *);
201         int             (*bus_resume)(struct usb_hcd *);
202         int         (*start_port_reset)(struct usb_hcd *, unsigned port_num);
203         void            (*hub_irq_enable)(struct usb_hcd *);
204                 /* Needed only if port-change IRQs are level-triggered */
205 };
```

和usb_driver一样，和pci_driver也一样，所有的xxx_driver都有一堆函数指针，具体的主机控制器驱动就靠它们来活的，不多说了，太深入了，做人要懂得适可而止，这里只说下函数指针之外的东西，也就是开头儿的那三个。description直白点说就是驱动的大名，比如对于UHCI，它是“uhci_hcd”，对于EHCI，它就是“ehci_hcd”。product_desc和struct usb_hcd里的那个是一个样儿。hcd_priv_size还是有点儿意思的，前面喝那杯茶的时候提到过，每个主机控制器驱动都会有一个私有结构体，藏在struct usb_hcd最后的那个变长数组里，这个变也是相对的，在创建usb_hcd的时候也得知道它能变多长，不然谁知道要申请多少内存啊，这个长度就hcd_priv_size。

81 行，flags，属于 HCD 的一些标志，可用值就在 82，83 行。它们什么意思？书到用时方恨少，flags 到用时才可知。

69，70，85~91，这几行都是专为 root hub 服务的。佛说，一花一世界，一叶一如来。一个 host controller 对应一个 root hub，即使某些嵌入式系统里，硬件上 host controller 没有集成 root hub，软件上也需要虚拟一个出来，也就是所谓的 virtual root hub。它位置是特殊的，但需要提供的功能和其它 hub 是没有什么差别的，仅仅是在和 host controller 的软硬件接口上有一些特别的规定。那个网络红人 ayawawa 再怎么觉得自己有多特殊，觉得“漂亮者皆不如我聪明，聪明者皆不如我漂亮”，她也始终是个需要男人去关怀的女人，

root hub 再怎么特别也始终是一个 hub，是一个 usb 设备，也不能脱离 usb 这个大家庭，也要向组织注册，也要有自己的设备生命线，既然这里遇到了，就说说它的这条线。

/*----------root hub 的生命线----------

还是要先声明一下，root hub 的生命线只是咱们的主线里的一个岔路口，不会沿着它去仔细的走，只会尽量的使用快镜头去展示一下。如果有哪里不明白，等走完了那条主线，你再回过头来看看，很多事情都是在我们蓦然回首的时候才明白的。

基于 root hub 和 host controller 这种极为暧昧极为特殊的关系，UHCI、EHCI 等主机控制器驱动程序在自己的初始化代码里都有大量的篇幅大量的心思花在 root hub 上面。不管是 UHCI，还是 EHCI，一般来说都是 PCI 设备，是 PCI 设备都应该有个 struct pci_driver 结构体，都应该有一个属于自己的 probe。在这个 probe 里，也都会调用 usb_create_hcd()来创建一个属于自己的 usb_hcd，也都会调用 usb_add_hcd()将这个刚刚创建的 usb_hcd 注册到 usb 组织里。于是 root hub 便披着皇帝的新装，走着猫步登场了。这里只贴与 root hub 相关的一些主要代码

```
1580          if ((rhdev = usb_alloc_dev(NULL, &hcd->self, 0)) == NULL) {
1581               dev_err(hcd->self.controller, "unable to allocate root hub\n");
1582                  retval = -ENOMEM;
1583                  goto err_allocate_root_hub;
1584          }
1585          rhdev->speed = (hcd->driver->flags & HCD_USB2) ? USB_SPEED_HIGH :
1586                                  USB_SPEED_FULL;
1587          hcd->self.root_hub = rhdev;
```

在 usb_add_hcd 函数里，首先会调用咱们已经亲密接触过的 usb_alloc_dev()来为 root hub 准备一个 struct usb_device 结构体，我唯一想再提一下的是这时 root hub 的设备类型被赋值为 usb_device_type，所属总线类型赋值为 usb_bus_type。然后根据各个 host controller 的实际情况，设定它的速度，现在你应该知道 struct hc_driver 里的那个漏过的 flags 是干吗的了吧，它等于 HCD_USB2 时表示这个 host controller 绑定的 root hub 是高速 hub，等于 HCD_USB1 时表示是低速/全速 hub。struct usb_device 结构体准备妥当之后，就要赋给 struct usb_bus 的 root_hub 元素。这些都只不过是准备工作，重头戏都在 usb_add_hcd()最后调用的 register_root_hub()里面。

register_root_hub()非常肯定的将 root hub 的设备号 devnum 设置为 1，于是就少了选择地址设置地址的过程，root hub 直接跳跃式发展进入了 Address 状态，咱们耗费几代人，期待数十年才能完成的壮举，root hub 轻而易举的就实现了。当然，是个人都知道在 Address 状态是好处多多便利多多的，不然房地产就支柱不起来了。起码这时可以很方便的获得 root hub 的设备描述符，可以调用 usb_new_device 将 root hub 送给无所不能的设备模型，去寻找它命中的驱动。

```
817          retval = usb_new_device (usb_dev);
```

```
818         if (retval) {
819                 dev_err (parent_dev, "can't register root hub for %s, %d\n",
820                                 usb_dev->dev.bus_id, retval);
821         }
```

如果这些都一切顺利的话，register_root_hub()在最后就会将 struct usb_hcd 的 rh_registered 设置为 1，表示 root hub 已经找到组织并加入到革命队伍里了。

Linux设备模型根据root hub所属的总线类型usb_bus_type将它添加到usb总线的那条有名的设备链表里，然后会轮询usb总线的另外一条有名的驱动链表，针对每个找到的驱动去调用usb总线的match函数，也就是咱们以前一再遇到以后不断遇到的usb_device_match()，为root hub牵线搭桥寻找另一个匹配的半圆。要注意，root hub的设备类型是usb_device_type，所以在usb_device_match()里走的设备那条路，匹配成功的是那个对所有usb_device_type类型的设备都来者不拒的花心大萝卜，usb世界里唯一的那个usb设备驱动（不是usb接口驱动）struct device_driver结构体对象usb_generic_driver。所以接下来要调用的就是usb_generic_driver的probe函数generic_probe()，去配置root hub，然后root hub就大步迈进了Configured状态。因为root hub只有一个配置，所以generic_probe()配置root hub时并没有多少选择的烦恼，如果有所疑问，可以look下hcd.c的开头就已经设定好的root hub设备描述符

```
119 /* usb 2.0 root hub device descriptor */
120 static const u8 usb2_rh_dev_descriptor [18] = {
121         0x12,       /*  __u8  bLength; */
122         0x01,       /*  __u8  bDescriptorType; Device */
123         0x00, 0x02, /*  __le16 bcdUSB; v2.0 */
124
125         0x09,       /*  __u8  bDeviceClass; HUB_CLASSCODE */
126         0x00,       /*  __u8  bDeviceSubClass; */
127         0x01,       /*  __u8  bDeviceProtocol; [ usb 2.0 single TT ]*/
128         0x40,       /*  __u8  bMaxPacketSize0; 64 Bytes */
129
130         0x00, 0x00, /*  __le16 idVendor; */
131         0x00, 0x00, /*  __le16 idProduct; */
132         KERNEL_VER, KERNEL_REL, /*  __le16 bcdDevice */
133
134         0x03,       /*  __u8  iManufacturer; */
135         0x02,       /*  __u8  iProduct; */
136         0x01,       /*  __u8  iSerialNumber; */
137         0x01        /*  __u8  bNumConfigurations; */
138 };
```

这张表是 const 的，也就是说在整个 usb 的世界里它都是只能去读不能去修改的，要用严谨科学的态度去对待 root hub。明眼人一眼就能看到躲在它最后的那个 0x01，这就是 root

hub 支持的配置数量。在它下边儿还有针对 usb 1.1 的 root hub 设备描述符，大同小异就不贴了。

既然进入了 Configured 状态，就可以无拘无束的使用 root hub 提供的所有功能了，不过，对于咱们使用 usb 的劳苦大众来说，实际在起作用的还是设备里的接口，所以 generic_probe()接下来就会根据 root hub 使用的配置，为它所有的接口准备 struct usb_interface 结构体对象，这时接口所属的总线类型仍然为 usb_bus_type，设备类型就不一样了，为 usb_if_device_type，早先说过的那句总线有总线的类型，设备有设备的类型到这里就应该再加上一句，接口有接口的类型。usb_if_device_type 在 message.c 里定义

```
1382 struct device_type usb_if_device_type = {
1383         .name =         "usb_interface",
1384         .release =      usb_release_interface,
1385         .uevent =       usb_if_uevent,
1386 };
```

为 root hub 的接口准备 struct usb_interface 结构体也没费太大功夫，因为 spec 里规定了 hub 上除了端点 0，就只有一个中断的 IN 端点，并不用准备太多的东西。root hub 的配置描述符也已经在 hcd.c 的开头儿指定好了，确实只有一个接口，一个设置，一个端点，去 look 一下

```
166 /* Configuration descriptors for our root hubs */
167
168 static const u8 fs_rh_config_descriptor [] = {
169
170         /* one configuration */
171         0x09,       /*  __u8  bLength; */
172         0x02,       /*  __u8  bDescriptorType; Configuration */
173         0x19, 0x00, /*  __le16 wTotalLength; */
174         0x01,       /*  __u8  bNumInterfaces; (1) */
175         0x01,       /*  __u8  bConfigurationValue; */
176         0x00,       /*  __u8  iConfiguration; */
177         0xc0,       /*  __u8  bmAttributes;
178                                 Bit 7: must be set,
179                                     6: Self-powered,
180                                     5: Remote wakeup,
181                                     4..0: resvd */
182         0x00,       /*  __u8  MaxPower; */
183
184         /* USB 1.1:
185          * USB 2.0, single TT organization (mandatory):
186          *      one interface, protocol 0
187          *
```

```
188          * USB 2.0, multiple TT organization (optional):
189          *      two interfaces, protocols 1 (like single TT)
190          *      and 2 (multiple TT mode) ... config is
191          *      sometimes settable
192          *      NOT IMPLEMENTED
193          */
194
195         /* one interface */
196         0x09,       /*  __u8  if_bLength; */
197         0x04,       /*  __u8  if_bDescriptorType; Interface */
198         0x00,       /*  __u8  if_bInterfaceNumber; */
199         0x00,       /*  __u8  if_bAlternateSetting; */
200         0x01,       /*  __u8  if_bNumEndpoints; */
201         0x09,       /*  __u8  if_bInterfaceClass; HUB_CLASSCODE */
202         0x00,       /*  __u8  if_bInterfaceSubClass; */
203         0x00,       /*  __u8  if_bInterfaceProtocol; [usb1.1 or single tt] */
204         0x00,       /*  __u8  if_iInterface; */
205
206         /* one endpoint (status change endpoint) */
207         0x07,       /*  __u8  ep_bLength; */
208         0x05,       /*  __u8  ep_bDescriptorType; Endpoint */
209         0x81,       /*  __u8  ep_bEndpointAddress; IN Endpoint 1 */
210         0x03,       /*  __u8  ep_bmAttributes; Interrupt */
211         0x02, 0x00, /*  __le16 ep_wMaxPacketSize; 1 + (MAX_ROOT_PORTS / 8) */
212         0xff        /*  __u8  ep_bInterval; (255ms -- usb 2.0 spec) */
213 };
```

174 行的 0x01，199 行 0x00，200 行的 0x01，分别指定了接口数量，使用的设置，端点的数目。也就是说，如果加上端点 0，root hub 只有两个端点，而另外一个端点为 IN 端点，端点号为 1，中断类型，每次可以处理的最大字节数为 2，期望 host controller 访问自己的时间间隔为 256ms。这里打个岔问个问题，为什么说这个中断端点的 wMaxPacketSize 为 2，211 行不是明确写着 0x02，0x00 么？呵呵，协议里说了，usb 设备的各种描述符都是按照 little-endian 方式的字节序存储的，先是低字节然后是高字节。

为root hub的那个独苗儿接口准备好struct usb_interface结构体之后应该怎么做？佛又说了，一个接口一个驱动，接下来显然是应该将它送给设备模型去寻找它命中注定的驱动了。然后此处省略 2008 字，又到了 usb_device_match()。不过你说这个时候设备和接口两条路它应该走哪条？它的类型已经设置成usb_if_device_type了，设备那条路把门儿的根本就不会让它进，所以它必须得去走接口那条路。

有关在接口这条路上它怎么去寻找另一半儿，咱现在不多说，日后在主线里说，现在只天马行空的扯一下。首先回头瞧下 root hub 的设备描述符，它的 bDeviceClass 被指定为 0x09，在 include/linux/usb/ch9.h 里，它对应的是 USB_CLASS_HUB，再回头看看

root hub 配置描述符的 201 行，bInterfaceClass 也被指定成 0x09，也就是说同样为 USB_CLASS_HUB。你再去 hub 驱动对应的 hub.c 文件里看看那里的 hub_id_table，是不是应该明白点什么了？对头，root hub 里的那个接口所对应的驱动就是 hub 驱动，虽然它很特殊，可它也是个 hub 啊。root hub 一边和 host controller 相生相依，一边和 hub 驱动眉来眼去花前月下，多么巨大的讽刺。

接下来要做的就是调用 hub 驱动里的 hub_probe()。不管是 root hub 还是一般的 hub，走到这一步都是必然的，不同的是对于 root hub 在 host controller 初始化的过程中就会去调用 hub_probe()。再省略 2008 字后，hub_probe()通过 hub_configure()创建并初始化了一个中断的 urb，然后在 hub_activate()里调用 usb_submit_urb()将它提交给 core，接着就又回到了 HCD。如果觉得省略了这么多让你很不爽，那就再去看看《我是 hub》吧。

每个 hub 都要需要进行中断传输来获得 hub 的状态变化，不然 hub 驱动里那个著名的 hub_events 就活不下去了，你连在 hub 上的 usb 设备系统也就察觉不到了，你也就不能用 usb 摄像头来泡 mm 了，……，多米诺骨牌的一连串反应。都走到了 21 世纪，离奥运都不到 300 天了，都应该知道中断传输不会提交一次就完事儿了，这么一次就完事儿 hub 不会满意，每个男人女人都不会满意。你需要在你的结束处理函数离提交再提交，host controller 都不嫌烦，你烦什么啊。中断传输都是有个间隔时间的，这个时间不由你决定，也不由我决定，咱们只能期待，决定权在 host controller 那里。host controller 就决定了，对于 root hub 要每隔 250ms 去访问一次，你再看上面 root hub 的描述符，里面显示它期待的时间为 0xff，也就是 256ms，250 与 256 也差不了多少，root hub 也应该满意了。这个固定的访问频率 host controller 是怎么实现的？就是通过 struct usb_hcd 里的那个定时器 rh_timer，定时器就是专门干这种事儿的，root hub 每提交一次 urb，在 HCD 里都会对它初始化一次，指定 250ms 之后去获得 root hub 的状态，然后就让 urb 返回给 root hub，于是 root hub 再提交，再 250ms 后返回，……。

但是这种对 root hub 的轮询机制是早先版本里的，现在就不一样了，世道变了，队伍不好带了。struct usb_hcd 里出现了 uses_new_polling，看名字就知道是一种新机制出现了。这也是 Alan Stern 在 2005 年的阳春三月春心萌动孕育出来的，三月真是一年里的好时候。旧社会主要靠剥削，新社会主要靠奉献，旧机制主要靠轮询，新机制主要靠中断。在新的机制里，root hub 仍然是要提交一个中断 urb 的，不同的是这时它可以不用再请定时器来帮忙，在有设备插入时，host controller 会在自己的中断处理函数里调用一个名叫 usb_hcd_poll_rh_status 的函数去获得 root hub 的状态并将 urb 的所有权归还给 hub 驱动。

那是不是就可以不需要定时器 rh_timer 了？事情没这么简单，也不能就这么过河拆桥，由于某些不可言状的原因，原来的机制还必须得保留着，给不同的 host controller 选择使用。为了让新旧机制和谐的运作，usb_hcd_poll_rh_status 多了一个职务，就是作为 rh_timer 的定时器函数，而且 struct usb_hcd 里除了 uses_new_polling 之外就又多了 poll_rh 等几个元素。如果 uses_new_polling 没有被设置，则一定会采用旧的轮询机制，

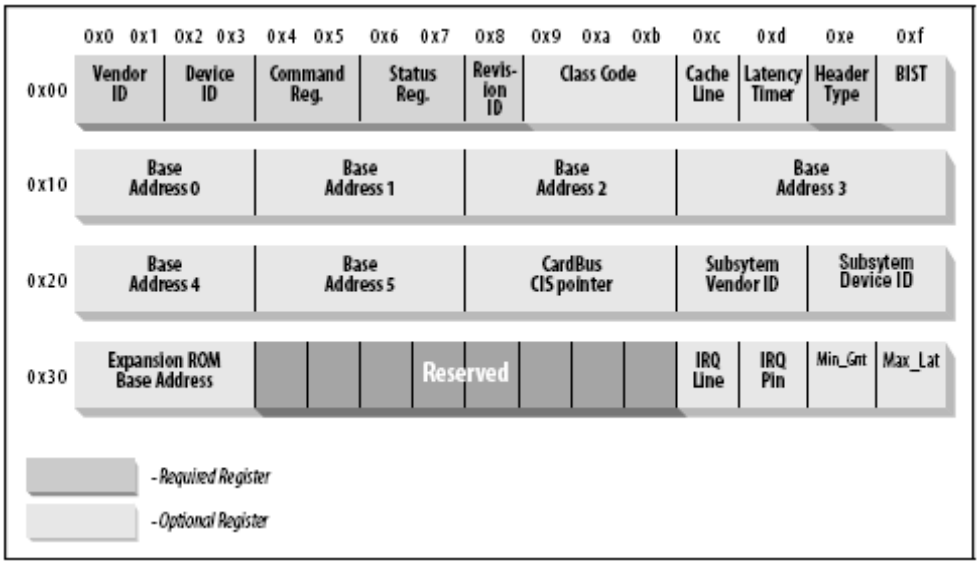
如果 uses_new_polling 已经被设置了，但同时又设置了 poll_rh，也是要采用旧机制，否则采用的就是新机制。至于 status_urb，其实就是 root hub 提交的那个 urb。

root hub 的生命线就还是说到这里吧，适可而止适可而止。

----------*/

回到 struct usb_hcd，看 92 行，wireless，是不是无线 usb。

94~97 这几行都是与主机控制器的娘家 PCI 有关的，说到 PCI 就不得不说到那张著名的表。PCI spec 说了，要想在我这儿混，谁都必须不能少了那张表。就好像木子美对一男八卦记者说的：要采访我，必须先和我上床；在床上能用多长时间，我就给你多长时间的采访。



不过强制的也不一定都是坏事儿，起码这张表不是，中断号啊等很多有用的东西都在里面准备好了，有了这张表儿，写 PCI 驱动的身体甭儿好，吃饭甭儿香。94 行的 irq 就躲在上面表儿里的倒数第四个 byte，HCD 可以直接拿来用，根本就不用再去申请，一提到申请个什么，谁都知道那是多么一个艰辛的过程。接下来的 regs，rsrc_start，rsrc_len 就与中间的那几个 Base Address0~5 脱不开关系了，牵涉到所谓的 I/O 内存和 I/O 端口，简单说一下。

大家都知道 CPU 牛 X，是众人瞩目的焦点，但是它再牛也不可能一个人战斗，电脑运转是个集体项目，只有一个卡卡是不够的，还有很多千奇百怪的外设。卡卡再厉害也不能从自己门口带球带到对方门口，他需要跟小皮，加加配合，交流，CPU 也需要跟各种外设配合交流，它需要访问外设里的那些寄存器或者内存。现在差别就出来了，主要是空间的差别，一些 CPU 芯片有两个空间，即 I/O 空间和内存空间，提供有专门访问外设 I/O 端口的指令，而另外一些只有一个空间，即内存空间。外设的 I/O 端口可以映射在 I/O 空间也可以映射

到内存空间，CPU 通过访问这两个空间来访问外设，I/O 空间有 I/O 空间访问的接口，内存空间有内存空间访问的接口。当然某些外设不但有寄存器，还有内存，也就是 I/O 内存，比如 EHCI/OHCI，它们需要映射到内存空间。但是不管映射到哪个空间，访问 I/O 端口还是 I/O 内存，CPU 必须知道它们映射后的地址，不然没有办法配合交流。

上面表里中间的那些 Base Address 保存的就是 PCI 设备里 I/O 内存或 I/O 端口的首尾位置还有长度，驱动里使用时要首先把它们给读出来，如果要映射到 I/O 空间，则要根据读到的值向系统申请 I/O 端口资源，如果要映射到内存空间，除了要申请内存资源，还要使用 ioremap 等进行映射。

96 行的 rsrc_start 和 97 行的 rsrc_len 保存的就是从表里读出来的 host controller 的 I/O 端口或内存的首地址和长度，95 行的 regs 保存的是调用 ioremap_nocache 映射后的内存地址。

98 行，power_budget，能够提供的电流。

101 行，*pool [HCD_BUFFER_POOLS]，几个 dma 池。因为 HCD_BUFFER_POOLS 在 100 行定义为 4，所以这里就表示每个主机控制器可以有 4 个 dma 池。

我们知道主机控制器是可以进行 DMA 传输的，镜头再回到前面的 struct urb，那里有两个成员 transfer_dma 和 setup_dma，当时只是说你可以使用 usb_buffer_alloc 分配好 dma 缓冲区给它们，然后再告诉 HCD 你的 urb 已经有了，HCD 就可以不用再进行复杂的 DMA 映射了，并没有提到你这个获取的 dma 缓冲区是从哪里来的。俺都暗示到这种地步了，你用屁股也能猜出来是从这里所说的 dma 池子里来的了。

混了这么久，俺对池子还是有比较美好的回忆的，还亲手搭建过线程池、数据库连接池等等池子，像线程啊数据库连接啊什么的创建销毁的时候不是比较的耗资源么，就先建个池子预先创建好一批线程什么的放里边儿，用的时候就从里面取出来，不用的时候就再放里面，用的越多就越划的来，号称将负担均分了。就像皇马那种俱乐部的会员制，你小贝大罗不是贵么，咱不怕，有千千万万个会员顶着，分到每个人头上就只相当于掉几根头发而已，人多力量大在哪儿都是适用的，哪像俺们米兰只有一个吹牛皮的老贝，小罗、埃托奥谁都想要谁又谁都没来。所以说，池子是多么的重要啊。

当然，dma 池没这么幼稚，它还有其它的内涵。一般来说 DMA 映射获得的都是以页为单位的内存，urb 需要不了这么大，如果需要比较小的 DMA 缓冲区，就离不开 DMA 池了。还是看看主机控制器的这几个池子是怎么创建的，在 buffer.c 文件里

```
40 /**
41  * hcd_buffer_create - initialize buffer pools
42  * @hcd: the bus whose buffer pools are to be initialized
43  * Context: !in_interrupt()
44  *
```

```
45  * Call this as part of initializing a host controller that uses the dma
46  * memory allocators.  It initializes some pools of dma-coherent memory that
47  * will be shared by all drivers using that controller, or returns a negative
48  * errno value on error.
49  *
50  * Call hcd_buffer_destroy() to clean up after using those pools.
51  */
52 int hcd_buffer_create(struct usb_hcd *hcd)
53 {
54         char            name[16];
55         int             i, size;
56
57         if (!hcd->self.controller->dma_mask)
58                 return 0;
59
60         for (i = 0; i < HCD_BUFFER_POOLS; i++) {
61                 if (!(size = pool_max [i]))
62                         continue;
63                 snprintf(name, sizeof name, "buffer-%d", size);
64                 hcd->pool[i] = dma_pool_create(name, hcd->self.controller,
65                                 size, size, 0);
66                 if (!hcd->pool [i]) {
67                         hcd_buffer_destroy(hcd);
68                         return -ENOMEM;
69                 }
70         }
71         return 0;
72 }
```

这里首先要判断下这个主机控制器支持不支持 DAM，如果不支持的话再创建什么 DMA 池就是纯粹无稽之谈了。如果支持 DMA，就逐个适用 dma_pool_alloc 来创建 DMA 池，如果创建失败了，就调用同一个文件里的 hcd_buffer_destroy 来将已经创建成功的池子给销毁掉。

```
75 /**
76  * hcd_buffer_destroy - deallocate buffer pools
77  * @hcd: the bus whose buffer pools are to be destroyed
78  * Context: !in_interrupt()
79  *
80  * This frees the buffer pools created by hcd_buffer_create().
81  */
82 void hcd_buffer_destroy(struct usb_hcd *hcd)
83 {
84         int             i;
```

```
85
86          for (i = 0; i < HCD_BUFFER_POOLS; i++) {
87                  struct dma_pool         *pool = hcd->pool[i];
88                  if (pool) {
89                          dma_pool_destroy(pool);
90                          hcd->pool[i] = NULL;
91                  }
92          }
93 }
```

这里调用的 dma_pool_destroy 和上面的 dma_pool_alloc 也是相映成趣，都是 DMA 池子领域的小白领。带翅膀的丘比特说了，每个人的人生都要找到四个人，第一个是自己，第二个是你最爱的人，第三个是最爱你的人，第四个是共度一生的人。他还说了，每个 DMA 池子都要找到四个函数，一个用来创建，一个用来销毁，一个用来取内存，一个用来放内存。上边儿只遇到了创建和销毁的，还少两个取和放的，再去同一个文件里找找

```
96 /* sometimes alloc/free could use kmalloc with GFP_DMA, for
97  * better sharing and to leverage mm/slab.c intelligence.
98  */
99
100 void *hcd_buffer_alloc(
101         struct usb_bus  *bus,
102         size_t                  size,
103         gfp_t                   mem_flags,
104         dma_addr_t              *dma
105 )
106 {
107         struct usb_hcd          *hcd = bus_to_hcd(bus);
108         int                     i;
109
110         /* some USB hosts just use PIO */
111         if (!bus->controller->dma_mask) {
112                 *dma = ~(dma_addr_t) 0;
113                 return kmalloc(size, mem_flags);
114         }
115
116         for (i = 0; i < HCD_BUFFER_POOLS; i++) {
117                 if (size <= pool_max [i])
118                         return dma_pool_alloc(hcd->pool [i], mem_flags, dma);
119         }
120         return dma_alloc_coherent(hcd->self.controller, size, dma, 0);
121 }
122
123 void hcd_buffer_free(
```

```
124         struct usb_bus  *bus,
125         size_t                  size,
126         void                    *addr,
127         dma_addr_t              dma
128 )
129 {
130         struct usb_hcd          *hcd = bus_to_hcd(bus);
131         int                     i;
132
133         if (!addr)
134                 return;
135
136         if (!bus->controller->dma_mask) {
137                 kfree(addr);
138                 return;
139         }
140
141         for (i = 0; i < HCD_BUFFER_POOLS; i++) {
142                 if (size <= pool_max [i]) {
143                         dma_pool_free(hcd->pool [i], addr, dma);
144                         return;
145                 }
146         }
147         dma_free_coherent(hcd->self.controller, size, addr, dma);
148 }
```

又可以找到两个dma_pool_alloc和dma_pool_free，现在可以凑齐四个了。不过咱们不用管它们四个到底什么长相，只要它们能够干活儿就成了，这里要注意的是几个问题。第一个是即使你的主机控制器不支持DMA，这几个函数也是可以用的，只不过创建的不是DMA池子，取的也不是DMA缓冲区，此时，DMA池子不存在，hcd_buffer_alloc获取的只是适用kmalloc申请的普通内存，当然相应的，你必须在它没有利用价值的时候使用hcd_buffer_free将它释放掉。

第二个问题是 size 的问题，也是每个男人女人的问题。看到它们里面都有个 pool_max 吧，这是个同一个文件里定义的数组

```
25 /* FIXME tune these based on pool statistics ... */
26 static const size_t     pool_max [HCD_BUFFER_POOLS] = {
27         /* platforms without dma-friendly caches might need to
28          * prevent cacheline sharing...
29          */
30         32,
31         128,
32         512,
```

```
33          PAGE_SIZE / 2
34          /* bigger --> allocate pages */
35 };
```

这个数组里定义的就是四个池子中每个池子里保存的DMA缓冲区的size。注意这里虽说只定义了四种size，但是并不说明你使用 hcd_buffer_alloc获取DMA缓冲区的时候不能指定更大的size，如果谁谁太贪心了，欲望太强烈了，这几个池子都满足不了她的要求，那就会使用dma_alloc_coherent为她建立一个新的DMA映射。还有，每个人的情况都不一样，不可能都会完全恰好和上面定义的四种size一致，那也不用怕，这不是病，使用这个size获取DMA缓冲区的时候，池子会选择一个略大一些的回馈过去。

还是让咱们抛开 size，回到 struct usb_hcd 中来，103 行，state，主机控制器的状态，紧挨着它的下面那些行就是相关的可用值和宏定义。咱们自己的状态还都稀里糊涂的，就先不说它们了。

126 行，如果不是有短暂失忆症或选择性失忆症的话，是会知道这是什么意思用来干吗的。

设备的生命线（九）

聊完了struct usb_hcd和struct usb_bus，算是已经向HCD片儿区的老大们拜过山头了，接下来就该看看usb_submit_urb()最后的那个遗留问题usb_hcd_submit_urb()了，要有心理准备，也是个一百多行的狠角色。现在内核里有个很不好的现象，设计结构比复杂，写函数比长。像一个中介语重心长的说：我承认的确房屋中介有不好的现象，收看房费，收差价，很是让人生气，作为业内人士我感到很心酸，但是还是有好的啊。不管怎地苦的都是我们，如果你缺少动力往下看，就去看一遍福布斯 美国富翁排行榜，如果上面没有你的名字，你就继续往下看，这是勉励俺的，也拿来与你共勉。

```
916 /* may be called in any context with a valid urb->dev usecount
917  * caller surrenders "ownership" of urb
918  * expects usb_submit_urb() to have sanity checked and conditioned all
919  * inputs in the urb
920  */
921 int usb_hcd_submit_urb (struct urb *urb, gfp_t mem_flags)
922 {
923         int                     status;
924         struct usb_hcd          *hcd = bus_to_hcd(urb->dev->bus);
925         struct usb_host_endpoint *ep;
926         unsigned long           flags;
927
928         if (!hcd)
```

```
929                 return -ENODEV;
930
931         usbmon_urb_submit(&hcd->self, urb);
932
933         /*
934          * Atomically queue the urb,  first to our records, then to the HCD.
935          * Access to urb->status is controlled by urb->lock ... changes on
936          * i/o completion (normal or fault) or unlinking.
937          */
938
939         // FIXME:  verify that quiescing hc works right (RH cleans up)
940
941         spin_lock_irqsave (&hcd_data_lock, flags);
942         ep = (usb_pipein(urb->pipe) ? urb->dev->ep_in : urb->dev->ep_out)
943                         [usb_pipeendpoint(urb->pipe)];
944         if (unlikely (!ep))
945                 status = -ENOENT;
946         else if (unlikely (urb->reject))
947                 status = -EPERM;
948         else switch (hcd->state) {
949         case HC_STATE_RUNNING:
950         case HC_STATE_RESUMING:
951 doit:
952                 list_add_tail (&urb->urb_list, &ep->urb_list);
953                 status = 0;
954                 break;
955         case HC_STATE_SUSPENDED:
956                 /* HC upstream links (register access, wakeup signaling) can
work
957                  * even when the downstream links (and DMA etc) are quiesced;
let
958                  * usbcore talk to the root hub.
959                  */
960                 if (hcd->self.controller->power.power_state.event ==
PM_EVENT_ON
961                                 && urb->dev->parent == NULL)
962                         goto doit;
963                 /* FALL THROUGH */
964         default:
965                 status = -ESHUTDOWN;
966                 break;
967         }
968         spin_unlock_irqrestore (&hcd_data_lock, flags);
969         if (status) {
```

```
970                 INIT_LIST_HEAD (&urb->urb_list);
971                 usbmon_urb_submit_error(&hcd->self, urb, status);
972                 return status;
973         }
974
975         /* increment urb's reference count as part of giving it to the HCD
976          * (which now controls it).  HCD guarantees that it either returns
977          * an error or calls giveback(), but not both.
978          */
979         urb = usb_get_urb (urb);
980         atomic_inc (&urb->use_count);
981
982         if (urb->dev == hcd->self.root_hub) {
983                 /* NOTE:  requirement on hub callers (usbfs and the hub
984                  * driver, for now) that URBs' urb->transfer_buffer be
985                  * valid and usb_buffer_{sync,unmap}() not be needed, since
986                  * they could clobber root hub response data.
987                  */
988                 status = rh_urb_enqueue (hcd, urb);
989                 goto done;
990         }
991
992         /* lower level hcd code should use *_dma exclusively,
993          * unless it uses pio or talks to another transport.
994          */
995         if (hcd->self.uses_dma) {
996                 if (usb_pipecontrol (urb->pipe)
997                         && !(urb->transfer_flags & URB_NO_SETUP_DMA_MAP))
998                         urb->setup_dma = dma_map_single (
999                                         hcd->self.controller,
1000                                         urb->setup_packet,
1001                                         sizeof (struct usb_ctrlrequest),
1002                                         DMA_TO_DEVICE);
1003                  if (urb->transfer_buffer_length != 0
1004                          && !(urb->transfer_flags &
URB_NO_TRANSFER_DMA_MAP))
1005                          urb->transfer_dma = dma_map_single (
1006                                         hcd->self.controller,
1007                                         urb->transfer_buffer,
1008                                         urb->transfer_buffer_length,
1009                                         usb_pipein (urb->pipe)
1010                                             ? DMA_FROM_DEVICE
1011                                             : DMA_TO_DEVICE);
1012         }
```

```
1013
1014         status = hcd->driver->urb_enqueue (hcd, ep, urb, mem_flags);
1015 done:
1016         if (unlikely (status)) {
1017                 urb_unlink (urb);
1018                 atomic_dec (&urb->use_count);
1019                 if (urb->reject)
1020                         wake_up (&usb_kill_urb_queue);
1021                 usbmon_urb_submit_error(&hcd->self, urb, status);
1022                 usb_put_urb (urb);
1023         }
1024         return status;
1025 }
```

usb_hcd_submit_urb 是 hcd.c 里的，目标也很明确，就是将提交过来的 urb 指派给合适的主机控制器驱动程序。core 目录下面以 hcd 打头的几个文件严格来说不能算是 HCD，只能算 HCDI，即主机控制器驱动的接口层，用来衔接具体的主机控制器驱动和 usb core 的。

924 行，bus_to_hcd 在哪里提到过一下，是用来获得 struct usb_bus 结构体对应的 struct usb_hcd 结构体，urb 要去的那个设备所在的总线是在设备生命线的开头儿就初始化好了的，忘了可以再蓦然回首一下。bus_to_hcd 还有个兄弟 hcd_to_bus，都在 hcd.h 里定义

```
131 static inline struct usb_bus *hcd_to_bus (struct usb_hcd *hcd)
132 {
133         return &hcd->self;
134 }
135
136 static inline struct usb_hcd *bus_to_hcd (struct usb_bus *bus)
137 {
138         return container_of(bus, struct usb_hcd, self);
139 }
```

这俩函数傻强都能看懂，继续看 928 行，也是大多数函数开头儿必备的常规检验，如果 usb_hcd 都还是空的，那就别开国际玩笑了，返回吧。

931 行，usbmon_urb_submit 就是与前面 Greg 孕育出来的 usb Monitor 有关的，如果你编译内核的时候没有配置上 CONFIG_USB_MON，它就啥也不是，一个空函数，一具空壳。

941 行，去获得一把锁，这把锁在 hcd.c 的开头儿就已经初始化好了，所以说是把全局锁

```
102 /* used when updating hcd data */
```

```
103 static DEFINE_SPINLOCK(hcd_data_lock);
```

前边儿多次遇到过自旋锁，不过一直没功夫说它，现在就简单介绍一下。它和信号量，还有前面提到的 completion 一样都是 linux 里用来进行代码同步的，为什么要进行同步？你要知道在 linux 这个庞大负责的世界里，不是只有你一个人在战斗，可能同时有多个线程，多杆枪在战斗，那么只要他们互相之间有一定的共享，就必须要保证一个人操作这个共享的时候让其它人知道，这么说吧，你和另外一个人带领两只队伍去打土匪窝，结果你方武力值，智力值比较高，先占领了山头儿，你得插把旗子表示你已经占领了，让友军不要再打了，过来享受胜利果实吧，里边土匪太太，美酒美食一堆一堆的，如果你什么都不说，没插旗子也没其它什么暗号表示一下，只顾自个儿享受了，那边儿正打着过瘾谁知道在匪窝儿里的是你还是土匪啊。所以说同步是多重要啊，保持共享代码的状态一致是多么重要啊。

自旋锁身为同步机制的一种，自然也有它独特的本事，它可以用在中断上下文或者说原子上下文使用。上下文就是你代码运行的环境，linux 的这个环境使用二分法可以分成两种，能睡觉的环境和不能睡觉的环境。像信号量和 completion 就只能用在可以睡觉的环境，而自旋锁就用在不能睡觉的环境里。而咱们的 usb_submit_urb 还有 usb_hcd_submit_urb 必须得在两种环境里都能够使用，所以使用的是自旋锁，那在什么时候都不能睡觉了还有心情去调用它们那？想想 urb 的那个结束处理函数，它就是不能睡觉的，但它里面必须得能够重新提交 urb。

那再说说 hcd_data_lock 这把锁都是用来保护些什么的，为什么要使用它？主机控制器的 struct usb_hcd 结构体在它的驱动里早就初始化好了，就那么一个，但同一时刻是可能有多个 urb 向同一主机控制器申请进行传输，可能有多个地方都希望访问它里面的内容的，比如 948 行的 state 元素，显然就要同步了，hcd_data_lock 这把锁就是专门用来保护主机控制器的这个结构体的。

942 行，遇到多次也说过多次了，知道了 pipe，就可以从 struct usb_device 里的两个数组 ep_in 和 ep_out 里拿出对应端点的 struct usb_host_endpoint 结构体。写代码的哥们儿也知道咱们说过多次了，就直接把这一堆搞一行里了。

944~973 这些行都是做检验的，写个代码真费劲儿，到处都是地雷暗礁，到处都要检验。就好像去年以前还处于公粮时代的时候，那些验粮的，要一关又一关的，发生了多少可歌可泣的故事啊。验粮的都要牙好，写代码看代码的当然都要耐心好了。

944 行，显然都走到这一步了，目的端点为空的可能性太小了，所以加上了 unlikely。

946 行，前面还费了点口舌说过，urb 里的这个 reject，只有 usb_kill_urb 有特权修改它，如果走到这里发现它的值大于 0 了，那就说明哪里调用了 usb_kill_urb 要终止这次传输，所以就还是返回吧，不过这种可能性比较小，没人无聊到那种地步，总是刚提交就终止，吊主机控制器胃口，所以仍然加上unlikely。

948 行，如果上面那两个检验都通过了，现在就 case 一下主机控制器的状态，如果为 HC_STATE_RUNNING 或 HC_STATE_RESUMING 就说明主机控制器这边儿没问题，尽管将这个 urb 往端点的那个 urb 队列里塞好了，952 行就是完成这个的，struct urb 那么变态的结构都熬过来了，这行是小 case 了。如果主机控制器的状态为 HC_STATE_SUSPENDED，但它的上行链路能够工作，且这个 urb 是送往 root hub 的，则将其塞到 root hub 的 urb 队列里。

然后判断上面几次检验的结果，如果一切正常，则继续往下走，否则就黯然回头吧。

979 行，检验都通过了，可以放心的增加 urb 的引用计数了。

980 行，将 urb 的 use_count 也增加 1，表示 urb 已经被 HCD 接受了，正在被处理着。你如果对这两个引用计数什么差别还有疑问，再蓦然回首一下看看说 struct urb 时讲的。

982 行，判断这个 urb 是不是流向 root hub 的，如果是，它就走向了 root hub 的生命线。不过，毕竟你更关注的是你的 usb 设备，应该很少有机会和欲望直接和 root hub 交流些什么。

995 行，如果这个主机控制器支持 DMA，可你却没有告诉它 URB_NO_SETUP_DMA_MAP 或 URB_NO_TRANSFER_DMA_MAP 这两个标志，它就会认为你在 urb 里没有提供 DMA 的缓冲区，就会调用 dma_map_single 将 setup_packet 或 transfer_buffer 映射为 DMA 缓冲区。

1014 行，终于可以将 urb 扔给具体的主机控制器驱动程序了，urb 可以欢快的尽情呼喊，UHCI，OHCI，EHCI，我来了！

下面的路就让 urb 去走吧，咱们说到这里也该回头了，经过了这么多事，遇到了这么多人，我始终都不能忘怀自己是从设置设备地址，发送 SET_ADDRESS 请求给主机控制器开始，这么一路走过来的，到现在，设备已经可以进入 Address 状态，这桩心愿已了，该继续看设备的那条生命线了。

设备的生命线（十）

经过了许多事

你是不是觉得累

这样的心情

我曾有过几回

现在的你我想一定

很疲惫

内核代码就象酒

有的苦有的烈

这样的滋味

你我早晚要体会

把那内核当作一场宿醉

明日的代码莫再要装着昨天的伤悲

请与我举起杯

跟内核干杯

跟着设备的生命线走到现在，我算是明白了，什么东西的发展都是越往后越高级越复杂，就好像人一样，从 Attached 走到 Powered 只是弹指一挥间，从 Powered 再到 Default 虽说要复位一下，也算是三下五除二了，再从 Default 走到 Address 简直练吃奶劲儿都使出来了，应该把阿 Q 拉过来念叨两句“Address？有趣！来了一群鬼佬，叫到，Address，Address，于是就 Address 了。”再给张小表，看看现在和上次那张表出现的时候有什么变化。

state	**USB_STATE_ADDRESS**
speed	**taken**
ep0	**ep0.urb_list，描述符长度/类型，wMaxPacketSize**

接下来设备的目标当然就是Configured了，My god！又要经过多少事，遇到多少人？如果实在觉得辛苦，可以去穿件绿衣服，因为 忍者 神龟先生说了：要想生活过得去，背上就得带点绿！

要进入 Configured 状态，你得去配置设备，当然不能是盲目的去配置，要知道设备是可能有多个配置的，所以你要有选择有目的有步骤有计划的去配置，要做这样一个四有新人，就要先去获得设备的设备描述符，message.c 中的 usb_get_device_descriptor()就是 core 里专门干这个的。

```
842 /*
843  * usb_get_device_descriptor - (re)reads the device descriptor (usbcore)
844  * @dev: the device whose device descriptor is being updated
845  * @size: how much of the descriptor to read
```

```
 * Context: !in_interrupt ()
 *
 * Updates the copy of the device descriptor stored in the device structure,
 * which dedicates space for this purpose.
 *
 * Not exported, only for use by the core.  If drivers really want to read
 * the device descriptor directly, they can call usb_get_descriptor() with
 * type = USB_DT_DEVICE and index = 0.
 *
 * This call is synchronous, and may not be used in an interrupt context.
 *
 * Returns the number of bytes received on success, or else the status code
 * returned by the underlying usb_control_msg() call.
 */
int usb_get_device_descriptor(struct usb_device *dev, unsigned int size)
{
        struct usb_device_descriptor *desc;
        int ret;

        if (size > sizeof(*desc))
                return -EINVAL;
        desc = kmalloc(sizeof(*desc), GFP_NOIO);
        if (!desc)
                return -ENOMEM;

        ret = usb_get_descriptor(dev, USB_DT_DEVICE, 0, desc, size);
        if (ret >= 0)
                memcpy(&dev->descriptor, desc, size);
        kfree(desc);
        return ret;
}
```

这个函数比较的精悍，先是准备了一个 struct usb_device_descriptor 结构体，然后就用它去调用 message.c 里的 usb_get_descriptor()获得设备描述符，获得之后再把得到的描述符复制到设备 struct usb_device 结构体的 descriptor 成员里。因此，这个函数成功与否的关键就在 usb_get_descriptor()。其实对于写驱动的来说，眼里是只有 usb_get_descriptor()没有 usb_get_device_descriptor()的，不管你想获得哪种描述符都是要通过 usb_get_descriptor()，而 usb_get_device_descriptor()是专属内核用的接口。

```
/**
 * usb_get_descriptor - issues a generic GET_DESCRIPTOR request
 * @dev: the device whose descriptor is being retrieved
 * @type: the descriptor type (USB_DT_*)
```

```
600  * @index: the number of the descriptor
601  * @buf: where to put the descriptor
602  * @size: how big is "buf"?
603  * Context: !in_interrupt ()
604  *
605  * Gets a USB descriptor.  Convenience functions exist to simplify
606  * getting some types of descriptors.  Use
607  * usb_get_string() or usb_string() for USB_DT_STRING.
608  * Device (USB_DT_DEVICE) and configuration descriptors (USB_DT_CONFIG)
609  * are part of the device structure.
610  * In addition to a number of USB-standard descriptors, some
611  * devices also use class-specific or vendor-specific descriptors.
612  *
613  * This call is synchronous, and may not be used in an interrupt context.
614  *
615  * Returns the number of bytes received on success, or else the status code
616  * returned by the underlying usb_control_msg() call.
617  */
618 int usb_get_descriptor(struct usb_device *dev, unsigned char type, unsigned
char index, void *buf, int size)
619 {
620         int i;
621         int result;
622
623         memset(buf,0,size);     // Make sure we parse really received data
624
625         for (i = 0; i < 3; ++i) {
626                 /* retry on length 0 or stall; some devices are flakey */
627                 result = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
628                                 USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
629                                 (type << 8) + index, 0, buf, size,
630                                 USB_CTRL_GET_TIMEOUT);
631                 if (result == 0 || result == -EPIPE)
632                         continue;
633                 if (result > 1 && ((u8 *)buf)[1] != type) {
634                         result = -EPROTO;
635                         continue;
636                 }
637                 break;
638         }
639         return result;
640 }
```

参数 type 就是用来区分不同的描述符的，协议里说了，GET_DESCRIPTOR 请求主要就

是适用于三种描述符，设备描述符，配置描述符和字符串描述符。参数 index 是要获得的描述符的序号，如果希望得到的这种描述符设备里可以有多个，你需要指定获得其中的哪个，比如配置描述符就可以有多个，不过对于设备描述符来说，是只有一个的，所以这里的 index 应该为 0。参数 buf 和 size 就是描述你用来放置获得的描述符的缓冲区的。

这个函数的内容挺单调的，主要就是调用了一个 usb_control_msg()，你如果到现在还觉得 usb_control_msg()只是个熟悉的陌生人，那俺也就太失败了。这里要说的第一个问题是它的一堆参数，这就需要认真了解一下 spec 9.4.3 里的这张表

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID (refer to Section 9.6.7)	Descriptor Length	Descriptor

GET_DESCRIPTOR 请求的数据传输方向很明显是 device-to-host 的，而且还是协议里规定所有设备都要支持的标准请求，也不是针对端点或者接口什么的，而是针对设备的，所以 bRequestType 只能为 0x80，就是上面表里的 10000000B，也等于 628 行的 USB_DIR_IN。wValue 的高位字节表示描述符的类型，低位字节表示描述符的序号，所以就有 629 行的(type << 8) + index。wIndex 对于字符串描述符应该设置为使用语言的 ID，对于其它的描述符应该设置为 0，所以也有了 629 行中间的那个 0。至于 wLength，就是描述符的长度，对于设备描述符，一般来说你都会指定为 USB_DT_DEVICE_SIZE 吧。

USB_CTRL_GET_TIMEOUT 是定义在 include/linux/usb.h 里的一个宏，值为 5000，表示有 5s 的超时时间。

```
1324 /*
1325  * timeouts, in milliseconds, used for sending/receiving control messages
1326  * they typically complete within a few frames (msec) after they're issued
1327  * USB identifies 5 second timeouts, maybe more in a few cases, and a few
1328  * slow devices (like some MGE Ellipse UPSes) actually push that limit.
1329  */
1330 #define USB_CTRL_GET_TIMEOUT    5000
1331 #define USB_CTRL_SET_TIMEOUT    5000
```

第二个问题就是为什么会有 3 次循环。这个又要归咎于一些不守规矩的厂商了，搞出的设备古里古怪的，比如一些 usb 读卡器，一次请求还不定能成功，但是设备描述符拿不到接下来就没法子走了，所以这里多试几次，再不成功，就成鬼了。至于 631 到 636 行之间的代码都是判断是不是成功得到请求的描述符的，这个版本的内核这里的判断还比较混乱，就不多说了，你只要知道((u8 *)buf)[1] != type 是用来判断获得描述符是不是请求的类型就可以了。

现在设备描述符已经有了，但是只有设备描述符是远远不够的，你从设备描述符里只能知道它一共支持几个配置，具体每个配置是何方神圣，是公的还是母的都不知道，你要配置一个设备总得知道这些吧，总不能学李湘说“其实新郎是谁并不重要”，那种酷劲儿不是人人都能学来的。所以接下来就要获得各个配置的配置描述符，并且拿结果去充实 struct usb_device 的 config、rawdescriptors 等相关元素。core 内部并不直接调用上面的 usb_get_descriptor() 去完成这个任务，而是调用 config.c 里的 usb_get_configuration()，为什么？core 总是需要做更多的事情，不然就不叫 core 了。

```
474 // hub-only!! ... and only in reset path, or usb_new_device()
475 // (used by real hubs and virtual root hubs)
476 int usb_get_configuration(struct usb_device *dev)
477 {
478         struct device *ddev = &dev->dev;
479         int ncfg = dev->descriptor.bNumConfigurations;
480         int result = -ENOMEM;
481         unsigned int cfgno, length;
482         unsigned char *buffer;
483         unsigned char *bigbuffer;
484         struct usb_config_descriptor *desc;
485
486         if (ncfg > USB_MAXCONFIG) {
487                 dev_warn(ddev, "too many configurations: %d, "
488                     "using maximum allowed: %d\n", ncfg, USB_MAXCONFIG);
489                 dev->descriptor.bNumConfigurations = ncfg = USB_MAXCONFIG;
490         }
491
492         if (ncfg < 1) {
493                 dev_err(ddev, "no configurations\n");
494                 return -EINVAL;
495         }
496
497         length = ncfg * sizeof(struct usb_host_config);
498         dev->config = kzalloc(length, GFP_KERNEL);
499         if (!dev->config)
500                 goto err2;
501
502         length = ncfg * sizeof(char *);
503         dev->rawdescriptors = kzalloc(length, GFP_KERNEL);
504         if (!dev->rawdescriptors)
505                 goto err2;
506
507         buffer = kmalloc(USB_DT_CONFIG_SIZE, GFP_KERNEL);
```

```
508         if (!buffer)
509                 goto err2;
510         desc = (struct usb_config_descriptor *)buffer;
511
512         for (cfgno = 0; cfgno < ncfg; cfgno++) {
513                 /* We grab just the first descriptor so we know how long
514                  * the whole configuration is */
515                 result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno,
516                     buffer, USB_DT_CONFIG_SIZE);
517                 if (result < 0) {
518                         dev_err(ddev, "unable to read config index %d "
519                             "descriptor/%s\n", cfgno, "start");
520                         dev_err(ddev, "chopping to %d config(s)\n", cfgno);
521                         dev->descriptor.bNumConfigurations = cfgno;
522                         break;
523                 } else if (result < 4) {
524                         dev_err(ddev, "config index %d descriptor too short "
525                             "(expected %i, got %i)\n", cfgno,
526                             USB_DT_CONFIG_SIZE, result);
527                         result = -EINVAL;
528                         goto err;
529                 }
530                 length = max((int) le16_to_cpu(desc->wTotalLength),
531                     USB_DT_CONFIG_SIZE);
532
533                 /* Now that we know the length, get the whole thing */
534                 bigbuffer = kmalloc(length, GFP_KERNEL);
535                 if (!bigbuffer) {
536                         result = -ENOMEM;
537                         goto err;
538                 }
539                 result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno,
540                     bigbuffer, length);
541                 if (result < 0) {
542                         dev_err(ddev, "unable to read config index %d "
543                             "descriptor/%s\n", cfgno, "all");
544                         kfree(bigbuffer);
545                         goto err;
546                 }
547                 if (result < length) {
548                         dev_warn(ddev, "config index %d descriptor too short
"
549                             "(expected %i, got %i)\n", cfgno, length, result);
550                         length = result;
```

```
551                 }
552
553                 dev->rawdescriptors[cfgno] = bigbuffer;
554
555                 result = usb_parse_configuration(&dev->dev, cfgno,
556                     &dev->config[cfgno], bigbuffer, length);
557                 if (result < 0) {
558                         ++cfgno;
559                         goto err;
560                 }
561         }
562         result = 0;
563
564 err:
565         kfree(buffer);
566         dev->descriptor.bNumConfigurations = cfgno;
567 err2:
568         if (result == -ENOMEM)
569                 dev_err(ddev, "out of memory\n");
570         return result;
571 }
```

说代码前先说点理论，不然要被这么生猛的代码给吓倒了。不管过多少河拐几道弯，要想得到配置描述符，最终都不可避免的要向设备发送 GET_DESCRIPTOR 请求，这就需要以 USB_DT_CONFIG 为参数调用 usb_get_descriptor 函数，也就需要知道该为获得的描述符准备多大的一个缓冲区，本来这个长度应该很明确的为 USB_DT_CONFIG_SIZE，它表示的就是配置描述符的大小，但是实际上不是这么回事儿，USB_DT_CONFIG_SIZE 只表示配置描述符本身的大小，并不表示 GET_DESCRIPTOR 请求返回结果的大小。因为向设备发送 GET_DESCRIPTOR 请求时，设备并不单单返回一个配置描述符了事，而是一股脑儿的将这个配置下面的所有接口描述符，端点描述，还有 class-或 vendor-specific 描述符都返回了给你，这要比商场里那些买 300 送 100 的优惠力度大得多。那么这个总长度如何得到那？在神秘的配置描述符里有这样一个神秘的字段 wTotalLength，它里面记录的就是这个总长度，那么问题就简单了，可以首先发送 USB_DT_CONFIG_SIZE 个字节的请求过去，获得这个配置描述符的内容，从而获得那个总长度，然后以这个长度再请求一次，这样就可以获得一个配置下面所有的描述符内容了。上面的 usb_get_configuration() 采用的就是这个处理方法。

479 行，获得设备理配置描述符的数目。

486 行，这些检验又来了，在光天化日之下莫明其妙的受到戴大盖帽的盘问很不爽是吧，但这就是他们的规矩他们的工作，不然你让他们做什么。USB_MAXCONFIG 是 config.c 理定义的

```
14 #define USB_MAXCONFIG                    8       /* Arbitrary limit */
```

限制了一个设备最多只能支持 8 种配置拥有 8 个配置描述符，如果超出了这个限制，489 行就强制它为这个最大值，你一个设备要想在 linux 里混就得守这里的规矩，自由民主只是相对的。不过如果设备里没有任何一个配置描述符，什么配置都没有，就想裸身蒙混过关，那是不可能的，492 行这关就过不去，你设备赤身裸体没错，可是拿出来给人看就有错了，大白天在外滩喷泉里洗澡，以为自己是那个欲望主妇里的伊娃可以随便露啊，不是影响市容影响民风影响上海美好形象么。

498 行，struct usb_device 里的 config 表示的是设备拥有的所有配置，你设备有多少个配置就为它准备多大的空间。

503 行，rawdescriptors 还认识吧，这是个字符指针数组里的每一项都指向一个使用 GET_DESCRIPTOR 请求去获取配置描述符时所得到的结果。

507 行，准备一个大小为 USB_DT_CONFIG_SIZE 的缓冲区，第一次发送 GET_DESCRIPTOR 请求要用的。

512 行，剩下的主要就是这个 for 循环了，获取每一个配置的那些描述符。

515 行，诚如上面所说的，首先发送 USB_DT_CONFIG_SIZE 个字节请求，获得配置描述符的内容。然后对返回的结果进行检验，知道为什么 523 行会判断结果是不是小于 4 么？答案尽在配置描述符中，里面的 3，4 字节就是 wTotalLength，只要得到前 4 个字节，就已经完成任务能够获得总长度了。

534 行，既然总长度已经有了，那么这里就为接下来的 GET_DESCRIPTOR 请求准备一个大点的缓冲区。

539 行，现在可以获得这个配置相关的所有描述符了。然后是对返回结果的检验，再然后就是将得到的那一堆数据的地址赋给 rawdescriptors 数组里的指针。

555 行，从这个颇有韵味的数字 555 开始，你将会遇到另一个超级变态的函数，它将对前面 GET_DESCRIPTOR 请求获得的那堆数据做处理。

设备的生命线（十一）

现在已经使用 GET_DESCRIPTOR 请求取到了包含一个配置里所有相关描述符内容的一堆数据，这些数据是 raw 的，即原始的，所有数据不管是配置描述符、接口描述符还是端点描述符都不分男女不分彼此的挤在一起，这放在今天当然是有伤风化的，再说群租也是要禁

止的，所以得想办法将它们给分开，丁是丁卯是卯的，于是 usb_parse_configuration() 和上海的那个群租管理条例一起登上了历史舞台，显然它们两个不管是谁想简短几句就搞定是不可能的，不过也没什么可怕的，咱写不会，看还不会么？和 mm 打交道，要记住一点：做不到健谈，就装酷，说话不会，闭嘴还不会么？

```
264 static int usb_parse_configuration(struct device *ddev, int cfgidx,
265     struct usb_host_config *config, unsigned char *buffer, int size)
266 {
267         unsigned char *buffer0 = buffer;
268         int cfgno;
269         int nintf, nintf_orig;
270         int i, j, n;
271         struct usb_interface_cache *intfc;
272         unsigned char *buffer2;
273         int size2;
274         struct usb_descriptor_header *header;
275         int len, retval;
276         u8 inums[USB_MAXINTERFACES], nalts[USB_MAXINTERFACES];
277
278         memcpy(&config->desc, buffer, USB_DT_CONFIG_SIZE);
279         if (config->desc.bDescriptorType != USB_DT_CONFIG ||
280             config->desc.bLength < USB_DT_CONFIG_SIZE) {
281                 dev_err(ddev, "invalid descriptor for config index %d: "
282                     "type = 0x%X, length = %d\n", cfgidx,
283                     config->desc.bDescriptorType, config->desc.bLength);
284                 return -EINVAL;
285         }
286         cfgno = config->desc.bConfigurationValue;
287
288         buffer += config->desc.bLength;
289         size -= config->desc.bLength;
290
291         nintf = nintf_orig = config->desc.bNumInterfaces;
292         if (nintf > USB_MAXINTERFACES) {
293                 dev_warn(ddev, "config %d has too many interfaces: %d, "
294                     "using maximum allowed: %d\n",
295                     cfgno, nintf, USB_MAXINTERFACES);
296                 nintf = USB_MAXINTERFACES;
297         }
298
299         /* Go through the descriptors, checking their length and counting the
300          * number of altsettings for each interface */
301         n = 0;
302         for ((buffer2 = buffer, size2 = size);
```

```
303                size2 > 0;
304               (buffer2 += header->bLength, size2 -= header->bLength)) {
305
306                   if (size2 < sizeof(struct usb_descriptor_header)) {
307                           dev_warn(ddev, "config %d descriptor has %d excess "
308                               "byte%s, ignoring\n",
309                               cfgno, size2, plural(size2));
310                           break;
311                   }
312
313                   header = (struct usb_descriptor_header *) buffer2;
314                   if ((header->bLength > size2) || (header->bLength < 2)) {
315                           dev_warn(ddev, "config %d has an invalid descriptor "
316                               "of length %d, skipping remainder of the
config\n",
317                               cfgno, header->bLength);
318                           break;
319                   }
320
321                   if (header->bDescriptorType == USB_DT_INTERFACE) {
322                           struct usb_interface_descriptor *d;
323                           int inum;
324
325                           d = (struct usb_interface_descriptor *) header;
326                           if (d->bLength < USB_DT_INTERFACE_SIZE) {
327                                   dev_warn(ddev, "config %d has an invalid "
328                                       "interface descriptor of length %d, "
329                                       "skipping\n", cfgno, d->bLength);
330                                   continue;
331                           }
332
333                           inum = d->bInterfaceNumber;
334                           if (inum >= nintf_orig)
335                                   dev_warn(ddev, "config %d has an invalid "
336                                       "interface number: %d but max is %d\n",
337                                       cfgno, inum, nintf_orig - 1);
338
339                           /* Have we already encountered this interface?
340                            * Count its altsettings */
341                           for (i = 0; i < n; ++i) {
342                                   if (inums[i] == inum)
343                                           break;
344                           }
345                           if (i < n) {
```

```
346                             if (nalts[i] < 255)
347                                     ++nalts[i];
348                     } else if (n < USB_MAXINTERFACES) {
349                             inums[n] = inum;
350                             nalts[n] = 1;
351                             ++n;
352                     }
353
354             } else if (header->bDescriptorType == USB_DT_DEVICE ||
355                         header->bDescriptorType == USB_DT_CONFIG)
356                     dev_warn(ddev, "config %d contains an unexpected "
357                         "descriptor of type 0x%X, skipping\n",
358                         cfgno, header->bDescriptorType);
359
360         }       /* for ((buffer2 = buffer, size2 = size); ...) */
361         size = buffer2 - buffer;
362         config->desc.wTotalLength = cpu_to_le16(buffer2 - buffer0);
363
364         if (n != nintf)
365                dev_warn(ddev, "config %d has %d interface%s, different from "
366                     "the descriptor's value: %d\n",
367                     cfgno, n, plural(n), nintf_orig);
368         else if (n == 0)
369                 dev_warn(ddev, "config %d has no interfaces?\n", cfgno);
370         config->desc.bNumInterfaces = nintf = n;
371
372         /* Check for missing interface numbers */
373         for (i = 0; i < nintf; ++i) {
374                 for (j = 0; j < nintf; ++j) {
375                         if (inums[j] == i)
376                                 break;
377                 }
378                 if (j >= nintf)
379                         dev_warn(ddev, "config %d has no interface number "
380                             "%d\n", cfgno, i);
381         }
382
383         /* Allocate the usb_interface_caches and altsetting arrays */
384         for (i = 0; i < nintf; ++i) {
385                 j = nalts[i];
386                 if (j > USB_MAXALTSETTING) {
387                         dev_warn(ddev, "too many alternate settings for "
388                             "config %d interface %d: %d, "
389                             "using maximum allowed: %d\n",
```

```
390                         cfgno, inums[i], j, USB_MAXALTSETTING);
391                 nalts[i] = j = USB_MAXALTSETTING;
392         }
393
394         len = sizeof(*intfc) + sizeof(struct usb_host_interface) * j;
395         config->intf_cache[i] = intfc = kzalloc(len, GFP_KERNEL);
396         if (!intfc)
397                 return -ENOMEM;
398         kref_init(&intfc->ref);
399 }
400
401 /* Skip over any Class Specific or Vendor Specific descriptors;
402  * find the first interface descriptor */
403 config->extra = buffer;
404 i = find_next_descriptor(buffer, size, USB_DT_INTERFACE,
405     USB_DT_INTERFACE, &n);
406 config->extralen = i;
407 if (n > 0)
408         dev_dbg(ddev, "skipped %d descriptor%s after %s\n",
409             n, plural(n), "configuration");
410 buffer += i;
411 size -= i;
412
413 /* Parse all the interface/altsetting descriptors */
414 while (size > 0) {
415         retval = usb_parse_interface(ddev, cfgno, config,
416             buffer, size, inums, nalts);
417         if (retval < 0)
418                 return retval;
419
420         buffer += retval;
421         size -= retval;
422 }
423
424 /* Check for missing altsettings */
425 for (i = 0; i < nintf; ++i) {
426         intfc = config->intf_cache[i];
427         for (j = 0; j < intfc->num_altsetting; ++j) {
428                 for (n = 0; n < intfc->num_altsetting; ++n) {
429                         if (intfc->altsetting[n].desc.
430                             bAlternateSetting == j)
431                                 break;
432                 }
433                 if (n >= intfc->num_altsetting)
```

```
434                                 dev_warn(ddev, "config %d interface %d has no "
435                                         "altsetting %d\n", cfgno, inums[i], j);
436                 }
437         }
438
439         return 0;
440 }
```

代码太生猛了，还是先说点理论垫垫底儿，其实前面也说到过的，使用 GET_DESCRIPTOR 请求时，得到的数据并不是杂乱无序的，而是有规可循的，一般来说，配置描述符后面跟的是第一个接口的接口描述符，接着是这个接口里第一个端点的端点描述符，如果有 class-和 vendor-specific 描述符的话，会紧跟在对应的标准描述符后面，不管接口有多少端点有多少都是按照这个规律顺序排列。当然有些厂商会特立独行一些，非要先返回第二个接口然后再返回第一个接口，但配置描述符后面总归先是接口描述符再是端点描述符。

267 行，buffer 里保存的就是 GET_DESCRIPTOR 请求获得的那堆数据，要解析这些数据，不可避免的要对 buffer 指针进行操作，这里先将它备份一下。

278 行，config 是参数里传递过来的，是设备 struct usb_device 结构体里的 struct usb_host_config 结构体数组 config 中的一员。不出意外的话 buffer 的前 USB_DT_CONFIG_SIZE 个字节对应的就是配置描述符，那么这里的意思就很明显了。然后做些检验，看看这 USB_DT_CONFIG_SIZE 字节的内容究竟是不是正如我们所期待的那样是个配置描述符，如果不是，那 buffer 里的数据问题可就大了，没什么利用价值了，还是返回吧，不必要再接着解析了。小心谨慎点总是没错的，毛主席教导我们要时刻保持革命斗争警惕性。

288 行，buffer 的前 USB_DT_CONFIG_SIZE 个字节已经理清了，接下来该解析剩下的数据了，buffer 需要紧跟形势的发展，位置和长度都要做相应的修正。

291 行，获得这个配置所拥有的接口数目，不能简单一赋值就完事儿了，得知道系统里对这个数目是有个 USB_MAXINTERFACES 这样的限制的。世青赛限制了年龄必须在 20 岁以下，大牌们想参加怎么办，改年龄啊，把年龄改成 20 不就符合标准了，这里也是，如果数目比这个限制还大，就改为 USB_MAXINTERFACES。

302~360 行，这函数真是酷到家了，连里面一个循环都这么长这么酷，不过别看它 cool，完成的事情却很单一，就是统计记录一下这个配置里每个接口所拥有的设置数目。俺喝水只喝纯净水，牛奶只喝纯牛奶，所以俺很单纯，也很善良，所以这里会提醒你一下，千万别被写代码的哥们儿给迷惑了，这个循环里使用的是 buffer2 和 size2， buffer 和 size 的两个替身，专门拍飚车跳崖什么刺激镜头的角色，拍完了就得收拾行李走人连演员列表都进不去淹死了也没人给你道歉的那种，buffer 和 size 就停在 302 行享受阳光海滩，等着拍下面的吻戏床戏，同样很刺激的那种镜头。

306 行，这里遇到一个新的结构 struct usb_descriptor_header，在 include/linux/usb/ch9.h 里定义

```
194 /* All standard descriptors have these 2 fields at the beginning */
195 struct usb_descriptor_header {
196         __u8  bLength;
197         __u8  bDescriptorType;
198 } __attribute__ ((packed));
```

这个结构比俺还单纯，就包括了两个成员，你研究一下所有的那些标准描述符，会兴奋的发现它们的前两个字节都是一样的，一个表示描述符的长度，一个表示描述符的类型，就好像你顺着族谱向前研究那么几十代几百代，兴奋的发现自己原来和贝克汉姆巴菲特是同一个祖先一样，但是如果认为自己可以像贝克汉姆那样踢球像巴菲特那样炒股那就错了。那么为什么要专门搞这么一个结构？试想一下，有块数据缓冲区，让你判断一下里面保存的是哪个描述符，或者是其它什么东西，你怎么做？你当然可以直接将它的前两个字节内容读出来，判断判断 bDescriptorType，再判断判断 bLength，不过这样的代码就好像你自己画的一副抽象画，太艺术化了，过个若干年自己都不知道啥意思，更别说别人了，你纵使不能像 linus 那样写程序，可也别不拿豆包当干粮不拿看你代码的人当回事儿。写 C 不是写机器码，尽折腾那些原始的二进制数据。313 行做了个很好的示范，把 buffer2 指针转化为 struct usb_descriptor_header 的结构体指针，然后就可以使用‘->’来取出 bLength 和 bDescriptorType，这样写的人顺心看的人舒心，你好我好大家好。

那么 306 行就表示如果 GET_DESCRIPTOR 请求返回的数据里除了包括一个配置描述符外，连两个字节都没有，那就说明这个配置在进行裸体行为艺术，能看不能用。

321 行，如果这是个接口描述符就说明这个配置的某个接口拥有一个设置，是没有什么所谓的设置描述符的，一个接口描述符就代表了存在一个设置，接口描述里的 bInterfaceNumber 会指出这个设置隶属于哪个接口。那么这里除了是接口描述符还有可能是什么？还有可能是 class-和 vendor-specific 描述符。

325 行，既然觉得这是个接口描述符，就把这个指针转化为 struct usb_interface_descriptor 结构体指针，你可别被 C 里的这种指针游戏给转晕了，一个地址如果代码不给它赋予什么意义，它除了表示一个地址外就什么都不是，就好像单单说外滩 3 号外滩 9 号什么的，你除了知道它是外滩那边的一个地址外一点概念都没有，但是如果你取 google 一下或者无聊的去看一下里边儿都有些什么东西，它在你的头脑里就不再仅仅是一个地址。同样一个地址，上面转化为 struct usb_descriptor_header 结构体指针和这里转化为 struct usb_interface_descriptor 结构体指针，它就不再仅仅是一个地址，而是代表了不同的含义，外滩 3 号里可以卖阿玛尼开 Jean Georges 也可以堆 10 顿洋垃圾，就看它被赋予了什么。

326 行，仍然不忘保持革命斗争警惕性，年轻单纯期待富婆生活的少女们要记住并不是谁说他是富商他就是富商的，他还有可能是骗子。这里我们也要记住，骑白马的不一定是王子，

他还可能是唐僧；带翅膀的也不一定是天使，妈妈说，那是鸟人；bDescriptorType 等于 USB_DT_INTERFACE 并不说明它就一定是接口描述符了，它的 bLength 还必须要等于 USB_DT_INTERFACE_SIZE。bLength 和 bDescriptorType 一起才能决定一个描述符。

341~352 这几行是用来考验咱们的耐心和勇气的，作为一个男人当然会去努力弄懂它，时代在变，时代女性对男人的要求也在变：做男人得做金刚那样的男人——在世界最高的大楼上为心爱的女人打飞机。要做这样一个顺应潮流的男人首先要明白 n、inums 和 nalts 这几个枯燥的东东是表示什么的，n 记录的是接口的数目，数组 inums 里的每一项都表示一个接口号，数组 nalts 里的每一项记录的是每个接口拥有的设置数目，inums 和 nalts 两个数组里的元素是一一对应的，inums[0]就对应 nalts[0]，inums[1]就对应 nalts[1]。其次还要谨记一个残酷的事实，发送 GET_DESCRIPTOR 请求时，设备并不一定会按照接口 1，接口 2 这样的顺序循规蹈矩的返回数据，虽说协议里是这么要求的，但都在江湖行走谁能没点个性。

361 行，buffer 的最后边儿可能会有些垃圾数据，为了去除这些洋垃圾，这里需要将 size 和配置描述符里的那个 wTotalLength 修正一下。借此地呼吁一下：抵制洋垃圾，从现在人人做起。

364 行，经过上面那个超酷的循环之后，如果统计得到的接口数目和配置描述符里的 bNumInterfaces 不符，或者干脆就没有发现配置里有什么接口，就警告一下。

373 行，又一个 for 循环，目的是看看是不是遗漏了哪个接口号，比如说配置 6 个接口，为什么是 6 那，因为俺的幸运数字是 6，呵呵，每个接口号都应该对应数组 inums 里的一项，如果在 inums 里面没有发现这个接口号，比如 2 吧，那 2 这个接口号就神秘失踪了，你找不到接口 2。这个当然也属于违章驾驶，需要警告一下，开票罚 600，不开票罚 200，你自己选。

384 行，再一个 for 循环，struct usb_interface_caches 做嘛用的早就说过了，USB_MAXALTSETTING 的定义在 config.c 里

```
11 #define USB_MAXALTSETTING               128     /* Hard limit */
```

一个接口最多可以有 128 个设置，足够了。394 行根据每个接口拥有的设置数目为对应的 intf_cache 数组项申请内存。

403 行，配置描述符后面紧跟的不一定就是接口描述符，还可能是 class-和 vendor-specific 描述符，如果有的话。不管有没有，先把 buffer 的地址赋给 extra，如果没有扩展的描述符，则 404 行返回的 i 就等于 0，extralen 也就为 0。

404 行，调用 find_next_descriptor()在 buffer 里寻找配置描述符后面跟着的第一个接口描述符。它也在 config.c 里定义，进去看看

```
22 static int find_next_descriptor(unsigned char *buffer, int size,
23     int dt1, int dt2, int *num_skipped)
24 {
25         struct usb_descriptor_header *h;
26         int n = 0;
27         unsigned char *buffer0 = buffer;
28
29         /* Find the next descriptor of type dt1 or dt2 */
30         while (size > 0) {
31                 h = (struct usb_descriptor_header *) buffer;
32                 if (h->bDescriptorType == dt1 || h->bDescriptorType == dt2)
33                         break;
34                 buffer += h->bLength;
35                 size -= h->bLength;
36                 ++n;
37         }
38
39         /* Store the number of descriptors skipped and return the
40          * number of bytes skipped */
41         if (num_skipped)
42                 *num_skipped = n;
43         return buffer - buffer0;
44 }
```

这个函数需要传递两个描述符类型的参数，不用去费劲儿请什么私家侦探，也不用去费劲儿查手机帐单，32 行已经清清楚楚的表明它是脚踩两只船的，一个多情的种。它不是专一的去寻找一种描述符，而是去寻找两种描述符，比如你指定 dt1 为 USB_DT_INTERFACE，dt2 为 USB_DT_ENDPOINT 时，只要能够找到接口描述符或端点描述符中的一个，这个函数就返回。usb_parse_configuration 函数的 404 行只需要寻找下一个接口描述符，所以 dt1 和 dt2 都设置为 USB_DT_INTERFACE。

这个函数结束后，num_skipped 里记录的是搜索过程中忽略的 dt1 和 dt2 之外其它描述符的数目，返回值表示搜索结束时 buffer 的位置比搜索开始时前进的字节数。其它没什么好讲的，大家都是高手，古龙大侠告诉我们高手过招要简洁。还是回到 usb_parse_configuration 函数。

410 行，根据 find_next_descriptor 的结果修正 buffer 和 size。你可能因为受过很多面试的摧残，或者代码里错过很多次，对 C 里的按引用传递和按值传递已经烂熟于心，看到 find_next_descriptor()那里传递的是 buffer，一个指针，条件反射的觉得它里面对 buffer 的修改必定影响了外面的 buffer，所以认为 buffer 已经指向了寻找到的接口描述符。但是大富翁里的阿土拨已经说了“生活不如意十之八九”，你的这种如意算盘此时并不能如意，find_next_descriptor 里修改的只是参数里 buffer 的值，并没有修改它指向的内容，对于地址本身来说仍然只能算是按值传递，怎么修改都影响不到函数外边，所以这里的 410

行仍然要对 buffer 的位置进行修正。

414 行，事不过三，三个 for 循环之后轮到了一个 while 循环，如果 size 大于 0，就说明配置描述符后面找到了一个接口描述符，根据这个接口描述符的长度，已经可以解析出一个完整的接口描述符了，但是仍然没到乐观的时候，这个接口描述符后面还会跟着一群端点描述符，再然后还会有其它的接口描述符，路漫漫其修远兮，我们要上下而摸索。所以我们又迎来了另一个变态函数 usb_parse_interface，先不管这个它长什么样子，毕竟 usb_parse_configuration()就快到头儿了，暂时只需要知道它返回的时候，buffer 的位置已经在下一个接口描述符那里了，还是那个理儿，对 buffer 地址本身来说是按值传递的，所以 420 行要对这个位置和长度进行下调整以适应新形势。那么这个 while 循环的意思就很明显了，对 buffer 一段一段的解析，直到再也找不到接口描述符了。

425 行，最后这个 for 循环没啥实质性的内容，就是找一下每个接口是不是有哪个设置编号给漏过去了，只要有耐心，你就能看得懂。咱们接下来还是看 config.c 里的那个 usb_parse_interface()

```
158 static int usb_parse_interface(struct device *ddev, int cfgno,
159     struct usb_host_config *config, unsigned char *buffer, int size,
160     u8 inums[], u8 nalts[])
161 {
162         unsigned char *buffer0 = buffer;
163         struct usb_interface_descriptor *d;
164         int inum, asnum;
165         struct usb_interface_cache *intfc;
166         struct usb_host_interface *alt;
167         int i, n;
168         int len, retval;
169         int num_ep, num_ep_orig;
170
171         d = (struct usb_interface_descriptor *) buffer;
172         buffer += d->bLength;
173         size -= d->bLength;
174
175         if (d->bLength < USB_DT_INTERFACE_SIZE)
176                 goto skip_to_next_interface_descriptor;
177
178         /* Which interface entry is this? */
179         intfc = NULL;
180         inum = d->bInterfaceNumber;
181         for (i = 0; i < config->desc.bNumInterfaces; ++i) {
182                 if (inums[i] == inum) {
183                         intfc = config->intf_cache[i];
184                         break;
185                 }
```

```
186         }
187         if (!intfc || intfc->num_altsetting >= nalts[i])
188                 goto skip_to_next_interface_descriptor;
189
190         /* Check for duplicate altsetting entries */
191         asnum = d->bAlternateSetting;
192         for ((i = 0, alt = &intfc->altsetting[0]);
193               i < intfc->num_altsetting;
194              (++i, ++alt)) {
195                 if (alt->desc.bAlternateSetting == asnum) {
196                         dev_warn(ddev, "Duplicate descriptor for config %d "
197                             "interface %d altsetting %d, skipping\n",
198                             cfgno, inum, asnum);
199                         goto skip_to_next_interface_descriptor;
200                 }
201         }
202
203         ++intfc->num_altsetting;
204         memcpy(&alt->desc, d, USB_DT_INTERFACE_SIZE);
205
206         /* Skip over any Class Specific or Vendor Specific descriptors;
207          * find the first endpoint or interface descriptor */
208         alt->extra = buffer;
209         i = find_next_descriptor(buffer, size, USB_DT_ENDPOINT,
210             USB_DT_INTERFACE, &n);
211         alt->extralen = i;
212         if (n > 0)
213                 dev_dbg(ddev, "skipped %d descriptor%s after %s\n",
214                     n, plural(n), "interface");
215         buffer += i;
216         size -= i;
217
218         /* Allocate space for the right(?) number of endpoints */
219         num_ep = num_ep_orig = alt->desc.bNumEndpoints;
220         alt->desc.bNumEndpoints = 0;            // Use as a counter
221         if (num_ep > USB_MAXENDPOINTS) {
222                 dev_warn(ddev, "too many endpoints for config %d interface %d
"
223                     "altsetting %d: %d, using maximum allowed: %d\n",
224                     cfgno, inum, asnum, num_ep, USB_MAXENDPOINTS);
225                 num_ep = USB_MAXENDPOINTS;
226         }
227
228         if (num_ep > 0) {       /* Can't allocate 0 bytes */
```

```
229                 len = sizeof(struct usb_host_endpoint) * num_ep;
230                 alt->endpoint = kzalloc(len, GFP_KERNEL);
231                 if (!alt->endpoint)
232                         return -ENOMEM;
233         }
234
235         /* Parse all the endpoint descriptors */
236         n = 0;
237         while (size > 0) {
238                 if (((struct usb_descriptor_header *)
buffer)->bDescriptorType
239                      == USB_DT_INTERFACE)
240                         break;
241                 retval = usb_parse_endpoint(ddev, cfgno, inum, asnum, alt,
242                     num_ep, buffer, size);
243                 if (retval < 0)
244                         return retval;
245                 ++n;
246
247                 buffer += retval;
248                 size -= retval;
249         }
250
251         if (n != num_ep_orig)
252                 dev_warn(ddev, "config %d interface %d altsetting %d has %d "
253                     "endpoint descriptor%s, different from the interface "
254                     "descriptor's value: %d\n",
255                     cfgno, inum, asnum, n, plural(n), num_ep_orig);
256         return buffer - buffer0;
257
258 skip_to_next_interface_descriptor:
259         i = find_next_descriptor(buffer, size, USB_DT_INTERFACE,
260             USB_DT_INTERFACE, NULL);
261         return buffer - buffer0 + i;
262 }
```

171 行，传递过来的 buffer 里开头儿那部分只能是一个接口描述符，没有什么可质疑的，所以这里将地址转化为 struct usb_interface_descriptor 结构体指针，然后调整 buffer 的位置和 size。

175 行，只能是并不说明它就是，只有 bLength 等于 USB_DT_INTERFACE_SIZE 才说明开头儿的 USB_DT_INTERFACE_SIZE 字节确实是个接口描述符。否则就没必要再对这些数据进行什么处理了，直接跳到最后吧。先看看这个函数的最后都发生了什么，从新的位置开始再次调用 find_next_descriptor()在 buffer 里寻找下一个接口描述符。

179 行，因为数组 inums 并不一定是按照接口的顺序来保存接口号的，inums[1]对应的可能是接口 1 也可能是接口 0，所以这里要用 for 循环来寻找这个接口对应着 inums 里的哪一项，从而根据在数组里的位置获得接口对应的 struct usb_interface_cache 结构体。usb_parse_configuration()已经告诉了我们，同一个接口在 inums 和 intf_cache 这两个数组里的位置是一样的。

191 行，获得这个接口描述符对应的设置编号，然后根据这个编号从接口的 cache 里搜索看这个设置是不是已经遇到过了，如果已经遇到过，就没必要再对这个接口描述符进行处理，直接跳到最后，否则意味着发现了一个新的设置，要将它添加到 cache 里，并 cache 里的设置数目 num_altsetting 加 1。要记住，设置是用 struct usb_host_interface 结构来表示的，一个接口描述符就对应一个设置。

208 行，这段代码好熟悉啊。现在 buffer 开头儿的那个接口描述符已经理清了，要解析它后面的那些数据了。先把位置赋给这个刚解析出来的接口描述符的 extra，然后再从这个位置开始去寻找下一个距离最近的一个接口描述符或端点描述符。如果这个接口描述符后面还跟有 class-或 vendor-specific 描述符，则 find_next_descriptor 的返回值会大于 0，buffer 的位置和 size 也要进行相应的调整，来指向新找到的接口描述符或端点描述符。

这里 find_next_descriptor 的 dt1 参数和 dt2 参数就不再一样了，因为如果一个接口只用到端点 0，它的接口描述符后边儿是不会跟有端点描述符的。

219 行，获得这个设置使用的端点数目，然后将相应接口描述符里的 bNumEndpoints 置 0，为什么？你要往下看。USB_MAXENDPOINTS 在 config.c 里定义

```
12 #define USB_MAXENDPOINTS                30      /* Hard limit */
```

为什么这个最大上限为 30？前面也提到过，如果你不想频繁的蓦然回首那就简单认为是协议里这么规定的好了。然后根据端点数为接口描述符里的 endpoint 数组申请内存。

237 行，走到这里，buffer 开头儿的那个接口描述符已经理清了，而且也找到了下一个接口描述符或端点描述符的位置，该从这个新的位置开始解析了，于是又遇到了一个似曾相识的 while 循环。238 行先判断一下前面找到的是接口描述符还是端点描述符，如果是接口描述符就中断这个 while 循环，返回与下一个接口描述符的距离。否则说明在 buffer 当前的位置上待着的是一个端点描述符，因此就要迎来另一个函数 usb_parse_endpoint 对面紧接着的数据进行解析。usb_parse_endpoint()返回的时候，buffer 的位置已经在下一个端点描述符那里了，247 行调整 buffer 的位置长度，这个 while 循环的也很明显了，对 buffer 一段一段的解析，直到遇到下一个接口描述符或者已经走到 buffer 结尾。现在看看 config.c 里定义的 usb_parse_endpoint 函数

```
46 static int usb_parse_endpoint(struct device *ddev, int cfgno, int inum,
47     int asnum, struct usb_host_interface *ifp, int num_ep,
48     unsigned char *buffer, int size)
```

```
49 {
50         unsigned char *buffer0 = buffer;
51         struct usb_endpoint_descriptor *d;
52         struct usb_host_endpoint *endpoint;
53         int n, i, j;
54
55         d = (struct usb_endpoint_descriptor *) buffer;
56         buffer += d->bLength;
57         size -= d->bLength;
58
59         if (d->bLength >= USB_DT_ENDPOINT_AUDIO_SIZE)
60                 n = USB_DT_ENDPOINT_AUDIO_SIZE;
61         else if (d->bLength >= USB_DT_ENDPOINT_SIZE)
62                 n = USB_DT_ENDPOINT_SIZE;
63         else {
64                 dev_warn(ddev, "config %d interface %d altsetting %d has an "
65                     "invalid endpoint descriptor of length %d, skipping\n",
66                     cfgno, inum, asnum, d->bLength);
67                 goto skip_to_next_endpoint_or_interface_descriptor;
68         }
69
70         i = d->bEndpointAddress & ~USB_ENDPOINT_DIR_MASK;
71         if (i >= 16 || i == 0) {
72                 dev_warn(ddev, "config %d interface %d altsetting %d has an "
73                     "invalid endpoint with address 0x%X, skipping\n",
74                     cfgno, inum, asnum, d->bEndpointAddress);
75                 goto skip_to_next_endpoint_or_interface_descriptor;
76         }
77
78         /* Only store as many endpoints as we have room for */
79         if (ifp->desc.bNumEndpoints >= num_ep)
80                 goto skip_to_next_endpoint_or_interface_descriptor;
81
82         endpoint = &ifp->endpoint[ifp->desc.bNumEndpoints];
83         ++ifp->desc.bNumEndpoints;
84
85         memcpy(&endpoint->desc, d, n);
86         INIT_LIST_HEAD(&endpoint->urb_list);
87
88         /* If the bInterval value is outside the legal range,
89          * set it to a default value: 32 ms */
90         i = 0;          /* i = min, j = max, n = default */
91         j = 255;
92         if (usb_endpoint_xfer_int(d)) {
```

```
93                 i = 1;
94                 switch (to_usb_device(ddev)->speed) {
95                 case USB_SPEED_HIGH:
96                         n = 9;          /* 32 ms = 2^(9-1) uframes */
97                         j = 16;
98                         break;
99                 default:                /* USB_SPEED_FULL or _LOW */
100                         /* For low-speed, 10 ms is the official minimum.
101                          * But some "overclocked" devices might want faster
102                          * polling so we'll allow it. */
103                         n = 32;
104                         break;
105                 }
106         } else if (usb_endpoint_xfer_isoc(d)) {
107                 i = 1;
108                 j = 16;
109                 switch (to_usb_device(ddev)->speed) {
110                 case USB_SPEED_HIGH:
111                         n = 9;          /* 32 ms = 2^(9-1) uframes */
112                         break;
113                 default:                /* USB_SPEED_FULL */
114                         n = 6;          /* 32 ms = 2^(6-1) frames */
115                         break;
116                 }
117         }
118         if (d->bInterval < i || d->bInterval > j) {
119                 dev_warn(ddev, "config %d interface %d altsetting %d "
120                     "endpoint 0x%X has an invalid bInterval %d, "
121                     "changing to %d\n",
122                     cfgno, inum, asnum,
123                     d->bEndpointAddress, d->bInterval, n);
124                 endpoint->desc.bInterval = n;
125         }
126
127         /* Skip over any Class Specific or Vendor Specific descriptors;
128          * find the next endpoint or interface descriptor */
129         endpoint->extra = buffer;
130         i = find_next_descriptor(buffer, size, USB_DT_ENDPOINT,
131             USB_DT_INTERFACE, &n);
132         endpoint->extralen = i;
133         if (n > 0)
134                 dev_dbg(ddev, "skipped %d descriptor%s after %s\n",
135                     n, plural(n), "endpoint");
136         return buffer - buffer0 + i;
```

```
137
138 skip_to_next_endpoint_or_interface_descriptor:
139         i = find_next_descriptor(buffer, size, USB_DT_ENDPOINT,
140             USB_DT_INTERFACE, NULL);
141         return buffer - buffer0 + i;
142 }
```

一个一个变态的函数看过来，到现在都已经麻木了，古语云，生活就像被强奸，如果无力反抗就闭上眼睛享受吧。遇到现在这种鬼天气，你可以骂老天“你让夏天和冬天同房了吧？生出这鬼天气。”但是遇到这种函数，你谁都不能骂，对 linus，对 Greg，对 Alan，你有的只能是崇敬。

55 行，buffer 开头儿只能是一个端点描述符，所以这里将地址转化为 struct usb_endpoint_descriptor 结构体指针，然后调整 buffer 的位置和 size。

59 行，这里要明白的是端点描述符与配置描述符、接口描述符不一样，它是可能有两种大小的。

70 行，得到端点号。这里的端点号不能为 0，因为端点 0 是没有描述符的，也不能大于 16，为什么？同样如果你不想蓦然回首，就当成协议里规定的吧。

79 行，要知道这个 bNumEndpoints 在 usb_parse_interface()的 220 行是被赋为 0 了的。

82 行，要知道这个 endpoint 数组在 usb_parse_interface()的 230 行也是已经申请好内存了的。从这里你应该明白 bNumEndpoints 是被当成了一个计数器，发现一个端点描述符，它就加 1，并把找到的端点描述符 copy 到设置的 endpoint 数组里。

86 行，初始化端点的 urb 队列 urb_list。

88~125 行，这堆代码的目的是处理端点的 bInterval，你要想不被它们给忽悠了，得明白几个问题。第一个就是，i，j，n 分别表示什么。90~117 这么多行就为了给它们选择一个合适的值，i 和 j 限定了 bInterval 的一个范围，bInterval 如果在这里边儿，它就是合法的，如果超出了这个范围，它就是非法的，就要修理修理它，像 124 行做的那样将 n 赋给它，那么 n 表示的就是 bInterval 的一个默认值。i 和 j 的默认值分别为 0 和 255，也就是说合法的范围默认是 0~255，对于批量端点和控制端点，bInterval 对你我来说并没有太大的用处，不过协议里还是规定了，这个范围只能为 0~255。对于中断端点和等时端点，bInterval 表演的舞台就很大了，对这个范围也要做一些调整。

第二个问题就是如何判断端点是中断的还是等时的。这涉及到两个函数 usb_endpoint_xfer_int 和 usb_endpoint_xfer_isoc，它们都在 include/linux/usb.h 里定义

```
589 /**
590  * usb_endpoint_xfer_int - check if the endpoint has interrupt transfer type
591  * @epd: endpoint to be checked
592  *
593  * Returns true if the endpoint is of type interrupt, otherwise it returns
594  * false.
595  */
596 static inline int usb_endpoint_xfer_int(const struct usb_endpoint_descriptor
*epd)
597 {
598         return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
599                 USB_ENDPOINT_XFER_INT);
600 }
601
602 /**
603  * usb_endpoint_xfer_isoc - check if the endpoint has isochronous transfer type
604  * @epd: endpoint to be checked
605  *
606  * Returns true if the endpoint is of type isochronous, otherwise it returns
607  * false.
608  */
609 static inline int usb_endpoint_xfer_isoc(const struct usb_endpoint_descriptor
*epd)
610 {
611         return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
612                 USB_ENDPOINT_XFER_ISOC);
613 }
```

这俩函数so直白，一点都不含蓄，你根本不用去猜它们的心思就能明明白白了。一桌麻将还差两个，另外两个就是 usb_endpoint_xfer_bulk和 usb_endpoint_xfer_control，用来判断批量端点和控制端点的。

第三个问题是 to_usb_device。usb_parse_endpoint()的参数是 struct device 结构体，要获得设备的速度就需要使用 to_usb_device 将它转化为 struct usb_device 结构体，这是个 include/linux/usb.h 里定义的宏

```
410 #define to_usb_device(d) container_of(d, struct usb_device, dev)
```

OK，接着继续看 usb_parse_endpoint 的 129 行，现在你对这几行玩的把戏应该很明白了。这里接着在 buffer 里寻找下一个端点描述符或者接口描述符。

经过 usb_parse_configuration、usb_parse_interface 和 usb_parse_endpoint 这三个函数一步一营的层层推进，通过 GET_DESCRIPTOR 请求所获得那堆数据现在已经解析的清清白白。现在，设备的各个配置信息已经了然于胸，那接下来设备的那条生命线该怎么去走？它已经可以进入 Configured 状态了么？事情没这么简单，光是获得设备各个配置信息没用，要进入 Configured 状态，你还得有选择有目的有步骤有计划的去配置设备，那怎么去有选择有目的有步骤有计划？这好像就不是 core 能够答复的问题了，毕竟它并不知道你希望你的设备采用哪种配置，只有你的设备的驱动才知道，所以接下来设备要做的是去在设备模型的茫茫人海中寻找属于自己的驱动。

做为一个负责任的男人，绝对不能忘记的是设备的那个struct usb_device结构体在出生的时候就带有usb_bus_type和usb_device_type这样的胎记，Linux设备模型根据总线类型usb_bus_type将设备添加到usb总线的那条有名的设备链表里，然后去轮询usb总线的另外一条有名的驱动链表，针对每个找到的驱动去调用usb总线的match函数，也就是usb_device_match()，去为设备寻找另一个匹配的半圆。match函数会根据设备的自身条件和类型usb_device_type安排设备走设备那条路，从而匹配到那个对所有usb_device_type类型的设备都来者不拒的花心大萝卜，usb世界里唯一的那个usb设备驱动（不是usb接口驱动）struct device_driver结构体对象usb_generic_driver。

驱动的生命线（一）

给女生：“如果有一个男生追你，那你不过是达到了咱学校 mm 的平均水平；如果有五个男生追你，那你勉强可算是班花系花级别；如果有十个男生追你，~~~哼哼，这事儿也就是在咱交大有~~”

usb_generic_driver 不仅仅有十个男生追，是个 usb 设备都会拜倒在她的石榴裙下，争先恐后的找她配对儿，什么班花系花校花什么的根本就是在诋毁她的身份。但是她要想成为一个 usb 世界花，必须得满足一个前提，就是她必须得来到 usb 的这个大世界里，在 usb 的大舞台上展示自己的魅力，林妹妹那种没有登过 T 台的世界名模毕竟是个特殊到不能再特殊的特例。

现在开始就沿着 usb_generic_driver 的成名之路走一走，设备的生命线你可以想当然的认为是从你的 usb 设备连接到 hub 的某个端口时开始，驱动的生命线就必须得回溯到 usb 子系统的初始化函数 usb_init 了。

```
895         retval = usb_register_device_driver(&usb_generic_driver,
THIS_MODULE);
896         if (!retval)
897                 goto out;
```

在 usb 子系统初始化的时候就调用 driver.c 里的 usb_register_device_driver 函数将 usb_generic_driver 注册给系统了，怀胎十月，嗷嗷一声之后，usb 世界里的一个超级大美女诞生了。这个美女长什么样早先贴过了，现在看看带她来到这个世界上的 usb_register_device_driver 函数。

```
661 /**
662  * usb_register_device_driver - register a USB device (not interface) driver
663  * @new_udriver: USB operations for the device driver
664  * @owner: module owner of this driver.
665  *
666  * Registers a USB device driver with the USB core.  The list of
667  * unattached devices will be rescanned whenever a new driver is
668  * added, allowing the new driver to attach to any recognized devices.
669  * Returns a negative error code on failure and 0 on success.
670  */
671 int usb_register_device_driver(struct usb_device_driver *new_udriver,
672                 struct module *owner)
673 {
674         int retval = 0;
675
676         if (usb_disabled())
677                 return -ENODEV;
678
679         new_udriver->drvwrap.for_devices = 1;
680         new_udriver->drvwrap.driver.name = (char *) new_udriver->name;
681         new_udriver->drvwrap.driver.bus = &usb_bus_type;
682         new_udriver->drvwrap.driver.probe = usb_probe_device;
683         new_udriver->drvwrap.driver.remove = usb_unbind_device;
684         new_udriver->drvwrap.driver.owner = owner;
685
686         retval = driver_register(&new_udriver->drvwrap.driver);
687
688         if (!retval) {
689                 pr_info("%s: registered new device driver %s\n",
690                         usbcore_name, new_udriver->name);
691                 usbfs_update_special();
692         } else {
693                 printk(KERN_ERR "%s: error %d registering device "
694                         "       driver %s\n",
695                         usbcore_name, retval, new_udriver->name);
696         }
697
698         return retval;
699 }
```

676 行，判断一下 usb 子系统是不是在你启动内核的时候就被禁止了，如果是的话，这个超级大美女的生命也就太短暂了，正是应了那句“自古美女多薄命”的古话，不过一般来说，你不至于无聊到那种地步和她过不去吧。usb_disabled 在 usb.c 里定义

```
852 /*
853  * for external read access to <nousb>
854  */
855 int usb_disabled(void)
856 {
857         return nousb;
858 }
```

如果你不是存心和我过不去的话，是应该知道 nosub 表示什么意思的。

679 行，看到没，for_devices就是在这儿被初始化为 1 的，有了它，match里的那个is_usb_device_driver把门儿的才有章可循有凭可依。

下面就是充实了下 usb_generic_driver 里嵌入的那个 struct device_driver 结构体，usb_generic_driver 就是通过它和设备模型搭上关系的。name 就是 usb_generic_driver 的名字，即 usb，所属的总线类型同样被设置为 usb_bus_type，然后是指定 probe 函数和 remove 函数。

686 行，调用设备模型的函数 driver_register 将 usb_generic_driver 添加到 usb 总线的那条驱动链表里，然后它就可以接受 usb 设备的讨好、献殷勤、表白。

usb_generic_driver 和 usb 设备匹配成功后，就会调用 682 行指定的 probe 函数 usb_probe_device()，现在看看 driver.c 里定义的这个函数

```
151 /* called from driver core with dev locked */
152 static int usb_probe_device(struct device *dev)
153 {
154         struct usb_device_driver *udriver =
to_usb_device_driver(dev->driver);
155         struct usb_device *udev;
156         int error = -ENODEV;
157
158         dev_dbg(dev, "%s\n", __FUNCTION__);
159
160         if (!is_usb_device(dev))        /* Sanity check */
161                 return error;
162
163         udev = to_usb_device(dev);
164
165         /* TODO: Add real matching code */
```

```
166
167         /* The device should always appear to be in use
168          * unless the driver suports autosuspend.
169          */
170         udev->pm_usage_cnt = !(udriver->supports_autosuspend);
171
172         error = udriver->probe(udev);
173         return error;
174 }
```

154 行，to_usb_device_driver 是 include/linux/usb.h 里定义的一个宏，和前面遇到的那个 to_usb_device 有异曲同工之妙，

```
889 #define to_usb_device_driver(d) container_of(d, struct usb_device_driver, \
890                 drvwrap.driver)
```

160 行，match 函数 543 行的那个把门儿的又调到这儿来把门儿了，如果你这个设备的类型不是 usb_device_type，那怎么从前面走到这儿的他管不着，但是到他这里就不可能再蒙混过关了。你的设备虽然和咱们的大美女 usb_generic_driver 是 match 成功了，但是还要经过双方父母的近距离审查，没房子没车子没票子没马子这种条件怎么配得上美女那，该躲哪儿哭泣就躲哪儿哭泣去吧。

163 行，说曹操曹操就到，前面刚提到 to_usb_device 它这就来了。

170 行，pm_usage_cnt和supports_autosuspend这两个前面都提到过那么一下，现在将那两个片断给回顾一下。每个struct usb_interface或struct usb_device里都有一个pm_usage_cnt，每个struct usb_driver或struct usb_device_driver里都有一个supports_autosuspend。提到pm_usage_cnt时说只有它为 0 时才会允许接口autosuspend，提到supports_autosuspend时说如果它为 0 就不再允许绑定到这个驱动的接口autosuspend。接口乎？设备乎？有些时候需要那么难得糊涂一下。需要的时候，群众的 眼睛是雪亮的，接口是接口设备是设备，不需要的时候，群众是不明真相的，接口设备一个样。这里就是不需要的时候，所以将上面的话里的接口换成设备套一下就是：pm_usage_cnt为 0 时才会允许设备autosuspend，supports_autosuspend为 0 就不再允许绑定到这个驱动的设备autosuspend。

所有的 usb 设备都是绑定到 usb_generic_driver 上面的，usb_generic_driver 的 supports_autosuspend 字段又是为 1 的，所以这行就是将设备 struct usb_device 结构体的 pm_usage_cnt 置为了 0，也就是说允许设备 autosuspend。但是不是说这里将 pm_usage_cnt 轻轻松松置为 0，设备就能够 autosuspend 了，什么事都是说起来简单，做起来就不是那么回事儿，驱动必须得实现一对儿 suspend/resume 函数供 PM 子系统那块驱使，usb_generic_driver 里的这对函数就是 generic_suspend/generic_resume。此处不是久留之地，就不多说它们了。

172 行，飘过了上面有关电源管理的那行之后，这里要调用 usb_generic_driver 自己私有的 probe 函数 generic_probe()对你的设备进行进一步的审查，它在 generic.c 里定义

```
153 static int generic_probe(struct usb_device *udev)
154 {
155         int err, c;
156
157         /* put device-specific files into sysfs */
158         usb_create_sysfs_dev_files(udev);
159
160         /* Choose and set the configuration.  This registers the interfaces
161          * with the driver core and lets interface drivers bind to them.
162          */
163         c = choose_configuration(udev);
164         if (c >= 0) {
165                 err = usb_set_configuration(udev, c);
166                 if (err) {
167                         dev_err(&udev->dev, "can't set config #%d, error
%d\n",
168                                         c, err);
169                         /* This need not be fatal.  The user can try to
170                          * set other configurations. */
171                 }
172         }
173
174         /* USB device state == configured ... usable */
175         usb_notify_add_device(udev);
176
177         return 0;
178 }
```

这函数说简单也简单，说复杂也复杂，简单的是外表，复杂的是内心。用一句话去概括它的中心思想，就是从设备可能的众多配置中选择一个合适的，然后去配置设备，从而让设备进入期待已久的 Configured 状态。概括了中心思想，再去看看细节。先看看是怎么选择一个配置的，调用的是 generic.c 里的 choose_configuration 函数。

```
42 static int choose_configuration(struct usb_device *udev)
43 {
44         int i;
45         int num_configs;
46         int insufficient_power = 0;
47         struct usb_host_config *c, *best;
48
49         best = NULL;
50         c = udev->config;
```

```
51          num_configs = udev->descriptor.bNumConfigurations;
52          for (i = 0; i < num_configs; (i++, c++)) {
53                  struct usb_interface_descriptor *desc = NULL;
54
55                  /* It's possible that a config has no interfaces! */
56                  if (c->desc.bNumInterfaces > 0)
57                          desc = &c->intf_cache[0]->altsetting->desc;
58
59                  /*
60                   * HP's USB bus-powered keyboard has only one configuration
61                   * and it claims to be self-powered; other devices may have
62                   * similar errors in their descriptors.  If the next test
63                   * were allowed to execute, such configurations would always
64                   * be rejected and the devices would not work as expected.
65                   * In the meantime, we run the risk of selecting a config
66                   * that requires external power at a time when that power
67                   * isn't available.  It seems to be the lesser of two evils.
68                   *
69                   * Bugzilla #6448 reports a device that appears to crash
70                   * when it receives a GET_DEVICE_STATUS request!  We don't
71                   * have any other way to tell whether a device is self-powered,
72                   * but since we don't use that information anywhere but here,
73                   * the call has been removed.
74                   *
75                   * Maybe the GET_DEVICE_STATUS call and the test below can
76                   * be reinstated when device firmwares become more reliable.
77                   * Don't hold your breath.
78                   */
79  #if 0
80                  /* Rule out self-powered configs for a bus-powered device */
81                  if (bus_powered && (c->desc.bmAttributes &
82                                          USB_CONFIG_ATT_SELFPOWER))
83                          continue;
84  #endif
85
86                  /*
87                   * The next test may not be as effective as it should be.
88                   * Some hubs have errors in their descriptor, claiming
89                   * to be self-powered when they are really bus-powered.
90                   * We will overestimate the amount of current such hubs
91                   * make available for each port.
92                   *
93                   * This is a fairly benign sort of failure.  It won't
94                   * cause us to reject configurations that we should have
```

```
95                  * accepted.
96                  */
97
98                 /* Rule out configs that draw too much bus current */
99                 if (c->desc.bMaxPower * 2 > udev->bus_mA) {
100                         insufficient_power++;
101                         continue;
102                 }
103
104                 /* When the first config's first interface is one of Microsoft's
105                  * pet nonstandard Ethernet-over-USB protocols, ignore it
unless
106                  * this kernel has enabled the necessary host side driver.
107                  */
108                 if (i == 0 && desc && (is_rndis(desc) || is_activesync(desc)))
{
109 #if !defined(CONFIG_USB_NET_RNDIS_HOST)
&& !defined(CONFIG_USB_NET_RNDIS_HOST_MODULE)
110                         continue;
111 #else
112                         best = c;
113 #endif
114                 }
115
116                 /* From the remaining configs, choose the first one whose
117                  * first interface is for a non-vendor-specific class.
118                  * Reason: Linux is more likely to have a class driver
119                  * than a vendor-specific driver. */
120                 else if (udev->descriptor.bDeviceClass !=
121                                                 USB_CLASS_VENDOR_SPEC &&
122                                 (!desc || desc->bInterfaceClass !=
123                                                 USB_CLASS_VENDOR_SPEC)) {
124                         best = c;
125                         break;
126                 }
127
128                 /* If all the remaining configs are vendor-specific,
129                  * choose the first one. */
130                 else if (!best)
131                         best = c;
132         }
133
134         if (insufficient_power > 0)
135                 dev_info(&udev->dev, "rejected %d configuration%s "
```

```
136                         "due to insufficient available bus power\n",
137                         insufficient_power, plural(insufficient_power));
138
139         if (best) {
140                 i = best->desc.bConfigurationValue;
141                 dev_info(&udev->dev,
142                         "configuration #%d chosen from %d choice%s\n",
143                         i, num_configs, plural(num_configs));
144         } else {
145                 i = -1;
146                 dev_warn(&udev->dev,
147                         "no configuration chosen from %d choice%s\n",
148                         num_configs, plural(num_configs));
149         }
150         return i;
151 }
```

设备各个配置的详细信息在设备自身的漫漫人生旅途中就已经获取存放在相关的几个成员里了，怎么从中挑选一个让人满意的？显然谁都会说去一个一个的浏览每个配置，看看有没有称心如意的，于是就有了 52 行的 for 循环。

刚看到这个 for 循环就有点傻眼了，就好像武侠小说里一个闭上眼睛准备好要受虐的人却突然发现神功秘笈奇花异果绝世美女都跑到自己怀里了一样，有点不大相信，居然注释要远远多于代码，很少受过这样的优待，都要让人忍不住去学琼瑶阿姨娇嗔一句：你好好讨厌好讨厌哦!。

这么一个 for 循环，咱们把它分成三大段，59~84 这一大段你什么都可以不看，就是不能不看那个#if 0，一见到它，你就应该像迎面见到一个大美女那样热血沸腾，因为它就意味着你可以华丽丽的飘过这么一大段了。

第二段是 98~102 行，这一段牵扯到人世间最让人无可奈何的一对矛盾，索取与给予。一个配置索取的电流比 hub 所能给予的还要大，显然它不会是一个让人满意的配置。

第三段是 108~131 行，关于这段只说一个原则，linux 更 care 那些标准的东西，比如 USB_CLASS_VIDEO、USB_CLASS_AUDIO 等等这样子的设备和接口就更讨人喜欢一些，所以就会优先选择非 USB_CLASS_VENDOR_SPEC 的接口。

for 循环之后，残余的那些部分都是调试用的，输出一些调试信息，不需要去关心，不过里面出现了个有趣的函数 plural，它是一个在 generic.c 开头儿定义的内联函数

```
23 static inline const char *plural(int n)
24 {
25         return (n == 1 ? "" : "s");
```

```
26 }
```

参数 n 为 1 返回一个空字符串，否则返回一个‘s’，瞄一下使用了这个函数的那几个打印语句，就明白它是用来打印一个英语名词的单复数的，复数的话就加上个 s。呵呵，那些不规律的名词单复数形式咋办？凉拌！

不管你疑惑也好，满意也好，choose_configuration 就是这样按照自己的标准挑选了一个比较合自己心意的配置，接下来当然就是要用这个配置去配置设备以便让它迈进 Configured 状态了。

驱动的生命线（二）

关系是跑出来的，感情是喝出来的，朋友是处出来的，事业是干出来的，但设备是配置出来的，绝非吹出来的。core 配置设备使用的是 message.c 里的 usb_set_configuration 函数

```
1388 /*
1389  * usb_set_configuration - Makes a particular device setting be current
1390  * @dev: the device whose configuration is being updated
1391  * @configuration: the configuration being chosen.
1392  * Context: !in_interrupt(), caller owns the device lock
1393  *
1394  * This is used to enable non-default device modes.  Not all devices
1395  * use this kind of configurability; many devices only have one
1396  * configuration.
1397  *
1398  * @configuration is the value of the configuration to be installed.
1399  * According to the USB spec (e.g. section 9.1.1.5), configuration values
1400  * must be non-zero; a value of zero indicates that the device in
1401  * unconfigured.  However some devices erroneously use 0 as one of their
1402  * configuration values.  To help manage such devices, this routine will
1403  * accept @configuration = -1 as indicating the device should be put in
1404  * an unconfigured state.
1405  *
1406  * USB device configurations may affect Linux interoperability,
1407  * power consumption and the functionality available.  For example,
1408  * the default configuration is limited to using 100mA of bus power,
1409  * so that when certain device functionality requires more power,
1410  * and the device is bus powered, that functionality should be in some
1411  * non-default device configuration.  Other device modes may also be
1412  * reflected as configuration options, such as whether two ISDN
```

```
1413  * channels are available independently; and choosing between open
1414  * standard device protocols (like CDC) or proprietary ones.
1415  *
1416  * Note that USB has an additional level of device configurability,
1417  * associated with interfaces.  That configurability is accessed using
1418  * usb_set_interface().
1419  *
1420  * This call is synchronous. The calling context must be able to sleep,
1421  * must own the device lock, and must not hold the driver model's USB
1422  * bus mutex; usb device driver probe() methods cannot use this routine.
1423  *
1424  * Returns zero on success, or else the status code returned by the
1425  * underlying call that failed.  On successful completion, each interface
1426  * in the original device configuration has been destroyed, and each one
1427  * in the new configuration has been probed by all relevant usb device
1428  * drivers currently known to the kernel.
1429  */
1430 int usb_set_configuration(struct usb_device *dev, int configuration)
1431 {
1432         int i, ret;
1433         struct usb_host_config *cp = NULL;
1434         struct usb_interface **new_interfaces = NULL;
1435         int n, nintf;
1436
1437         if (configuration == -1)
1438                 configuration = 0;
1439         else {
1440                 for (i = 0; i < dev->descriptor.bNumConfigurations; i++) {
1441                         if (dev->config[i].desc.bConfigurationValue ==
1442                                         configuration) {
1443                                 cp = &dev->config[i];
1444                                 break;
1445                         }
1446                 }
1447         }
1448         if ((!cp && configuration != 0))
1449                 return -EINVAL;
1450
1451         /* The USB spec says configuration 0 means unconfigured.
1452          * But if a device includes a configuration numbered 0,
1453          * we will accept it as a correctly configured state.
1454          * Use -1 if you really want to unconfigure the device.
1455          */
1456         if (cp && configuration == 0)
```

```
1457                 dev_warn(&dev->dev, "config 0 descriptor??\n");
1458
1459         /* Allocate memory for new interfaces before doing anything else,
1460          * so that if we run out then nothing will have changed. */
1461         n = nintf = 0;
1462         if (cp) {
1463                 nintf = cp->desc.bNumInterfaces;
1464                 new_interfaces = kmalloc(nintf * sizeof(*new_interfaces),
1465                                 GFP_KERNEL);
1466                 if (!new_interfaces) {
1467                         dev_err(&dev->dev, "Out of memory");
1468                         return -ENOMEM;
1469                 }
1470
1471                 for (; n < nintf; ++n) {
1472                         new_interfaces[n] = kzalloc(
1473                                         sizeof(struct usb_interface),
1474                                         GFP_KERNEL);
1475                         if (!new_interfaces[n]) {
1476                                 dev_err(&dev->dev, "Out of memory");
1477                                 ret = -ENOMEM;
1478 free_interfaces:
1479                                 while (--n >= 0)
1480                                         kfree(new_interfaces[n]);
1481                                 kfree(new_interfaces);
1482                                 return ret;
1483                         }
1484                 }
1485
1486                 i = dev->bus_mA - cp->desc.bMaxPower * 2;
1487                 if (i < 0)
1488                         dev_warn(&dev->dev, "new config #%d exceeds power "
1489                                         "limit by %dmA\n",
1490                                         configuration, -i);
1491         }
1492
1493         /* Wake up the device so we can send it the Set-Config request */
1494         ret = usb_autoresume_device(dev);
1495         if (ret)
1496                 goto free_interfaces;
1497
1498         /* if it's already configured, clear out old state first.
1499          * getting rid of old interfaces means unbinding their drivers.
1500          */
```

```
1501        if (dev->state != USB_STATE_ADDRESS)
1502                usb_disable_device (dev, 1);    // Skip ep0
1503
1504        if ((ret = usb_control_msg(dev, usb_sndctrlpipe(dev, 0),
1505                        USB_REQ_SET_CONFIGURATION, 0, configuration, 0,
1506                        NULL, 0, USB_CTRL_SET_TIMEOUT)) < 0) {
1507
1508                /* All the old state is gone, so what else can we do?
1509                 * The device is probably useless now anyway.
1510                 */
1511                cp = NULL;
1512        }
1513
1514        dev->actconfig = cp;
1515        if (!cp) {
1516                usb_set_device_state(dev, USB_STATE_ADDRESS);
1517                usb_autosuspend_device(dev);
1518                goto free_interfaces;
1519        }
1520        usb_set_device_state(dev, USB_STATE_CONFIGURED);
1521
1522        /* Initialize the new interface structures and the
1523         * hc/hcd/usbcore interface/endpoint state.
1524         */
1525        for (i = 0; i < nintf; ++i) {
1526                struct usb_interface_cache *intfc;
1527                struct usb_interface *intf;
1528                struct usb_host_interface *alt;
1529
1530                cp->interface[i] = intf = new_interfaces[i];
1531                intfc = cp->intf_cache[i];
1532                intf->altsetting = intfc->altsetting;
1533                intf->num_altsetting = intfc->num_altsetting;
1534                kref_get(&intfc->ref);
1535
1536                alt = usb_altnum_to_altsetting(intf, 0);
1537
1538                /* No altsetting 0?  We'll assume the first altsetting.
1539                 * We could use a GetInterface call, but if a device is
1540                 * so non-compliant that it doesn't have altsetting 0
1541                 * then I wouldn't trust its reply anyway.
1542                 */
1543                if (!alt)
1544                        alt = &intf->altsetting[0];
```

```
1545
1546                    intf->cur_altsetting = alt;
1547                    usb_enable_interface(dev, intf);
1548                    intf->dev.parent = &dev->dev;
1549                    intf->dev.driver = NULL;
1550                    intf->dev.bus = &usb_bus_type;
1551                    intf->dev.type = &usb_if_device_type;
1552                    intf->dev.dma_mask = dev->dev.dma_mask;
1553                    device_initialize (&intf->dev);
1554                    mark_quiesced(intf);
1555                    sprintf (&intf->dev.bus_id[0], "%d-%s:%d.%d",
1556                             dev->bus->busnum, dev->devpath,
1557                             configuration, alt->desc.bInterfaceNumber);
1558            }
1559            kfree(new_interfaces);
1560
1561            if (cp->string == NULL)
1562                    cp->string = usb_cache_string(dev,
cp->desc.iConfiguration);
1563
1564            /* Now that all the interfaces are set up, register them
1565             * to trigger binding of drivers to interfaces.  probe()
1566             * routines may install different altsettings and may
1567             * claim() any interfaces not yet bound.  Many class drivers
1568             * need that: CDC, audio, video, etc.
1569             */
1570            for (i = 0; i < nintf; ++i) {
1571                    struct usb_interface *intf = cp->interface[i];
1572
1573                    dev_dbg (&dev->dev,
1574                            "adding %s (config #%d, interface %d)\n",
1575                            intf->dev.bus_id, configuration,
1576                            intf->cur_altsetting->desc.bInterfaceNumber);
1577                    ret = device_add (&intf->dev);
1578                    if (ret != 0) {
1579                            dev_err(&dev->dev, "device_add(%s) --> %d\n",
1580                                    intf->dev.bus_id, ret);
1581                            continue;
1582                    }
1583                    usb_create_sysfs_intf_files (intf);
1584            }
1585
1586            usb_autosuspend_device(dev);
1587            return 0;
```

1588 }

这个函数很迷信，从 1388 行开始，到 1588 行结束，怎么着它都要发发，希望咱们看的时候也能沾上点财气运气。那就先对着它许个愿吧，并不是真的相信它，但是反正也是免费的，而且也没有证据证明它不灵。

说代码前咱们再聊点这个函数背后的人生哲学，你设备不是和 usb_generic_driver 这个大美女配对成功了么，可是这并不是就说你可以高枕无忧了，要想保持和她的这种亲密关系，你得想办法让她得到满足，你就得抱着一颗勇敢的心，准备着让她去配置，准备着她想让你什么样你就什么样。你要明白，吸引住男人的办法就是让他一直得不到，吸引住女人的办法正好相反，就是让她一直满足。从这个角度看，这个函数就可以泾渭分明的分成三个部分三个阶段，从 1432 到 1491 的这几十行是准备阶段，做做常规检查啊，申请申请内存啊，搞点前戏什么的。1498 到 1520 这部分可是重头戏，就是在这里设备从 Address 发展到了 Configured，可算是高潮阶段，别看它短，这是事物发展的规律，也是每个男人女人的规律。1522 到 1584 这阶段也挺重要的，主要就是充实充实设备的每个接口并提交给设备模型，为它们寻找命中注定的接口驱动，最后温存温存，过了这个后戏阶段，usb_generic_driver 也就彻底从你设备那儿得到满足了，generic_probe 的历史使命也就完成了。事物的发展大体上就脱离不了这三个阶段，再比如股票，买、卖、回味。

先看第一阶段，要做一个好男人，按顺序一步一步来。1437 行，configuration 是前边儿 choose_configuration()那里返回回来的，找到合意的配置的话，就返回那个配置的 bConfigurationValue 值，没有找到称心的配置的话，就返回-1，所以这里的 configuration 值就可能有两种情况，或者为-1，或者为配置的 bConfigurationValue 值。当 configuration 为-1 时这里为啥又要把它改为 0 捏？要知道 configuration 这个值是要在后面的高潮阶段里发送 SET_CONFIGURATION 请求时用的，关于 SET_CONFIGURATION 请求，spec 里说，这个值必须为 0 或者与配置描述符的 bConfigurationValue 一致，如果为 0，则设备收到 SET_CONFIGURATION 请求后，仍然会待在 Address 状态。这里当 configuration 为-1 也就是没有发现满意的配置时，设备不能进入 Configured，所以要把 configuration 的值改为 0，以便满足 SET_CONFIGURATION 请求的要求。

那接下来的问题就出来了，在没有找到合适配置的时候直接给 configuration 这个参数传个 0，也就是让 choose_configuration()返回个 0 不就得了，干吗还这么麻烦先返回个-1 再把它改成 0，不是脱裤子放屁多此一举么？你别愤怒，这归根结底还是那句话，林子大了什么鸟都有，有些设备就是有拿 0 当配置 bConfigurationValue 值的毛病，你又不能不让它用，不是有句俗话么：既然说人生是本书，那出现几个错别字就没什么大不了的。人生都能出现错字，那 usb 世界里出现几个有毛病的设备也没啥大不了的，这里妥协一下就是了，想让设备回到 Address 状态时，usb_set_configuration()就别传递 0 了，传递个-1，里边儿去处理一下。如果 configuration 值为 0 或大于 0 的值，就从设备 struct usb_device 结构体的 config 数组里将相应配置的描述信息，也就是 struct usb_host_config 结构体给取出来。

1448 行，如果没有拿到配置的内容，configuration 值就必须为 0 了，让设备待在 Address 那儿别动。这也很好理解，配置的内容都找不到了，还配置个什么劲儿。当然，如果拿到了配置的内容，但同时 configuration 为 0，这就是对应了上面说的那种有毛病的设备的情况，就提出一下警告，告诉你不正常现象出现了。

1461 行，过了这个 if，第一阶段就告结束了。如果配置是实实在在存在的，就为它使用的那些接口都准备一个 struct usb_interface 结构体。new_interfaces 是开头儿就定义好的一个 struct usb_interface 结构体指针数组，数组的每一项都指向了一个 struct usb_interface 结构体，所以这里申请内存也要分两步走，先申请指针数组的，再申请每一项的。

驱动的生命线（三）

准备工作该做的都做了，别嫌太麻烦，什么事情都要经过这么一个阶段，大家都明白。现在看看第二阶段的重头戏，看看设备是怎么从 Address 进入 Configured 的。1501 行，这行主要就是对那些刚出去偷过情的人说的，如果已经在 Configured 状态了，就得做些清理工作，别被老婆发现了，要装作若无其事的退回到 Address 状态。都清理些什么怎么去清理？别着急，要想学会，得仔细研究下 message.c 里的 usb_disable_device 函数。

```
1024 /*
1025  * usb_disable_device - Disable all the endpoints for a USB device
1026  * @dev: the device whose endpoints are being disabled
1027  * @skip_ep0: 0 to disable endpoint 0, 1 to skip it.
1028  *
1029  * Disables all the device's endpoints, potentially including endpoint 0.
1030  * Deallocates hcd/hardware state for the endpoints (nuking all or most
1031  * pending urbs) and usbcore state for the interfaces, so that usbcore
1032  * must usb_set_configuration() before any interfaces could be used.
1033  */
1034 void usb_disable_device(struct usb_device *dev, int skip_ep0)
1035 {
1036         int i;
1037
1038         dev_dbg(&dev->dev, "%s nuking %s URBs\n", __FUNCTION__,
1039                         skip_ep0 ? "non-ep0" : "all");
1040         for (i = skip_ep0; i < 16; ++i) {
1041                 usb_disable_endpoint(dev, i);
1042                 usb_disable_endpoint(dev, i + USB_DIR_IN);
1043         }
1044         dev->toggle[0] = dev->toggle[1] = 0;
```

```
1045
1046         /* getting rid of interfaces will disconnect
1047          * any drivers bound to them (a key side effect)
1048          */
1049         if (dev->actconfig) {
1050                 for (i = 0; i < dev->actconfig->desc.bNumInterfaces; i++) {
1051                         struct usb_interface    *interface;
1052
1053                         /* remove this interface if it has been registered */
1054                         interface = dev->actconfig->interface[i];
1055                         if (!device_is_registered(&interface->dev))
1056                                 continue;
1057                         dev_dbg (&dev->dev, "unregistering interface %s\n",
1058                                 interface->dev.bus_id);
1059                         usb_remove_sysfs_intf_files(interface);
1060                         device_del (&interface->dev);
1061                 }
1062
1063                 /* Now that the interfaces are unbound, nobody should
1064                  * try to access them.
1065                  */
1066                 for (i = 0; i < dev->actconfig->desc.bNumInterfaces; i++) {
1067                         put_device (&dev->actconfig->interface[i]->dev);
1068                         dev->actconfig->interface[i] = NULL;
1069                 }
1070                 dev->actconfig = NULL;
1071                 if (dev->state == USB_STATE_CONFIGURED)
1072                         usb_set_device_state(dev, USB_STATE_ADDRESS);
1073         }
1074 }
```

经过研究我们可以发现，usb_disable_device 函数的清理工作主要有两部分，一是将设备里所有端点给 disable 掉，一是将设备当前配置使用的每个接口都从系统里给 unregister 掉，也就是将接口和它对应的驱动给分开。前面不是说要那些行为不端背着老婆玩儿偷情的男人要向这个函数学习么，学习什么？就是也要分两个方面做清理工作，一是喷点香水或者洒点酒气什么的，将谁谁的味道给 disable 掉，去掉，一是从手机电脑等里面将谁谁的暧昧信息给删掉，将自己和那个谁谁的关系给断掉，省得老婆发现了伤心，要爱惜老婆爱惜生命。

先说下第二部分的工作，1409 行，actconfig 表示的是设备当前激活的配置，只有它不为空时才有接下来清理的必要。

1050~1061 这个 for 循环就是将这个配置的每个接口从设备模型的体系中删除掉，将它们和对应的接口驱动分开，没有驱动了，这些接口也就丧失了能力，当然也就什么作用都发挥

不了了，这也是名字里那个 disable 的真正含意所在。

1066~1070 行，将 actconfig 的 interface 数组置为空，然后再将 actconfig 置为空，这里你可能会有的一个问题是，为什么只是置为空，既然要清理 actconfig，为什么不直接将它占用的内存给释放掉？这个问题问的好，说明你足够细心，不过你应该注意到 actconfig 只是一个指针，一个地址，你应该首先弄清楚这个地址里保存的是什么东西再决定是不是将它给释放掉，那这个指针指向哪儿？它指向设备 struct usb_device 结构的 config 数组里的其中一项，当前被激活的是哪一个配置，它就指向 config 数组里的哪一项，你这里只是不想让设备当前激活任何一个配置而已，没必要将 actconfig 指向的那个配置给释放掉吧，前面在设备生命线那里走了那么久，历尽千辛万苦才将设备各个配置的内容给拿过来放到 config 数组里，你这里如果给释放掉，对得起谁啊，岂不要哭死。

那这么说的话另一个问题就出来了，既然 actconfig 指向了 config 里的一项，那为什么要把那个配置的 interface 数组给置为空，这不是修改了那个配置的内容，从而也修改了 config 数组的内容么？你先别着急，俺帮你回忆一下，在设备生命线那里取配置描述符的，解析返回的那堆数据时，只是把每个配置里的 cache 数组，也就是 intf_cache 数组给初始化了，并没有为 interface 数组充实任何的内容，这里做清理工作的目的就是要恢复原状，当然要将它置为空了，那么配置的 interface 数组又在哪里被充实了那？usb_set_configuration 函数里第二个高潮阶段之后不是还有个第三个阶段么，就在那里，你那时激活了哪个配置，就为哪个配置的 interface 数组动手术，填点东西。

1071 行，如果这个设备此时确实是在 Configured 状态，就让它回到 Address。

现在回头来说说第一部分的清理工作。这个部分主要就是为每个端点调用了 usb_disable_endpoint 函数，将挂在它们上面的 urb 给取消掉。为什么要这么做？你想想，能调用到 usb_disable_device 这个函数，一般来说设备的状态要发生变化了，设备的状态都改变了，那设备的那些端点的状态要不要改变？还有挂在它们上面的那些 urb 需不需要给取消掉？这些都是很显然的事情，就拿现在让设备从 Configured 回到 Address 来说吧，在 Address 的时候，你只能通过缺省管道也就是端点 0 对应的管道与设备进行通信的，但是在 Configured 的时候，设备的所有端点都是能够使用的，它们上面可能已经挂了一些 urb 正在处理或者将要处理，那么你这时让设备要从 Configured 变到 Address，是不是应该先将这些 urb 给取消掉？

还有个问题是参数 skip_ep0 是嘛意思？这里 for 循环的 i 是从 skip_ep0 开始算起，也就是说 skip_ep0 为 1 的话，就不需要对端点 0 调用 usb_disable_endpoint 函数了，按常理来说，设备状态改变了，是需要把每个端点上面的 urb 给取消掉的，这里面当然也要包括端点 0，但是写代码的哥们儿这里搞出个 skip_ep0 自然有他们的玄机，蓦然回首一下，usb_set_configuration()调用这个函数的时候参数 skip_ep0 的值是什么？是 1，因为这时候是从 Configured 回到 Address，这个过程中，其它端点是从能够使用变成了不能使用，但端点 0 却是一直都很强势，虽说是设备发生了状态的变化，但在这两个状态里它都是要正常使用的，所以就没必要 disable 它了。

什么时候需要 disable 端点 0？目前版本的内核里俺只发现了两种情况，一是设备要断开的时候，一是设备从 Default 进化到 Address 的时候，虽说不管是 Default 还是 Address，端点 0 都是需要能够正常使用的，但因为地址发生改变了，毫无疑问，你需要将挂在它上面的 urb 清除掉。俺当时讲设备生命线的时候，在设置完设备地址，设备进入 Address 后，第二种情况的这个步骤给有意无意的飘过了，主要是当时也不影响理解，现在既然遇到了，就把它给补上吧。

在设备生命线的那个过程中，设置完设备地址，让设备进入 Address 状态后，立马就调用了 hub.c 里一个名叫 ep0_reinit 的函数

```
2066 static void ep0_reinit(struct usb_device *udev)
2067 {
2068         usb_disable_endpoint(udev, 0 + USB_DIR_IN);
2069         usb_disable_endpoint(udev, 0 + USB_DIR_OUT);
2070         udev->ep_in[0] = udev->ep_out[0] = &udev->ep0;
2071 }
```

这个函数里只对端点 0 调用了 usb_disable_endpoint()，但是端点 0 接下来还是要使用的,不然你就取不到设备那些描述符了,所以接着重新将 ep0 赋给 ep_in[0]和 ep_out[0]。多说无益，还是到 usb_disable_endpoint()里面去看看吧。

```
975 /**
976  * usb_disable_endpoint -- Disable an endpoint by address
977  * @dev: the device whose endpoint is being disabled
978  * @epaddr: the endpoint's address.  Endpoint number for output,
979  *      endpoint number + USB_DIR_IN for input
980  *
981  * Deallocates hcd/hardware state for this endpoint ... and nukes all
982  * pending urbs.
983  *
984  * If the HCD hasn't registered a disable() function, this sets the
985  * endpoint's maxpacket size to 0 to prevent further submissions.
986  */
987 void usb_disable_endpoint(struct usb_device *dev, unsigned int epaddr)
988 {
989         unsigned int epnum = epaddr & USB_ENDPOINT_NUMBER_MASK;
990         struct usb_host_endpoint *ep;
991
992         if (!dev)
993                 return;
994
995         if (usb_endpoint_out(epaddr)) {
996                 ep = dev->ep_out[epnum];
997                 dev->ep_out[epnum] = NULL;
```

```
998         } else {
999                 ep = dev->ep_in[epnum];
1000                 dev->ep_in[epnum] = NULL;
1001         }
1002         if (ep && dev->bus)
1003                 usb_hcd_endpoint_disable(dev, ep);
1004 }
```

这个函数先获得端点号和端点的方向，然后从 ep_in 或 ep_out 两个数组里取出端点的 struct usb_host_endpoint 结构体，并将数组里的对应项置为空，要注意的是这里同样不是释放掉数组里对应项的内存而是置为空。这两个数组里的 ep_in[0]和 ep_out[0]是早就被赋值了，至于剩下的那些项是在什么时候被赋值的，又是指向了什么东西，就是 usb_set_configuration 函数第三个阶段的事了。

不要怪俺说得比较粗略，只是都在前面说过了，你既然已经看到这里了，只要用过那么一点点心就会明白这里是什么意思。呼吁一下：要用心啊，要用心去爱你的邻居，不过不要让她的老公知道。

最后 1003 行调用了一个 usb_hcd_endpoint_disable 函数，主要的工作还得它来做，不过这已经深入 HCD 的腹地了，就不多说了，还是飘回 usb_disable_device()吧。在为每个端点都调用了 usb_disable_endpoint()之后，还有一个小步骤要做，就是将设备 struct usb_device 结构体的 toggle 数组置为 0。至于 toggle 数组干吗的，为啥要被初始化为 0，你还是蓦然回首到设备那节去看吧。俺要接着飘回 usb_set_configuration()了。

1504 行，又一次与熟悉的陌生人 usb_control_msg()相遇了，每当我们需要向设备发送请求的时候它就会适时的出现，我们每个人是不是也都希望在自己的生活里有这么一个角色？当你需要的时候，她就会出现在你身边，你快乐的时候，她能够分享你的快乐，你痛苦的时候，她也能够给你慰藉，你无聊的时候，她能够陪你去逛衡山路，你烦躁的时候，她能够用自己的柔情平和你的心情。

usb_control_msg 这次出现的目的当然是为了 SET_CONFIGURATION 请求，这里只说一下它的那堆参数，看一下 spec 9.4.7 的那张表

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None

SET_CONFIGURATION 请求不需要 DATA transaction，而且还是协议里规定所有设备都要支持的标准请求，也不是针对端点或者接口什么的，而是针对设备的，所以 bRequestType 只能为 0x80，就是上面表里的 00000000B，也就是 1505 行的第一个 0，wValue 表示配置的 bConfigurationValue，就是 1505 行的 configuration。

1514 行，将激活的那个配置的地址赋给 actconfig。如果 cp 为空，重新设置设备的状态为 Address，并将之前准备的那些 struct usb_interface 结构体和 new_interfaces 释放掉，然后返回。扫一下前面的代码，cp 有三种可能为空，一是参数 configuration 为-1，一是参数 configuration 为 0，且从设备的 config 数组里拿出来的就为空，一是 SET_CONFIGURATION 请求除了问题。不管怎么说，走到 1515 行，cp 还是空的，你就要准备返回了。

1520 行，事情在这里发展达到了高潮的顶端，设置设备的状态为 Configured。

驱动的生命线（四）

设备自从有了 Address，拿到了各种描述符，就在那儿看 usb_generic_driver 忙活了，不过还算没白忙，设备总算是幸福的进入 Configured 了。从设备这儿咱们应该学到点幸福生活的秘诀，就是找到你所喜欢的事，然后找到愿意为你来做这件事的人。

Address 有点像你合几代人之力辛辛苦苦才弄到的一套新房子，如果不装修，它对你来说的意义也就是你的户口可以从人才中心挪过去了，可以对人说你的地址是哪儿哪儿了，可实际上你在里边儿什么也干不了，你还得想办法去装修它，Configured 就像是已经装修好的，装修好了，你和你老婆才能够在里边儿想干做什么就干什么，下面咱就看看 usb_generic_driver 又对设备做了些什么。

1525 行，nintf 就是配置里接口的数目，那么这个 for 循环显然就是在对配置里的每个接口做处理。

1530 行，这里要明白的是，cp 里的两个数组 interface 和 intf_cache，一个是没有初始化的，一个是已经动过手术很饱满的。正像前面提到的，这里需要为 interface 动手术，拿 intf_cache 的内容去充实它。

1536 行，获得这个接口的 0 号设置。咱们已经知道，因为某些厂商有特殊的癖好，导致了 struct usb_host_config结构里的数组interface并不一定是按照接口号的顺序存储的，你必须使用 usb_ifnum_to_if()来获得指定接口号对应的struct usb_interface结构体。现在咱们需要再多知道一点，接口里的altsetting数组也不一定是按照设置编号来顺序存储的，你必须使用usb_altnum_to_altsetting()来获得接口里的指定设置。它在usb.c里定义

```
100 /**
101  * usb_altnum_to_altsetting - get the altsetting structure with a given
102  *      alternate setting number.
103  * @intf: the interface containing the altsetting in question
```

```
104  * @altnum: the desired alternate setting number
105  *
106  * This searches the altsetting array of the specified interface for
107  * an entry with the correct bAlternateSetting value and returns a pointer
108  * to that entry, or null.
109  *
110  * Note that altsettings need not be stored sequentially by number, so
111  * it would be incorrect to assume that the first altsetting entry in
112  * the array corresponds to altsetting zero.  This routine helps device
113  * drivers avoid such mistakes.
114  *
115  * Don't call this function unless you are bound to the intf interface
116  * or you have locked the device!
117  */
118 struct usb_host_interface *usb_altnum_to_altsetting(const struct
usb_interface *intf,
119                                                     unsigned int altnum)
120 {
121         int i;
122
123         for (i = 0; i < intf->num_altsetting; i++) {
124                 if (intf->altsetting[i].desc.bAlternateSetting == altnum)
125                         return &intf->altsetting[i];
126         }
127         return NULL;
128 }
```

这个函数依靠一个 for 循环来解决问题，就是轮询接口里的每个设置，比较它们的编号与你指定的编号是不是一样。原理简单，操作也简单，有点不简单的是前面调用它的时候为什么指定编号 0，也就是获得 0 号设置。这还要归咎于 spec 里说了这么一句，接口的默认设置总是 0 号设置，所以这里的目的就是获得接口的默认设置，如果没有拿到设置 0，接下来就在 1544 行拿 altsetting 数组里的第一项来充数。范伟说了，没有 IP 卡，IQ 卡也成。

1546 行，指定刚刚拿到的设置为当前要使用的设置。

1547 行，前边儿遇到过 device 和 endpoint 的 disable 函数，这里遇到个 interface 的 enable 函数，同样在 message.c 里定义

```
1104 /*
1105  * usb_enable_interface - Enable all the endpoints for an interface
1106  * @dev: the device whose interface is being enabled
1107  * @intf: pointer to the interface descriptor
1108  *
1109  * Enables all the endpoints for the interface's current altsetting.
```

```
1110  */
1111 static void usb_enable_interface(struct usb_device *dev,
1112                                  struct usb_interface *intf)
1113 {
1114         struct usb_host_interface *alt = intf->cur_altsetting;
1115         int i;
1116
1117         for (i = 0; i < alt->desc.bNumEndpoints; ++i)
1118                 usb_enable_endpoint(dev, &alt->endpoint[i]);
1119 }
```

这个函数同样也是靠一个 for 循环来解决问题，轮询前面获得的那个接口设置使用到的每个端点，调用 message.c 里的 usb_enable_endpoint()将它们统统 enable。

```
1077 /*
1078  * usb_enable_endpoint - Enable an endpoint for USB communications
1079  * @dev: the device whose interface is being enabled
1080  * @ep: the endpoint
1081  *
1082  * Resets the endpoint toggle, and sets dev->ep_{in,out} pointers.
1083  * For control endpoints, both the input and output sides are handled.
1084  */
1085 static void
1086 usb_enable_endpoint(struct usb_device *dev, struct usb_host_endpoint *ep)
1087 {
1088         unsigned int epaddr = ep->desc.bEndpointAddress;
1089         unsigned int epnum = epaddr & USB_ENDPOINT_NUMBER_MASK;
1090         int is_control;
1091
1092         is_control = ((ep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
1093                         == USB_ENDPOINT_XFER_CONTROL);
1094         if (usb_endpoint_out(epaddr) || is_control) {
1095                 usb_settoggle(dev, epnum, 1, 0);
1096                 dev->ep_out[epnum] = ep;
1097         }
1098         if (!usb_endpoint_out(epaddr) || is_control) {
1099                 usb_settoggle(dev, epnum, 0, 0);
1100                 dev->ep_in[epnum] = ep;
1101         }
1102 }
```

这个函数的前边儿几行没什么好说的，分别获得端点地址，端点号，还有是不是控制端点。稍微有那么点嚼头儿的是后面的两个 if，它们分别根据端点的方向来初始化设备的 ep_in 和 ep_out 数组，还有就是调用了 include/linux/usb.h 里的一个叫 usb_settoggle 的宏。

```
1424 /*The DO/D1 toggle bits ... USE WITH CAUTION (they're almost hcd-internal) */
1425 #define usb_gettoggle(dev, ep, out) (((dev)->toggle[out] >> (ep)) & 1)
1426 #define usb_dotoggle(dev, ep, out)  ((dev)->toggle[out] ^= (1 << (ep)))
1427 #define usb_settoggle(dev, ep, out, bit) \
1428                 ((dev)->toggle[out] = ((dev)->toggle[out] & ~(1 << (ep))) | \
1429                   ((bit) << (ep)))
```

这三个宏都是用来处理端点的 toggle 位的，也就是 struct usb_device 里的数组 toggle[2]。toggle[0]对应的是 IN 端点，toggle[1]对应的是 OUT 端点，上面宏参数里的 out 用来指定端点是 IN 还是 OUT，ep 指的不是端点的结构体，而仅仅是端点号。usb_gettoggle 用来得到端点对应的 toggle 值，usb_dotoggle 用来将端点的 toggle 位取反，也就是原来为 1 就置为 0，原来为 0 就置为 1，usb_settoggle 看起来就要复杂点，意思是如果 bit 为 0,则将 ep 所对应的那个 toggle 位 reset 成 0,如果 bit 为 1,则 reset 为 1。((dev)->toggle[out] & ~(1<<(ep)))就是把 1 左移 ep 位，比如 ep 为 3，那么就是得到了 1000，然后取反，得到 0111，（当然高位还有更多个 1），然后(dev)->toggle[out]和 0111 相与，这就是使得 toggle[out]的第 3 位清零而其它位都不变。咱们这里调用 usb_settoggle 的时候，bit 传递的是 0，用来将端点的 toggle 位清零，原因早先也提到过，就是对于批量传输、控制传输和中断传输来说，数据包最开始都是被初始化为 DATA0 的，然后才一次传 DATA0，一次传 DATA1。

现在看看为什么两个 if 语句里都出现有 is_control。控制传输使用的是 message 管道，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。这里使用两个 if，而不是 if-else 组合，目的就是塞进去一个 is_control，表示只要是控制端点，就将它的端点号对应的 IN 和 OUT 两个方向上的 toggle 位还有 ep_int 和 ep_out 都给初始化了。当然，所谓的控制端点一般也就是指端点 0。

endpoint 的 enable 函数要比 disable 函数清爽的多，disable 的时候还要深入到 HCD 的腹地去撤销挂在它上面的各个 urb，而 enable 的时候就是简单设置一下 toggle 位还有那两个数组就好了，要知道它的 urb 队列 urb_list 是早在从设备那里获取配置描述符并去解析那一大堆数据的时候就初始化好了的。enable 之后，接口里的各个端点便都处于了欣欣向荣的可用状态，你就可以在驱动里向指定的端点提交 urb 了。当然，到目前这个时候接口还仍然是接口，驱动（接口驱动）还仍然是驱动，它们中间还缺少那根著名的红线。

然后看看跟在 usb_enable_interface()后面的那几行，接口所属的总线类型仍然为 usb_bus_type，设备类型变为 usb_if_device_type，dma_mask 被设置为你的设备的 dma_mask，而你设备的 dma_mask 很早以前就被设置为了 host controller 的 dma_mask。

1553 行，device_initialize 在初始化设备 struct usb_device 结构体的时候遇到过一次，这里初始化接口的时候又遇到了。

1554 行，将接口的 is_active 标志初始化为 0，表示它还没有与任何驱动绑定，就是还没有找到另一半儿。mark_quiesced 在 usb.h 里定义

```
105 static inline void mark_quiesced(struct usb_interface *f)
106 {
107         f->is_active = 0;
108 }
```

1559 行，for 循环结束了，new_interfaces 的历史使命也就结束了。我想你应该会明白的是，这里的 kfree 释放的只是 new_interfaces 指针数组的内存，而不包括它里面各个指针所指向的内存，至于那些数据，都已经在前面被赋给配置里的 interface 数组了。

1561 行，获得配置的字符串描述符，怎么获得？先飘过去把剩下的说了再说它。

1570 行，这个 for 循环结束，usb_set_configuration()的三个阶段也就算结束了，设备和大美女 usb_generic_driver 上上下下忙活了这么久也都很累了，接下来就该接口和接口驱动去忙活了。

这个for循环将前面那个for循环准备好的每个接口送给设备模型，Linux设备模型会根据总线类型usb_bus_type将接口添加到usb总线的那条有名的设备链表里，然后去轮询usb总线的另外一条有名的驱动链表，针对每个找到的驱动去调用usb总线的match函数，也就是usb_device_match()，去为接口寻找另一个匹配的半圆。你说这个时候设备和接口两条路它应该走哪条？它的类型已经设置成usb_if_device_type了，设备那条路把门儿的根本就不会让它进，所以它必须得去走接口那条路。

字符串描述符

关于字符串描述符，前面的前面已经简单描述过了，但是因为现在长夜漫漫，孤枕难眠，所以多说点。字符串描述符的地位仅次于设备/配置/接口/端点四大描述符，那四大设备必须得支持，而字符串描述符对设备来说则是可选的，类似于网球里马德里大师赛与四大满贯之间的地位差异，四大满贯是个碗儿都争着抢着爬着也要去，而号称第五大满贯的马德里大师赛却动不动就会被大碗儿们因各种理由给涮了。

这并不是就说字符串描述符不重要，对咱们来说，字符串要比数字亲切的多，提供字符串描述符的设备也要比没有提供的设备亲切的多，不会有人会专门去记前面使用 lsusb 列出的 04b4 表示的是 Cypress Semiconductor Corp.，如果谁真费心去记了，俺也不会 pfpf，而是只会发出和看到 06 年那几个超女时一样的感叹：动物的种类在减少，人的种类在增加吗？

一提到字符串，不可避免就得提到字符串使用的语言。字符串亲切是亲切，但不像数字那样全球通用，使用中文了，老外看不懂，使用法文阿拉伯文什么的咱又看不懂，你知道目前世界上有多少种语言吗？有得说七千多种，有得说五千多种，无一定论，不过使用人口超过 100 万的语言也足足有 140 多种。字符串描述符也需要应对多种语言的情况，当然这并不是说设备里就要存储这么多不同语言的字符串描述符，这未免要求过高了些，代价也忒昂贵了些，要知道这些描述符不会凭空生出来，是要存储在设备的 EEPROM 里的，此物是需要 MONEY 的，现在都在提倡节约型社会，要节约 MONEY，尽量少占用 EEPROM，要节约用水，尽量和女友一起洗澡。所以说只提供几种语言的字符串描述符就可以了，甚至说只提供一种语言，比如英语就可以了。

其实咱们现在说的语言的问题和秦始皇统一六国时遇到的语言问题一样，就是太多了，鸡说鸡的，鸭说鸭的，交流成问题。你说啥时候地球上或整个宇宙上只说一种语言了，咱也不用费劲去学什么英语了，就是外星人来了，大家和外星 mm 交流也不成问题。不过不管哪种语言，在 PC 里或者设备里存放都只能用二进制数字，这就需要在语言与数字之间进行编码，这个所谓的编码和这个世界上其它事物一样，都是有多种的，你说连人的种类都有很多了，人造出来的编码种类能没有很多么？起码每种语言都会存在独立的编码方式，咱们的简体中文可以使用 GB2312、GBK、GB18030 等，台湾那边儿是繁体，用的就是 big5，这么一来每种语言自己内部交流是不成问题了，可相互之间就像鸡同鸭讲了。于是世界上的某些角落就出现了那么一些有志青年，立志要将各种语言的编码体系给统一起来，于是就出现了 UNICODE 和 ISO-10646。比起他们，俺是太没追求了，父亲问我人生有什么追求？我回答金钱和美女，父亲凶狠的打了我的脸，我回答事业与爱情，父亲赞赏的摸了我的头。

Spec里就说了，字符串描述符使用的就是UNICODE编码，usb设备里的字符串可以通过它来支持多种语言，不过你需要在向设备请求字符串描述符的时候指定一个你期望看到的一种语言，俗称语言ID，即Language ID。这个语言ID使用两个字节表示，所有可以使用的语言ID在 http://www.usb.org/developers/docs/USB_LANGIDs.pdf 文档里都有列出来，从这个文档里你也可以明白为啥要使用两个字节，而不是一个字节表示。这么说吧，比如中文是 0X04，但是中文还有好几种，所以就需要另外一个字节表示是哪种中文，简体就是 0X02，注意合起来表示简体中文并不是 0X0402 或者 0X0204，因为这两个字节并不是分的那么清，bit0~9 一共 10 位去表示Primary语言ID，其余 6 位去表示Sub语言ID，毕竟一种语言的Sub语言不可能会特别的多，没必要分去整整一半 8bits，所以简体中文的语言ID就是 0X0804。

不多罗唆，还是结合代码，从上节飘过的 usb_cache_string 说起，看看如何去获得一个字符串描述符，它在 message.c 里定义

```
817 /**
818  * usb_cache_string - read a string descriptor and cache it for later use
819  * @udev: the device whose string descriptor is being read
820  * @index: the descriptor index
821  *
822  * Returns a pointer to a kmalloc'ed buffer containing the descriptor string,
```

```
823  * or NULL if the index is 0 or the string could not be read.
824  */
825 char *usb_cache_string(struct usb_device *udev, int index)
826 {
827         char *buf;
828         char *smallbuf = NULL;
829         int len;
830
831         if (index > 0 && (buf = kmalloc(256, GFP_KERNEL)) != NULL) {
832                 if ((len = usb_string(udev, index, buf, 256)) > 0) {
833                         if ((smallbuf = kmalloc(++len, GFP_KERNEL)) == NULL)
834                                 return buf;
835                         memcpy(smallbuf, buf, len);
836                 }
837                 kfree(buf);
838         }
839         return smallbuf;
840 }
```

每个成年人都有那么一个身份证号码，每个字符串描述符都有一个序号，身份证号可能会重复，字符串描述符这个序号是不能重复的，不过这点不用你我操心，都是设备已经固化好了的东西，重复不重复也不是咱们要操心的事。咱们要操心的事太多了，要操心吃还要操心睡，加菲猫告诉我们，除了吃和睡，生命或许还有别的意义，不过我觉得没有就挺好。

也好理解，什么东西一多了，最好最节约最省事的区分方式就是编号，字符串描述符当然可以有很多个，参数的 index 就是表示了你希望获得其中的第几个。但是不可疏忽大意的是，你不能指定 index 为 0，0 编号是有特殊用途的，你指定 0 了就什么也得不到。你去华为找工号 000 的，不会有人应你，根本就没这人，你找 001，这次有人应你，不过是保安，赶你走的，没事儿找任老总干吗，没看一个接一个的自杀正是多事之秋么。

有关这个函数，还需要明白两点，第一是它采用的方针策略，就是苦活儿累活儿找个 usb_string()去做，自己一边儿看着，加菲猫还告诉我们，工作好有意思耶！尤其是看着别人工作。这个 usb_string()怎么工作的之后再看，现在只要注意下它的参数，比 usb_cache_string()的参数多了两个，就是 buf 和 size，也就是需要传递一个存放返回的字符串描述符的缓冲区。但是你调用 usb_cache_string()的时候并没有指定一个明确的 size，usb_cache_string()也就不知道你想要的那个字符串描述符有多大，于是它就采用了这么一个技巧，先申请一个足够大的缓冲区，这里是 256 字节，拿这个缓冲区去调用 usb_string()，通过 usb_string()的返回值会得到字符串描述符的真实大小，然后再拿这个真实的大小去申请一个缓冲区，并将大缓冲区里放的字符串描述符数据拷贝过来，这时那个大缓冲区当然就没什么利用价值了，于是再把它给释放掉。

第二就是申请那个小缓冲区的时候，使用的并不是 usb_string()的返回值，而是多了 1 个

字节，也就是说要从大缓冲区里多拷一个字节到小缓冲区里，为什么？这牵涉到 C 里字符串方面那个人见人愁鬼见鬼哭的代码杀手——字符串结束符。如果你说俺是危言耸听夸大其实，那只能说明你不是天才就是 C 代码写的少，咱不说 C++，因为 C++里更多的是用 string。

字符串都需要那么一个结束符，这点是个正常人都知道的，但并不是每个正常人都能每时每刻的记得给字符串加上这么一个结束符。就像是个人都知道钞票不是万能的，但并不是每个人都知道：钞票不是万能的，有时还需要信用卡。可能你小心了 1000 次，但在第 1001 次的时候你给忘记了，你的代码就可能就可能挂了。不说玄乎儿的了，搞点实在的，给个例子

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[])
5 {
6       #define MAX (100)
7
8       char buf[512], tmp[32];
9       int i, count = 0;
10
11      for ( i = 0; i< MAX; ++i){
12              sprintf(tmp, "0x%.4X,", i);
13
14              strcat(buf, tmp);
15
16              if (count++ == 10){
17                      printf("%s\n", buf);
18                      buf[0] = '\0';
19                      count = 0;
20                      }
21      }
22
23      return 0;
24 }
```

这程序简单直白，打印 100 个数，每行 10 个，你觉得它有什么毛病没？当然俺不是说算法上的问题，这本来就演示用的，你只需要看看它能不能得到预期的结果。

俺就直说了吧，这程序问题大大的，在第 10 行少了这么一句

```
        buf[0] = '\0'; ////////////////// here !
```

就是说忘记将 buf 初始化了，传递给 strcat 的是一个没有初始化的 buf，这个时候 buf 的

内容都非 0，strcat 不知道它的结尾在哪里，它能猜到 buf 的开始，却猜不到 buf 的结束，多伤感啊。当然初始化的方法有多种，你可以使用 memset 将整个 buf 初始化为 0，但是如果 buf 比较大的话，memset 就比较的消耗 CPU 了，要时刻牢记咱们要创建一个节约型的社会，而这里 buf[0] = '\0'就足够用了，所以没必要使用 memset 去全方位的搞一下。为了更好的说明问题，这里贴一下内核里对 strcat 的定义

```
171 char *strcat(char *dest, const char *src)
172 {
173         char *tmp = dest;
174
175         while (*dest)
176                 dest++;
177         while ((*dest++ = *src++) != '\0')
178                 ;
179         return tmp;
180 }
```

strcat 会从 dest 的起始位置开始去寻找一个字符串的结束符，只有找到，175 行的 while 循环才会结束。“骰子已经掷了，就这样吧！”义无返顾的恺撒毅然跨越卢比肯河，一举击溃了庞培，最终占领了罗马古城。但是如果 dest 没有初始化过，义无反顾的 strcat 并不会有好运得到恺撒的结果。本来 strcat 的目的是将 tmp 追加到 buf 里字符串的后面，但是因为 buf 没有初始化，没有那么一个结束符，strcat 就会一直的找下去，就算它运气好在哪儿停了下来，如果这个停下的位置超出了 buf 的范围，就会把 src 的数据写到不应该的地方，就可能会破坏其它可能很重要的数据，你的系统可能就挂掉了。

解决这种问题的方法很简单，就是记着遇到指针、数组什么的使用前首先统统的初始化掉，到最后真觉得哪里不必要影响性能了再去优化它，还是那句至理名言：过早优化便是罪。

问个大家都会笑的问题，这个字符串结束符具体是什么东东？C 和 C++里一般是指'\0'，像上面的那句 buf[0] = '\0'。这里再引用 spec 里的一句话：The UNICODE string descriptor is not NULL-terminated. 什么是 NULL-terminated 字符串？其实就是以'\0'结尾的字符串，咱们看看内核里对 NULL 的定义

```
6 #undef NULL
7 #if defined(__cplusplus)
8 #define NULL 0
9 #else
10 #define NULL ((void *)0)
11 #endif
```

春春是一个女人，但她又不仅仅是一个女人，0 是一个整数，但它又不仅仅是一个整数。由于各种标准转换，0 可以被用于作为任意的整型、浮点类型、指针。0 的类型将由上下文来确定。典型情况下 0 被表示为一个适当大小的全零二进制位的模式。所以，无论 NULL 是

定义为常数 0 还是((void *)0)这个零指针，NULL 都是指的是 0 值，而不是非 0 值。而字符的'\0'和整数的 0 也可以通过转型来相互转换。再多说一点，'\0'是 C 语言定义的用'\'加 8 进制 ASCII 码来表示字符的一种方法，'\0'就是表示一个 ASCII 码值为 0 的字符。

所以更准确的说，NULL-terminated 字符串就是以 0 值结束的字符串，那么 spec 的那句话说字符串描述符不是一个 NULL-terminated 字符串，意思也就是字符串描述符没有一个结束符，你从设备那里得到字符串之后得给它追加一个结束符。本来 usb_string()里已经为 buf 追加好了，但是它返回的长度里还是没有包括进这个结束符的 1 个字节，所以 usb_cache_string()为 smallbuf 申请内存的时候就得多准备那么一个字节，以便将 buf 里的那个结束符也给拷过来。现在就看看 usb_string()的细节，定义在 message.c 里

```
733 /**
734  * usb_string - returns ISO 8859-1 version of a string descriptor
735  * @dev: the device whose string descriptor is being retrieved
736  * @index: the number of the descriptor
737  * @buf: where to put the string
738  * @size: how big is "buf"?
739  * Context: !in_interrupt ()
740  *
741  * This converts the UTF-16LE encoded strings returned by devices, from
742  * usb_get_string_descriptor(), to null-terminated ISO-8859-1 encoded ones
743  * that are more usable in most kernel contexts.  Note that all characters
744  * in the chosen descriptor that can't be encoded using ISO-8859-1
745  * are converted to the question mark ("?") character, and this function
746  * chooses strings in the first language supported by the device.
747  *
748  * The ASCII (or, redundantly, "US-ASCII") character set is the seven-bit
749  * subset of ISO 8859-1. ISO-8859-1 is the eight-bit subset of Unicode,
750  * and is appropriate for use many uses of English and several other
751  * Western European languages.  (But it doesn't include the "Euro" symbol.)
752  *
753  * This call is synchronous, and may not be used in an interrupt context.
754  *
755  * Returns length of the string (>= 0) or usb_control_msg status (< 0).
756  */
757 int usb_string(struct usb_device *dev, int index, char *buf, size_t size)
758 {
759         unsigned char *tbuf;
760         int err;
761         unsigned int u, idx;
762
763         if (dev->state == USB_STATE_SUSPENDED)
764                 return -EHOSTUNREACH;
765         if (size <= 0 || !buf || !index)
```

```
766                 return -EINVAL;
767         buf[0] = 0;
768         tbuf = kmalloc(256, GFP_KERNEL);
769         if (!tbuf)
770                 return -ENOMEM;
771
772         /* get langid for strings if it's not yet known */
773         if (!dev->have_langid) {
774                 err = usb_string_sub(dev, 0, 0, tbuf);
775                 if (err < 0) {
776                         dev_err (&dev->dev,
777                                 "string descriptor 0 read error: %d\n",
778                                 err);
779                         goto errout;
780                 } else if (err < 4) {
781                         dev_err (&dev->dev, "string descriptor 0 too
short\n");
782                         err = -EINVAL;
783                         goto errout;
784                 } else {
785                         dev->have_langid = 1;
786                         dev->string_langid = tbuf[2] | (tbuf[3]<< 8);
787                                 /* always use the first langid listed */
788                         dev_dbg (&dev->dev, "default language 0x%04x\n",
789                                 dev->string_langid);
790                 }
791         }
792
793         err = usb_string_sub(dev, dev->string_langid, index, tbuf);
794         if (err < 0)
795                 goto errout;
796
797         size--;         /* leave room for trailing NULL char in output buffer
*/
798         for (idx = 0, u = 2; u < err; u += 2) {
799                 if (idx >= size)
800                         break;
801                 if (tbuf[u+1])                  /* high byte */
802                         buf[idx++] = '?';  /* non ISO-8859-1 character */
803                 else
804                         buf[idx++] = tbuf[u];
805         }
806         buf[idx] = 0;
807         err = idx;
```

```
808
809         if (tbuf[1] != USB_DT_STRING)
810                 dev_dbg(&dev->dev, "wrong descriptor type %02x for string %d
(\"%s\")\n", tbuf[1], index, buf);
811
812  errout:
813         kfree(tbuf);
814         return err;
815 }
```

763 行，这几行做些例行检查，设备不能是挂起的，index 也不能是 0 的，只要传递了指针就是需要检查的。

767 行，初始化 buf，usb_cache_string()并没有对这个 buf 初始化，所以这里必须要加上这么一步。当然 usb_string()并不仅仅只有在 usb_cache_string()里调用，可能会在很多地方调用到它，不过不管在哪里，这里谨慎起见，还是需要这么一步。

768 行，申请一个 256 字节大小的缓冲区。前面一直强调说要初始化要初始化，怎么到这里俺就自己打自己一耳光，没有去初始化 tbuf？这是因为没必要，为什么没必要，你看看 usb_string()最后面的那一堆就明白了。

773 行，struct usb_device 里有 have_langid 和 string_langid 这么两个字段是和字符串描述符有关的，string_langid 用来指定使用哪种语言，have_langid 用来指定 string_langid 是否有效。如果 have_langid 为空，就说明没有指定使用哪种语言，那么获得的字符串描述符使用的是哪种语言就完全看设备的脸色和心情了。你可能会疑惑为什么当 have_langid 为空的时候，要在 774 行和 793 行调用两次 usb_string_sub()？就像 usb_string() 是替 usb_cache_string() 做苦工的一样，usb_string_sub() 是替 usb_string()做苦工的，也就是很说 usb_string()是靠 usb_string_sub()去获得字符串描述符的，那问题就变成为什么 have_langid 为空的时候，要获取两遍的字符串描述符？

你可以比较一下两次调用 usb_string_sub()的参数有什么区别，第一次调用的时候，语言 ID 和 index 都为 0，第二次调用的时候就明确的指定了语言 ID 和 index。这里的玄机就在 index 为 0 的时候，也就是 0 编号的字符串描述符是什么东东，前面只说了它有特殊的用途就甩一甩衣袖飘过了，飘得过初一飘不过十五，现在是怎么也飘不过了，必须得解释一下。

有困难要找居委会，有问题就要找协议，spec 9.6.7 里说了，编号 0 的字符串描述符包括了设备所支持的所有语言 ID，对应的就是 Table 9-15

Table 9-15. String Descriptor Zero, Specifying Languages Supported by the Device

Offset	Field	Size	Value	Description
0	*bLength*	1	N+2	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	STRING Descriptor Type
2	*wLANGID[0]*	2	Number	LANGID code zero
...	...	...	...	...
N	*wLANGID[x]*	2	Number	LANGID code *x*

第一次调用 usb_string_sub()就是为了获得这张表，获得这张表做嘛用？你接着往下看。

775 行，usb_string_sub()返回个负数，就表示没有获得这张表，没有取到 0 号字符串描述符。如果返回值比 4 要小，就表示获得的表里没有包含任何一个语言 ID。要知道一个语言 ID 占用 2 个字节，还有前两个字节表示表的长度还有类型，所以得到的数据至少要为 4，才能够得到一个有效的语言 ID。如果返回值比 4 要大，就使用获得的数据的第 3，4 两个字节设置 string_langid，同时设置 have_langid 为 1。现在很明显了，773~791 这一堆就是在你没有指定使用哪种语言的时候，去获取设备里默认使用的语言，也就是 0 号字符串描述符里的第一个语言 ID 所指定的语言。如果没有找到这个默认的语言 ID，即 usb_string_sub()返回值小于 4 的情况，就没有办法再去获得其它的字符串描述符了，因为没有指定语言啊，设备不知道你是要英语还是中文还是别的，它没有办法猜你的心思，应付不来这种无所适从的状况。

793 行，使用指定的语言 ID，或者前面获得的默认语言 ID 去获得想要的那个字符串描述符。现在看看定义在 message.c 里的 usb_string_sub 函数。

```
696 static int usb_string_sub(struct usb_device *dev, unsigned int langid,
697                 unsigned int index, unsigned char *buf)
698 {
699         int rc;
700
701         /* Try to read the string descriptor by asking for the maximum
702          * possible number of bytes */
703         if (dev->quirks & USB_QUIRK_STRING_FETCH_255)
704                 rc = -EIO;
705         else
706                 rc = usb_get_string(dev, langid, index, buf, 255);
707
708         /* If that failed try to read the descriptor length, then
709          * ask for just that many bytes */
710         if (rc < 2) {
711                 rc = usb_get_string(dev, langid, index, buf, 2);
```

```
712                 if (rc == 2)
713                         rc = usb_get_string(dev, langid, index, buf, buf[0]);
714         }
715
716         if (rc >= 2) {
717                 if (!buf[0] && !buf[1])
718                         usb_try_string_workarounds(buf, &rc);
719
720                 /* There might be extra junk at the end of the descriptor */
721                 if (buf[0] < rc)
722                         rc = buf[0];
723
724                 rc = rc - (rc & 1); /* force a multiple of two */
725         }
726
727         if (rc < 2)
728                 rc = (rc < 0 ? rc : -EINVAL);
729
730         return rc;
731 }
```

这个函数首先检查一下你的设备是不是属于那种有怪僻的,如果是一个没有毛病遵纪守法的合格设备，就调用 usb_get_string() 去帮着自己获得字符串描述符。USB_QUIRK_STRING_FETCH_255 就是在 include/linux/usb/quirks.h 里定义的那些形形色色的毛病之一，《我是 Hub》里详细的讲了设备的 quirk，说了 USB_QUIRK_STRING_FETCH_255 就表示设备在获取字符串描述符的时候会 crash。

usb_string_sub()的核心就是 message.c 里定义 usb_get_string 函数

```
642 /**
643  * usb_get_string - gets a string descriptor
644  * @dev: the device whose string descriptor is being retrieved
645  * @langid: code for language chosen (from string descriptor zero)
646  * @index: the number of the descriptor
647  * @buf: where to put the string
648  * @size: how big is "buf"?
649  * Context: !in_interrupt ()
650  *
651  * Retrieves a string, encoded using UTF-16LE (Unicode, 16 bits per character,
652  * in little-endian byte order).
653  * The usb_string() function will often be a convenient way to turn
654  * these strings into kernel-printable form.
655  *
656  * Strings may be referenced in device, configuration, interface, or other
```

```
657  * descriptors, and could also be used in vendor-specific ways.
658  *
659  * This call is synchronous, and may not be used in an interrupt context.
660  *
661  * Returns the number of bytes received on success, or else the status code
662  * returned by the underlying usb_control_msg() call.
663  */
664 static int usb_get_string(struct usb_device *dev, unsigned short langid,
665                         unsigned char index, void *buf, int size)
666 {
667         int i;
668         int result;
669
670         for (i = 0; i < 3; ++i) {
671                 /* retry on length 0 or stall; some devices are flakey */
672                 result = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
673                         USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
674                         (USB_DT_STRING << 8) + index, langid, buf, size,
675                         USB_CTRL_GET_TIMEOUT);
676                 if (!(result == 0 || result == -EPIPE))
677                         break;
678         }
679         return result;
680 }
```

我已经不记得这是第多少次遇到 usb_control_msg()了，佛说前生 500 次的回眸才换得今生的一次擦肩而过，那咱们现在至少也成千上万次回眸了。老习惯，还是简单说一下它的一堆参数，wValue 的高位字节表示描述符的类型，低位字节表示描述符的序号，所以有 674 行的(USB_DT_STRING << 8) + index，wIndex 对于字符串描述符应该设置为使用语言的 ID，所以有 674 行的 langid，至于 wLength，就是描述符的长度，对于字符串描述符很难有一个统一的确定的长度,所以一半来说上头儿传递过来的通常是一个比较大的 255 字节。

和获得设备描述符时一样，因为一些厂商搞出的设备古灵精怪的，可能需要多试几次才能成功。要容许设备犯错误，就像人总要犯错误一样，否则正确之路人满为患了。

还是回过头去看 usb_string_sub 函数，如果 usb_get_string()成功的得到了期待的字符串描述符，则返回获得的字节数，如果这个数目小于 2，就再读两个字节试试，要想明白这两个字节是什么内容，需要看看 spec Table 9-16

Table 9-16. UNICODE String Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	STRING Descriptor Type
2	*bString*	N	Number	UNICODE encoded string

Table 9-15 是 0 号字符串描述符的格式，这个 Table 9-16 是其它字符串描述符的格式，很明显可以看到，它的前两个字节分别表示了长度和类型，如果读 2 个字节成功的话，就可以准确的获得这个字符串描述符的长度，然后可以再拿这个准确的长度去请求一次。

该尝试的都尝试了，现在看看 716 行，分析一下前面调用 usb_get_string()的结果，如果几次尝试之后，它的返回值还是小于 2，那就返回一个错误码。如果你的辛苦没有白费，rc 大于等于 2，说明终于获得了一个有效的字符串描述符。

717 行，buf 的前两个字节有一个为空时，也就是 Table 9-16 的前两个字节有一个为空时，调用了 message.c 里定义的 usb_try_string_workarounds 函数

```
682 static void usb_try_string_workarounds(unsigned char *buf, int *length)
683 {
684         int newlength, oldlength = *length;
685
686         for (newlength = 2; newlength + 1 < oldlength; newlength += 2)
687                 if (!isprint(buf[newlength]) || buf[newlength + 1])
688                         break;
689
690         if (newlength > 2) {
691                 buf[0] = newlength;
692                 *length = newlength;
693         }
694 }
```

这个函数的目的是从 usb_get_string()返回的数据里计算出前面有效的那一部分的长度。它的核心就是 686 行的那个 for 循环，不过要搞清楚这个循环，还真不是一件容易的事儿，得有相当的理论功底，你看现在十七大一开，党章卖的多火爆，因为大家都知道了不能整天在那里促膝谈性了，要多学点理论。

不久之前刚说了字符串描述符使用的是 UNICODE 编码，其实 UNICODE 指的是包含了字符集、编码、字型等等很多规范的一整套系统，字符集仅仅描述符系统中存在哪些字符，并进行分类，并不涉及如何用数字编码表示的问题。UNICODE 使用的编码形式主要就是两种 UTF，即 UTF-8 和 UTF-16。使用 usb_get_string()获得的字符串使用的是 UTF-16 编

码规则，而且是 little-endpoint 的，每个字符都需要使用两个字节来表示。你看这个 for 循环里 newlength 每次加 2，就是表示每次处理一个字符的，但是要弄明白怎么处理的，还需要知道这两个字节分别是什么东东，这就不得不提及 ASCII、ISO-8859-1 等几个名词儿。

ASCII 是用来表示英文的一种编码规范，表示的最大字符数为 256，每个字符占 1 个字节，但是英文字符没那么多，一般来说 128 个也就够了（最高位为 0），这就已经完全包括了控制字符、数字、大小写字母，还有其它一些符号。对于法语、西班牙语和德语之类的西欧语言都使用叫做 ISO-8859-1 的东东，它扩展了 ASCII 码的最高位，来表示像 n 上带有一个波浪线（241），和 u 上带有两个点（252）这样的字符。而 Unicode 的低字节，也就是在 0 到 255 上同 ISO-8859-1 完全一样，它接着使用剩余的数字，256 到 65535，扩展到表示其它语言的字符。所以可以说 ISO-8859-1 就是 Unicode 的子集，如果 Unicode 的高字节为 0，则它表示的字符就和 ISO-8859-1 完全一样了。

有上面的理论垫底儿，咱们再看看这个 for 循环，newlength 从 2 开始，是因为前两个字节应该是表示长度和类型的，这里只逐个儿对上面 Table 9-16 里的 bString 中的每个字符做处理。还要知道 usb_get_string()得到的结果是 little-endpoint 的，所以 buf[newlength]和 buf[newlength + 1]分别表示一个字符的低字节和高字节，那么 isprint(buf[newlength]就是用来判断一下这个 Unicode 字符的低字节是不是可以 print 的，如果不是，就没必要再往下循环了，后边儿的字符也不再看了，然后就到了 690 行的 if，将 newlength 赋给 buf[0]，即 bLength。length 指向的是 usb_get_string()返回的原始数据的长度，692 行使用 for 循环计算出的有效长度将它给修改了。isprint 在 include/linux/ctype.h 里定义，你可以去看看，这里就不多说了。

这个 for 循环终止的条件有两个，另外一个就是 buf[newlength + 1]，也就是这个 Unicode 字符的高字节不为 0，这时它不存在对应的 ISO-8859-1 形式，为什么加上这个判断？你接着看。

usb_string_sub()的 721 行，buf[0]表示的就是 bLength 的值，如果它小于 usb_get_string()获得的数据长度，说明这些数据里存在一些垃圾，有一些混进革命队伍里的反动势力，要把他们给揪出来排除掉。要知道这个 rc 是要做为真实有效的描述符长度返回的，所以这个时候需要将 buf[0]赋给 rc。

724 行，每个 Unicode 字符需要使用两个字节来表示，所以 rc 必须为偶数，2 的整数倍，如果为奇数，就得将最后那一个字节给抹掉，也就是将 rc 减 1。咱们可以学习一下这里将一个数字转换为偶数时采用的技巧，(rc & 1)在 rc 为偶数时等于 0，为奇数时等于 1，再使用 rc 减去它，得到的就是一个偶数。

从 716~725 这几行，咱们应该看得出，在成功获得一个字符串描述符时，usb_string_sub()返回的是一个 NULL-terminated 字符串的长度，并没有涉及到结束符。牢记这一点，咱们回到 usb_string 函数的 797 行，先将 size，也就是 buf 的大小减 1，目的就是为结束符

保留 1 个字节的位置。

798 行，tbuf 里保存的是 GET_DESCRIPTOR 请求返回的原始数据，err 是 usb_string_sub()的返回值，一切顺利的话，它表示的就是一个有效的描述符的大小。这里 idx 的初始值为 0，而 u 的初始值为 2，目的是从 tbuf 的第三个字节开始拷贝给 buf，毕竟咱们的目的是获得字符串描述符里的那个 bString，而不是整个描述符的内容。u 每次都是增加 2，这是因为采用的 UTF-16 是用两个字节表示一个字符的，所以循环一次要处理两个字节。

801 行，这个 if-else 组合你可能比较糊涂，要搞清楚，还要蓦然回首看一下前面刚普及过的一些理论。tbuf 里每个 Unicode 字符两个字节，又是 little-endpoint 的，所以 801 行就是判断这个 Unicode 字符的高位字节是不是为 0，如果不为 0，则 ISO-8859-1 里没有这个字符，就设置 buf 里的对应字符为‘?’。如果它的高位字节为 0，就说明这个 Unicode 字符 ISO-8859-1 里也有，就直接将它的低位字节赋给 buf。这么一个 for 循环下来，就将 tbuf 里的 Unicode 字符串转化成了 buf 里的 ISO-8859-1 字符串。

806 行，为 buf 追加一个结束符。咱们这节也就结束了。

接口的驱动

从上节的上节我们已经知道，usb_generic_driver在自己的生命线里，以一己之力将设备的各个接口送给了 linux 的设备模型，让 usb 总线的 match 函数，也就是 usb_device_match()，在自己的那条驱动链表里为它们寻找一个合适的接口驱动程序。现在让咱们轻声的问一句，这些接口驱动都从哪里来？

这就要说到每个玩儿 linux 的人都会知道的那几个著名的命令 insmod，modprobe，rmmod。你 insmod 或 modprobe 驱动的时候，经过一个曲折的过程，会调用到你驱动里的那个 xxx_init 函数，进而去调用 usb_register()将你的驱动提交给设备模型，添加到 usb 总线的驱动链表里。你 rmmod 驱动时候，同样经过一个曲折的过程之后，调用到你驱动里的那个 xxx_cleanup 函数，进而调用 usb_deregister()将你的驱动从 usb 总线的驱动链表里删除掉。现在就看看 include/linux/usb.h 里定义的 usb_register 函数

```
916 static inline int usb_register(struct usb_driver *driver)
917 {
918         return usb_register_driver(driver, THIS_MODULE, KBUILD_MODNAME);
919 }
```

看到这个函数，让人不得不感叹一下，现在什么都要讲究包装，明星们是靠绯闻去包装，这不，何洁大妹子在玩过走光，整容等等之后，还不怎么的过瘾，又玩起了绯闻。内核里也免

不了这个俗，注册个驱动也要包装个两层，不过这个包装不是为了出名，而是方便大伙儿的。

本来在两年以前是没这么多道道儿的，也没有 usb_register_driver 这么一个函数，只有个 usb_register()，它直接就把啥事儿都做了。而且那个时候 struct usb_driver 结构里还有一个有名的 owner 字段，每个在那个岁月里写过 usb 驱动的人都会认得它，并且都会毫无犹豫的将它设置为 THIS_MODULE。但是经过岁月的洗礼，在早先贴出来的 struct usb_driver 结构内容里，你发现 owner 已经无影无踪了。要搞清楚这个历史变迁的来龙去脉，你得知道这个 owner 和 THIS_MODULE 都代表了什么。

从那个时代走过来的人，都应该知道 owner 是一个 struct module *类型的结构体指针，现在告诉你的是每个 struct module 结构体在内核里都代表了一个内核模块，就像十七大里的每个代表都代表了一批人，至于代表了什么人，选他们的人才知道，同样，每个 struct module 结构体代表了什么模块，对它进行初始化的模块才知道。当然，初始化这个结构不是写驱动的人该做的事，是在刚才略过的那个从 insmod 或 modprobe 到你驱动的 xxx_init 函数的曲折过程中做的事。insmod 命令执行后，会调用 kernel/module.c 里的一个系统调用 sys_init_module，它会调用 load_module 函数，将用户空间传入的整个内核模块文件创建成一个内核模块，并返回一个 struct module 结构体，从此，内核中便以这个结构体代表这个内核模块。

再看看 THIS_MODULE 宏是什么意思，它在 include/linux/module.h 里的定义是

```
85 #define THIS_MODULE (&__this_module)
```

是一个 struct module 变量，代表当前模块，与那个著名的 current 有几分相似，可以通过 THIS_MODULE 宏来引用模块的 struct module 结构，比如使用 THIS_MODULE->state 可以获得当前模块的状态。现在你应该明白为啥在那个岁月里，你需要毫不犹豫毫不迟疑的将 struct usb_driver 结构里的 owner 设置为 THIS_MODULE 了吧，这个 owner 指针指向的就是你的模块自己。那现在 owner 咋就说没就没了那？这个说来可就话长了，咱就长话短说吧。不知道那个时候你有没有忘记过初始化 owner，反正是很多人都会忘记，大家都把注意力集中到 probe、disconnect 等等需要动脑子的角色上面了，这个不需要动脑子，只需要花个几秒钟指定一下的 owner 反倒常常被忽视，这个就是容易得到的往往不去珍惜，不容易得到的往往日日思量着去争取。于是在 2006 年的春节前夕，在咱们都无心工作无心学习等着过春节的时候，Greg 坚守一线，去掉了 owner，于是千千万万个写 usb 驱动的人再也不用去时刻谨记初始化 owner 了。咱们是不用设置 owner 了，可 core 里不能不设置，struct usb_driver 结构里不是没有 owner 了么，可它里面嵌的那个 struct device_driver 结构里还有啊，设置了它就可以了。于是 Greg 同时又增加了 usb_register_driver()这么一层，usb_register()可以通过将参数指定为 THIS_MODULE 去调用它，所有的事情都挪到它里面去做。反正 usb_register()也是内联的，并不会增加调用的开销。现在是时机看看 usb_register_driver 函数了

```
719 /**
720  * usb_register_driver - register a USB interface driver
```

```
721  * @new_driver: USB operations for the interface driver
722  * @owner: module owner of this driver.
723  * @mod_name: module name string
724  *
725  * Registers a USB interface driver with the USB core.  The list of
726  * unattached interfaces will be rescanned whenever a new driver is
727  * added, allowing the new driver to attach to any recognized interfaces.
728  * Returns a negative error code on failure and 0 on success.
729  *
730  * NOTE: if you want your driver to use the USB major number, you must call
731  * usb_register_dev() to enable that functionality.  This function no longer
732  * takes care of that.
733  */
734 int usb_register_driver(struct usb_driver *new_driver, struct module *owner,
735                         const char *mod_name)
736 {
737         int retval = 0;
738
739         if (usb_disabled())
740                 return -ENODEV;
741
742         new_driver->drvwrap.for_devices = 0;
743         new_driver->drvwrap.driver.name = (char *) new_driver->name;
744         new_driver->drvwrap.driver.bus = &usb_bus_type;
745         new_driver->drvwrap.driver.probe = usb_probe_interface;
746         new_driver->drvwrap.driver.remove = usb_unbind_interface;
747         new_driver->drvwrap.driver.owner = owner;
748         new_driver->drvwrap.driver.mod_name = mod_name;
749         spin_lock_init(&new_driver->dynids.lock);
750         INIT_LIST_HEAD(&new_driver->dynids.list);
751
752         retval = driver_register(&new_driver->drvwrap.driver);
753
754         if (!retval) {
755                 pr_info("%s: registered new interface driver %s\n",
756                         usbcore_name, new_driver->name);
757                 usbfs_update_special();
758                 usb_create_newid_file(new_driver);
759         } else {
760                 printk(KERN_ERR "%s: error %d registering interface "
761                         "       driver %s\n",
762                         usbcore_name, retval, new_driver->name);
763         }
764
```

```
765         return retval;
766 }
```

这函数和前面见过的usb_register_device_driver长的很相，你如果是从那里一路看过来的话，不用俺说什么，你都会明明白白它的意思。不过，本着与人为善的态度，俺还是要简单提一点，for_devices在 742 行设置成了 0，有了这行，match里的那个is_usb_device_driver把门儿的才不会把它当成设备驱动放过去。然后就是在 752 行将你的驱动提交给设备模型，从而添加到usb总线的驱动链表里，从此之后，接口和接口驱动就可以通过usb总线的match函数传情达意眉来眼去。当然，usb的世界里不存在什么一见钟情，也不相信什么一见钟情，因为你不能一眼看出对方挣多少钱，它们要想对上眼，都得严格的满足对方的条件，我们可能那样无条件爱另一人吗？陌陌生生，他又没生我，我又没生他。

还是那个 match

从上次在几米的向左走向右走遇到 usb 总线的那个 match 函数 usb_device_match()开始到现在，遇到了设备，遇到了设备驱动，遇到了接口，也遇到了接口驱动，期间还多次遇到 usb_device_match()，又多次与它擦肩而过，“我们以前都失散过，十三年以后，还不是再遇见？”

其实每个人都有一条共同之路，与正义和良知初恋，失身于上学，嫁给了钱，被世俗包养。每个设备也都有一条共同之路，与 hub 初恋，失身于 usb_generic_driver，嫁给了接口驱动，被 usb 总线保养。人类从没有真正自由过，少年时我们坐在课室里动弹不得，稍后又步入办公室，无论外头阳光多好，还得超时加班，终于铅华洗尽，遍历人间沧桑，又要为子女忙碌，有几个人可以真正做自己想做的事？设备也没有真正自由过，刚开始时在 Default 状态动弹不得，稍后步入 Address，无论外头风光多好，都得与 usb_generic_driver 长厢厮守，没得选择，终于达到了 Configured，又必须为自己的接口殚精竭虑，以便 usb_device_match()能够为它们找一个好人家。

不管怎么说，在这里我们会再次与 usb_device_match()相遇，看看它怎么在接口和驱动之间搭起那座桥。

```
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 }
```

设备那条路已经走过了，现在走走 552 行接口那条路。558 行，接口驱动的 for_devices 在 usb_register _driver()里被初始化为 0，所以这个把门儿的会痛痛快快的放行，继续往下走，561 行，遇到一对儿似曾相识的宏 to_usb_interface 和 to_usb_driver，之所以说似曾相识,是因为早先已经遇到过一对儿 to_usb_device 和 to_usb_device_driver。这两对儿一对儿用于接口和接口驱动，一对儿用于设备和设备驱动，意思都很直白，还是看看 include/linux/usb.h 里的定义

```
159 #define to_usb_interface(d) container_of(d, struct usb_interface, dev)
857 #define to_usb_driver(d) container_of(d, struct usb_driver, drvwrap.driver)
```

再往下走，就是两个函数 usb_match_id 和 usb_match_dynamic_id，它们都是用来完成实际的匹配工作的，只不过前一个是从驱动的 id_table 里找，看接口是不是被驱动所支持，后一个是从驱动的动态 id 链表 dynids 里找。就像女人分结婚的女人与不结婚的两种，男人分自愿结婚与被迫结婚的两种，驱动的 id 表也分 id_table 和 dynids 两种。显然 564~570 这几行的意思就是将 id_table 放在一个比较高的优先级的位置，从它里面找不

到接口了才再从动态 id 链表里找。

当时讲到 struct usb_driver 结构的时候并没有详细讲它里面表示动态 id 的那个结构体 struct usb_dynids，但是做人要厚道，不能太 CCTV，所以现在补充一下，这个结构的定义在 include/linux/usb.h 里

```
760 struct usb_dynids {
761         spinlock_t lock;
762         struct list_head list;
763 };
```

它只有两个字段，一把锁，一个链表，都是在 usb_register _driver()里面初始化的，这个 list 是驱动动态 id 链表的头儿，它里面的每个节点是用另外一个结构 struct usb_dynid 来表示

```
765 struct usb_dynid {
766         struct list_head node;
767         struct usb_device_id id;
768 };
```

这里面就出现了一个 struct usb_device_id 结构体，也就是设备的 id，每次添加一个动态 id，就会向驱动的动态 id 链表里添加一个 struct usb_dynid 结构体。你现在应该可以想像到 usb_match_id 和 usb_match_dynamic_id 这两个函数除了查找的地方不一样，其它应该是没什么差别的。所以接下来咱们只深入探讨一下 usb_match_id 函数，至于 usb_match_dynamic_id()，如果你实在无聊暂时找不到人生目标也可以去看看。它们都在 driver.c 里定义

```
447 /**
448  * usb_match_id - find first usb_device_id matching device or interface
449  * @interface: the interface of interest
450  * @id: array of usb_device_id structures, terminated by zero entry
451  *
452  * usb_match_id searches an array of usb_device_id's and returns
453  * the first one matching the device or interface, or null.
454  * This is used when binding (or rebinding) a driver to an interface.
455  * Most USB device drivers will use this indirectly, through the usb core,
456  * but some layered driver frameworks use it directly.
457  * These device tables are exported with MODULE_DEVICE_TABLE, through
458  * modutils, to support the driver loading functionality of USB hotplugging.
459  *
460  * What Matches:
461  *
462  * The "match_flags" element in a usb_device_id controls which
463  * members are used.  If the corresponding bit is set, the
464  * value in the device_id must match its corresponding member
```

```
465  * in the device or interface descriptor, or else the device_id
466  * does not match.
467  *
468  * "driver_info" is normally used only by device drivers,
469  * but you can create a wildcard "matches anything" usb_device_id
470  * as a driver's "modules.usbmap" entry if you provide an id with
471  * only a nonzero "driver_info" field.  If you do this, the USB device
472  * driver's probe() routine should use additional intelligence to
473  * decide whether to bind to the specified interface.
474  *
475  * What Makes Good usb_device_id Tables:
476  *
477  * The match algorithm is very simple, so that intelligence in
478  * driver selection must come from smart driver id records.
479  * Unless you have good reasons to use another selection policy,
480  * provide match elements only in related groups, and order match
481  * specifiers from specific to general.  Use the macros provided
482  * for that purpose if you can.
483  *
484  * The most specific match specifiers use device descriptor
485  * data.  These are commonly used with product-specific matches;
486  * the USB_DEVICE macro lets you provide vendor and product IDs,
487  * and you can also match against ranges of product revisions.
488  * These are widely used for devices with application or vendor
489  * specific bDeviceClass values.
490  *
491  * Matches based on device class/subclass/protocol specifications
492  * are slightly more general; use the USB_DEVICE_INFO macro, or
493  * its siblings.  These are used with single-function devices
494  * where bDeviceClass doesn't specify that each interface has
495  * its own class.
496  *
497  * Matches based on interface class/subclass/protocol are the
498  * most general; they let drivers bind to any interface on a
499  * multiple-function device.  Use the USB_INTERFACE_INFO
500  * macro, or its siblings, to match class-per-interface style
501  * devices (as recorded in bInterfaceClass).
502  *
503  * Note that an entry created by USB_INTERFACE_INFO won't match
504  * any interface if the device class is set to Vendor-Specific.
505  * This is deliberate; according to the USB spec the meanings of
506  * the interface class/subclass/protocol for these devices are also
507  * vendor-specific, and hence matching against a standard product
508  * class wouldn't work anyway.  If you really want to use an
```

```
509  * interface-based match for such a device, create a match record
510  * that also specifies the vendor ID.  (Unforunately there isn't a
511  * standard macro for creating records like this.)
512  *
513  * Within those groups, remember that not all combinations are
514  * meaningful.  For example, don't give a product version range
515  * without vendor and product IDs; or specify a protocol without
516  * its associated class and subclass.
517  */
518 const struct usb_device_id *usb_match_id(struct usb_interface *interface,
519                                          const struct usb_device_id *id)
520 {
521         /* proc_connectinfo in devio.c may call us with id == NULL. */
522         if (id == NULL)
523                 return NULL;
524
525         /* It is important to check that id->driver_info is nonzero,
526            since an entry that is all zeroes except for a nonzero
527            id->driver_info is the way to create an entry that
528            indicates that the driver want to examine every
529            device and interface. */
530         for (; id->idVendor || id->bDeviceClass || id->bInterfaceClass ||
531                id->driver_info; id++) {
532                 if (usb_match_one_id(interface, id))
533                         return id;
534         }
535
536         return NULL;
537 }
```

522 行，参数 id 指向的是驱动的那个设备花名册，即 id_table，如果它为空，那肯定就是不可能会匹配成功了，这样的驱动就如 usb 驱动里的修女，起码表面上是对所有的接口不感兴趣的。

530 行，你可能会问为什么这里不详细介绍一下 struct usb_device_id 结构，主要是《我是 U 盘里》已经说得非常之详细和有趣了，俺这里实在没必要耗费时间和口舌去说它，另一方面，它里面的那些元素都相当的暴露和直白，我相信依你的智商一眼就能明白个八九不离十，借那位亚裔的色情女星宫城咪咪竞选美州长时的竞选口号给 struct usb_device_id 结构用用：我始终暴露和诚实。

那么这个 for 循环就是轮询设备花名册里的每个设备，如果符合了条件 id->idVendor || id->bDeviceClass || id->bInterfaceClass || id->driver_info，就调用函数 usb_match_one_id 做深层次的匹配。本来，在动态 id 出现之前这个地方是没有

usb_match_one_id 这么一个函数的，所有的匹配都在这个 for 循环里直接做了，但是动态 id 出现之后，同时出现了前面提到的 usb_match_dynamic_id 函数，要在动态 id 链表里做同样的匹配，这就要避免代码重复，于是就将那些重复的代码提出来，组成了 usb_match_one_id 函数。

for 循环的条件里可能出现的一种情况是，id 的其它字段都为空，只有 driver_info 字段有实实在在的内容，这种情况下匹配是肯定成功的，不信的话等会儿你可以看 usb_match_one_id()，这种驱动对 usb 接口来说是比较随便的那种，不管啥接口都能和她对得上眼，为什么会出现这种情况？咱们已经知道，匹配成功后，接着就会调用驱动自己的 probe 函数，驱动在它里面还会对接口做进一步的检查，如果真出现了这里所说的情况，意思也就是驱动将所有的检查接口，和接口培养感情的步骤都揽在自己的 probe 函数里了，它会在那个时候将 driver_info 的内容取出来，然后想怎么处理就怎么处理，本来么，id 里边儿的 driver_info 就是给驱动保存数据用的。

还是看看 usb_match_one_id()究竟是怎么匹配的吧，定义也在 driver.c 里

```
404 /* returns 0 if no match, 1 if match */
405 int usb_match_one_id(struct usb_interface *interface,
406                      const struct usb_device_id *id)
407 {
408         struct usb_host_interface *intf;
409         struct usb_device *dev;
410
411         /* proc_connectinfo in devio.c may call us with id == NULL. */
412         if (id == NULL)
413                 return 0;
414
415         intf = interface->cur_altsetting;
416         dev = interface_to_usbdev(interface);
417
418         if (!usb_match_device(dev, id))
419                 return 0;
420
421         /* The interface class, subclass, and protocol should never be
422          * checked for a match if the device class is Vendor Specific,
423          * unless the match record specifies the Vendor ID. */
424         if (dev->descriptor.bDeviceClass == USB_CLASS_VENDOR_SPEC &&
425                         !(id->match_flags & USB_DEVICE_ID_MATCH_VENDOR) &&
426                         (id->match_flags & (USB_DEVICE_ID_MATCH_INT_CLASS |
427                                 USB_DEVICE_ID_MATCH_INT_SUBCLASS |
428                                 USB_DEVICE_ID_MATCH_INT_PROTOCOL)))
429                 return 0;
430
431         if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_CLASS) &&
```

```
432             (id->bInterfaceClass != intf->desc.bInterfaceClass))
433                 return 0;
434
435         if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_SUBCLASS) &&
436             (id->bInterfaceSubClass != intf->desc.bInterfaceSubClass))
437                 return 0;
438
439         if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_PROTOCOL) &&
440             (id->bInterfaceProtocol != intf->desc.bInterfaceProtocol))
441                 return 0;
442
443         return 1;
444 }
```

412 行，这个 id 指向的就是驱动 id_table 里的某一项了。

415 行，获得接口采用的设置，设置里可是有接口描述符的，要匹配接口和驱动，接口描述符里的信息是必不可少的。

416 行，从接口的 struct usb_interface 结构体获得 usb 设备的 struct usb_device 结构体，interface_to_usbdev 的定义在 include/linux/usb.h 里

```
160 #define interface_to_usbdev(intf) \
161         container_of(intf->dev.parent, struct usb_device, dev)
```

usb 设备和它里面的接口是怎么关联起来的呢？就是上面的那个 parent，接口的 parent 早在 usb_generic_driver 的 generic_probe 函数向设备模型提交设备里的每个接口的时候就被初始化好了，而且指定为接口所在的那个 usb 设备。这么一回顾，interface_to_usbdev 的意思就很明显了，什么事就怕你去回忆，从这个角度看，金三顺的那句“回忆是没有任何力量的。”和整部电视剧一样，纯粹就是瞎扯淡。

418 行，这里又冒出来个 usb_match_device()，接口和驱动之间的感情还真不是那么好培养的，一层一层的，真是应了加菲猫的那句名言：爱情就象照片，需要大量的暗房时间来培养。不过既然存在就是有来头的，它也不会毫无根据的出现，这里虽说是在接口和接口驱动之间匹配，但是接口的 parent 也是必须要符合条件的，这即合情也合理啊，你好不容易鼓足了勇气向一个走在大街上一见钟情的 mm 表白，你觉得 mm 的第一反应是什么？依照行规，很可能就是：你爸是干吗的？是大款么？是当官的么？你要说不，那就别等第二反应了。所以说接口要想得到驱动的芳心，自己的 parent 符合驱动的条件也是很重要的，usb_match_device()就是专门来匹配接口 parent 的。同样在 driver.c 里定义

```
368 /* returns 0 if no match, 1 if match */
369 int usb_match_device(struct usb_device *dev, const struct usb_device_id *id)
370 {
371         if ((id->match_flags & USB_DEVICE_ID_MATCH_VENDOR) &&
```

```
372             id->idVendor != le16_to_cpu(dev->descriptor.idVendor))
373                 return 0;
374
375         if ((id->match_flags & USB_DEVICE_ID_MATCH_PRODUCT) &&
376             id->idProduct != le16_to_cpu(dev->descriptor.idProduct))
377                 return 0;
378
379         /* No need to test id->bcdDevice_lo != 0, since 0 is never
380            greater than any unsigned number. */
381         if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_LO) &&
382             (id->bcdDevice_lo > le16_to_cpu(dev->descriptor.bcdDevice)))
383                 return 0;
384
385         if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_HI) &&
386             (id->bcdDevice_hi < le16_to_cpu(dev->descriptor.bcdDevice)))
387                 return 0;
388
389         if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_CLASS) &&
390             (id->bDeviceClass != dev->descriptor.bDeviceClass))
391                 return 0;
392
393         if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_SUBCLASS) &&
394             (id->bDeviceSubClass!= dev->descriptor.bDeviceSubClass))
395                 return 0;
396
397         if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_PROTOCOL) &&
398             (id->bDeviceProtocol != dev->descriptor.bDeviceProtocol))
399                 return 0;
400
401         return 1;
402 }
```

这个函数采用了排比的修辞手法，美观的同时也增加了可读性。这一个个的 if 条件里都有一部分是将 id 的 match_flags 和一个宏相与,所以弄明白 match_flags 的意思就很关键。罢了罢了，本来说不再浪费口舌在 id 里的那些字段上了，不过为了减少你蓦然回首的次数，这里再说一下这个 match_flags。

驱动的花名册里每个设备都对应了一个 struct usb_device_id 结构体，这个结构体里有很多字段，都是驱动设定好的条条框框，接口必须完全满足里面的条件才能够被驱动所接受，所以说匹配的过程就是检查接口是否满足这些条件的过程。同志们，千万不要认为爱情是骗来的，感情是睡来的，其实都是一个条件一个条件对照出来的，谁让这个世道上男人在某些方面要处于弱势地位那。

当然你可以每次都按照 id 的内容一个一个的比较下去，但是经常来说，一个驱动往往只是想设定其中的某几项，并不要求 struct usb_device_id 结构里的所有那些条件都要满足，萝卜白菜各有所爱么，有的人觉得你有身体就够了，有的人觉得你有钱就够了，有得人觉得你不但得有身体有钱还要有权，如果你运气好，还可能碰到个只要你有时间的。match_flags 就是为了方便各种各样的需求而生的，驱动可以将自己的条件组合起来，match_flags 的每一位对应一个条件，驱动 care 哪个条件了，就将那一位置 1，否则就置 0。当然，内核里对每个驱动可能会 care 的条件都定义成了宏，供驱动去组合，它们都在 include/linux/mod_devicetable.h 里定义

```
122 /* Some useful macros to use to create struct usb_device_id */
123 #define USB_DEVICE_ID_MATCH_VENDOR              0x0001
124 #define USB_DEVICE_ID_MATCH_PRODUCT             0x0002
125 #define USB_DEVICE_ID_MATCH_DEV_LO              0x0004
126 #define USB_DEVICE_ID_MATCH_DEV_HI              0x0008
127 #define USB_DEVICE_ID_MATCH_DEV_CLASS           0x0010
128 #define USB_DEVICE_ID_MATCH_DEV_SUBCLASS        0x0020
129 #define USB_DEVICE_ID_MATCH_DEV_PROTOCOL        0x0040
130 #define USB_DEVICE_ID_MATCH_INT_CLASS           0x0080
131 #define USB_DEVICE_ID_MATCH_INT_SUBCLASS        0x0100
132 #define USB_DEVICE_ID_MATCH_INT_PROTOCOL        0x0200
```

你用自己的火眼金睛很容易的就能看出来这些数字分别表示了一个 u16 整数，也就是 match_flags 中的某一位。驱动比较在意哪个方面，就可以将 match_flags 里对应的位置 1，在和接口匹配的时候自动就会去比较驱动设置的那个条件是否满足。那整个 usb_match_device()函数就没什么说的了，就是从 match_flags 那里得到驱动都在意哪些条件，然后将设备保存在自己描述符里的自身信息与 id 里的相应条件进行比较，有一项比较不成功就说明匹配失败，如果一项符合了就接着看下一项，接口 parent 都满足条件了，就返回 1，表示匹配成功了。

还是回到 usb_match_one_id()继续往下看，假设你运气还不错，parent 满足了驱动的所有条件，得以走到了 424 行。这行还要对接口的 parent 做点最后的确认，某办法，不都说自己有本事不如 parent 有本事么，驱动谨慎一点也可以理解。那么这行的意思就是，如果接口的 parent，usb 设备是属于厂商定义的 class，也就是不属于 storage 等等标准的 class，就不再检查接口的 class，subclass 和 protocol 了，除非 match_flags 里指定了条件 USB_DEVICE_ID_MATCH_VENDOR。431 行之后的三个 if 也不用多说，前面是检查接口 parent 的，这里就是检查接口本身是不是满足驱动的条件的。

当上面各个函数进行的所有检查都完全匹配时，usb 总线的 match 函数 usb_device_match 就会返回 1 表示匹配成功，之后接着就会去调用驱动的 probe 函数做更深入的处理，什么样的处理？这是每个驱动才知道的事情，反正到此为止，core 的任务是已经圆满完成了，咱们的故事也就该结束了。

这个 core 的故事，从 match 开始，到 match 结束，它虽说不会遍及 core 的边边角角所有部分，但应该也有那么十之七八。在 match 的两端是设备和设备的驱动，是接口和接口的驱动，这个故事里遇到的人，遇到的事，早就安排在那里了，由不得我们去选择。在人生的路口上，早已经安排了那些人，那些事，决定你向左走还是向右走。既然如此，那就随便走好了，想那么多干什么呢？