Linux 下的硬件驱动—USB 设备（上）（驱动配置部分）

USB 设备越来越多，而 Linux 在硬件配置上仍然没有做到完全即插即用，对于 Linux 怎样配置和使用他们，也越来越成为困扰我们的一大问题。本文着力从 Linux 系统下设备驱动的架构，去阐述怎样去使用和配置以及怎样编制 USB 设备驱动。对于一般用户，可以使我们明晰 Linux 设备驱动方式，为更好地配置和使用 USB 设备提供了方便；而对于希望开发 Linux 系统下 USB 设备驱动的程序员，提供了初步学习 USB 驱动架构的机会。

前言

USB 是英文"Universal Serial Bus"的缩写，意为"通用串行总线"。是由 Compaq(康柏)、DEC、IBM、Intel、NEC、微软以及 Northern Telecom（北方电讯）等公司于 1994 年 11 月共同提出的，主要目的就是为了解决接口标准太多的弊端。USB 使用一个 4 针插头作为标准插头，并通过这个标准接头，采用菊花瓣形式把所有外设连接起来，它采用串行方式传输数据，目前最大数据传输率为 12Mbps, 支持多数据流和多个设备并行操作，允许外设热插拔。

目前 USB 接口虽然只发展了 2 代（USB1.0/1.1，USB2.0），但是 USB 综合了一个多平台标准的所有优点 -- 包括降低成本，增加兼容性，可连接大量的外部设备，融合先进的功能和品质。使其逐步成为 PC 接口标准，进入了高速发展期。

那么对于使用 Linux 系统，正确支持和配置常见的 USB 设备，就是其使用必不可少的关键一步。

相关技术基础

模块（驱动程序）

模块（module）是在内核空间运行的程序，实际上是一种目标对象文件，没有链接，不能独立运行，但是可以装载到系统中作为内核的一部分运行，从而可以动态扩充内核的功能。模块最主要的用处就是用来实现设备驱动程序。

Linux 下对于一个硬件的驱动，可以有两种方式：直接加载到内核代码中，启动内核时就会驱动此硬件设备。另一种就是以模块方式，编译生成一个.o 文件。当应用程序需要时再加载进内核空间运行。所以我们所说的一个硬件的驱动程序，通常指的就是一个驱动模块。

设备文件

对于一个设备，它可以在/dev 下面存在一个对应的逻辑设备节点，这个节点以文件的形式存在，但它不是普通意义上的文件，它是设备文件，更确切的说，它是设备节点。这个节点是通过 mknod 命令建立的，其中指定了主设备号和次设备号。主设备号表明了某一类设备，一般对应着确定的驱动程序；次设备号一般是区分不同属性，例如不同的使用方法，不同的位置，不同的操作。这个设备号是从/proc/devices 文件中获得的，所以一般是先有驱动程序在内核中，才有设备节点在目录中。这个设备号（特指主设备号）的主要作用，就是声明设备所使用的驱动程序。驱动程序和设备号是一一对应的，当你打开一个设备文件时，操作系统就已经知道这个设备所对应的驱动程序。

SCSI 设备

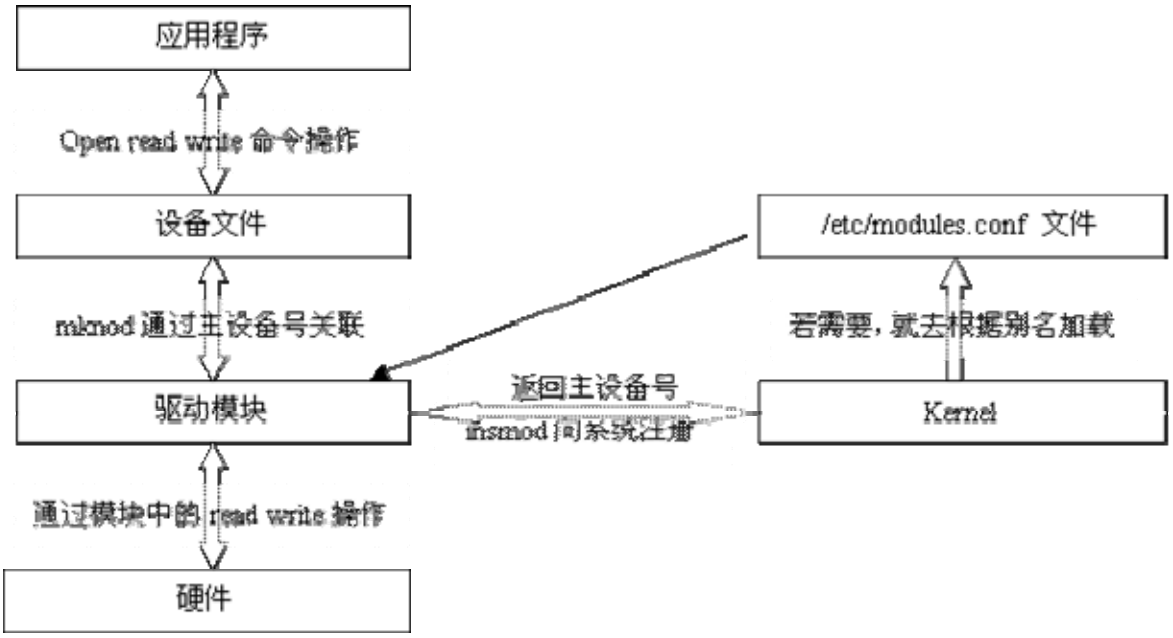
SCSI 是有别于 IDE 的一个计算机标准接口。现在大部分平板式扫描仪、CD-R 刻录机、MO 光磁盘机等渐渐趋向使用 SCSI 接口，加之 SCSI 又能提供一个高速传送通道，所以，接触到 SCSI 设备的用户会越来越多。Linux 支持很多种的 SCSI 设备，例如：SCSI 硬盘、SCSI 光驱、SCSI 磁带机。更重要的是，Linux 提供了 IDE 设备对 SCSI 的模拟（ide-scsi.o 模块），我们通常会就把 IDE 光驱模拟为 SCSI 光驱进行访问。因为在 Linux 中很多软件都只能操作 SCSI 光驱。例如大多数刻录软件、一些媒体播放软件。通常我们的 USB 存储设备，也模拟为 SCSI 硬盘而进行访问。

Linux 硬件驱动架构

对于一个硬件，Linux 是这样来进行驱动的：首先，我们必须提供一个.o 的驱动模块文件（这里我们只说明模块方式，其实内核方式是类似的）。我们要使用这个驱动程序，首先要加载运行它（insmod *.o）。

这样驱动就会根据自己的类型（字符设备类型或块设备类型，例如鼠标就是字符设备而硬盘就是块设备）向系统注册，注册成功系统会反馈一个主设备号，这个主设备号就是系统对它的唯一标识（例如硬盘块设备在/proc/devices 中显示的主设备号为 3 ，我们用 ls -l /dev/had 看到的主设备就肯定是 3)。驱动就是根据此主设备号来创建一个一般放置在/dev 目录下的设备文件（mknod 命令用来创建它，它必须用主设备号这个参数)。在我们要访问此硬件时，就可以对设备文件通过 open、read、write 等命令进行。而驱动就会接收到相应的 read、write 操作而根据自己的模块中的相应函数进行了。

其中还有几个比较有关系的东西：一个是/lib/modules/2.4.XX 目录，它下面就是针对当前内核版本的模块。只要你的模块依赖关系正确（可以通过 depmod 设置)，你就可以通过 modprobe 命令加载而不需要知道具体模块文件位置。 另一个是/etc/modules.conf 文件，它定义了一些常用设备的别名。系统就可以在需要此设备支持时，正确寻找驱动模块。例如 alias eth0 e100，就代表第一块网卡的驱动模块为 e100.o。他们的关系图如下：



配置 USB 设备

内核中配置.

要启用 Linux USB 支持，首先进入"USB support"节并启用"Support for USB"选项（对应模块为 usbcore.o)。尽管这个步骤相当直观明了，但接下来的 Linux USB 设置步骤则会让人感到糊涂。特别地，现在需要选择用于系统的正确 USB 主控制器驱动程序。选项是"EHCI" （对应模块为 ehci-hcd.o)、"UHCI"（对应模块为 usb-uhci.o)、"UHCI (alternate driver)"和"OHCI" （对应模块为 usb-ohci.o)。这是许多**人对 Linux 的 USB 开始感到困惑的地方。**

要理解"EHCI"及其同类是什么，首先要知道每块支持插入 USB 设备的主板或 PCI 卡都需要有 USB 主控制器芯片组。这个特别的芯片组与插入系统的 USB 设备进行相互操作，并负责处理允许 USB 设备与系统其它部分通信所必需的所有低层次细节。

Linux USB 驱动程序有三种不同的 USB 主控制器选项是因为在主板和 PCI 卡上有三种不同类型的 USB 芯片。"EHCI"驱动程序设计成为实现新的高速 USB 2.0 协议的芯片提供支持。"OHCI"驱动程序用来为非 PC 系统上的（以及带有 SiS 和 ALi 芯片组的 PC 主板上的）USB 芯片提供支持。"UHCI"驱动程序用来为大多数其它 PC 主板（包括 Intel 和 Via）上的 USB 实现提供支持。只需选择与希望启用的 USB 支持的类型对应的"?HCI"驱动程序即可。如有疑惑，为保险起见，可以启用"EHCI"、"UHCI" （两者中任选一种，它们之间没有明显的区别）和"OHCI"。(赵明注：根据文档，EHCI 已经包含了 UHCI 和 OHCI，但目前就我个人的测试，单独加 EHCI 是不行的，通常我的做法是根据主板类型加载 UHCI 或 OHCI 后，再加载 EHCI

这样才可以支持 USB2.0 设备)。

启用了"USB support"和适当的"?HCI"USB 主控制器驱动程序后，使 USB 启动并运行只需再进行几个步骤。应该启用"Preliminary USB device filesystem"，然后确保启用所有特定于将与 Linux 一起使用的实际 USB 外围设备的驱动程序。例如，为了启用对 USB 游戏控制器的支持，我启用了"USB Human Interface Device (full HID) support"。我还启用了主"Input core support" 节下的"Input core support"和"Joystick support"。

一旦用新的已启用 USB 的内核重新引导后，若/proc/bus/usb 下没有相应 USB 设备信息，应输入以下命令将 USB 设备文件系统手动挂装到 /proc/bus/usb：

```
# mount -t usbdevfs none /proc/bus/usb
```

为了在系统引导时自动挂装 USB 设备文件系统，请将下面一行添加到 /etc/fstab 中的 /proc 挂装行之后：

```
none /proc/bus/usb usbdevfs defaults 0 0
```

模块的配置方法.

在很多时候，我们的 USB 设备驱动并不包含在内核中。其实我们只要根据它所需要使用的模块，逐一加载。就可以使它启作用。

首先要确保在内核编译时以模块方式选择了相应支持。这样我们就应该可以在/lib/modules/2.4.XX 目录看到相应.o 文件。在加载模块时，我们只需要运行 modprobe xxx.o 就可以了（modprobe 主要加载系统已经通过 depmod 登记过的模块，insmod 一般是针对具体.o 文件进行加载）

对应 USB 设备下面一些模块是关键的。

usbcore.o 要支持 usb 所需要的最基础模块

usb-uhci.o （已经提过）

usb-ohci.o （已经提过）

uhci.o 另一个 uhci 驱动程序，我也不知道有什么用，一般不要加载，会死机的

ehci-hcd.o （已经提过 usb2.0）

hid.o USB 人机界面设备，像鼠标呀、键盘呀都需要

usb-storage.o USB 存储设备，U 盘等用到

相关模块

ide-disk.o IDE 硬盘

ide-scsi.o 把 IDE 设备模拟 SCSI 接口

scsi_mod.o SCSI 支持

注意 kernel config 其中一项：

```
   Probe all LUNs on each SCSI device
```

最好选上，要不某些同时支持多个口的读卡器只能显示一个。若模块方式就要带参数安装或提前在/etc/modules.conf 中加入以下项，来支持多个 LUN。

```
add options scsi_mod max_scsi_luns=9
```

sd_mod.o SCSI 硬盘

sr_mod.o SCSI 光盘

sg.o　　SCSI 通用支持（在某些探测 U 盘、SCSI 探测中会用到）

常见 USB 设备及其配置

在 Linux 2.4 的内核中已经支持不下 20 种设备。它支持几乎所有的通用设备如键盘、鼠标、modem、打印机等，并不断地添加厂商新的设备象数码相机、MP3、网卡等。下面就是几个最常见设备的介绍和使用方法：

USB 鼠标：

键盘和鼠标属于低速的输入设备，对于已经为用户认可的 PS/2 接口，USB 键盘和 USB 鼠标似乎并没有太多更优越的地方。现在的大部分鼠标采用了 PS/2 接口，不过 USB 接口的鼠标也越来越多，两者相比，各有优势：一般来说，USB 的鼠标接口的带宽大于 PS/2 鼠标，也就是说在同样的时间内，USB 鼠标扫描次数就要多于 PS/2 鼠标，这样在定位上 USB 鼠标就更为精确；同时 USB 接口鼠标的默认采样率也比较高，达到 125HZ，而 PS/2 接口的鼠标仅有 40HZ（Windows 9x/Me）或是 60HZ（Windows NT/2000）。

对于 USB 设备你当然必须先插入相应的 USB 控制器模块：usb-uhci.o 或 usb-ohci.o

```
modprobe usb-uhci
```

USB 鼠标为了使其正常工作，您必须先插入模块 usbmouse.o 和 mousedev.o

```
modprobe usbmouse
modprobe mousedev
```

若你把 HID input layer 支持和 input core 支持也作为模块方式安装，那么启动 hid 模块和 input 模块也是必要的。

```
modprobe hid
modprobe input
```

USB 键盘：

一般的，我们现在使用的键盘大多是 PS/2 的，USB 键盘还比较少见，但是下来的发展，键盘将向 USB 接口靠拢。使用 USB 键盘基本上没有太多的要求，只需在主板的 BIOS 设定对 USB 键盘的支持，就可以在各系统中完全无障碍的使用，而且更可以真正做到在即插即用和热插拔使用,并能提供两个 USB 连接埠：让您

可以轻易地直接将具有 USB 接头的装置接在您的键盘上，而非计算机的后面。
同样你当然必须先插入相应的 USB 控制器模块：usb-uhci.o 或 usb-ohci.o

```
modprobe usb-uhci
```

然后您还必须插入键盘模块 usbkbd.o，以及 keybdev.o，这样 usb 键盘才能够正常工作。此时，运行的系统命令：

```
modprobe usbkbd
modprobe keybdev
```

同样若你把 HID input layer 支持和 input core 支持也作为模块方式安装,那么启动 hid 模块和 input 模块也是必要的。

U 盘和 USB 读卡器：

数码存储设备现在对我们来说已经是相当普遍的了。CF 卡、SD 卡、Memory Stick 等存储卡已经遍及我们的身边，通常，他们的读卡器都是 USB 接口的。另外，很多 MP3、数码相机也都是 USB 接口和计算机进行数据传递。更我们的 U 盘、USB 硬盘，作为移动存储设备，已经成为我们的必须装备。
在 Linux 下这些设备通常都是以一种叫做 usb-storage 的方式进行驱动。要使用他们必须加载此模块

```
modprobe usb-storage
```

当然，usbcore.o 和 usb-uhci.o 或 usb-ohci 也肯定是不可缺少的。另外，若你系统中 SCSI 支持也是模块方式，那么下面的模块也要加载

```
modprobe scsi_mod
modprobe sd_mod
```

在加载完这些模块后，我们插入 U 盘或存储卡，就会发现系统中多了一个 SCSI 硬盘，通过正确地 mount 它，就可以使用了（SCSI 硬盘一般为/dev/sd?，可参照文章后面的常见问题解答）。

```
mount /dev/sda1 /mnt
```

Linux 支持的其他 USB 设备。
MODEM--（比较常见）
网络设备

摄像头--（比较常见）例如 ov511.o
联机线--可以让你的两台电脑用 USB 线实现网络功能。usbnet.o
显示器--（我没见过）
游戏杆
电视盒--（比较常见）
手写板--（比较常见）
扫描仪--（比较常见）
刻录机--（比较常见）
打印机--（比较常见）
注意：上面所说的每个驱动模块，并不是都要手动加载，有很多系统会在启动或你的应用需要时自动加载的，写明这些模块，是便于你在不能够使用 USB 设备时，可以自行检查。只要用 lsmod 确保以上模块已经被系统加载，你的设备就应该可以正常工作了。当然注意有些模块已经以内核方式在 kernel 启动时存在了（这些模块文件在/lib/modules/2.4.XX 中是找不到的）。

最常遇见的 USB 问题

有 USB 设备的系统安装完 redhat 7.3 启动死机问题
有USB设备,当你刚装完redhat 7.3第一次启动时,总会死掉。主要原因是Linux在安装时探测到有usb-uhci和 ehci-hcd 两个控制器，但在启动时，加载完 usb-uhci 再加载 ehci-hcd 就会有冲突。分析认为 redhat7.3 系统内核在支持 USB2.0 标准上存在问题。在其他版本的 Linux 中均不存在此问题。
解决办法：在 lilo 或 grub 启动时用命令行传递参数 init=/sbin/init。这样在启动后就不运行其他服务而直接启动 shell。然后运行
mount -o remount,rw / 使/ 可写，init 直接启动的系统默认只 mount /为只读
然后 vi /etc/modules.config 文件
删除 alias usb-controller1 ehci-hcd 一行。或前面加#注释掉
然后 mount -o remount,ro / 使/ 只读，避免直接关机破坏文件系统
然后就可以按 Ctrl-Alt-Delete 直接重启了
或许，你有更简单的办法：换 USB 键盘和鼠标为 PS2 接口，启动后修改/etc/modules.config 文件。
我们已经知道 U 盘在 Linux 中会模拟为 SCSI 设备去访问，可怎么知道它对应那个 SCSI 设备呢？
方法 1：推测。通常你第一次插入一个 SCSI 设备，它就是 sda，第二个就是 sdb 以此类推。你启动 Linux 插入一个 U 盘，就试试 sda，换了一个就可能是 sdb。这里注意两个特例：1） 你用的是联想 U 盘，它可能存在两个设备区（一个用于加密或启动电脑），这样就可能一次用掉两个 sda、sdb，换个 U 盘就是 sdc、sdd。2） 联想数码电脑中，可能已经有了六合一读卡器。它同样也是 USB 存储设备。它会占掉一个或两个 SCSI 设备号。
方法 2：看信息。其实，只要你提前把 usb-storage.o、scsi_mod.o、sd_mod.o 模块加载（直接在 kernel 中也可以）了，在你插入和拔出 U 盘时，系统会自动打出信息如下：

```
SCSI device sda: 60928 512-byte hdwr sectors ( 31 MB )
sda: Write Protect is on
```

根据此信息，你就知道它在 sda 上了。当然，可能你的系统信息级别比较高，上述信息可能没有打出，这时候你只要 tail /var/log/messages 就可以看到了。
方法 3：同样，cat /proc/partitions 也可以看到分区信息，其中 sd?就是 U 盘所对应的了。若根本没有 sd 设备，就要检查你的 SCSI 模块和 usb-storage 模块是否正确加载了。
在使用 U 盘或存储卡时，我该 mount /dev/sda 还是/dev/sda1 呢？
这是一个历史遗留问题。存储卡最初尺寸很小，很多厂商在使用时，就直接使用存储，不含有分区表信息。

而随着存储卡尺寸的不断扩大，它也就引入了类似硬盘分区的概念。例如/dev/hda 你可以分成主分区 hda1、hda2 扩展分区 hda3，然后把扩展分区 hda3 又分为逻辑分区 hda5、hda6、hda7 等。这样，通常的 U 盘就被分成一个分区 sda1，类似把硬盘整个分区分成一个主分区 hda1。实际上，我们完全可以通过 fdisk /dev/sda 对存储卡进行完全类似硬盘的分区方式分成 sda1、sda2 甚至逻辑分区 sda5、sda6。实际上，对 USB 硬盘目前你的确需要这样，因为它通常都是多少 G 的容量。而且通常，它里面就是笔记本硬盘。

一个好玩的问题。你在 Linux 下用 fdisk /dev/sda 对 U 盘进行了多分区，这时候到 windows 下，你会发现怎么找，怎么格式化，U 盘都只能找到第一个分区大小尺寸，而且使用看不出任何问题。这主要是 windows 驱动对 U 盘都只支持一个分区的缘故。你是不是可以利用它来进行一些文件的隐藏和保护？你是不是可以和某些人没玩过 Linux 的人开些玩笑：你的 U 盘容量变小了 J。

现在较多的数码设备也和 windows 一样，是把所有 U 盘容量分为一个，所以在对待 U 盘的时候，通常你 mount 的是 sda1。但对于某些特殊的数码设备格式化的 U 盘或存储卡（目前我发现的是一款联想的支持模拟 USB 软盘的 U 盘和我的一个数码相机），你就要 mount /dev/sda。因为它根本就没分区表（若 mount /dev/sda1 通常的效果是死掉）。其实，这些信息，只要你注意了/proc/partitions 文件，都应该注意到的。

每次插入 U 盘，都要寻找对应设备文件名，都要手动 mount，我能不能做到象 windows 那样插入就可以使用呢。

当然可以，不过你需要做一些工作。我这里只提供一些信息帮助你去尝试完成设置：Linux 内核提供了一种叫 hotplug 支持的东西，它可以让你系统在 PCI 设备、USB 等设备插拔时做一些事情。而 automount 功能可以使你的软驱、光盘等设备的分区自动挂载和自动卸载。你甚至可以在 KDE 桌面中创建相应的图标，方便你操作。具体设置方法就要你自己去尝试了。反正我使用 Linux 已经麻木了，不就是敲一行命令嘛。

Linux 下的硬件驱动——USB 设备（下）（驱动开发部分）

USB 驱动开发

在掌握了 USB 设备的配置后，对于程序员，我们就可以尝试进行一些简单的 USB 驱动的修改和开发了。这一段落，我们会讲解一个最基础 USB 框架的基础上，做两个小的 USB 驱动的例子。

USB 骨架

在 Linux kernel 源码目录中 driver/usb/usb-skeleton.c 为我们提供了一个最基础的 USB 驱动程序。我们称为 USB 骨架。通过它我们仅需要修改极少的部分，就可以完成一个 USB 设备的驱动。我们的 USB 驱动开发也是从她开始的。

那些 linux 下不支持的 USB 设备几乎都是生产厂商特定的产品。如果生产厂商在他们的产品中使用自己定义的协议，他们就需要为此设备创建特定的驱动程序。当然我们知道，有些生产厂商公开他们的 USB 协议，并帮助 Linux 驱动程序的开发，然而有些生产厂商却根本不公开他们的 USB 协议。因为每一个不同的协议都会产生一个新的驱动程序，所以就有了这个通用的 USB 驱动骨架程序， 它是以 pci 骨架为模板的。

如果你准备写一个 linux 驱动程序，首先要熟悉 USB 协议规范。USB 主页上有它的帮助。一些比较典型的驱动可以在上面发现，同时还介绍了 USB urbs 的概念，而这个是 usb 驱动程序中最基本的。

Linux USB 驱动程序需要做的第一件事情就是在 Linux USB 子系统里注册，并提供一些相关信息，例如这个驱动程序支持那种设备，当被支持的设备从系统插入或拔出时，会有哪些动作。所有这些信息都传送到 USB 子系统中，在 usb 骨架驱动程序中是这样来表示的：

```
static struct usb_driver skel_driver = {
     name:        "skeleton",
     probe:       skel_probe,
     disconnect:  skel_disconnect,
```

```
    fops:        &skel_fops,
    minor:       USB_SKEL_MINOR_BASE,
    id_table:    skel_table,
};
```

变量 name 是一个字符串，它对驱动程序进行描述。probe 和 disconnect 是函数指针，当设备与在 id_table 中变量信息匹配时，此函数被调用。
fops 和 minor 变量是可选的。大多 usb 驱动程序钩住另外一个驱动系统，例如 SCSI，网络或者 tty 子系统。这些驱动程序在其他驱动系统中注册，同时任何用户空间的交互操作通过那些接口提供，比如我们把 SCSI 设备驱动作为我们 USB 驱动所钩住的另外一个驱动系统，那么我们此 USB 设备的 read、write 等操作，就相应按 SCSI 设备的 read、write 函数进行访问。但是对于扫描仪等驱动程序来说，并没有一个匹配的驱动系统可以使用，那我们就要自己处理与用户空间的 read、write 等交互函数。Usb 子系统提供一种方法去注册一个次设备号和 file_operations 函数指针，这样就可以与用户空间实现方便地交互。
USB 骨架程序的关键几点如下：
USB 驱动的注册和注销
Usb 驱动程序在注册时会发送一个命令给 usb_register，通常在驱动程序的初始化函数里。
当要从系统卸载驱动程序时，需要注销 usb 子系统。即需要 usb_unregister 函数处理：

```
static void __exit usb_skel_exit(void)
{
   /* deregister this driver with the USB subsystem */
   usb_deregister(&skel_driver);
}
module_exit(usb_skel_exit);
```

当 usb 设备插入时，为了使 linux-hotplug（Linux 中 PCI、USB 等设备热插拔支持）系统自动装载驱动程序，你需要创建一个 MODULE_DEVICE_TABLE。代码如下（这个模块仅支持某一特定设备）：

```
/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID,
      USB_SKEL_PRODUCT_ID) },
    { }                     /* Terminating entry */
};

MODULE_DEVICE_TABLE (usb, skel_table);
```

USB_DEVICE 宏利用厂商 ID 和产品 ID 为我们提供了一个设备的唯一标识。当系统插入一个 ID 匹配的 USB 设备到 USB 总线时，驱动会在 USB core 中注册。驱动程序中 probe 函数也就会被调用。usb_device 结构指针、接口号和接口 ID 都会被传递到函数中。

```
static void * skel_probe(struct usb_device *dev,
unsigned int ifnum, const struct usb_device_id *id)
```

驱动程序需要确认插入的设备是否可以被接受，如果不接受，或者在初始化的过程中发生任何错误，probe 函数返回一个 NULL 值。否则返回一个含有设备驱动程序状态的指针。通过这个指针，就可以访问所有结构中的回调函数。
在骨架驱动程序里，最后一点是我们要注册 devfs。我们创建一个缓冲用来保存那些被发送给 usb 设备的数据和那些从设备上接受的数据，同时 USB urb 被初始化，并且我们在 devfs 子系统中注册设备，允许 devfs 用户访问我们的设备。注册过程如下：

```
/* initialize the devfs node for this device
   and register it */
sprintf(name, "skel%d", skel->minor);
skel->devfs = devfs_register
              (usb_devfs_handle, name,
               DEVFS_FL_DEFAULT, USB_MAJOR,
               USB_SKEL_MINOR_BASE + skel->minor,
               S_IFCHR | S_IRUSR | S_IWUSR |
               S_IRGRP | S_IWGRP | S_IROTH,
               &skel_fops, NULL);
```

如果 devfs_register 函数失败，不用担心，devfs 子系统会将此情况报告给用户。
当然最后，如果设备从 usb 总线拔掉，设备指针会调用 disconnect 函数。驱动程序就需要清除那些被分配了的所有私有数据、关闭 urbs，并且从 devfs 上注销调自己。

```
/* remove our devfs node */
devfs_unregister(skel->devfs);
```

现在，skeleton 驱动就已经和设备绑定上了，任何用户态程序要操作此设备都可以通过 file_operations 结构所定义的函数进行了。首先，我们要 open 此设备。在 open 函数中 MODULE_INC_USE_COUNT 宏是一个关键，它的作用是起到一个计数的作用，有一个用户态程序打开一个设备，计数器就加一，例如，我们以模块方式加入一个驱动，若计数器不为零，就说明仍然有用户程序在使用此驱动，这时候，你就不能通过 rmmod 命令卸载驱动模块了。

```
/* increment our usage count for the module */
MOD_INC_USE_COUNT;
++skel->open_count;
```

```
/* save our object in the file's private structure */
file->private_data = skel;
```

当 open 完设备后，read、write 函数就可以收、发数据了。
skel 的 write、和 read 函数
他们是完成驱动对读写等操作的响应。
在 skel_write 中，一个 FILL_BULK_URB 函数，就完成了 urb 系统 callbak 和我们自己的 skel_write_bulk_callback 之间的联系。注意 skel_write_bulk_callback 是中断方式，所以要注意时间不能太久，本程序中它就只是报告一些 urb 的状态等。
read 函数与 write 函数稍有不同在于：程序并没有用 urb 将数据从设备传送到驱动程序，而是我们用 usb_bulk_msg 函数代替，这个函数能够不需要创建 urbs 和操作 urb 函数的情况下，来发送数据给设备，或者从设备来接收数据。我们调用 usb_bulk_msg 函数并传提一个存储空间，用来缓冲和放置驱动收到的数据，若没有收到数据，就失败并返回一个错误信息。
usb_bulk_msg 函数
当对 usb 设备进行一次读或者写时，usb_bulk_msg 函数是非常有用的；然而，当你需要连续地对设备进行读/写时，建议你建立一个自己的 urbs，同时将 urbs 提交给 usb 子系统。
skel_disconnect 函数
当我们释放设备文件句柄时，这个函数会被调用。MOD_DEC_USE_COUNT 宏会被用到（和 MOD_INC_USE_COUNT 刚好对应，它减少一个计数器），首先确认当前是否有其它的程序正在访问这个设备，如果是最后一个用户在使用，我们可以关闭任何正在发生的写，操作如下：

```
/* decrement our usage count for the device */
--skel->open_count;
if (skel->open_count <= 0) {
   /* shutdown any bulk writes that might be
      going on */
   usb_unlink_urb (skel->write_urb);
   skel->open_count = 0;
}
/* decrement our usage count for the module */
MOD_DEC_USE_COUNT;
```

最困难的是，usb 设备可以在任何时间点从系统中取走，即使程序目前正在访问它。usb 驱动程序必须要能够很好地处理解决此问题，它需要能够切断任何当前的读写，同时通知用户空间程序：usb 设备已经被取走。如果程序有一个打开的设备句柄，在当前结构里，我们只要把它赋值为空，就像它已经消失了。对于每一次设备读写等其它函数操作，我们都要检查 usb_device 结构是否存在。如果不存在，就表明设备已经消失，并返回一个-ENODEV 错误给用户程序。当最终我们调用 release 函数时，在没有文件打开这个设备时，无论 usb_device 结构是否存在、它都会清空 skel_disconnect 函数所作工作。
Usb 骨架驱动程序，提供足够的例子来帮助初始人员在最短的时间里开发一个驱动程序。更多信息你可以到 linux usb 开发新闻组去寻找。
U 盘、USB 读卡器、MP3、数码相机驱动

对于一款 windows 下用的很爽的 U 盘、USB 读卡器、MP3 或数码相机，可能 Linux 下却不能支持。怎么办？其实不用伤心，也许经过一点点的工作，你就可以很方便地使用它了。通常是此 U 盘、USB 读卡器、MP3 或数码相机在 WindowsXP 中不需要厂商专门的驱动就可以识别为移动存储设备，这样的设备才能保证成功，其他的就看你的运气了。

USB 存储设备，他们的 read、write 等操作都是通过上章节中提到的钩子，把自己的操作钩到 SCSI 设备上去的。我们就不需要对其进行具体的数据读写处理了。

第一步：我们通过 cat /proc/bus/usb/devices 得到当前系统探测到的 USB 总线上的设备信息。它包括 Vendor、ProdID、Product 等。下面是我买的一款杂牌 CF 卡读卡器插入后的信息片断：

```
T: Bus=01 Lev=01 Prnt=01 Port=01 Cnt=02 Dev#= 5 Spd=12 MxCh= 0
D: Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=8 #Cfgs= 1
P: Vendor=07c4 ProdID=a400 Rev= 1.13
S: Manufacturer=USB
S: Product=Mass Storage
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=70mA
I: If#= 0 Alt= 0 #EPs= 2 Cls=08(vend.) Sub=06 Prot=50 Driver=usb-storage
E: Ad=81(I) Atr=02(Bulk) MxPS= 64 Ivl= 0ms
E: Ad=02(O) Atr=02(Bulk) MxPS= 64 Ivl= 0ms
```

其中，我们最关心的是 Vendor=07c4 ProdID=a400 和 Manufacturer=USB（果然是杂牌，厂商名都看不到）Product= Mass Storage。

对于这些移动存储设备，我们知道 Linux 下都是通过 usb-storage.o 驱动模拟成 scsi 设备去支持的，之所以不支持，通常是 usb-storage 驱动未包括此厂商识别和产品识别信息（在类似 skel_probe 的 USB 最初探测时被屏蔽了）。对于 USB 存储设备的硬件访问部分，通常是一致的。所以我们要支持它，仅需要修改 usb-storage 中关于厂商识别和产品识别列表部分。

第二部，打开 drivers/usb/storage/unusual_devs.h 文件，我们可以看到所有已知的产品登记表，都是以 UNUSUAL_DEV (idVendor, idProduct, bcdDeviceMin, bcdDeviceMax, vendor_name, product_name, use_protocol, use_transport, init_function, Flags）方式登记的。其中相应的涵义，你就可以根据命名来判断了。所以只要我们如下填入我们自己的注册，就可以让 usb-storage 驱动去认识和发现它。

```
UNUSUAL_DEV(07c4, a400, 0x0000, 0xffff,
" USB ", " Mass Storage ",
US_SC_SCSI, US_PR_BULK, NULL,
US_FL_FIX_INQUIRY | US_FL_START_STOP |US_FL_MODE_XLATE )
```

注意：添加以上几句的位置，一定要正确。比较发现，usb-storage 驱动对所有注册都是按 idVendor, idProduct 数值从小到大排列的。我们也要放在相应位置。

最后，填入以上信息，我们就可以重新编译生成内核或 usb-storage.o 模块。这时候插入我们的设备就可以跟其他 U 盘一样作为 SCSI 设备去访问了。

键盘飞梭支持

目前很多键盘都有飞梭和手写板，下面我们就尝试为一款键盘飞梭加入一个驱动。在通常情况，当我们插入 USB 接口键盘时，在/proc/bus/usb/devices 会看到多个 USB 设备。比如：你的 USB 键盘上的飞梭会是一个，你的手写板会是一个，若是你的 USB 键盘有 USB 扩展连接埠，也会看到。
下面是具体看到的信息

```
T:  Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12  MxCh= 2
B:  Alloc= 11/900 us ( 1%), #Int=  1, #Iso=  0
D:  Ver= 1.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 0.00
S:  Product=USB UHCI Root Hub
S:  SerialNumber=d800
C:* #Ifs= 1 Cfg#= 1 Atr=40 MxPwr=  0mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   8 Ivl=255ms
T:  Bus=02 Lev=01 Prnt=01 Port=01 Cnt=01 Dev#=  3 Spd=12  MxCh= 3
D:  Ver= 1.10 Cls=09(hub  ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=07e4 ProdID=9473 Rev= 0.02
S:  Manufacturer=ALCOR
S:  Product=Movado USB Keyboard
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   1 Ivl=255ms
```

找到相应的信息后就可开始工作了。实际上，飞梭的定义和键盘键码通常是一样的，所以我们参照 drivers/usb/usbkbd..c 代码进行一些改动就可以了。因为没能拿到相应的硬件 USB 协议，我无从知道飞梭在按下时通讯协议众到底发什么，我只能把它的信息打出来进行分析。幸好，它比较简单，在下面代码的 usb_kbd_irq 函数中 if(kbd->new[0] == (char)0x01)和 if(((kbd->new[1]>>4)&0x0f)!=0x7)就是判断飞梭左旋。usb_kbd_irq 函数就是键盘中断响应函数。他的挂接，就是在 usb_kbd_probe 函数中

```
FILL_INT_URB(&kbd->irq, dev, pipe, kbd->new, maxp > 8 ? 8 : maxp,
        usb_kbd_irq, kbd, endpoint->bInterval);
```

一句中实现。
从 usb 骨架中我们知道，usb_kbd_probe 函数就是在 USB 设备被系统发现是运行的。其他部分就都不是关键了。你可以根据具体的探测值（Vendor=07e4 ProdID=9473 等）进行一些修改就可以了。值得一提的是，在键盘中断中，我们的做法是收到 USB 飞梭消息后，把它模拟成左方向键和右方向键，在这里，就看你想怎么去响应它了。当然你也可以响应模拟成 F14、F15 等扩展键码。
在了解了此基本的驱动后，对于一个你已经拿到通讯协议的键盘所带手写板，你就应该能进行相应驱动的开发了吧。
程序见附录 1： 键盘飞梭驱动。
使用此驱动要注意的问题：在加载此驱动时你必须先把 hid 设备卸载，加载完 usbhkey.o 模块后再加载 hid.o。因为若 hid 存在，它的 probe 会屏蔽系统去利用我们的驱动发现我们的设备。其实，飞梭本来就是

一个 hid 设备，正确的方法，或许你应该修改 hid 的 probe 函数，然后把我们的驱动融入其中。

参考资料

《LINUX 设备驱动程序》
ALESSANDRO RUBINI 著
LISOLEG 译
《Linux 系统分析与高级编程技术》
周巍松 编著
Linux Kernel-2.4.20 源码和文档说明

附录 1：键盘飞梭驱动

```
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/input.h>
#include <linux/init.h>
#include <linux/usb.h>
#include <linux/kbd_ll.h>

/*
 * Version Information
 */
#define DRIVER_VERSION ""
#define DRIVER_AUTHOR "TGE HOTKEY "
#define DRIVER_DESC "USB HID Tge hotkey driver"

#define USB_HOTKEY_VENDOR_ID 0x07e4
#define USB_HOTKEY_PRODUCT_ID 0x9473
//厂商和产品 ID 信息就是/proc/bus/usb/devices 中看到的值

MODULE_AUTHOR( DRIVER_AUTHOR );
MODULE_DESCRIPTION( DRIVER_DESC );

struct usb_kbd {
    struct input_dev dev;
    struct usb_device *usbdev;
    unsigned char new[8];
    unsigned char old[8];
    struct urb irq, led;
//  devrequest dr;
//这一行和下一行的区别在于 kernel2.4.20 版本对 usb_kbd 键盘结构定义发生了变化
      struct usb_ctrlrequest dr;
    unsigned char leds, newleds;
    char name[128];
```

```
    int open;
};
//此结构来自内核中 drivers/usb/usbkbd..c

static void usb_kbd_irq(struct urb *urb)
{
    struct usb_kbd *kbd = urb->context;
        int *new;
        new = (int *) kbd->new;

    if(kbd->new[0] == (char)0x01)
    {
        if(((kbd->new[1]>>4)&0x0f)!=0x7)
        {
handle_scancode(0xe0,1);
handle_scancode(0x4b,1);
                handle_scancode(0xe0,0);
                handle_scancode(0x4b,0);
        }
        else
        {
                handle_scancode(0xe0,1);
                handle_scancode(0x4d,1);
                handle_scancode(0xe0,0);
                handle_scancode(0x4d,0);
        }
    }


    printk("new=%x %x %x %x %x %x %x %x",
        kbd->new[0],kbd->new[1],kbd->new[2],kbd->new[3],
        kbd->new[4],kbd->new[5],kbd->new[6],kbd->new[7]);

}

static void *usb_kbd_probe(struct usb_device *dev, unsigned int ifnum,
                           const struct usb_device_id *id)
{
    struct usb_interface *iface;
        struct usb_interface_descriptor *interface;
    struct usb_endpoint_descriptor *endpoint;
        struct usb_kbd *kbd;
        int  pipe, maxp;
```

```
    iface = &dev->actconfig->interface[ifnum];
        interface = &iface->altsetting[iface->act_altsetting];

    if ((dev->descriptor.idVendor != USB_HOTKEY_VENDOR_ID) ||
        (dev->descriptor.idProduct != USB_HOTKEY_PRODUCT_ID) ||
        (ifnum != 1))
    {
        return NULL;
    }
    if (dev->actconfig->bNumInterfaces != 2)
    {
        return NULL;
    }

    if (interface->bNumEndpoints != 1) return NULL;

        endpoint = interface->endpoint + 0;

        pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
        maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));

        usb_set_protocol(dev, interface->bInterfaceNumber, 0);
        usb_set_idle(dev, interface->bInterfaceNumber, 0, 0);

    printk(KERN_INFO "GUO: Vid = %.4x, Pid = %.4x, Device = %.2x, ifnum = %.2x, bufCount = %.8x\\n",
    dev->descriptor.idVendor,dev->descriptor.idProduct,dev->descriptor.bcdDevice,      ifnum,
maxp);

        if (!(kbd = kmalloc(sizeof(struct usb_kbd), GFP_KERNEL))) return NULL;
        memset(kbd, 0, sizeof(struct usb_kbd));

        kbd->usbdev = dev;

        FILL_INT_URB(&kbd->irq, dev, pipe, kbd->new, maxp > 8 ? 8 : maxp,
        usb_kbd_irq, kbd, endpoint->bInterval);

    kbd->irq.dev = kbd->usbdev;

    if (dev->descriptor.iManufacturer)
                usb_string(dev, dev->descriptor.iManufacturer, kbd->name, 63);

    if (usb_submit_urb(&kbd->irq)) {
                kfree(kbd);
                return NULL;
```

```
        }

    printk(KERN_INFO "input%d: %s on usb%d:%d.%d\\n",
                kbd->dev.number, kbd->name, dev->bus->busnum, dev->devnum, ifnum);

        return kbd;
}

static void usb_kbd_disconnect(struct usb_device *dev, void *ptr)
{
    struct usb_kbd *kbd = ptr;
        usb_unlink_urb(&kbd->irq);
        kfree(kbd);

}

static struct usb_device_id usb_kbd_id_table [] = {
    { USB_DEVICE(USB_HOTKEY_VENDOR_ID, USB_HOTKEY_PRODUCT_ID) },
    { }                     /* Terminating entry */
};

MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);

static struct usb_driver usb_kbd_driver = {
    name:       "Hotkey",
    probe:      usb_kbd_probe,
    disconnect: usb_kbd_disconnect,
    id_table:   usb_kbd_id_table,
    NULL,
};

static int __init usb_kbd_init(void)
{
    usb_register(&usb_kbd_driver);
    info(DRIVER_VERSION ":" DRIVER_DESC);
    return 0;
}

static void __exit usb_kbd_exit(void)
{
    usb_deregister(&usb_kbd_driver);
}

module_init(usb_kbd_init);
```

```
module_exit(usb_kbd_exit);
```