第十三章

USB 驱动程序

通用串行总线（USB）是主机和外围设备之间的一种连接。USB最初是为了替代许多不同的低速总线（包括并行、串行和键盘连接）而设计的，它以单一类型的总线连接各种不同类型的设备（注1）。USB的发展已经超越了这些低速的连接方式，它现在可以支持几乎所有可以连接到PC上的设备。最新的USB规范修订增加了理论上高达480 Mbps的高速连接。

从拓扑上来看，一个USB子系统并不是以总线的方式来布置的；它是一棵由几个点对点的连接构建而成的树。这些连接是连接设备和集线器（hub）的四线电缆（地线、电源线和两根信号线），这和以太网双绞线类似。USB主控制器（host controller）负责询问每一个USB设备是否有数据需要发送。因为这种拓扑布局的原因，一个USB设备在没有主控制器要求的情况下是不能发送数据的。这种配置便于搭建一个非常简易的即插即用类型的系统，藉此，设备可以由主机自动地配置。

USB总线在技术层面上是非常简单的，因为它是一个单主方式的实现，在此方式下，主机轮询各种不同的外围设备。尽管存在这种内在的局限性，USB总线有一些吸引人的特性，例如设备具有要求一个固定的数据传输带宽的能力，以可靠地支持视频和音频I/O。USB另一个重要的特性是它只担当设备和主控制器之间通信通道的角色，对它所发送的数据没有任何特殊的内容和结构上的要求（注2）。

USB协议规范定义了一套任何特定类型的设备都可以遵循的标准。如果一个设备遵循该标准，就不需要一个特殊的驱动程序。这些不同的特定类型称为类（class），包括存储

注1： 本章部分内容基于Linux内核USB代码的内核文档，这些文档由内核的USB开发者编写，并且按照GPL条款发布。

注2： 实际上，还是存在一些结构，但通常被降低为满足某几个预定义类之一的通信需求：例如，键盘不需要分配带宽，而某些摄像头需要。

设备、键盘、鼠标、游戏杆、网络设备和调制解调器。对于不符合这些类的其他类型的设备，需要针对特定的设备编写一个特定于供货商的驱动程序。视频设备和USB到串口转换设备是一个很好的例子，对于它们没有已定义的标准，来自不同制造商的每一种不同的设备都需要对应的驱动程序。

这些特性，加上设计上与生俱来的热插拔能力，使得USB 成为一个便利和低成本的机制，它可以连接多个设备到计算机，而不需要关闭系统、打开机箱、拧螺丝钉和插拔电线。

Linux 内核支持两种主要类型的USB 驱动程序：宿主（host）系统上的驱动程序和设备（device）上的驱动程序。从宿主的观点来看（一个普通的USB宿主是一个桌面计算机），宿主系统的USB 驱动程序控制插入其中的USB 设备，而USB 设备的驱动程序控制该设备如何作为一个USB 设备和主机通信。由于术语“ USB 设备驱动程序 ”（USB device drivers）非常易于混淆，USB 开发者创建了术语“ USB 器件驱动程序 ”（USB gadget drivers）来描述控制连接到计算机（不要忘了 Linux 还运行于很多小型嵌入式设备上）的USB 设备的驱动程序。本章将详细介绍运行于桌面计算机上的USB 系统是如何运作的。USB 器件驱动程序此刻还未列入本书的内容范围。

如图13-1所示，USB驱动程序存在于不同的内核子系统（块设备、网络设备、字符设备等等）和USB 硬件控制器之中。USB 核心为USB 驱动程序提供了一个用于访问和控制USB 硬件的接口，而不必考虑系统当前存在的各种不同类型的USB 硬件控制器。

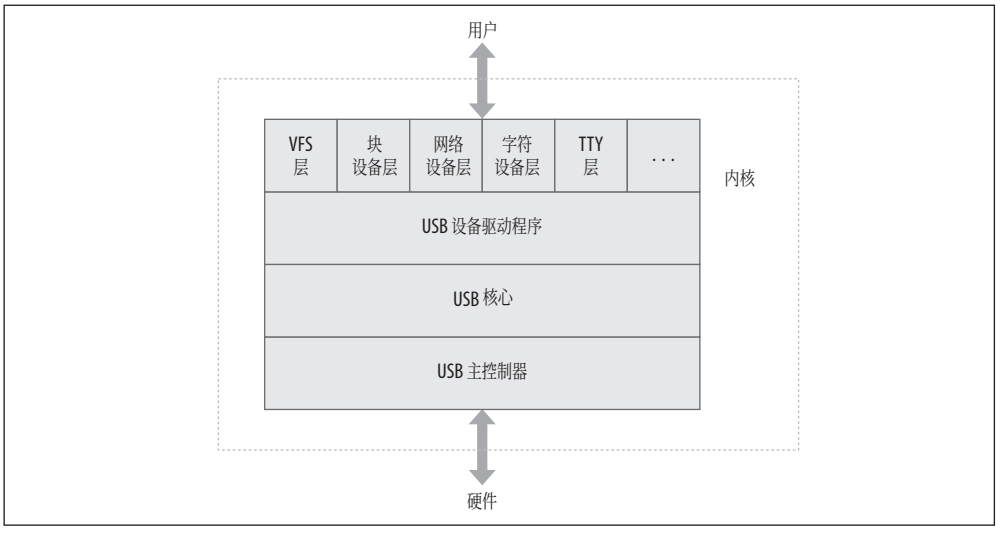

图 13-1：USB 驱动程序概观

USB 设备基础

USB 设备是一个非常复杂的东西，官方 USB 文档（可由 *http://www.usb.org* 获取）中有详细的描述。幸运的是，Linux 内核提供了一个称为 USB 核心（USB core）的子系统来处理大部分的复杂性。本章描述驱动程序和 USB 核心之间的接口。图 13-2 展示了 USB 设备的构成，包括配置、接口和端点，以及 USB 驱动程序如何绑定到 USB 接口上，而不是整个 USB 设备。

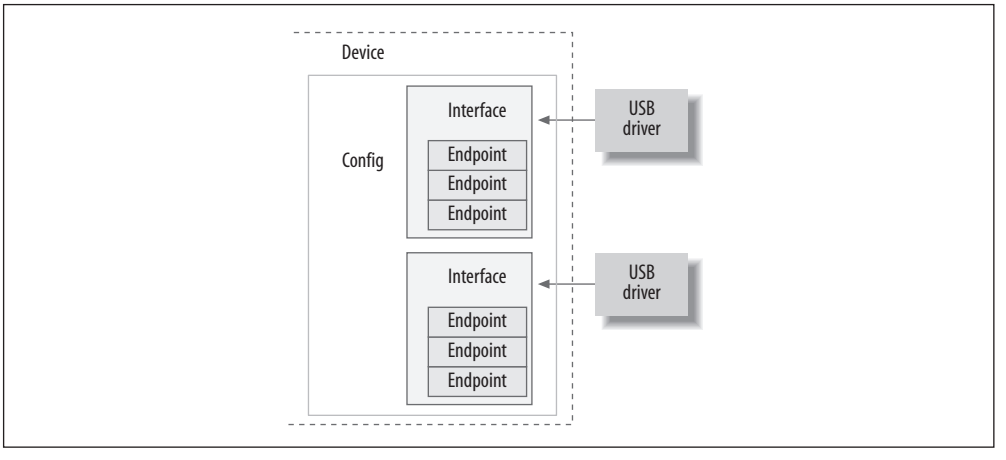

图 13-2：USB 设备概观

端点

USB 通信最基本的形式是通过一个名为端点（endpoint）的东西。USB 端点只能往一个方向传送数据，从主机到设备（称为输出端点）或者从设备到主机（称为输入端点）。端点可以看作是单向的管道。

USB 端点有四种不同的类型，分别具有不同的传送数据的方式：

控制

控制端点用来控制对 USB 设备不同部分的访问。它们通常用于配置设备、获取设备信息、发送命令到设备，或者获取设备的状态报告。这些端点一般体积较小。每个 USB 设备都有一个名为“端点 0”的控制端点，USB 核心使用该端点在插入时进行设备的配置。USB 协议保证这些传输始终有足够的保留带宽以传送数据到设备。

中断

每当 USB 宿主要求设备传输数据时，中断端点就以一个固定的速率来传送少量的数据。这些端点是 USB 键盘和鼠标所使用的主要传输方式。它们通常还用于发送数据到 USB 设备以控制设备，不过一般不用来传输大量的数据。USB 协议保证这些传输始终有足够的保留带宽以传送数据。

批量

批量（bulk）端点传输大批量的数据。这些端点通常比中断端点大得多（它们可以一次持有更多的字符）。它们常见于需要确保没有数据丢失的传输的设备。USB 协议不保证这些传输始终可以在特定的时间内完成。如果总线上的空间不足以发送整个批量包，它将被分割为多个包进行传输。这些端点通常出现在打印机、存储设备和网络设备上。

等时

等时（isochronous）端点同样可以传送大批量的数据，但数据是否到达是没有保证的。这些端点用于可以应付数据丢失情况的设备，这类设备更注重于保持一个恒定的数据流。实时的数据收集（例如音频和视频设备）几乎毫无例外都使用这类端点。

控制和批量端点用于异步的数据传输，只要驱动程序决定使用它们。中断和等时端点是周期性的。也就是说，这些端点被设置为在固定的时段连续地传输数据，基于此，USB 核心为它们保留了相应的带宽。

内核中使用 `struct usb_host_endpoint` 结构体来描述 USB 端点。该结构体在另一个名为 `struct usb_endpoint_descriptor` 的结构体中包含了真正的端点信息。后一个结构体包含了所有的 USB 特定的数据，这些数据的格式是由设备自己定义的。该结构体中驱动程序需要关心的字段有：

`bEndpointAddress`

这是特定端点的 USB 地址。这个 8 位的值中还包含了端点的方向。该字段可以结合位掩码 `USB_DIR_OUT` 和 `USB_DIR_IN` 来使用，以确定该端点的数据是传向设备还是主机。

`bmAttributes`

这是端点的类型。该值可以结合位掩码 `USB_ENDPOINT_XFERTYPE_MASK` 来使用，以确定此端点的类型是 `USB_ENDPOINT_XFER_ISOC`、`USB_ENDPOINT_XFER_BULK` 还是 `USB_ENDPOINT_XFER_INT`。这些宏分别表示等时、批量和中断端点。

`wMaxPacketSize`

这是该端点一次可以处理的最大字节数。注意，驱动程序可以发送数量大于此值的

数据到端点，但是在实际传输到设备的时候，数据将被分割为wMaxPacketSize大小的块。对于高速设备，通过使用高位中一些额外的位，该字段可以用来支持端点的高带宽模式。请参考 USB 规范以了解具体实现的详情。

bInterval
: 如果端点是中断类型，该值是端点的间隔设置 —— 也就是说，端点的中断请求间隔时间。该值以毫秒为单位。

该结构体的字段并没有采用“传统的”Linux 内核命名方案。这是因为这些字段直接对应于USB规范中的字段名字。USB内核程序员认为使用规范指定的名字比使用Linux程序员熟悉的变量命名方式更加重要，因为这样便于规范的阅读。

接口

USB 端点被捆绑为接口。USB 接口只处理一种 USB 逻辑连接，例如鼠标、键盘或者音频流。一些USB 设备具有多个接口，例如 USB 扬声器可以包括两个接口：一个USB 键盘用于按键和一个 USB 音频流。因为一个 USB 接口代表了一个基本功能，而每个 USB 驱动程序控制一个接口，因此，以扬声器为例，Linux 需要两个不同的驱动程序来处理一个硬件设备。

USB接口可以有其他的设置，这些是和接口的参数不同的选择。接口的最初状态是在第一个设置，编号为 0。其他的设置可以用来以不同的方式控制端点，例如为设备保留大小不同的 USB 带宽。每个带有等时端点的设备对同一个接口使用不同的设置。

内核使用 struct usb_interface 结构体来描述 USB 接口。USB 核心把该结构体传递给 USB 驱动程序，之后由 USB 驱动程序来负责控制该结构体。该结构体中的重要字段有：

struct usb_host_interface *altsetting
: 一个接口结构体数组，包含了所有可能用于该接口的可选设置。每个 struct usb_host_interface结构体包含一套由上述struct usb_host_endpoint结构体定义的端点配置。注意，这些接口结构体没有特定的次序。

unsigned num_altsetting
: altsetting 指针所指的可选设置的数量。

struct usb_host_interface *cur_altsetting
: 指向 altsetting 数组内部的指针，表示该接口的当前活动设置。

`int minor`
: 如果捆绑到该接口的USB驱动程序使用USB主设备号，这个变量包含USB核心分配给该接口的次设备号。这仅在一个成功的`usb_register_dev`调用之后才有效（在本章稍后描述）。

`struct usb_interface`结构体中还有其他的字段，不过USB驱动程序不需要考虑它们。

配置

USB接口本身被捆绑为配置。一个USB设备可以有多个配置，而且可以在配置之间切换以改变设备的状态。例如，一些允许下载固件到其上的设备包含多个配置以完成这个工作，而一个时刻只能激活一个配置。Linux对多个配置的USB设备处理得不是很好，不过，幸好这种情况很少发生。

Linux使用`struct usb_host_config`结构体来描述USB配置，使用`struct usb_device`结构体来描述整个USB设备。USB设备驱动程序通常不需要读取或者写入这些结构体中的任何值，因此这里就不详述它们了。想要深入探究的读者可以在内核源代码树的*include/linux/usb.h*文件中找到对它们的描述。

USB设备驱动程序通常需要把一个给定的`struct usb_interface`结构体的数据转换为一个`struct usb_device`结构体，USB核心在很多函数调用中都需要该结构体。`interface_to_usbdev`就是用于该转换功能的函数。

可以期待的是，当前需要`struct usb_device`结构体的所有USB调用将来会变为使用一个`struct usb_interface`参数，而且驱动程序不再需要去做转换的工作。

概言之，USB设备是非常复杂的，它由许多不同的逻辑单元组成。这些逻辑单元之间的关系可以简单地描述如下：

- 设备通常具有一个或者更多的配置
- 配置经常具有一个或者更多的接口
- 接口通常具有一个或者更多的设置
- 接口没有或者具有一个以上的端点

USB 和 Sysfs

由于单个USB物理设备的复杂性，在sysfs中表示该设备也相当复杂。无论是物理USB

设备（用struct usb_device表示）还是单独的USB接口（用struct usb_interface表示），在sysfs中均表示为单独的设备（这是因为这些结构体都包含一个struct device结构体）。以仅包含一个USB接口的简易USB鼠标为例，下面是该设备的sysfs目录树：

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|   |-- bAlternateSetting
|   |-- bInterfaceClass
|   |-- bInterfaceNumber
|   |-- bInterfaceProtocol
|   |-- bInterfaceSubClass
|   |-- bNumEndpoints
|   |-- detach_state
|   |-- iInterface
|   `-- power
|       `-- state
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- detach_state
|-- devnum
|-- idProduct
|-- idVendor
|-- maxchild
|-- power
|   `-- state
|-- speed
`-- version
```

struct usb_device表示为目录树中的：

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
```

而鼠标的USB接口（USB鼠标驱动程序所绑定的接口）位于如下目录：

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0
```

我们将描述内核如何分类USB设备，以帮助理解上面这些长长的设备路径名的含义。

第一个USB设备是一个根集线器（root hub）。这是一个USB控制器，通常包含在一个PCI设备中。之所以这样命名该控制器，是因为它控制着连接到其上的整个USB总线。该控制器是连接PCI总线和USB总线的桥，也是该总线上的第一个USB设备。

所有的根集线器都由USB核心分配了一个独特的编号。在我们的例子中，根集线器名为usb2，因为它是注册到USB 核心的第二个根集线器。单个系统中可以包含的根集线器的编号在任何时候都是没有限制的。

USB总线上的每个设备都以根集线器的编号作为其名字中的第一个号码。该号码随后是一个横杠字符和设备所插入的端口号。因为我们例子中的设备插入到第一个端口，1 被添加到了名字中。因此，主USB 鼠标设备的设备名是2-1。因为该USB 设备包含一个接口，导致了树中的另一个设备被添加到sysfs 路径中。USB 接口的命名方案是设备名直到该接口为止：在我们的例子中，是 2-1 后面加一个冒号和USB 配置的编号，然后是一个句点和接口的编号。因此对于本例而言，设备名是2-1:1.0，因为它是第一个配置，具有接口编号零。

概言之，USB sysfs 设备命名方案为：

根集线器 - 集线器端口号:配置 . 接口

随着设备更深地进入USB 树，和越来越多的USB 集线器的使用，集线器的端口号被添加到跟随着链中前一个集线器端口号的字符串中。对于一个两层的树，其设备名类似于：

根集线器 - 集线器端口号 - 集线器端口号:配置 . 接口

从前面的USB 设备和接口的目录列表可以看到，所有的USB特定信息都可以从sysfs 直接获得（例如，idVendor、idProduct 和 bMaxPower 信息）。这些文件中的一个，即 *bConfigurationValue*，可以被写入以改变当前使用的活动USB配置。当内核不能够确定选择哪一个配置以恰当地操作设备时，这对于具有多个配置的设备很有用。许多USB调制解调器需要向该文件中写入适当的配置值，以便把恰当的USB驱动程序绑定到该设备。

sysfs 并没有展示USB 设备所有的不同部分，它只限于接口级别。设备可能包含的任何可选配置都没有显示，还有和接口相关联的端点的细节。这个信息可以从 *usbfs* 文件系统找到，该文件系统被挂装到系统的*/proc/bus/usb/*目录。*/proc/bus/usb/devices* 文件确实显示了和sysfs 所展示的所有信息相同的信息，还有系统中存在的所有 USB 设备的可选配置和端点信息。*usbfs* 还允许用户空间的程序直接访问 USB 设备，这使得许多内核驱动程序可以迁移到用户空间，从而更加容易维护和调试。USB扫描仪是一个很好的例子，它不再存在于内核中，因为它的功能现在包含在了用户空间的 SANE 库程序中。

USB urb

Linux 内核中的USB 代码通过一个称为 *urb*（USB 请求块）的东西和所有的USB 设备通信。这个请求块使用struct urb结构体来描述，可以从*include/linux/usb.h*文件中找到。

urb被用来以一种异步的方式往/从特定的USB设备上的特定USB端点发送/接收数据。它的使用和文件系统异步I/O代码中的kiocb结构体以及网络代码中的struct skbuff很类似。USB设备驱动程序可能会为单个端点分配许多urb，也可能对许多不同的端点重用单个的urb，这取决于驱动程序的需要。设备中的每个端点都可以处理一个urb队列，所以多个urb可以在队列为空之前发送到同一个端点。一个urb的典型生命周期如下：

- 由USB设备驱动程序创建。
- 分配给一个特定USB设备的特定端点。
- 由USB设备驱动程序递交到USB核心。
- 由USB核心递交到特定设备的特定USB主控制器驱动程序。
- 由USB主控制器驱动程序处理，它从设备进行USB传送。
- 当urb结束之后，USB主控制器驱动程序通知USB设备驱动程序。

urb可以在任何时刻被递交该urb的驱动程序取消掉，或者被USB核心取消，如果该设备已从系统中移除。urb被动态地创建，它包含一个内部引用计数，使得它们可以在最后一个使用者释放它们时自动地销毁。

本章描述的处理urb的过程是很有用的，因为它使得流处理和其他复杂的、重叠的通信成为可能，而这使驱动程序可以获得最高可能的数据传输速度。不过如果只是想要发送单独的数据块或者控制消息，而不关心数据的吞吐率，过程就不必如此繁琐。（请参考“不使用urb的USB传输”一节）。

struct urb

struct urb结构体中USB设备驱动程序需要关心的字段有：

struct usb_device *dev
: urb所发送的目标struct usb_device指针。该变量在urb可以被发送到USB核心之前必须由USB驱动程序初始化。

unsigned int pipe
: urb所要发送的特定目标struct usb_device的端点信息。该变量在urb可以被发送到USB核心之前必须由USB驱动程序初始化。

  驱动程序必须使用下列恰当的函数来设置该结构体的字段，具体取决于传输的方向。注意每个端点只能属于一种类型。

unsigned int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)
把指定 USB 设备的指定端点号设置为一个控制 OUT 端点。

unsigned int usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint)
把指定 USB 设备的指定端点号设置为一个控制 IN 端点。

unsigned int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint)
把指定 USB 设备的指定端点号设置为一个批量 OUT 端点。

unsigned int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint)
把指定 USB 设备的指定端点号设置为一个批量 IN 端点。

unsigned int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint)
把指定 USB 设备的指定端点号设置为一个中断 OUT 端点。

unsigned int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint)
把指定 USB 设备的指定端点号设置为一个中断 IN 端点。

unsigned int usb_sndisocpipe(struct usb_device *dev, unsigned int endpoint)
把指定 USB 设备的指定端点号设置为一个等时 OUT 端点。

unsigned int usb_rcvisocpipe(struct usb_device *dev, unsigned int endpoint)
把指定 USB 设备的指定端点号设置为一个等时 IN 端点。

unsigned int transfer_flags
该变量可以被设置为许多不同的位值，取决于USB驱动程序对urb的具体操作。可用的值包括：

URB_SHORT_NOT_OK
如果被设置，该值说明任何可能发生的对IN端点的简短读取应该被USB核心当作是一个错误。该值只对从USB设备读取的urb有用，对用于写入的urb没意义。

URB_ISO_ASAP
如果该urb是等时的，当驱动程序想要该urb被调度时可以设置这个位，只要

带宽的利用允许它这么做，而且想要在此时设置urb中的start_frame变量。如果一个等时的urb没有设置该位，驱动程序必须指定start_frame的值，如果传输在当时不能启动的话必须能够正确地恢复。详情请参阅下一节的等时urb部分。

URB_NO_TRANSFER_DMA_MAP
当urb包含一个即将传输的DMA缓冲区时应该设置该位。USB核心使用transfer_dma变量所指向的缓冲区，而不是transfer_buffer变量所指向的。

URB_NO_SETUP_DMA_MAP
和URB_NO_TRANSFER_DMA_MAP位类似，该位用于控制带有已设置好的DMA缓冲区的urb。如果它被设置，USB核心使用setup_dma变量所指向的缓冲区，而不是setup_packet变量。

URB_ASYNC_UNLINK
如果被设置，对该urb的*usb_unlink_urb*调用几乎立即返回，该urb的链接在后台被解开。否则，此函数一直等到urb被完全解开链接和结束才返回。使用该位时要小心，因为它可能会造成非常难以调试的同步问题。

URB_NO_FSBR
仅由UHCI USB主控制器驱动程序使用，指示它不要企图使用前端总线回收（Front Side Bus Reclamation）逻辑。该位通常不应该被设置，因为带有UHCI主控制器的机器会导致大量的CPU负荷，而PCI总线忙于等待一个设置了该位的urb。

URB_ZERO_PACKET
如果被设置，一个批量输出urb以发送一个不包含数据的小数据包来结束，这时数据对齐到一个端点数据包边界。一些断线的USB设备（例如许多USB到IR设备）需要该位才能正确地工作。

URB_NO_INTERRUPT
如果被设置，当urb结束时，硬件可能不会产生一个中断。对该位的使用应当小心谨慎，只有把多个urb排队到同一个端点时才使用。USB核心的函数使用该位来进行DMA缓冲区传输。

void *transfer_buffer
指向用于发送数据到设备（OUT urb）或者从设备接收数据（IN urb）的缓冲区的指针。为了使主控制器正确地访问该缓冲区，必须使用kmalloc来创建它，而不是在栈中或者静态内存中。对于控制端点，该缓冲区用于传输数据的中转。

dma_addr_t transfer_dma
用于以DMA方式传输数据到USB设备的缓冲区。

`int transfer_buffer_length`

`transfer_buffer`或者`transfer_dma`变量所指向的缓冲区的大小（因为一个urb只能使用其中一个）。如果该值为0，两个传输缓冲区都没有被USB核心使用。

对于一个OUT端点，如果端点的最大尺寸小于该变量所指定的值，到USB设备的传输将被分解为更小的数据块以便正确地传输数据。这种大数据量的传输以连续的USB帧的方式进行。在一个urb中提交一个大数据块然后让USB主控制器把它分割为更小的块，比以连续的次序发送更小的缓冲区的速度快得多。

`unsigned char *setup_packet`

指向控制urb的设置数据包的指针。它在传输缓冲区中的数据之前被传送。该变量只对控制urb有效。

`dma_addr_t setup_dma`

控制urb用于设置数据包的DMA缓冲区。它在普通传输缓冲区中的数据之前被传送。该变量只对控制urb有效。

`usb_complete_t complete`

指向一个结束处理例程的指针，当urb被完全传输或者发生错误时，USB核心将调用该函数。在该函数内，USB驱动程序可以检查urb，释放它，或者把它重新提交到另一个传输中去。（有关结束处理例程的详情请参阅“结束urb：结束回调处理例程”一节）。

`usb_complete_t`的类型定义为：

```
typedef void (*usb_complete_t)(struct urb *, struct pt_regs *);
```

`void *context`

指向一个可以被USB驱动程序设置的数据块。它可以在结束处理例程中当urb被返回到驱动程序时使用。有关该变量的详情请参阅随后的小节。

`int actual_length`

当urb结束之后，该变量被设置为urb所发送的数据（OUT urb）或者urb所接收的数据（IN urb）的实际长度。对于IN urb，必须使用该变量而不是`transfer_buffer_length`变量，因为所接收的数据可能小于整个缓冲区的尺寸。

`int status`

当urb结束之后，或者正在被USB核心处理时，该变量被设置为urb的当前状态。USB驱动程序可以安全地访问该变量的唯一时刻是在urb结束处理例程中（在“结束urb：结束回调处理例程”一节中描述）。该限制是为了防止当urb正在被USB核心处理时竞态的发生。对于等时urb，该变量的一个成功值（`0`）只表示urb是否已经被解开链接。如果要获取等时urb的详细状态，应该检查`iso_frame_desc`变量。

该变量的有效值包括：

0
: urb 传输成功。

-ENOENT
: urb 被 *usb_kill_urb* 调用终止。

-ECONNRESET
: urb 被 *usb_unlink_urb* 调用解开链接，urb 的 `transfer_flags` 变量被设置为 `URB_ASYNC_UNLINK`。

-EINPROGRESS
: urb 仍然在被 USB 主控制器处理。如果驱动程序中检查到该值，说明存在代码缺陷。

-EPROTO
: urb 发生了下列错误之一：

  - 在传输中发生了 bitstuff 错误。
  - 硬件没有及时接收到响应数据包。

-EILSEQ
: urb 传输中发生了 CRC 校验不匹配。

-EPIPE
: 端点被中止。如果涉及的端点不是控制端点，可以调用 *usb_clear_halt* 函数来清除该错误。

-ECOMM
: 传输时数据的接收速度比把它写到系统内存的速度快。该错误值仅发生在 IN urb 上。

-ENOSR
: 传输时从系统内存获取数据的速度不够快，跟不上所要求的 USB 数据速率。该错误值仅发生在 OUT urb 上。

-EOVERFLOW
: urb 发生了“串扰（babble）”错误。“串扰”错误发生在端点接收了超过端点指定最大数据包尺寸的数据时。

-EREMOTEIO
: 仅发生在 urb 的 `transfer_flags` 变量被设置 `URB_SHORT_NOT_OK` 标志时，表示 urb 没有接收到所要求的全部数据量。

`-ENODEV`
USB 设备已从系统移除。

`-EXDEV`
仅发生在等时 urb 上，表示传输仅部分完成。为了确定所传输的内容，驱动程序必须查看单个帧的状态。

`-EINVAL`
urb 发生了很糟糕的事情。USB 内核文档描述了该值的含义：

> 等时错乱，如果发生这种情况：退出系统然后回家

如果urb结构体中的某一个参数没有被正确地设置或者*usb_submit_urb*调用中的不正确函数参数把 urb 提交到了 USB 核心，也可能发生这个错误。

`-ESHUTDOWN`
USB 主控制器驱动程序发生了严重的错误；设备已经被禁止，或者从系统脱离，而 urb 在设备被移除之后提交。如果当 urb 被提交到设备时设备的配置被改变，也可能发生这个错误。

一般来说，错误值 `-EPROTO`、`-EILSEQ` 和 `-EOVERFLOW` 表示设备、设备的固件或者把设备连接到计算机的电缆发生了硬件故障。

`int start_frame`
设置或者返回初始的帧数量，用于等时传输。

`int interval`
urb 被轮询的时间间隔。仅对中断或者等时 urb 有效。该值的单位随着设备速度的不同而不同。对于低速和满速的设备，单位是帧，相当于毫秒。对于其他设备，单位是微帧（microframe），相当于毫秒的 1/8。对于等时或者中断 urb，在 urb 被发送到 USB 核心之前，USB 驱动程序必须设置该值。

`int number_of_packets`
仅对等时 urb 有效，指定该 urb 所处理的等时传输缓冲区的数量。对于等时 urb，在 urb 被发送到 USB 核心之前，USB 驱动程序必须设置该值。

`int error_count`
由USB核心设置，仅用于等时urb结束之后。它表示报告了任何一种类型错误的等时传输的数量。

`struct usb_iso_packet_descriptor iso_frame_desc[0]`
仅对等时 urb 有效。该变量是一个 `struct usb_iso_packet_descriptor` 结构体数组。该结构体允许单个urb 一次定义许多等时传输。它还用于收集每个单独传输的传输状态。

struct usb_iso_packet_descriptor由下列字段组成：

unsigned int offset
该数据包的数据在传输缓冲区中的偏移量（第一个字节为0）。

unsigned int length
该数据包的传输缓冲区大小。

unsigned int actual_length
该等时数据包接收到传输缓冲区中的数据长度。

unsigned int status
该数据包的单个等时传输的状态。它可以把相同的返回值作为主struct urb结构体的状态变量。

创建和销毁urb

struct urb结构体不能在驱动程序中或者另一个结构体中静态地创建，因为这样会破坏USB核心对urb所使用的引用计数机制。它必须使用*usb_alloc_urb*函数来创建。该函数原型如下：

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

第一个参数，iso_packets，是该urb应该包含的等时数据包的数量。如果不打算创建等时urb，该值应该设置为0。第二个参数，mem_flags，和传递给用于从内核分配内存的*kmalloc*函数（这些标志的详情参见第八章的“标志参数”一节）的标志有相同的类型。如果该函数成功地为urb分配了足够的内存空间，指向该urb的指针将被返回给调用函数。如果返回值为NULL，说明USB核心内发生了错误，驱动程序需要进行适当的清理。

当一个urb被创建之后，在它可以被USB核心使用之前必须被正确地初始化。关于如何初始化不同类型的urb，参见随后的小节。

驱动程序必须调用*usb_free_urb*函数来告诉USB核心驱动程序已经使用完urb。该函数只有一个参数：

```
void usb_free_urb(struct urb *urb);
```

这个参数是指向所需释放的struct urb的指针。在该函数被调用之后，urb结构体就消失了，驱动程序不能再访问它。

中断 urb

*usb_fill_int_urb*是一个辅助函数, 用来正确地初始化即将被发送到USB设备的中断端点的 urb：

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
                      unsigned int pipe, void *transfer_buffer,
                      int buffer_length, usb_complete_t complete,
                      void *context, int interval);
```

该函数包含很多的参数：

`struct urb *urb`
: 指向需初始化的 urb 的指针。

`struct usb_device *dev`
: 该 urb 所发送的目标 USB 设备。

`unsigned int pipe`
: 该 urb 所发送的目标 USB 设备的特定端点。该值是使用前述 *usb_sndintpipe* 或 *usb_rcvintpipe* 函数来创建的。

`void *transfer_buffer`
: 用于保存外发数据或者接收数据的缓冲区的指针。注意它不能是一个静态的缓冲区，必须使用 *kmalloc* 调用来创建。

`int buffer_length`
: `transfer_buffer` 指针所指向的缓冲区的大小。

`usb_complete_t complete`
: 指向当该 urb 结束之后调用的结束处理例程的指针。

`void *context`
: 指向一个小数据块，该块被添加到 urb 结构体中以便进行结束处理例程后面的查找。

`int interval`
: 该 urb 应该被调度的间隔。有关该值的正确单位，请参考前面对 `struct urb` 结构体的描述。

批量 urb

批量urb的初始化和中断urb很相似。所使用的相关函数是*usb_fill_bulk_urb*, 原型如下：

```
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,
                       unsigned int pipe, void *transfer_buffer,
```

```
                        int buffer_length, usb_complete_t complete,
                        void *context);
```

该函数的参数和 *usb_fill_int_urb* 函数完全一样。不过，没有时间间隔参数，因为批量 urb 没有时间间隔值。请注意，无符号整型 pipe 变量必须使用 *usb_sndbulkpipe* 或 *usb_rcvbulkpipe* 函数来初始化。

usb_fill_int_urb 函数不在 urb 中设置 `transfer_flags` 变量，因此，驱动程序必须自己来修改该字段。

控制 urb

控制 urb 的初始化方法和批量 urb 几乎一样，调用 *usb_fill_control_urb* 函数。

```
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev,
                          unsigned int pipe, unsigned char *setup_packet,
                          void *transfer_buffer, int buffer_length,
                          usb_complete_t complete, void *context);
```

函数参数和 *usb_fill_bulk_urb* 函数完全一样，除了一个新的参数，即 `unsigned char *setup_packet`，它指向即将被发送到端点的设置数据包的数据。同样，无符号整型 pipe 变量必须使用 *usb_sndctrlpipe* 或 *usb_rcvictrlpipe* 函数来初始化。

usb_fill_control_urb 函数不设置 urb 中的 `transfer_flags` 变量，因此驱动程序必须自己修改该字段。大部分的驱动程序不使用该函数，因为使用“ 不使用 urb 的 USB 传输 ”一节中描述的同步 API 调用更加简单。

等时 urb

不幸的是，等时 urb 没有和中断、控制和批量 urb 类似的初始化函数。因此它们在被提交到 USB 核心之前，必须在驱动程序中“ 手工地 ”进行初始化。下面是一个关于如何正确地初始化该类型 urb 的例子。它是从主内核源代码树的 *drivers/usb/media* 目录下的 *konicawc.c* 内核驱动程序中拿出来的。

```
urb->dev = dev;
urb->context = uvd;
urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp-1);
urb->interval = 1;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = cam->sts_buf[i];
urb->complete = konicawc_isoc_irq;
urb->number_of_packets = FRAMES_PER_DESC;
urb->transfer_buffer_length = FRAMES_PER_DESC;
for (j=0; j < FRAMES_PER_DESC; j++) {
        urb->iso_frame_desc[j].offset = j;
```

```
            urb->iso_frame_desc[j].length = 1;
    }
```

提交 urb

一旦 urb 被 USB 驱动程序正确地创建和初始化之后，就可以提交到 USB 核心以发送到 USB 设备了。这是通过调用 *usb_submit_urb* 函数来完成的。

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

urb参数是指向即将被发送到设备的urb的指针。mem_flags参数等同于传递给*kmalloc* 调用的同一个参数，用于告诉 USB 核心如何在此时及时地分配内存缓冲区。

当一个urb 被成功地提交到USB 核心之后，在接收函数被调用之前不能访问该urb 结构体中的任何字段。因为*usb_submit_urb*函数可以在任何时刻调用（包括从一个中断上下文中），mem_flags变量的内容必须是正确的。其实只有三个有效的值可以被使用，取决于 *usb_submit_urb* 何时被调用：

GFP_ATOMIC
: 只要下列条件成立就应该使用该值：

  - 调用者是在一个urb 结束处理例程、中断处理例程、底半部、tasklet 或者定时器回调函数中。
  - 调用者正持有一个自旋锁或读写锁。注意如果持有了信号量，该值就不需要了。
  - current->state 不是 TASK_RUNNING。该状态永远是 TASK_RUNNING，除非驱动程序自己改变了当前的状态。

GFP_NOIO
: 如果驱动程序处于块I/O路径中应该使用该值。在所有存储类型的设备的错误处理路径中也应该使用它。

GFP_KERNEL
: 该值应该在前述类别之外的所有情况中使用。

结束 urb：结束回调处理例程

如果调用*usb_submit_urb*成功，把对urb的控制转交给USB核心，该函数返回0；否则，返回负的错误号。如果函数调用成功，当 urb 结束的时候 urb 的结束处理例程（由结束函数指针指定）正好被调用一次。当该函数被调用时，USB 核心结束了对URB 的处理，此刻对它的控制被返回给设备驱动程序。

只有三种结束 urb 和调用结束函数的情形：

- urb 被成功地发送到了设备，设备返回了正确的确认。对于OUT urb而言就是数据被成功地发送，对于INT urb而言就是所请求的数据被成功地接收到。如果确实这样，urb 中的 status 变量被设置为 0。
- 发送数据到设备或者从设备接收数据时发生了某种错误。错误情况由urb结构体中的 status 变量的错误值来指示。
- urb 从USB 核心中被“ 解开链接 ”。当驱动程序通过 *usb_unlink_urb* 或 *usb_kill_urb* 调用告诉 USB 核心取消一个已提交的 urb 时，或者当设备从系统中被移除而一个 urb 已经提交给它时，会发生这种情况。

本章的稍后将给出一个如何在 urb 结束调用内检测各种不同的返回值的示例。

取消 urb

应该调用 *usb_kill_urb* 或 *usb_unlink_urb* 函数来终止一个已经被提交到USB核心的urb。

```
int usb_kill_urb(struct urb *urb);
int usb_unlink_urb(struct urb *urb);
```

这两个函数的 `urb` 参数是指向即将被取消的 urb 的指针。

如果调用 *usb_kill_urb* 函数，该 urb 的生命周期将被终止。通常是当设备从系统中被断开时，在断开回调函数中调用该函数。

对于某些驱动程序而言，应该使用 *usb_unlink_urb* 函数来告诉 USB 核心终止一个 urb。该函数并不等到urb完全被终止之后才返回到调用函数。这对于在中断处理例程中或者持有一个自旋锁时终止一个urb是很有用的，因为等待一个urb完全被终止需要USB核心具有使调用进程睡眠的能力。该函数需要被要求终止的 urb 中的 `URB_ASYNC_UNLINK` 标志值被设置才能正确地工作。

编写 USB 驱动程序

编写一个USB 设备驱动程序的方法和`pci_driver`类似：驱动程序把驱动程序对象注册到 USB 子系统中，稍后再使用制造商和设备标识来判断是否已经安装了硬件。

驱动程序支持哪些设备？

`struct usb_device_id`结构体提供了一列不同类型的该驱动程序支持的USB设备。

USB核心使用该列表来判断对于一个设备该使用哪一个驱动程序，热插拔脚本使用它来确定当一个特定的设备插入到系统时该自动装载哪一个驱动程序。

`struct usb_device_id` 结构体包括下列字段：

`__u16 match_flags`
: 确定设备和结构体中下列字段中的哪一个相匹配。这是一个由 *include/linux/mod_devicetable.h* 文件中指定的不同的 `USB_DEVICE_ID_MATCH_*` 值定义的位字段。通常不直接设置该字段，而是使用稍后介绍的 `USB_DEVICE` 类型的宏来初始化。

`__u16 idVendor`
: 设备的USB制造商ID。该编号是由USB论坛指派给其成员的，不会由其他人指定。

`__u16 idProduct`
: 设备的USB产品ID。所有指派了制造商ID的制造商都可以随意地赋予其产品ID。

`__u16 bcdDevice_lo`
`__u16 bcdDevice_hi`
: 定义了制造商指派的产品的版本号范围的最低和最高值。`bcdDevice_hi` 值包括在内；该值是最高编号的设备的编号。这两个值都以二进制编码的十进制（BCD）格式来表示。这些变量，加上 `idVendor` 和 `idProduct`，用来定义设备的特定版本号。

`__u8 bDeviceClass`
`__u8 bDeviceSubClass`
`__u8 bDeviceProtocol`
: 分别定义设备的类型、子类型和协议。这些编号由USB论坛指派，定义在USB规范中。这些值详细说明了整个设备的行为，包括该设备上的所有接口。

`__u8 bInterfaceClass`
`__u8 bInterfaceSubClass`
`__u8 bInterfaceProtocol`
: 和上述设备特定的值很类似，这些值分别定义类型、子类型和单个接口的协议。这些编号由USB论坛指派，定义在USB规范中。

`kernel_ulong_t driver_info`
: 该值不是用来比较是否匹配的，不过它包含了驱动程序在USB驱动程序的探测回调函数中可以用来区分不同设备的信息。

对于PCI设备，有许多用来初始化该结构体的宏：

`USB_DEVICE(vendor, product)`
创建一个`struct usb_device_id`结构体，仅和指定的制造商和产品ID值相匹配。该宏常用于需要一个特定驱动程序的USB设备。

`USB_DEVICE_VER(vendor, product, lo, hi)`
创建一个`struct usb_device_id`结构体，仅和某版本范围内的指定制造商和产品ID值相匹配。

`USB_DEVICE_INFO(class, subclass, protocol)`
创建一个`struct usb_device_id`结构体，仅和USB设备的指定类型相匹配。

`USB_INTERFACE_INFO(class, subclass, protocol)`
创建一个`struct usb_device_id`结构体，仅和USB接口的指定类型相匹配。

因此，对于一个只控制来自单一制造商的单一USB设备的简单USB设备驱动程序来说，`struct usb_device_id`表将被定义为：

```
/* 该驱动程序支持的设备列表 */
static struct usb_device_id skel_table [  ] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { }                 /* 终止入口项 */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

对于PC驱动程序，`MODULE_DEVICE_TABLE`宏是必需的，以允许用户空间的工具判断出该驱动程序可以控制什么设备。但是对于USB驱动程序来说，字符串usb必须是该宏中的第一个值。

注册USB驱动程序

所有USB驱动程序都必须创建的主要结构体是`struct usb_driver`。该结构体必须由USB驱动程序来填写，包括许多回调函数和变量，它们向USB核心代码描述了USB驱动程序。

`struct module *owner`
指向该驱动程序的模块所有者的指针。USB核心使用它来正确地对该USB驱动程序进行引用计数，使它不会在不合适的时刻被卸载掉。该变量应该被设置为`THIS_MODULE`宏。

`const char *name`
指向驱动程序名字的指针。在内核的所有USB驱动程序中它必须是唯一的，通常被设置为和驱动程序模块名相同的名字。如果该驱动程序运行在内核中，可以在sysfs的*/sys/bus/usb/drivers/*下面找到它。

`const struct usb_device_id *id_table`

指向`struct usb_device_id`表的指针，该表包含了一列该驱动程序可以支持的所有不同类型的USB设备。如果没有设置该变量，USB驱动程序中的探测回调函数不会被调用。如果想要驱动程序对于系统中的每一个USB设备都被调用，创建一个只设置`driver_info`字段的条目：

```
static struct usb_device_id usb_ids[  ] = {
    {.driver_info = 42},
    {  }
};
```

`int (*probe) (struct usb_interface *intf, const struct usb_device_id *id)`

指向USB驱动程序中的探测函数的指针。当USB核心认为它有一个`struct usb_interface`可以由该驱动程序处理时，它将调用该函数（在“探测和断开的细节”一节中描述）。USB核心用来作判断的指向`struct usb_device_id`的指针也被传递给该函数。如果USB驱动程序确认传递给它的`struct usb_interface`，它应该恰当地初始化设备然后返回0。如果驱动程序不确认该设备，或者发生了错误，它应该返回一个负的错误值。

`void (*disconnect) (struct usb_interface *intf)`

指向USB驱动程序中的断开函数的指针。当`struct usb_interface`被从系统中移除或者驱动程序正在从USB核心中卸载时，USB核心将调用该函数（在“探测和断开的细节”一节中描述）。

因此，创建一个有效的`struct usb_driver`结构体只需要初始化五个字段：

```
static struct usb_driver skel_driver = {
    .owner = THIS_MODULE,
    .name = "skeleton",
    .id_table = skel_table,
    .probe = skel_probe,
    .disconnect = skel_disconnect,
};
```

`struct usb_driver`还包含了另外几个回调函数，这些函数不是很常用，对于一个USB驱动程序的正常工作不是必需的：

`int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf)`

指向USB驱动程序中的*ioctl*函数。如果该函数存在，当用户空间的程序对*usbfs*文件系统中的设备文件进行了*ioctl*调用，而和该设备文件相关联的USB设备附着在该USB驱动程序上时，它将被调用。实际上，只有USB集线器驱动程序使用该ioctl，其他的USB驱动程序都没有使用它的真实需要。

`int (*suspend) (struct usb_interface *intf, u32 state)`
指向USB驱动程序中的挂起函数的指针。当设备将被USB核心挂起时调用该函数。

`int (*resume) (struct usb_interface *intf)`
指向USB驱动程序中的恢复函数的指针。当设备将被USB核心恢复时调用该函数。

以 `struct usb_driver` 指针为参数的 *usb_register_driver* 函数调用把 `struct usb_driver`注册到USB 核心。传统上是在USB 驱动程序的模块初始化代码中完成该工作的：

```
static int __init usb_skel_init(void)
{
    int result;

    /* 把该驱动程序注册到 USB 子系统 */
    result = usb_register(&skel_driver);
    if (result)
        err("usb_register failed. Error number %d", result);

    return result;
}
```

当USB 驱动程序将要被卸载时，需要把 `struct usb_driver`从内核中注销。通过调用 *usb_deregister_driver* 来完成该工作。当该调用发生时，当前绑定到该驱动程序上的任何 USB 接口都被断开，断开函数将被调用。

```
static void __exit usb_skel_exit(void)
{
    /* 把该驱动程序从 USB 子系统注销 */
    usb_deregister(&skel_driver);
}
```

探测和断开的细节

上一节描述的 `struct usb_driver`结构体中，驱动程序指定了两个 USB 核心在适当时间调用的函数。当一个设备被安装而USB核心认为该驱动程序应该处理时，探测函数被调用；探测函数应该检查传递给它的设备信息，确定驱动程序是否真的适合该设备。当驱动程序因为某种原因不应控制设备时，断开函数被调用，它可以做一些清理的工作。

探测和断开回调函数都是在USB集线器内核线程的上下文中被调用的，因此在其中睡眠是合法的。然而，建议大部分的工作尽可能地在用户打开设备时完成，从而把USB 探测的时间减到最少。因为 USB 核心在单一线程中处理 USB 设备的添加和删除，任何低速的设备驱动程序都可以减慢 USB 设备的探测时间，从而影响用户的使用。

在探测回调函数中，USB 驱动程序应该初始化任何可能用于控制 USB 设备的局部结构体。它还应该把所需的任何设备相关信息保存到局部结构体中，因为在此时做该工作是比较容易的。例如，USB 驱动程序通常需要探测设备的端点地址和缓冲区大小，因为需要它们才能和设备进行通信。这里是一些示例代码，它们探测批量类型的 IN 和 OUT 端点，把相关的信息保存到一个局部设备结构体中：

```
/* 设置端点信息 */
/* 只使用第一个批量 IN 和批量 OUT 端点 */
iface_desc = interface->cur_altsetting;
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i].desc;

    if (!dev->bulk_in_endpointAddr &&
        (endpoint->bEndpointAddress & USB_DIR_IN) &&
        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
                =  = USB_ENDPOINT_XFER_BULK)) {
        /* 发现一个批量 IN 类型的端点 */
        buffer_size = endpoint->wMaxPacketSize;
        dev->bulk_in_size = buffer_size;
        dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
        dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
        if (!dev->bulk_in_buffer) {
            err("Could not allocate bulk_in_buffer");
            goto error;
        }
    }

    if (!dev->bulk_out_endpointAddr &&
        !(endpoint->bEndpointAddress & USB_DIR_IN) &&
        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
                =  = USB_ENDPOINT_XFER_BULK)) {
        /* 发现一个批量 OUT 类型的端点 */
        dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
    }
}
if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
    err("Could not find both bulk-in and bulk-out endpoints");
    goto error;
}
```

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

```
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i].desc;
```

然后，在我们有了一个端点，而还没有发现批量 IN 类型的端点时，查看该端点的方向是否为 IN。这可以通过检查位掩码 USB_DIR_IN 是否包含在 bEndpointAddress 端点变量中来确定。如果是的话，我们测定该端点类型是否批量，这通过首先以

USB_ENDPOINT_XFERTYPE_MASK位掩码来取bmAttributes变量的值，然后检查它是否和 USB_ENDPOINT_XFER_BULK 值匹配来完成：

```
if (!dev->bulk_in_endpointAddr &&
    (endpoint->bEndpointAddress & USB_DIR_IN) &&
    ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
           =  = USB_ENDPOINT_XFER_BULK)) {
```

如果所有这些检测都通过了，驱动程序就知道它已经发现了正确的端点类型，可以把该端点的相关信息保存到一个局部结构体中，以便稍后使用它来和端点进行通信：

```
/* 发现一个批量 IN 类型的端点 */
buffer_size = endpoint->wMaxPacketSize;
dev->bulk_in_size = buffer_size;
dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
if (!dev->bulk_in_buffer) {
    err("Could not allocate bulk_in_buffer");
    goto error;
}
```

因为 USB 驱动程序需要在设备生命周期的稍后时间获取和该 struct usb_interface 相关联的局部数据结构体，所以可以调用 *usb_set_intfdata* 函数：

```
/* 把数据指针保存到这个接口设备中 */
usb_set_intfdata(interface, dev);
```

该函数接受一个指向任意数据类型的指针，把它保存到struct usb_interface结构体中以方便后面的访问。应该调用 *usb_get_intfdata* 函数来获取数据：

```
struct usb_skel *dev;
struct usb_interface *interface;
int subminor;
int retval = 0;

subminor = iminor(inode);

interface = usb_find_interface(&skel_driver, subminor);
if (!interface) {
    err ("%s - error, can't find device for minor %d",
         __FUNCTION__, subminor);
    retval = -ENODEV;
    goto exit;
}

dev = usb_get_intfdata(interface);
if (!dev) {
    retval = -ENODEV;
    goto exit;
}
```

usb_get_intfdata 通常在USB 驱动程序的打开函数和断开函数中被调用。正是归功于这两个函数，USB驱动程序不需要维护一个静态的指针数组来存储系统中所有当前设备的设备结构体。对设备信息的非直接引用使得任何USB驱动程序都可以支持数量不限的设备。

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

```
/* 现在可以注册设备了，它已准备好了 */
retval = usb_register_dev(interface, &skel_class);
if (retval) {
    /* 某些情况造成我们不能注册该驱动程序 */
    err("Not able to get a minor for this device.");
    usb_set_intfdata(interface, NULL);
    goto error;
}
```

*usb_register_dev*函数需要一个指向struct usb_interface的指针和一个指向struct usb_class_driver结构的指针。这个 struct usb_class_driver用于定义许多不同的参数，在注册一个次设备号时USB 驱动程序需要 USB 核心知道这些参数。该结构体包括如下的变量：

`char *name`

sysfs 用来描述设备的名字。前导路径名，如果存在的话，只用在devfs中，本书不涉及该内容。如果设备的编号需要出现在名字中，名字字符串应该包含字符`%d`。例如，为了创建devfs名字`usb/foo1`和sysfs类型名字`foo1`，名字字符串应该设置为`usb/foo%d`。

`struct file_operations *fops;`

指向 `struct file_operations`的指针，驱动程序定义该结构体，用它来注册为字符设备。有关该结构体的更多情况请参阅第三章。

`mode_t mode;`

为该驱动程序创建的devfs 文件的模式；在其他情况下没有使用。该变量的一个典型设置是`S_IRUSR`和`S_IWUSR`值的组合，只提供了设备文件属主的读和写访问权限。

`int minor_base;`

这是为该驱动程序指派的次设备号范围的开始值。和该驱动程序相关联的所有设备

都是以唯一的、以该值开始的递增的次设备号来创建的。任何时刻只能允许有16个设备和该驱动程序相关联，除非内核的CONFIG_USB_DYNAMIC_MINORS配置选项被打开。如果如此，该变量将被忽略，以先来先办的方式来分配设备的所有次设备号。建议打开了该选项的系统使用类似udev这样的程序来管理系统中的设备节点，因为一个静态的*/dev*树不会工作正常。

当一个USB设备被断开时，和该设备相关联的所有资源都应该被尽可能地清理掉。在此时，如果已经在探测函数中调用了*usb_register_dev*来为该USB设备分配一个次设备号的话，必须调用*usb_deregister_dev*函数来把次设备号交还USB核心。

在断开函数中，从接口获取之前调用*usb_set_intfdata*设置的任何数据也是很重要的。然后设置struct usb_interface结构体中的数据指针为NULL，以防止任何不适当的对该数据的进行的错误访问。

```
static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;

    /* 防止skel_open()和skel_disconnect()竞争 */
    lock_kernel();

    dev = usb_get_intfdata(interface);
    usb_set_intfdata(interface, NULL);

    /* 返回次设备号 */
    usb_deregister_dev(interface, &skel_class);

    unlock_kernel();

    /* 减小使用计数 */
    kref_put(&dev->kref, skel_delete);

    info("USB Skeleton #%d now disconnected", minor);
}
```

注意上述代码片段中的*lock_kernel*调用。它获取了大内核锁，以使断开回调函数在试图获取一个正确的接口数据结构体指针时不会和打开调用遭遇竞态。因为打开函数是在大内核锁被获取的情况下被调用的，如果断开函数也获取了同一个锁，驱动程序中只有一个部分可以访问和设置接口数据指针。

就在USB设备的断开回调函数被调用之前，所有正在传输到设备的urb都被USB核心取消，因此驱动程序不必要对这些urb显式地调用*usb_kill_urb*。在USB设备已经被断开之后，如果驱动程序试图通过调用*usb_submit_urb*来提交一个urb给它，提交将会失败并返回错误值-EPIPE。

提交和控制 urb

当驱动程序有数据要发送到USB 设备时（典型地发生在驱动程序的写函数中），必须分配一个urb 来把数据传输给设备：

```
urb = usb_alloc_urb(0, GFP_KERNEL);
if (!urb) {
    retval = -ENOMEM;
    goto error;
}
```

在urb 被成功地分配之后，还应该创建一个 DMA 缓冲区来以最高效的方式发送数据到设备，传递给驱动程序的数据应该复制到该缓冲区中：

```
buf = usb_buffer_alloc(dev->udev, count, GFP_KERNEL, &urb->transfer_dma);
if (!buf) {
    retval = -ENOMEM;
    goto error;
}
if (copy_from_user(buf, user_buffer, count)) {
    retval = -EFAULT;
    goto error;
}
```

一旦数据从用户空间正确地复制到了局部缓冲区中，urb必须在可以被提交给USB 核心之前被正确地初始化：

```
/* 正确地初始化 urb */
usb_fill_bulk_urb(urb, dev->udev,
          usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
          buf, count, skel_write_bulk_callback, dev);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

现在urb被正确地分配了，数据被正确地复制了，urb 被正确地初始化了，它就可以被提交给USB 核心以传输到设备：

```
/* 把数据从批量端口发出 */
retval = usb_submit_urb(urb, GFP_KERNEL);
if (retval) {
    err("%s - failed submitting write urb, error %d", __FUNCTION__, retval);
    goto error;
}
```

在urb 被成功地传输到USB 设备之后（或者传输中发生了某些事情），urb 回调函数将被USB核心调用。在我们的例子中，我们初始化urb，使之指向*skel_write_bulk_callback*函数，它就是被调用的函数：

```
static void skel_write_bulk_callback(struct urb *urb, struct pt_regs *regs)
{
```

```
        /* sync/async 解链接故障不是错误 */
        if (urb->status &&
            !(urb->status =  = -ENOENT ||
              urb->status =  = -ECONNRESET ||
              urb->status =  = -ESHUTDOWN)) {
            dbg("%s - nonzero write bulk status received: %d",
                __FUNCTION__, urb->status);
        }

        /* 释放已分配的缓冲区 */
        usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                urb->transfer_buffer, urb->transfer_dma);
    }
```

回调函数中做的第一件事情是检查 urb 的状态，以确定该 urb 是否已经成功地结束。错误值，-ENOENT、-ECONNRESET 和 -ESHUTDOWN 不是真的传输错误，只是报告一次成功的传输的相关情况（请参考“ struct urb ”一节中描述的 urb 可能的错误的列表）。之后回调函数释放传输时分配给该 urb 的缓冲区。

当 urb 回调函数正在运行时另一个 urb 被提交到设备是很常见的。这对于发送流式数据到设备很有用。不要忘了 urb 回调函数是运行在中断上下文中的，因此它不应该进行任何内存分配、持有任何信号量或者做任何其他可能导致进程睡眠的事情。当在回调函数内提交一个 urb 时，如果它在提交过程中需要分配新的内存块的话，使用 GFP_ATOMIC 标志来告诉 USB 核心不要睡眠。

不使用 urb 的 USB 传输

有时候 USB 驱动程序只是要发送或者接收一些简单的 USB 数据，而不想把创建一个 struct urb、初始化它、然后等待该 urb 接收函数运行这些麻烦事都走一遍。有两个提供了更简单接口的函数可以使用。

usb_bulk_msg

usb_bulk_msg 创建一个 USB 批量 urb，把它发送到指定的设备，然后在返回调用者之前等待它的结束。它定义为：

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
                 void *data, int len, int *actual_length,
                 int timeout);
```

该函数的参数为：

struct usb_device *usb_dev
: 指向批量消息所发送的目标 USB 设备的指针。

`unsigned int pipe`
该批量消息所发送的目标 USB 设备的特定端点。该值是调用 *usb_sndbulkpipe* 或 *usb_rcvbulkpipe* 来创建的。

`void *data`
如果是一个OUT端点，它是指向即将发送到设备的数据的指针。如果是一个IN端点，它是指向从设备读取的数据应该存放的位置的指针。

`int len`
data 参数所指缓冲区的大小。

`int *actual_length`
指向保存实际传输字节数的位置的指针，至于是传输到设备还是从设备接收取决于端点的方向。

`int timeout`
以jiffies为单位的应该等待的超时时间。如果该值为0，该函数将一直等待消息的结束。

如果函数调用成功，返回值为 0；否则，返回一个负的错误值。该错误值和“ struct urb ”一节中描述的urb错误编号相匹配。如果成功，`actual_length`参数包含从该消息发送或者接收的字节数。

下面是一个使用该函数调用的例子：

```
/* 进行阻塞的批量读以从设备获取数据 */
retval = usb_bulk_msg(dev->udev,
              usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
              dev->bulk_in_buffer,
              min(dev->bulk_in_size, count),
              &count, HZ*10);

/* 如果读成功，复制数据到用户空间 */
if (!retval) {
    if (copy_to_user(buffer, dev->bulk_in_buffer, count))
        retval = -EFAULT;
    else
        retval = count;
}
```

该例子说明了一个从IN端点的简单的批量读。如果读取成功，数据被复制到用户空间。通常在USB 驱动程序的读函数中完成这个工作。

不能在一个中断上下文中或者在持有自旋锁的情况下调用*usb_bulk_msg*函数。同样，该函数不能被任何其他函数取消，因此使用它的时候要小心；确保驱动程序的断开函数了解足够的信息，在允许自身从内存中被卸载之前等待该调用的结束。

usb_control_msg

除了允许驱动程序发送和接收USB控制消息之外，*usb_control_msg*函数的运作和*usb_bulk_msg*函数类似：

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe,
                    __u8 request, __u8 requesttype,
                    __u16 value, __u16 index,
                    void *data, __u16 size, int timeout);
```

该函数的参数和*usb_bulk_msg*很相似，但有几个重要的区别：

`struct usb_device *dev`
: 指向控制消息所发送的目标USB设备的指针。

`unsigned int pipe`
: 该控制消息所发送的目标USB设备的特定端点。该值是调用usb_sndctrlpipe或usb_rcvctrlpipe来创建的。

`__u8 request`
: 控制消息的USB请求值。

`__u8 requesttype`
: 控制消息的USB请求类型值。

`__u16 value`
: 控制消息的USB消息值。

`__u16 index`
: 控制消息的USB消息索引值。

`void *data`
: 如果是一个OUT端点，它是指向即将发送到设备的数据的指针。如果是一个IN端点，它是指向从设备读取的数据应该存放的位置的指针。

`__u16 size`
: data参数所指缓冲区的大小。

`int timeout`
: 以jiffies为单位的应该等待的超时时间。如果该值为0，该函数将一直等待消息的结束。

如果函数调用成功，它返回传输到设备或者从设备读取的字节数；如果不成功，它返回一个负的错误值。

request、requesttype、value 和 index 参数都直接映射到 USB 规范的 USB 控制消息定义处。关于这些参数的有效值和如何使用，请参考 USB 规范的第九章。

和 *usb_bulk_msg* 函数一样，*usb_control_msg* 函数不能在一个中断上下文中或者持有自旋锁的情况下调用。同样，该函数不能被任何其他函数取消，因此使用它的时候要小心；确保驱动程序的断开函数了解足够的信息，在允许自身从内存中被卸载之前等待该调用的结束。

其他 USB 数据函数

USB 核心中的许多辅助函数可以用来从所有 USB 设备中获取标准的信息。这些函数不能在一个中断上下文中或者持有自旋锁的情况下调用。

usb_get_descriptor 函数从指定的设备获取指定的 USB 描述符。该函数定义为：

```
int usb_get_descriptor(struct usb_device *dev, unsigned char type,
                       unsigned char index, void *buf, int size);
```

USB 驱动程序可以使用该函数来从 struct usb_device 结构体中获取任何没有存在于已有 struct usb_device 和 struct usb_interface 结构体中的设备描述符，例如音频描述符或者其他的类型特定信息。该函数的参数为：

struct usb_device *usb_dev
: 指向想要获取描述符的目标 USB 设备的指针。

unsigned char type
: 描述符的类型。该类型在 USB 规范中有描述，可以是下列类型中的一种：

```
USB_DT_DEVICE
USB_DT_CONFIG
USB_DT_STRING
USB_DT_INTERFACE
USB_DT_ENDPOINT
USB_DT_DEVICE_QUALIFIER
USB_DT_OTHER_SPEED_CONFIG
USB_DT_INTERFACE_POWER
USB_DT_OTG
USB_DT_DEBUG
USB_DT_INTERFACE_ASSOCIATION
USB_DT_CS_DEVICE
USB_DT_CS_CONFIG
USB_DT_CS_STRING
USB_DT_CS_INTERFACE
USB_DT_CS_ENDPOINT
```

`unsigned char index`
应该从设备获取的描述符的编号。

`void *buf`
指向复制描述符到其中的缓冲区的指针。

`int size`
`buf` 变量所指内存的大小。

如果该函数调用成功，它返回从设备读取的字节数。否则，它返回一个由该函数调用的底层的 *usb_control_msg* 函数返回的一个负的错误值。

usb_get_descriptor 调用更常用于从 USB 设备获取一个字符串。因为这很常见，所以提供了一个名为 *usb_get_string* 的辅助函数来完成该工作：

```
int usb_get_string(struct usb_device *dev, unsigned short langid,
                   unsigned char index, void *buf, int size);
```

如果成功，该函数返回从设备接收的字符串的字节数。否则，它返回一个由该函数调用的底层的 *usb_control_msg* 函数返回的一个负的错误值。

如果该函数调用成功，它返回一个以 UTF-16LE 格式（Unicode，每个字符 16 位，小端字节序）编码的字符串，保存在 buf 参数所指的缓冲区中。因为这种格式不是很有用，有另一个名为 *usb_string* 的函数返回从 USB 设备读取的已经转换为 ISO 8859-1 格式的字符串。这种字符集是 Unicode 的一个 8 位的子集，是英语和其他西欧语言字符串的最常见格式。因为它是 USB 设备的字符串的典型格式，建议使用 *usb_string* 函数而不是 *usb_get_string* 函数。

快速参考

本节总结本章中介绍的符号：

`#include <linux/usb.h>`
和 USB 相关的所有内容所在的头文件。所有的 USB 设备驱动程序都必须包括该文件。

`struct usb_driver;`
描述 USB 驱动程序的结构体。

`struct usb_device_id;`
描述该驱动程序支持的 USB 设备类型的结构体。

```
int usb_register(struct usb_driver *d);
void usb_deregister(struct usb_driver *d);
```

用于往 USB 核心注册和注销 USB 驱动程序的函数。

```
struct usb_device *interface_to_usbdev(struct usb_interface *intf);
```

从一个 `struct usb_interface *` 获取一个控制的 `struct usb_device *`。

```
struct usb_device;
```

控制整个 USB 设备的结构体。

```
struct usb_interface;
```

主要的 USB 设备结构体，所有的 USB 驱动程序都用它来和 USB 核心进行通信。

```
void usb_set_intfdata(struct usb_interface *intf, void *data);
void *usb_get_intfdata(struct usb_interface *intf);
```

用于设置和获取 `struct usb_interface` 内私有数据指针段的函数。

```
struct usb_class_driver;
```

描述了想要使用 USB 主设备号和用户空间程序进行通信的 USB 驱动程序的结构体。

```
int usb_register_dev(struct usb_interface *intf, struct usb_class_driver
                     *class_driver);
void usb_deregister_dev(struct usb_interface *intf, struct usb_class_driver
                        *class_driver);
```

用于注册和注销特定的 `struct usb_interface *` 结构体的函数，使用一个 `struct usb_class_driver *` 结构体。

```
struct urb;
```

描述一个 USB 数据传输的结构体。

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
void usb_free_urb(struct urb *urb);
```

用于创建和销毁一个 `struct urb *` 的函数。

```
int usb_submit_urb(struct urb *urb, int mem_flags);
int usb_kill_urb(struct urb *urb);
int usb_unlink_urb(struct urb *urb);
```

用于开始和终止一个 USB 数据传输。

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev, unsigned int
  pipe, void *transfer_buffer, int buffer_length, usb_complete_t complete,
  void *context, int interval);
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev, unsigned int
  pipe, void *transfer_buffer, int buffer_length, usb_complete_t complete,
  void *context);
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev, unsigned
  int pipe, unsigned char *setup_packet, void *transfer_buffer, int
  buffer_ length, usb_complete_t complete, void *context);
```

用于在一个 `struct urb` 被提交到USB 核心之前对它进行初始化的函数。

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data,
  int len, int *actual_length, int timeout);
int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
  __u8 requesttype, __u16 value, __u16 index, void *data, __u16 size,
  int timeout);
```

用于在不使用 `struct urb` 的情况下发送或接收 USB 数据的函数。