第 17 章 USB 设备驱动

USB 设备驱动和 PCI 设备驱动是 PC 中最主要的两种设备驱动程序。与 PCI 协议相比，USB 协议更复杂，涉及面较多。本章将介绍 USB 设备驱动开发。首先介绍 USB 协议，使读者对 USB 协议有个整体认识。然后介绍 USB 设备在 WDM 中的开发框架。由于操作系统的 USB 总线驱动程序提供了丰富的功能调用，因此开发 USB 驱动开发变得相对简单，只需要调用 USB 总线驱动接口。

17.1 USB 总线协议

USB 总线协议比 PCI 协议复杂的多，涉及 USB 物理层协议，又涉及 USB 传输层协议等。对于 USB 驱动程序开发者来说，不需要对 USB 协议的每个细节都很清楚。本节概要地介绍 USB 总线协议，并对驱动开发者需要了解的地方进行详细介绍。

17.1.1 USB 设备简介

USB 即通用串行总线（Universal Serial Bus），是一种支持即插即用的新型串行接口。也有人称之为“菊链（daisy－chaining）”，是因为在一条“线缆”上有链接 127 个设备的能力。USB 要比标准串行口快得多，其数据传输率可达每秒 4Mb~12Mb（而老式的串行口最多是每秒 115Kb）。除了具有较高的传输率外，它还能给外围设备提供支持。

需要注意的是，这不是一种新的总线标准，而是计算机系统连接外围设备（如键盘、鼠标、打印机等）的输入/输出接口标准。到现在为止，计算机系统连接外围设备的接口还没有统一的标准，例如，键盘的插口是圆的、连接打印机要用 9 针或 25 针的并行接口、鼠标则要用 9 针或 25 针的串行接口。USB 能把这些不同的接口统一起来，仅用一个 4 针插头作为标准插头，如图 17-1 所示。通过这个标准插头，采用菊花链形式可以把所有的外设连接起来，并且不会损失带宽。USB 正在取代当前 PC 上的串口和并口。

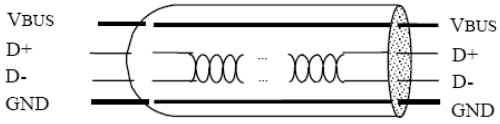

图 17-1　USB 的四条传输线

以 USB 方式连接设备时，所有的外设都在机箱外连接，连接外设不必再打开机箱；允许外设热插拔，而不必关闭主机电源。USB 采用“级联”方式，即每个 USB 设备用一个 USB 插头连接到另一个外设的 USB 插座上，而其本身又提供一个 USB 插座供下一个 USB 外设连接用。通过这种类似菊花链式的连接，一个 USB 控制器可以连接多达 127 个外设，而每个外设间距离（线缆长度）可达 5 米。USB 能智能识别 USB 链上外围设备的插入或拆卸。

它可使多个设备在一个端口上运行，速度也比现在的串行口或并行口快得多，而且其总的连线在理论上说可以无限延长。对 PC 来说，以上这些都是一些难得的优点，因为不再需要 PS/2 端口、MIDI 端口等各种不同的端口了，还可以随时随地在各种设备上任意插拔。可以在一个端口上运行鼠标、控制手柄、键盘以及其他输入装置（例如数码相机），而且，也不必重新启动系统去做这些工作。现在 USB 设备正在快速增多，且由于操作系统已内置支持 USB 的功能，因而用户现在就可以方便地使用。显然，USB 为 PC 的外设扩充提供了一个很好的解决方案。

目前 USB 技术的发展，已经允许用户在不使用网卡、HUB 的情况下，直接通过 USB 技术将几台计算机连接起来组成小型局域网，用户只需要给各台计算机起个名字就可以开始工作。这种网络具备 Ethernet 网络的各种优点，同时少了 Ethernet 网络的许多限制。假设一位用户上班时使用笔记本电脑，回家时使用 PC 机，为实现数据传输，他可以通过采用 USB 技术的接口将两部电脑连接起来交换资源，其数据传输速度可达 12Mbps，这是传统串行口无法比拟的。而且用户在组网的时候根本无须考虑 DIP、IRQ 等问题。此类技术除支持兼容 Ethernet 的软硬件外，也支持标准的网络通信协议，包括 IPX/SPX、NetBEUI 和 TCP/IP，这为通过 USB 技术组成的小局域网连接至大型网络或 Internet 提供了条件。

17.1.2　USB 连接拓扑结构

USB 设备的连接如图 17-2 所示，对于每个 PC 来说，都有一个或者多个称为 Host 控制器的设备，该 Host 控制器和一个根 Hub 作为一个整体。这个根 Hub 下可以接多级的 Hub，每个子 Hub 又可以接子 Hub。每个 USB 作为一个节点接在不同级别的 Hub 上。

（1）USB Host 控制器：每个 PC 的主板上都会有多个 Host 控制器，这个 Host 控制器其实就是一个 PCI 设备，挂载在 PCI 总线上。Host 控制器的驱动由微软公司提供，如图 17-3 所示，这是笔者 PC 中的 Host 控制器及 USB Hub 的驱动。值得注意的是，这里 Host 分别有两种驱动，一种是 1.0，另一种是 2.0，分别对应着 USB 协议 1.0 和 USB 协

议 2.0。

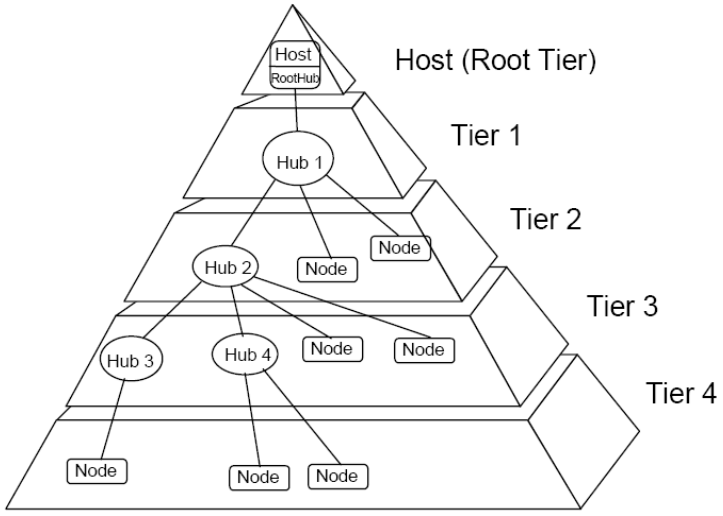

图 17-2　USB 连接拓扑结构

（2）USB Hub：每个 USB Host 控制器都会自带一个 USB Hub，被称为根(Root)Hub。这个根 Hub 可以接子(Sub)Hub，每个 Hub 上挂载 USB 设备。一般 PC 有 8 个 USB 口，通过外接 USB Hub，可以插更多的 USB 设备。当 USB 设备插入到 USB Hub 或从上面拔出时，都会发出电信号通知系统。这样可以枚举 USB 设备，例如当被插入的时候，系统就会创建一个 USB 物理总线，并询问用户安装设备驱动。如图 17-4 所示为一个典型的 USB Hub 的示意图。

图 17-3　USB Host 和 USB Hub 驱动

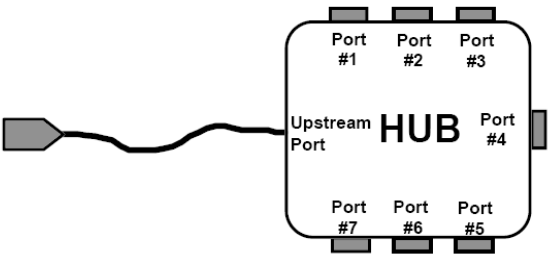

图 17-4　USB Hub 示意图

（3）USB 设备：USB 设备就是插在 USB 总线上工作的设备，广义地讲 USB Hub 也算是 USB 设备。每个根 USB Hub 下可以直接或间接地连接 127 个设备，并且彼此不会干扰。对于用户来说，可以看成是 USB 设备和 USB 控制器直接相连，之间通信需要满足 USB 的通信协议。

有的 USB 设备功能单一，直接挂载在 USB Hub 上。而有的 USB 设备功能复杂，会将

多个 USB 功能连在一起，成为一个复合设备，它甚至可以自己内部带一个 Hub，这个 Hub 下接多个 USB 子设备，其和多个子设备作为一个整体当做一个 USB 设备，如图 17-5 所示。

以上是 USB 的物理拓扑结构，但对于用户来说，可以略去 USB Hub 的概念，或者说 USB Hub 的概念对于用户可以看成是透明的。用户只需要将 USB 设备理解成一个 USB Host 连接多个逻辑设备。可能逻辑设备 1 和逻辑设备 2 是集中在第一个物理设备里，例如有的手机连接计算机后，系统会当做多个 USB 设备加载。因此，作为用户需要用如图 17-6 所示的逻辑拓扑结构理解 USB 拓扑结构。

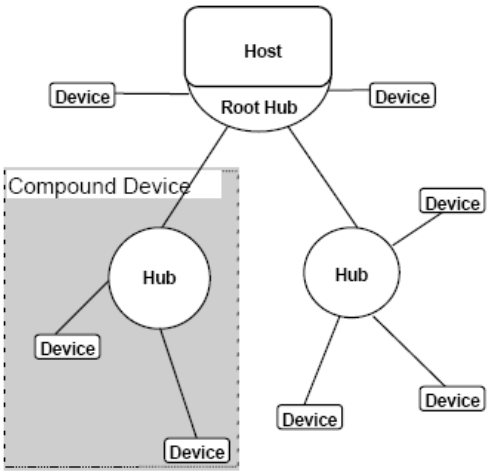

图 17-5　符合设备

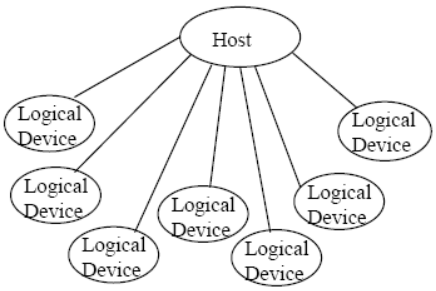

图 17-6　USB 逻辑拓扑结构

但对于具体 USB 设备来说，每个 USB 设备的传输绝对不会影响其他 USB 设备的传输。例如，在有 USB 设备传输的时候，其他 USB 设备的带宽不会被占用。对于 USB 设备来说，每个 USB 设备是直接连接到 USB Host 控制器上的。因此，应该用如图 17-7 所示的视角考虑 USB 设备的通信。

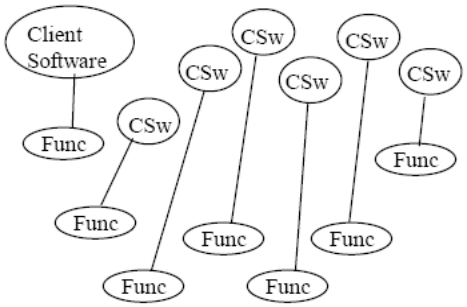

图 17-7　用户对 USB 设备的观察

17.1.3　USB 通信的流程

USB 的连接模式是 Host 和 Devcie 的连接模式，它不同于早期的串口和并口，所有的

请求必须是 Host 向 Device 发出，这就使 Host 端设计相对复杂，而 Device 端设计相对简单。Host 端会在主板的南桥设计好，而 Device 的厂商众多，厂商只需要遵循 USB 协议，重点精力可以放在设备的研发上，而与 PC 的通信不用过多考虑。

在 USB 的通信中，可以看成是一个分层的协议。分为三个层次，即最底层 USB 总线接口层、USB 设备层、功能通信层，如图 17-8 所示。

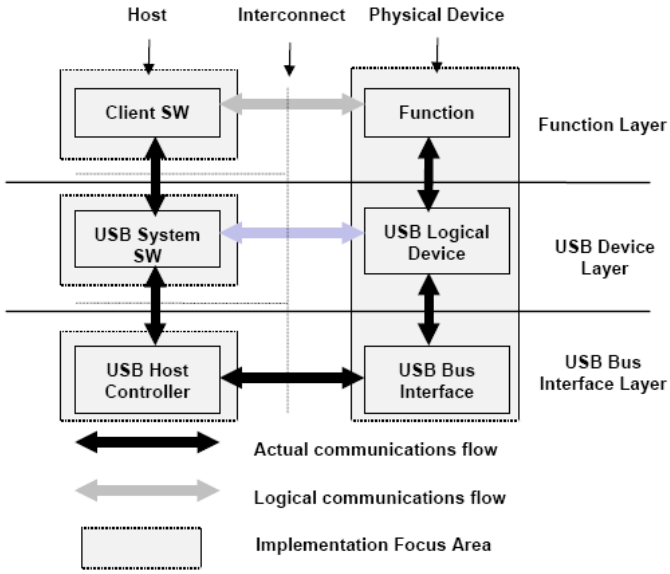

图 17-8　USB 协议

以 USB 摄像头设备为例，视频播放软件想通过 USB 总线得到 USB 摄像头捕捉的视频数据，这就相当于在功能层上。Clinet SW 是视频播放软件，Function 是 USB 摄像头。而这些数据的读取需要 USB 设备层提供的服务，在这一层上，主要是 USB 设备的驱动调度 Host 控制器向 USB 摄像头发出读请求。每个 USB 设备会有多个管道，使用哪个管道，传输的大小都需要指定。这个层次的 USB System SW 就是 USB 摄像头的驱动程序。而在 USB 设备一端一般会有小单片机或者处理芯片负责响应这种读请求，而这一层的传输又依赖于 USB 总线接口层的服务。在这一层，完全是 USB 的物理协议，包括如何分成更小的包（packages）传输，如何保证每次包传输不丢失数据等。

对于 USB 设备驱动程序员，主要是工作在 USB 设备层，向“上”对应用程序提供读写等接口，向“下”将读取某个管道的请求发往 USB Host 控制器驱动程序，它实现了最底层的传输请求。

对于每个 USB 设备，都有一个或者多个的接口（Interface），每个 Interface 都有多个端点（Endpoints），每个端点通过管道（Pipes）和 USB Host 控制器连接。每个 USB 设备都会有一个特殊的端点，即 Endpoint0，它负责传输设备的描述信息，同时也负责传输 PC 与设备之间的控制信息，如图 17-9 所示。

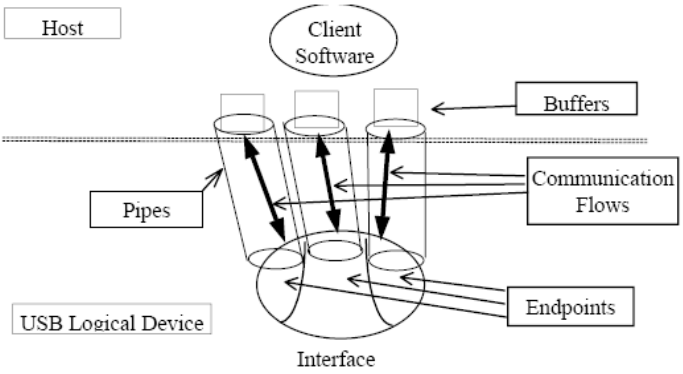

图 17-9　USB 管道与端点

17.1.4　USB 四种传输模式

当 USB 插入 USB 总线时，USB 控制器会自动为该 USB 设备分配一个数字来标示这个设备。另外，在设备的每个端点都有一个数字来表明这个端点。

USB 设备驱动向 USB 控制器驱动请求的每次传输被称为一个事务（Transaction），事务有四种类型，分别是 Bulk Transaction、Control Transaction、Interrupt Transaction 和 Isochronous Transaction。每次事务都会分解成若干个数据包在 USB 总线上传输。每次传输必须历经两个或三个部分，第一部分是 USB 控制器向 USB 设备发出命令，第二部分是 USB 控制器和 USB 设备之间传递读写请求，其方向主要看第一部分的命令是读还是写，第二部分有时候可以没有。第三部分是握手信号。以下针对这四种传输，分别进行讲解。

1．Bulk 传输事务

顾名思义，改种事务传输主要是大块的数据，传送这种事务的管道叫做 Bulk 管道。这种事务传输的时候分为三部分，如图 17-10 所示。第一部分是 Host 端发出一个 Bulk 的令牌请求，如果令牌是 IN 请求则是从 Device 到 Host 的请求，如果是 OUT 令牌，则是从 Host 到 Device 端的请求。

第二部分是传送数据的阶段，根据先前请求的令牌的类型，数据传输有可能是 IN 方向，也有可能是 OUT 方向。传输数据的时候用 DATA0 和 DATA1 令牌携带着数据交替传送。

第三部分是握手信号。如果数据是 IN 方向，握手信号应该是 Host 端发出，如果是 OUT 方向，握手信号应该是 Device 端发出。握手信号可以为 ACK，表示正常响应，也可以是 NAK 表示没有正确传送。STALL 表示出现主机不可预知的错误。

在第二部分，即传输数据包的时候，数据传送由 DATA0 和 DATA1 数据包交替发送。数据传输格式 DATA1 和 DATA0，这两个是重复数据，确保在 1 数据丢失时 0 可以补上，不至于数据丢失。如图 17-11 所示。

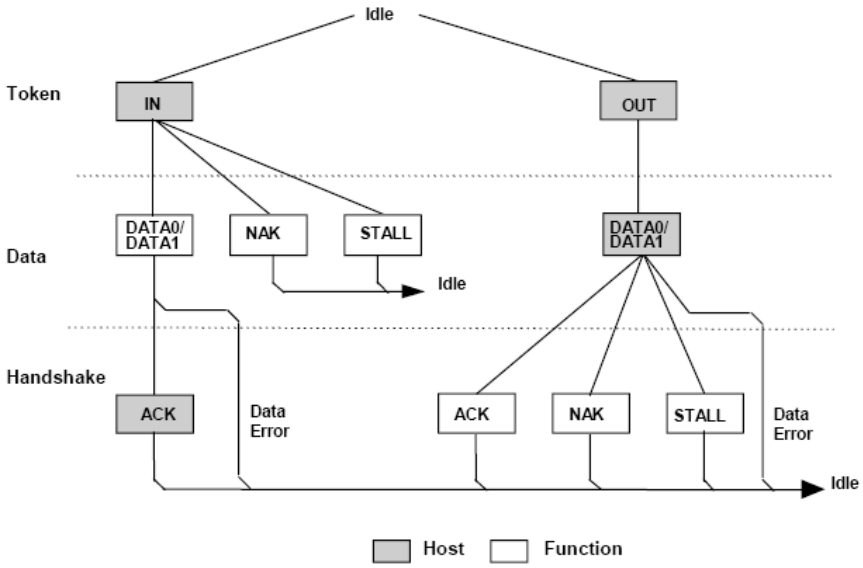

图 17-10　Bulk 传输

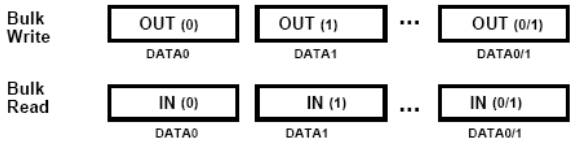

图 17-11　Bulk 传输时的令牌

2．控制传输事务

控制传输是负责向 USB 设置一些控制信息，传送这种事务的管道是控制管道。在每个 USB 设备中都会有控制管道，也就是说控制管道在 USB 设备中是必须的。控制传输也分为三个阶段，即令牌阶段、数据传送阶段、握手阶段，如图 17-12 所示。

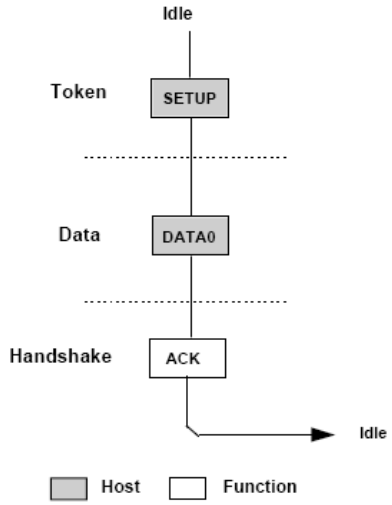

图 17-12　控制传输事务

3．中断传输事务

在 USB 设备中，有种处理机制类似于 PCI 中断的机制，这就是中断事务。中断事务的数据量很小，一般用于通知 Host 某个事件的来临，例如 USB 鼠标，鼠标移动或者鼠标单击等操作都会通过中断管道来向 Host 传送事件。在中断事务中，也分为三个阶段，即令牌阶段、数据传输阶段、握手阶段，如图 17-13 所示。

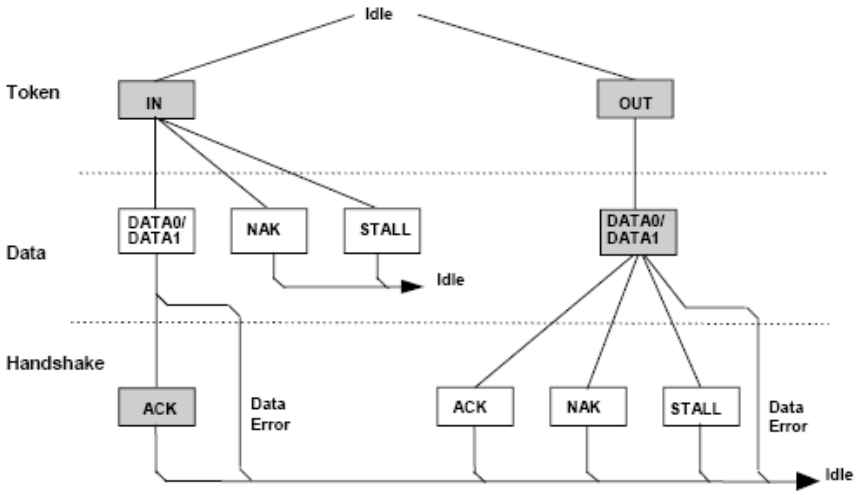

图 17-13　中断传输事务

4．同步传输事务

USB 设备中还有一种事务叫同步传输事务，这种事务能保证传输的同步性。例如，在 USB 摄像头中传输视频数据的时候会采用这种事务，这种事务能保证每秒有固定的传输量，但与 Bulk 传输不同，它允许有一定的误码率，这样符合视频会议等传输的需求，因为视频会议首先要保证实时性，在一定条件下，允许有一定的误码率。同步传输事务有只有两个阶段，即令牌阶段、数据阶段，因为不关心数据的正确性，故没有握手阶段，如图 17-14 所示。

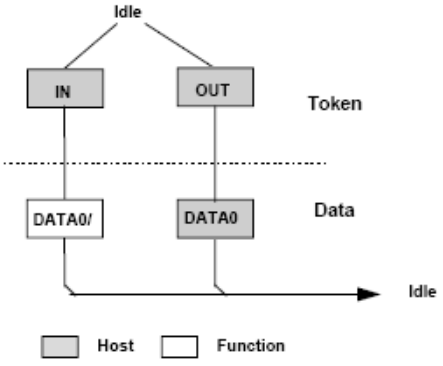

图 17-14　同步传输事务

17.2　Windows 下的 USB 驱动

在 Windows 上开发 USB 驱动相对来说比较简单，主要是因为微软已经提供了完备的 USB 总线驱动，程序员编写的设备驱动只需调用总线驱动即可。在 Windows 上还有一些工具软件可以帮助开发者查看 USB 的各类信息，包括设备描述符、配置描述符等。当然，这些描述符在驱动中也会用到。本节将介绍这些工具软件，并介绍这些描述符。

17.2.1　观察 USB 设备的工具

在学习编写 USB 驱动之前，有几个 USB 查看工具需要向读者介绍一下，通过用这些工具能方便地学习 USB 协议。

首先需要介绍的就是 DDK 中提供的工具，该工具叫 usbview，位于 DDK 的子目录 src\wdm\usb\usbview 下，需要用 DDK 编译环境进行编译。如图 17-15 所示为 usbview 的界面，在笔者的计算机里插入了一个 USB 移动硬盘，在这个软件中已经清楚地列举除了该 USB 设备的各个信息，如图设备描述符、管道描述符等。

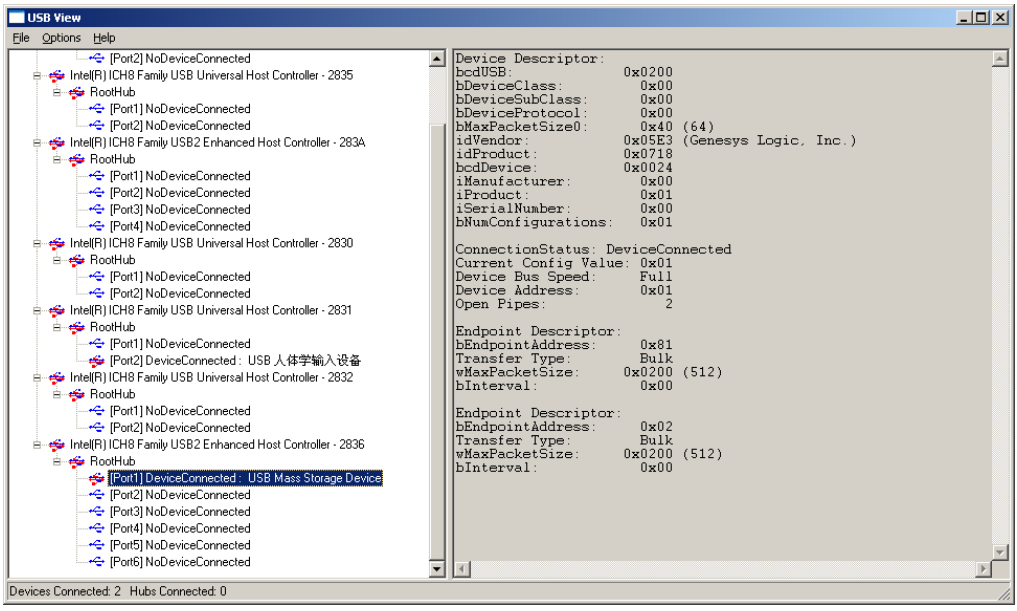

图 17-15　USBView

另一个有用的工具是 BusHound，如图 17-16 所示。BusHound 用于监视 USB 设备的传输数据，它 的实现原理是在 USB 设备驱动之上加载一层过滤驱动程序，将 IRP 进行拦截，因此可以观察到所有 USB 数据的传输。使用该软件时需要指明监视哪种 USB 设备，如图 17-16 所示，在需要监视的设备上打钩。笔者这里监视的是一个 USB 移动硬盘。另外，在下面会列出该设备的基本信息，如管道 0 是控制管道，管道 1 是输出管道，管道 2 是输入

管道。

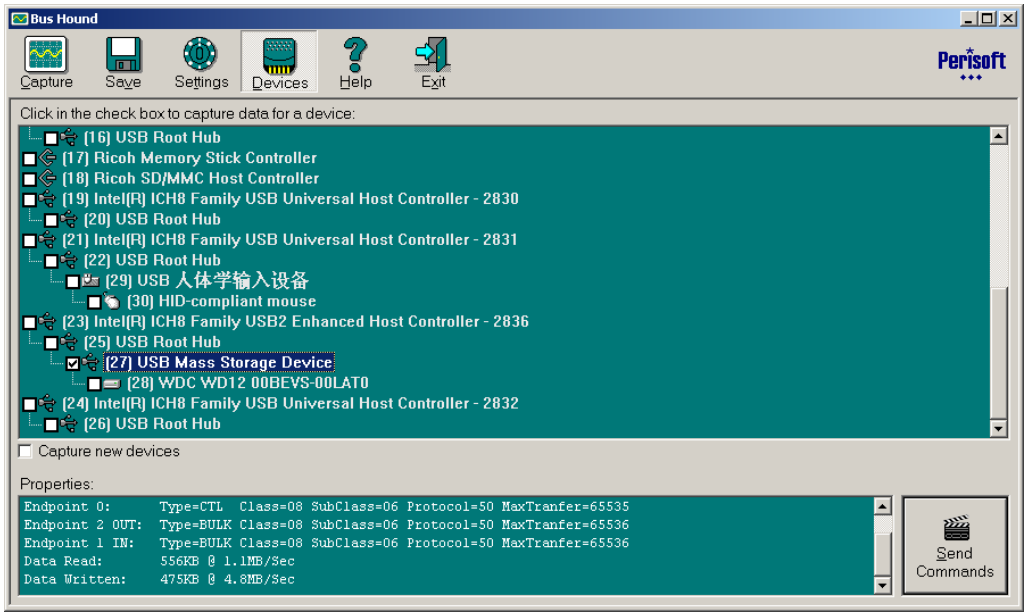

图 17-16　BusHound

该软件将 USB 的传输完全进行监视，包括每个 USB 的各个管道中的传输情况，都一一进行记录，非常有利于调试驱动。如图 17-17 所示，Device 一栏中标识是何种设备，例如 27.2 意味着第 27 号设备的第 2 号管道。Phase 一栏标识传输是输入还是输出。在 Data 一栏中记录着一次传输的具体内容。

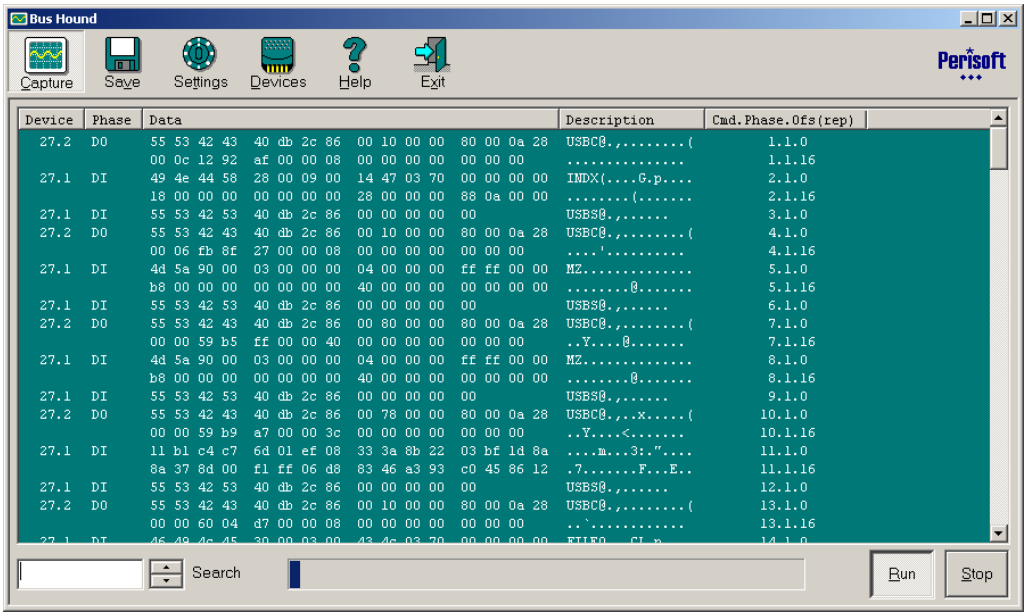

图 17-17　BusHound 监视数据传输

17.2.2 USB 设备请求

USB 的请求是通过控制管道传输的，请求是 8 个字节，按照如表 17-1 所示的排列发送：

表 17-1

偏移量	变量	大小	数值	描　　述
0	bmRequestType	1 字节	位图	第 7 位: 数据方向位 0 = 主机到设备 1 = 设备到主机 第 6-5 位: 类型 0 = 标准 1 = 类 2 = 厂商自定义 3 = 保留 第 4-0 位: 接收者 0 = 对设备的请求 1 = 对接口的请求 2 = 对管道（端点）的请求 3 = 其他 4-31 = 保留
1	bRequest	1 字节	数值	请求类别
2	wValue	2 字节	数值	不同请求含义不同
4	wIndex	2 字节	数值	不同请求含义不同
6	wLength	2 字节	数值	表示需要有多少数据返回

其中，bRequest 代表不同的 USB 请求，它们分别是以下的几种请求，定义在 DDK 的 usb100.h 文件中：

```
#define USB_REQUEST_GET_STATUS                  0x00
#define USB_REQUEST_CLEAR_FEATURE               0x01
#define USB_REQUEST_SET_FEATURE                 0x03
#define USB_REQUEST_SET_ADDRESS                 0x05
#define USB_REQUEST_GET_DESCRIPTOR              0x06
#define USB_REQUEST_SET_DESCRIPTOR              0x07
#define USB_REQUEST_GET_CONFIGURATION           0x08
#define USB_REQUEST_SET_CONFIGURATION           0x09
#define USB_REQUEST_GET_INTERFACE               0x0A
#define USB_REQUEST_SET_INTERFACE               0x0B
```

17.2.3 设备描述符

在控制管道发起 USB 设备请求，其中很常见的请求是 USB_REQUEST_GET_DESCRIPTOR，即请求 USB 设备回答设备或者管道描述符。在请求描述符时，

bmRequestType 可以指定是针对设备还是针对管道的。

当请求设备描述符后，设备会回答主机该设备的设备描述符，设备描述符是一种固定的数据结构，它定义在 DDK 中的 usb100.h 文件中。

```
typedef struct  USB DEVICE DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
   USHORT bcdUSB;
   UCHAR bDeviceClass;
   UCHAR bDeviceSubClass;
   UCHAR bDeviceProtocol;
   UCHAR bMaxPacketSize0;
   USHORT idVendor;
   USHORT idProduct;
   USHORT bcdDevice;
   UCHAR iManufacturer;
   UCHAR iProduct;
   UCHAR iSerialNumber;
   UCHAR bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
```

- bLength：设备描述符的 bLength 域应等于 18。
- bDescriptorType：bDescriptorType 域应等于 1，以指出该结构是一个设备描述符。
- bcdUSB：bcdUSB 域包含该描述符遵循的 USB 规范的版本号（以 BCD 编码）。现在，设备可以使用值 0x0100 或 0x0110 来指出它所遵循的是 1.0 版本还是 1.1 版本的 USB 规范。
- bDeviceClass：指出设备类型。
- bDeviceSubClass：指出设备子类型。
- bDeviceProtocol：指出设备类型所使用的协议。
- bMaxPacketSize0：设备描述符的 bMaxPacketSize0 域，给出了默认控制端点（端点 0）上的数据包容量的最大值。
- idVendor：厂商代码。
- idProduct：厂商专用的产品标识。
- bcdDevice：bcdDevice 指出设备的发行版本号（0x0100 对应版本 1.0）。
- iManufacturer、iProduct、iSerialNumber：iManufacturer、iProduct 和 iSerialNumber 域指向一个串描述符，该串描述符用人类可读的语言描述设备生产厂商、产品和序列号。这些串是可选的，0 值代表没有描述串。如果在设备上放入了序列号串，Microsoft 建议应使每个物理设备的序列号唯一。
- bNumConfigurations：bNumConfigurations 指出该设备能实现多少种配置。Microsoft 的驱动程序仅工作于设备的第一种配置（1 号配置）。

如图 17-18 所示为笔者用 BusHound 截获的 USB 移动硬盘的请求设备描述符，前面已经介绍过，在控制管道中，传输分为三个阶段。第一阶段是令牌阶段，这里 Host 向设备发送“80 06 00 01 00 00 12 00”8 个字节，可以参见表 17-1 中的解释。第二阶段是数据传

输阶段，方向是由设备传给主机，这个例子中设备给主机传递了 18（0x12）个字节，这 18 个字节对应着 USB_DEVICE_DESCRIPTOR 数据结构。第三阶段是握手阶段，在 BusHound 软件中没有体现出来。

图 17-18　用 BusHound 抓取设备描述符

17.2.4　配置描述符

每个设备有一个或多个配置描述符，它们描述了设备能实行的各种配置方式。DDK 中定义的配置描述符结构如下：

```
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT wTotalLength;
    UCHAR bNumInterfaces;
    UCHAR bConfigurationValue;
    UCHAR iConfiguration;
    UCHAR bmAttributes;
    UCHAR MaxPower;
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR;
```

- bLength：bLength 应该为 9。
- bDescriptorType：bDescriptorType 应该为 2，即是一个 9 字节长的配置描述符。
- wTotalLength：wTotalLength 域为该配置描述符长度加上该配置内所有接口和端点描述符长度的总和。通常，主机在发出一个 GET_DESCRIPTOR 请求并正确接收到 9 字节长的配置描述符后，就会再发出一个 GET_DESCRIPTOR 请求并指定这个总长度。第二个请求把这个大联合描述符传输回来。
- bNumInterfaces：指出该配置有多少个接口。
- bConfigurationValue：bConfigurationValue 域是该配置的索引值。
- iConfiguration：iConfiguration 域是一个可选的串描述符索引，指向描述该配置的 Unicode 字符串。此值为 0 表明该配置没有串描述符。
- bmAttributes：bmAttributes 字节包含描述该配置中设备电源和其他特性的位掩码。
- MaxPower：MaxPower 域中指出要从 USB 总线上获取的最大电流量。

如图 17-19 所示为笔者用 BusHound 截获的 USB 移动硬盘的请求配置描述符。其中第一行是 Host 向 Device 发送 Token 令牌，而第二行是 Device 向 Host 返回的数据。

图 17-19　用 BusHound 抓取设配置描述符

17.2.5　接口描述符

每个配置有一个或多个接口描述符，它们描述了设备提供功能的接口。

```
typedef struct  USB INTERFACE DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
   UCHAR bInterfaceNumber;
   UCHAR bAlternateSetting;
   UCHAR bNumEndpoints;
   UCHAR bInterfaceClass;
   UCHAR bInterfaceSubClass;
   UCHAR bInterfaceProtocol;
   UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR;
```

- bLength：bLength 应该为 9。
- bDescriptorType：bDescriptorType 域应为 4。
- bInterfaceNumber：bInterfaceNumber 是索引值。
- bAlternateSetting：bAlternateSetting 是索引值。用在 SET_INTERFACE 控制事务中以指定要激活的接口。
- bNumEndpoints：bNumEndpoints 域指出该接口有多少个端点，不包括端点 0，端点 0 被认为总是存在的，并且是接口的一部分。
- bInterfaceClass：bInterfaceClass 为接口类。
- bInterfaceSubClass：bInterfaceSubClass 为子接口类。
- bInterfaceProtocol：bInterfaceProtocol 为协议。
- iInterface：iInterface 是一个串描述符的索引，0 表示该接口无描述串。

如图 17-20 所示为笔者用 BusHound 截获的 USB 移动硬盘的请求配置描述符和接口描述符。其中第一行是 Host 向 Device 发送 Token 令牌，而第二行是 Device 向 Host 返回的数据。可以看出图中有一个配置描述符，后面紧接着一个接口描述符（“09 04 00 00 02 00 06 50 05”），后面还有两个接口描述符（下一节介绍）。

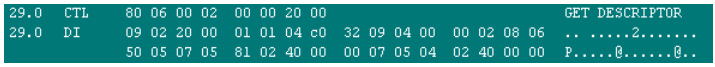

图 17-20　用 BusHound 抓取接口描述符

17.2.6　端点描述符

接口可以没有或有多个端点描述符，它们描述了处理事务的端点。DDK 中定义的端点描述符结构如下：

```
typedef struct  USB ENDPOINT DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
   UCHAR bEndpointAddress;
```

```
    UCHAR bmAttributes;
    USHORT wMaxPacketSize;
    UCHAR bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;
```

- bLength：bLength 应为 7。
- bDescriptorType：bDescriptorType 应该为 5。
- bEndpointAddress：bEndpointAddress 域编码端点的方向性和端点号。
- bmAttributes：bmAttributes 的低两位指出端点的类型。0 代表控制端点，1 代表等时端点，2 代表批量端点，3 代表中断端点。
- wMaxPacketSize：wMaxPacketSize 值指出该端点在一个事务中能传输的最大数据量。
- bInterval：中断端点和等时端点描述符还有一个用于指定循检间隔时间的 bInterval 域。

17.3 USB 驱动开发实例

本节具体介绍如何进行 USB 驱动的开发，本节采用的源码来源自 DDK 的源程序，其位置在 DDK 子目录的 src\wdm\usb\bulkusb 目录下。该示例很全面地支持了即插即用 IRP 的处理，也很全面地支持了电源管理，同时很好地支持了 USB 设备的 bulk 读写。如果从头开发 USB 驱动，往往很难达到 USB 驱动的稳定性，所以强烈建议读者在此驱动修改的基础上进行 USB 驱动开发。

17.3.1 功能驱动与物理总线驱动

DDK 已经为 USB 驱动开发人员提供了功能强大的 USB 物理总线驱动（PDO），程序员需要做的事情是完成功能驱动（FDO）的开发。驱动开发人员不需要了解 USB 如何将请求转化成数据包等细节，程序员只需要指定何种管道，发送何种数据即可。

当功能驱动想向某个管道发出读写请求时，首先构造请求发给 USB 总线驱动。这种请求是标准的 USB 请求，被称为 URB（USB Request Block），即 USB 请求块。这种 URB 被发送到 USB 物理总线驱动以后，被 USB 总线驱动所解释，进而转化成请求发往 USB HOST 驱动或者 USB HUB 驱动，如图 17-21 所示。

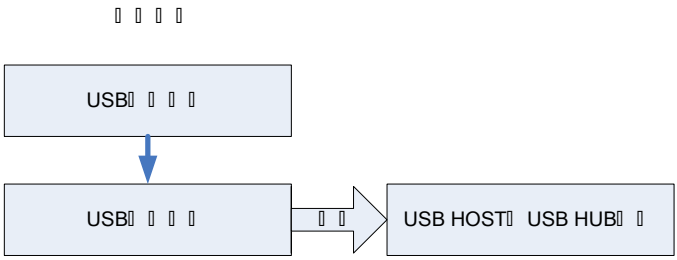

图 17-21 总线驱动与功能驱动的关系

可以看出，USB 总线驱动完成了大部分工作，并留给 USB 功能驱动标准的接口，即 URB 请求。USB 驱动开发人员只需要根据不同的 USB 设备的设计要求，在相应的管道中发起 URB 请求即可。

17.3.2　构造 USB 请求包

USB 驱动在与 USB 设备通信的时候，如在控制管道中获取设备描述符、配置描述符、端点描述符，或者在 Bulk 管道中获取大量数据，都是通过创建 USB 请求包（URB）来完成的。URB 中填充需要对 USB 的请求，然后将 URB 作为 IRP 的一个参数传递给底层的 USB 总线驱动。在 USB 总线驱动中，能够解释不同 URB，并将其转化为 USB 总线上的相应数据包。

DDK 提供了构造 URB 的内核函数 UsbBuildGetDescriptorRequest，其声明如下：

```
VOID
  UsbBuildGetDescriptorRequest(
    IN OUT PURB  Urb,
    IN USHORT  Length,
    IN UCHAR  DescriptorType,
    IN UCHAR  Index,
    IN USHORT  LanguageId,
    IN PVOID  TransferBuffer  OPTIONAL,
    IN PMDL  TransferBufferMDL  OPTIONAL,
    IN ULONG  TransferBufferLength,
    IN PURB  Link  OPTIONAL
    );
```

- Urb：用来输出的 URB 结构的指针。
- Length：用来描述该 URB 结构的大小。
- DescriptorType：描述该 URB 的类型。它可以是 USB_DEVICE_DESCRIPTOR_TYPE、USB_CONFIGURATION_DESCRIPTOR_TYPE 和 USB_STRING_DESCRIPTOR_TYPE。
- Index：用来描述设备描述符的索引。
- LanguageId：用来描述语言 ID。
- TransferBuffer：如果用缓冲区读取设备，TransferBuffer 是缓冲区内存的指针。
- TransferBufferMDL：如果用直接读取内存时，TransferBufferMDL 是直接读取内存时 MDL 的指针。
- TransferBufferLength：对于该 URB 所操作内存的大小。

在功能驱动中，所有与 USB 的通信，都需要用这个函数创建 URB，并通过 IRP 发送到底层 USB 总线驱动，以下是一个最基本的示例。

```
#001            UsbBuildGetDescriptorRequest(
#002                  urb,
#003                  (USHORT) sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST),
#004                  USB_DEVICE_DESCRIPTOR_TYPE,
#005                  0,
```

```
#006                    0,
#007                    deviceDescriptor,
#008                    NULL,
#009                    siz,
#010                    NULL);
```

17.3.3 发送 USB 请求包

功能驱动将 URB 包构造完毕后，就可以发送到底层总线驱动上了。URB 包要和一个 IRP 相关联起来，这就需要用 IoBuildDeviceIoControlRequest 创建一个 IO 控制码的 IRP，然后将 URB 作为 IRP 的参数，用 IoCallDriver 将 URB 发送到底层总线驱动上。由于上层驱动无法知道底层驱动是同步还是异步完成的，因此需要做一个判断。if 语句判断当异步完成 IRP 时，用事件等待总线驱动完成这个 IRP。

```
#001   //该函数实现对发送 URB 到 USB 物理总线驱动
#002   NTSTATUS
#003   CallUSBD(
#004      IN PDEVICE_OBJECT DeviceObject,
#005      IN PURB          Urb
#006      )
#007   {
#008      PIRP             irp;
#009      KEVENT           event;
#010      NTSTATUS         ntStatus;
#011      IO_STATUS_BLOCK    ioStatus;
#012      PIO_STACK_LOCATION nextStack;
#013      PDEVICE_EXTENSION  deviceExtension;
#014
#015      //首先是变量初始化
#016      irp = NULL;
#017      deviceExtension = DeviceObject->DeviceExtension;
#018      //初始化事件
#019      KeInitializeEvent(&event, NotificationEvent, FALSE);
#020     //创建 IO 控制码相关的 IRP
#021      irp = IoBuildDeviceIoControlRequest(IOCTL_INTERNAL_USB_SUBMIT_URB,
#022                                      deviceExtension->TopOfStackDeviceObject,
#023                                      NULL,
#024                                      0,
#025                                      NULL,
#026                                      0,
#027                                      TRUE,
#028                                      &event,
#029                                      &ioStatus);
#030
#031      if(!irp) {
#032         //如果 IRP 创建失败则返回
#033         BulkUsb_DbgPrint(1, ("IoBuildDeviceIoControlRequest failed\n"));
#034         return STATUS_INSUFFICIENT_RESOURCES;
#035      }
#036     //得到下一层设备栈
#037      nextStack = IoGetNextIrpStackLocation(irp);
#038      ASSERT(nextStack != NULL);
#039      nextStack->Parameters.Others.Argument1 = Urb;
```

```
#040      BulkUsb_DbgPrint(3, ("CallUSBD::"));
#041      BulkUsb_IoIncrement(deviceExtension);
#042     //通过 IoCallDriver 将 IRP 发送到底层驱动
#043      ntStatus = IoCallDriver(deviceExtension->TopOfStackDeviceObject, irp);
#044     //如果 IRP 是异步完成时，等待其结束
#045      if(ntStatus == STATUS_PENDING) {
#046         //等待 IRP 结束
#047         KeWaitForSingleObject(&event,
#048                               Executive,
#049                               KernelMode,
#050                               FALSE,
#051                               NULL);
#052         ntStatus = ioStatus.Status;
#053      }
#054      //调用结束
#055      BulkUsb_DbgPrint(3, ("CallUSBD::"));
#056      BulkUsb_IoDecrement(deviceExtension);
#057      return ntStatus;
#058   }
```

此段代码可以在配套光盘中本章的 sys 目录下找到。

17.3.4　USB 设备初始化

USB 驱动的初始化和一般驱动类似，首先是进入入口函数 DriverEntry，在 DriverEntry 函数中，分别指定各个 IRP 的派遣函数地址、指定 AddDevice 例程函数地址、指定 Unload 例程函数地址等。

在 AddDevice 例程中，创建功能设备对象，然后将该对象挂载在总线设备对象之上，从而形成设备栈。另外为设备创建一个设备链接，便于应用程序可以找到这个设备。也可以根据具体需要，从注册表中读取一些必要的设置。

17.3.5　USB 设备的插拔

由于 USB 设备驱动是基于 WDM 框架的，因此需要对即插即用消息进行处理。BulkUSB 程序对即插即用 IRP 的支持非常完善，具体可以参照其代码，这里简单提一下其对插拔的处理。

插拔设备会设计 4 个即插即用 IRP，包括 IRP_MN_START_DEVICE、IRP_MN_STOP_DEVICE、IRP_MN_EJECT 和 IRP_MN_SURPRISE_REMOVAL。其中，IRP_MN_START_DEVICE 消息是当驱动争取加载并运行时，操作系统的即插即用管理器会将这个 IRP 发往设备驱动。因此，当获得这个 IRP 后，USB 驱动需要获得 USB 设备类别描述符，如设备描述符、配置描述符、接口描述符、端点描述符等。并通过这些描述符，从中获取有用信息，记录在设备扩展中。

IRP_MN_STOP_DEVICE 是设备关闭前，即插即用管理器发的 IRP。USB 驱动获得这个 IRP 时，应该尽快结束当前执行的 IRP，并将其逐个取消掉。另外，在设备扩展中还应该

有表示当前状态的变量，当 IRP_MN_STOP_DEVICE 来临时，将当前状态记录成停止状态。

IRP_MN_EJECT 是设备被正常弹出，而 IRP_MN_SURPRISE_REMOVAL 则是设备非自然弹出，有可能意外掉电或者强行拔出等。在这种 IRP 到来的时候，应该强迫所有未完成的读写 IRP 结束并取消。并且将当前的设备状态设置成设备被拔掉。

17.3.6 USB 设备的读写

USB 设备接口主要是为了传送数据，80%的传输是通过 Bulk 管道。在 BulkUSB 驱动中，Bulk 管道的读取是在 IRP_MJ_READ 和 IRP_MJ_WRITE 的派遣函数中，这样在应用程序中就可以通过 ReadFile 和 WriteFile 等 API 对设备进行操作了。

在 IRP_MJ_READ 和 IRP_MJ_WRITE 的派遣例程中设置了完成例程，如图 17-22 所示。其原理是将读写的大小分成单位为 BULKUSB_MAX_TRANSFER_SIZE 的若干块，依次将请求发往底层 USB 总线驱动。第一个块是派遣例程先设置 BULKUSB_MAX_TRANSFER_SIZE 大小的读写，并设置完成例程，然后将请求发往 USB 总线驱动。当 USB 总线驱动完成 BULKUSB_MAX_TRANSFER_SIZE 大小的读写后，会调用读写的完成例程。

这时候在完成例程中再次发起 BULKUSB_MAX_TRANSFER_SIZE 大小的读写，并将请求发往底层 USB 总线驱动，当 USB 总线驱动完成后，又会进入完成例程。之后发送第三个数据块，并且依此类推直到传送完毕。

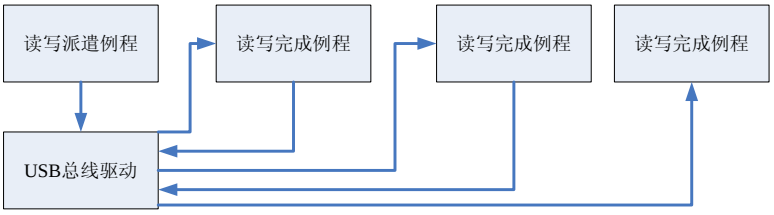

图 17-22 USB 读写派遣例程与完成例程

以下是 BulkUSB 的读写派遣函数的部分代码：

```
#001   NTSTATUS
#002   BulkUsb_DispatchReadWrite(
#003       IN PDEVICE_OBJECT DeviceObject,
#004       IN PIRP           Irp
#005       )
#006   {
#007       PMDL                mdl;
#008       PURB                urb;
#009       ULONG               totalLength;
#010       ULONG               stageLength;
#011       ULONG               urbFlags;
#012       BOOLEAN             read;
#013       NTSTATUS            ntStatus;
#014       ULONG_PTR           virtualAddress;
#015       PFILE_OBJECT        fileObject;
#016       PDEVICE_EXTENSION   deviceExtension;
```

```
#017      PIO_STACK_LOCATION   irpStack;
#018      PIO_STACK_LOCATION   nextStack;
#019      PBULKUSB_RW_CONTEXT  rwContext;
#020      PUSBD_PIPE_INFORMATION pipeInformation;
#021
#022     //初始化变量
#023     urb = NULL;
#024      mdl = NULL;
#025      rwContext = NULL;
#026      totalLength = 0;
#027      irpStack = IoGetCurrentIrpStackLocation(Irp);
#028      fileObject = irpStack->FileObject;
#029      read = (irpStack->MajorFunction == IRP_MJ_READ) ? TRUE : FALSE;
#030      deviceExtension = (PDEVICE_EXTENSION) DeviceObject->DeviceExtension;
#031     //....略
#032
#033     //设置完成例程的参数
#034      rwContext = (PBULKUSB_RW_CONTEXT) ExAllocatePool(NonPagedPool,
#035                            sizeof(BULKUSB_RW_CONTEXT));
#036     //...略
#037      if(Irp->MdlAddress) {
#038         totalLength = MmGetMdlByteCount(Irp->MdlAddress);
#039      }
#040     //设置 URB 标志
#041      urbFlags = USBD_SHORT_TRANSFER_OK;
#042      virtualAddress = (ULONG_PTR) MmGetMdlVirtualAddress(Irp->MdlAddress);
#043
#044     //判断是读还是写
#045      if(read) {
#046         urbFlags |= USBD_TRANSFER_DIRECTION_IN;
#047      }
#048      else {
#049         urbFlags |= USBD_TRANSFER_DIRECTION_OUT;
#050      }
#051
#052     //设置本次读写的大小
#053      if(totalLength > BULKUSB_MAX_TRANSFER_SIZE) {
#054         stageLength = BULKUSB_MAX_TRANSFER_SIZE;
#055      }
#056      else {
#057         stageLength = totalLength;
#058      }
#059
#060     //建立 MDL
#061      mdl = IoAllocateMdl((PVOID) virtualAddress,
#062                          totalLength,
#063                          FALSE,
#064                          FALSE,
#065                          NULL);
#066     //将新 MDL 进行映射
#067      IoBuildPartialMdl(Irp->MdlAddress,
#068                        mdl,
#069                        (PVOID) virtualAddress,
#070                        stageLength);
#071
#072     //申请 URB 数据结构
```

```
    #073      urb = ExAllocatePool(NonPagedPool,sizeof(struct _URB_BULK_OR_INTERRUPT_
TRANSFER));
    #074
    #075     //建立 Bulk 管道的 URB
    #076      UsbBuildInterruptOrBulkTransferRequest(
    #077                           urb,
    #078                           sizeof(struct _URB_BULK_OR_INTERRUPT_TRANSFER),
    #079                           pipeInformation->PipeHandle,
    #080                           NULL,
    #081                           mdl,
    #082                           stageLength,
    #083                           urbFlags,
    #084                           NULL);
    #085
    #086     //设置完成例程参数
    #087      rwContext->Urb           = urb;
    #088      rwContext->Mdl           = mdl;
    #089      rwContext->Length         = totalLength - stageLength;
    #090      rwContext->Numxfer        = 0;
    #091      rwContext->VirtualAddress  = virtualAddress + stageLength;
    #092      rwContext->DeviceExtension = deviceExtension;
    #093     //设置设备堆栈
    #094      nextStack = IoGetNextIrpStackLocation(Irp);
    #095      nextStack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
    #096      nextStack->Parameters.Others.Argument1 = (PVOID) urb;
    #097      nextStack->Parameters.DeviceIoControl.IoControlCode =
    #098                                        IOCTL_INTERNAL_USB_SUBMIT_URB;
    #099     //设置完成例程
    #100      IoSetCompletionRoutine(Irp,
    #101                          (PIO_COMPLETION_ROUTINE)BulkUsb_ReadWriteCompletion,
    #102                          rwContext,
    #103                          TRUE,
    #104                          TRUE,
    #105                          TRUE);
    #106     //将当前 IRP 阻塞
    #107      IoMarkIrpPending(Irp);
    #108     //将 IRP 转发到底层 USB 总线驱动
    #109      ntStatus = IoCallDriver(deviceExtension->TopOfStackDeviceObject,
    #110                           Irp);
    #111     //...略去对不成功时的处理
    #112      return STATUS_PENDING;
    #113   }
```

此段代码可以在配套光盘中本章的 sys 目录下找到。

17.4 小结

本章介绍了 USB 总线协议的基本框架，其中包括 USB 总线的拓扑结构，USB 通信的流程，还有 USB 的四种传输模式。笔者用一些工具软件带领读者分析了各种 USB 令牌、设备描述符等。

USB 驱动程序的主要功能就是设置这些 USB 令牌，和获取 USB 设备描述符。USB 驱动程序将这些请求最终转化为 USB 请求包（URB 包），然后发往 USB 总线驱动程序。USB

总线驱动提供了丰富的功能，它封装了 USB 协议，提供了标准的接口。这使得 USB 驱动程序的编写变得简单，程序员不必过多地了解 USB 总线协议，就可以编写出功能强大的 USB 驱动程序。