USB驱动开发

By NeMoon

2007.03

第一章 关于USB技术的简单介绍

- 1. USB的全称为Universal Serial Bus，即通用串行总线，USB技术由Compaq、DEC、IBM、Intel、Microsoft、NEC和Northern Telecom等公司共同协商制定，该项目技术对微型计算机外部设备的连接和使用方式作出了新的规定，USB技术的开发和应用是本世纪末微型计算机外部总线结构的重大变革。USB具有以下特点：
- (1)有较高的传输速率。USB1.1支持全速和低速两种方式：全速速率为12Mb/s，低速速率为1.5Mb/s；USB2.0除支持USB1.1的两种速度方式外，还增加了速率可达480Mb/s的高速方式。

- (2)使用方便灵活。USB支持即插即用和热插拔，它允许在任何时候连接和断开外设，当外设被连接时，系统会自动检测到外设并准备使用。

- (3)易于扩展。通过根集线器可携带127个设备，真正实现多个外设共用一个接口。

- 高速和低速设备
- USB规范中定义了两种设备，高速设备和低速设备。低速设备以1.5Mb/sec速率通信，高速设备以12Mb/sec速率通信。hub能用电子方式区分这两种设备。发生在总线上的通讯通常都是高速的，hub一般不向低速设备发送数据。操作系统把任何发往低速设备的消息前加上一个前导包，这将使hub临时降为低速，并完成低速设备的数据发送。

- 随着计算机技术和信息技术的飞速发展，计算机或工控机经常被用来对各种数据进行采集。现在常用的方法主要有两种：

  (1)通过数据采集板卡，常用的有A/D卡等。采用板卡不仅安装麻烦、体格昂贵、易受环境的影响，降低系统的精度和稳定性。

  (2)普通的外置式采样系统一般通过RS-232与PC机相连，由于PC机的限制，RS-232的最高数据传输速率不超过115Kb，对于中高速的采集系统也很难达到要求。

  USB的出现，很好地解决了以上这些冲突，很容易就能实现低成本、高带宽、易扩展、高可靠性、安装方便、多点的数据采集，已逐步成为现代数据传输的发展趋势。

- 第二章　USB设备驱动程序开发介绍
- 固件设计
- 设备驱动程序设计
- 应用软件设计

- 固件设计

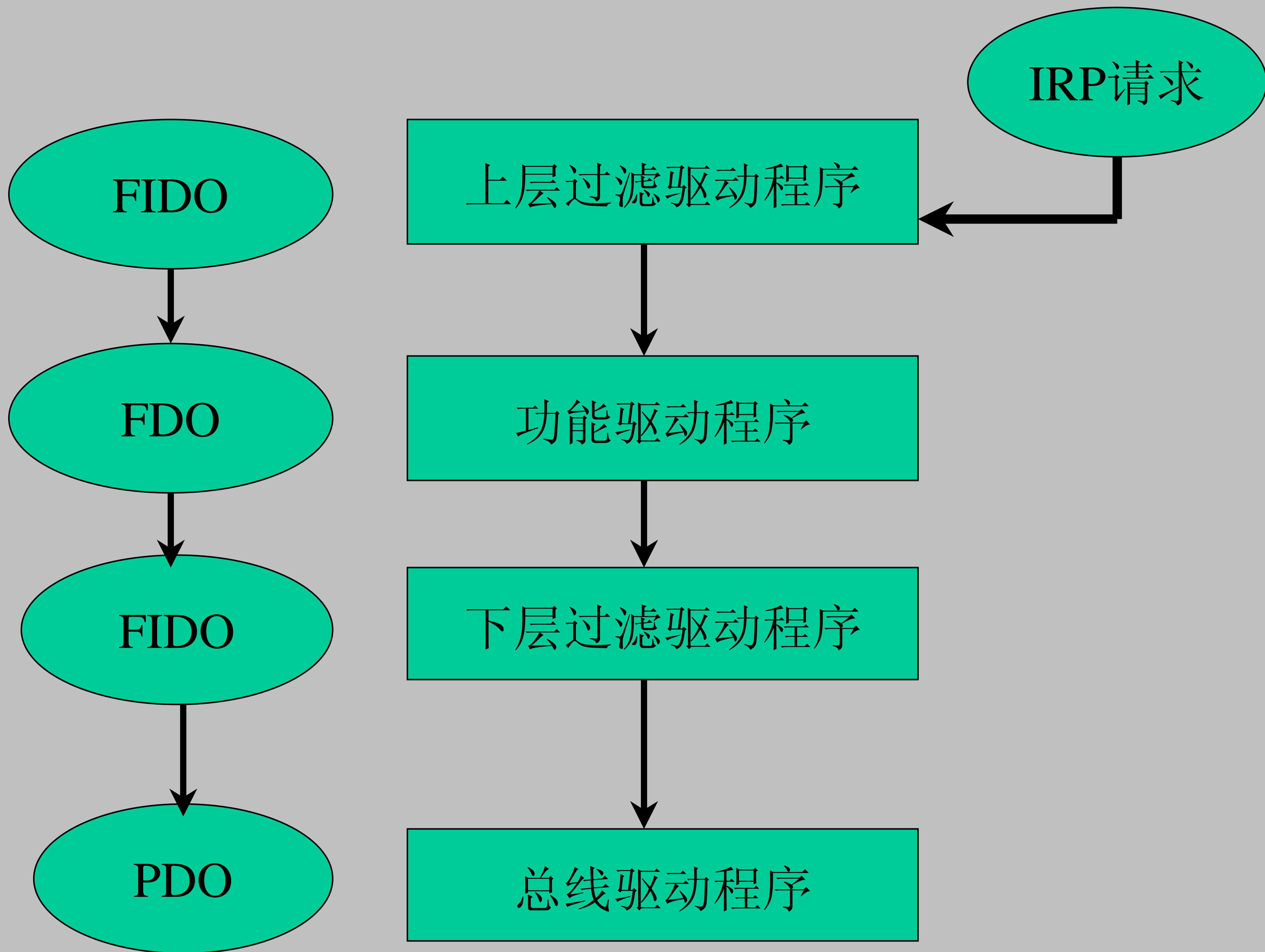
- 在实际开发中使用了两种传输方式：**控制传输**和**块传输**。**控制传输**用来实现位于主机上的USB总线驱动程序（USB.SYS ）以及编写的功能驱动程序对设备的各种控制操作，而**块传输**用来完成将采集数据从设备传送到主机。USB控制器的工作原理可以简单地描述为：当USB控制器从USB总线检测到主机启动的某一传输请求时，USB控制器通过中断方式将此请求通知主机系统，主机系统通过访问USB控制器的状态寄存器和数据寄存器获得与此次传输有关的各种参数，并根据具体传输参数，对USB控制器的控制寄存器和数据寄存器进行相应的操作，以完成主机的传输请求。

- 设备驱动程序设计

- USB设备驱动程序的设计是基于美国微软公司极力推荐的WDM(Windows Driver Model,Windows驱动程序模型)。WDM采用分层驱动程序模型，对于USB设备来说，可分为USB总线驱动程序和USB功能驱动程序。USB总线驱动程序由操作系统提供，它位于USB功能驱动程序的下面，负责与实际的硬件打交道，实现烦琐的低层通信。USB功能驱动程序由设备开发者编写，位于USB总线驱动程序的上面，不与实际的硬件打交道，而是通过向USB总线驱动程序发送包含URB (USB Request Block，USB请求块)的IRP(I/O Request Packet,I/O请求包)，来实现对USB设备信息的发送或接收。

- WDM模型使用了层次结构。图中左边是一个设备对象堆栈。设备对象是系统为帮助软件管理硬件而创建的数据结构。一个物理硬件可以有多个这样的数据结构。处于堆栈最底层的设备对象称为物理设备对象(physical device object)，或简称为PDO。在设备对象堆栈的中间某处有一个对象称为功能设备对象(functional device object)，或简称FDO。在FDO的上面和下面还会有一些过滤器设备对象(filter device object)。位于FDO上面的过滤器设备对象称为上层过滤器，位于FDO下面(但仍在PDO之上)的过滤器设备对象称为下层过滤器。

•

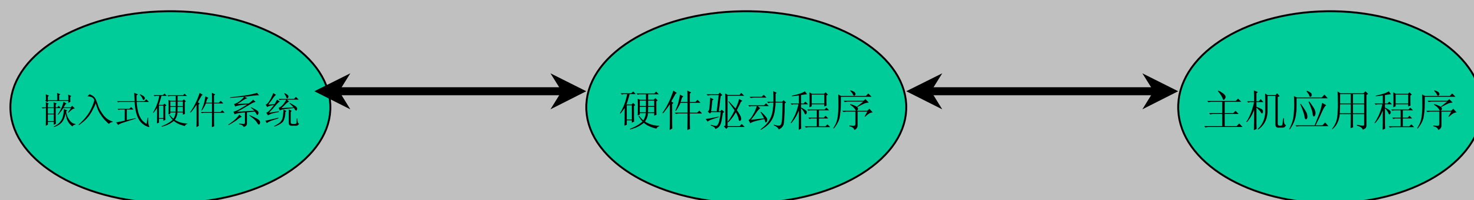


IRP请求
FIDO
上层过滤驱动程序
FDO
功能驱动程序
FIDO
下层过滤驱动程序
PDO
总线驱动程序

- USB数据流动

- USB提供了在一台主机和若干台附属的USB设备之间的通信功能，从终端用户的角度看到的USB系统。但在实际的实现上，具体的系统要比这复杂，不同层次的实现者对USB的有不同要求，这使得我们必须从不同的层次观察USB系统。USB系统提出了一些重要的概念和情况来支持现代个人计算机所提出的可靠性要求，所以USB的分层理解是必须的。

- # 应用软件设计
- 应用软件设计由两部分组成：链接库程序和应用程序。链接库负责与USB功能驱动程序通信并接受应用程序的各种操作请求，而应用程序负责对所采集的数据进行分析处理。
- # 我们在这里就可以总结一下：
- 一套完整的嵌入式系统，由硬件和软件组合而成。应用程序要控制硬件系统的具体工作，但是现在可以看到，它并不是直接参与硬件的控制，在中间层，还有一套具体的驱动程序来执行应用层传递过来的I/O请求，同样，硬件系统要返回给应用程序的数据也要经过驱动程序的处理。



- USB定义了四种数据传输方式
- 控制 ———用于发送和接收USB定义的结构化信息
- 批量 ———用于发送或接收小块无结构数据
- 中断 ———与批量管道相似，但包括一个最大延迟
- 等时 ———用于发送或接收有周期保证的大块无结构数据

- 控制(Control)方式传送：控制传送是双向传送，数据量通常较小。USB系统软件用来主要进行查询、配置和给USB设备发送通用的命令。控制传送方式可以包括8、16、32和64字节的数据，这依赖于设备和传输速度。

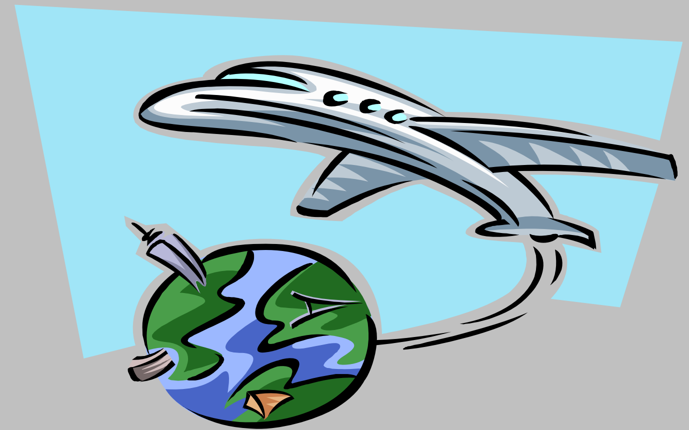
- 同步(isochronous)方式传送：同步传输提供了确定的带宽和间隔时间（latency)。它被用于时间严格并具有较强容错性的流数据传输，或者用于要求恒定的数据传送率的即时应用中。例如执行即时通话的网络电话应用时，使用同步传输模式是很好的选择。同步数据要求确定的带宽值和确定的最大传送次数。对于同步传送来说，即时的数据传递比完美的精度和数据的完整性更重要一些。

- 中断(interrupt)方式传送：中断方式传输主要用于定时查询设备是否有中断数据要传送。

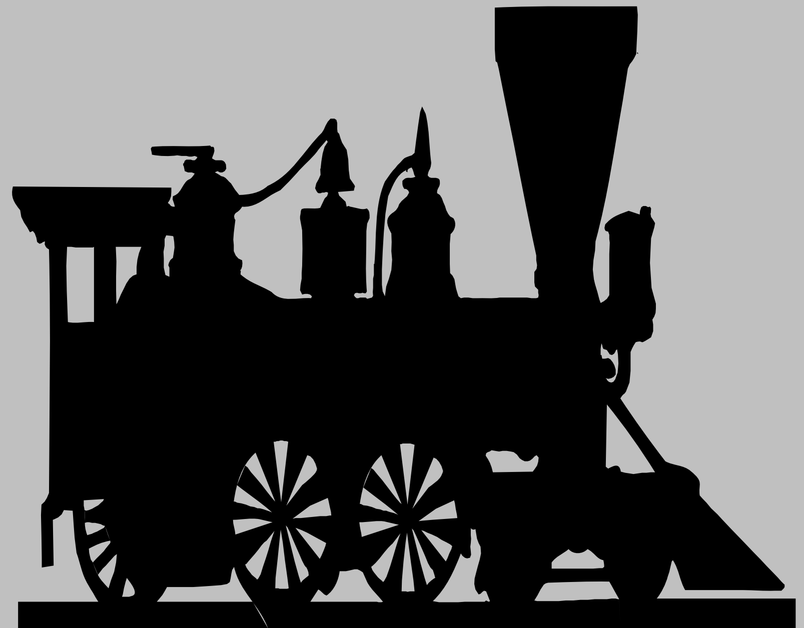
- 设备的端点模式器的结构决定了它的查询频率，从1到255ms之间。这种传输方式典型的应用在少量的分散的、不可预测数据的传输。键盘、操纵杆和鼠标就属于这一类型。

- ·大量(bulk)传送：主要应用在数据大量传送传送和接受数据上，同时又没有带宽和间隔时间要求的情况下，要求保证传输。打印机和扫描仪属于这种类型。这种类型的设备适合于传输非常慢和大量被延迟的传输，可以等到所有其它类型的数据的传送完成之后再传送和接收数据。

- 描述符的概念
- USB设备硬件中的数据结构称为描述符，可以被主机软件识别。每个描述符开始于一个两字节的头，头中指出该描述符的字节长度(包括头)和描述符类型。描述符的长度对于相同的描述符类型是固定的，即所有给定类型的描述符长度相同。在描述符头中保存明确的长度便于描述符将来的扩展。
- 设备描述符
- 每个设备都有一个唯一的设备描述符，它向主机软件标识该设备。主机使用GET_DESCRIPTOR控制事务直接从设备的0号端点读取该描述符。该描述符在美国微软公司（MS）所提供的设备开发说明书（DDK）中的定义如下：



- 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;

- 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;

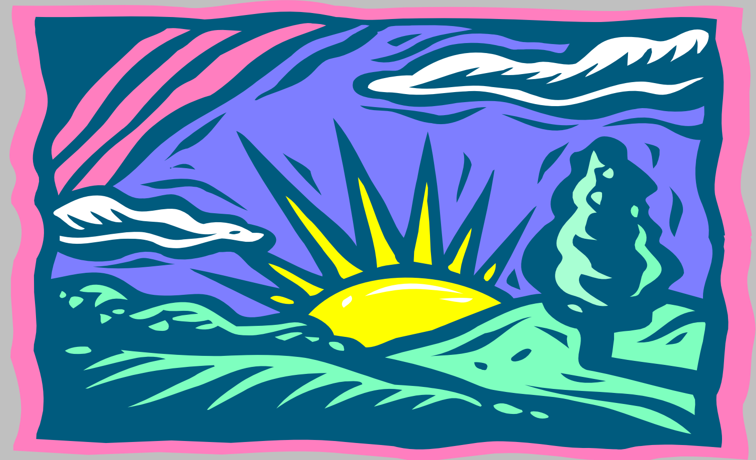
- USB请求使用总线驱动程序

- 与传统PC总线(如PCI总线)设备的驱动程序相比，USB设备驱动程序从不直接与硬件对话。相反，它仅靠创建URB(USB请求块)并把URB提交到总线驱动程序就可完成硬件操作。

- 可以把USBD.SYS看作是接受URB的实体，向USBD的调用被转化为带有主功能代码为IRP_MJ_INTERNAL_DEVICE_CONTROL的IRP。然后USBD再调度总线时间，发出URB中指定的操作。

• 初始化请求

- 为了创建一个URB，你首先应该为URB分配内存，然后调用初始化例程把URB结构中的各个域填入请求要求的内容，例如，当你为响应IRP_START_DEVICE请求而配置设备时，首要的任务就是读取该设备的设备描述符。
- USB_DEVICE_DESCRIPTOR dd;
- URB urb;
- UsbBuildGetDescriptorRequest(&urb,
- sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
- USB_DEVICE_DESCRIPTOR_TYPE,
- 0,
- 0,
- &dd,
- NULL,
- sizeof(dd),
- NULL);

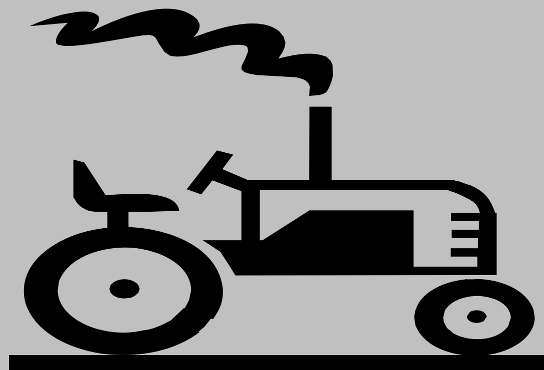


- 为了防止描述符的数据结构建立失败，我们可以在系统堆上为URB动态地分配内存：

- PURB urb = (PURB) ExAllocatePool(NonPagedPool, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST));
- if (!urb)
- return STATUS_INSUFFICIENT_RESOURCES; UsbBuildGetDescriptorRequest(urb, ...);
- ... ExFreePool(urb);

- # 发送URB
- 创建完URB后，你需要创建并发送一个内部I/O控制(IOCTL)请求到USBD驱动程序，USBD驱动程序位于驱动程序层次结构的低端。在大多数情况下，需要等待设备回应，可以使用下面辅助函数：

- NTSTATUS SendAwaitUrb(PDEVICE_OBJECT fdo, PURB urb)
- {
- PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
- KEVENT event;
- KeInitializeEvent(&event, NotificationEvent, FALSE);
- IO_STATUS_BLOCK iostatus;
- PIRPIrp=IoBuildDeviceIoControlRequest(IOCTL_INTERNAL_USB_SUBMIT_URB,
- pdx->LowerDeviceObject,
- NULL,
- 0,
- NULL,
- 0,
- TRUE,
- &event,
- &iostatus);



- PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
- stack->Parameters.Others.Argument1 = (PVOID) urb;
- NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
- if (status == STATUS_PENDING)
- { KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
- status = iostatus.Status;
- }
- return status;
- }

- 注意
- 需要强调的是驱动程序把URB包装成一个带有IRP_MJ_INTERNAL_DEVICE_CONTROL主功能码的普通IRP。为了使上层过滤器驱动程序可以发送自己的URB，每个USB设备驱动程序应有一个可以把IRP传递到下一层的派遣函数。

- **URB返回的状态**
- 当提交一个URB到USB总线驱动程序时，我们最终将收到一个描述该操作结果的NTSTATUS代码。其间，总线驱动程序使用另一组类型名为USBD_STATUS的状态代码。这些代码并不是NTSTATUS代码。
- 当USBD完成一个URB时，它就把URB的**UrbHeader.Status**域设置为某个USBD_STATUS值。DDK中的**URB_STATUS**宏可以简化这个值的存取：

- NTSTATUS status = SendAwaitUrb(fdo, &urb);
- USBD_STATUS ustatus = URB_STATUS(&urb);
- ……..

- 这样，我们演示了USB驱动开发中最简单的一个流程，也就是在当应用程序请求与硬件系统通信时候，在系统内部，USB驱动程序如何完成这个功能的。

- 第三章 关于**USB**设备的配置
- **USB**总线驱动程序自动检测新插入的**USB**设备。然后它读取设备内的设备描述符以查明插入的是何种设备，描述符中的厂商和产品标识以及其它描述符一同决定具体安装哪一个驱动程序。
- 配置管理器调用驱动程序的**AddDevice**函数。**AddDevice**做所有你已知的任务：创建设备对象，把设备对象连接到驱动程序堆栈上，等等。最后，配置管理器向驱动程序发送一个即插即用请求**IRP_MN_START_DEVICE**。

- 通过调用一个名为**StartDevice**的辅助函数并传递一些参数，这些参数描述了赋予设备的经过转换的和未经转换的I/O资源。我们不必再为USB驱动程序的I/O资源担心了，因为它们不用任何I/O资源。所以可以按下面写StartDevice的辅助函数：
- NTSTATUS StartDevice(PDEVICE_OBJECT fdo)
- {
- PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
- *<configure device>*
- return STATUS_SUCCESS;
- }

- StartDevice的执行过程大致如下，首先为设备选择一个配置。如果你的设备象大多数设备一样，应该仅有一种配置。选定了某个配置后，接着应该选择配置中的一个或多个接口。支持多接口的设备并不少见。选定了一个配置和一组接口后，应该向总线驱动程序发送配置选择URB。最后，总线驱动程序向设备发出命令使能选定的配置和接口。总线驱动程序负责创建管道和用于访问管道的句柄，管道提供功能驱动程序与选定接口端点之间的通信，它同时还创建配置句柄和接口句柄。我们可以从完成的URB中提取这些句柄并保存为以后使用。至此，设备的配置过程全部结束。

- 寻找句柄
- 配置选择URB成功完成后，我们应该把一些句柄保存下来供以后使用：
- URB成员**UrbSelectConfiguration.ConfigurationHandle**返回该配置句柄。
- USBD_INTERFACE_INFORMATION结构中的**InterfaceHandle**返回接口句柄。
- 每个USBD_PIPE_INFORMATION结构中都含有与每个端点对应的管道句柄**PipeHandle**。

- 例如，在设备扩展中保存了两个句柄：
- typedef struct _DEVICE_EXTENSION {
- ...
- USBD_CONFIGURATION_HANDLE hconfig;
- USBD_PIPE_HANDLE hpipe;
- }
- DEVICE_EXTENSION , *PDEVICE_EXTENSION;
- pdx->hconfig = selurb->UrbSelectConfiguration.ConfigurationHandle;
- pdx->hpipe = interfaces[0].Interface->Pipes[0].PipeHandle;
- ExFreePool(selurb);

- 关闭设备
- 当驱动程序接到一个IRP_MN_STOP_DEVICE请求时，应该把设备置成未配置状态(配置选择号0)，创建并提交一个含NULL配置指针的配置选择URB即可以达到这个目的：
- URB urb;
- UsbBuildSelectConfigurationRequest(
- &urb,
- sizeof(_URB_SELECT_CONFIGURATION),
- NULL);
- SendAwaitUrb(fdo, &urb);

- 设备关闭后，停止工作。

- 至此，USB驱动开发简单介绍就结束了，由于水平有限，理解有误之处难免，请大家指正！
- 感谢观看！
-
- 再见