USB 开发步骤

USB 开发步骤之标准篇

通用串行总线（Universal Serial Bus）是用于将适用 USB 的外围设备连接到主机的外部总线结构,其主要是用在中速和低速的外设。USB 是通过 PCI 总线和 PC 的内部系统数据线连接，实现数据的传送。USB 同时又是一种通信协议，他支持主系统(host)和 USB 的外围设备(device)之间的数据传送，在 USB 的网络协议中，每个 USB 的系统有且只有一个 host，因此，很多的朋友问我是否可以将两台 PC 的 USB 口通过 A-A 头连接起来，是否可以实现通信，这样是不行的，因为对于电脑主板上的 USB 设备，都是 host,如果连起来就是两个 host 的通信，这样一来的一个 USB 的系统有了两个的 host，与它的网络协议冲突。Anchorchip 出了一个可以直接连接的设备(好象是 AN2720SC)，实际上是一个由两个背靠背的 USB 的 device 组合起来的一块芯片，要卖80多个刀乐，太贵了，呵呵！

USB 的优点有以下几条：

USB 为所有的 USB 外设提供了单一的、易于操作的标准的连接类型。这样一来就简化了 USB 外设的设计，同时也简化了用户在判断哪个插头对应哪个插槽时的任务,实现了单一的数据通用接口。

USB 排除了各个设备象鼠标、调制解调器、键盘和打印机设备对去系统资源的需求，因而减少了硬件的复杂性和对端口的占用，整个的 USB 的系统只有一个端口和一个中断，节省了系统资源。

USB 支持热插拔(hot plug)，也就是说在不关 PC 的情况下可以安全的插上和断开 USB 设备，动态的加载驱动程序。其他普通的外围连接标准，如 SCSI 设备等必须在关掉主机的情况下才能增加或移走外围设备。

USB 支持 PNP。当插入 USB 设备的时候，计算机系统检测该外设并且通过自动的加载相关的驱动程序来对该设备进行配置，并使其正常工作。

USB 在设备供电方面提供了灵活性。USB 直接连接到 Hub 或者是连接到 Host 的设备可以通过 USB 电缆供电,也可以通过电池或者其它的电力设备来供电,或使用两种供电方式的组合.并且支持节约能源的挂机和唤醒模式。

USB 提供全速12Mbps 的速率和低速1.5Mbps 的速率来适应各种不同类型的外设。

针对不能处理突然发生的非连续传送的设备，如音频和视频设备，USB 可以保证其固定带宽。

为了适应各种不同类型外围设备的要求，USB 提供了四种不同的数据传送类型。

USB 使得多个外围设备可以跟主机通信。

USB 的目的:1,使用方便 2,可以提供实时的数据给 PC 3,端口的灵活扩展性

USB 标准可以在 www.usb.org/developer 中找到,并且你还可以在该站点找到另外的一个 USB 的测试工具:usbcomp.exe,它包含一个 usbcheck 的工具可以检测到设备是否一些 USB 的高层次的要求。同时它还有一个 usbcheck 的工具可以检测 HID(human interface device)的设备。而 Win98还有一个"Ignore hubs"(Memphis only)的检测窗口。在 W2K DDK 中包含的一个 USBView 的工具可以看出系统中所有的 USB 总线以及 USB 总线上的所有的设备。

USB 论坛(USB forum)的成员每年只需要支付$2500就可以获得一个 Vendor ID,其实,每个 Vendor ID 的零售价格只是$200,不过每个 USB 论坛的成员可以在关于 USB 的支持方面可以得到许多的好处。(对于俺们中国人来说,去弄一个什么 USB 成员是很浪费钱的)

USB 的设备类型(device class)

虽然 USB 设备都会表现 USB 的一些基本的特征。但是,USB 的设备还是可以分成多个不同类型,同类型的设备可以拥有一些共同的行为特征和工作协议,从而使设备的驱动程序的书写变得简单一些。下表中就给出一些基本的 USB 的设备类型分类。

设备类型(device class) 设备举例 类型常量(Class constant)

音频(audio) 扬声器 USB_DEVICE_CLASS_AUDIO

通信 MODEM USB_DECICE_CLASS_COMMUNICATIONS

HID 键盘 鼠标 USB_DEVICE_CLASS_HUMAN INTERFACE

显示 监视器 USB_DEVICE_CLASS_MONITOR

物理回应设备 动力回馈式游戏操纵杆 USB_DEVICE_CLASS_PHYSICAL_INTERFACE

电源 不间断电源供应 USB_DEVICE_CLASS_POWER

打印机 USB_DEVICE_CLASS_PRINTER

大量的存储器 硬盘 USB_DEVICE_CLASS_STORAGE

HUB USB_DEVICE_CLASS_HUB

USB 的基本特性

每一个设备(device)会有一个或者多个的逻辑连接点在里面,每个连接点叫 endpoint.每个 endpoint 有四种数据传送方式:控制(Control)方式传送;同步(isochronous)方式传送;中断(interrupt)方式传送;大量(bulk)传送.但是所有的 endpoint0都被用来传送配置和控制信息。

在 host 和设备的 endpoint 之间的连接叫作管道"pipe",endpoint0叫做缺省(default pipe)。

对于同样性质的一组的 endpoint 的组合叫做接口(interface),如果一个设备包含不止一个的接口就可以称之为复合设备(composite device)。

同样的道理,对于同样的类型的接口的组合可以称之为"配置"(configuration)。但是每次只能有一个配置是可用的,而一旦该配置激活,里面的接口和 endpoint 就都同时可以使用。

host 从设备发过来的描述字(descriptors)中来判断用的是哪个配置,哪个接口等等,而这些的描述字通常是在 endpoint0中传送。

Windows USB 驱动程序接口

系统中的 USB 的驱动程序完成许多的工作。

实际上对于一些 HID 的 USB 设备,象键盘,鼠标和游戏操纵杆之类的设备可以自动的被系统识别并且支持.而除此之外的设备就需要自己写一个驱动程序来完成硬件和软件之间的联系。在核心模式(kernel mode)下,驱动程序用 IOCTL 来组织和操作一些由其他部分发过来的要求和命令。而 IOCTL 又是通过 URB(USB request blocks)来实现数据的传送的。

在正式的介绍 USB 的驱动程序之前,先还是来看看 USB 的物理和逻辑结构。

传输方式

在 USB 的数据传送的方式下,有四种的传输方式:控制(Control)同步(isochronous)中断(interrupt)大量(bulk)。如果你是从硬件开始来设计整个的系统,你还要正确选择传送的方式,而作为一个驱动程序的书写者,就只需要弄清楚他是采用的什么工作方式就行了。

通常所有的传送方式下的主动权都在 PC 边,也就是 host 边。

#2

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。控制传输典型地用在主计算机和 USB 外设之间的端点(Endpoint)0之间的传输，但是指定供应商的控制传输可能用到其它的端点。

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。中断方式传送是单向的并且对于 host 来说只有输入的方式。

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。

USB 将其有效的带宽分成各个不同的桢(frame),每桢通常是1ms 时间长。每个设备每桢只能传送一个同步的传送包。在完成了系统的配置信息和连接之后,USB 的 host 就会对不同的传送点和传送方式做一个统筹安排,用来适应整个的 USB 的带宽。通常情况下,同步方式和中断方式的传送会占据整个带宽的90%,剩下的就安排给控制方式传送数据。

USB 的低层结构

USB 设备

USB 的设备可以接在 PC 上的任意的 USB 接口上。而使用 HUB 还可以扩展使更多的 USB 设备连接到系统中,USB 的 HUB 有一个上行的端口(到 host),有多个的下行端口(连接其它的设备),从而可以使整个的系统可以扩展的连接127个外设,其中 HUB 也酸外设。对于 USB 系统来说,USB 的 host 永远是 PC 边,所有的其他连接到 host 都称为设备,在设备与设备之间是无法实现直线通信的,只有通过 host 的管理与调节才能够实现数据的互相传送.在系统中,通常会有一个根 HUB,这个 HUB 一般有两个下行的端口。

一个 PC 可以拥有一个或多个的 USB host 控制器。一般有两种类型的控制器:UHCI(USB host 控制器接口),OHCI(开放的 host 控制器接口)。Windows 的 USB 类驱动程序对于每一种的控制器类型都有一种 miniclass 驱动程序来支持。

USB 的物理信号

USB 的电缆有四根线,两根传送的是5V 的电源,有一些直接和电源 HUB 相连的设备可以直接利用它来供电。另外的两根是数据线,数据线是单工的,在整个的一个系统中的数据速率是一定的,要么是高速,要么是低速,没有一个可以中间变速的设备来实现数据码流的变速.在这一点上,USB 和1394有明显的差别。

USB 的总线可以在不使用的时候被挂起,这样一来就可以节约能源。

在有些时候的总线还有可能挡机(stall),比如说象数据传送的时候突然被打断，这个时候通过 host 的重新配置可以实现总线的重新工作。

低层协议

USB 的物理协议规定了大多数的在总线上的数据格式，通常一个全速的数据桢可以最多有的1500bytes,而对于低速的桢最多有187bytes。

桢通常是用来分配带宽给不同的数据传送方式。同时由于桢结构的规律性，桢的这种特性也可以用来做同步信号来使用。

一个最小的 USB 的数据块叫做包(packet)，包包括同步信号，包标识（packet ID)，CRC 和传送的数据。

Packet ID 共有以下十种：token OUT IN SOF SETUP

data DATA0 DATA1

handshake ACK NAK STALL

special PRE

Transactions（数据交换）

一个 transaction 是在 host 和设备（device)之间的不连续相互数据交换，通常由 host 开始交换，交换的开始是由 Token 的包开始的，接下来是双方向上的数据包，在数据包传送完之后，就会由设备（device）返回一个握手（handshake)包。USB 系统通过 IN,OUT,和 SETUP 的包来指定 USB 地址和 endpoint（最多是128个，0通常被用来用做缺省的传送配置信息的),并且这些被指定的设备必须通过上面形式的包来回应这种形式的指定。每个 SETUP 的包包含8个 byte 的数据，数据用来指示传送的数据类型。对于 DATA 数据包来

说，设置两种类型的数据包是为了能够在传送数据的时候做到更加的精确。ACK handshake 的包用来指示数据传送的正确性，而 STALL handshake 则表示数据包在传送的过程中出了故障，并且请示 host 重新发数据或者清除这次传送。PRE 格式的包主要是用在在一个 USB 的系统中如果存在不同速率的设备的时候，将不同于总线速度的设备中就会回应一个 PRE 的包从而会忽略该设备。

各种不同类型的包的大小是不同的，DATA 的数据包最大是1023bytes.

Start of Frame(SOF)

SOF 是 host 用来指示 frame 的开头的。SOF 的包包括11个 bit 的桢序号，从0到0X7FF(i.e. USBD_ISO_START_FRAME_RANGE-1)，SOF 对于所有的高速设备来说是有效的。

Power

每个设备可以从总线上获得100mA 的电流，如果特殊的向系统申请，最多可以获得500mA 的电流，在挂机的状态下，电流只有500uA.

驱动程序的安装步骤

Windows 用设备描述字或者接口描述字来了解到底是什么样的设备被接入到系统。Windows 初始化的 Hardware ID 中有设备提供商的 ID 域（IDVendor,IDProduct,和 BCDDevice)。如果你没有向系统提供一个 INF 文件的话，系统就会自动选择提供一个兼容 ID（可能不是工作得很好，就像你买了一个 Rockswell 的 Modem，而你使用标准 Modem 的驱动程序，你的 Modem 可能会工作的有很多的毛病，也可能跑得飞快，电脑的事情，什么都可能发生，就像中国足球.....我在九四年就发誓不再为中国足球恼火，可是俺前不久还是骂了一下那个叫章鱼鳞的小伙子，怎么就.....好歹还是一孩子，就原谅一回吧，哎!）

USB 的新特性

共享性 一个物理设备可以使用许多不同的 pipe

实时性 可以实现和一个设备之间有效的实时通信

动态性 可以实现接口间的动态切换

联合性 不同的而又有相近的特性的接口可以联合起来，

多能性 各个不同的接口可以使用不同的供电模式

自动性 缺省的 pipe 的使用使基系统的建立和配置变得自动并且快速

以上几个方面只是简要的介绍了一下 USB 的标准的一些情况，介绍得非常非常之浅，还有象 USB 的 host 在系统中的唯一性和 device 的带宽分布，以及 hub，和 USB 的电气特性等等，以及网络分层结构等方面我就不赘述了，在标准里面有详细的叙述。鄙人仅致力于用中文给大家一个比较浅显的介绍，希望不会给大家不正确的引导。（诸位大虾倘要做 USB 设备，当阅读美利坚合众国之原版文章。切记，切记！不瞒大家，朕亦十分反感大不列颠国之文字，一曰，吾弟问朕，国人何以皆学洋文，朕曰寡人如何得知，料想倘念好洋文，就有机会去诳洋人钱财。）

本来想单独开一个 USB2.0的页，可是敲中文实在是太麻烦，再加上现在的 USB2.0只是一个 DEMO 的期间，USB2.0的器件更是没有，关于 USB2.0没有什么改变，所有的硬件不用改变，就可以跑 USB2.0,到时候400多 Mb 可以让你任何的器件都可以通过 USB 来玩了，不过，现在只是在 USB 的高速的 host 和 USB 的高速 HUB 间可以有这么高的速度，普通的外设就差了一些，不知道以后会不会变化！稍后我将给出 USB2.0的专题介绍

推荐文档资料：

USB 标准(http://www.usb.org)[在 USB 开发其间，推荐你将它设为 IE 的首页]

Quick and EZ Guide to USB [快速入门]

USB 的1.1版本升级部分

http://usbing.net/index/Article/ShowArticle.asp?ArticleID=327

说明：上面文章系转贴，原作者网站为：http://embuffalo.myetang.com/

#3

USB 开发步骤之硬件篇

现在的 USB 生产厂商很多很多，几乎所有的硬件厂商都有 USB 的产品。我了解的公司有 Intel, 国半, Cypress, AnchorChips 这几家，Intel 作为 USB 标准的制订者之一，又加上 Intel 的龙头老大的地位，现在的计算机主板上的 HOST 基本上采用的都是 INTEL 的芯片。而在外设的 USB 控制器方面，大家的性能都差不多。其中 CYPESS 公司的器件以便宜见长，在收购了 ANCHORCHIPS 公司之后，CYPRESS 基本上从低速到全速的器件都包涵到了，有一些器件的性能还超过 INTEL，当然价格要便宜得多（注：以上评价纯属个人意见，仅供参考）。就象 CPU 唯 INTEL 马首是瞻一样，很多的厂家都喜欢把自己的东西和 INTEL 的做比较，我个人认为要做的话，还是 INTEL 的要好一些!

USB 器件一般来说是有两种类型的，一种是 MCU 集成在芯片里面的，象上面提到的 Intel,Cypree,National semiconductors, Anchorchips 等等;另外的还有就是单独的一个芯片实现 USB 的 Engine 的功能，象 philips，PDIUSBD11,PDIUSBP11A,PDIUSBD12等，还有象 lucent 也出了两款器件，USS-820,USS-620,820 是普通的高速设备，具有8个双向的 endpoints,而620是具有 DMA 功能，使用起来比820占用系统资源要少得多！还有很多家，因为我没有接触过其它厂家的东西，所以没有发言权！呵呵！不过总的来说，在这个方面 philips 和 National Semiconductors 的东东要大众一些，开发的话应该上手快一些！不过 philips 的和51的接口搞的偶烦了好几天，比较的麻拐一些!

好了，既然把话题说开了，就接着说说 USB 的 HUB 的器件，主要的我没有做过，好象国内的 USB 的 HUB 还没有什么厂家开始做，即使有，也是非常少的，HUB 的器件我重点推荐 philips 的东西，可以选择 PDUSBH 系列产品，这一个系列的产品主要是在于用它设计 HUB 的测试方面有很好的功能，从而减少了开发的难度（这是一个米国朋友告诉偶的，E 文翻译过来，如有错误，敬请注意）。

对了，还有一个问题，起码有不少于十个人来问我，两台 PC 间如何通过 USB 口直接连接起来，再次重申一遍，由于 USB 的网络协议的关系，两台 PC 的 USB 都是 host，所以没有办法连，如果连的话，必须要使用 USB 的桥！这个桥的芯片有以下几种，一是 anchorchips 的 EZlink,还有 Prolific Technology Inc 的 PL2301，PL2302，这个，我只知道 anchorchips 的东东，要买好几十给美金，我想如果是家庭用的话，没有人会发神经病去卖这个鸟东西，反正两个网卡才几块银子。还有 USB 的 Audio 方面的器件，有 Dalas 和 philips 的东西可以选，当然还有其它的，反正都是巨大无比的公司，应该质量上没有特别的差别，只是在一些小的细节上有所不用，各位老大自己慢慢斟酌了。

最后重申：以上几个方面的建议纯属个人建议，不代表官方意见和公司意见，我没有从上面的推荐公司中获得一分钱的好处（NND，白做了一把广告）。

下面就以 INTEL 公司的8X930AX USB 芯片作一些介绍：

8X930AX USB 微处理器采用的是 MCS51作为它的控制 CPU，工作在12MHz 的工作频率，有256Kbytes 的存储空间，如果不够还能够在外围扩展。它有11个中断源，其中有三个分配给 USB 设备(device)。当然他也还有一些其它的特性，在此不作详细的介绍了。您可以到 http://www.intel.com/ 去获得详细的资料。它的主要缺点是价格上可能偏高，开发费用略高，但是开发难度要相对小一些!

如何给 USB 器件设计一个 BUFFER(Intel)

下面以 CYPRESS 公司的 CY7C6XX 系列产品作一些介绍：

CYPRESS 公司的芯片在低速的场合做得很好，比如国内所有的 USB 鼠标都是采用的 CYPRESS 公司的芯片(真的是便宜，不由得你不买他的东东)，但是在高速方面支持得不是很好，好象是在 CY36系列产品上存在着

速度快就 CRASH 的毛病，在收购了 ANCHORCHIPS 之后情况才有所改观。
在我这次更新（四月）主页的时候，我得知他们在很多方面都做了改进，他们的产品应该还是很有竞争力的。
[个人意见] 其实我觉得 USB 的硬件所有的厂家都差不多，差别只是在于他们采用的控制器的不同，除了各个公司的 CPU 不同（像 ST 就用他们的 ST8或者 ST16，Intel 当然就用他们的看家本领，而 ANCHORCHIPS 就只好用人家的51了）以外的东西基本一致，就只有一个 SIE(什么是 SIE，可以参考这篇文章)，高速的器件还会有一个 DMA 的控制器，使他的数据可以跑得很快。当然你还要看他们的开发包的费用了，具体的报价我没有比较过，有兴趣的可以比较一把，别忘了给我一份，呵呵！
我对于硬件最想说的一点是他的速度的算法，USB 的速度是跟 USB 其间内部提供的缓冲区的大小有关，还有一个就是 USB 标准中提到的每1ms 和每10ms（还有255ms 的时候）发一桢数据的区别，这是低速和全速的主要区别，而缓冲区的大小直接的影响到每桢传送的数据量的大小，这一点大家在算 USB 的速度的时候一定要注意。在做设计之前一定计算好自己所要求的带宽和芯片的速度匹不匹配，否则会引起不必要的麻烦（洒家开始的时候就被害惨了）。
下面是一个带宽的计算公式：
BW=EP*INT(BPF/(OH+EP))/125
其中：BW 表示带宽(Mbits/sec)，EP 表示 ENDPOINT 的大小（bytes)，BPF 表示每桢的字节数，OH 表示开销。

你可以很快的根据你的设计计算出你的 USB 器件的带宽。下面给出一个例子：
BPF 187 OH 46
ENDPOINT SIZE（BYES） MAX BW（Mbits/sec）
1 0.024
2 0.048
4 0.096
8 0.192
希望可以从这个例子中得到 USB 的带宽的计算方法。而对于四种不同的传输方式来说，主要的不同是体现在 BPF 和 OH 的不同上。具体可以参照 USB 的标准的第八章。还可以看一看这篇文章。
选好方案之后，到了具体的控制程序设计阶段，就只是一个 CPU 的程序控制问题，对于华夏发达的单片机水平来说，这是非常容易解决的，倘若我再重复讲一下单片机中断抑或讲一段什么例子程序的话，我怀疑会被人砍翻在地，因此寡人只好封刀引退。若有不明，请参考谁谁谁谁谁等高手之微处理器专著。
也有一些大佬们来问了一下一些关于这些控制方面的问题，很多是没有仔细的看文档资料，关于一些 USB 的芯片的特殊功能寄存器在说明书里面都有详细的说明，操作方法跟51的器件里面的 SFR 一模一样。
上面文章系转贴，原作者网站为：http://embuffalo.myetang.com/
#4
我这里重点的介绍如何写驱动程序，对于一些应用程序我就不做介绍了，因为我对于那些高层的东西写得很少。倘若再讲，有班门弄斧之嫌，呵呵！
作为 WIN98和 WIN2K 推荐的一项新技术来说，USB 的驱动程序和以往的直接跟硬件打交道的 WIN95的 VXD 的方式的驱动程序不同，它应该是 WDM 类型的。
USB 的 WDM 接口框图如下（这个图可以说是 USB 软件总体框图）
对于 HID 的设备，就可以采用上图左上边的结构，其它类的话采用右上的结构，其实右边的结构可以又细分成两层，一层是 Class Driver，一层是 Miniport Driver。而倒数第三行的 UHCD 和 OpenHCI 分别是由 INTEL 和 COMPAQ 两位老大定的一个和硬件有关的底层驱动程序标准，各位可以根据所需要的选择。
对于 USB 的驱动程序，大家还得去了解 WDM 驱动程序的写法，或者早些时候的 NT 驱动程序，其实 WDM 驱动

程序可以看做是NT驱动程序的一个update，只是增加了一些新的特性。

“写驱动程序是一个很漫长和繁琐的工作，在此之前，你最好要熟悉硬件，熟悉C/C++，还要用过DDK，会用一些调试程序，如SOFTICE和WINDBG之类。如果一切就绪，你就可以开始写驱动程序，工作的进程有时侯会取决于你的运气”。（这是一位留美的朋友对我说的，我写出来和大家共享）

下面是我从一个朋友那里得到的一篇文章的摘要：

NT驱动程序的分层结构

驱动程序是指管理某个外围设备的一段程序代码。NT采用更灵活的分层驱动方法，允许杂应用程序和硬件之间 存在几个驱动程序层次。分层机制允许NT更加广泛地定义驱动程序，包括文件系统、逻辑卷管理器和各种网络组件，各种物理设备驱动程序等等。

1、 设备驱动程序

这些是管理实际数据传输和控制特定类型的物理设备的操作的驱动程序，包括开始和完成I/O操作，处理中断和执行特定的设备要求的任何差错处理。

2、 中间驱动程序

NT允许在物理设备驱动程序上分层任意数目的中间驱动程序。这些中间层次提供扩展I/O系统的功能一种方法，而不必修改底层的驱动程序。这也是微软鼓吹的他们的系统灵活的一面！实际上我觉得这样反而牺牲了一些效率上的东西。

3、 文件系统驱动程序(FSD)

FSD是一类比较特殊的驱动程序,通常负责维护各种文件系统 所需要的磁盘结构。注意我们并不能使用DDK来开发FSD，而必须使用Microsoft的文件系统开发人员工具包。

一般比较少写中间过滤驱动程序，过滤驱动程序它截获和修改高层发送给类驱动程序的请求。这样就允许利用现有类驱动程序的功能，而不必从头开始写所有程序。NT内核模式对象在我们的实际开发过程中的对象是设备，由于端口驱动程序已经隐藏了硬件控制操作，因此我在这里不讲述跟硬件相关的部份。如果今后的开发对象不同，需要对硬件进行操作的时候，可能会对中断、DMA等有比较详细的了解，这些内容可以参考DDK帮助。

NT使用对象技术管理所有的数据，下面分别对一般驱动程序所涉及的一些对象做一介绍。不过在介绍这些对象之前，有必要先对驱动程序的结构做一介绍。

驱动程序结构

NT驱动程序和一般的DOS/Windows C语言程序不一样，它没有main()或者WinMain()函数入口。和DLL类似地，它向操作系统显露一个名称为DriverEntry()的函数，在启动驱动程序的时候，操作系统将调用这个入口。DriverEntry除了做一些必 要的设备初始化工作外，还初始化一些Dispatch例程入口。我们知道，NT应用和设备驱动程序打交道主要是通过CreateFile、 ReadFile、WriteFile 和DeviceIoControl等Win32 API来进行 的。这些API其实都对应着驱动程序的一些Dispatch例程。而驱动程序除了DriverEntry以外，主要就是由这些Dispatch例 程组成的。例如调用Win32 API CreateFile的时候，操作系统最终转化为对驱动程序IRP_MJ_CREATE功能代码所对应的 Dispatch例程的调用，如果驱动程序没有提供该例程， CreateFile调用就会失败。

NT中一些常用的功能代码和Win32 API的对象关系如下所示。

功能代码	说明
IRP_MJ_CREATE	打开设备CreateFile
IRP_MJ_CLEANUP	在关闭设备时，取消挂起的I/O请求
CloseHandle	

IRP_MJ_CLOSE 关闭设备 CloseHandle

IRP_MJ_READ 从设备获得数据 ReadFile

IRP_MJ_WRITE 向设备发送数据 WriteFile

IRP_MJ_DEVICE_CONTROL 对用户模式或内核模式客户程序可用的控制操作 DeviceIoControl

IRP_MJ_INTERNAL_DEVICE_CONTROL 只对内核模式客户程序可用的控制操作

IRP_MJ_QUERY_INFORMATION 得到文件的长度 GetFileLength

IRP_MJ_SET_INFORMATION 设置文件的长度 SetFileLength

IRP_MJ_FLUSH_BUFFERS 写输出缓冲区或丢弃输入缓冲区 FlushFileBuffers FlushConsoleInputBuffer PurgeComm

IRP_MJ_SHUTDOWN 系统关闭 InitialSystemShutdown

和上面的驱动程序支持的功能代码相对应，一般的驱动程序看起来就象下面的样子。

```
DriverEntry(…) // 驱动程序入口
{
…
DeviceObject->MajorFunction[IRP_MJ_CREATE] = XXDriverCreateClose; //XX 对应的是你自己给你的驱动程序的命名
DeviceObject->MajorFunction[IRP_MJ_CLOSE] = XXDriverCreateClose;
DeviceObject->MajorFunction[IRP_MJ_READ] = XXDriverReadWrite;
DeviceObject->MajorFunction[IRP_MJ_WRITE] = XXDriverReadWrite;
…
}
XXDriverCreateClose(…) // 对应 IRP_MJ_CREATE 和 IRP_MJ_CLOSE 的例程
{
//…….
}
XXDriverDeviceControl(…)// 对应 IRP_MJ_DEVICE_CONTROL 的例程
{
//…….
}
XXDriverReadWrite(…) // 对应 IRP_MJ_READ 和 IRP_MJ_WRITE 的例程
{
//…….
}
```

一个驱动程序并不需要支持所有的功能代码，比如如果一个驱动程序根本就不必要与用户模式客户程序交互，那么就不用支持 IRP_MJ_CREATE 和 IRP_MJ_CLOSE。又如设备不支持设备读写，就不用支持 IRP_MJ_READ 和 IRP_MJ_WRITE。 驱动程序对象是在操作系统启动驱动程序、在调用驱动程序 入口 DriverEntry 之前就已经创建好了的，并且作为 DriverEntry 函数的参数传递给驱动程序。如果驱动程序启动失败，操作 系统将删除该对象。该对象的数据结构如下。注意下表并不是完整地列出了 ntddk.h 中的 DEVICE_OBJECT 结构体的所有数 据项，这里仅列出了一般驱动程序可能使用到的数据项。

Driver 对象数据项　说明

PDEVICE_OBJECT　DeviceObject　　由本驱动程序创建的 Device 对象的链表

ULONG Flags　　PDRIVER_INITIALIZE DriverInit　　驱动程序初始化例程(一般较少用)

PDRIVER_STARTIO DriverStartIo　　StartIo 例程入口，一般该例程对低层设备驱动程序用得较多，高层驱动程序较少使用本例程。

PDRIVER_UNLOAD　DriverUnload　卸载驱动程序例程，如果想在控制面版的设备里停止该设备，应该提供本例程。

PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1]　　驱动程序的 Dispatch 例程表

在上面提到过驱动程序是管理同类型的所有设备，所以上面的 结构中 DeviceObject 指向的就不是单个的设备对象,而是一个对象链表,这个链表的维护在下面介绍 Device 对象时可以看到。Device 对象与 Device Extension 驱动程序在调用 IoCreateDevice 函数成功后就创建了一个 Device 对象。下面对 Device 对象几个比较重要的数据做一介绍。

Device 对象数据项　　说明

PVOID DeviceExtension　指向 Device Extension 结构的指针

PDRIVER_OBJECT DriverObject 指向这个设备的 Driver 对象的指针，IoCreateDevice 会 自动填写本数据。

ULONG Flags　指定这个设备的缓冲策略

PDEVICE_OBJECT NextDevice　指向属于这个驱动程序的下一个设备对象，依靠本数据来维护设备对象链表

CCHAR StackSize　发送到这个设备的 IRP 需要的 I/O 堆栈单元的最小数目，一般对分层驱动程序来说，本数据应该比其下层设备的大1

ULONG AlignmentRequirement 缓冲区要求的内存对齐，一般对分层驱动程序来说，本值应该 和其下层设备的对齐一致

Device 记录着设备的特徵和状态信息，对系统上的每个虚拟的、逻辑的和物理的设备都有一个 Device 对象。例如对一个硬盘驱动程序，对一个物理硬盘有一个名称为 Partition0的 Device 对象，对应整个物理磁盘，同时对硬盘的每个分区，也都有一个 Device 对象，它们的名称分别为 PartitionX(X 从1开始，每个分区对应一个数字)。

Device Extension 是连接到 Device 对象的一个很重要的数据结构，它的数据结构是由驱动程序设计者自己来确定的，在 调用 IoCreateDevice 的时候应该指定它的大小，Device Extension 其实是由操作系统在非份页内存池中为每个 Device 对象分配的一块内存。由于驱动程序必须是完全可重入的， 因此使用任何全局变量和静态变量都不是好的办法，一般来 说和设备有关的任何需要保持的信息都应该放到 Device Extension 里去。

设备的缓冲策略也必须提一下，这里的 Flag 的缓冲策略主要 决定设备读写(功能代码 IRP_MJ_READ 和 IRP_MJ_WRITE)时候的 缓冲策略，另外功能代码 IRP_MJ_DEVICE_CONTROL 时候的缓冲 策略是由 IOCTL 控制代码本身来决定的。两者不能混为一谈。 在下面我将专门用一节来讨论 I/O 的缓冲策略。

I/O 请求包(IRP)

在上面的结构里面已经出现了 IRP 了，在这里对它做一说明。 在 NT 中，几乎所有的 I/O 都是包驱动的，可以说驱动程序和操作系统其他部份都是通过 I/O 请求包来进行交互的。我们 来看看一个 I/O 请求的执行过程。

(1) 操作系统的 I/O 管理器从非分页内存分配一个 IRP，响应一个 I/O 请求。基于由客户指定的 I/O 函数，I/O 管理器将该 IRP 传递给合适的驱动程序的 Dispatch 例程。

(2) Dispatch 例程检查请求的参数是否有效，如果有效，驱动程序根据请求的内容进行一系列的操作。否

则设置错 误状态信息直接返回。
(3) 操作完成时，将数据(如果有)和状态信息存放到 IRP 中 并返回给 I/O 管理器。
(4) I/O 管理器对返回的 IRP 进行适当的处理后将最后状态和 数据(如果有)返回给用户。
一个 IRP 的主要数据项如下表所示。
IRP 包括一个 IRP 头和一个 IRP stack 的区域。由于 WDM 的模式下都是包驱动的，所里 IRP 可以说是一个非常重要的东东。还有那个该死的 URB(God damn URB!)[人一辈子真的很过瘾，有些人或有些事你明明不喜欢或做不来，可是有时侯你又不得不硬着头皮去做，就像一大堆球迷围着一堆狗屎般的中国足球，那班傻儿真是要钱不要脸，丢咱中国人的脸]
#5
IRP 主要数据项 说明
IO_STATUS_BLOCK IoStatus 存放 I/O 请求的状态
PVOID AssociatedIrp.SystemBuffer 如果设备执行缓冲 I/O，则为指向系统空间缓冲区的指针。 否则为 NULL
PMDL MdlAddress 如果设备执行直接 I/O，指向用户空间缓冲区的内存描述表的指针
PVOID UserBuffer I/O 缓冲区的用户空间地址
BOOLEAN Cancel 指示 IRP 已被取消
关于 AssociatedIrp.SystemBuffer、MdlAddress 和 UserBuffer 将在 下面的 I/O 缓冲区策略里面更详细地讨论。
NT 还有更多其他的对象，例如中断对象、Controller 对象、定时器对象等等，但在我们开发的驱动程序中并没有用到，因此在这里不做介绍。
I/O 缓冲策略
很明显的，驱动程序和客户应用程序经常需要进行数据交换，但我们知道驱动程序和客户应用程序可能不在同一个地址空间，因此操作系统必须解决两者之间的数据交换。这就就设计到设备的 I/O 缓冲策略。
读写请求的 I/O 缓冲策略
前面说到通过设置 Device 对象的 Flag 可以选择控制处理读写请求的 I/O 缓冲策略。下面对这些缓冲策略分别做一介绍。
1、缓冲 I/O(DO_BUFFERED_IO)
在读写请求的一开始，I/O 管理器检查用户缓冲区的可访问性，然后分配与调用者的缓冲区一样大的非分页池，并把它的地址放在 IRP 的 AssociatedIrp.SystemBuffer 域中。驱动程序就利用这个域来进行实际数据的传输。
对于 IRP_MJ_READ 读请求，I/O 管理器还把 IRP 的 UserBuffer 域设置 成调用者缓冲区的用户空间地址。当请求完成时，I/O 管理器利用 这个地址将数据从驱动程序的系统空间拷贝回调用者的缓冲区。对 于 IRP_MJ_WRITE 写请求，UserBuffer 被设置为 NULL，并把用户缓冲 区的数据拷贝到系统缓冲区中。
2、 直接 I/O(DO_DIRECT_IO)
I/O 管理器首先检查用户缓冲区的可访问性，并在物理内存中锁定它。然后它为该缓冲区创建一个内存描述表(MDL),并把 MDL 的地址 存放在 IRP 的 MdlAddress 域中。AssociatedIrp.SystemBuffer 和 UserBuffer 都被设置为 NULL。驱动程序可以调用函数 MmGetSystemAddressForMdl 得到用户缓冲区的系统空间地址，从而 进行数据操作。这个函数将调用者的缓冲区映射到非份页的地址空 间。驱动程序完成 I/O 请求后，系统自动从系统空间解除缓冲区的映射。
3、 这两种方法都不是
这种情况比较少用，因为这需要驱动程序自己来处理缓冲问题。 I/O 管理器仅把调用者缓冲区的用户空间地址放到 IRP 的 UserBuffer 域中。我们并不推荐这种方式。

IOCTL 缓冲区的缓冲策略

IOCTL 请求涉及来自调用者的输入缓冲区和返回到调用者的输出 缓冲区。为了理解 IOCTL 请求，我们先来看看 WIN32 API DeviceIoControl 函数的原型。

```
BOOL DeviceIoControl (
HANDLE hDevice, // 设备句柄
DWORD dwIoControlCode, // IOCTL 请求操作代码
LPVOID lpInBuffer, // 输入缓冲区地址
DWORD nInBufferSize, // 输入缓冲区大小
LPVOID lpOutBuffer, // 输出缓冲区地址
DWORD nOutBufferSize, // 输出缓冲区大小
LPDWORD lpBytesReturned, // 存放返回字节数的指针
LPOVERLAPPED lpOverlapped // 用于同步操作的 Overlapped 结构体指针
);
```

IOCTL 请求有四种缓冲策略，下面一一介绍。

1、 输入输出缓冲 I/O(METHOD_BUFFERED)

I/O 管理器首先分配一个非分页池，它足够大地存放调用者的输入或输出缓冲区(不管哪个更大)。非分页缓冲区的地址放在 IRP 的 AssociatedIrp.SystemBuffer 域中，然后把 IOCTL 的输入数据拷贝 到这个非份页缓冲区中，并把 IRP 的 UserBuffer 域设置成调用者输出缓冲区的用户空间地址。当驱动程序完成 IOCTL 请求时，I/O 管理器将这个非份页缓冲区中的数据拷贝到调用者的输出缓冲区。注意这里同一个非份页池同时用于输入和输出缓冲区，因此驱动程序在向缓冲区写东西之前应该把输入的所有数据读出来。

2、 直接输入缓冲输出 I/O(METHOD_IN_DIRECT)

I/O 管理器首先检查调用者输入缓冲区的可访问性，并在物理内存中将其锁定。然后为该输入缓冲区创建一个 MDL，并把指定该 MDL 的指针存放到 IRP 的 MdlAddress 域中。同时，I/O 管理器还在非份页池中分配一输出缓冲区，并把这个缓冲区的地址存放在 IRP 的 AssociatedIrp.SystemBuffer 域中，并把 IRP 的 UserBuffer 域设置成调用者输出缓冲区的用户空间地址。当驱动程序完成 IOCTL 请求时，I/O 管理器将非份页缓冲区中的数据拷贝到调用者的输出缓冲区。

3、 缓冲输入直接输出 I/O(METHOD_OUT_DIRECT)

I/O 管理器首先检查调用者输出缓冲区的可访问性，并在物理内存中将其锁定。然后为该输出缓冲区创建一个 MDL，并把指定该 MDL 的指针存放到 IRP 的 MdlAddress 域中。同时，I/O 管理器还在非份页池中分配一输入缓冲区，并把这个缓冲区的地址存放在 IRP 的 AssociatedIrp.SystemBuffer 域中， 同时把调用者用户输入缓冲区中的数据拷贝到系统缓冲区中，并把 IRP 的 UserBuffer 域设置为 NULL。

4、 上面三种方法都不是(METHOD_NEITHER)

I/O 管理器把调用者的输入缓冲区的地址放到 IRP 当前 I/O 堆栈单元的 Parameters.DeviceIoControl.TypeInputBuffer 域中，把输出缓冲 区的地址存放到 IRP 的 UserBuffer 域中。这两个地址都是用户空间地 址。

从上面的说明可以看出，在执行缓冲 I/O 时，I/O 管理器将在非份页池 中分配内存，如果调用者的缓冲区比较大时，分配的非份页池也将 比较大。非份页池是系统比较宝贵的资源，因此，如果调用者的缓 冲区比较大时，我们一般采用直接 I/O 的方式(例如磁盘读写请求等)， 这样不仅节省系统资源，另一方面由于省去了 I/O 管理器在系统缓冲 区和调用者缓冲区之间的数据拷贝，也提高了效率，这对存在大量 数据传送的驱动程序尤其明显。

可以注意到 DDK 中的 Samples 下，几乎所有的例程的读写请求都是直 接 I/O 的，而对于 IOCTL 请求则是缓冲区 I/O 的居多。

开始驱动程序设计

下面的文字是从 Microsoft 的 DDK 帮助中节选出来的，它让我们明 白在开始设计驱动程序应该注意些什么问题，这些都是具有普遍 意义的开发准则。应该支持哪些 I/O 请求在开始写任何代码之前， 应该首先确定我们的驱动程序应该处理哪些 IRP 例程。

如果你在设计一个设备驱动程序，你应该支持和其他相同类型 设备的 NT 驱动程序相同的 IRP_MJ_XXX 和 IOCTL 请求代码。

如果你是在设计一个中间层 NT 驱动程序，应该首先确认你下层 驱动程序所管理的设备，因为一个高层的驱动程序必须具有低层 驱动程序绝大多数 IRP_MJ_XXX 例程入口。高层驱动程序在接到 I/O 请求时，在确定自身 IRP 当前堆栈单元参数有效的前提下 ，设置好 IRP 中下一个低层驱动程序的堆栈单元，然后再调用 IoCallDriver 将请求传递给下层驱动程序处理。

一旦决定好了你的驱动程序应该处理哪些 IRP_MJ_XXX，就可以开始 确定驱动程序应该有多少个 Dispatch 例程。当然也可以考虑把某些 RP_MJ_XXX 处理的例程合并为同一例程处理。例如在 ChangerDisk 和 VDisk 里，对 IRP_MJ_CREATE 和 IRP_MJ_CLOSE 处理的例程就是同一函数。 对 IRP_MJ_READ 和 IRP_MJ_WRITE 处理的例程也是同一个函数。

应该有多少个 Device 对象？

一个驱动程序必须为它所管理的每个可能成为 I/O 请求的目标的物理和逻辑设备创建一个命名 Device 对象。一些低层的驱动程序还可能要创建一些不确定数目的 Device 对象。例如一个硬盘驱动程序必须为每一个物理硬盘创建一个 Device 对象，同时还必须为每个物理磁盘上的每个逻辑分区创建一个 Device 对象。一个高层驱动驱动程序必须为它所代表的虚拟设备创建一个 Device 对象，这样更高层的驱动程序才能连接它们的 Device 对象到这个驱动程序的 Device 对象。另外，一个高层驱动程序通常为它低层驱动 程序所创建的 Device 对象创建一系列的虚拟或逻辑 Device 对象。

尽管你可以分阶段来设计你的驱动程序，因此一个处在开发阶段的 驱动程序不必一开始就创建出所有它将要处理的所有 Device 对象。 但从一开始就确定好你最终要创建的所有 Device 对象将有助于设计者所要解决的任何同步问题。另外，确定所要创建的 Device 对象还有助于你定义 Device 对象的 Device Extension 的内容和数据结构。

开始驱动程序开发

驱动程序的开发是一个从粗到细逐步求精的过程。NT DDK 的 src\ 目录下有一个庞大的样板代码，几乎覆盖了所有类型的设备驱动程序、高层驱动程序和过滤器驱动程序。在开始开发你的驱动程序之前，你应该在这个样板库下面寻找是否有和你所要开发的类似类型的例程。例如我们所开发的驱动程序，虽然 DDK 对 USB 描述得不是很详细，我们还是可以在 src\storage\class 目录发现很多和 USB 设备有关的驱动程序。

下面我们来看开发驱动程序的基本步骤。

最简的驱动程序框架

1、 写一个 DriverEntry 例程，在里面调用 IoCreateDevice 创建 一个 Device 对象。

2、 写一个处理 IRP_MJ_CREATE 请求的 Dispatch 例程的基本框架 (参见 DDK Kernel-Mode Drivers 4.4.3 描述的一个 DispatchCreate 例程所要完成的最基本工作。当然写了 DispatchCreate 例程后， 要在 DriverEntry 例程为 IRP_MJ_CREATE 初始化例程入口)。如果驱动程序创建了多于一个 Device 对象，则必须为 IRP_MJ_CLOSE 请求写一个例程，该例程通常情况下可以和 DispatchCreate 共用一个例程，参见参见 DDK Kernel-Mode Drivers 4.4.3。

#6

、 编译连接你的驱动程序。

用下面的方法来测试你的驱动程序。

首先按上面介绍的方法安装好驱动程序。

其次我们还得为 NT 逻辑设备名称和目标 Device 对象名称之间建立 起符号连接，我们在前面已经知道 Device 对象名称对 WIN32用户模式 是不可见的,是不能直接通过 API 来访问的,WIN 32 API 只能访问 NT 逻辑设备名称。我们可以通过修改注册表来建立这两种名称之间的符 号连接。运行 REGEDT32.EXE 在 \HKEY_LOCAL_MACHINE\ System\ CurrentControlSet\Control\ Session Manager\ DOS Devices 下建立起符号连接(这种符号连接也可以在驱动程序里调用函数 IoCreateSymbolicLink 来创建)。

重新启动系统。

编写一个简单的测试程序调用 WIN32API CreateFile 函数以刚才你命名的 NT 逻辑设备名打开这个设备。如果打开成功，那么你也就成功地写出了一个最简单的驱动程序了。

支持更多的设备 I/O 请求

例如你的驱动程序可能需要对 IRP_MJ_READ 请求做出响应(完成后可用 WIN32 API ReadFile 函数进行测试)。如果你的驱动程序需要能够手工卸载，那么还必须对 IRP_MJ_CLOSE 做出响应。为你所需要处理 IRP_MJ_XXX 写好处理例程，并在 DriverEntry 里面初始化好这些例 程入口。

一个低层的驱动程序可能需要最起码一个 StartIo，ISR 和 DpcForIsr 例程，可能需要一个 SynchCritSection 例程，如果设备使用了 DMA， 那么可能还需要一个 AdapterControl 例程。关于这些例程，请参考 DDK 相应文档。

对于高层驱动程序可能需要一个或多个 IoCompletion 例程，最起码 完成检查 I/O 状态块然后调用 IoCompleteRequest 的工作。 如果需要，还要对 Device Extension 数据结构和内容做些修改。

驱动程序的书写过程的确是很烦人的，从你开始理解结构开始，你就像掉在一个泥潭里一样，无论你如何出拳，发觉总是稀泥一堆。即使你是计算机高手，可以写三千行源代码没犯一个错，一次写完，一次就编译通过（我的一个“同事”在面试的时候对我们老板说的，我想他说的对，他没犯一个错，而是犯了三十万零一个半错，不过，由不的你不信，俺朋友老板就信世间有这类高手，并供为上宾），你还得了解一些基础的硬件知识，你还要了解你的驱动程序的设备的种类，设备的硬件结构，一些特殊的寄存器，或许一些更基础的汇编程序你也的去跑一遍。还的去看什么微微有点软出的什么鸟 DDK（这玩意是最重要的），我看像敌敌畏（啊啊，给我一杯敌敌畏，让我不用写程序.....哈哈，我的水平直逼牛得滑了，好耍！好耍!)。然后你开始写了一大堆你自认为不比“葵花宝典”差的驱动程序，嘿嘿，你发觉整个程序就是编译不过去，就好像你花十块零五毛 RMB 买了本“葵花宝典”，终于下定决心按照书的首页要求的引刀什么的，可是你发觉费了九马二骡之力引完了刀，神功依旧未成，点解！你又得去学什么程序调试，去 Debug，俺们称其为捉虫，NN 的，TMD，虫没有捉到，脑袋可肿的大大的。什么 SOFTICE,WINDBG 之类，尽是一些系统杀手的角色，一不小心改错了一个内存地址，哼，我 CRASH 你的机器，你只好又装机，又调试，又死翘翘，你不见密西西比河不死心，又重来一遍，如此三番，惹得你无名火起，起身饮茶，又见隔壁部门的老板正和小蜜在讨论周末去哪里加班工作，不由的气不打一处出，大吼一声“呔，来将何人！洒家张翼德在此！”.....哎，人在老板下，哪能不干活，只有硬头再上.....无数次的失败，无数次的徒劳之后，终于让你的机器跑的欢极了，你不由的小哼一句“对面的小蜜看过来，这里的男孩很能干！”

驱动程序真的得看个人造化，若你有张无忌般奇遇，有韦小宝般艳福，有段舆般韧劲，（对了，还要有东方不败般的勇气）还有什么做不了的。哈哈！

#7

USB 设备的 Bulk 模式驱动程序设计

[文章导读]

本文介绍了 USB 设备 Bulk 模式驱动程序的设计,该设计使用 FIFO 消息队列、信号量机制和定时器中断机制

摘要 本文介绍了 USB 设备 Bulk 模式驱动程序的设计。该设计使用 FIFO 消息队列、信号量机制和定时器中

断机制，可在不同的操作系统中实现。文中所用到的程序体系结构对于实现不同USB设备进行Bulk模式通讯是通用的。

关键词 USB；Bulk模式；驱动设计

引言

通用串行总线（USB）是一种串行接口，具有自动配置能力和良好的兼容性，从而简化了计算机与外设的连接，被计算机外设硬件制造商广泛采纳。USB总线标准由1.1版升级到2.0版后，传输率由12Mbps增加到了480Mbps，更适宜于高速数据传输。USB设备支持打印机、扫描仪、数码相机等外设时，由于这些外设与主机间传输的数据量大，要求驱动程序采用Bulk模式进行高速数据传输。

USB设备驱动的整体结构

USB设备驱动的整体结构包括如下五个主要部分：USB应用程序接口、USB设备驱动函数、USB中断服务程序、USB回调接口程序、USB标准事件处理程序。

USB应用程序接口

USB应用程序接口主要功能是对USB驱动器进行软硬件初始化、打开端口、关闭端口、读端口、写端口和端口控制操作。当设备驱动器装入系统设备表时，I/O系统就调用该应用程序接口。

USB应用程序接口的一个例程所包含的函数：

- USB_init( ) -- USB端口驱动函数的安装和初始化、硬件配置。
- USB_open( ) -- 打开USB端口。
- USB_close( ) -- 关闭USB端口。
- USB_read( ) -- 对USB端口进行读操作。
- USB_write( ) -- 对USB端口进行写操作。
- USB_ioctl( ) -- 对USB设备进行I/O控制操作。

USB设备驱动函数

1、USB_init（）-- 初始化USB端口

USB_init函数初始化特定USB端口驱动器，进行软硬件配置。

初始化步骤如下：

（1） 将USB设备驱动器安装到I/O系统设备表中。

（2） 获取USB控制器使用的中断号。

（3） 获取各端口所需的系统资源，包括内存、信号量和消息队列。

（4） 初始化USB驱动器数据结构与USB端口状态寄存器。

（5） 启动USB标准事件处理程序。

（6） 启用控制端口0和USB中断最小支持集。

2、USB_open（）-- 打开USB端口

USB_open函数允许应用程序打开一个USB端口，选择DMA数据传输方式。

执行打开调用的典型步骤如下：

（1） 如果不是默认的控制端口0，要检查端口状态是否为"CONFIGURED"。

（2） 如果不允许多次打开，要确认端口还没有打开。

（3） 确认端口对当前选择的接口有效。

（4） 选择DMA传输，设置DMA控制器使用该端口的FIFO作为目的地址。

（5） 设置端口为打开状态。

3、USB_close（）-- 关闭USB端口

USB_close函数允许USB应用程序关闭一个端口，并关闭DMA通道。

执行关闭调用的典型步骤为：

（1） 关闭 DMA 通道，放弃端口对 DMA 控制器的使用。

（2） 设置端口为关闭状态。

4、USB_read（）-- 对 USB 端口进行读操作

USB_read 函数允许 USB 应用程序从输出端口或控制端口读取数据。

调用读函数的典型步骤为：

（1） 设置端口号、类型和方向。

（2） 确认端口处于打开状态。

（3） 设置端口信号量，避免多次调用。

（4） 调用 readDMA( ) 进行 DMA 写操作。函数内执行步骤为：

(a) 启动 DMA 从端口接收 FIFO 的读操作，将数据从端口传送到内存。

(b) 等待 DMA 完成中断。（DMA 中断向端口消息队列发送一条消息表示数据传输完成。）

(c) 重复进行（a)～（c）步骤直至接收到全部数据或 USB 主机结束传输。如果 USB 控制器检测到短包中断或零字节包，或者出现等待超时，则停止 DMA 传送，并转（d）步骤执行。

(d) 向 USB 主机发送一个零字节包，完成控制状态步骤。

（5） 释放端口信号量。

（6） 返回接收到的字节数或错误信息。

5、USB_write（）-- 对 USB 端口进行写操作

USB_write 函数允许 USB 应用程序写数据到输入端口或控制端口。

执行写调用的典型步骤为：

（1） 确认端口号、类型和方向。

（2） 确认端口处于打开状态。

（3） 获取端口信号量，避免多次调用。

（4） 调用 writeDMA( )进行 DMA 写操作。函数内执行步骤为：

(a) DMA 将内存数据传送到目标端口的 FIFO，并等待 DMA 完成中断。

(b) 若从接收 FIFO 接收到 USB 主机的一个零字节包，或者出现等待超时，则退出并返回传输的字节数。

(c) 重复执行（a)～（c）步骤直至全部数据传输完毕时，转（d）步骤执行。

(d) 强制向 USB 主机发送最后的零字节包或短包用来结束传输过程。

（5） 如果是控制端口，等待来自 USB 主机的零字节包，完成控制状态步骤。

（6） 释放端口信号量。

（7） 返回传输的字节数或错误信息。

6、USB_ioctl（）-- 对 USB 设备进行 I/O 控制操作

USB_ioctl 函数设置端口状态寄存器并执行 I/O 端口控制功能。

USB 应用程序根据控制对象不同分别调用提供应用程序控制 USB 接口的能力的 controlIoctl()和 epIoctl()函数。controlIoctl()函数执行 USB 控制器整体 I/O 出控制功能。epIoctl()函数执行个别 USB 端口的 I/O 控制功能。。

1) controlIoctl( ) -- 控制器控制函数

controlIoctl()函数对 USB 控制器进行控制操作。执行 I/O 功能之前获取 USB 控制器信号量，避免多次调用影响正在传输数据的端口。完成 I/O 操作后释放信号量。

USB 控制器应支持的控制功能包括：

• 支持远程唤醒功能。

• 设置 USB 端口进入/退出挂起状态。

• 复位 USB 端口。

• 设置 USB 控制器消息队列等待超时。

• 为 DMA 选择端口 FIFO。

• 允许/禁止可选中断。

• 读取帧时间戳起始位。

• 进行枚举测试。

• 返回接口、备用接口和当前 USB 端口配置状态。

2) epIoctl() -- 端口控制函数

epIoctl()函数对端口进行控制操作。执行 I/O 功能之前获取 USB 控制器信号量，避免多次调用影响正在传输数据的端口。完成 I/O 操作后释放信号量。

端口应支持的控制功能包括：

（1） 获得 USB 端口状态。

（2） 设置 USB 端口进入/退出阻塞状态。

（3） 设置 USB 端口消息队列等待超时。

USB 中断服务程序

USB 控制器产生单一中断，多个端口共享。每个端口产生 ACK、NACK/ERROR 中断。输出端口产生接收零字节包或短包中断。控制端口0接收设置包时产生中断。USB 控制器产生 USB 事件中断，如帧起始（SOF）、挂起、恢复和复位。

USB 中断服务程序执行下列步骤：

（1） 识别发生了 USB 中断的类型。

（2） 清除中断产生的条件。

（3） 读 USB 状态寄存器，获取当前配置、接口或帧起始时间戳状态信息。

（4） 向 USB 控制器消息队列或回调函数的接收消息队列发送一条消息。

USB 标准事件处理程序

USB 驱动器初始化后，启动 USB 标准事件处理程序负责处理枚举过程和异步 USB 事件。

事件处理程序使用控制端口0，直到完成枚举过程。当 USB 应用程序处于非活动状态时，除控制端口0以外端口均不可访问。事件处理程序在端口0上执行控制操作，响应 USB 标准请求，并负责通知 USB 应用程序枚举完成和接口活动状态，USB 事件通过回调接口传递到 USB 外设应用程序。当对 USB 端口枚举操作完成，USB 应用程序就可打开并使用 USB 端口。

处理一个 USB 任务的执行过程为：

（1） 读取 USB 控制器消息队列。

（2） 如果接收到设置包，则调用标准请求处理函数。

（3） 如果接收到事件，则调用 USB 事件处理函数。

（4） 确定当前状态和有效配置/接口。

（5） 更新 USB 控制器和端口数据结构。

（6） 重复（1）～（5）步骤。

USB 回调接口程序

回调应用程序接口是向应用程序提供反馈信息的一种接口，包括向应用程序通知 USB 事件的消息，如复位、配置改变、接口改变、挂起、恢复和帧起始。使用 USB 的应用程序要以下列方式回应这些消息：

（1） 复位 关闭端口，等待枚举测试。

（2） 配置改变 关闭端口，按新配置打开端口。

（3） 接口改变 关闭端口，从新接口打开端口。

（4） 挂起 进入低功耗模式。

（5） 恢复 退出低功耗模式。

（6） 帧起始 执行应用程序规定的处理。

总结

本文提供了进行USB端口Bulk模式驱动程序设计的过程，实现在USB接口设备与Host主机之间进行高速数据传输，对于嵌入式环境以及windows多线程环境下的USB设备的高速数据传输同样适用。

#8

再发一个RS485主从式多机通讯协议,

一、数据传输协议

此协议定义了一个控制器能认识使用的消息结构,而不管它们是经过何种网络进行通信的。它描述了一控制器请求访问其它设备的过程，如何回应来自其它设备的请求，以及怎样侦测错误并记录。它制定了消息域格局和内容的公共格式。

此协议决定了每个控制器须要知道它们的设备地址，识别按地址发来的消息，决定要产生何种行动。如果需要回应，控制器将生成反馈信息按本协议发出。

1、数据在网络上转输

控制器通信使用主—从技术，即仅一设备（主设备）能初始化传输（查询）。其它设备（从设备）根据主设备查询提供的数据作出相应反应。

主设备可单独和从设备通信，也能以广播方式和所有从设备通信。如果单独通信，从设备返回一消息作为回应，如果是以广播方式查询的，则从设备不作任何回应。协议建立了主设备查询的格式：设备（或广播）地址、功能代码、所有要发送的数据、一错误检测域。

从设备回应消息也由协议构成，包括确认要行动的域、任何要返回的数据、和一错误检测域。如果在消息接收过程中发生一错误(无相应的功能码)，或从设备不能执行其命令，从设备将建立一错误消息并把它作为回应发送出去。

2、在对等类型网络上转输

在对等网络上，控制器使用对等技术通信，故任何控制都能初始和其它控制器的通信。这样在单独的通信过程中，控制器既可作为主设备也可作为从设备。

在消息位，本协议仍提供了主—从原则，尽管网络通信方法是“对等”。如果一控制器发送一消息，它只是作为主设备，并期望从设备得到回应。同样，当控制器接收到一消息，它将建立一从设备回应格式并返回给发送的控制器。

3、查询—回应周期

（1）查询

查询消息中的功能代码告之被选中的从设备要执行何种功能。数据段包含了从设备要执行功能的任何附加信息。错误检测域为从设备提供了一种验证消息内容是否正确的方法。

（2）回应

如果从设备产生一正常的回应，在回应消息中的功能代码是在查询消息中的功能代码的回应。数据段包括了从设备收集的数据。如果有错误发生，功能代码将被修改以用于指出回应消息是错误的，同时数据段包含了描述此错误信息的代码。错误检测域允许主设备确认消息内容是否可用。

二、传输方式

控制器能设置传输模式为RS485串行传输，通信参数为9600，n，8,1。在配置每个控制器的时候，在一个网络上的所有设备都必须选择相同的串口参数。

地址 功能代码 数据数量 数据1 ...…. 数据n CRC字节

每个字节的位

- 1个起始位
- 8个数据位，最小的有效位先发送
- 1个停止位

错误检测域

- CRC(循环冗余码校验)

三、消息帧

1．帧格式

传输设备将消息转为有起点和终点的帧，这就允许接收的设备在消息起始处开始工作，读地址分配信息，判断哪一个设备被选中（广播方式则传给所有设备），判知何时信息已完成。错误消息也能侦测到并能返回结果。

消息发送至少要以10ms 时间的停顿间隔开始。传输的第一个域是设备地址。网络设备不断侦测网络总线，包括停顿间隔时间内。当第一个域（地址域）接收到，每个设备都进行解码以判断是否发往自己的。在最后一个传输字符之后，一个至少10ms 时间的停顿标定了消息的结束。一个新的消息可在此停顿后开始。

整个消息帧必须作为一连续的流转输。如果在帧完成之前有超过5ms 时间的停顿时间，接收设备将刷新不完整的消息并假定下一字节是一个新消息的地酚颉M　　兀　绻　桓鲂孪　⑹谛∮?ms 的时间内接着前个消息开始，接收的设备将认为它是前一消息的延续。这将导致一个错误，因为在最后的 CRC 域的值不可能是正确的。一典型的消息帧如下所示：

起始间隔 设备地址 功能代码 数据数量及数据 CRC 校验 结束

2、地址域

消息帧的地址域包含一个字符8Bit。可能的从设备地址是0...247 (十进制)。单个设备的地址范围是1...247。主设备通过将要联络的从设备的地址放入消息中的地址域来选通从设备。当从设备发送回应消息时，也把自己的地址放入回应的地址域中，以便主设备知道是哪一个设备作出回应。

地址0是用作广播地址，以使所有的从设备都能认识。

3、如何处理功能域

消息帧中的功能代码域包含了一个字符8Bits。可能的代码范围是十进制的1...255。当然，有些代码是适用于所有控制器，有此是应用于某种控制器，还有些保留以备后用。

当消息从主设备发往从设备时，功能代码域将告之从设备需要执行哪些行为。例如去读取当前检测参量的值或开关状态，读从设备的诊断状态，允许调入、记录、校验在从设备中的程序等。

当从设备回应时，它使用功能代码域来指示是正常回应(无误)还是有某种错误发生（称作异议回应）。对正常回应，从设备仅回应相应的功能代码。对异议回应，从设备返回一等同于正常代码的代码，但功能代码的最高位为逻辑1。

例如：一从主设备发往从设备的消息要求读一组保持寄存器，将产生如下功能代码：

0 0 0 0 0 0 1 1 （十六进制03H）

对正常回应，从设备仅回应同样的功能代码。对异议回应，它返回：

1 0 0 0 0 0 1 1 （十六进制83H）

除功能代码因异议错误作了修改外，从设备将一独特的代码放到回应消息的数据域中，这能告诉主设备发生了什么错误。

主设备应对程序得到异议的回应后，典型的处理过程是重发消息，或者诊断发给从设备的消息并报告给操作员。

4、数据域

从主设备发给从设备消息的数据域包含附加的信息：从设备用于进行执行由功能代码所定义的行为所必须

的数据。

如果没有错误发生，从设备返回的数据域包含请求的数据。如果有错误发生，此域包含一异议代码，主设备应用程序可以用来判断采取下一步行动。

在某种消息中数据域可以是0长度。例如，主设备要求从设备回应通信事件记录，从设备回应不需任何附加的信息。

数据域最长为70字节。

5、错误检测域

错误检测域包含一字节8Bits。错误检测域的内容是通过对消息内容进行循环冗长检测方法得出的。CRC 域附加在消息的最后，故 CRC 字节是发送消息的最后一个字节。

四、错误检测方法

1、超时检测

用户要给主设备配置一预先定义的超时时间间隔，这个时间间隔要足够长，以使任何从设备都能作为正常反应。如果从设备检测到一传输错误，消息将不会接收，也不会向主设备作出回应。这样超时事件将触发主设备来处理错误。发往不存在的从设备的地址也会产生超时。

2、CRC 检测

CRC 域是一个字节，检测了整个消息的内容。它由传输设备计算后加入到消息中。接收设备重新计算收到消息的 CRC，并与接收到的 CRC 域中的值比较，如果两值不同，则有误，从设备对本消息不作回应。

通讯网络只设有一个主机，所有通信都由他发起。网络可支持254个之多的远程从属控制器，但实际所支持的从机数要由所用通信设备决定。