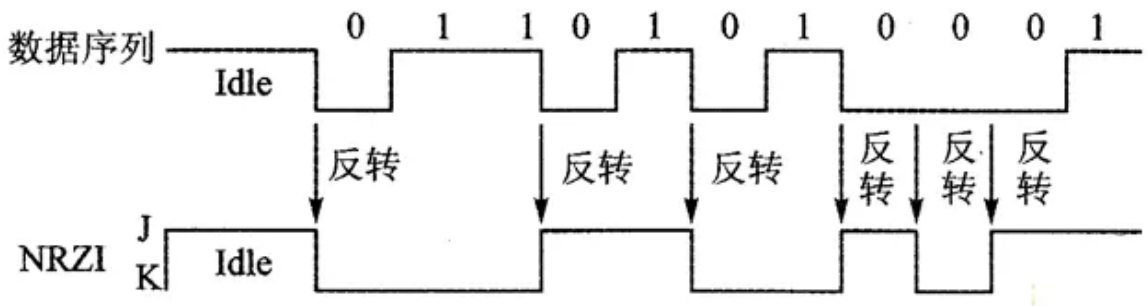
一、USB 整体架构

1.1 概念

USB 的全称是 Universal Serial Bus，通用串行总线。它的出现主要是为了简化个人计算机与外围设备的连接，增加易用性。USB支持热插拔，并且是即插即用的，另外，它还具有很强的可扩展性，传输速度也很快，这些特性使支持 USB 接口的电子设备更易用、更大众化。

1.2 NRZI 编码

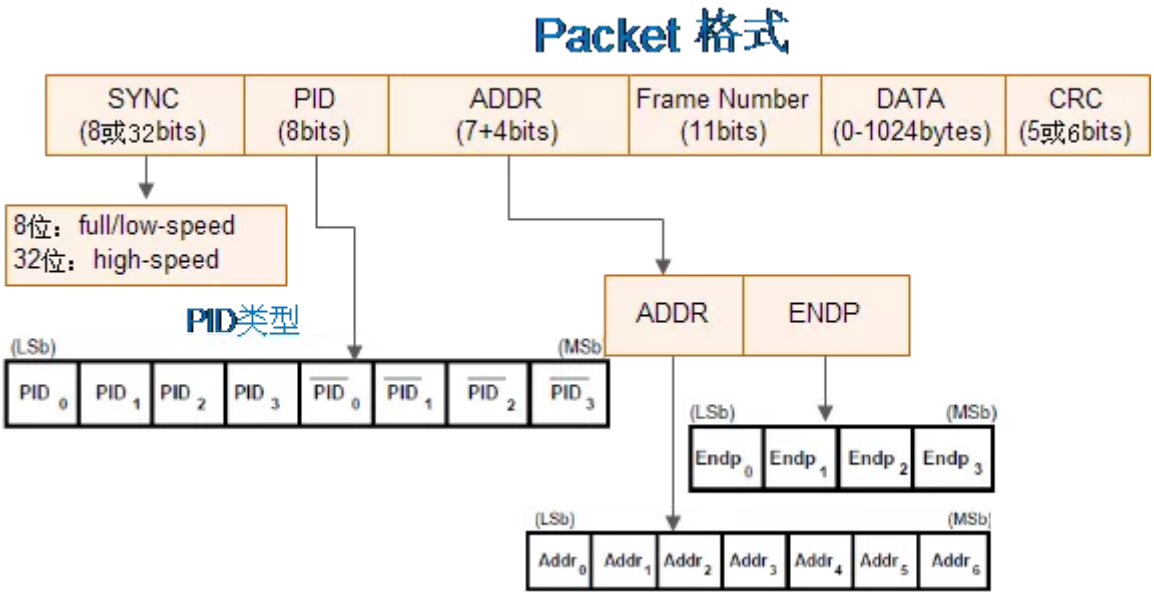


USB采用差分信号传输，使用的是如上图所示的 NRZI 编码方式：

- **数据为0时，电平翻转；**
- **数据为1时，电平不翻转。**
- **如果出现6个连续的数据1，则插入一个数据0，强制电平翻转，以便时钟同步。**

上面的一条线表示的是原始数据序列，下面的一条线表示的是经过 NRZI 编码后的数据序列。

1.3 包(packet)格式



USB总线上的传输数据是以包为基本单位的，包格式如上图所示。根据PID的不同，USB协议中规定的包类型有令牌包、数据包、握手包和特殊包等。

USB芯片（硬件）会完成CRC校验、位填充、PID识别、数据包切换、握手等协议处理。

1.4 数据传输规范和约定

- USB 传输是主从模式，主机负责发起数据传输过程，从机负责应答。
- USB 传输使用小端结构(Little-Endian)，一个字节在 USB 总线上的传输先后顺序为：b0 b1 b2 ... b7（与 I2C 相反，I2C是大端结构)。

> 关于大小端的概念请参考扩展部分

- 数据传输方向均以主机为参考

  比如：

  IN令牌包 用来通知设备返回一个数据包

  数据包的传输方向：主机←从机(IN)

  OUT令牌包 用来通知设备将要输出一个数据包

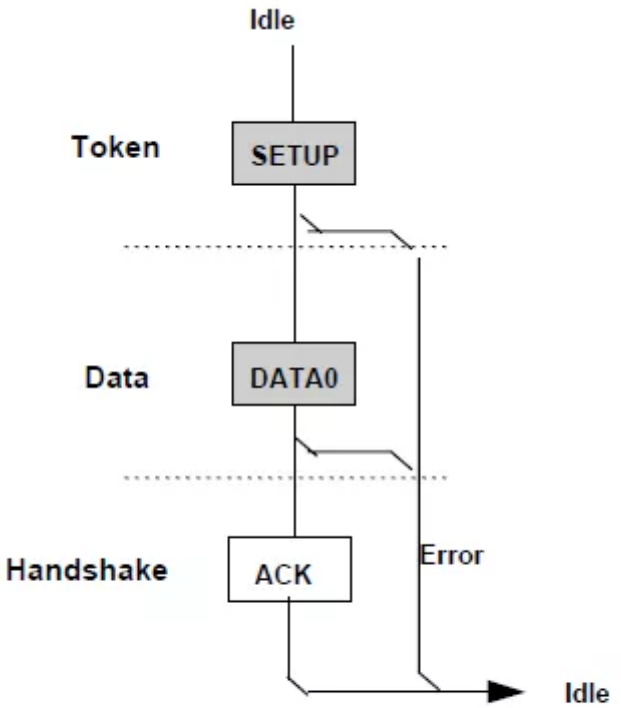
  数据包的传输方向：主机→从机(OUT)

1.5 四种传输模式

针对不同的数据传输场景，USB 分为四种数据传输模式，这四种传输模式分别由不同的包(packet)组成，并且有不同的数据处理策略。每种数据传输模式的流程示意图以及应用场景如下：

1.5.1 控制传输 Control Transfers

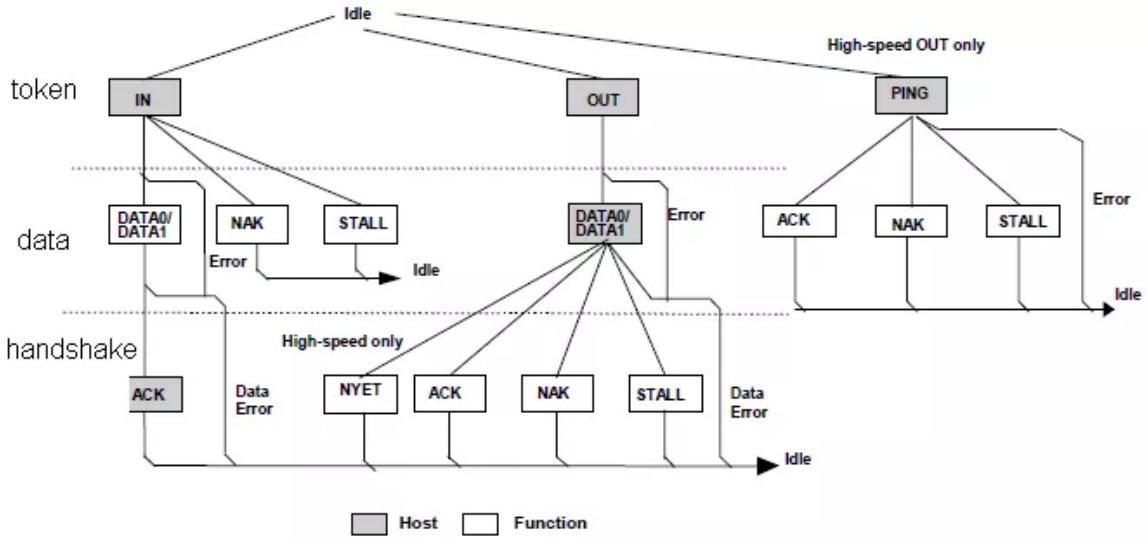
用于枚举过程，要保证数据传输过程的数据完整性。



Control SETUP Transaction

1.5.2 批量传输 Bulk Transfers

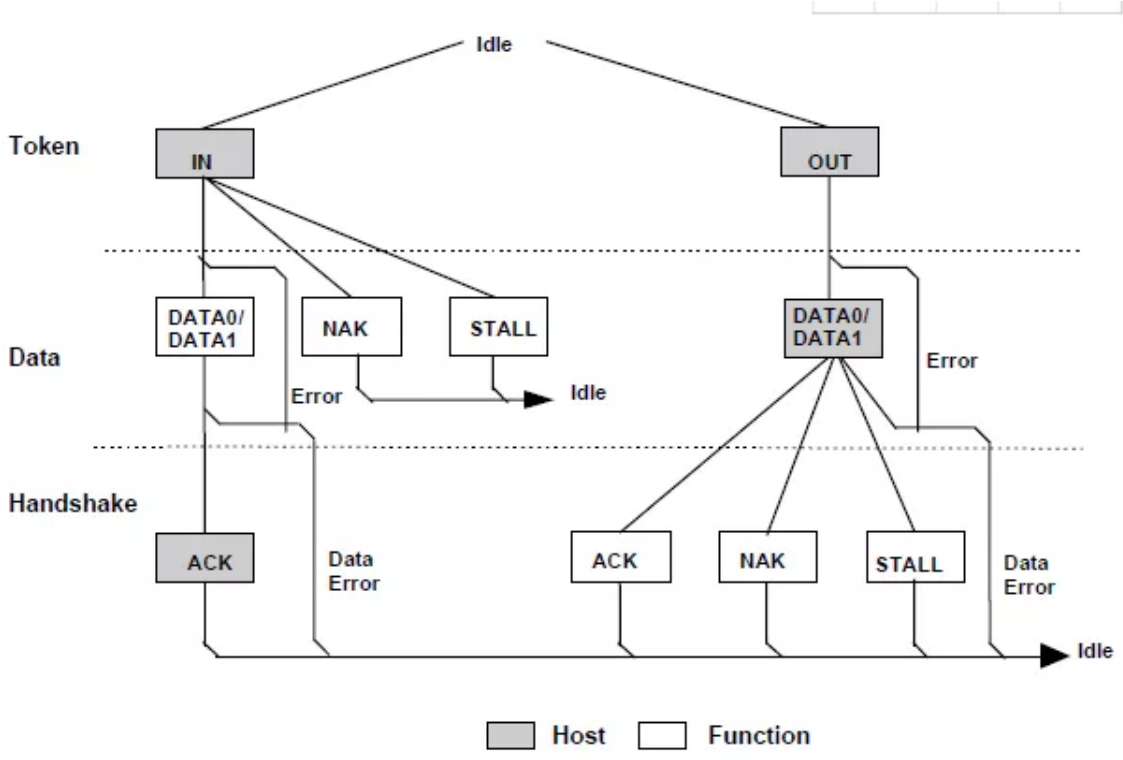
用于数据量大、对实时性要求不高的场合，如U盘。



Bulk Transaction Format

1.5.3 中断传输 Interrupt Transfers

用于数据量小的场合，保证查询频率，如鼠标、键盘。

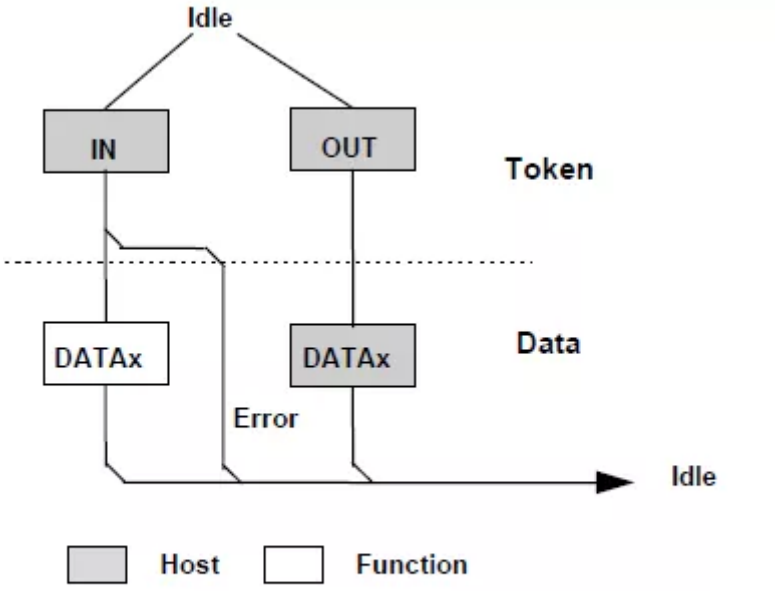


Interrupt Transaction Format

1.5.4 同步传输(等时传输) Isochronous Transfers

用于数据量大、同时对实时性要求较高的场合，如音视频。

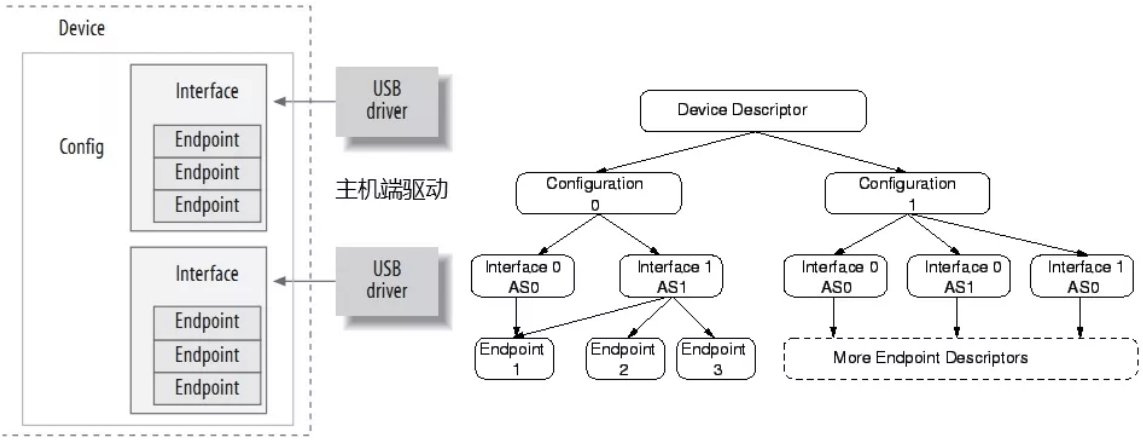
不保证数据完整性，没有 ACK/NAK 应答包，不进行数据重传。



Isochronous Transaction Format

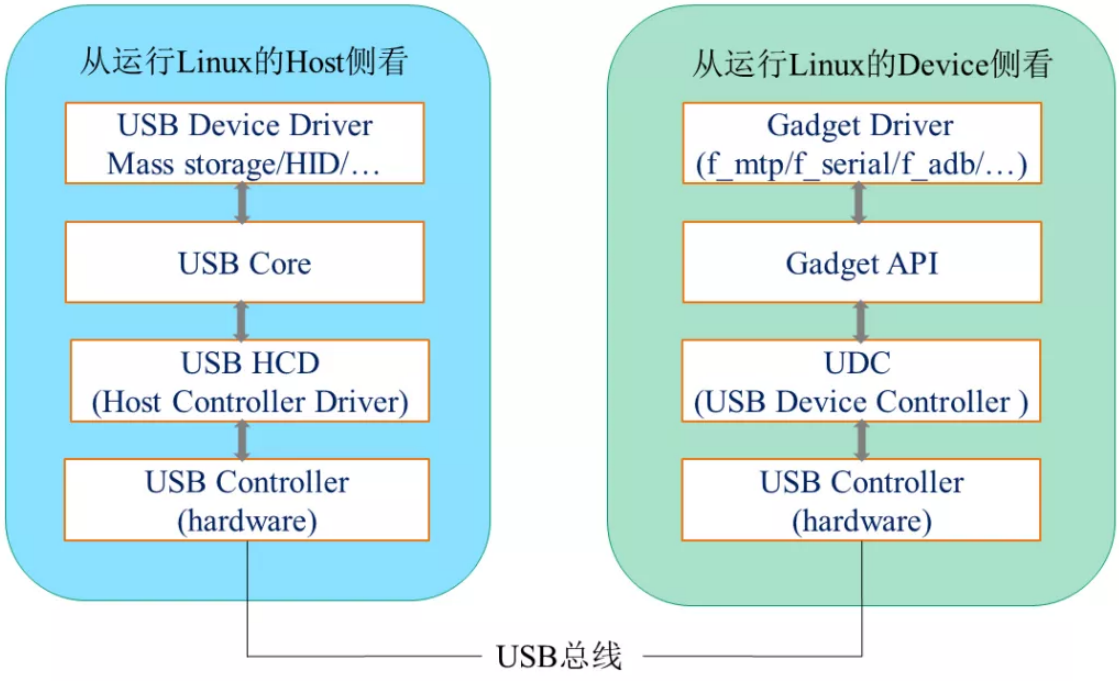
1.6 USB 设备结构及描述符

USB设备的构成



- 一个 USB 设备通常有一个或多个配置，但在同一时刻只能有一个配置；
- 一个配置通常有一个或多个接口；
- 一个接口通常有一个或多个端点；
- 驱动是绑定到 USB 接口上的，而不是整个 USB 设备。
- 枚举过程中，device 将各种描述符返回给 host。

1.7 Linux USB 驱动总体结构



从Host侧看，在Linux驱动中，处于USB驱动最底层的是USB主机控制器硬件，在其上运行的是USB主机控制器驱动，在主机控制器上的为USB核心层，再上层为USB设备驱动层（插入主机上的U盘、鼠标、USB转串口等设备驱动）。主机控制器驱动负责识别和控制插入其中的USB设备，USB设备驱动控制USB设备如何与主机通信，USB Core则负责USB驱动管理和协议处理的主要工作。

从Device侧看，UDC驱动程序直接访问硬件，控制USB设备和主机间的底层通信。Gadget API是UDC驱动程序回调函数的包装。Gadget Driver具体控制USB设备功能的实现。

1.8 USB 描述符

对应上述 USB 设备的构成，USB 采用描述符来描述 USB 设备的属性，在 USB 协议的第九章(chaper 9)中，有对 USB描 述符的详细说明，在 Linux 驱动的以下文件中，定义了 USB 描述符的结构体，文件名直接命名为ch9.h。

结构体在 include\linux\usb\ch9.h 中定义和具体含义：

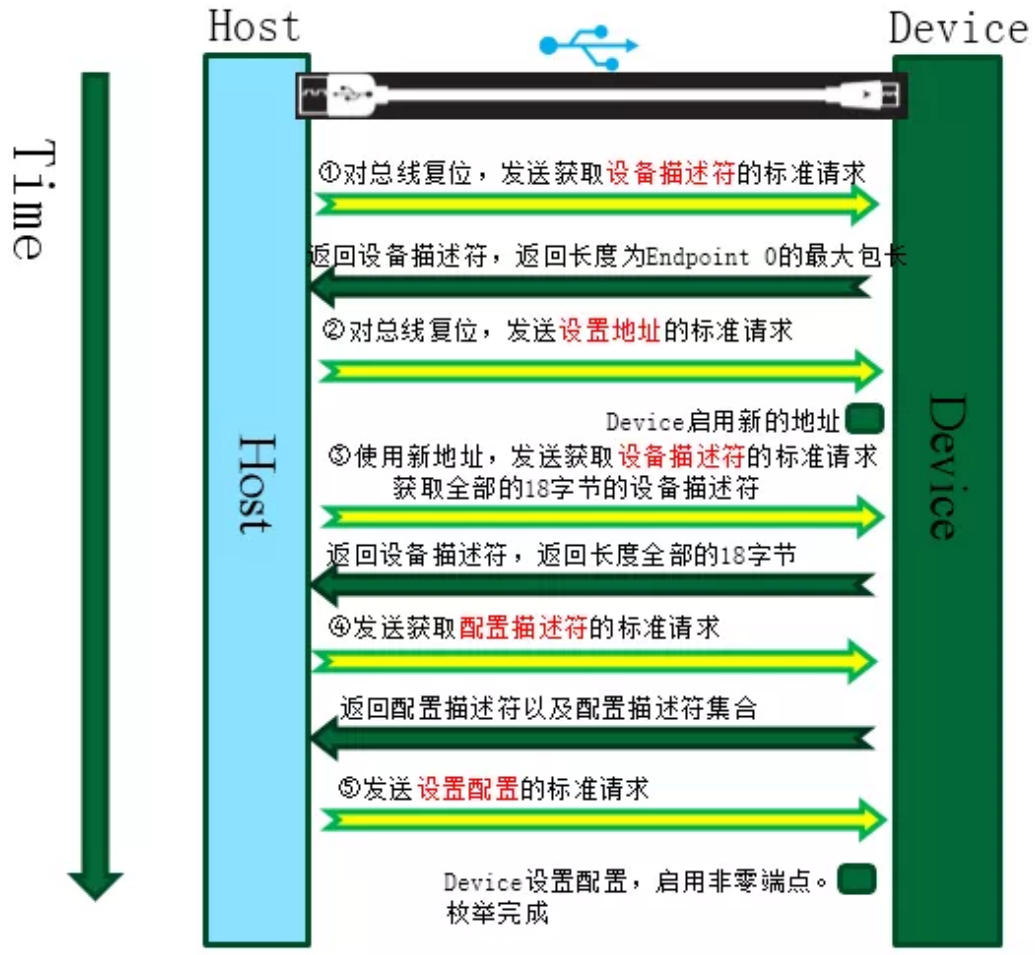
```
//设备描述符结构体
/* USB_DT_DEVICE: Device descriptor */
struct usb_device_descriptor {
__u8 bLength; //该描述符结构体大小（18字节）
__u8 bDescriptorType; //描述符类型（本结构体中固定为0x01)
__le16 bcdUSB; //USB 版本号
__u8 bDeviceClass; //设备类代码（由USB官方分配）
__u8 bDeviceSubClass; //子类代码（由USB官方分配）
__u8 bDeviceProtocol; //设备协议代码（由USB官方分配）
__u8 bMaxPacketSize0; //端点0的最大包大小（有效大小为8,16,32,64）
__le16 idVendor; //生产厂商编号（由USB官方分配）
__le16 idProduct; //产品编号（制造厂商分配）
__le16 bcdDevice; //设备出厂编号
__u8 iManufacturer; //设备厂商字符串索引
__u8 iProduct; //产品描述字符串索引
__u8 iSerialNumber; //设备序列号字符串索引
__u8 bNumConfigurations; //当前速度下能支持的配置数量
} __attribute__ ((packed));


// 配置描述符结构体
struct usb_config_descriptor {
__u8 bLength; //该描述符结构体大小
__u8 bDescriptorType; //描述符类型（本结构体中固定为0x02)
__le16 wTotalLength; //此配置返回的所有数据大小
__u8 bNumInterfaces; //此配置的接口数量
__u8 bConfigurationValue; //Set_Configuration 命令所需要的参数值
__u8 iConfiguration; //描述该配置的字符串的索引值
__u8 bmAttributes; //供电模式的选择
__u8 bMaxPower; //设备从总线提取的最大电流
} __attribute__ ((packed));


//接口描述符结构体
struct usb_interface_descriptor {
__u8 bLength; //该描述符结构大小
__u8 bDescriptorType; //接口描述符的类型编号(0x04）
__u8 bInterfaceNumber; //接口描述符的类型编号(0x04）
__u8 bAlternateSetting; //接口描述符的类型编号(0x04）
__u8 bNumEndpoints; //该接口使用的端点数，不包括端点0
__u8 bInterfaceClass; //接口类型
__u8 bInterfaceSubClass; //接口子类型
__u8 bInterfaceProtocol; //接口遵循的协议
__u8 iInterface; //描述该接口的字符串索引值
} __attribute__ ((packed));


// 端点描述符结构体
struct usb_endpoint_descriptor {
```

```
__u8 bLength; //端点描述符字节数大小（7个字节）
__u8 bDescriptorType; //端点描述符类型编号（0x05）
__u8 bEndpointAddress; //端点地址及输入输出属性
__u8 bmAttributes; //端点的传输类型属性
__le16 wMaxPacketSize; //端点收、发的最大包大小
__u8 bInterval; //主机查询端点的时间间隔
/* NOTE: these two are _only_ in audio endpoints. */
/* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
__u8 bRefresh; //声卡用到的变量
__u8 bSynchAddress;
} __attribute__ ((packed));
```

1.9 枚举示意图

USB 枚举实际上是 host 检测到 device 插入后，通过发送各种标准请求，请 device 返回各种 USB 描述符的过程。

USB 枚举的示意图如下：



说明：总线复位操作是hub通过驱动数据线到复位状态SE0(Single-Ended 0,即D+和D-全为低电平)，并持续至少10ms。

1.10 USB 标准请求的结构

上述提及的USB标准请求的结构如下：

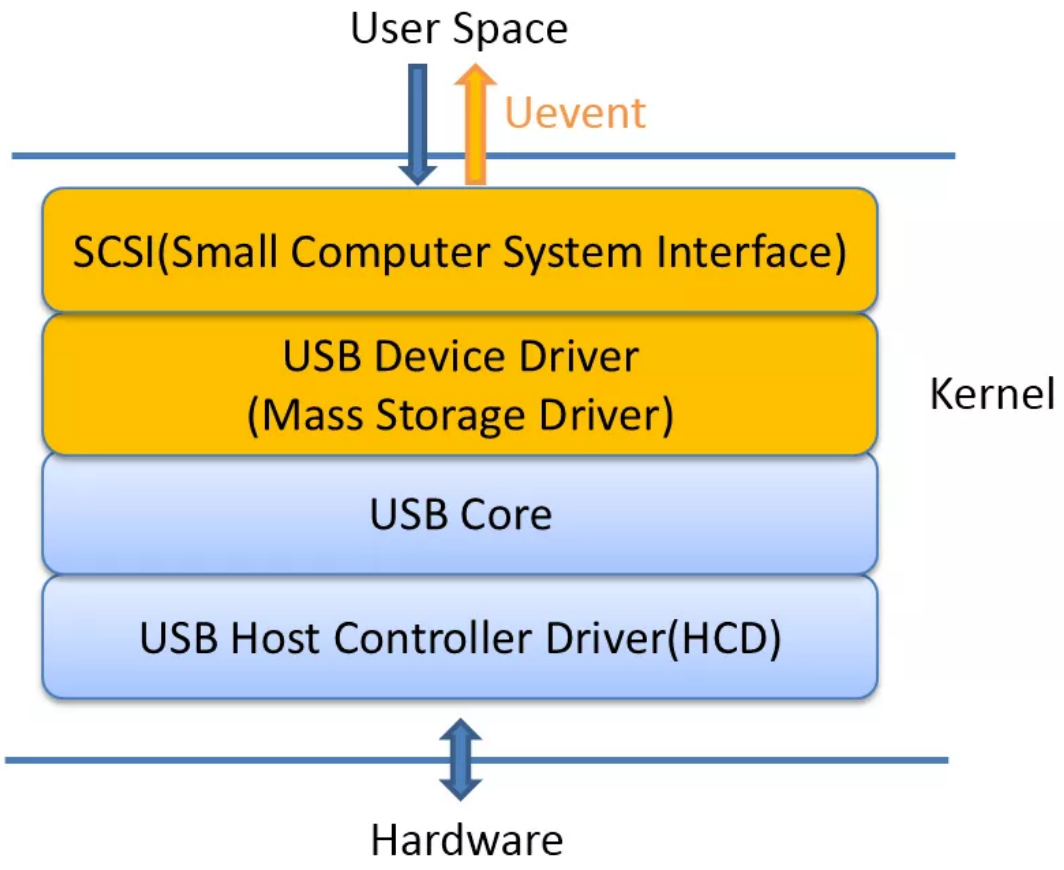
setup: bRequest:6,bRequestType:80,wValue:100, wLength:12,wIndex:0 （十六进制）

域	bmRequestType	bRequest	wValue	wIndex	wLength
字节数	1	1	2	2	2
描述	请求的特性	请求代码	该域的意义由具体的请求决定		数据过程所需要传输的字节数

bRequest	Value	bRequest	Value
GET_STATUS	0	SET_DESCRIPTOR	7
CLEAR_FEATURE	1	GET_CONFIGURATION	8
——	2	SET_CONFIGURATION	9
SET_FEATURE	3	GET_INTERFACE	10
——	4	SET_INTERFACE	11
SET_ADDRESS	5	SYNCH_FRAME	12
GET_DESCRIPTOR	6		

1.11 U盘

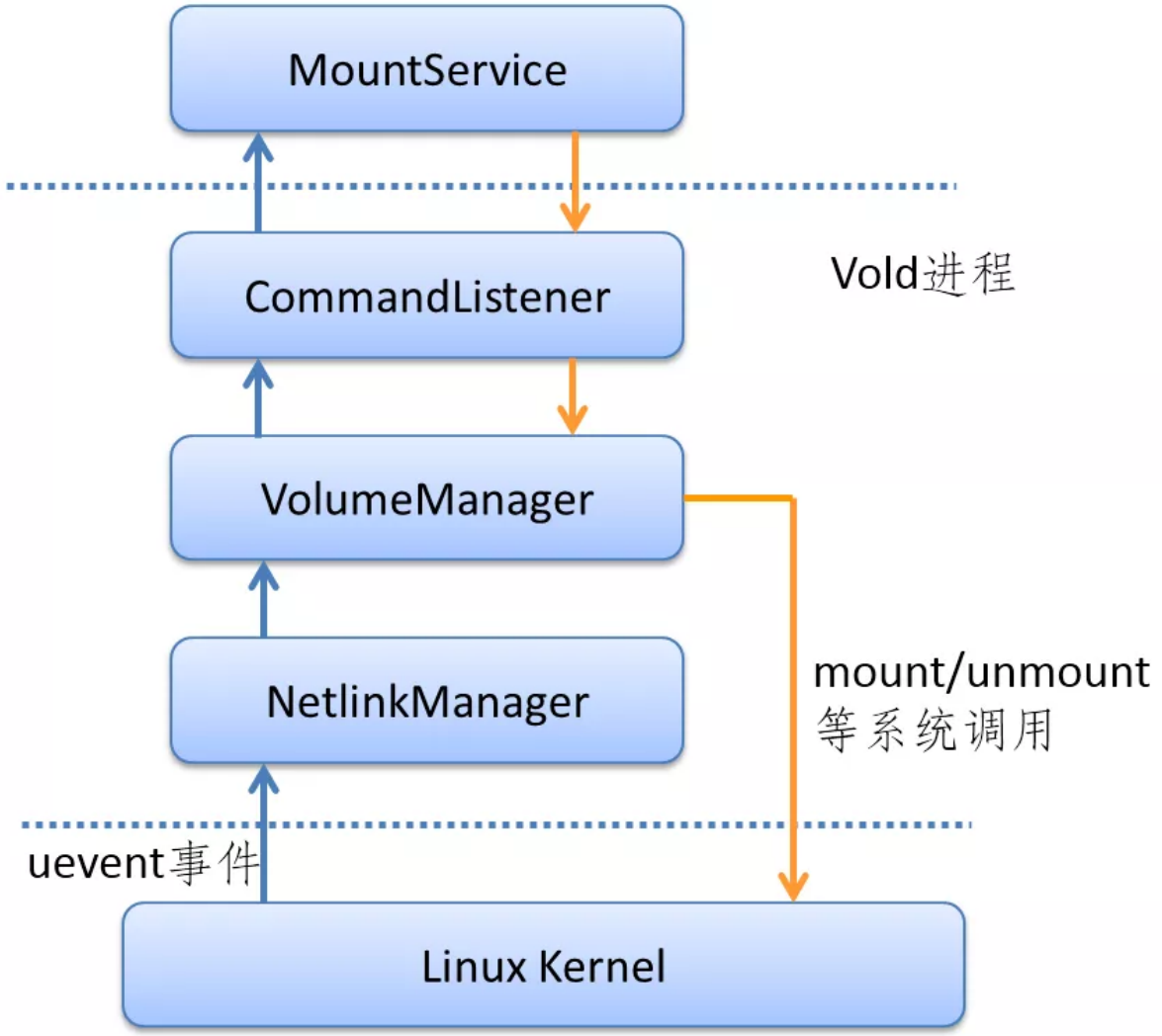
1.11.1 U盘驱动框架



如上图所示，USB Device Driver 识别到U盘设备后，还需要将U盘模拟为SCSI(小型计算机系统接口)设备，才能与 User Space 进行数据传输。

1.11.2 U盘 mount 流程

Linux Kernel 将U盘模拟为 SCSI 设备后，会向 vold(volume deamon) 发送 Uevent，并按以下流程完成U盘的mount：

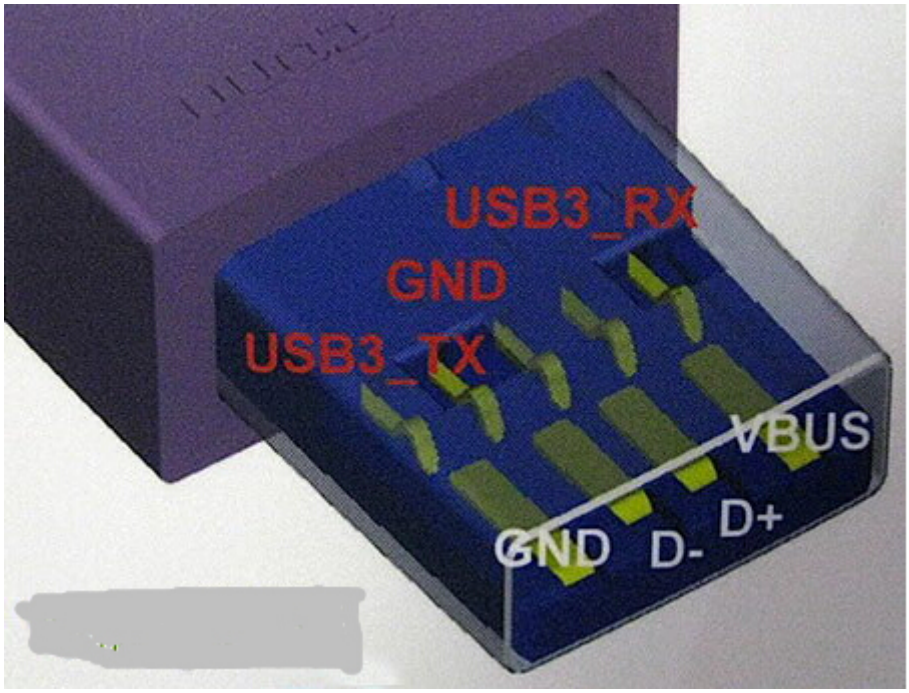


二、 USB 3.0

2.1 构成

目前市场上有大量在使用的个人电脑只支持USB 2.0。还有USB 2.0外围设备使用数量较多。USB3.0需要保持向后兼容性。从硬件上来看，实际 usb3.0 和 usb2.0 已经是两种设备。

在A型座的usb口上，我们能明显看到，usb2.0仍旧使用的是GND，D+，D-，VBUS。在3.0中则使用的是一对USB3_TX差分线，USB3_RX差分线，GND，BUS。**所以在物理上我们要明确，实际上 USB2.0 和 USB3.0 已经不是一个设备。**虽然USB3.0仍然延续了大部分2.0的概念。



2.2 通讯流程

Usb3.0 在框架层级是向后兼容 USB 2.0 的。**在传输的类型上，仍旧是控制传输，中断传输，批量传输，同步传输四种。**然而，USB 2.0和超高速协议还是有一些根本性的差异：

- USB 2.0使用三部分事务交易（令牌，数据和握手），而超高速对这相同的三部分的使用是不相同的。对于输出（OUTs），令牌被列入数据包；而对于输入（INs），令牌则被握手所取代。
- USB 2.0不支持突发（bursting）而超高速支持连续突发（continuous bursting）。
- USB 2.0是一个半双工广播（broadcast）总线，而超高速是双重单工（dual-simplex）单播（unicast）总线，这就允许同时进行IN和OUT事务交易。
- USB 2.0使用轮询模型，而超高速使用异步通知。
- USB 2.0没有流（Streaming）的能力，而超高速支持对批量端点的流（Streaming）。

2.3 USB3.0 OTG

在 usb2.0 时代，为了满足移动设备单 usb 口既可以为主也可以为从的需求，出现了 otg 功能。

Usb2.0的otg是通过micro或miniusb座子上的第5个id pin上的电平来完成识别，当id pin的电平为高，则为从机，当该电平为低时，则为主机。我们市面上买的otg线，内部电路就是把id pin与GND线相接，以实现otg线插入后，手机可以作为host端。

在 usb3.0 中，id pin 的功能同样被强大的 typec 所取代，主从的识别将通过cc来识别。同时主从双方也可以通过cc的通讯来切换角色。

在usb2.0中，供电方与受电方和设备的主从关系是绑定的，只有host可以给devices供电。usb3.0中则完全不同，两者完全独立，在做host的同时，依旧可以接受供电。解决了“手机没电时，就无法插usb设备”的问题

同时在硬件上，我们需要明确usb 3.0 otg的组成部分

三、Android 中的 USB 技术

- UsbManager

  Usb的管理类
- UsbDevice

  Usb设备
- UsbInterface

  Usb对应的接口，通过接口拿到内部匹配Usbpoint
- UsbEndPoint

  Usb通信数据的传输主要其实就是通过这个类来进行的
- UsbDeviceConnection

  Usb连接器

扩展

一、大小端模式

大端模式，是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，数据从高位往低位放；这和我们的阅读习惯一致。

小端模式，是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低。

下面以 unsigned int value = 0x12345678为例，分别看看在两种字节序下其存储情况，我们可以用 unsigned char buf[4] 来表示 value

Big-Endian: 低地址存放高位，如下：

```
低地址
---------------
buf[0] (0x12) -- 高位字节
buf[1] (0x34)
buf[2] (0x56)
buf[3] (0x78) -- 低位字节
---------------
高地址
```

Little-Endian: 低地址存放低位，如下：

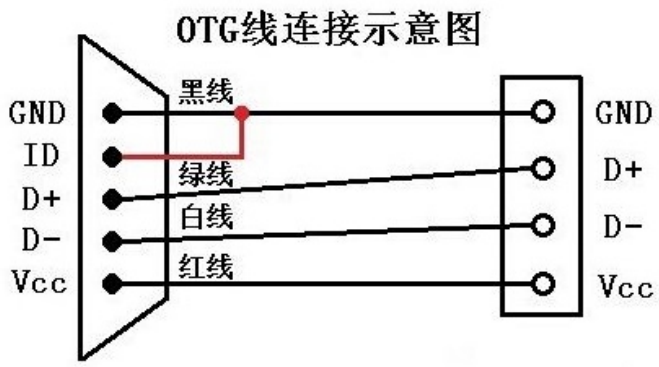
```
低地址
---------------
buf[0] (0x78) -- 低位字节
buf[1] (0x56)
buf[2] (0x34)
buf[3] (0x12) -- 高位字节
--------------
高地址
```

内存地址	小端模式存放内容	大端模式存放内容
0x4000	0x78	0x12
0x4001	0x56	0x34
0x4002	0x34	0x56
0x4003	0x12	0x78

OTG

概念

USB OTG是USB On-The-Go的缩写，是近年发展起来的技术，2001年12月18日由USB Implementers Forum公布，**主要应用于各种不同的设备或移动设备间的联接，进行数据交换**，特别是PAD、移动电话、消费类设备。



OTG 线 接线示意图

OTG技术主要是基于USB技术的发展和普及而来，完全兼容USB2.0协议。使用OTG可以脱离计算机设备，利用各种设备上的USB口进行数据交换。解决了各种设备间不同制式的连接接口的数据交换不便的问题。

OTG 的使用

鉴于设备接口及USB类型的不一致，一般OTG有如下几类：

- USB2.0 OTG：包括Micro 5PIN OTG（常见安卓手机）、Mini 5PIN OTG（常见安卓平板）
- Micro USB3.0 OTG：三星Note3、Galaxy S5等在2016年以前的安卓手机OTG接口
- Type-C OTG：目前支持Type-C接口的设备（比如小米Mix系列等）
- Lightning OTG：苹果手机专用OTG

由此可以看出，使用 OTG 时要根据设备的接口选用不同的 OTG 类型。OTG 的具体使用比较简单，且一般都大同小异。在使用OTG功能时要分如下情况：

- 设备连接：要使用针对不同设备接口的 OTG 连接线/转接头，将两端分别接到两台设备的的相应的USB接口中
- 数据存储：有相应接口的U盘，直接将接口插入的手机中即可

参考资料

带你遨游USB世界

https://blog.csdn.net/weixin_38737818/article/details/103893358