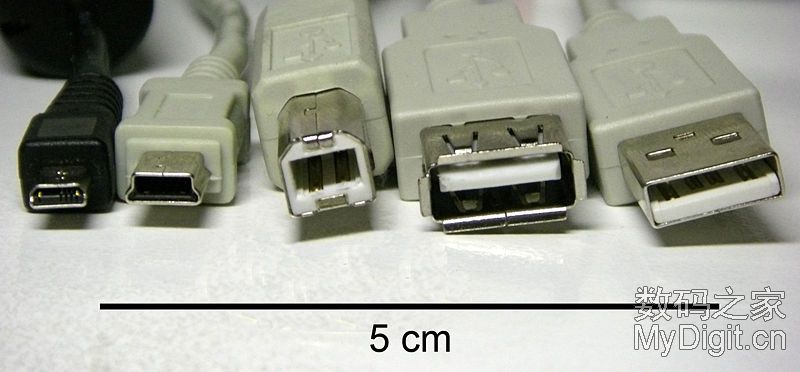
# USB术语

## 硬件接口



从左往右依次为：miniUSB公口(A型插头)、miniUSB公口(B型插头)、USB公口(B型)、USB母口(A型插座)、USB公口(A型插头) 。

### USB接口：



### Mini-USB接口：



## USB设备

USB设备分为三种：HOST，OTG，DEVICE。其中ID脚在OTG功能中才使用。

如果你的系统是用做Slave，就使用B接口。任何时候都悬空，都做Slave。如果是OTG设备，它的系统控制器会判断外部设备的ID脚的电平，决定是什么样的设备插入：如果是高电平，则是B接头插入，此时OTG设备就做主模式(master mode)；如果ID为低，则是A接口插入，然后OTG设备就会使用HNP对话协议来决定它跟外设哪个做Master，哪个做Slave。还有种主动OTG设置，就是接入外设后，在OTG设备中点击HOST使能，会强制ID拉低。此时的Master和Slave关系确定。  
      我们手机上一般用的都是B型Mini-USB口

## OTG

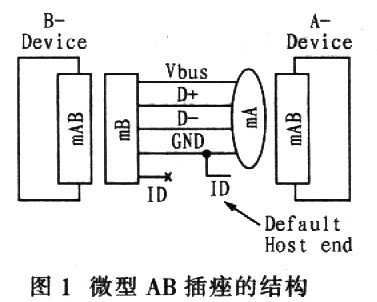
特性：**既能充当HOST又能用做Device**，即DRD（Dual-role-devices）；OTG 设备依然可以连接到PC上用做普通的外设，即POD（Peripheral-Only Device）。协议简述：**USB OTG 中引入了两个新的协议**， **SRP**（Session Request Protocol）和**HNP**（Host Negotiation Protocol），即会话请求协议和主机交流协议。

### SRP

SRP 用于B设备向A设备请求建立会话和使用总线。OTG系统中的A设备一般采用电池供电，因此，OTG系统电源管理是非常重要的。为了节省OTG系统的功耗，OTG规定A设备在没有总线活动的时候，可以关掉VBus上的电源。这样，当一个B设备连接到A设备上之后，就要初始化SRP，并发送给A设备，请求A设备在VBus上提供电流支持，进而进行通信。在OTG中，1个会话定义为A类设备向电源总线VBUS有效供电的时间。DRD既可作为A设备，也可作为B设备，因此 DRD必须支持初始化SRP和响应SRP，POD只能作B设备，所以只能初始化SRP。

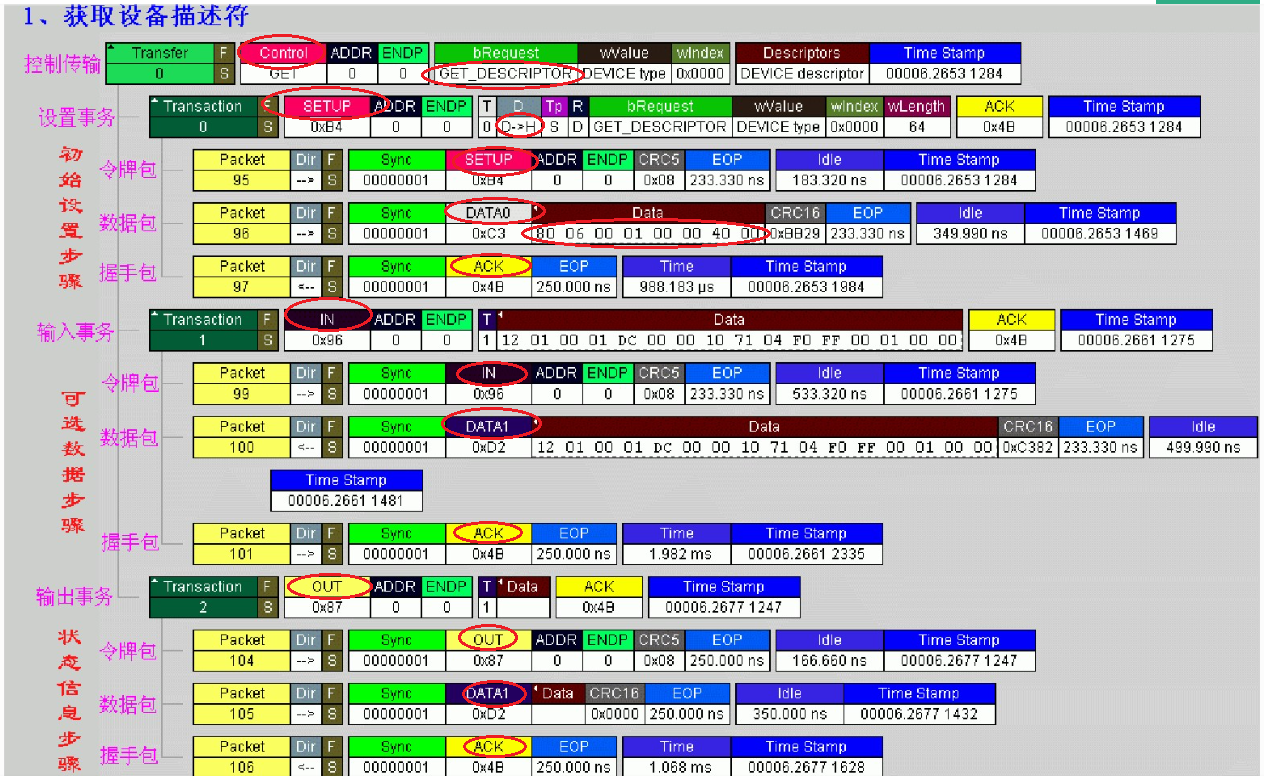
### HNP

HNP 用于两个OTG设备互连时角色切换。当2个DRD互连时，哪个设备作为主机，为什么要作为主机？为了解决这两个问题，在OTG中提出了新的协议——主机协商协议（HNP）。在OTG中还定义了一种新的接口——微型AB插座(mini-AB receptacle)以及微型A插头(mi-ni-A plug)和微型B插头(mini-B plug)。在微型AB插座、微型A插头和微型B插头中增加了1个引脚——ID引脚，为了实现HNP，简而言之，两个OTG设备互连时，可以通过HNP协议协商主从地位。协商过程如下：



在微型A插头中，ID引脚接地，在微型B插头中，ID引脚悬空。在OTG中，如果设备的ID引脚接地（如图l右侧的设备），则此设备默认为主机，否则为外设。同时，在设备连接使用过程中，通过主机协商协议，允许主机和外设功能互换。例如，假设图1左边的B-Device为1个手持PDA，右边的A-Device为1个打印机。由于连接线的关系，打印机初始化为主机。但是打印机的驱动程序存在PDA中，这时需要PDA作为主机，打印机作为外设。通过HNP可以方便实现此功能，而不必拔下连接线调换插头方向，重新连接打印机和PDA。

# USB协议



## 基本概念

传输类型: 控制、批量、中断、等时 //Transfer

事务类型: IN、OUT、SETUP， 1个传输 由 多个事务组成 //Transaction

包：Packet， 1个事务 由 多个Packet组成 //Packet

管道：Host 与 Endpoint 之间的关联，一个USB设备可以有多个管道(pipes)。

高速USB 总线的帧周期为125us，

全速以及低速 USB 总线的帧周期为 1ms。

## 1. USB传输类型

在USB的传输中，定义了4种传输类型：

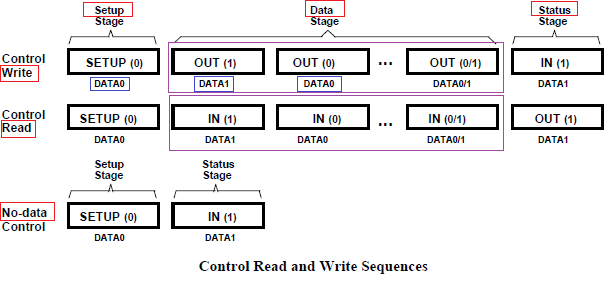
• 控制传输 (Control Transfer)

• 中断传输 (Interrupt Transfer)

• 批量传输 (Bulk Transfer)

• 同步传输 (Isochronous)

### 1.1 控制传输 (Control Transfer)



**控制传输过程：Setup – Data(Write/Read/NoData) – Status(ACK/NAK/STALL)**

#### 1.1.1 建立阶段 -- Setup

主机从USB设备获取配置信息，并设置设备的配置值。建立阶段的数据交换包含了SETUP令牌封包、紧随其后的DATA0数据封包以及ACK握手封包。它的作用是执行一个设置（概念含糊）的数据交换，并定义此控制传输的内容(即：在Data Stage中IN或OUT的data包个数，及发送方向，在Setup Stage已经被设定)。

#### 1.1.2 数据阶段 -- Data

根据数据阶段的数据传输的方向，控制传输又可分为3种类型：

1) 控制读取（读取USB描述符）

2) 控制写入（配置USB设备）

3) 无数据控制

数据传输阶段：用来传输主机与设备之间的数据。

• 控制读取

是将数据从设备读到主机上，读取的数据USB设备描述符。该过程如下图的【Control Read】所示。对每一个数据信息包而言，首先，主机会发送一个IN令牌信息包，表示要读数据进来。然后，设备将数据通过DATA1/DATA0数据信息包回传给主机。最后，主机将以下列的方式加以响应：当数据已经正确接收时，主机送出ACK令牌信息包；当主机正在忙碌时，发出NAK握手信息包；当发生了错误时，主机发出STALL握手信息包。

• 控制写入

是将数据从主机传到设备上，所传的数据即为对USB设备的配置信息，该过程如下的图【Control Wirte】所示。对每一个数据信息包而言，主机将会送出一个OUT令牌信息包，表示数据要送出去。紧接着，主机将数据通过DATA1/DATA0数据信息包传递至设备。最后，设备将以下列方式加以响应：当数据已经正确接收时，设备送出ACK令牌信息包；当设备正在忙碌时，设备发出NAK握手信息包；当发生了错误时，设备发出STALL握手信息包。

#### 1.1.3 状态阶段 -- status

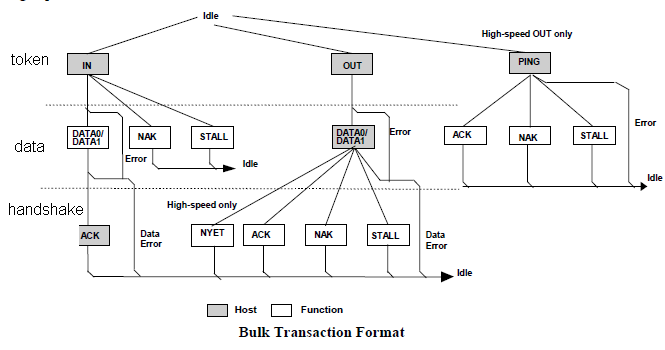
状态阶段：用来表示整个传输的过程已完全结束。

状态阶段传输的方向必须与数据阶段的方向相反，即原来是IN令牌封包，这个阶段应为OUT令牌封包；反之，原来是OUT令牌封包，这个阶段应为IN令牌封包。

对于【控制读取】而言，主机会送出OUT令牌封包，其后再跟着0长度的DATA1封包。而此时，设备也会做出相对应的动作，送ACK握手封包、NAK握手封包或STALL握手封包。

相对地对于【控制写入】传输，主机会送出IN令牌封包，然后设备送出表示完成状态阶段的0长度的DATA1封包，主机再做出相对应的动作：送ACK握手封包、NAK握手封包或STALL握手封包。

### 1.2 批量传输 (Bulk Transfer)



**批量传输过程：Token(IN/OUT) – DATA(Data0-Data1-Data0…) – Handshake(ACK/NAK/STALL)**

#### 批量传输特点：

•用于传输大量数据，要求传输不能出错，但对时间没有要求，适用于打印机、存储设备等。

•批量传输是可靠的传输，需要握手包来表明传输的结果。若数据量比较大，将采用多次批量事务传输来完成全部数据的传输，传输过程中数据包的PID 按照 DATA0-DATA1-DATA0-…的方式翻转，以保证发送端和接收端的同步。

• USB 允许连续 3次以下的传输错误，会重试该传输，若成功则将错误次数计数器清零，否则累加该计数器。超过三次后，HOST 认为该端点功能错误（STALL），放弃该端点的传输任务。

• 一次批量传输（Transfer）由 1 次到多次批量事务传输（Transaction）组成。

• 翻转同步：发送端按照 DATA0-DATA1-DATA0-…的顺序发送数据包，只有成功的事务传输才会导致 PID 翻转，也就是说发送端只有在接收到 ACK 后才会翻转 PID，发送下一个数据包，否则会重试本次事务传输。同样，若在接收端发现接收到到的数据包不是按照此顺序翻转的，比如**连续收到两个 DATA0，那么接收端认为第二个 DATA0 是前一个 DATA0 的重传**。

它通过在硬件级执行“错误检测”和“重传”来确保host与device之间“准确无误”地传输数据，即可靠传输。它由三种包组成(即IN事务或OUT事务)：**token，data，handshake。**

1) For IN Token (即：IN Transaction)

• ACK: 表示host正确无误地接收到数据

• NAK: 指示设备暂时不能返回或接收数据 (如：设备忙)

• STALL:指示设备永远停止，需要host软件的干预 (如：设备出错)

2) For OUT Token (即：OUT Transaction)

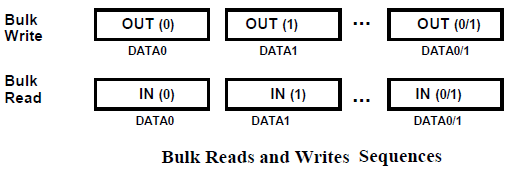
如果接收到的数据包有误，如：CRC错误，Device不发送任何handshake包

• ACK: Device正确无误地接收到数据包，且通知Host可以按顺序发送下一个数据包

• NAK: Device正确无误地接收到数据包，且通知Host重传数据，由于Device临时状况(如buffer满)

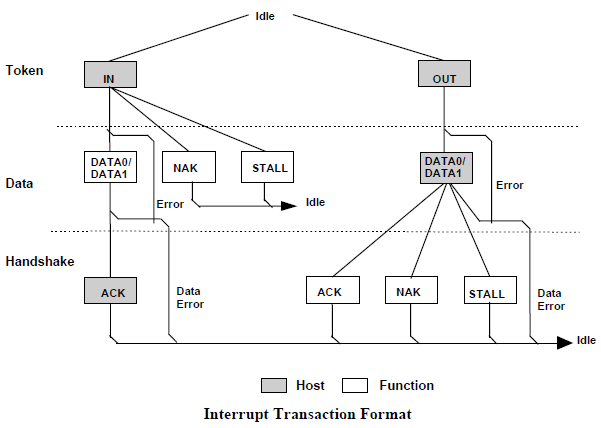
• STALL: 指示Device endpoint已经停止，且通知Host不再重传

3) Bulk读写序列



即由一系统IN事务或OUT事务组成。

### 1.3 中断传输(Interrupt Transfer)



**中断传输过程：Token(IN/OUT) – DATA(Data0-Data1-Data0…) – Handshake(ACK/NAK/STALL)**

#### 中断传输特点

**中断传输在流程上除不支持PING 之外，其他的跟批量传输是一样的。**他们之间的**区别也仅在于事务传输发生的端点不一样、支持的最大包长度不一样、优先级不一样等**这样一些对用户来说透明的东西。

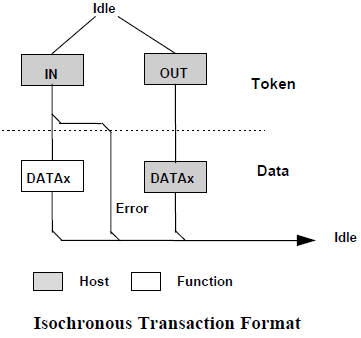
主机在排定中断传输任务时，会根据对应**中断端点描述符**中指定的查询间隔发起中断传输。中断传输有较高的优先级，仅次于同步传输。

同样中断传输也采用PID翻转的机制来保证收发端数据同步。下图为中断传输的流程图。

中断传输方式总是用于对设备的查询，以确定是否有数据需要传输。因此中断传输的方向总是从USB设备到主机。

DATA0或DATA1中的包含的是中断信息，而不是中断数据。

### 1.4 同步传输(Isochronous Transfer)



**同步传输过程：Token -- Datax**

#### 同步传输的特点：

同步传输不支持“handshake”和“重传能力”，所以它是不可靠传输。

同步传输是不可靠的传输，所以它没有握手包，也不支持PID翻转。主机在排定事务传输时，同步传输有最高的优先级。

同步传输适用于必须以固定速率抵达或在指定时刻抵达，可以容忍偶尔错误的数据上。实时传输一般用于麦克风、喇叭、UVC Camera等设备。实时传输只需令牌与数据两个信息包阶段，没有握手包，故数据传错时不会重传。

### 1.5传输数据丢失原因

#### 1、数据丢失原因：

（1）主要原因是Windows不是实时系统，不能保证每次都是准时地完成BULK IN操作，特别是有其他进程运行时（如：笔记本的电源管理、实时杀毒等）。此外如果机器上有其他USB设备，也会造成USB带宽受限引起BULK 传输的实时性不能保证，造成数据丢失。

（2）BULK传输要进行CRC校验，一旦发现有错，数据作废，需要重传，但这是实时系统不允许的，当然也就造成数据丢失。

（3）Device的数据缓冲区容量不够大，以至于HOST稍有停顿，缓冲区就会溢出，造成数据丢失。

(4) 如CY7C68013A数据手册46页图21所示， 在Slave FIFO Asynchronous Write模式, SLWR周期必须 >=120ns,即SLWR频率<=8.3MHz，否则会导致端点FIFO中的数据丢失。

#### 2、解决办法：

（1）卸载主机中所有影响USB带宽和实时性的软件模块，特别是笔记本的电源管理和杀毒软件，拔掉(卸载)其它USB Devices。

（2）加大Device的发送Buffer（例如在数据源于端点之间加个大容量的FIFO，如：IDT72V2103等）和Host的接收Buffer容量， 加大HOST每次BULK传输的总字节数（可减少“微帧”间隔时间）。

（3）如果对个别数据出错（是出错，不是丢失！）的要求不是很严格（如用于观测的图像数据，个别数据出错看不出来，但丢失数据就有可能错行，彩色图像还会错色），可改用ISO方式(注意ISO每帧必须是1024字节)，即使传输出错（可能是个别数据出错，对视频图像影响不大！）也保留数据，不需重传，尽量保证传输的实时性，减少数据丢失。

## 2. 事务(Transaction)

1个传输由多个事务构成，事务包括: SETUP, IN, OUT

* **控制传输过程：Setup – Data(Write/Read/NoData) – Status(ACK/NAK/STALL)**
* **批量传输过程：Token(IN/OUT) – DATA(Data0-Data1-Data0…) – Handshake(ACK/NAK/STALL)**
* **中断传输过程：Token(IN/OUT) – DATA(Data0-Data1-Data0…) – Handshake(ACK/NAK/STALL)**
* **同步传输过程：Token -- Datax**

在USB上数据信息的一次接收或发送的处理过程称为事务处理（Transaction）即：The delivery of service to an endpoint。一个事务由一系统packet组成，具体由哪些packet组成，它取决于具体的事务。可能由如下包组成：

• 一个token packet

• 可选的data pcket

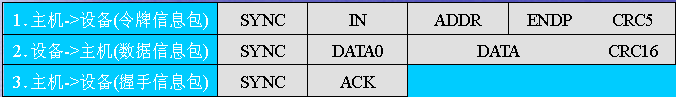
• 可选的handshake packet

• 可选的special packet

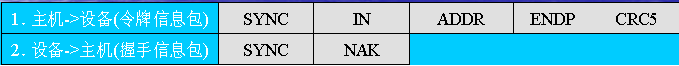
### 2.1 输入（IN）事务处理

输入事务处理：表示USB主机从总线上的某个USB设备接收一个数据包的过程。

•【正常】的输入事务处理



•【设备忙】时的输入事务处理



•【设备出错】时的输入事务处理



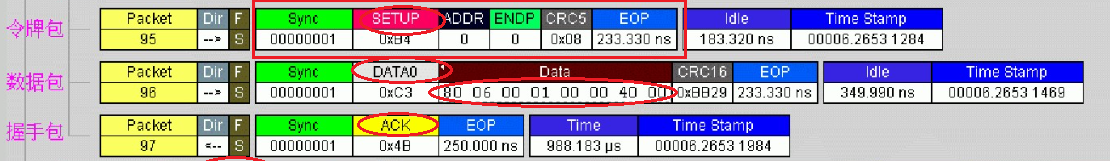
### 2.2. 输出（OUT）事务处理

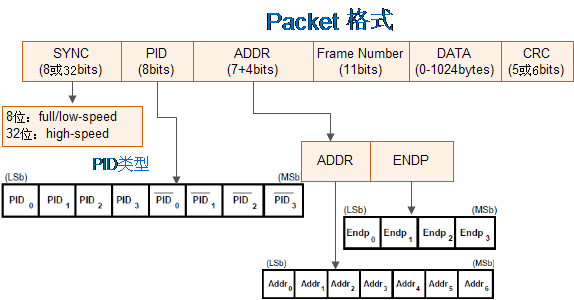
同上， IN改为OUT

### 设置（SETUP）事务处理

同上， IN改为SETUP

## 3. 包(Packet)

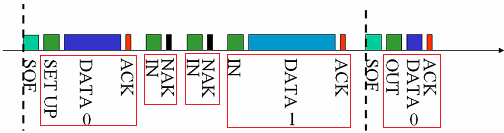




包（Packet）是USB系统中信息传输的基本单元，所有数据都是经过打包后在总线上传输的。数据在 USB总线上的传输以包为单位，包只能在帧内传输。**高速USB 总线的帧周期为125us，全速以及低速 USB 总线的帧周期为 1ms。**帧的起始由一个特定的包（SOF 包）表示，帧尾为 EOF。EOF不是一个包，而是一种电平状态，EOF期间不允许有数据传输。

注意：虽然高速USB总线和全速/低速USB总线的帧周期不一样，但是SOF包中帧编号的增加速度是一样的，因为在高速USB系统中，SOF包中帧编号实际上取得是计数器的高11位，最低三位作为微帧编号没有使用，因此其帧编号的增加周期也为 1mS。

**• USB总线上的情形是怎样的？**



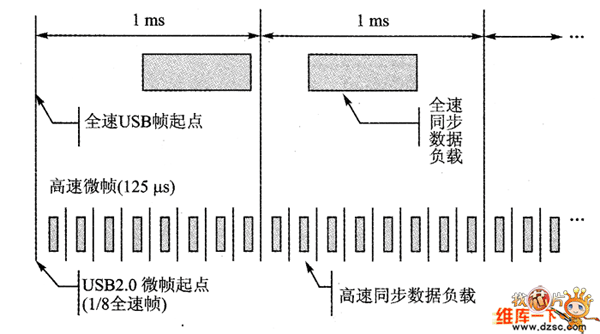
• 包是USB总线上数据传输的最小单位，不能被打断或干扰，否则会引发错误。若干个数据包组成一次事务传输，一次事务传输也不能打断，属于一次事务传输的几个包必须连续，不能跨帧完成。一次**传输**由一次到多次事务传输构成，**可以跨帧**完成。

### 帧与微帧的概念

根集线器会在每1 ms时，送出SOF封包。这介于2个SOF封包之间的时间，即称为**帧**（frame）。SOF封包虽是属于令牌封包的一种，但却具有独自的PID形态名称SOF。通常目标设各都利用SOF封包来辨识帧的起点。这个封包常用于等时传输。也就是**在1 ms的帧（高速是125 μS微帧，将1 ms切成8份）开始时，等时传输会利用SOF激活传输并达到同步传输的作用。**而在每一个帧开始时，SOF会传给所有连接上去的全速设各（包含集线器）。因此，SOF封包并不适用于低速设备。这个封包内包含了一个**帧码**，其可不断地递增，且在高达最大值时反转为0，重新再计

　　数一次。这个帧码是用来表示帧的计数值，因此，**8个微帧都使用同一个帧码值**。若必要时，高速设各可计算出SOF的重复使用次数，并计算出微帧的数量。通过缩短微帧的周期时间，便可减少高速设各对于缓冲存储器的需求。

如图1所示，高速的根集线器将会使用额外的SOF来传输8个微帧。有些书籍会把这种高速的SOF，另命名为uSOF。这种增加的微帧，同时也替高速的连接带来了更复杂的控制方式。



### 54233的概念

5： 一个令牌封包含盖了5个数据域SYNC、PID、ADDR、ENDP与CRC5。

4： 一个数据封包含了4个数据域：SYNC、PID、DATA与CRC16。

2: 一个握手封包含了2个数据域：SYNC与PID。

3: 令牌封包+数据封包+握手封包，即可组成一个数据交易。

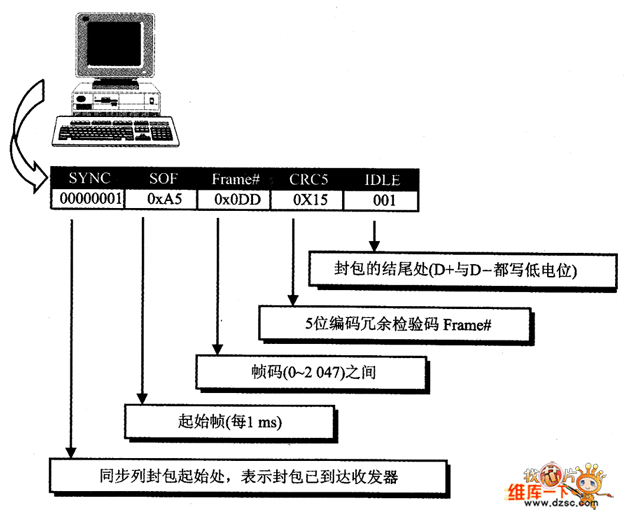
### 3.0 起始（SOP）封包： Start-of-Frame Packets

SOF包由Host发送给Device。

1) 对于full-speed总线，每隔1.00 ms ±0.0005 ms发送一次；

2) 对于high-speed总线，每隔125 μs ±0.0625 μs发送一次；

主机SOF封包的格式：PID数据域的值为0xA5。



此外，端点可以通过SOF封包来加以同步，或是以帧码值来作为时间的参考依据。当整个USB总线上没有USB传输时，SOF封包也可避免让设备切人低功率的中止（suspend）状态。再者，虽然在低速设各上，是看不到SOF封包的，但相反，设各的集线器使用了前面所提及的BOP（End of－Packet）信号，且在每一个帧设置一次。因此，有时后也称这种信号为设各的低速存活（keep alive）信号。所以说，SOF／uSOF封包是给全速／高速设各来使用的，而低速存活信号却可避免让低速设备切入中止状态中。

### 3.1 令牌封包: Token Packets

**PID 数据传输方向**

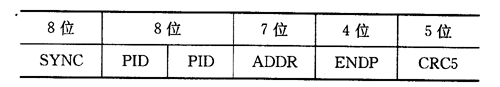
IN Device->Host

OUT Host->Device

SETUP Host->Device

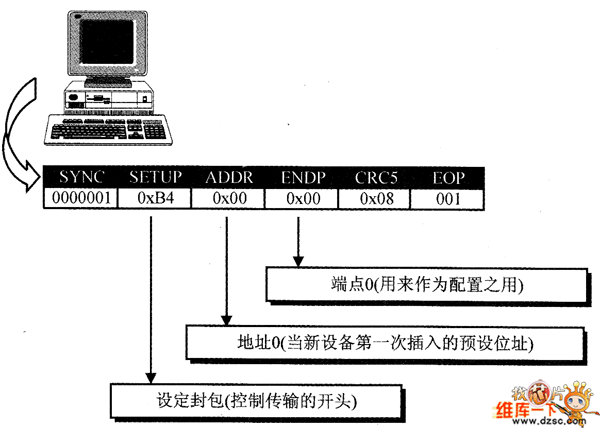
PING Device->Host

　一个令牌封包含盖了5个数据域SYNC、PID、ADDR、ENDP与CRC5。这即是54233的第1个数字：5。如下所列为其令牌封包的各个组成的数据域。



　　令牌封包的PID数据域（PID［1：0］＝［0，1］）中包含了OUT、IN、SETUP这3种PID类型名称。例如，在执行控制传输主机要通过预设的地址取得设备描述符（Get￣Descriptor），就必须先执行下列的SETUP令牌封包，作为每一次控制传输的开始，其中，PID栏变成SETUP的PID类型名称（0xB4）。IN令牌封包，则是主机用来通知设备，将要执行数据输入的工作。而OUT令牌封包则刚好相反。

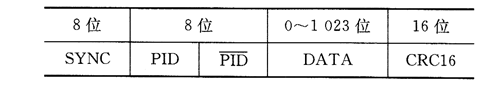
　　图3 显示了PC主机所起始的SETUP令牌封包。



### 3.2 数据封包: Data Packets

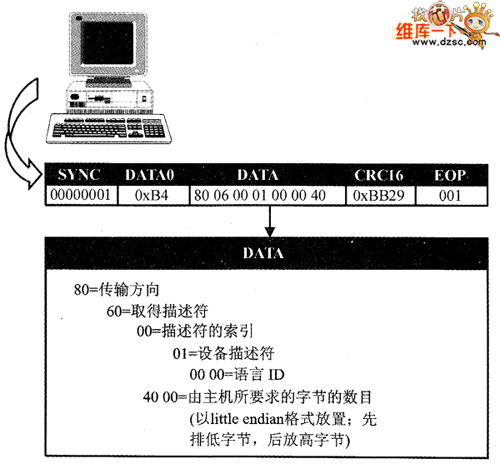
　　由SETUP、IN与OUT令牌封包所起始的数据传输，将会以DATAO、DATA1、DATA2与MDATA封包来加以实现。一个数据封包含了4个数据域：SYNC、PID、DATA与CRC16。各个字段的意义之前已有介绍过。这即是54233的第2个数字：4。在这里，要稍微注意的是DATA字段内所放置的位值，须根据USB设各的传输设备（低速、高速与全速）以及传输类型（中断传输、批量传输与等时传输）而定，且须以所设置的MaxPackSize字节为基本单位。也即是，若传输的数据不足MaxPackSize字节，或是传输到最后所剩余的也不足MaxPackSize字节，则仍须传输MaxPackSize个字节的数据域。

　　列出由4个数据域所组合而成的数据封包。



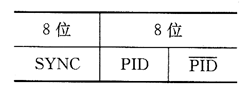
　　数据封包的PID数据域（PID［1：0］＝［1：1］）包含了4种类型：DATA0、DATA1、DATA2与MDATA。而根据USB规范，**最初的数据封包都以DATA0作为开始，其后才是DATA1，然后依此方式交替切换。这个动作称之为数据紧密连接（data toggle）。**这个动作有点类似将数据紧密连接。如此就可确保整个传输过程中，主机能与设各维持同步，且作为帧错之用。例如，如果两个连续的DATA0被接收到的话，意味着DATA1封包被遗漏掉，并产生了错误的状况。而DATA2与MDATA，则仅适用于高速的等时传输。

　　若主机要针对特别寻址的设各端点，送出取得设各描述符的命令，就可如图4所示，将含有命令的数据封包传出。其中，须特别注意的是，由于是控制传输，所以数据域中仅有8字节。至于“80 06 0001 0000 4000”的设备要求的意义。



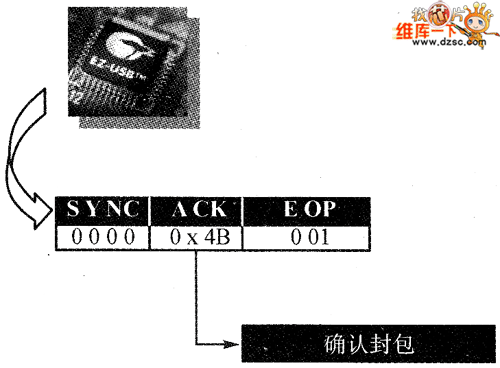
### 3.3 握手封包: Handshake Packets

　　握手封包是最简单的封包类型。在这个握手封包中，仅包含一个PID数据域。它的格式如下所列，仅包含SYNC与PID两个数据域，这即是54233的第3个数字：2。



　　含有ACK、NAK、STALL与NYET握手封包。

　　延伸上一个图例，如果设备已收到主机要执行取得设备描述符的命令，设备就以握手封包来加以响应。因此须注意的是，如果设备已准备接收的话就以ACK握手封包响应；如果尚未就绪就使用NAK握手封包响应；如果发生错误而停滞，就使用STALL握手封包响应。图5显示一个握手封包的格式，其中ACK的PID数据域值为Ox4B，刚好与SETUP的PID数据域值相反。



　　通过上述的3个封包，即可组成一个数据交易。当然，这即是54233的第4个数字：3。

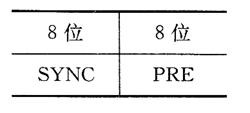
　　对于高速设备，为了改善NAK的机制，特别支持了NYET握手封包。这是由于当数据已经传输至总线时，通过NAK这个OUT数据交易的动作是不够的。况且若是在总线上存在着高频率的NAK传输过程，将会使得整个总线逐渐地被拖累，带宽被分享掉。此时，高速设各就可以使用特殊的PING封包（稍后会提及）来询问，是否接收器还有缓冲区空间来接收OUT数据交易。如果设各以ACK来响应，那么传送器就会安排OUT传输。反之，如果响应的是NYET，那么传送器就会以PING封包来查询。如此，总线上就会有最佳的使用率。

### 3.4 特殊封包

　　总共包含了4个特殊封包（PRE、ERR、SPLIT与PING）。其中，一个仅使用在低速设备，一个仅使用在高速设各，其余两个则是针对当低速或是全速设备连接上USB 2．0集线器后，再以高速的方式与主机通信时，才会用到。

　　（1）PRE封包

　　这个特殊前置（Special Preamble，PRE）封包拥有独自的PID类型名称PRE，其仅适用于主机想要从高速传输变成低速传输时所送出来的情形。也就是主机对于下端端口送出低速封包与低速设备通信之前，所必须先送出的PRE封包。在该PRE封包中，包含了前置码以告诉集线器，下一个封包是低速。如此，集线器将会以即将接上的低速设备开始执行通信的工作。此时，PRE封包会放在导引至低速设各的所有令牌、数据以及握手封包之前。而高速设各是将PRE以SPLIT封包来加以编码，因此不会重复地送出。对于原本已是低速的设备来说，不需要PRE封包。这种格式如下所列，仅包含两个8位的数据域：SYNC与PRE。



　　（2）PING封包

　　仅存于高速设备所使用的特殊封包是PING封包。主机会送出PING封包来找出是否高速设各端点在以批量或是包含以多个数据封包的控制传输来送出下一个数据封包之前，是否为忙碌的状态。这是由于传统的USB数据交易时，若常以NAK来响应批量或控制的OUT传输，通常都会导致浪费太多的带宽。因此，为了减少高速或控制的OUT端点的损失，USB 2．0新增了PING封包。一旦批量或控制传输的OUT数据交易被NAK响应后，主机控制器将会使用PING封包来查询高速非周期性的端点是否有足够的内存来接收wMaxPacketSize大小容量的数据。如果此端点具有足够的缓冲区来使用，就以ACK来响应之；反之，继续以NAK响应为止。

　　此外，高速非周期性的OUT端点也可以使用NYET来加以响应，以通知主机所要加载的数据是可以接收的，但是端点没有足够的内存。此时，主机会使用PING令牌，直到端点表示了针对下一个OUT数据交易，已有足够的缓冲区内存。

　　（3）SPLIT封包

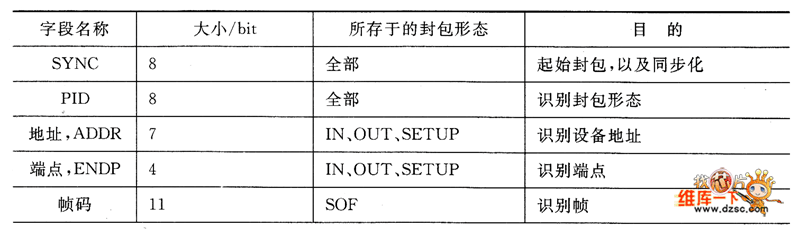
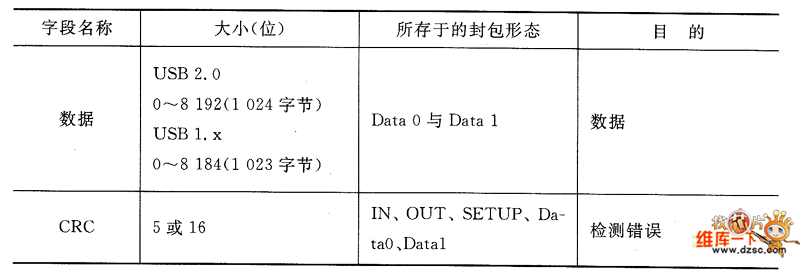
　　SPILIT封包定义了令牌封包为分割数据交易（split transactION）的一部分。为了最佳地使用总线时间，USB 2．0主机与集线器会以高速来送出低速与全速的通信数据。至于为什么需要分割数据交易呢？这是由于当主机开始传输一个针对低速或是全速的设各所预定的数据交易时，那么最接近设备的2．0集线器就有责任去实现与此设各的数据交易。此外，也负责存储任何回传的数据或是状态信息，以及以一个或是两个稍后的数据交易来加以回报回去。如此，整个总线就无须去针对实现一个低速的交易来持续地等待。而这个介于集线器与主机之间的特殊数据交易，就称之为分割数据交易。

　　（4）ERROR甚寸包

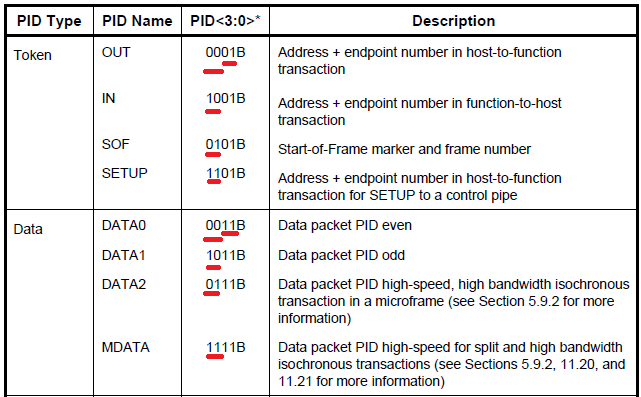
　　这个封包仅使用在分割数据交易时。2．0集线器会使用该封包并以低速或全速的数据交易来回报一个错误给主机。在此，读者是否发现该PID码值与PRE PID码值是一样的。但是其中，最大的差异是前者是应用在设各与集线器上，另一个则是应用在主机上。也即是集线器不会送出PRE封包给主机或是ERR封包给设各。

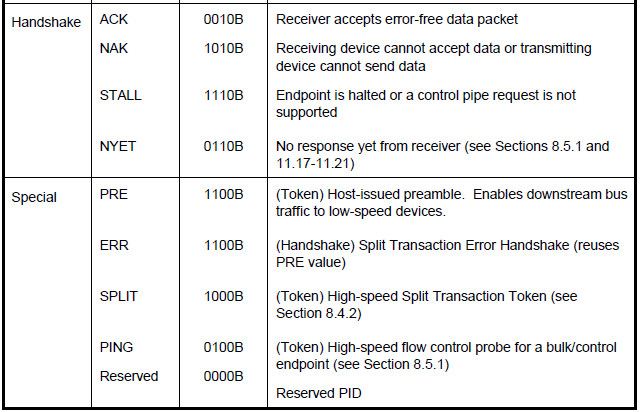
　　以下，将这些封包格式与字段等加以汇整，如表所列，并列出各个字段与其目的。

　　表 封包与字段之间的关系

### 3.5 PID类型(即包类型)





## 4. USB描述符

**USB协议中共定义了以下四种描述符：**

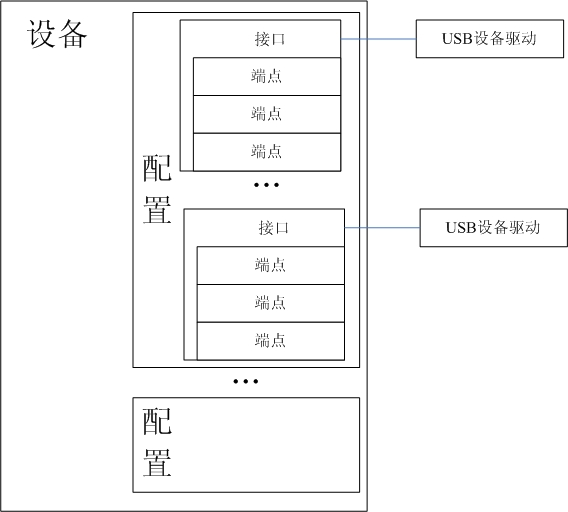
1) 设备描述符

2) 配置描述符

3) 接口描述符

4) 端点描述符

其关系如下图所示：

s

### 4.1 设备描述符

每个USB设备都有一个唯一的设备描述符，如下表所示：



### 4.2 配置描述符

每个USB设备都有默认的配置描述符，支持至少一个接口，每个配置描述符如下表：

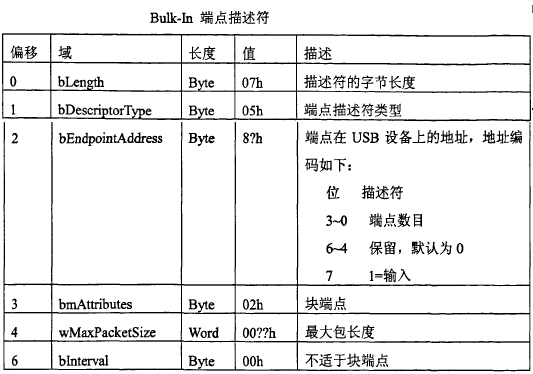


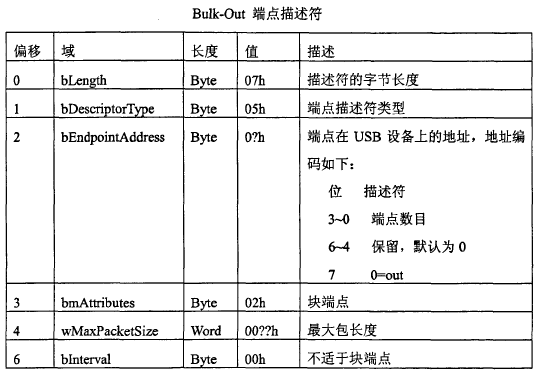
### 4.3 接口描述符

设备应至少支持一个接口，如：块传输数据接口，部分设备可能支持其它的接口。复合设备可以支持额外接口，以支持音频和视频功能。标准中并没有定义此类接口。接口可能有多个可选设置，主机将会检查每个可选的设置。

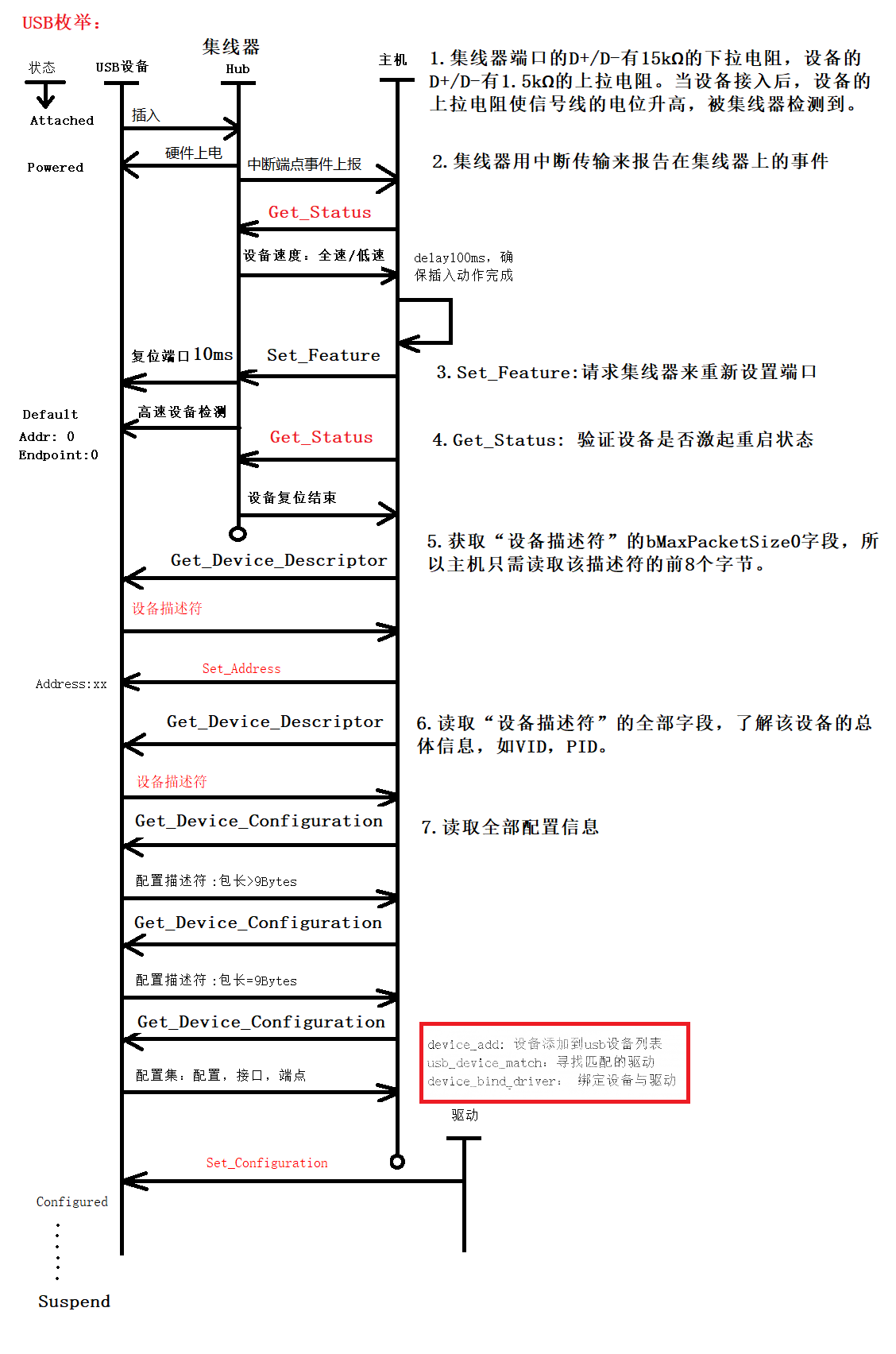
### 4.4 端点描述符

每个设备至少支持控制端点0。USB设备应该支持三类端点：控制端点、输入端点和输出端点。





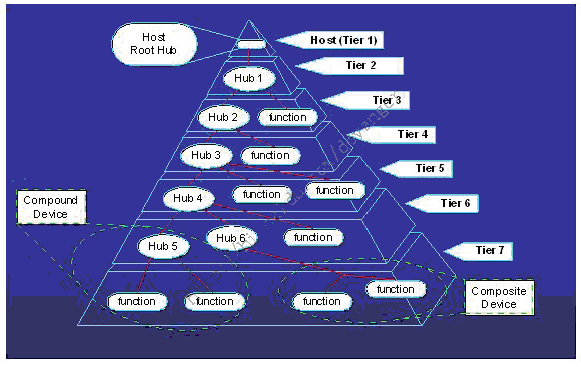
# USB枚举过程



## 1. 枚举是什么?

枚举就是从设备读取一些信息，知道设备是什么样的设备，如何进行通信，这样主机就可以根据这些信息来加载合适的驱动程序。调试USB设备，很重要的一点就是USB的枚举过程，只要枚举成功了，那么就已经成功大半了。

USB架构中， hub负责检测设备的连接和断开，利用其中断IN端点(Interrupt IN Endpoint)来向主机（Host）报告。在系统启动时，主机轮询它的根hub（Root Hub）的状态看是否有设备（包括子hub和子hub上的设备）连接。USB总线拓扑结构见下图（最顶端为主机的Root Hub）：



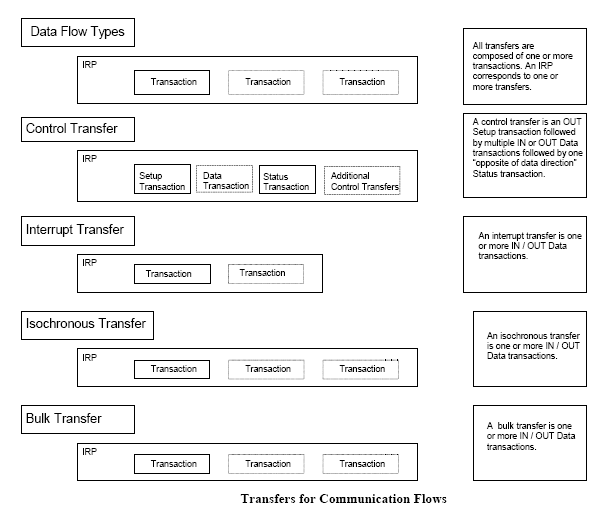
USB总线拓扑结构

一旦获悉有新设备连接上来，主机就会发送一系列的请求(Resqusts)给设备所挂载到的hub，再由hub建立起一条连接主机（Host）和设备（Device）之间的通信通道。然后主机以控制传输(Control Transfer)的方式，通过端点0(Endpoint 0)对设备发送各种请求，设备收到主机发来的请求后回复相应的信息，进行枚举（Enumerate）操作。所有的USB设备必须支持标准请求（StandardRequests），控制传输方式（Control Transfer）和端点0（Endpoint 0）。

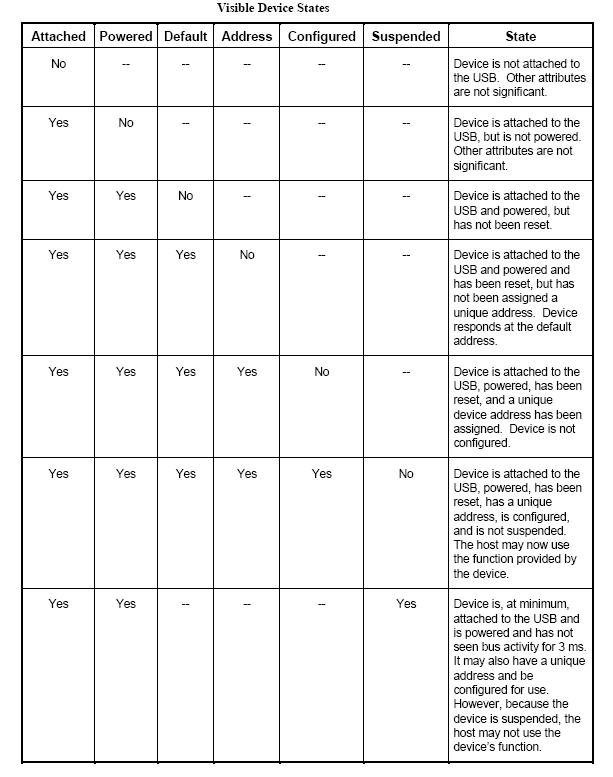
在讲解枚举之前，先大概说说USB的一种传输模式——控制传输。这种传输在USB中是非常重要的，它要保证数据的正确性，在设备的枚举过程中都是使用控制传输的。**控制传输分为三个阶段：①建立阶段。②数据阶段。③确认阶段。**

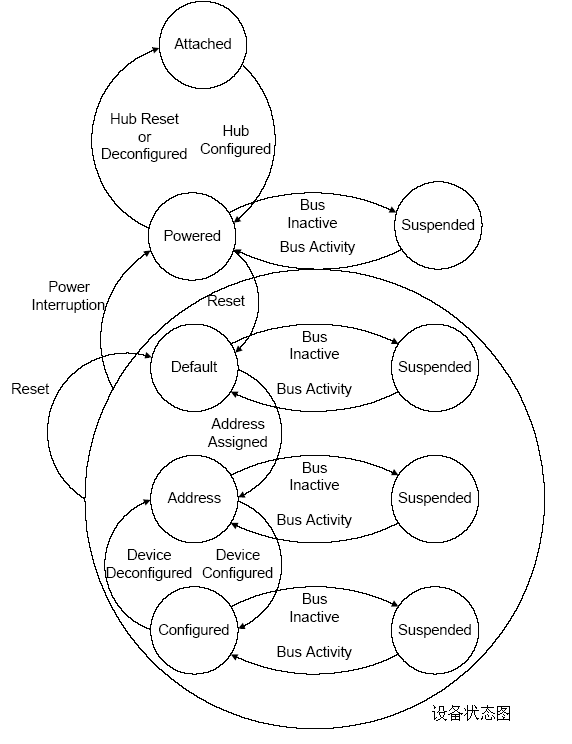
建立（setup）阶段：都是由USB主机发起，它是一个setup数据包，里面包含一些数据请求的命令以及一些数据。如果建立阶段是输入请求，那么数据阶段就要输入数据；如果建立阶段是输出请求，那么数据阶段就要输出数据。如果在数据阶段，即便不需要传送数据，也要发一个0长度的数据包。数据阶段过后就是确认阶段。确认阶段刚好跟数据阶段相反，如果是输入请求，则它是一个输出数据包；如果是输出请求，则它是一个输入数据包。确认阶段用来确认数据的正确传输。

### 1.1 通信传输流



### 1.2 设备状态图





### 1.3 状态详解

**1) 连接(Attached)**

设备可以连接到USB或者从USB上拔出.USB设备从总线上拨出后的状态在规范没定义,只说明一旦USB连到总线要求的操作以及属性.

**2) 上电(Powered)**

USB设备的电源可来自外部电源，也可从USB接口的集线器而来。电源来自外部电源的USB设备被称作自给电源式的(self-powered)。尽管自给电源式的USB设备可能在连接上USB接口以前可能已经带电，但它们直到连线上USB接口后才能被看作是加电状态(Powered state)。而这时候VBUS已经对设备产生作用了.

一个设备可能有既支持自给电源的，同时也支持总线电源式的配置。有一些支持其中的一种，而另一些设备配置可能只有在自给电源下才能被使用。**设备对电源支持的能力是通过配置描述表(configuration descriptor)来反映的**。当前的电源供给形式被作为设备状态的一部分被反映出来。设备可在任何时候改变它们的供电来源，比如说：从自给式向总线式改变，如果一个配置同时支持两种模式，那此状态的最大电源需求就是指设备在两种模式下从VBUS上获取电能的最大值。设备必须以此最大电源作为参照，而究竟处于何状态是不考虑的。如果有一配置仅支持一种电源模式，那么电源模式的改变会使得设备失去当前配置与地址，返回加电状态。如果一个设备是自给电源式，并且当前配置需要大于100mA电流，那么如果此设备转到了总线电源式，它必须返回地址状态(Address state)。自给电源式集线器使用VBUS来为集线控制器(Hub controller)提供电源，因而可以仍然保持配置状态(Configured state)，尽管自给电源停止提供电源。

**3)默认状态(Default)**

设备上电后,它不响应任何总线处理,直到总线接收到复位信号为止.接收到复位信号后,用默认的地址可以对设备寻址.

当用复位过程完成后,USB设备在正确的速度下操作(即低速/全速/高速).低速和全速的数据选择由设备的终端电阻决定.能进行高速操作的设备决定它是否在复位的过程的一部分执行高速操作.

能进行高速操作的设备在全速的电气环境中操作时,必须能以全速成功复位.设备成功复位后,设备必须成功响应设备和配置描述符请求,并且返回适当的信息.当在全速下工作时,设备可能或者不能支持预定义的功能.

**4) 地址(Address)**

所有的USB设备在加电复位以后都使用缺省地址。每一设备在连接或复位后由主机分配一个唯一的地址。当USB设备处于挂起状态时，它保持这个地址不变。

USB设备只对缺省通道(Pipe)请求发生响应，而不管设备是否已经被分配地址或在使用缺省地址。

**5) 配置状态( Configured )**

在USB设备正常工作以前，设备必须被正确配置。从设备的角度来看，配置包括一个将非零值写入设备配置寄存器的操作。配置一个设备或改变一个可变的设备设置会使得与这个相关接口的终端结点的所有的状态与配置值被设成缺省值。这包括将正在使用(date toggle)的结点(end point)的 (Date toggle)被设置成DATA0。

**6) 挂起状态**

为节省电源，USB设备在探测不到总线传输时自动进入中止状态。当中止时，USB设备保持本身的内部状态，包括它的地址及配置。

所有的设备在一段特定的时间内探测不到总线活动时必须进入中止态。不管设备是被分配了非缺省的地址或者是被配置了，已经连接的设备必须在任何加电的时刻随时准备中止。总线活动的中止可能是因为主机本身进入了中止状态。另外，USB设备必须在所连接的集线器端口失效时进入中止态。这就是所指的选择性中止(Selective suspend)。

USB设备在总线活动来到时结束中止态。USB设备也可以远程唤醒的电流信号来请求主机退出中止态或选择性中止态。具体设备具有的远程唤醒的能力是可选的，也就是说，如果一个设备有远程唤醒的能力，此设备必须能让主机控制此能力的有效与否。当设备复位时，远程唤醒能力必须被禁止。

## 2. 枚举步骤

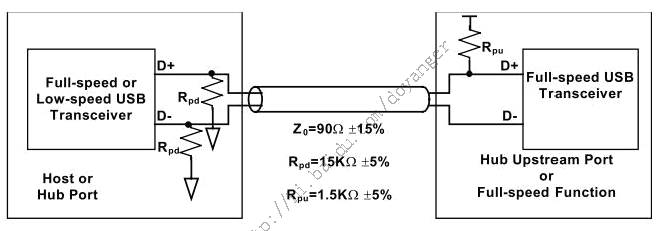
USB协议定义了设备的6种状态，仅在枚举过程中，设备就经历了4个状态的迁移：上电状态(Powered)，默认状态(Default)，地址状态(Address)和配置状态(Configured)（其他两种是连接状态和挂起状态（Suspend））。

### 2.1 用户把USB设备插入USB端口或给系统启动时设备上电

这里指的USB端口指的是主机下的根hub或主机下行端口上的hub端口。Hub给端口供电，连接着的设备处于上电状态。此时，USB设备处于加电状态，它所连接的端口是无效的。

### 2.2 Hub监测它各个端口数据线上(D+/D-)的电压

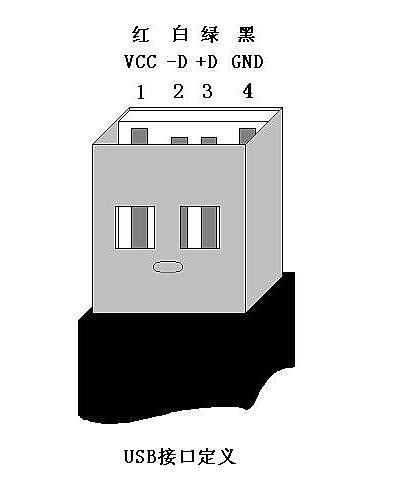
在hub端，数据线D+和D-都有一个阻值在14.25k到24.8k的下拉电阻Rpd，而在设备端，D+（全速，高速）和D-（低速）上有一个1.5k的上拉电阻Rpu。当设备插入到hub端口时，有上拉电阻的一根数据线被拉高到幅值的90%的电压（大致是3V）。**hub检测到它的一根数据线是高电平，就认为是有设备插入**，**并能根据是D+还是D-被拉高来判断到底是什么设备（全速/低速）**插入端口（全速、高速设备的区分在我将来的文章中描述）。如下图。



USB全速/高速设备上电连接

检测到设备后，hub继续给设备供电，但并不急于与设备进行USB传输。

USB接口定义如下图所示：



### 2.3 Host了解连接的设备

每个hub利用它自己的中断端点向主机报告它的各个端口的状态（对于这个过程，设备是看不到的，也不必关心），报告的内容只是hub端口的设备连接／断开的事件。如果有连接／断开事件发生，那么host会发送一个 **Get\_Port\_Status**请求(request)给hub以了解此次状态改变的确切含义。Get\_Port\_Status等请求属于所有hub都要求支持的hub类标准请求（standard hub-class requests）。

### 2.4 Hub检测所插入的设备是高速还是低速设备

hub通过检测USB总线空闲(Idle)时差分线的高低电压(D+/D-)来判断所连接设备的速度类型，当host发来Get\_Port\_Status请求时，hub就可以将此设备的速度类型信息回复给host。USB 2.0规范要求速度检测要先于复位（Reset）操作。

### 2.5 hub复位设备

主机一旦得知新设备已连上以后，它至少等待100ms以使得插入操作的完成以及设备电源稳定工作。然后主机控制器就向hub发出一个 **Set\_Port\_Feature**请求让hub复位其管理的端口(刚才设备插上的端口)。hub通过驱动数据线到复位状态(D+和D-全为低电平 )，并持续至少**10ms**。**当然，hub不会把这样的复位信号发送给其他已有设备连接的端口，所以其他连在该hub上的设备自然看不到复位信号，不受影响。**

### 2.6 Host检测所连接的全速设备是否是支持高速模式

因为根据USB 2.0协议，高速（High Speed）设备在初始时是默认全速（Full Speed ）状态运行，所以对于一个支持USB 2.0的高速hub，当它发现它的端口连接的是一个全速设备时，会进行高速检测，看看目前这个设备是否还支持高速传输，如果是，那就切到高速信号模式，否则就一直在全速状态下工作。

同样的，从设备的角度来看，如果是一个高速设备，在刚连接bub或上电时只能用全速信号模式运行（根据USB 2.0协议，高速设备必须向下兼容USB 1.1的全速模式）。随后hub会进行高速检测，之后这个设备才会切换到高速模式下工作。假如所连接的hub不支持USB 2.0，即不是高速hub，不能进行高速检测，设备将一直以全速工作。

### 2.7 Hub建立设备和主机之间的信息通道

主机不停地向hub发送Get\_Port\_Status请求，以查询设备是否复位成功。Hub返回的报告信息中有专门的一位用来标志设备的复位状态。

当hub撤销了复位信号，设备就处于默认／空闲状态（Default state），准备接收主机发来的请求。设备和主机之间的通信通过控制传输，**默认地址0，端点号0进行**。此时，设备能从总线上得到的最大电流是100mA。(**所有的USB设备在总线复位后其地址都为0，这样主机就可以跟那些刚刚插入的设备通过地址0通信。**)

### 2.8 主机发送Get\_Descriptor请求获取默认管道的最大包长度

默认管道（Default Pipe）在设备一端来看就是端点0。主机此时发送的请求是**默认地址0，端点0**，虽然所有未分配地址的设备都是通过地址0来获取主机发来的请求，但由于**枚举过程不是多个设备并行处理，而是一次枚举一个设备的方式进行**，所以不会发生多个设备同时响应主机发来的请求。

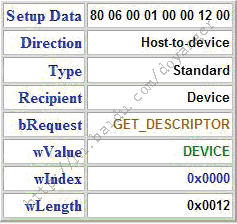
设备描述符的第8字节代表设备端点0的最大包大小。虽然说设备所返回的设备描述符（Device Descriptor）长度只有18字节，但系统也不在乎，此时，描述符的长度信息对它来说是最重要的，其他的瞄一眼就过了。当完成第一次的控制传输后，也就是完成控制传输的状态阶段，系统会要求hub对设备进行再一次的复位操作（USB规范里面可没这要求）。再次复位的目的是使设备进入一个确定的状态。

### 2.9 主机给设备分配一个地址

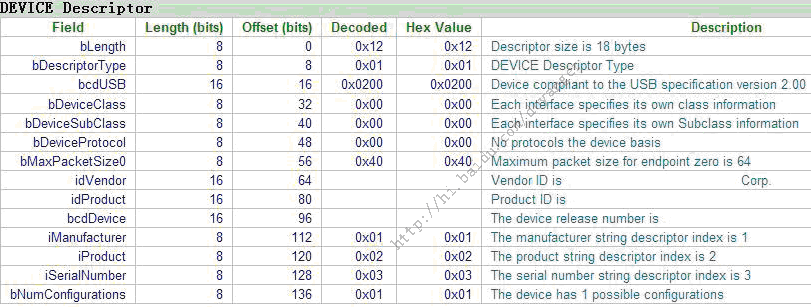
主机控制器通过Set\_Address请求向设备分配一个唯一的地址。在完成这次传输之后，设备进入地址状态（Address state），之后就启用新地址继续与主机通信。这个地址对于设备来说是终生制的，设备在，地址在；设备消失（被拔出，复位，系统重启），地址被收回。同一个设备当再次被枚举后得到的地址不一定是上次那个了。

### 2.10 主机获取设备的信息

主机发送 **Get\_Descriptor**请求到**新地址**读取**设备描述符**，这次主机发送Get\_Descriptor请求可算是诚心，它会认真解析设备描述符的内容。设备描述符内信息包括端点0的最大包长度，设备所支持的配置（Configuration）个数，设备类型，VID（Vendor ID，由USB-IF分配）， PID（Product ID，由厂商自己定制）等信息。Get\_Descriptor请求(Device type)和设备描述符（已抹去VID，PID等信息）见下图：



标准Get\_Descriptor请求



设备描述符（Device Descriptor）

之后主机发送**Get\_Descriptor**请求，读取**配置描述符**（Configuration Descriptor），字符串等，逐一了解设备更详细的信息。事实上，对于配置描述符的标准请求中，有时**wLength**一项会**大于**实际配置描述符的长度（**9字节**），比如255。这样的效果便是：主机发送了一个Get\_Descriptor\_Configuration 的请求，设备会把接口描述符，端点描述符等后续描述符一并回给主机，主机则根据描述符头部的标志判断送上来的具体是何种描述符。

接下来，主机就会获取配置描述符。配置描述符总共为9字节。主机在获取到配置描述符后，根据里面的配置集合总长度，再获取配置集合。配置集合包括**配置描述符，接口描述符，端点描符**等等。

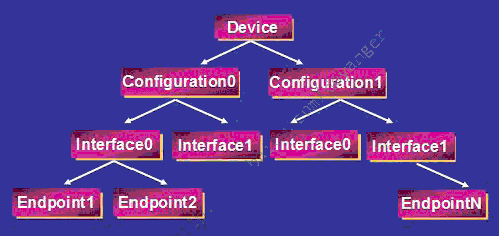
如果有**字符串描述符**的话，还要获取字符串描述符。另外HID设备还有HID描述符等。

### 2.11 主机给设备挂载驱动（复合设备除外）

主机通过解析描述符后对设备有了足够的了解，会选择一个最合适的驱动给设备。 然后tell the world（announce\_device）说明设备已经找到了，最后调用设备模型提供的接口device\_add将设备添加到 usb 总线的设备列表里，然后 usb总线会遍历驱动列表里的每个驱动，调用自己的 match（usb\_device\_match） 函数看它们和你的设备或接口是否匹配，匹配的话调用device\_bind\_driver函数，现在就将控制权交到设备驱动了。

对于复合设备，通常应该是不同的接口（Interface）配置给不同的驱动，因此，需要等到当设备被配置并把接口使能后才可以把驱动挂载上去。

设备-配置-接口-端点关系见下图：



USB 设备-配置-接口-端点关系

实际情况没有上述关系复杂。一般来说，一个设备就一个配置，一个接口，如果设备是多功能符合设备，则有多个接口。端点一般都有好几个，比如Mass Storage设备一般就有两个端点（控制端点0除外）。

### 2.12 设备驱动选择一个配置

驱动（注意，这里是驱动，之后的事情都是有驱动来接管负责与设备的通信）根据前面设备回复的信息，发送Set\_Configuration请求来正式确定选择设备的哪个配置（Configuration）作为工作配置（对于大多数设备来说，一般只有一个配置被定义）。至此，设备处于配置状态(Configured)，当然，设备也应该使能它的各个接口（Interface）。

对于复合设备，主机会在这个时候根据设备接口信息，给它们挂载驱动。

# 3. 控制传输

## 控制传输解析

控制传输命令：8字节

**第一个字节:**

**bmRequestType :**

0x80/0x81/0x82 : Device-to-host | Standard | Device/ Interface/ Endpoint //Get Standard

0x00/0x01/0x02 : Host-to-device | Standard | Device/ Interface/ Endpoint //Set Standard

0xA0/0xA1/0xA2 : Device-to-host | Class | Device/ Interface/ Endpoint //Get Class

0x20/0x21/0x22 : Host-to-device | Class | Device/ Interface/ Endpoint //Set Class

**第二个字节: bRequest：Standrad**

GET\_STATUS(0x00)

CLEAR\_FEATURE(0x01) / SET\_FEATURE(0x03)

SET\_ADDRESS(0x05) -- **bmRequestType:** Device

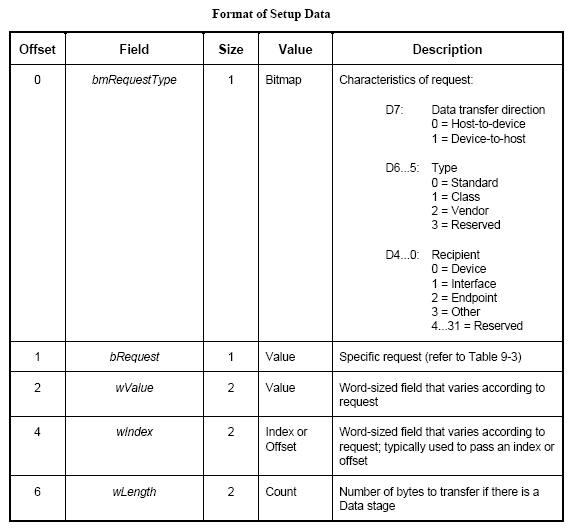
GET\_DESCRIPTOR(0x06) / SET\_DESCRIPTOR(0x07) – **bmRequestType:** Device

GET\_CONFIGURATION(0x08) / SET\_CONFIGURATION(0x09) – **bmRequestType:** Device

GET\_INTERFACE(0x0A) / SET\_INTERFACE(0x0B) – **bmRequestType:** Interface

**第二个字节: bRequest：Class**

控制传输是最重要和最复杂的一种传输类型，其中使用了一个8字节大小的DATA0数据包，这8个字节的数据包是主机用来发送**控制阶段**中的请求命令，而这些请求命令是主机配置USB设备的关键。



**1) bmRequestType(向谁请求)**

D7：传输方向

0=主机至设备；1=设备至主机

D6..5：命令类型

D6D5=00：标准请求命令； D6D5=01：类请求命令；

D6D5=10：用户定义命令； D6D5=11：保留。

D4..0：接受者类型

0=设备；1=接口 ；

2=端点；3=其它

4..31 保留

这个域表明此请求的特性。特别地，**这个域表明了第二阶段控制传输方向**。如果wLength域被设作0的话，表明没有数据传送阶段，那Direction位就会被忽略.

USB说明定义了一系列所有设备必须支持的标准请求。这些请求见下面的表<Standard Device Requests>。另外，一个设备类可定义更多的请求。设备厂商也可定义设备支持的请求.

请求可被导引到设备，设备接口，或某一个设备端结点(endpoint)上。这个请求域也指定了接收者。当指定的是**接口或端结点(endpoint)时，wIndex域指出那个接口或端节点**。

**2) bmRequest(什么请求)**

这个域标识特别的请求。bmRequestType域的Type字段可修改此域的含义。本文仅定义Type 字段为0即标准设备请求时bRequest域值的含义。

**3) wValue域**

此域用来传送当前请求的参数，随请求不同而变。

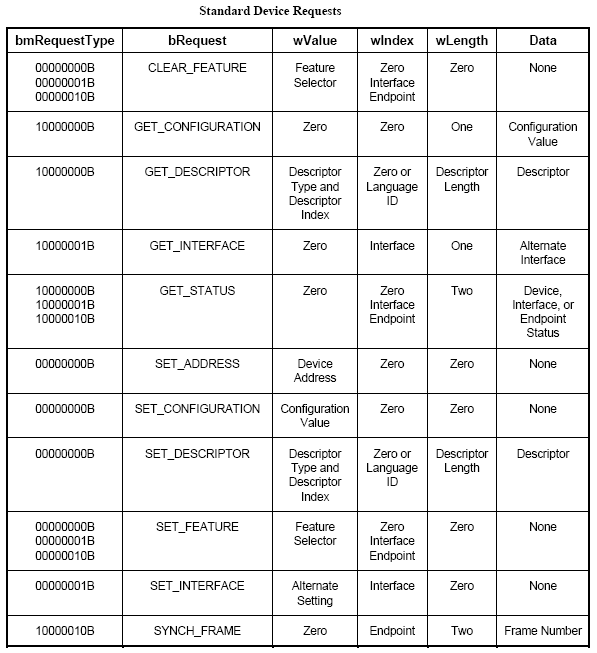
**4) wIndex域**

当bmRequestType的Recipient字段为接口或端点时，wIndex域用来表明是哪一个接口或端结。

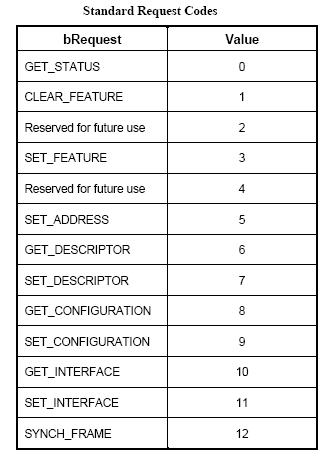
**5) wLength域**

这个域表明**第二阶段的数据传输长度**。传输方向由bmRequstType域的Direction位指出。wLength域为0则表明无数据传输。在输入请求下，设备返回的数据长度不应多于wLength，但可以少于。在输出请求下，wLength指出主机发出的确切数据量。如果主机发送多于wLength的数据，设备做出的响应是无定义的。

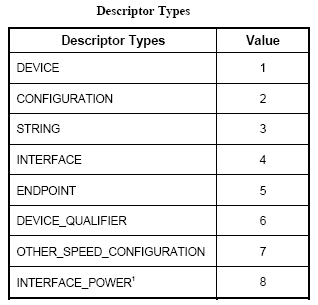
### 3.1 标准设备请求(bmRequestType)



### 3.2 标准请求码(bRequest)

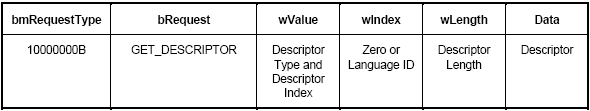


### 3.3 描述符类型(wValue)



### 3.4 GET\_DESCRIPTOR(读取描述符)

这个请求返回存在的**描述符**.



**wValue域：**

高一字节：标识描述表类型(Descriptor Types)

低一字节：表示描述符号的索引，将几个类型相同的描述符应用到设备中时，描述符索引用于选择一个特定的描述符（只能是**配置和字符串**描述符）。

例如：设备可以用几个配置描述符。对于其他可以通过GetDescriptor()请求取得的标准描述符来说，描述符的索引须为0。描述符索引值的范围从0开始到设备使用该类型描述符的数量减1。

**wIndex域：**

标识字串描述表的语言(Language ID)，如果是其它语言的话就设为0。

**wLength域：**

表示要返回多少字节。如果描述表长度大于wLength域值，那么只有描述表的初始部分被返回。如果描述表比wLength域值小，则发送一个短包来标志传输的结束。一个短包被定义成一个长度短于最大负载长度或一个空(NULL)包。

这个标准请求包括3种描述符：**设备(也就是设备的限定符)、配置(也就是其他速度配置)、及字符串**。能进行高速操作的设备支持设备限定描述符返回有关设备不支持的速度信息(包括默认端点的wMaxPacketSize和其他速度配置的数量)。

其他速度配置用与配置描述符相同的结构返回信息，但如果设备在其他速度下操作则返回配置信息。请求配置描述符将返回配置描述符，所有接口描述符和在这个请求中所有接口的端点描述符。

第一个接口描述符的设备请求会一次返回配置描述表，所有的接口描述表和所有接口的端节点的描述表。第一个接口描述符紧跟着配置描述符号，第一个接口的端节点的描述符号随后。如果有其它的接口与端节点，它们的描述符欲跟在第一个接口与端节点描述符之后。与类有关的描述符，和/或厂商定义的描述符跟在标准描述符之后.

所有的设备必须提供一个设备描述符并且至少一个配置描述符，如果一个设备不支持一个请求的描述符，则返回请求错误。

•缺省状态：此请求合法。

•地址状态：此请求合法。

•配置状态：此请求合法。

### 3.5 GET\_INTERFACE(取得接口)

这个请求向指定接口返回选中的备用设备。

一些USB设备有接口设置互斥的配置。这个请求允许主机确定当前选定的备用设置。如果wValue或者wLength的值与上面指定的不一致，那么设备的行为没有定义；如果指定的接口不存在，那么设备将用请求错误响应。

•默认状态：当设备处于默认状态时接收到这个请求，设备的行为没 有定义

•地址状态：设备给出请求错误

•配置状态：当设备处于配置状态时，这是一个有效的请求

# USB四大数据结构

主要有四个数据结构，分别是：

**USB设备：usb\_device**，保存了一个USB设备的信息，包括设备地址，设备描述符，配置描述符，等等

**USB总线系统：usb\_bus**，保存了一个USB总线系统的信息，包括总线上设备地址信息，根集线器，带宽使用情况等。一个USB总线系统肯定有一个主机控制器和一个根集线器。Linux支持多USB总线系统

**客户端驱动程序：usb\_driver**，保存了客户驱动信息，包括驱动名称，以及驱动提供给USB内核使用的函数指针等

**（USB Request Block）urb**：是进行USB通信的数据结构。Linux的USB子系统只使用这么一种数据结构来进行USB通信，urb包含了建立任何 USB传输所需的所有信息，并贯穿于USB协议栈对数据处理的整个过程。

## usb\_device

一个usb\_device代表一个USB设备。

|  |
| --- |
| struct usb\_device {  int devnum; //分配的设备地址，1-127  char devpath[16];  u32 route;  enum usb\_device\_state state; //设备状态，ATTACHED, POWERED, DEFAULT, ADDRESS, CONFIGURED, SUSPENDED  enum usb\_device\_speed speed; //设备速度，低速/全速/高速  struct usb\_tt \*tt; //事务处理解释器  int ttport; //设备所连接的具有事务处理解释器功能的集线器端口  unsigned int toggle[2]; //用于同步切换的位图，每个端点占用1位，[0]表示输入，[1]输出  struct usb\_device \*parent; //表示设备所连的上游集线器指针  struct usb\_bus \*bus; //设备所属的USB总线系统  struct usb\_host\_endpoint ep0;  struct device dev;  struct usb\_device\_descriptor descriptor; //设备描述符  struct usb\_host\_bos \*bos;  struct usb\_host\_config \*config; //指向设备的配置描述符和其所包含的接口描述符，端点描述符的指针  struct usb\_host\_config \*actconfig; //当前的配置描述符指针  struct usb\_host\_endpoint \*ep\_in[16];  struct usb\_host\_endpoint \*ep\_out[16];  char \*\*rawdescriptors; //Raw descriptors for each config  unsigned short bus\_mA;  u8 portnum;  u8 level;  unsigned can\_submit:1;  unsigned persist\_enabled:1;  unsigned have\_langid:1; //whether string\_langid is valid yet  unsigned authorized:1;  unsigned authenticated:1;  unsigned wusb:1;  unsigned lpm\_capable:1;  unsigned usb2\_hw\_lpm\_capable:1;  unsigned usb2\_hw\_lpm\_besl\_capable:1;  unsigned usb2\_hw\_lpm\_enabled:1;  unsigned usb2\_hw\_lpm\_allowed:1;  unsigned usb3\_lpm\_enabled:1;  int string\_langid; //language ID for strings  /\* static strings from the device \*/  char \*product;  char \*manufacturer;  char \*serial;  struct list\_head filelist; //Child devices, 只对当前设备是集线器的情况有效  int maxchild; //hub的下游端口数  u32 quirks;  atomic\_t urbnum;  unsigned long active\_duration;  #ifdef CONFIG\_PM  unsigned long connect\_time;  unsigned do\_remote\_wakeup:1;  unsigned reset\_resume:1;  unsigned port\_is\_suspended:1;  #endif  struct wusb\_dev \*wusb\_dev;  int slot\_id;  enum usb\_device\_removable removable;  struct usb2\_lpm\_parameters l1\_params;  struct usb3\_lpm\_parameters u1\_params;  struct usb3\_lpm\_parameters u2\_params;  unsigned lpm\_disable\_count;  } |

### 数据结构

1) USB设备：对应数据结构**struct** **usb**\_**device**

2) 配置：**struct** **usb\_host\_config** （任一时刻，只能有一个配置生效）

3) USB接口：**struct** **usb**\_**interface** （USB 核心将其传递给USB设备驱动，并由USB设备驱动负责后续的控制。一个USB接口代表一个基本功能，每个USB驱动控制一个接口。所以一个物理上的硬件设备可能需要 一个以上的驱动程序。）

4) 端点: **struct** **usb**\_**host**\_**endpoint** ，它所包含的真实端点信息在另一个结构中：**struct** **usb**\_**endpoint**\_**descriptor**（端点描述符，包含所有的USB特定数据）。

### USB 端点

USB 端点有 4 种不同类型, 分别具有不同的数据传送方式：

**1) 控制CONTROL**

每个 USB 设备都有一个控制端点称为"端点 0", 被 USB 核心用来在插入时配置设备。USB协议保证总有足够的带宽留给控制端点传送数据到设备.

**2)中断INTERRUPT**

每当 USB 主机向设备请求数据时，中断端点以固定的速率传送小量的数据。此为USB 键盘和鼠标的主要的数据传送方法。它还用以传送数据到USB设备来控制设备。通常不用来传送大量数据。USB协议保证总有足够的带宽留给中断端点传送数据到设备.

**3) 批量BULK**

批量端点用以传送大量数据。这些端点通常比中断端点大得多. 它们普遍用于不能有任何数据丢失的情况。USB 协议不保证传输在特定时间范围内完成。如果总线上没有足够的空间来发送整个BULK包，它被分为多个包进行传输。这些端点普遍用于打印机、USB Mass Storage和USB网络设备上。

**4) 等时ISOCHRONOUS**

等时端点也批量传送大量数据, 但是这个数据不被保证能送达。这些端点用在可以处理数据丢失的设备中，并且更多依赖于保持持续的数据流。如音频和视频设备等等。

控制和批量端点用于异步数据传送，而中断和等时端点是周期性的。这意味着这些端点被设置来在固定的时间连续传送数据，USB 核心为它们保留了相应的带宽。

当调用USB设备驱动调用usb\_submit\_urb提交urb请求时，将调用int usb\_hcd\_link\_urb\_to\_ep(struct usb\_hcd \*hcd, struct urb \*urb)把此urb增加到urb\_list的尾巴上。(hcd: Host Controller Driver,对应数据结构struct usb\_hcd )

|  |
| --- |
| **struct** **usb\_host\_endpoint** {  **struct** **usb**\_**endpoint**\_**descriptor** desc;  **struct** **usb**\_**ss**\_**ep**\_**comp**\_**descriptor** ss\_ep\_comp;  **struct** **list**\_**head** urb\_list;  void \*hcpriv;  **struct** **ep**\_**device** \*ep\_dev; /\* For sysfs info \*/  unsigned char \*extra; /\* Extra descriptors \*/  int extralen;  int enabled;  }; |

## usb\_bus

一个usb\_bus代表一个USB总线系统

|  |
| --- |
| **struct** **usb**\_**bus** {  **struct** **device** \***controller**; /\* host/master side hardware \*/  int **busnum**; //当前总线系统的序列号，Linux支持多总线系统并为它们编号  const char \*bus\_name; /\* stable id (PCI slot\_name etc) \*/  u8 uses\_dma; /\* Does the host controller use DMA? \*/  u8 uses\_pio\_for\_control;/\* Does the host controller use PIO for control transfers? \*/  u8 otg\_port; /\* 0, or number of OTG/HNP port \*/  unsigned is\_b\_host:1; /\* true during some HNP roleswitches \*/  unsigned b\_hnp\_enable:1; /\* OTG: did A-Host enable HNP? \*/  unsigned no\_stop\_on\_short:1; /\* Quirk: some controllers don't stop the ep queue on a short transfer with the URB\_SHORT\_NOT\_OK flag set. \*/  unsigned no\_sg\_constraint:1; /\* no sg constraint \*/  unsigned sg\_tablesize; /\* 0 or largest number of sg list entries \*/  int devnum\_next; /\* Next open device number in round-robin allocation \*/  **struct** **usb**\_**devmap** **devmap**; /\* device address allocation map \*///给连接到子系统上的设备分配设备号的数据结构  **struct** **usb**\_**device** \***root**\_**hub**; /\* Root hub \*/ //指向根Hub的指针  **struct** **usb**\_**bus** \***hs**\_**companion**; /\* Companion EHCI bus, if any \*/  **struct** **list**\_**head** **bus**\_**list**; /\* list of busses \*/ //双向链表指针，USB内核用一个双向链表来维护系统中所有USB总线系统  int **bandwidth**\_**allocated**; /\* on this bus: how much of the time reserved for periodic (intr/iso) requests is used, on average? Units: microseconds/frame. Limits: Full/low speed reserve 90%, while high speed reserves 80%.\*/ //当前子系统的带宽使用情况，单位是毫秒/帧，取值范围[0,900]  int **bandwidth**\_**int**\_**reqs**; /\* number of Interrupt requests \*/ //子系统中当前的中断传输的数量  int **bandwidth**\_**isoc**\_**reqs**; /\* number of Isoc. requests \*/ //子系统中当前的实时传输的数量  unsigned resuming\_ports; /\* bit array: resuming root-hub ports \*/  #if defined(CONFIG\_USB\_MON) || defined(CONFIG\_USB\_MON\_MODULE)  **struct** **mon**\_**bus** \*mon\_bus; /\* non-null when associated \*/  int monitored; /\* non-zero when monitored \*/  #endif  } |

## usb\_driver

客户端驱动程序为USB内核提供的调用接口

|  |
| --- |
| **struct** **usb**\_**driver** {  const char \***name**; //客户端驱动程序的字符串名称，用于避免重复安装和卸载  int (\***probe**) (struct usb\_interface \*intf,const struct usb\_device\_id \*id); //给USB内核提供的函数，用于判断驱动程序是否能对设备的某个接口进行驱动，如能则分配资源  void (\***disconnect**) (struct usb\_interface \*intf); //给USB内核提供的函数，用于释放设备的某个接口所占用的资源  int (\*unlocked\_ioctl) (struct usb\_interface \*intf, unsigned int code,void \*buf);  int (\*suspend) (struct usb\_interface \*intf, pm\_message\_t message);  int (\*resume) (struct usb\_interface \*intf);  int (\*reset\_resume)(struct usb\_interface \*intf);  int (\*pre\_reset)(struct usb\_interface \*intf);  int (\*post\_reset)(struct usb\_interface \*intf);  const **struct** **usb**\_**device**\_**id** \*id\_table;  **struct** **usb**\_**dynids** dynids;  **struct** **usbdrv**\_**wrap** drvwrap;  unsigned int no\_dynamic\_id:1;  unsigned int supports\_autosuspend:1;  unsigned int disable\_hub\_initiated\_lpm:1;  unsigned int soft\_unbind:1;  } |

## Urb

USB Request Block，包含了建立任何 USB传输所需的所有信息，并贯穿于USB协议栈对数据处理的整个过程

|  |
| --- |
| **struct** **urb** {  /\* private: usb core and host controller only fields in the urb \*/  **struct** **kref** **kref**; /\* reference count of the URB \*/ //与主机控制器相关数据，对USB内核层是透明  void \***hcpriv**; /\* private data for host controller \*/  atomic\_t use\_count; /\* concurrent submissions counter \*/  atomic\_t reject; /\* submissions will fail \*/  int unlinked; /\* unlink error code \*/  /\* public: documented fields in the urb that can be used by drivers \*/  **struct** **list**\_**head** **urb**\_**list**; /\* list head for use by the urb's current owner \*/ //双向指针，用于将此URB连接到处于活动的URB双向链表中  **struct** **list**\_**head** **anchor**\_**list**; /\* the URB may be anchored \*/  **struct** **usb**\_**anchor** \*anchor;  **struct** **usb**\_**device** \*dev; /\* (in) pointer to associated device \*/ // 接受此URB的USB设备指针  **struct** **usb**\_**host**\_**endpoint** \*ep; /\* (internal) pointer to endpoint \*/  unsigned int **pipe**; /\* (in) pipe information \*/ //表示设备的某个端点和客户端驱动程序之间的管道  unsigned int stream\_id; /\* (in) stream ID \*/  int **status**; /\* (return) non-ISO status \*/ //返回状态  unsigned int **transfer**\_**flags**; **//见下面的详细解释**  void \***transfer**\_**buffer**; /\* (in) associated data buffer \*/ //传输数据缓存区指针，接收或发送设备的数据，它必须是物理连续的，不可换页的内存块，用kmalloc（，GFP\_KERNEL）分配  dma\_addr\_t **transfer**\_**dma**; /\* (in) dma addr for transfer\_buffer \*/  struct scatterlist \*sg; /\* (in) scatter gather buffer list \*/  int num\_mapped\_sgs; /\* (internal) mapped sg entries \*/  int num\_sgs; /\* (in) number of entries in the sg list \*/  u32 **transfer**\_**buffer**\_**length**; /\* (in) data buffer length \*/ //缓存区长度  u32 **actual**\_**length**; /\* (return) actual transfer length \*/ //实际数据长度  unsigned char \***setup**\_**packet**; /\* (in) setup packet (control only) \*/ //用于指向控制传输中控制命令的指针，只适用控制传输  dma\_addr\_t **setup**\_**dma**; /\* (in) dma addr for setup\_packet \*/  int **start**\_**frame**; /\* (modify) start frame (ISO) \*/ //此请求所开始传输的帧号，只适用实时/中断传输。中断传输时，表示返回启动此请求的第一次中断传输的帧号。实时传输时，指明处理第一个实时请求数据报包的帧号，如果设置了USB\_ISO\_ASAP，此变量表示返回启动第一次实时传输的帧号。  int **number**\_**of**\_**packets**; /\* (in) number of ISO packets \*/ //此请求所包含的数据包数，只适合实时传输  int **interval**; /\* (modify) transfer interval ，1〈= interval〈=255 (INT/ISO) \*/ //中断传输的周期  int **error**\_**count**; /\* (return) number of ISO errors \*/ //发生传输错误次数的累加值，只适用实时传输  void \***context**; /\* (in) context for completion \*/ //回调函数中的参数  usb\_complete\_t **complete**; /\* (in) completion routine \*/ //指向回调函数的指针。当数据传输完成后，主机控制器驱动会回调该函数  **struct** **usb**\_**iso**\_**packet**\_**descriptor** **iso**\_**frame**\_**desc**[0]; /\*(in) ISO ONLY \*/ //要进行实时传输的结构数组，每个结构表示一次数据传输  } |

### transfer\_flags;

|  |
| --- |
| #define URB\_SHORT\_NOT\_OK 0x0001 /\* report short reads as errors \*/  #define URB\_ISO\_ASAP 0x0002 /\* iso-only; use the first unexpired slot in the schedule \*/  #define URB\_NO\_TRANSFER\_DMA\_MAP 0x0004 /\* urb->transfer\_dma valid on submit \*/  #define URB\_NO\_FSBR 0x0020 /\* UHCI-specific \*/  #define URB\_ZERO\_PACKET 0x0040 /\* Finish bulk OUT with short packet \*/  #define URB\_NO\_INTERRUPT 0x0080 /\* HINT: no non-error interrupt needed \*/  #define URB\_FREE\_BUFFER 0x0100 /\* Free transfer buffer with the URB \*/ |

**URB**\_**ISO**\_**ASAP** //用于实时传输，告诉主机控制器立即进行此请求的数据传输。如果没有置位，则需要给start\_frame赋值，用来通知主机控制器该在哪个帧上开始此请求的数据传输

**URB**\_**NO**\_**FSBR** //表示全速传输站用的带宽不要回收

**URB**\_**ZERO**\_**PACKET** //表示批量传输的数据长度等于端点的包最大长度时，主机控制器在发送完数据后，再发送一个零长度的包表示数据结束

**URB\_SHORT\_NOT\_OK //针对IN端口**

端点描述符里的wMaxPacketSize指明了端点一次能够处理的最大字节数。这个最大字节数只包括了Data 包里面数据字段，俗称data payload。如果从IN 端点那儿收到了一个比wMaxPacketSize 要短的包，同时也设置了URB\_SHORT\_NOT\_OK 这个标志，那么就可以认为传输出错了。如果端点想给你的数据大于wMaxPacketSize的大小，就需要分成多个wMaxPacketSize 大小的data payload 来传输。这时，最后一个data payload 的长度就会比wMaxPacketSize 要小，这种情况本来意味着端点已经传完了它想传的，释放完了自己的需求，这次传输就该结束了，不过如果你设置了URB\_SHORT\_NOT\_OK 标志，HCD 这边就会认为错误发生了。

**URB\_ISO\_ASAP //针对于等时传输**

等时传输和中断传输在spec 里都被认为是periodic transfers，也就是周期传输，咱们都知道在usb 的世界里都是主机占主导地位，设备是没多少发言权的，但是对于等时传输和中断传输，端点可以对主机表达自己一种美好的期望，希望主机能够隔多长时间访问自己一次，这个期望的时间就是这里说的周期。当然，期望与现实是有一段距离的。端点的这个期望能不能得到满足，要看主机控制器答应不答应。对于等时传输，一般来说也就一帧（微帧）一次，主机那儿也很忙，再多也抽不出空儿来。那么**如果你有个用于等时传输的urb，你提交给HCD 的时候，就得告诉HCD 它应该从哪一帧开始的，就要对下面要说的那个start\_frame 赋值**，也就是说告诉HCD 等时传输开始的那一帧（微帧）的帧号，如果你留心，应该还会记得前面说过在每帧或微帧（Mircoframe）的开始都会有个SOF Token 包，这个包里就含有个帧号字段，记录了那一帧的编号。这样的话，一是比较烦，还要去设置这个start\_frame，你说烦不烦，二是到你设置的那一帧的时候，如果主机控制器没空开始等时传输，你说怎么办，要知道usb 的世界里它可是老大。于是，就出现了URB\_ISO\_ASAP，它的意**思就是告诉HCD 啥时候不忙就啥时候开始，就不用指定什么开始的帧号了**，是不是感觉特轻松？所以说，你如果想进行等时传输，又不想标新立异的话，就还是把它给设置了吧。

**URB\_NO\_TRANSFER\_DMA\_MAP & URB\_NO\_SETUP\_DMA\_MAP**

一般来说，都是驱动里提供了kmalloc 等分配的缓冲区，HCD 做一定的DMA 映射处理，DMA 映射是干吗的？外设和内存之间进行数据交换，总要互相认识吧，外设是通过各种总线连到主机里边儿的，使用的是总线地址，而内存使用的是虚拟地址，它们之间本来就是两条互不相交的平行线，要让它们中间产生连接点，必须得将一个地址转化为另一个地址，这样才能找得到对方，才能互通有无，而DMA 映射就是干这个的。为了分担点HCD 的压力，于是就有了这里的两个标志，**告诉HCD 不要再自己做DMA 映射了，驱动提供的urb 里已经提供有DMA 缓冲区地址**，为领导分忧解难是咱们这些小百姓应该做的事情。具体提供了哪些DMA 缓冲区？就涉及到下面的transfer\_buffer，transfer\_dma，还有setup\_packet，setup\_dma 这两对儿了。

**URB\_ZERO\_PACKET**

这个标志表示批量的OUT 传输必须使用一个short packet 来结束。**批量传输的数据大于批量端点的wMaxPacketSize 时，需要分成多个Data 包来传输，最后一个data payload 的长度等于wMaxPacketSize 时，如果同时设置了URB\_ZERO\_PACKET 标志，就需要再发送一个长度为0 的数据包来结束这次传输，如果小于wMaxPacketSize 就没必要多此一举了。**你要问，当批量传输的数据小于wMaxPacketSize 时那？也没必要再发送0 长的数据包，因为此时发送的这个数据包本身就是一个short packet。

**URB\_NO\_INTERRUPT**

这个标志用来告诉HCD，**在URB 完成后，不要请求一个硬件中断**，当然这就意味着你的结束处理函数可能不会在urb 完成后立即被调用，而是在之后的某个时间被调用，咱们的usb core 会保证为每个urb 调用一次结束处理函数。

### transfer\_buffer & transfer\_dma & transfer\_buffer\_length

**transfer\_buffer** 是使用kmalloc 分配的缓冲区，**transfer**\_**dma** 是使用usb\_buffer\_alloc分配的dma 缓冲区，**transfer**\_**buffer**\_**length**指的就是transfer\_buffer 或transfer\_dma 的长度。

HCD 不会同时使用transfer\_buffer和transfer\_dma。如果你的**urb** 自带了transfer\_dma，就要同时设置URB\_NO\_TRANSFER\_DMA\_MAP 来告诉HCD 一声，不用它再费心做DMA 映射了。transfer\_buffer 是必须要设置的，因为不是所有的主机控制器都能够使用DMA 的，万一遇到这样的情况，也好有个备用。

### setup\_packet & setup\_dma

同样是两个缓冲区，一个是kmalloc分配的，一个是用usb\_buffer\_alloc分配的，不过，这两个缓冲区是**控制传输专用**的，记得struct usb\_ctrlrequest不？它们保存的就是一个struct usb\_ctrlrequest结构体，如果你的urb设置了setup\_dma，同样要设置URB\_NO\_SETUP\_DMA\_MAP标志来告诉HCD。如果进行的是控制传输，setup\_packet是必须要设置的，也是为了防止出现主机控制器不能使用DMA的情况。

### start\_frame

如果你没有指定URB\_ISO\_ASAP 标志，就必须自己设置start\_frame，指定等时传输在哪帧或微帧开始。如果指定了URB\_ISO\_ASAP，urb 结束时会使用这个值返回实际的开始帧号。

### interval

等时和中断传输专用。interval 间隔时间的意思，就是上面说的端点希望主机轮询自己的时间间隔。这个值和端点描述符里的bInterval 是一样的，你不能随便儿的指定一个，协议里对你能指定的值是有范围限制的，对于中断传输，全速时，这个范围为1~255ms，低速是为10~255ms，高速时为1~16，这个1~16 只是bInterval 可以取的值，实际的间隔时间需要计算一下，为2 的（bInterval-1）次方乘以125 微妙，也就是2 的（bInterval-1）次方个微帧。对于等时传输，没有低速，对于全速和高速，这个范围也是为1~16，间隔时间由2 的（bInterval-1）次方算出来，单位为帧或微帧。这样看来，每一帧或微帧里，你最多只能期望有一次等时和中断传输，不能再多了。

### complete

一个指向结束处理函数的指针，**传输成功完成，或者中间发生错误的时候就会调用它**，驱动可以在这里边儿检查urb 的状态，并做一些处理，比如可以释放这个urb，或者重新提交给HCD。就说摄像头吧，你向HCD 提交了个等时的urb 从摄像头那里读取视频数据，传输完成的时候调用了你指定的这个结束处理函数，并在里面取出了urb 里面获得的数据进行解码等处理，然后怎么着？总不会这一个urb 读取的数据就够你向mm 表白了吧，你的爱慕之情可是犹如滔滔江水连绵不绝，所以需要获得更多的数据，那你也总不会再去创建、初始化一个等时的urb 吧，即使再穷极无聊的人也不会那么做，明显刚刚的那个可以继续用的，只要将它再次提交给HCD 就可以了。这个函数指针的定义在include/linux/usb.h

typedef void (\*usb\_complete\_t)(struct urb \*);

# USB内核(USBD)提供的函数

主要有四类功能：

1. 客户端驱动管理，
2. USB设备的配置和管理，
3. 主机控制器的管理，
4. 协议控制命令集和数据传输的管理

## 客户端驱动管理: usb\_register()

### 安装与卸载

**int usb\_register(struct usb\_driver \*new\_driver);**

**void usb\_deregister(struct usb\_driver \*driver);**

USB内核通过一个双向链表usb\_driver\_list来管理所有客户端驱动，具体管理功能为安装和卸载两部分，对应于usb\_register和usb\_deregister，USB内核是动态安装和卸载设备驱动的。

客户端驱动程序应该在初始化函数中调用usb\_register，先检查驱动是否初次安装，根据USBD保存的次版本号数组（目前是16个）中该驱动对应项是否为空，如果不是则返回错误。如果是，将它加入到usb\_driver\_list中，并进行设备接口扫描usb\_scan\_devices，用来探测系统中哪些设备的接口可以被此驱动程序驱动，将会调用驱动提供的probe函数。usb\_register还通过深度优先算法按系统所具有的树型结构搜索系统中所有设备未被驱动的接口，由驱动检验能否驱动。如能，则分配必要的软件资源，配置并让其工作。

当客户端驱动需要从系统中卸掉时，会调用usb\_deregister，将USBD保存的次版本号数组该驱动对应项设为NULL，然后将它从usb\_driver\_list卸掉，然后断开驱动中所有被它驱动的设备接口连接，释放所有资源。usb\_deregister还会通知系统中可用的客户驱动程序，检验这些失去资源的设备接口能否被其他驱动程序驱动，如可用的话，则为其分配资源，让它们正常工作。

### 驱动可以调用的其他接口管理函数:

**void usb\_driver\_claim\_interface(struct usb\_driver \*driver, struct usb\_interface \*iface, void\* priv)**

**int usb\_interface\_claimed(struct usb\_interface \*iface)**

**void usb\_driver\_release\_interface(struct usb\_driver \*driver, struct usb\_interface \*iface)**

### 寻找设备与驱动的匹配：usb\_macth\_id()

**const struct usb\_device\_id \* usb\_match\_id(struct usb\_device \*dev, struct usb\_interface \*interface, const struct usb\_device\_id \*id);**

根据给定接口或设备，从usb\_device\_id数组中查找第一个相符的设备ID。它一般在驱动绑定接口时调用。

## USB设备的配置和管理：usb\_connect()

支持USB设备的热插拔，USBD提供了对设备进行配置和管理，包括 **插入**设备 和 拔下设备。

### 插入设备：

**void usb\_connect(struct usb\_device \*dev);**

**int usb\_new\_device(struct usb\_device \*dev);**

设备插入时，与之相联的集线器首先发现设备的插入信息，通过中断传输将信息传送给集线器的驱动，通过信息分析，确认有新设备插入到总线上，集线器驱动调用usb\_connect和usb\_new\_device来配置设备，并将其与对应的设备驱动建立联系。

usb\_new\_device，设定新设备信息。目前是查找并分配设备地址dev->devnum，真正发USB命令进行配置工作是由usb\_new\_device来完成的

参照协议，完成新设备的配置工作，包括usb\_set\_address来分配地址，usb\_get\_descriptor来获得设备描述符，usb\_get\_configuration来获得设备所有配置描述符，usb\_set\_configuration来激活缺省配置，usbdevfs\_add\_device来加入一个/proc/bus/usb入口，通过usb\_find\_drivers为缺省配置0的每个接口查找相应的驱动程序来进行驱动。

总线上的第一个设备根集线器和主机控制器是一体的，在启动时就认为是插上的，默认地址为0。Linux支持多USB总线，即多个主机控制器和根集线器，它们各自的设备地址是不相关的。

### 拔下设备：

**void usb\_disconnect(struct usb\_device \*\*pdev)**

设备拔下时，与之相联的集线器首先检测到设备的拔下信号，通过中断传输将信息传送给集线器的驱动，集线器的驱动先验证设备是否被拔下，如果是则调用usb\_disconnect进行处理。

断开设备后的处理。找到设备当前活动配置的每个接口的驱动程序，调用它们提供的disconnect接口函数，中断它们与各个接口的数据传输操作，释放它们为每个接口分配的资源。如果此设备是集线器，则递归调用usb\_disconnect来处理它的子设备。释放设备地址，并通过usbdevfs\_remove\_device释放给设备创建的inode（/proc/bus/usb入口），usb\_free\_dev释放USBD给设备分配的资源。

### 设备复位：

**int usb\_reset\_device(struct usb\_device \*dev) hub.c中**

USBD提供了usb\_reset\_device来进行设备的复位操作。它首先复位设备连接的集线器端口，然后与usb\_new\_device函数相似的步骤重新完成对设备的配置操作。调用此函数一定要慎重，如果处理不当，将会影响设备的工作

## 主机控制器的管理: usb\_register\_bus()

### 安装与卸载

**struct usb\_bus \*usb\_alloc\_bus(struct usb\_operations \*op)**

**void usb\_register\_bus(struct usb\_bus \*bus)；**

**void usb\_deregister\_bus (struct usb\_bus \*bus)；**

每个主机控制器拥有一个USB系统，称为一个USB总线。USBD支持多个主机控制器，即多个USB总线。当每增加一个主机控制器时，会给它分配一个usb\_bus结构。USBD动态安装和卸载主机驱动。

主机驱动安装时，它的初始化函数一方面完成主机控制器硬件的配置和初始化工作，另一方面调用usb\_alloc\_bus和usb\_register\_bus来将自己注册到USBD中去，供USB子系统访问。

创建主机控制器对应的总线结构usb\_bus，保存主机控制器给USBD提供的函数接口，并进行初始化。每个主机控制器都为USBD提供了一套函数接口，来进行实际的USB通信操作。

将USB总线结构usb\_bus注册到USBD中，即将其加入到USB内核的总线双向链表中usb\_bus\_list，并创建一个/proc/bus/usb入口

主机驱动卸载时，调用usb\_deregister\_bus和usb\_free\_bus来释放资源

### 其他的主控接口函数：

**设备管理 ：**

**struct usb\_device \*usb\_alloc\_dev(struct usb\_device \*parent, struct usb\_bus \*bus)**

**void usb\_free\_dev(struct usb\_device \*dev)**

**void usb\_inc\_dev\_use(struct usb\_device \*dev)**

**带宽管理**

**int usb\_check\_bandwidth (struct usb\_device \*dev, struct urb \*urb)**

**void usb\_claim\_bandwidth (struct usb\_device \*dev, struct urb \*urb, int bustime, int isoc)**

**void usb\_release\_bandwidth(struct usb\_device \*dev, struct urb \*urb, int isoc)**

## 协议控制命令集和数据传输管理

### 协议控制命令集

USBD为设备的客户端驱动提供了一套控制命令的接口函数，实现对设备的配置，控制和通信。具体可见 P256。这些接口函数都是通过usb\_control\_msg来进行实际发送的，客户端也可以通过调用usb\_control\_msg来完成自己的设备命令。usb\_control\_msg属于同步通信函数，不采用异步回调方式。

#### 标准命令的函数接口(standard): usb\_get\_configuration()…

**CLEAR\_FEATURE**

**int usb\_clear\_halt(struct usb\_device \*dev, int pipe);**

只提供针对停止工作的端点的清除操作，没有提供清除设备的远程唤醒的操作

**GET\_CONFIGURATION**

**int usb\_get\_configuration(struct usb\_device \*dev)**

**GET\_DESCRIPTOR**

**int usb\_get\_descriptor(struct usb\_device \*dev, unsigned char type, unsigned char index, void \*buf, int size)**

**int usb\_get\_string(struct usb\_device \*dev, unsigned short langid, unsigned char index, void \*buf, int size)**

**int usb\_get\_device\_descriptor(struct usb\_device \*dev)**

**int \_\_usb\_get\_extra\_descriptor(char \*buffer, unsigned size, unsigned char type, void \*\*ptr){**

**GET\_INTERFACE**

无

**GET\_STATUS**

**int usb\_get\_status(struct usb\_device \*dev, int type, int target, void \*data)**

**SET\_ADDRESS**

**int usb\_set\_address(struct usb\_device \*dev)**

**SET\_CONFIGURATION**

**int usb\_set\_configuration(struct usb\_device \*dev, int configuration)**

**SET\_DESCRIPTOR**

无，因为一般设备不支持该命令，不允许增加新描述符

**SET\_FEATURE**

无

**SET\_INTERFACE**

**int usb\_set\_interface(struct usb\_device \*dev, int interface, int alternate)**

**SYNCH\_FRAME**

无

#### 类命令的函数接口(class)

**GET\_DESCRIPTOR (class)**

**int usb\_get\_class\_descriptor(struct usb\_device \*dev, int ifnum, unsigned char type, unsigned char id, void \*buf, int size)**

**GET\_PROTOCOL**

**int usb\_get\_protocol(struct usb\_device \*dev, int ifnum)**

**SET\_PROTOCOL**

**int usb\_set\_protocol(struct usb\_device \*dev, int ifnum, int protocol)**

**GET\_REPORT**

**int usb\_get\_report(struct usb\_device \*dev, int ifnum, unsigned char type, unsigned char id, void \*buf, int size)**

**SET\_REPORT**

**int usb\_set\_report(struct usb\_device \*dev, int ifnum, unsigned char type, unsigned char id, void \*buf, int size)**

**SET\_IDLE**

**int usb\_set\_idle(struct usb\_device \*dev, int ifnum, int duration, int report\_id)**

### 数据传输的管理

数据传输都是使用USB内核提供的URB(USB Request Block)。具体可参见上面的数据结构。有些变量是针对特定传输类型的。

批量传输：无

控制传输：setup\_packet

中断传输：start\_frame, interval

实时传输：start\_frame, number\_packets, error\_count, timeout, iso\_frame\_desc

#### URB的接口函数：usb\_alloc\_urb()

**urb\_t \*usb\_alloc\_urb(int iso\_packets)**

用于给客户驱动分配URB，iso\_packets为该URB中需要传输的实时数据包的个数，其他传输为0

**void usb\_free\_urb(urb\_t\* urb)**

与usb\_alloc\_urb对应，释放分配的URB

**int usb\_submit\_urb(urb\_t \*urb)**

将一个或多个urb异步发送给USB内核处理。其中urb可以是一个，也可以是多个，并且可以对应于不同的端点

**int usb\_unlink\_urb(urb\_t \*urb)**

表示在urb传输处理完成之前，取消对它们的数据处理。一般是设备在工作过程中被拔下，或软件主动取消对某个数据传输的处理，或URB传输超时时调用

当主机控制器驱动将URB中的数据传输**处理完成**后，将会调用urb-> **complete**回调函数来通知客户驱动。当URB处理**超时**，客户端驱动会调用**usb\_unlink\_urb**来通知HCD**取消**对此URB数据的传输

#### 初始化URB：usb\_fill\_bulk\_urb()

初始化具体的urb包

|  |
| --- |
| static inline void usb\_fill\_bulk\_urb(  struct urb \*urb, //事务传输中的urb  struct usb\_device \*dev, //事务传输的目的设备  unsigned int pipe, // USB主机与USB设备之间数据传输的通道  void \*transfer\_buffer, //发送数据所申请的内存缓冲区首地址  int buffer\_length, //发送数据缓冲区的长度  usb\_complete\_t complete\_fn, //调用完成函数  void \*context) //complete函数的上下文    static inline void usb\_fill\_control\_urb(struct urb \*urb,  struct usb\_device \*dev,  unsigned int pipe,  unsigned char \*setup\_packet, //即将被发送的设备数据包  void \*transfer\_buffer,  int buffer\_length,  usb\_complete\_t complete\_fn,  void \*context)  static inline 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\_fn,  void \*context,  int interval) //中断传输中两个URB调度的时间间隔 |

不同的传输模式下，驱动为之申请不同的URB。其中，Linux内核只支持同步传输外的三种传输事件，ISO事务需要手工进行初始化工作。控制传输事务、批量传输事务、中断传输事务API如上所示。

#### 提交URB：usb\_submit\_urb()

URB初始化完成之后，USBD开始通过usb\_start\_wait\_urb()提交urb请求(它调用usb\_submit\_urb来真正的发送URB请求)，添加completition函数。

接下来，从message.c传到主机控制器(hcd.c)，开始真正的usb\_hcd\_submit\_urb()。此时，根据是否为根集线器，进入不同的工作队列。

usb\_start\_wait\_urb->

usb\_submit\_urb->

usb\_hcd\_submit\_urb

**root\_hub传输**

若为root hub，将调用rh\_urb\_enqueue()，共有两种传输事务(控制传输和中断传输)

|  |
| --- |
| static int rh\_urb\_enqueue (struct usb\_hcd \*hcd, struct urb \*urb)  {  if (usb\_endpoint\_xfer\_int(&urb->ep->desc)) // 中断传输  return rh\_queue\_status (hcd, urb);  if (usb\_endpoint\_xfer\_control(&urb->ep->desc)) // 控制传输  return rh\_call\_control (hcd, urb);  return -EINVAL;  } |

**非root\_hub传输**

对于非常root\_hub传输，它调用:

|  |
| --- |
| status = hcd->driver->urb\_enqueue(hcd, urb, mem\_flags); |

**批量传输**

root\_hub本身没有批量传输流程，按照控制传输流程，控制传输最终要通过switch语句跳转到Bulk-Only传输流程中。

#### 控制传输接口函数：usb\_control\_msg()

**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)**

构建一控制传输的URB，发送并等待完成或超时。该函数不能用在中断处理函数中，包括后半部分处理函数。如果需要发送异步信息或在中断处理函数中发送信息，则使用usb\_submit\_urb。

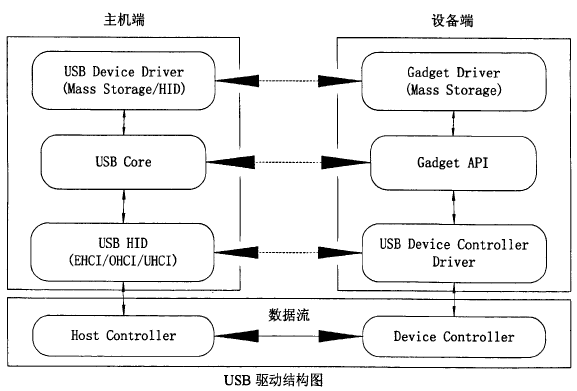
#### 批量传输接口函数：usb\_bulk\_msg()

**int usb\_bulk\_msg(struct usb\_device \*usb\_dev, unsigned int pipe, void \*data, int len, int \*actual\_length, int timeout)**

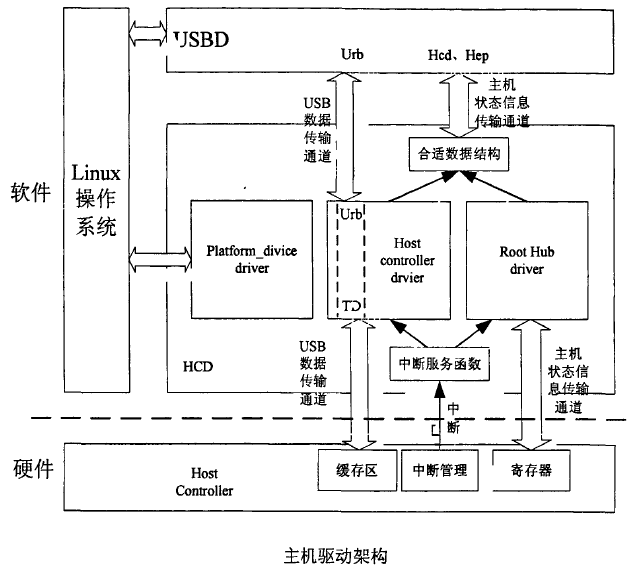
构建一批量传输的URB，发送并等待完成或超时。同usb\_control\_msg类似，该函数不能用在中断处理函数中，包括后半部分处理函数。如果需要发送异步信息或在中断处理函数中发送信息，则使用usb\_submit\_urb。

# USB驱动框架

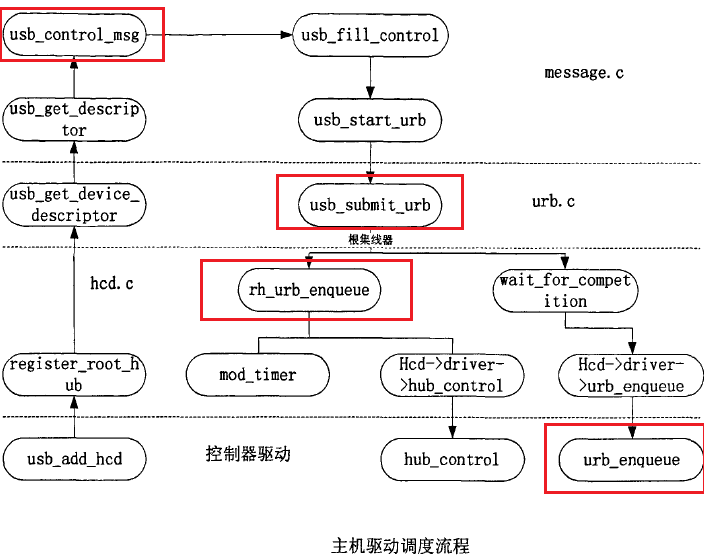
USB驱动架构如下图所示：



## 1 USB主机端驱动



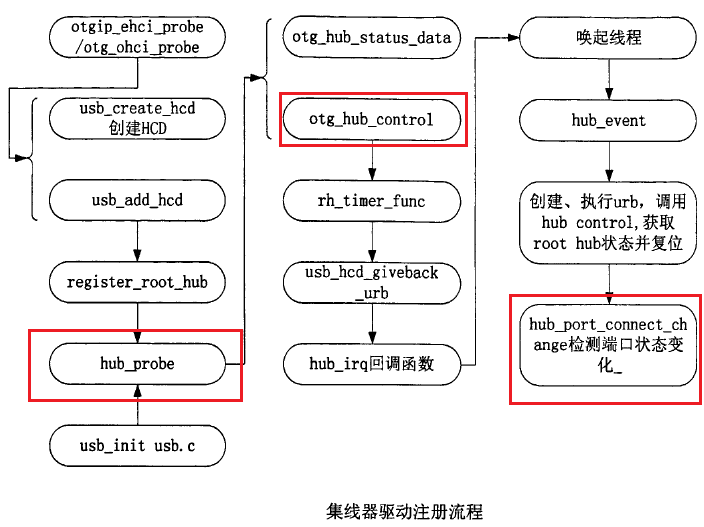
USB核心(USBD)是整个USB驱动的核心部分，从上图可知，一方面USBD对接收到USB主机控制器的数据进行处理，并传递给上层的设备端驱动软件；同时也接收来自上层的非USB格式数据流，进行相应的数据处理后传递给USB主机控制器驱动。



USB数据传输都以**URB(USB Request Block)请求、URB生成、URB递交、URB释放**为主线。从上图可知，当加载控制器驱动之后，注册根据集线器，hub和hcd驱动成为一个整体。接着，主机通过控制传输获取设备的控制描述符等信息，接着详述整个控制传输的流程。usb\_submit\_urb依据是否连接到根集线器来决定调用urb\_enqueue或rh\_urb\_enqueue函数。

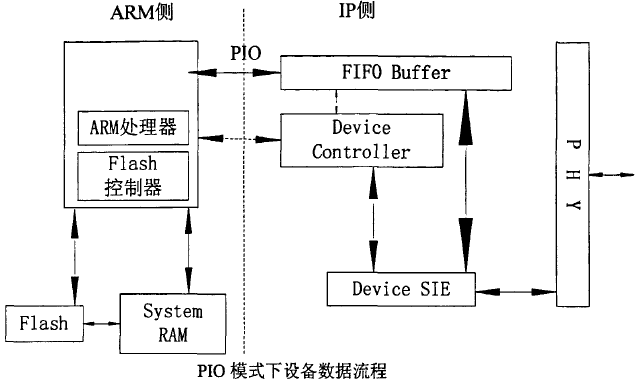
USB从设备通过集线器或根集线器连接到USB主机上。比如：主机通过根集线器与外界进行数据交互，**根集线器通过探测数据线状态的变化来通知USB主机是否有USB外围设备接入**。

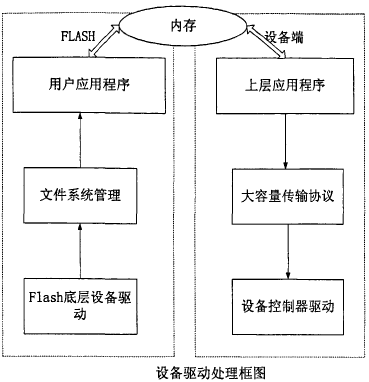
在主机端控制器驱动加载的过程中，注册了根集线器，然后匹配了相应的hub驱动程序，同时完成了对Hub的**轮询函数和状态处理函数**的设置。这样，**一旦hub集线器的状态发生变化，就会产生相应的中断，主机端控制器就会执行相应的中断处理函数**，下图为hub驱动程序的流程图。



USB Core中的usb\_init()函数中完成了对hub线程(khubd，在**usb\_hub\_init**函数中真正地创建)的创建，然后完成相应设备的探测。主机端控制器驱动进行探测时，将hub驱动和主机端控制器驱动结合在一起，相互之间完成调用。 相对于大容量存储设备与主机之间通过控制/批量传输，集线器与主机之间通过中断/控制方式完成数据交互。

## 2 USB设备端驱动





从上图可知，设备端驱动包含两部分：

1) 底层设备控制器驱动

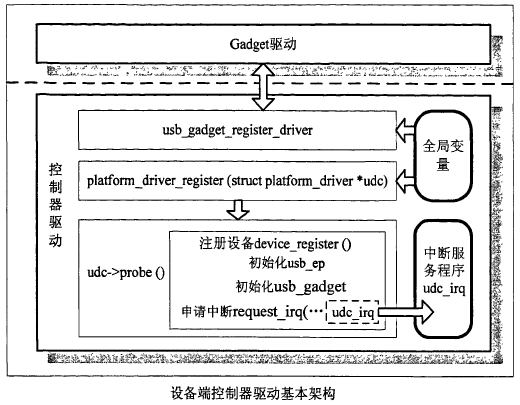
2) 上层大容量存储类驱动

### 2.1 设备控制器驱动

USB设备控制器驱动主要实现Gadget API定义的函数和中断服务函数，可按功能划分为：API函数实现模块和中断处理模块。

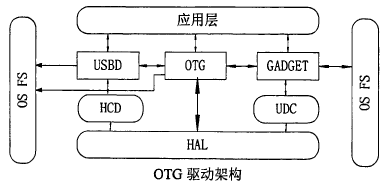
API函数主要实现Gadget API定义的函数功能，如结构体usb\_ep\_ops和usb\_gadget\_ops中的函数、usb\_gadget\_register\_driver函数。这些函数是供Gadget Driver调用。

中断处理模块主要处理设备控制器产生的各种中断，包括端点中断、复位、挂起等中断。



上图为设备端控制器基本架构，主要完成了Gadget驱动和控制器驱动绑定、usb\_gadget\_register\_driver注册。

## 3 OTG驱动



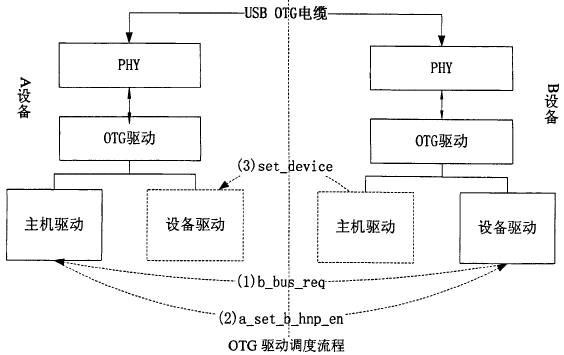
OS\_FS: 文件系统

USBD: USB核心

HCD: 主机控制器驱动

UDC: 设备端控制器驱动

OTG设备支持HNP和SRP协议。OTG设备通过USB OTG电缆连接到一起，其中接Mini-A接口的设备为A设备，默认为主机端，Mini-B接口的设备默认为B设备。当A、B设备完成数据交互之后，A、B设备之间的USB OTG电缆进入挂起状态，如下图所示：



当B设备写入b\_bus\_req，向A设备发起HNP请求。待A设备响应之后，A设备发送a\_set\_b\_hnp\_en，B设备响应之后即进入主机状态，同时发送请求使用A设备set\_device，这样A、B设备完成主从交换。

# USB传输流程

## 1 USB初始化过程

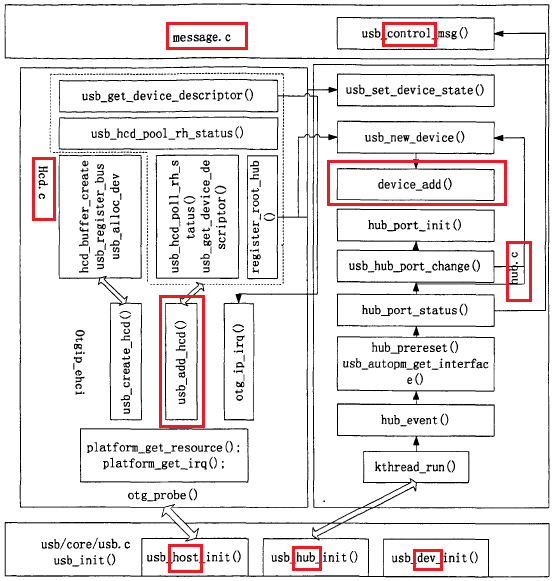
USB驱动作为一个系统，集成了众多的驱动模块，注册过程非常复杂。从USB系统的角度来说，USB主机驱动主要包含：

**1) USB核驱动**

**2) 主机控制器驱动**

**3) 集线器驱动**

驱动的加载执行流程如下图所示：



USB初始化过程

### 1.1 USB Core的初始化：usb\_init()

USB驱动从USB子系统的初始化开始，USB子系统的初始化在文件driver/usb/core/usb.c

|  |
| --- |
| subsys\_initcall(usb\_init);  module\_exit(usb\_exit); |

subsys\_initcall()是一个宏，可以理解为module\_init()。由于此部分代码非常重要，开发者把它看作一个子系统，而不仅仅是一个模块。USB Core这个模块代表的不是某一个设备，而是所有USB设备赖以生存的模块。在Linux中，像这样一个类别的设备驱动被归结为一个子系统。subsys\_initcall(usb\_init)告诉我们，usb\_init才是真正的初始化函数，而usb\_exit将是整个USB子系统结束时的清理函数。

### 1.2 主机控制器的初始化及驱动执行(以EHCI为例)：usb\_add\_hcd()

|  |
| --- |
| module\_init(otg\_init); 模块注册  static init \_\_init otg\_init(void);  platform\_driver\_register(); 平台注册  static int \_\_init otg\_probe(struct platform\_device \*pdev); 探测处理函数  reg = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0); 获取寄存器信息  data = platform\_get\_resource(pdev,IORESOURCE\_MEM, 1); 获取内存信息  irq = platform\_get\_irq(pdev,0); 获取中断号  usb\_create\_hcd(&otg\_hc\_driver, &pdev->dev, pdev->dev.bus\_id); |

分配和初始化HCD结构体。对设备数据空间进行分配，初始化计数器、总线、定时器、hcd结构体各成员值。

|  |
| --- |
| ret = usb\_add\_hcd(hcd,irq,SA\_INTERRUPT); |

完成HCD结构体的初始化和注册。申请buffer，注册总线、分配设备端内存空间，向中断向量表中申请中断，注册根集线器，对根集线器状态进行轮询。

### 1.3 注册集线器：register\_root\_hub()

|  |
| --- |
| register\_root\_hub(hcd); |

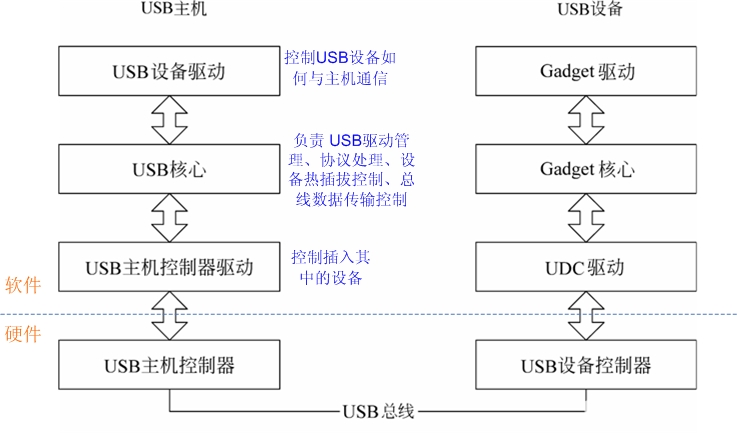
在USB系统驱动加载的过程中，创建了集线器的线程(khubd)，并且一直查询相应的线程事务。HCD驱动中，将集线器作为一个设备添加到主机控制器驱动中，然后进行集线器端口的初始化。在USB主机看来，根集线器本身也是USB主机的设备。USB主机驱动加载完成之后，即开始注册根集线器，并且作为一个设备加载到主机驱动之中。

USB主机和USB设备之间进行数据交互，USB设备本身并没有总线控制权，U盘被动地接收USB主机发送过来的信息并做出响应。USB主机控制器与根集线器构成了主机系统，然后外接其它的USB设备。

为了更好地探测到根集线器的状态变化，USB主机控制器驱动增加了状态轮询函数，以一定的时间间隔轮询根集线器状态是否发生变化。一旦根集线器状态发生变化，主机控制器就会产生相应的响应。

USB主机和USB设备之间的数据传输以URB(USB Request Block)的形式进行。

## 2 USB驱动工作过程



在Linux驱动中，USB驱动处于最底层的是USB主机控制器硬件，在其之上运行的是USB主机控制器驱动，主机控制器之上为USB核心层，再上层为USB设备驱动层。

在主机侧的层次结构中，要实现的USB驱动包括两类：USB主机控制器驱动和USB设备驱动，前者控制插入其中的USB设备，后者控制USB设备如何与主机通信。Linux内核USB核心负责USB驱动管理和协议处理的主要工作。主机控制器驱动和设备驱动之间的USB核心非常重要，其功能包括：通过定义一些数据结构、宏和功能函数，向上为设备驱动提供编程接口，向下为USB主机控制器驱动提供编程接口；通过全局变量维护整个系统的USB设备信息；完成设备热插拔控制、总线数据传输控制等。

在USB设备侧驱动程序分为3个层次：UDC驱动程序、Gadget API和Gadget驱动程序。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，向上层提供与硬件相关操作的回调函数。当前Gadget API是UDC驱动程序回调函数的简单包装。Gadget驱动程序具体控制USB设备功能的实现，使设备表现出“网络连接”、“打印机”或“USB Mass Storage”等特性，它使用Gadget API控制UDC实现上述功能。Gadget API把下层的UDC驱动程序和上层的Gadget驱动程序隔离开，使得在Linux系统中编写USB设备侧驱动程序时能够把功能的实现和底层通信分离。

USB通信最基本的形式是通过端点（USB端点分中断（Interrupt)、批量(Bulk)、等时(ISO)、控制(Control)四种，每种用途不同），**USB端点只能往一个方向传送数据**，从主机到设备或者从设备到主机，**端点可以看作是单向的管道**（pipe）。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。USB核心使用一个列表（是一个包含制造商ID和设备号ID的一个结构体）来判断对于一个设备该使用哪一个驱动程序，热插拨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执行哪一个驱动程序的Probe。

# LINUX/Android USB调试

Ubuntu下的准备工作：

（1）如何查看内核中的printk输出： dmesg， dmesg |head， dmesg |tail

（2）的USB抓包工具：usb\_mon

Ubuntu下已经内置了USB抓包工具---USB\_MON，make menuconfig 时候会让我们选择。

使用方法如下：

# mount -t debugfs none /sys/kernel/debug

# modprobe usbmon

查看当前的USB device：

# cat /sys/kernel/debug/usb/devices

下面的输出：参见Documentation/usb/usbmon.txt

T: Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 12 Spd=480 MxCh= 0

D: Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1

P: Vendor=601a ProdID=4740 Rev= 1.00

S: Manufacturer=Ingenic

S: Product=JZ4740 USB Boot Device

C:\* #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr= 2mA

I:\* If#= 0 Alt= 0 #EPs= 2 Cls=ff(vend.) Sub=00 Prot=50 Driver=skeleton

E: Ad=01(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms

E: Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms

监听 全部 Bus 上的全部USB数据包：

# cat /sys/kernel/debug/usb/usbmon/0u

监听 Bus01上的全部USB数据包：

# cat /sys/kernel/debug/usb/usbmon/1u

一旦有数据包，就会打印出来：参见Documentation/usb/usbmon.txt

f07dbb80 3233986312 S Co:1:012:0 s 40 01 80c0 0000 0000 0

f07dbb80 3234327703 C Co:1:012:0 0 0

f07dbb80 3234327880 S Co:1:012:0 s 40 02 0000 0200 0000 0

f07dbb80 3234329561 C Co:1:012:0 0 0

f07dbb80 3234329625 S Bi:1:012:1 -115 32 <

f07dbb80 3234331550 C Bi:1:012:1 -75 32 = 00000000 00000000 。。。

f07dbb80(URB标志) 3233986312(TimeStamp: 毫秒) S(Event Type) Co:1:012:0(URBType : BusNo : DeviceAddr : EndPointNum) s(URB Status) 40 01 80c0 0000 0000 0

URB tag ：标志

TimeStamp：毫秒数

Event Type： S - submission,

C - callback,

E - submission error.

URBType：BusNo：DeviceAddr：EndPointNum

Ci Co Control input and output

Zi Zo Isochronous input and output

Ii Io Interrupt input and output

Bi Bo Bulk input and output

URB Status： 0为没有错误。-XXX是错误码。

# end