

## 1. 概述

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。



## 2. 硬件

### 2.1 USB 原理图

这是战舰开发板中所使用的 USB 原理图,可以看到 D+上拉 1.5kΩ 电阻,是 full-speed 设备.

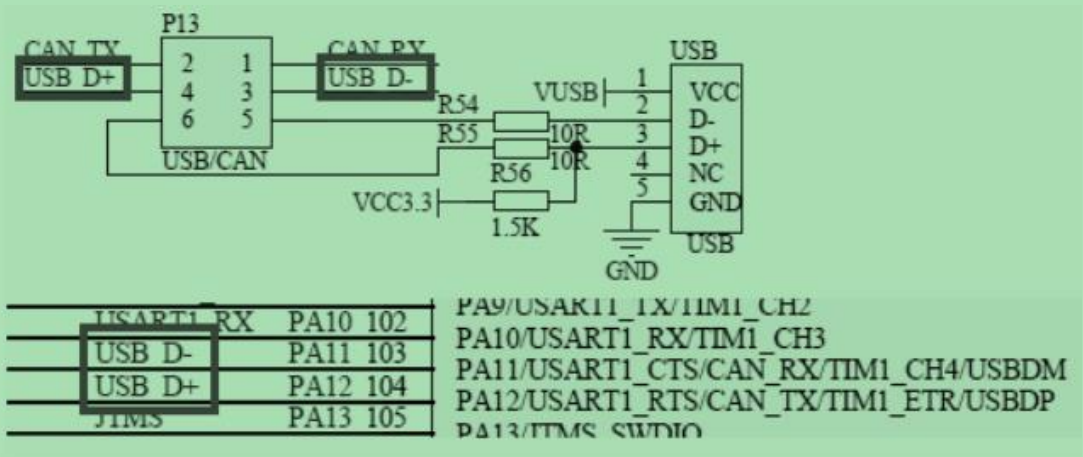


图 2-1. USB 设备原理图

## 3. 基本概念

### 3.1 USB 总线拓扑

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)：

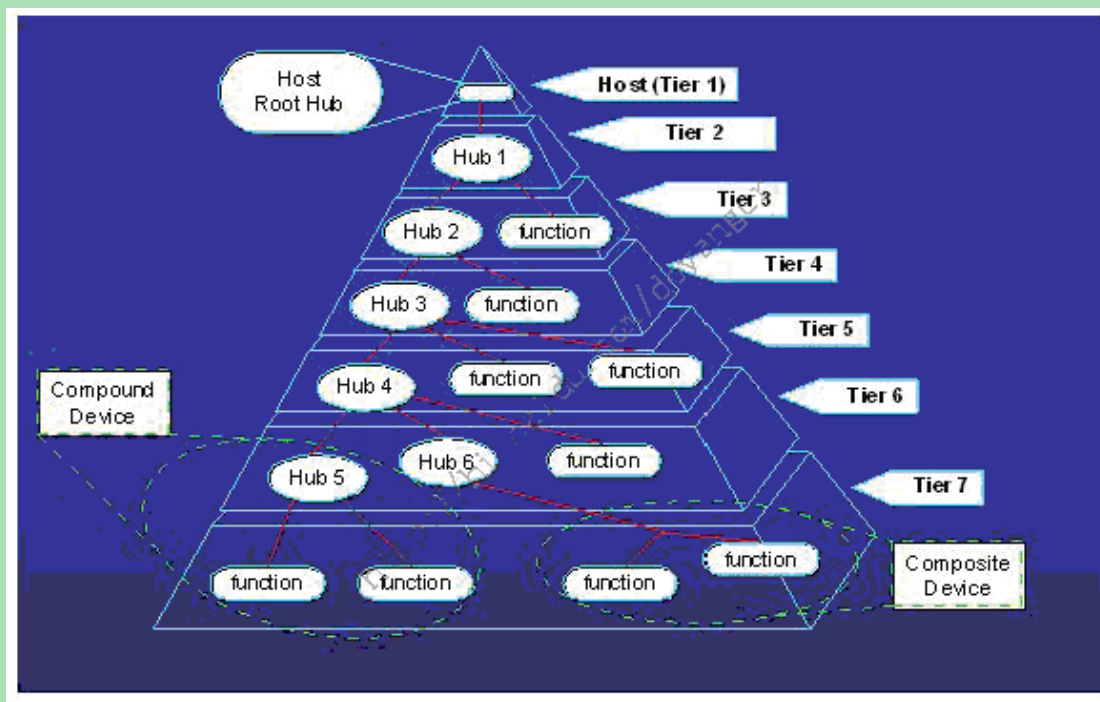


图 3-1. USB 总线拓扑结构

### 3.2 设备状态

USB 连接到主机后配置过程中的设备状态

Visible Device States

Attached	Powered	Default	Address	Configured	Suspended	State
No	--	--	--	--	--	Device is not attached to the USB. Other attributes are not significant.
Yes	No	--	--	--	--	Device is attached to the USB, but is not powered. Other attributes are not significant.
Yes	Yes	No	--	--	--	Device is attached to the USB and powered, but has not been reset.
Yes	Yes	Yes	No	--	--	Device is attached to the USB and powered and has been reset, but has not been assigned a unique address. Device responds at the default address.
Yes	Yes	Yes	Yes	No	--	Device is attached to the USB, powered, has been reset, and a unique device address has been assigned. Device is not configured.
Yes	Yes	Yes	Yes	Yes	No	Device is attached to the USB, powered, has been reset, has a unique address, is configured, and is not suspended. The host may now use the function provided by the device.
Yes	Yes	--	--	--	Yes	Device is, at minimum, attached to the USB and is powered and has not seen bus activity for 3 ms. It may also have a unique address and be configured for use. However, because the device is suspended, the host may not use the device's function.

表 3-1. USB 连接过程中的设备状态

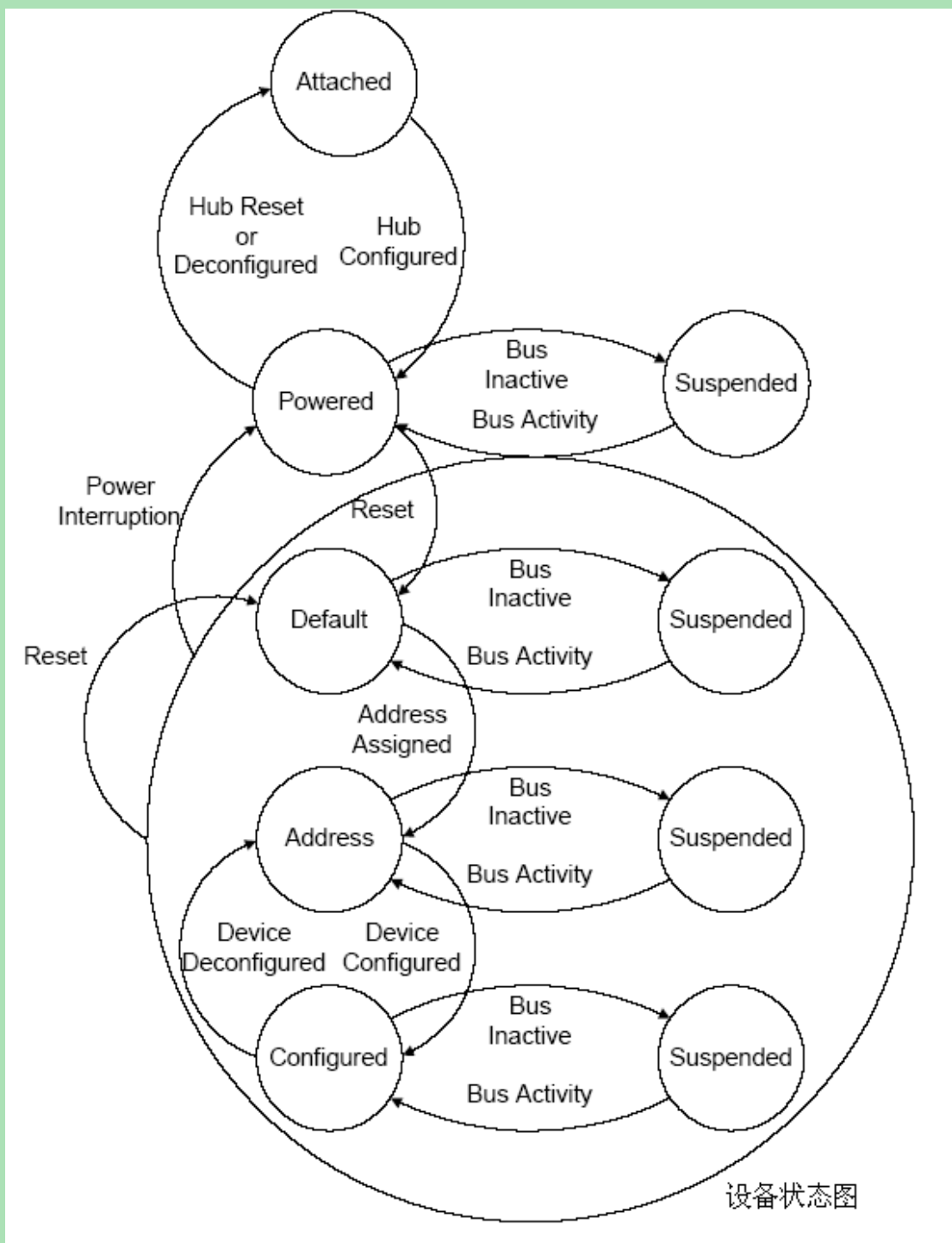


图 3-2. 设备状态图

### 3.3 设备状态详解

#### 1) 连接 (Attached)

设备可以连接到 USB 或者从 USB 上拔出. USB 设备从总线上拔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(data toggle)的结点(end point)的(Data 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。

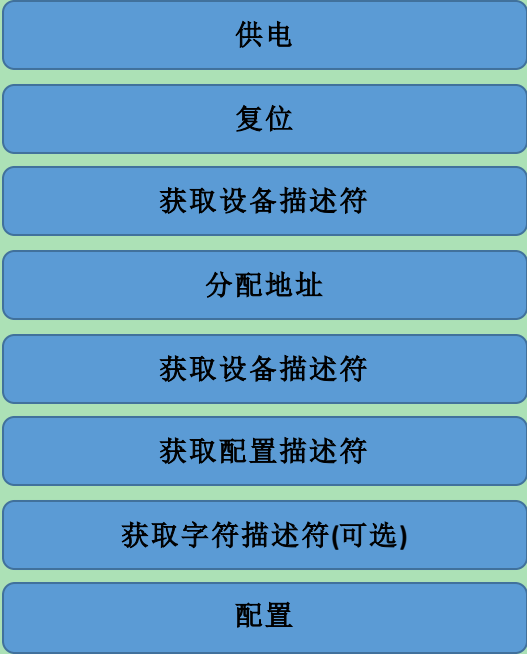
## 3.4 枚举过程

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)）。

枚举步骤简单表示如下



下面详细介绍设备枚举过程.

3. 4. 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。

3. 4. 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 R<sub>pd</sub>，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 R<sub>pu</sub>。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。如下图。

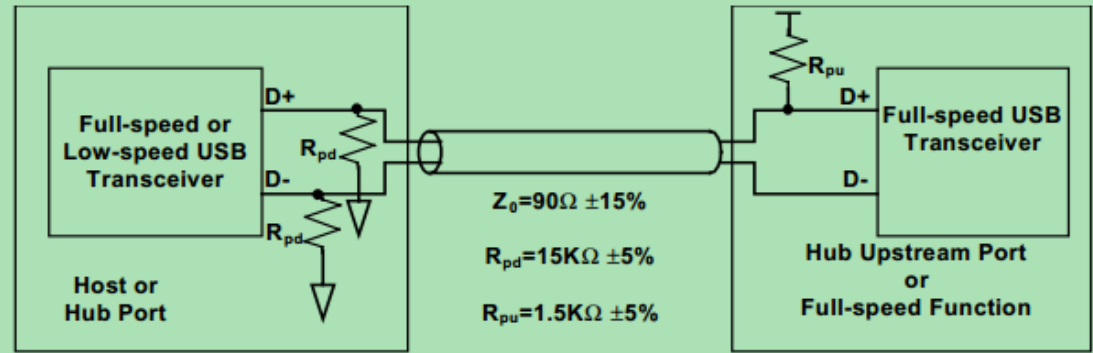


图 3-1. USB 全速设备电阻器连接

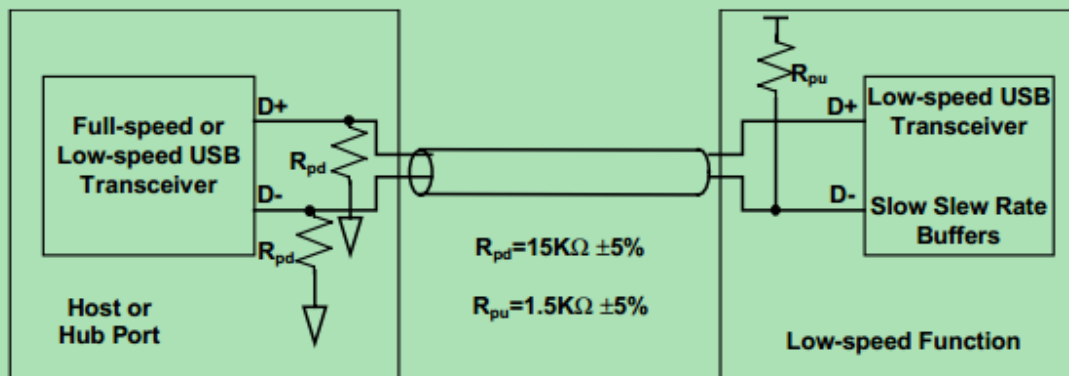


图 3-2. USB 低速设备电阻器连接

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。

USB2.0 连接线定义如下所示：

Contact Number	Signal Name	Typical WiringAssignment
1	VBUS	红
2	D-	白
3	D+	绿
4	GND	黑
Shell	Shield	Drain Wire

表 3-2. USB2.0 连接线定义

### 3.4.3 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）。

### 3.4.4 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。

### 3.4.5 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。

### 3.4.6 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 hub 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。

### 3.4.7 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。)

### 3.4.8 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。

### 3.4.9 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。

### 3.4.10 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。使用 USBTRACE (使用方法见附录 A) 捕获到的 Get\_Descriptor 请求 (Device type) 和设备描述符如下：

Get_Descriptor 请求	
TransferBuffer	0xFFFFFE00035916DB0
TransferBufferMDL	0xFFFFFE0003664EDF0
UrbLink	0x0
SetupPacket	0x80 0x6 0x0 0x2 0x0 0x0 0x22 0x0
RequestType	0x80 (Direction: Device-to-host, Type: Standard, Recipient: Device)
Request	0x6 (GET_DESCRIPTOR)
Value	0x200 (USB_CONFIGURATION_DESCRIPTOR_TYPE)
Index	0x0

Get_Descriptor 请求	
Length	0x22

表 3-3. 捕获到的 GET\_DESCRIPTOR 请求

Device Descriptor	
bLength	0x12
bcdUSB	0x0200 (USB 2.0)
bDeviceClass	0x0
bDeviceSubClass	0x0
bDeviceProtocol	0x0
bMaxPacketSize0	0x40
idVendor	0x483 (STMicroelectronics )
idProduct	0x5710 (Joystick in FS Mode )
bcdDevice	0x200
iManufacturer	0x1
iProduct	0x2
iSerialNumber	0x3
bNumConfigurations	0x1

表 3-4. 捕获到的设备描述符

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，端点描述符等等。

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。

### 3.4.11 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：

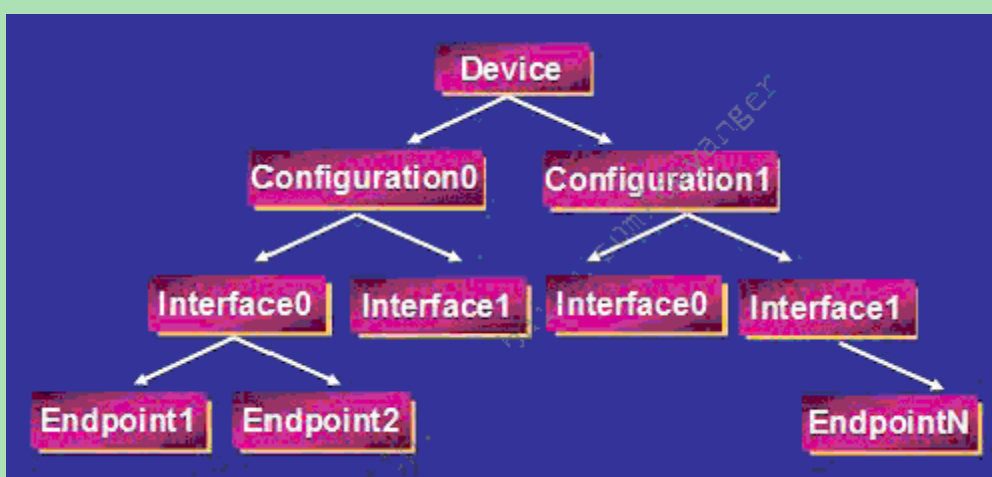


图 3-3. 设备-配置-接口-端点关系

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）。

#### 3. 4. 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。

## 4. 代码分析

前边我们一起详细了解了 USB 设备枚举过程, 现在就可以参考这抓包结果一步步分析代码的执行过程, 然后在关键处在辅以截图(附录 A 介绍了 USBTRACE 的使用方法).

### 4.1 USB 配置

先来看 main 函数中的一处与 USB 直接相关的代码, 初始化 USB 设置之前需要先重启 STM32 USB 模块, 通过下面代码实现:

```
usb_port_set(0);
delay_ms(300);
usb_port_set(1);
```

模块的重启通过配置 USB 控制寄存器(USB\_CNTR)的 PDWN 位(位 1)实现, 该位清除表示模块加电, 置位表示模块断电.

接下来就是 USB 配置与初始化的主要函数

```
USB_Interrupts_Config();
Set_USBClock();
USB_Init();
```

函数 USB\_Interrupts\_Config 用于设置 USB 唤醒中断和低优先级数据处理中断, 函数 Set\_USBClock 用于使能和配置 USB 时钟, 然后调用 USB\_Init 完成 USB 初始化.

### 4.2 USB 初始化

来看 usb\_init.c USB\_Init 函数:

```
void USB_Init(void)
{
    pInformation = &Device_Info;
    pInformation->ControlState = IN_DATA;
    pProperty = &Device_Property;
    pUser_Standard_Requests = &User_Standard_Requests;
    /* Initialize devices one by one */
    pProperty->Init();
    //Joystick_init();
}
```

pInformation 指向一个 DEVICE\_INFO 结构体的实例, 用于存储控制传输过程中接收到的 SETUP 事务的命令数据. pInformation->ControlState 为联合体 CONTROL\_STATE(定义于 usb\_core.c)的某一个值, 表示控制传输过程中设备可能的状态. 最后一行代码实际调用了 usb\_prop.c 中的 Joystick\_init 初始化函数.

### 4.3 真正的初始化函数

如上节所示, 真正的初始化函数是 Joystick\_init, 代码:

```
void Joystick_init(void)
{
    /* Update the serial number string descriptor with the data from the unique ID*/
    Get_SerialNum();

    pInformation->Current_Configuration = WAIT_SETUP;
```

```

    /* Connect the device */
    PowerOn();
    /* USB interrupts initialization */
    _SetISTR(0); /* clear pending interrupts (清除中断标志)*/
    wInterrupt_Mask = IMR_MSK; //这组寄存器用于定义 USB 模块的工作模式，中断的处理，
    设备的地址和读取当前帧的编号
    _SetCNTR(wInterrupt_Mask); /* set interrupts mask (设置相应的控制寄存器)*/

    bDeviceState = UNCONNECTED;
}

```

函数 `Get_SerialNum(hw_config.c)` 用于获取 `serialNumberStringDescriptor` (`StringDescriptor` 的一种) 并将其存入对应字符数组. 然后控制端点进入 `WAIT_SETUP` 状态. 函数 `PowerOn(usb_pwr.c)` 强制复位 USB 模块后使能了复位 (`RESETM`), 挂起 (`SUSPM`), 唤醒 (`WKUPM`) 中断. 使能了复位中断以后, 将进入到 USB 的复位中断里面去 (详见 4.3.1). 然后将 `IMR_MSK` 写入控制寄存器, 打开所有 USB 中断 (详见 4.3.2). 在 D+ 被接通上拉以后, 设备就能被主机检测到.

#### 4.3.1 复位中断的处理

复位中断由函数 `USB_Istr(usb_istr.c)` 处理, 下面截取了函数中关于复位中断处理部分:

```

void USB_Istr (void)
{
    wIstr = _GetISTR ();

    #if (IMR_MSK & ISTR_RESET)          // USB 复位请求中断
        if (wIstr & ISTR_RESET & wInterrupt_Mask)
        {
            _SetISTR ((u16)CLR_RESET); // 清除复位中断标志
            Device_Property.Reset ();   // 进入到复位中断( Joystick_Reset )
            #ifdef RESET_CALLBACK
                RESET_Callback ();
            #endif
        }
    #endif
    .....
}

```

不难发现 `USB_Istr` 起到派遣函数的作用, 复位中断的实际处理交由 `Device_Property.Reset` 所指函数 `Joystick_Reset(usb_prop.c)`.

函数 `Joystick_Reset` 源码, 用于初始化端点:

```

void Joystick_Reset(void)
{
    /* Set Joystick_DEVICE as not configured */
    pInformation->Current_Configuration = 0;
    pInformation->Current_Interface = 0; /*the default Interface*/
}

```

```

/* Current Feature initialization */
pInformation->Current_Feature = Joystick_ConfigDescriptor[7]; //供电模式选择

SetBTABLE(BTABLE_ADDRESS); //分组缓冲区描述表地址设置

/* Initialize Endpoint 0 */
SetEPTType(ENDP0, EP_CONTROL); //初始化为控制端点类型
SetEPTxStatus(ENDP0, EP_TX_STALL); //端点以 STALL 分组响应所有的发送请求
//也就是端点状态设置成发送无效, 也就是主机的 IN 令牌包来的时候, 回送一个
STALL
SetEPRxAddr(ENDP0, ENDP0_RXADDR); //设置端点 0 描述符的接受地址
SetEPTxAddr(ENDP0, ENDP0_TXADDR); //设置端点 0 描述符的发送地址
Clear_Status_Out(ENDP0); //仅用于控制端点 如果 STATUS_OUT 位被清除, OUT 分组可以包含任意长度的数据
SetEPRxCount(ENDP0, Device_Property.MaxPacketSize); //设置端点 0 的接受字节寄存器的最大值是 64
SetEPRxValid(ENDP0); //设置接受端点有效

/* Initialize Endpoint 1 */
SetEPTType(ENDP1, EP_INTERRUPT); //初始化为中断端点类型
SetEPTxAddr(ENDP1, ENDP1_TXADDR); //设置发送数据的地址
SetEPTxCount(ENDP1, 4); //设置发送的长度
SetEPRxStatus(ENDP1, EP_RX_DIS); //设置接受端点关闭
SetEPTxStatus(ENDP1, EP_TX_NAK); //设置发送端点端点非应答

bDeviceState = ATTACHED; //当前状态连接

/* Set this device to response on default address */
SetDeviceAddress(0); //设置设备用缺省地址相应
}

```

#### 4.3.2 初始化控制 USB 寄存器

函数 Joystick\_init 中初始化 USB 控制寄存器主要工作就是初始化中断设置. 宏 IMR\_MSK 是多个中断掩码集合, 它使能了下列中断:

宏	对应中断解释
CNTR_CTRM	正确传输(CTR)中断使能
CNTR_WKUPM	唤醒中断使能
CNTR_SUSPM	挂起(SUSP)中断使能
CNTR_ERRM	出错中断使能
CNTR_SOFM	帧首中断使能
CNTR_ESOFM	期望帧首中断使能
CNTR_RESETM	设置此位将向 PC 主机发送唤醒请求。根据 USB 协议, 如果此位在 1ms 到 15ms 内保持有效, 主机将对 USB 模块实行唤醒操作



表 4-1. 宏 IMR\_MSK 对应使能中断

4.4 获取设备描述符

主机进入控制传输的第一阶段: 建立事务, 发 SETUP 令牌包, 发请求数据包, 设备发 ACK 包.



主机发出对地址 0、端点 0 发出 SETUP 令牌包, 首先端点 0 寄存器的第 11 位 SETUP 位置位, 表明收到了 SETUP 令牌包.

由于此时端点 0 数据接收有效. 所以接下来主机的请求数据包被 SIE 保存到端点 0 描述附表的 RxADDR 里面. 收到的字节数保存到 RxCount 里面.

端点 0 寄存器的 CTR\_RX 被置位为 1, ISTR 的 CTR 置位为 1, DIR=1, EP\_ID=0, 表示端点 0 接收到主机来的请求数据. 此时设备已经 ACK 主机, 将触发正确传输完成中断. 下面就进入中断看一看.

CTR\_LP(usb\_int.c)低优先级中断处理函数在控制传输, 中断传输, 大容量传输下使用 (在单缓冲模式下使用). 当一次正确的 OUT、SETUP、IN 数据传输完成后, 硬件会自动设置端点寄存器 STAT\_RX 位为 NAK 状态, 使应用程序有足够的时间处理完当前传输的数据后, 响应下一个数据分组.

函数 CTR\_LP 负责 SETUP 处理代码段:

```
void CTR_LP(void)
```

```

{
    .....
    _ClearEP_CTR_RX(ENDP0); /* SETUP bit kept frozen while CTR_RX = 1 */
    Setup0_Process(); //处理 SETUP 事件(程序会进入到这个函数里面)
    /* before terminate set Tx & Rx status (恢复端点 0 发送接收状态) */
    _SetEPRxStatus(ENDP0, SaveRState);
    _SetEPTxStatus(ENDP0, SaveTState);
    return;
    .....
}

```

函数 Setup0\_Process(usb\_core.c) 中的到主机发送的设备请求命令, 然后调用相应函数.

```

u8 Setup0_Process(void)
{
    ..... //Get the device request data

    pInformation->ControlState = SETTING_UP;
    if (pInformation->USBwLength == 0)
    {
        /* Setup with no data stage */
        NoData_Setup0();
    } else
    {
        /* Setup with data stage */
        Data_Setup0(); //由于是有数据的传输, 所有要进入到这个函数
    }
    return Post0_Process();
}

```

函数 Data\_Setup0(usb\_core.c) 的定义描述 Proceed the processing of setup request with data stage.

```

void Data_Setup0(void)
{
    .....
}

```

#### 4.5 返回设备描述符数据

获取描述符的控制传输进入第二阶段, 主机首先发一个 IN 令牌包, 由于端点 0 发送有效, SIE 将数据返回主机, 主机方返回一个 ACK 后, 主机发送数据的 CTR 标志置位, DIR=0, EP\_ID=0, 表明主机正确收到了用户发过去的描述符. 固件程序由此进入中断. 此时是由 IN 引起的.

函数 CTR\_LP 负责 IN 处理代码段:

```

void CTR_LP(void)
{
    .....
    /* DIR = 0 */
}

```

```

/* DIR = 0      => IN   int */
/* DIR = 0 implies that (EP_CTR_TX = 1) always */

_ClearEP_CTR_TX(ENDP0); //清除正确发送标志位
In0_Process(); //处理 INT 事件

/* before terminate set Tx & Rx status (恢复端点 0 发送接收状态) */
_SetEPRxStatus(ENDP0, SaveRState);
_SetEPTxStatus(ENDP0, SaveTState);
return;
.....
}

```

主要是调用 `In0_Process(usb_core.c)` 完成剩下的工作:

```

u8 In0_Process(void)
{
    .....
}

```

主机收到 18 个字节的描述符后, 进入状态事务过程. 此过程的令牌包为 OUT, 字节数为 0. 只需要用户回一个 ACK. 所以中断处理程序会进入 `Out0_Process(usb_core.c)`, 由于此时状态为 `WAIT_STATUS_OUT`, 所以执行以下这段:

```

u8 Out0_Process(void)
{
    .....
}

```

#### 4.6 设置地址

获取设备描述符以后, 主机再一次的复位设备, 设备又进入初始状态. 开始枚举的第二步设置地址.



执行过程是这样的：

1) 重新从复位状态开始

在第一次获取设备描述符后，程序使端 0 的发送和接收都无效，状态也设置为 STALLED，所以主机先发一个复位，使得端点 0 接收有效，虽然说在 NAK 和 STALL 状态下，端点仍然可以响应和接收 SETUP 包。

2) 设置地址的建立阶段

主机先发一个 SETUP 令牌包，控制端点寄存器的 SETUP 标志置位，然后主机发了一个 OUT 包，共 8 个字节，里面包含设置地址的要求

设备在检验数据后，发一个 ACK 握手包，同时 CTR\_RX 置位，CTR 置位，数据已经保存到 RxADDR 所指向的缓冲此时 USB 产生数据接收中断。

由于 CTR\_RX 和 SETUP 同时置位，终端处理程序调用 Setup0\_Process(usb\_core.c)，所做的作仍然是先填充 pInformation 结构，获取请求特征码、请求代码和数据长度。

由于设备地址不会携带数据，所以接下来调用 NoData\_SetUp0(usb\_core.c)：

```
void NoData_Setup0(void)
{
    .....
    ControlState = WAIT_STATUS_IN; /* After no data stage SETUP */
}
```

```
.....  
}
```

前面把状态设置为 WAIT\_STATUS\_IN 是给 IN 令牌包的处理提供指示. 因为建立令牌阶段结束以后, 主机接着发一个 IN 令牌包, 设备返回 0 字节数据包后, 进入 CTR\_LP 对应的中断.

```
u8 In0_Process(void)  
{  
    .....  
    if ((pInformation->USBbRequest == SET_ADDRESS) &&  
        (Type_Recipient == (STANDARD_REQUEST | DEVICE_RECIPIENT)))  
    {  
        SetDeviceAddress(pInformation->USBwValue0);  
        pUser_Standard_Requests->User_SetDeviceAddress();  
    }  
    (*pProperty->Process_Status_IN)();  
    ControlState = STALLED;  
    .....  
  
    pInformation->ControlState = ControlState;  
  
    return Post0_Process();  
}
```

#### 4.7 从新地址获取设备描述符





1) 上一阶段末尾的状态

端点 0 的发送和接收都设置为:STALL, 只接收 SETUP 令牌包.

2) 建立阶段:主机发令牌包, 数据包, 设备 ACK

产生数据接收中断, 且端点 0 的 SETUP 置位, 调用 Setup0\_Process(usb\_core.c) 函数进行处理. 在 Setup0\_Process 中, 因为主机发送了请求数据 8 个字节, 由调 Data\_SetUp0 函数进行处理. 首先是获取设备描述符的长度, 描述符的起始地址, 传送的最大字节数, 根据这些参数确定本次能够传输的字节数. 然后调用 DataStageIn() 数进行实际的数据传输操作, 设备描述符必须在本次中断中就写入发送缓冲区, 因为很快就要进入数据阶段了. 在函数处理的最后:

```
vSetEPTxStatus(EP_TX_VALID);  
USB_StatusOut(); /*本来期待 IN 令牌包, 但用户可以取消数据阶段, 一般不会用到 */
```

3) 数据阶段:主机发 IN 包, 设备返回数据, 主机 ACK

本操作会产生数据发送完成中断, 由 In0\_Process 来处理中断, 它也调用 DataStageIn 函数来进行处理.

如果数据已经发送完:

```
ControlState = WAIT_STATUS_01T;  
vSetEPTxStatus(EP_TX_STALL);
```



```
//转入状态阶段
有可能的话:
SendLengthData();
ControlState = LAST_IN_DATA;
Data_Mul_MaxPacketSize = FALSE; //这一次发送 0 个字节, 状态转为最后输入阶段.
```

否则, 继续准备数据, 调整剩余字节数、发送指针位置, 等待主机的下一个 IN 令牌包.

4) 状态阶段: 主机发 OUT 包、0 字节包, 设备 ACK

数据发送完成中断, 调用 Out0\_Process 函数进行处理, 由于在数据阶段的末尾已经设置设备状态为: WAIT\_STATUS\_OUT, 所以处理函数基本上没有做什么事就退出了, 并将状态设为 STALLED.

## 5. 推荐参考

### 5.1 STM32--USB 详细使用说明和原帖(抓包工具是 HD-USB12, 对应软件 HD-DATA-CENTER)

[http://blog.sina.com.cn/s/blog\\_98ee3a930100wn6m.html](http://blog.sina.com.cn/s/blog_98ee3a930100wn6m.html)

<http://forum.eet->

[cn.com/BLOG\\_ARTICLE\\_2806.HTM?jumpo=view\\_welcomead\\_forum\\_1354841784500](cn.com/BLOG_ARTICLE_2806.HTM?jumpo=view_welcomead_forum_1354841784500)

### 5.2 Universal Serial Bus(非常详细完整)

[http://wiki.osdev.org/Universal\\_Serial\\_Bus](http://wiki.osdev.org/Universal_Serial_Bus)

### 5.3 USB2.0 文档(2000-4-27 USB2.0 官方文档)

[http://www.usb.org/developers/docs/usb20\\_docs/](http://www.usb.org/developers/docs/usb20_docs/)

### 5.4 USB 枚举过程(网友整理资料的快照)

<http://blog.csdn.net/myarrow/article/details/8270029>

### 5.5 触控 USB 鼠标实验代码(论坛置顶贴可以找到相关的代码)

[www.openedv.com](http://www.openedv.com)

## 6. 附录





### 附录 A USBTRACE 的使用

USBTRACE 3.0.1.82 只要简单设置就可以捕获新插入的设备(如何捕获枚举参考随软件文档 Capturing device enumeration), 捕获结果显示在 LogView 窗口中. 每行显示一个事务(transaction), 包括了序列号类型时间戳等具体内容(参考随软件文档 Log View).

Seq	Type	Time	Request	I/O	EndPoint	Device Object	IRP	Status	Buffer Snippet
108	URB	3.348000	GET_DESCRIPTOR_FROM_DEVIC...	OUT	0	0x37821060	0x37406720	STATUS_SUCCESS	
109	URB	3.348007	GET_DESCRIPTOR_FROM_DEVIC...	OUT	0	0x3403A040	0x37406720	STATUS_SUCCESS	
110	URB	3.348044	CONTROL_TRANSFER	IN	0	0x3403A040	0x37406720	STATUS_PENDING	
111	URB	3.348225	CONTROL_TRANSFER	IN	0	0x3403A040	0x37406720	STATUS_SUCCESS	12 01 00 02 00 00 00 00
112	URB	3.348238	GET_DESCRIPTOR_FROM_DEVIC...	OUT	0	0x37821060	0x36F8A5C0	STATUS_SUCCESS	
113	URB	3.348242	GET_DESCRIPTOR_FROM_DEVIC...	OUT	0	0x3403A040	0x36F8A5C0	STATUS_SUCCESS	
114	URB	3.348291	CONTROL_TRANSFER	IN	0	0x3403A040	0x36F8A5C0	STATUS_PENDING	
115	URB	3.348452	CONTROL_TRANSFER	IN	0	0x3403A040	0x36F8A5C0	STATUS_SUCCESS	09 02 22 00 01 01 00 00
116	URB	3.348478	GET_DESCRIPTOR_FROM_DEVIC...	OUT	0	0x37821060	0x37406720	STATUS_SUCCESS	
117	URB	3.348483	GET_DESCRIPTOR_FROM_DEVIC...	OUT	0	0x3403A040	0x37406720	STATUS_SUCCESS	
118	URB	3.348509	CONTROL_TRANSFER	IN	0	0x3403A040	0x37406720	STATUS_PENDING	
119	URB	3.348711	CONTROL_TRANSFER	IN	0	0x3403A040	0x37406720	STATUS_SUCCESS	09 02 22 00 01 01 00 00
120	URB	3.348753	SELECT_CONFIGURATION	OUT	0	0x37821060	0x37406720	STATUS_SUCCESS	
121	URB	3.348768	SELECT_CONFIGURATION	OUT	0	0x37821060	0x37406720	STATUS_SUCCESS	
122	URB	3.349030	SELECT_CONFIGURATION	IN	0	0x37821060	0x37406720	STATUS_SUCCESS	
123	URB	3.349047	SELECT_CONFIGURATION	IN	0	0x37821060	0x37406720	STATUS_SUCCESS	
124	URB	3.349054	CLASS_INTERFACE	OUT	0	0x37821060	0x37406720	STATUS_SUCCESS	

图 5-1. USBTRACE 部分捕获结果

Seq 列不同颜色含义如下：

	(白)	Outgoing : From host to device.
	(绿)	Incoming (Success) : From device to host.
	(红)	Incoming (Failed) : From device to host.
	(灰)	Outgoing request returned with status Pending from the device

## 附录 B 标准设备请求命令

标准的 USB 设备请求命令是用在控制传输中的“初始设置步骤”里的数据包阶段（即 DATA0，由八个字节构成）。标准 USB 设备请求命令共有 11 个，大小都是 8 个字节，具有相同的结构，由 5 个字段（域/field）构成，结构如下（括号中的数字表示字节数，首字母 bm, b, w 分别表示位图、字节，双字节）：

bmRequestType (1)+bRequest (1)+wValue (2)+wIndex (2)+wLength (2)

各字段的意义如下：

1) **bmRequestType 域**：D7D6D5D4D3D2D1D0

D7

=0 主机到设备

=1 设备到主机；

D6D5

=00 标准请求命令

=01 类请求命令

=10 用户定义的命令

=11 保留值

D4D3D2D1D0

=00000 接收者为设备

=00001 接收者为接口

=00010 接收者为端点

=00011 接收者为其他接收者

=其他 其他值保留

2) **bRequest 域**：请求命令代码，在标准的 USB 命令中，每一个命令都定义了编号，编号的值就为字段的值，编号 与命令名称如下（要注意这里的命令代码要与其他字段结合使用，可以说命令代码是标准请求命令代码的核心，正是因为这些命令代码而决定了 11 个 USB 标准请求命令）：

0 GET\_STATUS：用来返回特定接收者的状态

1 CLEAR\_FEATURE：用来清除或禁止接收者的某些特性

3 SET\_FEATURE：用来启用或激活命令接收者的某些特性

5 SET\_ADDRESS：用来给设备分配地址

6 GET\_DESCRIPTOR：用于主机获取设备的特定描述符

7 SET\_DESCRIPTOR：修改设备中有关的描述符，或者增加新的描述符

8 GET\_CONFIGURATION：用于主机获取设备当前设备的配置值（注同上面的不同）

9 SET\_CONFIGURATION：用于主机指示设备采用的要求的配置

10 GET\_INTERFACE：用于获取当前某个接口描述符编号

11 SET\_INTERFACE：用于主机要求设备用某个描述符来描述接口

12 SYNCH\_FRAME：用于设备设置和报告一个端点的同步帧

3) **wValue 域**：用来传送当前请求的参数，随请求不同而变。

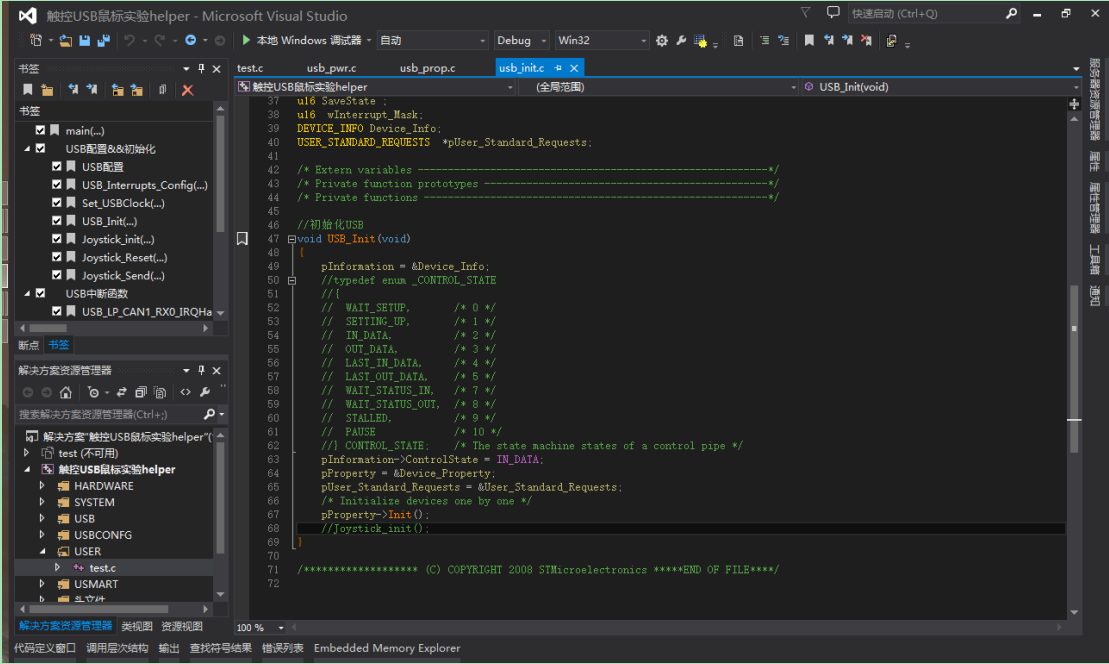
4) **wIndex 域**：当 bmRequestType 的 Recipient 字段为接口或端点时，wIndex 域用来表

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 bmRequestType 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。

### 附录 C 使用 Visual Studio 2013 进行 STM32 开发

VS2013 不愧是臃肿的 IDE! 安装插件 GDB 之后就可以识别你的 JTAG 调试代码了. 如果你的 GDB 没有激活, 使用它进行代码编辑也是不错的, 代码的编译调试工作完全可以交给其他编译器。



## 7. 后记

本文是学习 USB2.0 过程中参考网络上资料逐渐梳理(各处乱搬)出来的. 因水平很菜错误在所难免, 望不吝指教.

邮箱: funte91@163.com

QQ: 75157970

最后修改日期: 2015-3-15