USB 概述及协议

目录

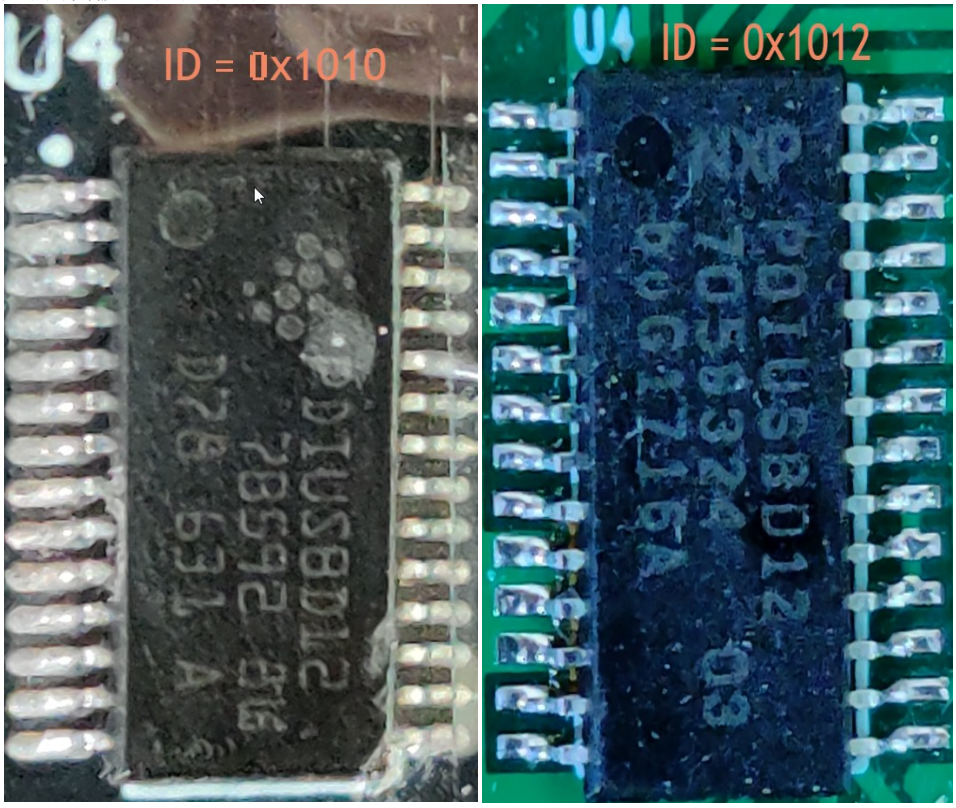
本文章主要是基于学习《圈圈教你玩USB》书籍的学习笔记,从根本上理解USB的通信过程及底层驱动的编写，为后续做USB相关的开发打下基础.

1 PDIUSBD12 Chip ID error

a) USB PHY芯片PDIUSBD12的芯片ID错误

在淘宝上购买的USB PHY芯片PDUSBD12, 编写固件读取芯片ID时获取到的ID号为0x1010, NXP官方PDIUSBD12芯片的正确ID为0x1012, 验证后发现购买到了假货.

以下为真假芯片对比图片



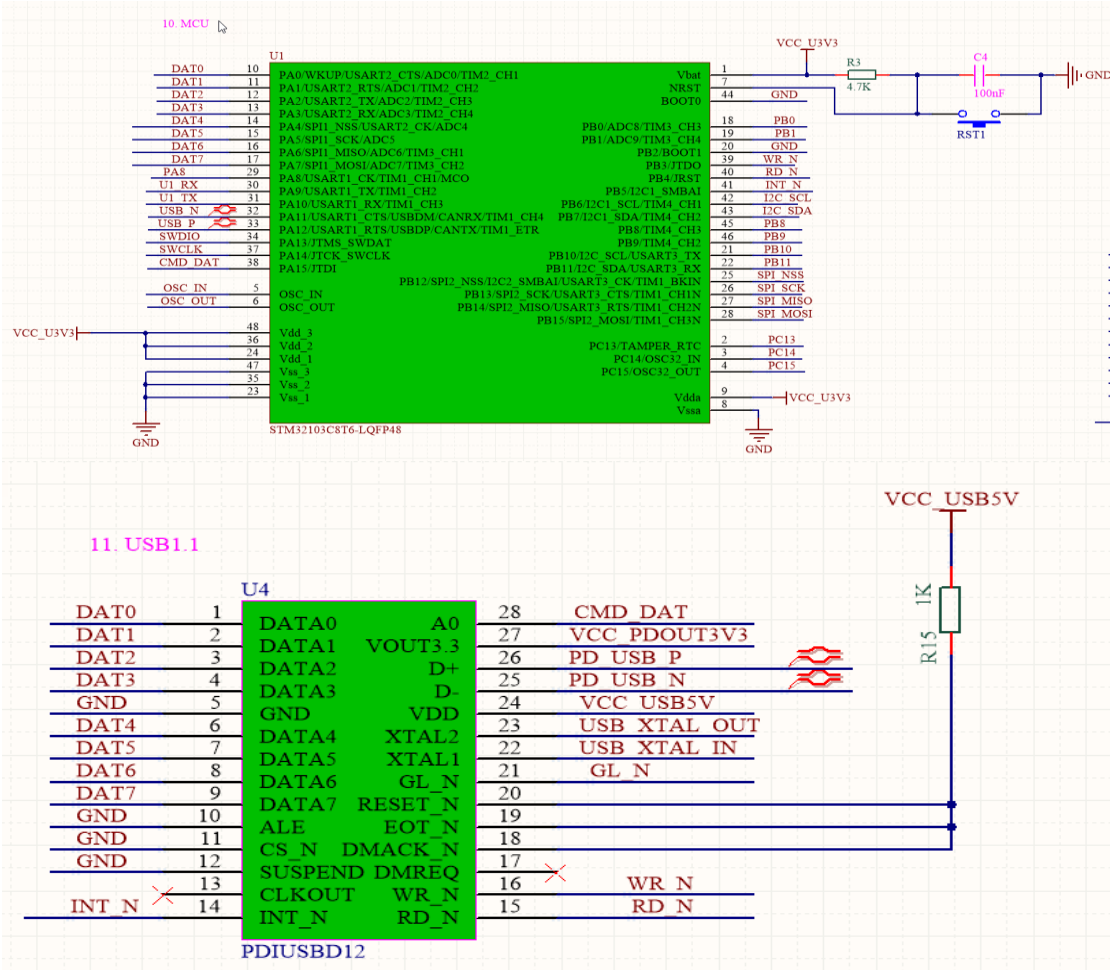
b) 固件验证

运行固件并单步调试，示波器在单步调试时测量时序控制的信号，发现信号电平及时序没有问题，但是电平信号有些许纹波，查找发现电源部分也有纹波.

```
int main(void)
{
    uint16_t D12_ID = 0;

    D12GPIOConfiguration();
    hw_USARTx_DMA_Init(USART1, 115200, USART1_SEND_BUF, USART1_RECV_BUF, USART_RxBUFFER_SIZE, DISABLE, DISABLE, ENABLE, DISABLE);

    DEBUG("\r\n**************************************************************\r\n");
    DEBUG("      STM32F10x General Config Library!\r\n");
    DEBUG("      Build Version: V%d.%d\r\n", VERSION, SUB_VERSION);
    DEBUG("      Compile Date : %s\r\n", __DATE__);
    DEBUG("      Compile Time : %s\r\n", __TIME__);
    DEBUG("      Author Name  : %s\r\n", Author_Name);
    DEBUG("**************************************************************\r\n");

    D12_ID = D12ReadID();
    DEBUG("Your PDIUSBD12 Chip ID is: 0x%x\r\n", D12_ID);

    if(0x1012 == D12_ID)
    {
        DEBUG("ID is correct! Congratulations!\r\n");
    }
    else
    {
        DEBUG("ID is incorrect! What a pity!\r\n");
    }

    while(1)
    {

    }
}
```

c) **硬件验证**

主控MCU的IO口与USB PHY芯片之间的连接



d) 验证

a) 已知OK的板子，焊接上假芯片，用已知OK的固件测试，读取芯片ID为0x1010；已知OK的板子配已知真芯片读取ID为0x1012

b) 已知OK的板子，焊接上假芯片，用已知OK的USB模拟鼠标的固件测试，在板子上电固件初始化完成后，在没有任何外部按键触发的情况下，串口打印USB处于一直触发的情况，这在OK的芯片上是没有出现的

c) 已知FAIL的板子，焊接上OK的芯片，用已知OK的固件测试，读取芯片ID为0x0000；且已知FAIL的板子配已知假芯片读取ID为0x0000

e) 总结

a) 假芯片

一些重要的芯片，还是在大网站(Digi- Key)，官网或官方代理采购，对于小批量试试能否索样.

b) 硬件设计问题

- 电源的纹波问题较严重

- JTAG 的 IO 口在设计时用做了 USB 芯片数据/命令复用选择，读写选通等功能
- 信号线通信时需要交叉的情况下，先经过斟酌后再进行图纸设计
- 数据通信的是 8bit 并口，但是官方库函数在进行端口读写时是 16bit 的，所以剩余的 8bit 如果同时在使用的话，硬件设计上需要避开(例如：PA0-PA7 作为并口数据通信，PA8-PA15 中的 PA9，PA10 同时复用为 USART 时，此时如果调用官方的库函数对端口进行读写时，会影响 USART 的打印输出)

2 USB鼠标的实现

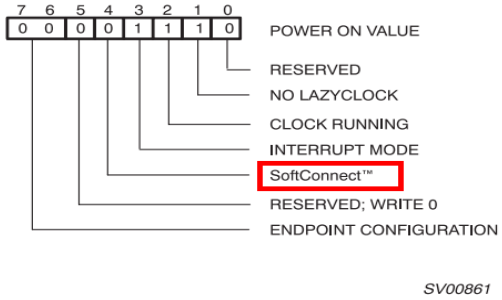
2.1) 首先需要模拟一个USB在复位的时候USB拔出然后再插入的过程(涉及到USB PHY芯片的软连接的连接及断开)通过PDUSBD12的设置模式命令(set mode)来设置芯片内部的USBDP的上拉软连接实现USB的插入检测.需要发送命令码F3之后，再发送2个字节的数据来进行设置，具体的2字节的设置数据见以下详细注释.

11.2.3 Set mode

Code (Hex) — F3

Transaction — write 2 bytes

The Set mode command is followed by two data writes. The first byte contains the configuration bits. The second byte is the clock division factor byte.



See Table 5 for bit allocation.

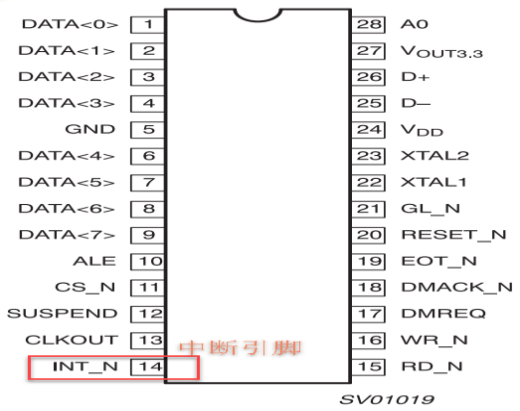
Fig 6. Set mode command, Configuration byte.

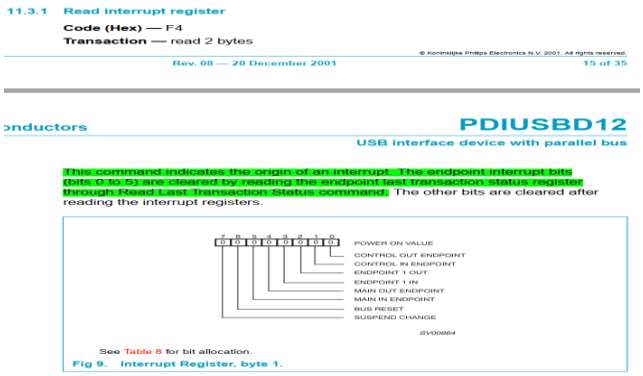
Bit	Symbol	Description
7 to 6	ENDPOINT CONFIGURATION	These two bits set the endpoint configurations as follows: mode 0 (Non-ISO mode) mode 1 (ISO-OUT mode) mode 2 (ISO-IN mode) mode 3 (ISO-I/O mode) See Section 8 “Endpoint description” for more details.
4	SoftConnect	A ‘1’ indicates that the upstream pull-up resistor will be connected if V_{BUS} is available. A ‘0’ means that the upstream resistor will not be connected. The programmed value will not be changed by a bus reset.
3	INTERRUPT MODE	A ‘1’ indicates that all errors and “NAKing” are reported and will generate an interrupt. A ‘0’ indicates that only OK is reported. The programmed value will not be changed by a bus reset.
2	CLOCK RUNNING	A ‘1’ indicates that the internal clocks and PLL are always running even during Suspend state. A ‘0’ indicates that the internal clock, crystal oscillator and PLL are stopped whenever not needed. To meet the strict Suspend current requirement, this bit needs to be set to ‘0’. The programmed value will not be changed by a bus reset.
1	NO LAZYCLOCK	A ‘1’ indicates that CLKOUT will not switch to LazyClock. A ‘0’ indicates that the CLKOUT switches to LazyClock 1ms after the Suspend pin goes HIGH. LazyClock frequency is 30 kHz ± 40%. The programmed value will not be changed by a bus reset.

```
    D12GPIOConfiguration();
    hw_USARTx_DMA_Init(USART2, 115200, USART1_SEND_BUF, USART1_RECV_BUF, USART_RxBUFFER_SIZE, DISABLE, DISABLE, ENABLE, DISABLE);

    DEBUG("\r\n*************************************************************\r\n");
    DEBUG("      USB protocol learning boards!\r\n");
    DEBUG("      FW build version: V%d.%d\r\n", VERSION, SUB_VERSION);
    DEBUG("      Compile Date : %s\r\n", __DATE__);
    DEBUG("      Compile Time : %s\r\n", __TIME__);
    DEBUG("      Author Name  : %s\r\n", Author_Name);
    DEBUG("*************************************************************\r\n");

    D12_ID = D12ReadID();
    DEBUG("Your PDIUSBD12 Chip ID is: 0x%x\r\n", D12_ID);

    if(0x1012 == D12_ID)
    {
        DEBUG("ID is correct! Congratulations!\r\n");
    }
    else
    {
        DEBUG("ID is incorrect! What a pity!\r\n");
    }

    UsbDisconnect();
    UsbConnect();

    while(1)
    {
        LEDsControl(LED1, LED_ON);
        bsp_sw_delay_ms(200);
        LEDsControl(LED1, LED_OFF);
        bsp_sw_delay_ms(200);
    }
```

```
*************************************************************************

     USB protocol learning boards!

     FW build version: V0.1

     Compile Date : Aug 24 2019

     Compile Time : 02:33:59

     Author Name  : Simon.Y

*************************************************************************

Your PDIUSBD12 Chip ID is: 0x1012

ID is correct! Congratulations!

USB disconnect.

USB connect.
```

2.2) USB的DP端的上拉软连接连接上后，相当于USB进行了插入主机，USB主机此时会发送一些获取描述符的请求，而对于USB PHY芯片而言，由于在set mode寄存器中设置了所有的中断都会有中断响应产生，这里将通过D12的中断产生后INT引脚置高对其中断寄存器中的终端标志位进行查询来查看是产生了哪种中断.





11.3.1 Read interrupt register

Code (Hex) — F4

Transaction — read 2 bytes

This command indicates the origin of an interrupt. The endpoint interrupt bits (bits 0 to 5) are cleared by reading the endpoint last transaction status register through Read Last Transaction Status command. The other bits are cleared after reading the interrupt registers.

See Table 8 for bit allocation.

Fig 9. Interrupt Register, byte 1.

2.3）对中断标志位进行判断，发现USB主机与USB设备在没有发生通信时(3ms没有数据通信)，USB的总线被挂起，挂起后USB主机对USB设备进行了复位，复位完成后USB主机与USB设备的端点0进行通信，可以看出USB主机向USB设备的端点0输出了数据.

```
if(!D12GetINTPin())
{
    InterruptSource = D12_ReadInterrupt_Register();

    if(InterruptSource & 0x80)      UsbBusSuspend();
    if(InterruptSource & 0x40)      UsbBusReset();
    if(InterruptSource & 0x20)      UsbEp2In();
    if(InterruptSource & 0x10)      UsbEp2Out();
    if(InterruptSource & 0x08)      UsbEp1In();
    if(InterruptSource & 0x04)      UsbEp1Out();
    if(InterruptSource & 0x02)      UsbEp0In();
    if(InterruptSource & 0x01)      UsbEp0Out();

}
```

```
Compile Date : Aug 26 2019
Compile Time : 12:48:47
Author Name  : Simon.Y
************************************************************
Your PDIUSBD12 Chip ID is: 0x1012
ID is correct! Congratulations!
断开USB连接.
连接USB.
Usb挂起中断.
Usb复位中断.
端点0输出.
端点0输出.
端点0输出.
```

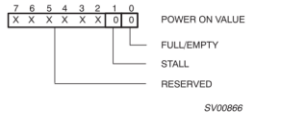
2.4）USB主机向USB设备的端点0究竟输出了什么数据呢？将数据读出来进行分析.对于USB PHY芯片D12的的数据读取方式为：(1)首先需要选取待读取数据的端点. (2)读取指定端点的数据并存放到相应的缓冲区.

11.3.2 Select Endpoint

Code (Hex) — 00 to 05

Transaction — read 1 byte (optional)

The Select Endpoint command initializes an internal pointer to the start of the selected buffer. Optionally, this command can be followed by a data read, which returns this byte.



FULL/EMPTY: A '1' indicates the buffer is full, '0' indicates an empty buffer.

STALL: A '1' indicates the selected endpoint is in the stall state.

Fig 11. Select Endpoint command: bit allocation.

11.3.5 Read buffer

Code (Hex) — F0

Transaction — read multiple bytes (max. 130)

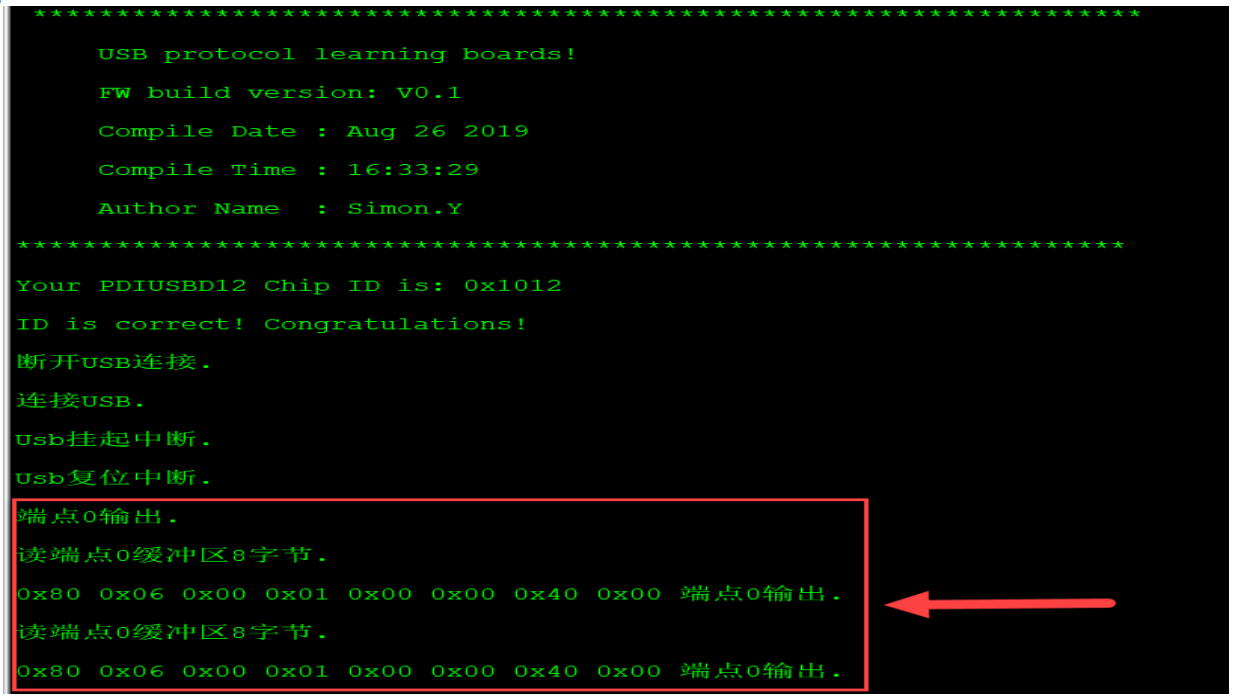
The Read Buffer command is followed by a number of data reads, which returns the contents of the selected endpoint data buffer. After each read, the internal buffer pointer is incremented by 1.

The buffer pointer is not reset to the top of the buffer by the Read Buffer command. This means that reading or writing a buffer can be interrupted by any other command (except for Select Endpoint).

The data in the buffer are organized as follows:

- byte 0: reserved; can have any value
- byte 1: number/length of data bytes
- byte 2: data byte 1
- byte 3: data byte 2
- etc.

The first two bytes will be skipped in the DMA read operation. Thus, the first read will get Data byte 1, the second read will get Data byte 2, etc. The PDIUSBD12 can determine the last byte of this packet through the EOP termination of the USB packet.



2.5)可以看出USB总线复位之后，USB主机向USB设备的端点0发送了8个字节，具体的数据为上面的截图所示。0x80, 0x06, 0x00, 0x01, 0x00, 0x00, 0x40, 0x00

2.6)对这些数据进行分析，由于是USB主机发送到USB设备的，这就涉及到USB协议的标准请求. 首先了解下USB标准设备请求的结构，摘录USB2.0协议的第9章节，可以查看USB设备的通用请求结构，可以将之前的数据与这个结构进行一一对应，来理解具体含义.

- 0x80: bmRequestType – 主机需要获取数据，数据需要从设备传向主机，请求的数据类型为标准类型，指定接收这为设备
- 0x06: bRequest – 此项由 bmrRequestType中断额D6:5的Type来修饰的特殊类型(非标准类型)，由标准需求代码可知，USB主机需要获取USB设备的描述符GET_DESCRIPTOR
- 0x00, 0x01: wValue – 此值为变量用于传输一个参数到设备，具体化需求. 两个字节，低字节表示索引号(用来指明同一种描述符中的具体某个具体描述符，如配置描述符集合中某个描述符)，高字节表示描述符的类型编号(即表9-5中的描述符类型)，这里的0x01表示主机需要获取设备描述符
- 0x00, 0x00: wIndex – 只在获取字符描述符时有用，表示字符串的语言ID，获取除字符串描述符的其它描述符时，wIndex值为0
- 0x40,0x00: wLength – 请求设备返回数据的字节数

9.4 Standard Device Requests

This section describes the standard device requests defined for all USB devices. Table 9-3 outlines the standard device requests, while Table 9-4 and Table 9-5 give the standard request codes and descriptor types, respectively.

USB devices must respond to standard device requests, even if the device has not yet been assigned an address or has not been configured.

Table 9-3. Standard Device Requests

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

9.3 USB Device Requests

All USB devices respond to requests from the host on the device's Default Control Pipe. These requests are made using control transfers. The request and the request's parameters are sent to the device in the Setup packet. The host is responsible for establishing the values passed in the fields listed in Table 9-2. Every Setup packet has eight bytes.

Table 9-2. Format of Setup Data

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

Table 9-4. Standard Request Codes

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

Table 9-5. Descriptor Types

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER[1]	8

2.7)由于USB PHY接收到数据的查看方式是通过在接收到数据后，将其INT引脚拉低并对相应的中断标识位置位一系列动作实现的，那么需要将中断标识位清零且对USB PHY中已经读取了缓存内容的缓存区进行清空，如果中断标志位没有进行清除，那么就会一直提示有中断发生.

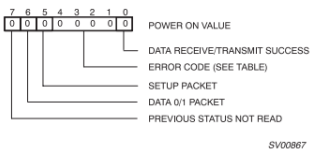
11.3.4 Read last transaction status register

Code (Hex) — 40 to 45

Transaction — read 1 byte

The Read Last Transaction Status command is followed by one data read that returns the status of the last transaction of the endpoint. This command also resets the corresponding interrupt flag in the interrupt register, and clears the status, indicating that it was read.

This command is useful for debugging purposes. Since it keeps track of every transaction, the status information is overwritten for each new transaction.



See Table 9 for bit allocation.

Fig 13. Read last transaction status register.

Table 9: Read last transaction status register: bit allocation

Bit	Symbol	Description
7	PREVIOUS STATUS NOT READ	A '1' indicates a second event occurred before the previous status was read.
6	DATA 0/1 PACKET	A '1' indicates the last successful received or sent packet had a DATA1 PID.
5	SETUP PACKET	A '1' indicates the last successful received packet had a SETUP token (this will always read '0' for IN buffers).
4 to 1	ERROR CODE	See Table 10 "Error codes".
0	DATA RECEIVE/TRANSMIT SUCCESS	A '1' indicates data has been received or transmitted successfully.

Table 10: Error codes

Error code (Binary)	Description
0000	No Error
0001	PID encoding Error; bits 7 to 4 are not the inversion of bits 3 to 0
0010	PID unknown; encoding is valid, but PID does not exist
0011	Unexpected packet; packet is not of the type expected (= token, data or acknowledge), or SETUP token to a non-control endpoint
0100	Token CRC Error
0101	Data CRC Error
0110	Time Out Error
0111	Never happens
1000	Unexpected End-Of-Packet
1001	Sent or received NAK
1010	Sent Stall, a token was received, but the endpoint was stalled

Table 10: Error codes...continued

Error code (Binary)	Description
1011	Overflow Error, the received packet was longer than the available buffer space
1101	Bitstuff Error
1111	Wrong DATA PID; the received DATA PID was not the expected one

2.8)在使用清除缓冲区命令对缓冲区进行清除之前需要确认当前Endpoint0 out收到的数据包是否为建立过程包，如果是的话，需要先发送建立包ACK(发送后认为建立包已经被正确接收，做为确保通信过程数据的正确性及完整性)，然后才能使用ClearBuffer命令清除OUT缓冲区及ValidateBuffer命令IN缓冲区.清除中断标志位及缓冲区后，由于USB主机无法收到USB设备的返回数据包，会对USB总线进行再次复位并再次发送获取设备描述符的请求，在发送大约4次之后，USB主机会因此无法加载驱动而无法识别USB设备.

```
Usb挂起中断.
Usb复位中断.
端点0输出.
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
Usb复位中断.
端点0输出.
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
Usb复位中断.
端点0输出.
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
Usb复位中断.
端点0输出.
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
```

2.9)USB主机需要USB设备返回设备描述符，整个过程总结如下图，这个完整的过程的解析，USB主机在多次复位设备之后，

无法获取到相应回包后将设备挂起。

```
Usb复位中断.

端点0输出.
Ep0输出端口接收到建立过程包(Setup Packet)
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00 bmRequestType[D7] --> 设备到主机的数据传输
bmRequestType[D6:D5] --> 标准设备请求
bmRequestType[D4:D0] --> 设备接收
bRequest --> bRequest_GET_DESCRIPTOR


Usb复位中断.

端点0输出.
Ep0输出端口接收到建立过程包(Setup Packet)
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00 bmRequestType[D7] --> 设备到主机的数据传输
bmRequestType[D6:D5] --> 标准设备请求
bmRequestType[D4:D0] --> 设备接收
bRequest --> bRequest_GET_DESCRIPTOR


端点0输出.
Ep0输出端口接收到建立过程包(Setup Packet)
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00 bmRequestType[D7] --> 设备到主机的数据传输
bmRequestType[D6:D5] --> 标准设备请求
bmRequestType[D4:D0] --> 设备接收
bRequest --> bRequest_GET_DESCRIPTOR


Usb挂起中断.
```

2.10)USB主机需要USB设备返回设备描述符，那么需要构建一个USB设备描述符通过端点0发送给USB主机。

- 首先需要明白设备描述的数据结构及相应的数据含义，标准设备描述符包含18个字节，具体含义如下图的Table9-8(摘自USB2.0官方协议).
- 有了设备描述符的包结构后，代码中需要构建一个数据结构作为数据包回包的结构.以下为数据结构相对应的意义.

```
typedef struct
{
    uint8_t bLength;                //描述符数据包字节数
    uint8_t bDescriptorType;        //描述符类型
    uint16_t bcdUSB;                //BCD码来表示USB的协议版本号
    uint8_t bDeviceClass;           //设备类代码(USB-IF分配)
    uint8_t bDeviceSubClass;        //设备子类代码(USB-IF分配)
    uint8_t bDeviceProtocol;        //设备协议码(由USB-IF分配)
    uint8_t bMaxPacketSize0;        //端点0最大的包长
    uint16_t idVendor;              //VID(厂商ID由USB-IF分配)
    uint16_t idProduct;             //产品ID(生产厂商自行分配)
    uint16_t bcdDevice;             //设备出厂编号
    uint8_t iManufacture;           //设备厂商字符串索引
    uint8_t iProduct;               //产品描述符字符串索引
    uint8_t iSerialNumber;          //设备序列号字符串索引
    uint8_t bNumConfigurations;     //当前速度下能支持的配置数量
}Device_Descriptor_t;
```

Table 9-8. Standard Device Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	DEVICE Descriptor Type
2	*bcdUSB*	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant.
4	*bDeviceClass*	1	Class	Class code (assigned by the USB-IF). If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
5	*bDeviceSubClass*	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* field. If the *bDeviceClass* field is reset to zero, this field must also be reset to zero. If the *bDeviceClass* field is not set to FFH, all values are reserved for assignment by the USB-IF.

Table 9-8. Standard Device Descriptor (Continued)

Offset	Field	Size	Value	Description
6	*bDeviceProtocol*	1	Protocol	Protocol code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* and the *bDeviceSubClass* fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class-specific protocols on a device basis. However, it may use class-specific protocols on an interface basis. If this field is set to FFH, the device uses a vendor-specific protocol on a device basis.
7	*bMaxPacketSize0*	1	Number	Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid)
8	*idVendor*	2	ID	Vendor ID (assigned by the USB-IF)
10	*idProduct*	2	ID	Product ID (assigned by the manufacturer)
12	*bcdDevice*	2	BCD	Device release number in binary-coded decimal
14	*iManufacturer*	1	Index	Index of string descriptor describing manufacturer
15	*iProduct*	1	Index	Index of string descriptor describing product
16	*iSerialNumber*	1	Index	Index of string descriptor describing the device's serial number
17	*bNumConfigurations*	1	Number	Number of possible configurations

2.11)USB设备返回设备描述符.

a) 对于D12，USB主机发送了获取设备描述符的请求，USB设备做为被动执行，其Ep0Out(端点0输出口会有数据接收)产生接收中断从而标志位置位. 通过Ep0Out端点的读缓冲区操作获取接收到的主机发送过来的数据，如下图中断额Read buffer的详细描述进行操作.

b) 然后对读取到的数据包的字段进行解析，解析出来为设备描述符的获取请求

c) 构建设备描述符的数据包(共18个字节)并进行发送，由于设备描述符的bMaxPacketSize0我们只能设置最大为16个字节(D12芯片的端点0最大包长)，所以数据需要作多次发送(一次最多只能发送16个字节)

d) 注意：当 USB设备发送数据包的长度为端点最大包长的整数倍时，需要发送一个0长度包作为数据结束包给USB主机.(由于USB设备发送设备描述符给USB主机后，USB主机可以了解到USB设备的端点的最大包长为16个字节,但是USB主机不通过解析数据包中的每个包长字节来做判断数据是否结束，毕竟有时会存在丢帧或者丢数据的可能性，而是当数据包为发送端点最大包长的整数倍时，在没有多余的数据发送时需要发送一个0长度的数据包做为结束包.打个比方，USB设备需要发送48个字节数据包，但是每次只能发送16个字节长度，需要连续发送3次才能发送完成，虽然USB主机在接收到第一个包的时候知道需要接收48个字节的数据包，但是USB主机不会去统计总长度，因为可能有丢数据的可能性，比如了丢了2个字节，但是USB设备在发送完48个字节后，再发送一个0长度数据包，在USB主机接收到这个包后，可以判断丢包率).

e) 在设备描述符的数据包构建好了之后，通过端点0将数据包发送出去UsbEp0SendData函数发送数据出去，这里注意，USB端点的发送和接收缓冲区其实是共用的，可以看Datasheet中的命令代码是共用一个.

f) 当发送数据包长度大于端点最大包长时，第一次在Ep0Out的ISR中断中解析处主机获取设备描述符之后，先发送最大包长16个字节后，缓冲区数据发送完成后，后面的数据或者数据包在主机发送了输入令牌包之后产生UsbEp0In中断，然后再UsbEp0InISR中发送剩余的数据包或者数据即可.

11.3.5 Read buffer

Code (Hex) — F0

Transaction — read multiple bytes (max. 130)

The Read Buffer command is followed by a number of data reads, which returns the contents of the selected endpoint data buffer. After each read, the internal buffer pointer is incremented by 1.

The buffer pointer is not reset to the top of the buffer by the Read Buffer command. This means that reading or writing a buffer can be interrupted by any other command (except for Select Endpoint).

The data in the buffer are organized as follows:

- byte 0: reserved; can have any value
- byte 1: number/length of data bytes
- byte 2: data byte 1
- byte 3: data byte 2
- etc.

The first two bytes will be skipped in the DMA read operation. Thus, the first read will get Data byte 1, the second read will get Data byte 2, etc. The PDIUSBD12 can determine the last byte of this packet through the EOP termination of the USB packet.

```
Your PDIUSBD12 Chip ID is: 0x1012

ID is correct! Congratulations!

断开USB连接.
USB连接.
<-----USB挂起中断.----->

<=====USB复位中断=====>
<-----Ep0输出中断.----->
通信建立包
Read Ep0 buffer 8 bytes.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
USB标准输入请求====>获取描述符：设备描述符.
```

```
void UsbEp0SendData(void)
{
    // 数据通过Ep0发送出去，最大发送的包长为端点0限定的最大包长
    if(SendLength > Device_Descriptor.bMaxPacketSize0)
    {
        D12WriteEndpointBuffer(1, Device_Descriptor.bMaxPacketSize0, pSendData);
        SendLength -= Device_Descriptor.bMaxPacketSize0;
        pSendData += Device_Descriptor.bMaxPacketSize0;
    }
    else
    {
        if(SendLength != 0)
        {
            D12WriteEndpointBuffer(1, SendLength, pSendData);
            SendLength = 0;
        }
        else
        {
            if(1 == NeedZeroPacket)
            {
                D12WriteEndpointBuffer(1, 0, pSendData);
                NeedZeroPacket = 0;
            }
        }
    }
}
```

```
Your PDIUSBD12 Chip ID is: 0x1012

ID is correct! Congratulations!

断开USB连接.
USB连接.
<-----USB挂起中断.----->

<=====USB复位中断=====>
<-----Ep0输出中断.----->
通信建立包
Read Ep0 buffer 8 bytes.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
USB标准输入请求=====>获取描述符：设备描述符.
Write Ep0 buffer 16 bytes
0x12 0x01 0x10 0x01 0x00 0x00 0x00 0x10 0x88 0x88 0x01 0x00 0x00 0x01 0x01 0x02

<-----Ep0输出中断.----->
普通数据包
Read Ep0 buffer 0 bytes.

<-----Ep0输入完成中断.----->
Write Ep0 buffer 2 bytes
0x03 0x01
<=====USB复位中断=====>
```

1. USB主机需要获取设备描述符
2. USB设备返回了第一个16字节的数据包
3. 接收到了USB主机的输入令牌包
4. 发送剩余的2个字节到USB主机

2.12)USB主机设置USB设备的地址，因为USB的地址为7位的，所以最多能同时挂128个USB设备，但是由于USB设备刚接入在没有分配USB地址时，使用的是USB地址0的端点0进行通信的，所以最大USB地址只能是127个，地址0需要预留给新接入的USB设备.USB主机设置USB设备地址的数据包中会带有USB主机分配给USB设备的USB地址. 在接收到设置地址包之后，固件操作D12芯片将获取到的USB地址写入到USB设备中，以便下次使用这个地址通信，设置完地址后，USB主机会使用新设置的地址与USB设备进行通信，重新获取设备描述符.

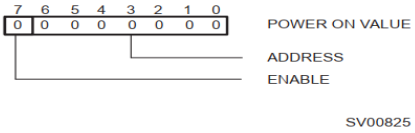
```
<=====USB复位中断=====>
<-----Ep0输出中断.----->
通信建立包
Read Ep0 buffer 8 bytes.
0x00 0x05 0x1d 0x00 0x00 0x00 0x00 0x00
USB标准输出请求====>设置地址：0x1d
Write Ep0 buffer 0 bytes

<-----Ep0输入完成中断.----->
<-----Ep0输出中断.----->
通信建立包
Read Ep0 buffer 8 bytes.
0x80 0x06 0x00 0x01 0x00 0x00 0x12 0x00
USB标准输入请求====>获取描述符：设备描述符.
Write Ep0 buffer 16 bytes
0x12 0x01 0x10 0x01 0x00 0x00 0x00 0x10 0x88 0x88 0x01 0x00 0x00 0x01 0x01 0x02
```

11.2.1 Set Address/Enable

Code (Hex) — D0

Transaction — write 1 byte

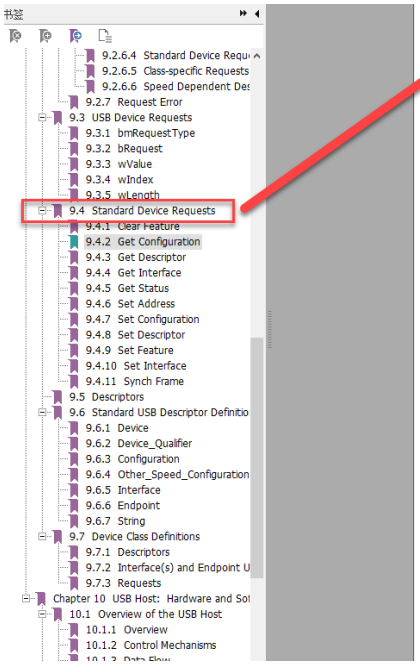
This command is used to set the USB assigned address and enable the function.



ADDRESS: The value written becomes the address.
ENABLE: A '1' enables this function.

Fig 4. Set Address/Enable command: bit allocation.

2.13)USB主机需要获取配置描述符.以下为USB的标准设备请求的格式及USB配置描述符的数据格式



9.4.3 Get Descriptor

This request returns the specified descriptor if the descriptor exists.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID (refer to Section 9.6.7)	Descriptor Length	Descriptor

The *wValue* field specifies the descriptor type in the high byte (refer to Table 9-5) and the descriptor index in the low byte. The descriptor index is used to select a specific descriptor (only for configuration and string descriptors) when several descriptors of the same type are implemented in a device. For example, a device can implement several configuration descriptors. For other standard descriptors that can be retrieved via a GetDescriptor() request, a descriptor index of zero must be used. The range of values used for a descriptor index is from 0 to one less than the number of descriptors of that type implemented by the device.

The *wIndex* field specifies the Language ID for string descriptors or is reset to zero for other descriptors. The *wLength* field specifies the number of bytes to return. If the descriptor is longer than the *wLength* field, only the initial bytes of the descriptor are returned. If the descriptor is shorter than the *wLength* field, the device indicates the end of the control transfer by sending a short packet when further data is requested. A short packet is defined as a packet shorter than the maximum payload size or a zero length data packet (refer to Chapter 5).

The standard request to a device supports three types of descriptors: device (also device_qualifier), configuration (also other_speed_configuration), and string. A high-speed capable device supports the device_qualifier descriptor to return information about the device for the speed at which it is not operating (including *wMaxPacketSize* for the default endpoint and the number of configurations for the other speed). The other_speed_configuration returns information in the same structure as a configuration descriptor, but for a configuration if the device were operating at the other speed. A request for a configuration descriptor returns the configuration descriptor, all interface descriptors, and endpoint descriptors for all of the

Table 9-10. Standard Configuration Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	CONFIGURATION Descriptor Type
2	*wTotalLength*	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class- or vendor-specific) returned for this configuration.
4	*bNumInterfaces*	1	Number	Number of interfaces supported by this configuration
5	*bConfigurationValue*	1	Number	Value to use as an argument to the SetConfiguration() request to select this configuration
6	*iConfiguration*	1	Index	Index of string descriptor describing this configuration

Table 9-10. Standard Configuration Descriptor (Continued)

Offset	Field	Size	Value	Description
7	*bmAttributes*	1	Bitmap	Configuration characteristics D7: Reserved (set to one) D6: Self-powered D5: Remote Wakeup D4...0: Reserved (reset to zero) D7 is reserved and must be set to one for historical reasons. A device configuration that uses power from the bus and a local source reports a non-zero value in *bMaxPower* to indicate the amount of bus power required and sets D6. The actual power source at runtime may be determined using the GetStatus(DEVICE) request (see Section 9.4.5). If a device configuration supports remote wakeup, D5 is set to one.
8	*bMaxPower*	1	mA	Maximum power consumption of the USB device from the bus in this specific configuration when the device is fully operational. Expressed in 2 mA units (i.e., 50 = 100 mA). Note: A device configuration reports whether the configuration is bus-powered or self-powered. Device status reports whether the device is currently self-powered. If a device is disconnected from its external power source, it updates device status to indicate that it is no longer self-powered. A device may not increase its power draw from the bus, when it loses its external power source, beyond the amount reported by its configuration. If a device can continue to operate when disconnected from its external power source, it continues to do so. If the device cannot continue to operate, it fails operations it can no longer support. The USB System Software may determine the cause of the failure by checking the status and noting the loss of the device's power source.

2.14)USB主机需要获取字符串描述符.字符串描述符包含厂商字符串描述符，产品字符串描述符及序列号字符串描述符，字符串描述符的index对应了设备描述符中的iManufacture, iProduct, iSerialNumber. 字符串描述符的字符内容编码都是unicode编码的，还有一种特殊的零字符描述符，指定设备支持的语言.

Table 9-16. UNICODE String Descriptor

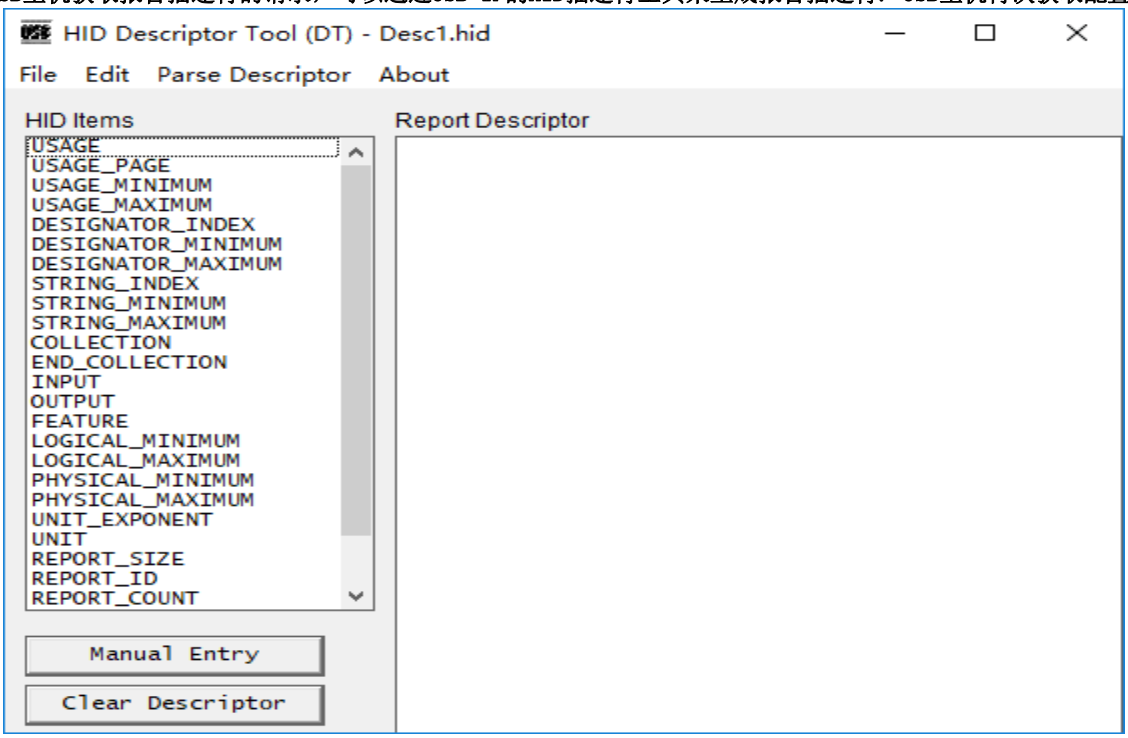
Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	STRING Descriptor Type
2	*bString*	N	Number	UNICODE encoded string

Table 9-15. String Descriptor Zero, Specifying Languages Supported by the Device

Offset	Field	Size	Value	Description
0	*bLength*	1	N+2	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	STRING Descriptor Type
2	*wLANGID[0]*	2	Number	LANGID code zero
...	...	...	...	...
N	*wLANGID[x]*	2	Number	LANGID code *x*

2.15)USB主机重新获取设备描述符，分两次获取配置描述符，完成后USB主机会对USB设备进行设置配置，设置配置的目的是通过非零的配置值来使能USB设备的非零端点，使能非零端点后就可以通过非零端点进行数据通信了.设置配置确认后USB设备需要给USB主机发送ACK来进行确认.

2.16)USB主机对USB设备进行USB类输出请求(因为USB设备需要作为USB HID设备使用)，输出请求USB设备设置空闲；然后USB主机获取报告描述符的请求，可以通过USB-IF的HID描述符工具来生成报告描述符. USB主机再次获取配置描述符.



2.17)USB设备的按键中断的方式来模拟鼠标的各种功能，例如：光标左右，上下移动及鼠标的按键等等.这里有一点需要注意，报告描述符中的鼠标光标及按键定义的顺序与后面USB通过端点返回的中断传输方式的报告的数据的顺序是一一对应的，且还有一个知识点是USB主机输出时，可以通过控制端点(Endpoint0)，但是HID鼠标设备回报数据时，必须通过之前进行设置配置的非零端点号进行数据回包(即USB输入).还有一个问题是鼠标按键按下回报数据时，按键按下和按键释放是两个不同的动作，会进行两次数据上报.

```
/*******************************************************************************
* Function Name  : EXTI15_10_IRQHandler
* Description    : This function handles External lines 15 to 10 interrupt request.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void EXTI15_10_IRQHandler(void)
{
    // Mouse button right
    if(EXTI_GetITStatus(EXTI_Line10) != RESET)
    {
        bsp_sw_delay_ms(10);
        if(RESET == GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_10)){
            report_data[0] |= 0x02;
        }else{
            report_data[0] &= ~0x02;
        }
        // Clear the EXTI line 10 pending bit
        EXTI_ClearITPendingBit(EXTI_Line10);
    }

    if(0 != ConfigValue){
        D12WriteEndpointBuffer(Ep1_IN, sizeof(report_data), report_data);
        LEDsControl(LED2, LED_ON);
    }
}
```

2.18)USB HID设备的报告描述符解析

以下为一个鼠标的报告描述符为例来了解下描述符中的16进制数据的具体含义，一般的网文多是如上的报告描述符，据说是用一个工具产生的，有一些还配上了中文的说明，但也多是描述性的，而不是拆分解释这些字符所代表的含义，比如，数组第一行的 0x05, 0x01, // USAGE_PAGE (Generic Desktop)

我们一眼就能看出双斜线后面的文字，是对前面数字的说明，即 0x05,0x01 所表达的是 USAGE_PAGE (Generic Desktop) 的含义但是，为何如此表达，则描述的不太清楚，对于熟悉的人而言，这自然不是问题，但对于新手，可就要费点功夫了。0x05, 0x01,代表 USAGE_PAGE(Generic Desktop)的功能，是由《device class definition for human interface device (HID)》第 24 页的 Report Descriptors 规定的。

分两部分，0x05 为一部分，表示前缀，0x01 为数据部分。

0x05 转换成二进制，就是 0000 01001，按照 HID 类协议 5.3 generic item format 的定义，这个字节被分成 3 个部分，bit0~bit1 代表的是这个前缀后面跟的数据长度，两位可以表示最大 4 字节的数据，即 bsize;bit2~bit3 代表的是这个前缀的类型，总共可以有三种类型：0=main，1=global，2=local，3=reserved；bit4~bit7 代表前缀的 tag，一般分为 input(二进制的 1000 00 nn,即 bit4~bit7=1000,代表一个 tag，bit2~bit3=00，代表 main，bit0~bit1=nn，代表这

个前缀后面还有 nn 所代表的数据)，output(二进制的 1001 00 nn)，feature(1011 00 nn)，collection(1010 00 nn)，end collection(1100 00 nn)，遵照这个原则，我们就可以解析 0x05 所表达的含义。

0x05 转换为二进制就是 0000 0101,其高 4 位全为 0，表示的 tag 为 usage page tag(协议 45 页)，bit2~bit3=01，表示的是类型，从协议中可以知道，这是一个全局类型的 item（协议 36 页），bit0~bit1=01，表示的是这个前缀后面跟着的数据长度为 1 字节，即 0x05 后面，有 0x01 作为这个前缀的数据部分，而 0x01 表示的是 general desktop page(《universal serial bus HID usage table》第五页，目录)，因此，这两个数字合起来就是 USAGE_PAGE (Generic Desktop)的含义。

用一个比较形象的描述方式来描述：每一个报告描述符需要描述的是 USB 设备的结构以及实现的功能.

1) 首先报告描述符需要一个 Usage page 来描述一个最外层的用途页类来表示 USB 设备的用途大类(Page ID)
2) 对于上述的用途大类，需要进一步详述说明 USB 设备的具体用途 Usage(Usage ID 来申明这个用途)，如 Keyboard
3) step1 中的 Usage ID 是一个键盘用途，需要进一步对这个用途的类型进行说明分解以便主机找到驱动，而 Usage type(用途类型)包含 Collection(集合)，Control(控制)及 Data(数据)这些基本的用途类型
4) 总结：step1 中在定义了 Usage Page 后，通过 step2 中的 Usage ID 对其进行说明(可以对应到 HID 协议中)，在 step3 在用途类型 Usage type 中对 Usage page 进行详细说明

```
code char MouseReportDescriptor[63] = {
      0x05, 0x01,             // USAGE_PAGE (Generic Desktop)      前缀字节(bTag, bType, bSize) 0x05, Usage page 0x01
      0x09, 0x06,             // USAGE (Keyboard)      前缀字节(bTag, bType, bSize) 0x09, Usage page 0x01 下的 Usage ID 0x06
      0xa1, 0x01,             // COLLECTION (Application)

      0x05, 0x07,             //   USAGE_PAGE (Keyboard)
      0x19, 0xe0,             //   USAGE_MINIMUM (Keyboard LeftControl)
      0x29, 0xe7,             //   USAGE_MAXIMUM (Keyboard Right GUI)
      0x15, 0x00,             //   LOGICAL_MINIMUM (0)
      0x25, 0x01,             //   LOGICAL_MAXIMUM (1)
      0x75, 0x01,             //   REPORT_SIZE (1)
      0x95, 0x08,             //   REPORT_COUNT (8)
      0x81, 0x02,             //   INPUT (Data,Var,Abs)
      0x95, 0x01,             //   REPORT_COUNT (1)
      0x75, 0x08,             //   REPORT_SIZE (8)
      0x81, 0x03,             //   INPUT (Cnst,Var,Abs)

      0x95, 0x05,             //   REPORT_COUNT (5)
      0x75, 0x01,             //   REPORT_SIZE (1)
      0x05, 0x08,             //   USAGE_PAGE (LEDs)
      0x19, 0x01,             //   USAGE_MINIMUM (Num Lock)
      0x29, 0x05,             //   USAGE_MAXIMUM (Kana)
      0x91, 0x02,             //   OUTPUT (Data,Var,Abs)

      0x95, 0x01,             //   REPORT_COUNT (1)
      0x75, 0x03,             //   REPORT_SIZE (3)
      0x91, 0x03,             //   OUTPUT (Cnst,Var,Abs)

      0x95, 0x06,             //   REPORT_COUNT (6)
      0x75, 0x08,             //   REPORT_SIZE (8)
      0x15, 0x00,             //   LOGICAL_MINIMUM (0)
      0x25, 0xFF,             //   LOGICAL_MAXIMUM (255)
      0x05, 0x07,             //   USAGE_PAGE (Keyboard)
      0x19, 0x00,             //   USAGE_MINIMUM (Reserved (no event indicated))
      0x29, 0x65,             //   USAGE_MAXIMUM (Keyboard Application)
      0x81, 0x00,             //   INPUT (Data,Ary,Abs)

      0xc0                    // END_COLLECTION

    };
```

2.19)USB通信层次实质

USB的通信过程实质上是从两个层面来进行的(或者理解为实现的)，分别为USB底层驱动层以及设备与主机之间的通信数据交互层.

1) USB驱动层(协议层用于驱动加载)：USB主机能实现对USB设备的驱动加载完成.
2) USB数据通信层(基于协议基础上的数据层)：USB主机及USB设备之间进行数据的交互，已实现某些特定数据的传输，识别及显示.

3 CDC的实现

3.1) CDC的定义

CDC(Communication Device Class)即为通信设备类，其是通过USB协议中的一个设备类，设备类字段为0x02.设备类代码可以在USB-IF的官网上找到.具体网页地址参考：https://www.usb.org/defined-class-codes.

Defined Class Codes

USB defines class code information that is used to identify a device's functionality and to nominally load a device driver based on that functionality. The information is contained in three bytes with the names Base Class, SubClass, and Protocol. (Note that 'Base Class' is used in this description to identify the first byte of the Class Code triple. That terminology is not used in the USB Specification). There are two places on a device where class code information can be placed.One place is in the Device Descriptor, and the other is in Interface Descriptors. Some defined class codes are allowed to be used only in a Device Descriptor, others can be used in both Device and Interface Descriptors, and some can only be used in Interface Descriptors. The table below shows the currently defined set of Base Class values, what the generic usage is, and where that Base Class can be used (either Device or Interface Descriptors or both).

Last Update: June 15, 2016

Base Class	Descriptor Usage	Description
00h	Device	Use class information in the Interface Descriptors
01h	Interface	Audio
02h	Both	Communications and CDC Control
03h	Interface	HID (Human Interface Device)
05h	Interface	Physical
06h	Interface	Image
07h	Interface	Printer
08h	Interface	Mass Storage
09h	Device	Hub

3.2) CDC底层驱动实现

a) 基于USB Keyboard的固件，修改设备描述符中的设备类为0x02(参考上面的图片中CDC类)

b) CDC类的子类及所用的协议必须指定为0x00

c) 字符串描述符自定义

d) 配置描述符：使用两个接口的，一个作为bulk的数据传输(端点2)，一个作为参数设置的(端点0)；使用端点2作为bulk数据传输的依据：由于USB的数据发送和接收都是以数据包的格式进行的，所以在发送和接收是就会存在封包及解包的过程，为了实现串口的全双工及避免数据丢失，需要用到USB端点的数据缓存区，而端点2的USB端点缓存区缓存空间最大，为64bytes且为双缓存.

e) CDC接口描述符(USB的接口需要一个数据类接口依附，例如HID设备中的接口描述符需要HID描述符一样)，要实现VCP功能，需要在接口描述符中指定使用CDC类,ACM(Abstract Control Model)子类及Common AT Commands协议.

f) 类特殊接口描述符(功能描述符)

3.3) 类请求实现

a)在上面的部分实现后，USB host识别除了USB device作为CDC的虚拟串口.

b) USB host在成功安装USB device的CDC驱动后会发出设置配置的输出请求，然后是USB类输入请求，有没当前FW没有实

现所以出现timeout,然后USB host又发送了USB类输入请求，同样是由于没有实现而timeout

c)基于USB host的这些请求实现相应的应答.了解USB类输入及输出请求中的数据包所代表的意思，参考USB-IF的USB CDC协议的page30(0x21: GET_LINE_CODING, 0x22:SET_CONTROL_LINE_STATE),其中GET_LINE_CODING为设置CDC串口通信的参数(波特率，停止位，校验及数据位数)，SET_CONTROL_LINE_STATE主要用于串口的流控制的，没有用到.

6.2 Management Element Requests

The Communications Interface Class supports class-specific requests, defined in this document and in the communications subclass specifications. This section describes the requests that are specific to the Communications Interface Class and common to several subclasses. These requests are sent over the management element and can apply to different device views as defined by the Communications Class interface codes. Detailed descriptions of those requests are provided in those subclass specifications. Table 19 provides a summary table of the Request Code values.

Table 19: Class-Specific Request Codes

Request Code	Value	reference
SEND_ENCAPSULATED_COMMAND	00h	6.2.1
GET_ENCAPSULATED_RESPONSE	01h	6.2.2
SET_COMM_FEATURE	02h	[USBPSTN1.2]
GET_COMM_FEATURE	03h	[USBPSTN1.2]
CLEAR_COMM_FEATURE	04h	[USBPSTN1.2]
RESERVED (future use)	05h-0Fh	
SET_AUX_LINE_STATE	10h	[USBPSTN1.2]
SET_HOOK_STATE	11h	[USBPSTN1.2]
PULSE_SETUP	12h	[USBPSTN1.2]
SEND_PULSE	13h	[USBPSTN1.2]
SET_PULSE_TIME	14h	[USBPSTN1.2]
RING_AUX_JACK	15h	[USBPSTN1.2]
RESERVED (future use)	16h-1Fh	
SET_LINE_CODING	20h	[USBPSTN1.2]
GET_LINE_CODING	21h	[USBPSTN1.2]
SET_CONTROL_LINE_STATE	22h	[USBPSTN1.2]
SEND_BREAK	23h	[USBPSTN1.2]
RESERVED (future use)	24h-2Fh	
SET_RINGER_PARMS	30h	[USBPSTN1.2]
GET_RINGER_PARMS	31h	[USBPSTN1.2]
SET_OPERATION_PARMS	32h	[USBPSTN1.2]
GET_OPERATION_PARMS	33h	[USBPSTN1.2]

6.3 Management Element Notifications

This section defines the Communications Interface Class notifications that the device uses to notify the host of interface, or endpoint events, that are common to several Communications subclasses. Detailed descriptions of those notifications are provided below and in those subclass specifications. Table 20 provides a summary table of the notification code values.

Table 20: Class-Specific Notification Codes

Notification Code	Value	Reference
NETWORK_CONNECTION	00h	6.3.1
RESPONSE_AVAILABLE	01h	6.3.2
RESERVED (future use)	02h-07h	
AUX_JACK_HOOK_STATE	08h	[USBPSTN1.2]
RING_DETECT	09h	[USBPSTN12]
RESERVED (future use)	0Ah-1Fh	
SERIAL_STATE	20h	[USBPSTN1.2]
RESERVED (future use)	21h-27h	
CALL_STATE_CHANGE	28h	[USBPSTN1.2]
LINE_STATE_CHANGE	29h	[USBPSTN1.2]
CONNECTION_SPEED_CHANGE	2Ah	6.3.3
RESERVED	2Bh-3Fh	
MDML SEMANTIC-MODEL-SPECIFIC NOTIFICATION	40h-5Fh	[USBWMC1.1]
RESERVED (future use)	60h-FFh	

6.3.10 SetLineCoding

This request allows the host to specify typical asynchronous line-character formatting properties, which may be required by some applications. This request applies to asynchronous byte stream data class interfaces and endpoints; it also applies to data transfers both from the host to the device and from the device to the host.

bmRequestType	bRequestCode	wValue	wIndex	wLength	Data
00100001B	SET_LINE_CODING	Zero	Interface	Size of Structure	Line Coding Structure

For the definition of valid properties, see Table 17, Section 6.3.111.1.1.

6.3.11 GetLineCoding

This request allows the host to find out the currently configured line coding.

bmRequestType	bRequestCode	wValue	wIndex	Wlength	Data
10100001B	GET_LINE_CODING	Zero	Interface	Size of Structure	Line Coding Structure

Table 17: Line Coding Structure

Offset	Field	Size	Value	Description
0	*dwDTERate*	4	Number	Data terminal rate, in bits per second.

Offset	Field	Size	Value	Description
4	*bCharFormat*	1	Number	Stop bits 0 - 1 Stop bit 1 - 1.5 Stop bits 2 - 2 Stop bits
5	*bParityType*	1	Number	Parity 0 - None 1 - Odd 2 - Even 3 - Mark 4 - Space
6	*bDataBits*	1	Number	Data bits (5, 6, 7, 8 or 16).

6.3.12 SetControlLineState

This request generates RS-232/V.24 style control signals.

bmRequestType	bRequestCode	wValue	wIndex	WLength	Data
00100001B	SET_CONTROL_LINE _STATE	Control Signal Bitmap	Interface	Zero	None

Table 18: Control Signal Bitmap Values for SetControlLineState

Bit position	Description
D15..D2	RESERVED (Reset to zero)
D1	Carrier control for half duplex modems. This signal corresponds to V.24 signal 105 and RS-232 signal RTS. 0 - Deactivate carrier 1 - Activate carrier The device ignores the value of this bit when operating in full duplex mode.
D0	Indicates to DCE if DTE is present or not. This signal corresponds to V.24 signal 108/2 and RS-232 signal DTR. 0 - Not Present 1 - Present

6.5.4 SerialState

This notification sends asynchronous notification of UART status.

bmRequestType	bNotification	wValue	wIndex	wLength	Data
10100001B	SERIAL_STATE	Zero	Interface	2	UART State bitmap

The *Data* field is a bitmapped value that contains the current state of carrier detect, transmission carrier, break, ring signal, and device overrun error. These signals are typically found on a UART and are used for communication status reporting. A state is considered enabled if its respective bit is set to 1.

SerialState is used like a real interrupt status register. Once a notification has been sent, the device will reset and re-evaluate the different signals. For the consistent signals like carrier detect or transmission carrier, this will mean another notification will not be generated until there is a state change. For the irregular signals like break, the incoming ring signal, or the overrun error state, this will reset their values to zero and again will not send another notification until their state changes.