USB 基本知识

USB 的重要关键字:

1、端点：位于 USB 设备或主机上的一个数据缓冲区，用来存放和发送 USB 的各种数据，每一个端点都有惟一的确定地址，有不同的传输特性（如输入端点、输出端点、配置端点、批量传输端点）

2、帧：时间概念，在 USB 中，一帧就是 1MS，它是一个独立的单元，包含了一系列总线动作，USB 将 1 帧分为好几份，每一份中是一个 USB 的传输动作。

3、上行、下行：设备到主机为上行，主机到设备为下行

下面以一问一答的形式开始学习吧。

问题一：USB 的传输线结构是如何的呢？

答案一：一条 USB 的传输线分别由地线、电源线、D+、D-四条线构成，D+和 D-是差分输入线，它使用的是 3.3V 的电压（注意哦，与 CMOS 的 5V 电平不同），而电源线和地线可向设备提供 5V 电压，最大电流为 500MA（可以在编程中设置的，至于硬件的实现机制，就不要管它了）。

问题二：数据是如何在 USB 传输线里面传送的

答案二：数据在 USB 线里传送是由低位到高位发送的。

问题三：USB 的编码方案？

答案三：USB 采用不归零取反来传输数据，当传输线上的差分数据输入 0 时就取反，输入 1 时就保持原值，为了确保信号发送的准确性，当在 USB 总线上发送一个包时，传输设备就要进行位插入***作（即在数据流中每连续 6 个 1 后就插入一个 0），从而强迫 NRZI 码发生变化。这个了解就行了，这些是由专门硬件处理的。

问题四：USB 的数据格式是怎么样的呢？

答案四：和其他的一样，USB 数据是由二进制数字串构成的，首先数字串构成域（有七种），域再构成包，包再构成事务（IN、OUT、SETUP），事务最后构成传输（中断传输、并行传输、批量传输和控制传输）。下面简单介绍一下域、包、事务、传输，请注意他们之间的关系。

（一）域：是 USB 数据最小的单位，由若干位组成（至于是多少位由具体的域决定），域可分为七个类型：

1、同步域（SYNC），八位，值固定为 0000 0001，用于本地时钟与输入同步

2、标识域（PID），由四位标识符+四位标识符反码构成，表明包的类型和格式，这是一个很重要的部分，这里可以计算出，USB 的标识码有 16 种，具体分类请看问题五。

3、地址域（ADDR）：七位地址，代表了设备在主机上的地址，地址 000 0000 被命名为零地址，是任何一个设备第一次连接到主机时，在被主机配置、枚举前的默认地址，由此可以知道为什么一个 USB 主机只能接 127 个设备的原因。

4、端点域（ENDP），四位，由此可知一个 USB 设备有的端点数量最大为 16 个。

5、帧号域（FRAM），11 位，每一个帧都有一个特定的帧号，帧号域最大容量 0x800，对于同步传输有重要意义（同步传输为四种传输类型之一，请看下面）。

6、数据域（DATA）：长度为 0~1023 字节，在不同的传输类型中，数据域的长度各不相同，但必须为整数个字节的长度

7、校验域（CRC）：对令牌包和数据包（对于包的分类请看下面）中非 PID 域进行校验的一种方法，CRC 校验在通讯中应用很泛，是一种很好的校验方法，至于具体的校验方法这里就不多说，请查阅相关资料，只须注意 CRC 码的除法是模 2 运算，不同于 10 进制中的除法。

（二）包：由域构成的包有四种类型，分别是令牌包、数据包、握手包和特殊包，前面三种是重要的包，不同的包的域结构不同，介绍如下

1、令牌包：可分为输入包、输出包、设置包和帧起始包（注意这里的输入包是用于设置输入命令的，输出包是用来设置输出命令的，而不是放据数的）

其中输入包、输出包和设置包的格式都是一样的：

SYNC+PID+ADDR+ENDP+CRC5（五位的校验码）　　（令牌包）

（上面的缩写解释请看上面域的介绍，PID 码的具体定义请看问题五）

帧起始包的格式：

SYNC+PID+11 位 FRAM+CRC5（五位的校验码）　　（帧起始包）

2、数据包：分为 DATA0 包和 DATA1 包，当 USB 发送数据的时候，当一次发送的数据长度大于相应端点的容量时，就需要把数据包分为好几个包，分批发送，DATA0 包和 DATA1 包交替发送，即如果第一个数据包是 DATA0，那第二个数据包就是 DATA1。但也有例外情况，在同步传输中（四类传输类型中之一），所有的数据包都是为 DATA0，格式如下：

SYNC+PID+0~1023 字节+CRC16　　（数据包）

3、握手包：结构最为简单的包，格式如下

SYNC+PID　　（握手包）

（注上面每种包都有不同类型的，USB1.1 共定义了十种包，具体请见问题五）

（三）事务：分别有 IN 事务、OUT 事务和 SETUP 事务三大事务，每一种事务都由令牌包、数据包、握手包三个阶段构成，这里用阶段的意思是因为这些包的发送是有一定的时间先后顺序的，事务的三个阶段如下：

1、令牌包阶段：启动一个输入、输出或设置的事务

2、数据包阶段：按输入、输出发送相应的数据

3、握手包阶段：返回数据接收情况，在同步传输的 IN 和 OUT 事务中没有这个阶段，这是比较特殊的。

事务的三种类型如下（以下按三个阶段来说明一个事务）：

1、 IN 事务：

令牌包阶段——主机发送一个 PID 为 IN 的输入包给设备，通知设备要往主机发送数据；

数据包阶段——设备根据情况会作出三种反应（要注意：数据包阶段也不总是传送数据的，根据传输情况还会提前进入握手包阶段）

1） 设备端点正常，设备往入主机里面发出数据包（DATA0 与 DATA1 交替）；

2） 设备正在忙，无法往主机发出数据包就发送 NAK 无效包，IN 事务提前结束，到了下一个 IN 事务才继续；

3） 相应设备端点被禁止，发送错误包 STALL 包，事务也就提前结束了，总线进入空闲状态。

握手包阶段——主机正确接收到数据之后就会向设备发送 ACK 包。

2、 OUT 事务：

令牌包阶段——主机发送一个 PID 为 OUT 的输出包给设备，通知设备要接收数据；

数据包阶段——比较简单，就是主机会设备送数据，DATA0 与 DATA1 交替

握手包阶段——设备根据情况会作出三种反应

1）设备端点接收正确，设备往入主机返回 ACK，通知主机可以发送新的数据，如果数据包发生了 CRC 校验错误，将不返回任何握手信息；

2） 设备正在忙，无法往主机发出数据包就发送 NAK 无效包，通知主机再次发送数据；

3） 相应设备端点被禁止，发送错误包 STALL 包，事务提前结束，总线直接进入空闲状态。

3、SETUT 事务：

令牌包阶段——主机发送一个 PID 为 SETUP 的输出包给设备，通知设备要接收数据；

数据包阶段——比较简单，就是主机会设备送数据，注意，这里只有一个固定为 8 个字节的 DATA0 包，这 8 个字节的内容就是标准的 USB 设备请求命令（共有 11 条，具体请看问题七）

握手包阶段——设备接收到主机的命令信息后，返回 ACK，此后总线进入空闲状态，并准备下一个传输（在 SETUP 事务后通常是一个 IN 或 OUT 事务构成的传输）

（四）传输：传输由 OUT、IN、SETUP 事务其中的事务构成，传输有四种类型，中断传输、批量传输、同步传输、控制传输，其中中断传输和批量转输的结构一样，同步传输有最简单的结构，而控制传输是最重要的也是最复杂的传输。

1、中断传输：由 OUT 事务和 IN 事务构成，用于键盘、鼠标等 HID 设备的数据传输中

2、批量传输：由 OUT 事务和 IN 事务构成，用于大容量数据传输，没有固定的传输速率，也不占用带宽，当总线忙时，USB 会优先进行其他类型的数据传输，而暂时停止批量转输。

3、同步传输：由 OUT 事务和 IN 事务构成，有两个特殊地方，第一，在同步传输的 IN 和 OUT 事务中是没有返回包阶段的；第二，在数据包阶段所有的数据包都为 DATA0

4、控制传输：最重要的也是最复杂的传输，控制传输由三个阶段构成（初始设置阶段、可选数据阶段、状态信息步骤），每一个阶段可以看成一个的传输，也就是说控制传输其实是由三个传输构成的，用来于 USB 设备初次加接到主机之后，主机通过控制传输来交换信息，设备地址和读取设备的描述符，使得主机识别设备，并安装相应的驱动程序，这是每一个 USB 开发者都要关心的问题。

1、初始设置步骤：就是一个由 SET 事务构成的传输

2、可选数据步骤：就是一个由 IN 或 OUT 事务构成的传输，这个步骤是可选的，要看初始

设置步骤有没有要求读/写数据（由 SET 事务的数据包阶段发送的标准请求命令决定）

3、 状态信息步骤：顾名思义，这个步骤就是要获取状态信息，由 IN 或 OUT 事务构成构成的传输，但是要注意这里的 IN 和 OUT 事务和之前的 INT 和 OUT 事务有两点不同：

1） 传输方向相反，通常 IN 表示设备往主机送数据，OUT 表示主机往设备送数据；在这里，IN 表示主机往设备送数据，而 OUT 表示设备往主机送数据，这是为了和可选数据步骤相结合；

2） 在这个步骤里，数据包阶段的数据包都是 0 长度的，即 SYNC+PID+CRC16

除了以上两点有区别外，其他的一样，这里就不多说

（思考：这些传输模式在实际***作中应如何通过什么方式去设置？）

题五：标识码有哪些？

答案五：如同前面所说的标识码由四位数据组成，因此可以表示十六种标识码，在 USB1.1 规范里面，只用了十种标识码，USB2.0 使用了十六种标识码，标识码的作用是用来说明包的属性的，标识码是和包联系在一起的，首先简单介绍一下数据包的类型，数据包分为令牌包、数据、握手包和特殊包四种（具体分类请看问题七），标识码分别有以下十六种：

令牌包 ：

0x01 输出(OUT）启动一个方向为主机到设备的传输，并包含了设备地址和标号

0x09 输入 (IN) 启动一个方向为设备到主机的传输，并包含了设备地址和标号

0x05 帧起始（SOF）表示一个帧的开始，并且包含了相应的帧号

0x0d 设置（SETUP）启动一个控制传输，用于主机对设备的初始化

数据包 ：

0x03　偶数据包（DATA0），

0x0b　奇数据包（DATA1）

握手包:

0x02　确认接收到无误的数据包（ACK）

0x0a　无效，接收（发送）端正在忙而无法接收（发送）信息

0x0e　错误，端点被禁止或不支持控制管道请求

特殊包 0x0C　前导，用于启动下行端口的低速设备的数据传输

问题六：USB 主机是如何识别 USB 设备的？

答案六：当 USB 设备插上主机时，主机就通过一系列的动作来对设备进行枚举配置（配置是属于枚举的一个态，态表示暂时的状态），这这些态如下：

1、接入态（Attached）：设备接入主机后，主机通过检测信号线上的电平变化来发现设备的接入；

2、供电态（Powered）：就是给设备供电，分为设备接入时的默认供电值，配置阶段后的供电值（按数据中要求的最大值，可通过编程设置）

3、缺省态（Default）：USB 在被配置之前，通过缺省地址 0 与主机进行通信；

4、地址态（Address）：经过了配置，USB 设备被复位后，就可以按主机分配给它的唯一地址来与主机通信，这种状态就是地址态；

5、配置态（Configured）：通过各种标准的 USB 请求命令来获取设备的各种信息，并对设备的某此信息进行改变或设置。

6、挂起态（Suspended）：总线供电设备在 3ms 内没有总线***作，即 USB 总线处于空闲状态的话，该设备就要自动进入挂起状态，在进入挂起状态后，总的电流功耗不超过 280UA。

问题七：刚才在答案四提到的标准的 USB 设备请求命令究竟是什么？

答案七：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），请看回问答四的内容。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 个字段构成（字段是标准请求命令的数据部分），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：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 接收者为设备

=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：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） 0　GET_STATUS：用来返回特定接收者的状态

1） 1　CLEAR_FEATURE：用来清除或禁止接收者的某些特性

2） 3　SET_FEATURE：用来启用或激活命令接收者的某些特性

3） 5　SET_ADDRESS：用来给设备分配地址

4） 6　GET_DEscriptOR：用于主机获取设备的特定描述符

5） 7　SET_DEscriptOR：修改设备中有关的描述符，或者增加新的描述符

6） 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）

7） 9　SET_CONFIGURATION：用于主机指示设备采用的要求的配置

8） 10　GET_INTERFACE：用于获取当前某个接口描述符编号

9） 11　SET_INTERFACE：用于主机要求设备用某个描述符来描述接口

10） 12 SYNCH_FRAME：用于设备设置和报告一个端点的同步帧

以上的 11 个命令要说得明白真的有一匹布那么长，请各位去看书吧，这里就不多说了，控制传输是 USB 的重心，而这 11 个命令是控制传输的重心，所以这 11 个命令是重中之重，这个搞明白了，USB 就算是入门了。

问题八：在标准的 USB 请求命令中，经常会看到 Descriptor，这是什么来的呢？

回答八：Descriptor 即描述符，是一个完整的数据结构，可以通过 C 语言等编程实现，并存储在 USB 设备中，用于描述一个 USB 设备的所有属性，USB 主机是通过一系列命令来要求设备发送这些信息的。它的作用就是通过如问答节中的命令***作来给主机传递信息，从而让主机知道设备具有什么功能、属于哪一类设备、要占用多少带宽、使用哪类传输方式及数据量的大小，只有主机确定了这些信息之后，设备才能真正开始工作，所以描述符也是十分重要的部分，要好好掌握。标准的描述符有 5 种，USB 为这些描述符定义了编号：

1——设备描述符

2——配置描述符

3——字符描述符

4——接口描述符

5——端点描述符

上面的描述符之间有一定的关系，一个设备只有一个设备描述符，而一个设备描述符可以包含多个配置描述符，而一个配置描述符可以包含多个接口描述符，一个接口使用了几个端点，就有几个端点描述符。这间描述符是用一定的字段构成的，分别如下说明：

1、设备描述符

```
struct  _DEVICE_DEscriptOR_STRUCT
{
  BYTE  bLength;                  //设备描述符的字节数大小，为 0x12
  BYTE  bDescriptorType;   //描述符类型编号，为 0x01
```

WORD bcdUSB; //USB 版本号

BYTE bDeviceClass; //USB 分配的设备类代码，0x01~0xfe 为标准设备类，0xff 为厂商自定义类型

//0x00 不是在设备描述符中定义的，如 HID

BYTE bDeviceSubClass; //usb 分配的子类代码，同上，值由 USB 规定和分配的

BYTE bDeviceProtocl; //USB 分配的设备协议代码，同上

BYTE bMaxPacketSize0; //端点 0 的最大包的大小

WORD idVendor; //厂商编号

WORD idProduct; //产品编号

WORD bcdDevice; //设备出厂编号

BYTE iManufacturer; //描述厂商字符串的索引

BYTE iProduct; //描述产品字符串的索引

BYTE iSerialNumber; //描述设备序列号字符串的索引

BYTE bNumConfiguration; //可能的配置数量

}

2、配置描述符

struct _CONFIGURATION_DEscriptOR_STRUCT

{

```
 BYTE  bLength;              //设备描述符的字节数大小，为 0x12
 BYTE  bDescriptorType;  //描述符类型编号，为 0x01
 WORD  wTotalLength;      //配置所返回的所有数量的大小
 BYTE  bNumInterface;     //此配置所支持的接口数量
 BYTE  bConfigurationVale;    //Set_Configuration 命令需要的参数值
 BYTE  iConfiguration;        //描述该配置的字符串的索引值
 BYTE  bmAttribute;           //供电模式的选择
 BYTE  MaxPower;              //设备从总线提取的最大电流
}
```

3、字符描述符

```
struct  _STRING_DEscriptOR_STRUCT
{
 BYTE  bLength;              //设备描述符的字节数大小，为 0x12
 BYTE  bDescriptorType;  //描述符类型编号，为 0x01
 BYTE  SomeDescriptor[36];          //UNICODE 编码的字符串
}
```

4、接口描述符

```
struct  _INTERFACE_DEscriptOR_STRUCT
```

{

BYTE bLength; //设备描述符的字节数大小，为 0x12

BYTE bDescriptorType; //描述符类型编号，为 0x01

BYTE bInterfaceNunber; //接口的编号

BYTE bAlternateSetting;//备用的接口描述符编号

BYTE bNumEndpoints; //该接口使用端点数，不包括端点 0

BYTE bInterfaceClass; //接口类型

BYTE bInterfaceSubClass;//接口子类型

BYTE bInterfaceProtocol;//接口所遵循的协议

BYTE iInterface; //描述该接口的字符串索引值

}

5、端点描述符

struct _ENDPOIN_DEscriptOR_STRUCT

{

BYTE bLength; //设备描述符的字节数大小，为 0x12

BYTE bDescriptorType; //描述符类型编号，为 0x01

BYTE bEndpointAddress; //端点地址及输入输出属性

BYTE bmAttribute; //端点的传输类型属性

WORD wMaxPacketSize; //端点收、发的最大包的大小 BYTE bInterval; //主机

查询端点的时间间隔}

在搞明白了上面的八个问题之后，就可以进入 USB 的下一步学习了。

【 基于 STM32 的 USB 程序开发笔记 】

目前市场上 USB 设备的种类繁多，但是这些设备会有一些共同的特性，根据这些特性可以把 USB 设备划分为不同的类，如显示设备、通信设备、音频设备、大容量存储设备、**人机接口设备（HID）。**

HID 类设备属于人机交互操作的设备。如 USB 鼠标，USB 键盘，USB 游戏操纵杆，USB 触摸板，USB 轨迹球、电话拨号设备、VCR 遥控等等设备。用于控制计算机操作的一些方面。(从 Windows98 操作系统开始，为 HID 类设备提供了通用的驱动程序，所以只要按照 HID 设备类的规范编写设备的固件程序，就能够让 Windows 系统自动识别设备，省去了复杂的驱动程序编写过程。)

使用 HID 设备的一个好处就是，操作系统自带了 HID 类的驱动程序，而用户无需去开发很麻烦的驱动程序，只要直接使用 API 调用即可完成通信。所以很多简单的 USB 设备，喜欢枚举成 HID 设备，这样就可以不用安装驱动而直接使用。

USB 设备有 4 种传输方式与主机进行通信： 控制方式、中断方式、批量方式和同步方式。

HID 只支持控制和中断传输方式。如图 2 所示，HID 设备必须要有默认的控制管道和一个中断输入端点；中断输出端点是可选的。

- 端点是地址，管道是路径；

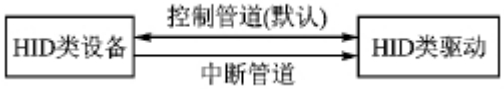


图 2　HID 类设备使用控制和中断传输方式

2. STM32 的参考手册，这对于设备底层 USB 的硬件配置以及事件驱动机制的了解尤为重要，你需要了解各个寄存器的功能以及如何操作，比如 CNTR、ISTR、EPnR、DADDR 等等，如果你想学习 USB，这个手册是必须的。
3. USB2.0 协议；第 9 章 USB Device Framework 的详细理解对于你的 USB Device 固件开发不可缺少（这里就是 STM32）。
4. ST 提供的 USB 固件库，这个类库较为散乱，但不可不参考；
5. USB 设备（DEVICE）从来只是被动触发，USB 主机（HOST）掌握主动权，发送什么数据，什么时候发送，是给设备数据还是从设备请求数据，都是由 USB 主机完成的，USB 设备只是配合主机完成设备的枚举、数据方向和大小。根据数据特性再决定该不该回复该如何回复、该不该接收该如何接收这些动作。

【修改 STM32 的 USB 例程为自己所用】

1. USB 设备和主机的通信需要先建立 virtual pipeline(管道)，然后设备**传回描述符给主机**。joystick 属于 HID 设备，工作在中断方式。但并非通常单片机所谓的中断，而实际上是查询的方式。

2. USB 数据 是由 二进制数字串 构成的；
   二进制数字串 ➔ 域（有七种） ➔ 包 ➔ 事务（IN、OUT、SETUP） ➔ 传输（中断传输、并行传输、批量传输和控制传输）。
3. 而描述符就需要**程序**来提供了。
   - usb_desc.c： 提供了设备、端点、接口、字符串、群组、制造商描述符；
   - usb_prop.c： 提供了 Device_Property（性能），Device_Table &USER_STANDARD_REQUEST（请求）结构描述，这 3 个东西定义于 usb_core.c；
   - hw_config.c： 提供了实际硬件需要的操作函数，Joystick_Send（）通过函数 UserTo**PMA**BufferCopy 和 SetEPTxValid 将坐标值发给了 USB 端口。
   - STM32f10x_it.c：里面有

【我们想实现一个 USB 功能，可以拿例子来改，那么具体要改哪些地方呢 ？】

- 首先要改各种描述符，然后是具体的数据处理。
  我们拿 USB 摇杆鼠标范例➔ 改成 USB 键盘。

USB 主机是通过请求设备的相关描述符来判断设备类型的，所以我们只需要修改相关描述符就能实现我们想要设备类型。描述符的配置位于 Descriptor.h 和 Descriptor.c 文件中。

步骤一：设备描述符。范例中的 USB 设备描述符如下：

```
/* USB Standard Device Descriptor */
const u8 Joystick_DeviceDescriptor[JOYSTICK_SIZ_DEVICE_DESC]=
{
 0x12,                  /*bLength */
 USB_DEVICE_DESCRIPTOR_TYPE, /*bDescriptorType*/
 0x00,                  /*bcdUSB */
 0x02,
 0x00,                  /*bDeviceClass*/
 0x00,                  /*bDeviceSubClass*/
 0x00,                  /*bDeviceProtocol*/
 0x40,                  /*bMaxPacketSize40*/
 0x83,  （低字节） /*idVendor (0x0483)    VID*/
 0x04,  （高字节）
 0x10,                  /*idProduct = 0x5710   PID*/
 0x57,
 0x00,                  /*bcdDevice rel. 2.00*/
 0x02,
 1,  /*Index of string descriptor describing  manufacturer */
 2, /*Index of string descriptor describing  product*/
 3,     /*Index of string descriptor describing the device serial number */
 0x01           /*bNumConfigurations*/
};
```

修改：修改这里的 idVendor（即 VID）和 idProduct（即 PID）即可。它们是用来供电脑端识别设备以加载驱动用的，所以必须不能跟现有的设备相冲突。

VID 和 PID 都是两字节，低字节在前，高字节在后。像这里的 VID 为 0x0483，写在里面就是

0x83,0x04。我们将 VID 改成 0x1234，将 PID 改成 0x4321，即：0x34，0x12，0x21，0x43。

步骤二：配置描述符（集合）。

配置描述符集合包括配置描述符、接口描述符、类特殊描述符（这里是 HID 描述符）、以及端点描述符。

如果你需要增加端点，那么在最后增加就行了，注意要记得修改 JOYSTICK_SIZ_CONFIG_DESC 的值为配置描述符集合的长度。

第一部分为配置描述符。**通常这里不需要修改**，除非你要改成该配置有多个接口（USB 复合设备），那么应该修改 bNumInterfaces，需要多少个就改成多少个，这里只有一个接口，所以值为 1。

第二部分为接口描述符，在接口描述符中**决定**该接口所实现的功能，例如 HID 设备，或者是大容量存储设备等等。

（bInterfaceNumber）：为该接口的**编号**，从 0 开始。这里只有一个接口，所以它的值为 0，如果又更多的接口，则依次编号。注意一个接口完整结束（包括该接口下的类特殊描述符和端点描述符）后，才可以开始一个新的接口。

（bNumEndpoints）：为该接口所使用的**端点数目**（不包括端点 0），原来的程序是实现鼠标功能的，所以只有一个输入端点。我们这里增加一个输出端点，用来控制 LED（键盘上有大写字母锁定、小键盘数字键锁定等指示灯），因此将 bNumEndpoints 改为 2。

（bInterfaceClass）：为接口所使用的**类**，这里指定为 HID 设备，USB 键盘和鼠标都是 HID 设备，这里不用修改，如果你要**实现其它设备**，请根据 **USB 协议**所规定的类来修改。

（bInterfaceSubClass）：为接口所使用的子类，在 HID 设备类下规定了两种子类，系统引导时**能用的**和**不能用的**，这里为 1，表示系统引导时能使用。

（bInterfaceProtocol）：为接口的协议，原来为鼠标，这里改为 1，键盘。

第三部分为 HID 描述符，只有 HID 设备才有，如果你要修改成其它设备，则用其它设备的类特殊描述符代替或者没有，在这里不用做修改。

第四部分为输入端点 1 的端点描述符，原来代码中，设置的端点最大包长度（wMaxPacketSize）为 4 字节，我们将其改成 8 字节。另外，我们再增加一个输出端点 1，将最后的输入端点 1 描述符复制一份，然后修改地址（bEndpointAddress）为 0x01，这表示该端点为输出端点，地址为 1。由 bEndpointAddress 的最高位表示方向，1 为输入，0 为输出，最后 4 位表示地址。最后，要记得在 usb_desc.h 文件中修改 JOYSTICK_SIZ_CONFIG_DESC 的长度为 41，因为我们增加了 7 字节。

（实际修改好的配置）描述符集合如下：

```
/* USB Configuration Descriptor */
/* All Descriptors (Configuration, Interface, Endpoint, Class, Vendor */
const u8 Joystick_ConfigDescriptor[JOYSTICK_SIZ_CONFIG_DESC] =
{
  //以下为配置描述符
  0x09, /* bLength: Configuation Descriptor size */
  USB_CONFIGURATION_DESCRIPTOR_TYPE, /* bDescriptorType: Configuration */
  JOYSTICK_SIZ_CONFIG_DESC,
  /* wTotalLength: Bytes returned */
```

```
  0x00,
  0x01,                 /*bNumInterfaces: 1 interface*/
  0x01,                 /*bConfigurationValue: Configuration value*/
  0x00,            /*iConfiguration: Index of string descriptor describing
                        the configuration*/
  0xC0,                 /*bmAttributes: self powered */
  0x32,                 /*MaxPower 100 mA: this current is used for detecting V
bus*/
  //以下为接口描述符
  /************** Descriptor of Joystick Mouse interface ***************/
  /* 09 */
  0x09,                 /*bLength: Interface Descriptor size*/
  USB_INTERFACE_DESCRIPTOR_TYPE,/*bDescriptorType: Interface descriptor type*/
  0x00,                 /*bInterfaceNumber: Number of Interface*/
  0x00,                 /*bAlternateSetting: Alternate setting*/
  0x02,                 /*bNumEndpoints*/
  0x03,                 /*bInterfaceClass: HID*/
  0x01,                 /*bInterfaceSubClass : 1=BOOT, 0=no boot*/
  0x01,                 /*bInterfaceProtocol : 0=none, 1=keyboard, 2=mouse*/
  0,                       /*iInterface: Index of string descriptor*/
  //以下为 HID 描述符
  /********************* Descriptor of Joystick Mouse HID ********************/
  /* 18 */
  0x09,                 /*bLength: HID Descriptor size*/
  HID_DESCRIPTOR_TYPE, /*bDescriptorType: HID*/
  0x00,                 /*bcdHID: HID Class Spec release number*/
  0x01,
  0x00,                 /*bCountryCode: Hardware target country*/
  0x01,                 /*bNumDescriptors: Number of HID class descriptors to f
ollow*/
  0x22,                 /*bDescriptorType*/
  JOYSTICK_SIZ_REPORT_DESC,/*wItemLength: Total length of Report descriptor*/
  0x00,

  //以下为输入端点 1 描述符
  /********************* Descriptor of Joystick Mouse endpoint ****************
/
  /* 27 */
  0x07,                   /*bLength: Endpoint Descriptor size*/
  USB_ENDPOINT_DESCRIPTOR_TYPE, /*bDescriptorType:*/
  0x81,                   /*bEndpointAddress: Endpoint Address (IN)*/
  0x03,                   /*bmAttributes: Interrupt endpoint*/
  0x08,                   /*wMaxPacketSize: 8 Byte max */
  0x00,
```

```
  0x20,                   /*bInterval: Polling Interval (32 ms)*/
  //以下为输出端但 1 描述符
  /* 34 */
  0x07,                   /*bLength: Endpoint Descriptor size*/
  USB_ENDPOINT_DESCRIPTOR_TYPE, /*bDescriptorType:*/
  0x01,                   /*bEndpointAddress: Endpoint Address (OUT)*/
  0x03,                   /*bmAttributes: Interrupt endpoint*/
  0x08,                   /*wMaxPacketSize: 8 Byte max */
  0x00,
  0x20,                   /*bInterval: Polling Interval (32 ms)*/
/* 41 */
};
```

步骤三：报告描述符，报告描述符比较复杂，直接给出修改好的报告描述符如下：

```
const u8 Joystick_ReportDescriptor[JOYSTICK_SIZ_REPORT_DESC] =
{
  0x05, 0x01, // USAGE_PAGE (Generic Desktop)
  0x09, 0x06, // USAGE (Keyboard)
  0xa1, 0x01, // COLLECTION (Application)
  0x05, 0x07, //         USAGE_PAGE (Keyboard/Keypad)
  0x19, 0xe0, //         USAGE_MINIMUM (Keyboard LeftControl)
  0x29, 0xe7, //         USAGE_MAXIMUM (Keyboard Right GUI)
  0x15, 0x00, //         LOGICAL_MINIMUM (0)
  0x25, 0x01, //         LOGICAL_MAXIMUM (1)
  0x95, 0x08, //         REPORT_COUNT (8)
  0x75, 0x01, //         REPORT_SIZE (1)
  0x81, 0x02, //         INPUT (Data,Var,Abs)
  0x95, 0x01, //         REPORT_COUNT (1)
  0x75, 0x08, //         REPORT_SIZE (8)
  0x81, 0x03, //         INPUT (Cnst,Var,Abs)
  0x95, 0x06, //     REPORT_COUNT (6)
  0x75, 0x08, //     REPORT_SIZE (8)
  0x25, 0xFF, //     LOGICAL_MAXIMUM (255)
  0x19, 0x00, //     USAGE_MINIMUM (Reserved (no event indicated))
  0x29, 0x65, //     USAGE_MAXIMUM (Keyboard Application)
  0x81, 0x00, //         INPUT (Data,Ary,Abs)
  0x25, 0x01, //         LOGICAL_MAXIMUM (1)
  0x95, 0x05, //     REPORT_COUNT (5)
  0x75, 0x01, //     REPORT_SIZE (1)
  0x05, 0x08, //     USAGE_PAGE (LEDs)
  0x19, 0x01, //     USAGE_MINIMUM (Num Lock)
  0x29, 0x02, //     USAGE_MAXIMUM (Caps Lock)
  0x91, 0x02, //     OUTPUT (Data,Var,Abs)
  0x95, 0x01, //     REPORT_COUNT (1)
```

```
  0x75, 0x06, //      REPORT_SIZE (6)
  0x91, 0x03, //      OUTPUT (Cnst,Var,Abs)
  0xc0                // END_COLLECTION
};
```

该报告描述符说明输入报告为 8 字节，**第一字节为特殊键，用位图表示，第二字节保留，第三至第八字节为普通按键。**我们将原来的摇杆功能改成键盘上的 4 个方向键，中键选择键为回车键，另外 KEY2 和 KEY3 分别做大写字母锁定键和数字锁锁定键。输出报告为 1 字节，其中最低两位分别为 Num Lock 灯和 Caps Lock 灯。

Joystick_StringLangID 描述符不用修改，Joystick_StringVendor(卖主)、Joystick_StringProduct 分别为厂商字符串和设备字符串，不改也可以，但是显示出来就是原来的内容，最好还是自己修改下。这里使用的是 Unicode 编码，可以直接使用圈圈以前写小程序自动生成该描述符，该工具的地址为：http://computer00.21ic.org/user1/2198/archives/2007/42769.html。Joystick_StringSerial 为产品序列号，它也是 Unicode 编码，这里可以不用修改，当然你修改也可以。这里我将厂商字符串改成“电脑圈圈的家当”，产品字符串改成“电脑圈圈修改的简易 USB 键盘”。好了，描述符改完了。

步骤四：修改数据处理。我们启用了一个新的端点，端点 1 输出，原来的程序中并未对它进行初始化，所以我们需要先增加对端点 1 输出的初始化。在 usb_prop.c 文件中，找到 void Joystick_Reset(void)函数，该函数是负责初始化端点的。原来对端点 1 输入的初始化设置为 4 字节，我们将它改成 8 字节。并增加对端点输出的初始化，

最终修改的代码部分如下：

```
    /* Initialize Endpoint In 1 */
    SetEPType(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, 8);                  //设置发送的长度
//    SetEPRxStatus(ENDP1, EP_RX_DIS);
    SetEPTxStatus(ENDP1, EP_TX_NAK);       //设置端点处于忙状态

    /* Initialize Endpoint Out 1 */
    SetEPRxAddr(ENDP1, ENDP1_RXADDR);      //设置接收数据的地址
    SetEPRxCount(ENDP1, 1);    //设置接收长度

    SetEPRxStatus(ENDP1, EP_RX_VALID);       //设置端点有效，可以接收数据
```

需要在 usb_conf.h 中增加对 ENDP1_RXADDR 的定义：

```
#define ENDP1_RXADDR                (0xD8)
```

步骤五：修改原来在 main 函数中发送数据的处理。

这里我们使用圈圈前几天写的按键及摇杆驱动（见 http://blog.ednchina.com/computer00/142610/message.aspx）。

修改主循环中的内容如下：

```
    while (1)
    {
```

```
        DelayXms(5);                //延时 5ms
        KeyScan();                  //扫描一次键盘
        if (KeyUp||KeyDown)
        {
            Joystick_Send(KeyPress); //发送按键
            KeyUp="0";                //清除事件
            KeyDown="0";
        }
    }
```

步骤六：在 hw_config.c 中修改 Joystick_Send 函数，根据不同的按键来发送按键情况，具体怎么修改这里就不说了，最后使用函数 UserToPMABufferCopy 将缓冲区中的数据复制到端点 1 的输出缓冲中，再使用函数 SetEPTxValid(ENDP1)使端点 1 数据有效，从而发送出去。对于输出，我们还需要增加一个回调函数来处理，因为原来的输出端点 1 的回调函数是个空函数。在 usb_conf.h 中找到#define EP1_OUT_Callback NOP_Process 一行，它将端点 1 输出回调函数定义为空处理函数。我们将它删除，换成我们自己的回调处理函数：void EP1_OUT_Callback(void);。然后回到 main.c 中增加该函数的实际代码，它主要用来控制 LED 的状态。在使用 LED 之前，当然要记得初始化这些 IO 口为输出状态，以及使能 PC 口的时钟，还有前面的键盘扫描也要增加对相应的 IO 口初始化，这些初始化代码在 void Set_System(void)函数中处理。LED 连接在 PC 口上，在 stm32f10x_conf.h 文件中，将#define _GPIOC 宏使能，原本该宏是被注释掉的，这样会提示 GPIOC 没有定义。

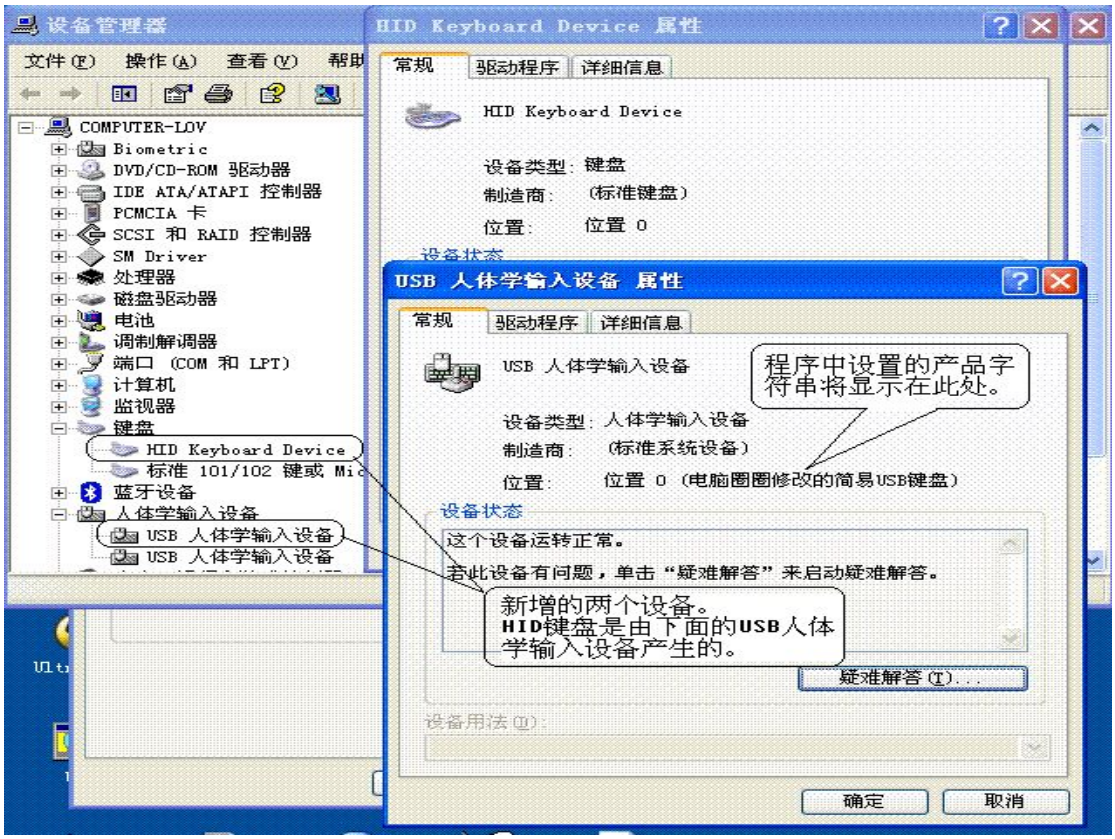
处理接收数据的**回调函数**和**发送数据的函数**代码分别如下：

```
void EP1_OUT_Callback(void)
{
  u8 DataLen; //保存接收数据的长度
  u8 DataBuffer[64]; //保存接收数据的缓冲区

  DataLen = GetEPRxCount(ENDP1); //获取收到的长度
  PMAToUserBufferCopy(DataBuffer, ENDP1_RXADDR, DataLen); //复制数据
  SetEPRxValid(ENDP1); //设置端点有效，以接收下一次数据

  if(DataLen==1) //收到一字节的输出报告
  {
   //D0 位表示数字键盘灯，D1 位表示大写字母锁定灯
   if(DataBuffer[0]&0x01)    //数字键盘灯亮
   {
     GPIOC->BSRR=(1<<6); //亮 LED3
   }
   else
   {
     GPIOC->BRR=(1<<6); //灭 LED3
   }
   if(DataBuffer[0]&0x02) //大写字母锁定键
   {
     GPIOC->BSRR=(1<<7); //亮 LED2
   }
```

```
    else
    {
      GPIOC->BRR=(1<<7); //灭 LED2
    }  }  }

void Joystick_Send(u8 Keys)
{
    u8 Buffer[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    u8 i;
    i="2";
          //对各个按键进行处理。注意，由于这里的摇杆 5 个按键
          //不可能同时按下，所以返回的普通键数量不会超过 6 个。
          //如果你的键盘同时按下的普通键能够超过 6 个的话，就需要做
          //点特殊处理了，将后面 6 字节全部设置为 0xFF，表示按键无法识别。
    if(Keys&KEY_UP)
    {
      Buffer[i]=0x52; //Keyboard UpArrow
      i++;
    }
    if(Keys&KEY_DOWN)
    {
      Buffer[i]=0x51; //Keyboard DownArrow
      i++;
    }
    if(Keys&KEY_LEFT)
    {
      Buffer[i]=0x50; //Keyboard LeftArrow
      i++;
    }
    if(Keys&KEY_RIGHT)
    {
      Buffer[i]=0x4F; //Keyboard RightArrow
      i++;
    }
    if(Keys&KEY_2)
    {
      Buffer[i]=0x39; //Keyboard Caps Lock
      i++;
    }
    if(Keys&KEY_3)
    {
      Buffer[i]=0x53; //Keypad Num Lock and Clear
      i++;
    }
```

```
    if(Keys&KEY_SEL)
    {
      Buffer[i]=0x28; //Keyboard Return (ENTER)
    }
            /*copy mouse position info in ENDP1 Tx Packet Memory Area*/
    UserToPMABufferCopy(Buffer, GetEPTxAddr(ENDP1), 8);
            /* enable endpoint for transmission */
    SetEPTxValid(ENDP1);
}
```

程序运行后，可在设备管理器中看到新增加的 USB 人体学输入设备和一个键盘设备。如下图：



【 初涉 USB，初学者 USB 入门总结——枚举】

【USB】

1. 不但固件程序需要编写；
2. PC 端的驱动也要编写；
3. 而且驱动写好了还要写个上机位才能看出效果；

这样调试起来十分困难，建议从 USB 的键盘，鼠标开始做，了解清楚了，再做自己的协议就比较简单了。

----------（第一）设备插入----------

1） 主机会轮回查询各个 USB 端口，主机检测到 D+与 D-之间有电压差，就认为有新的设置接入。主机等待 100ms 后发出复位请求。设备接到复位请求后将产生一个外部中断信号。

----------（第二）枚举过程（检测到设备插入以后）----------

2） 主机这时候只是知道有新的设备插入了，但是不知道插进来个什么东西，所以就开始询问它是什么设备，怎么用，负荷能力怎么样。**这个时侯就进入了枚举过程**。

步骤 1：（设备描述符）

地址：因为刚刚插入的设备没有分配地址，就用默认地址 0；

主机： 首先发送一个 Get_deor（获取**设备**描述符）指令包；

设备：设备接到包后就开始解析包（其实就是你在固件程序里判断处理），然后按固定格式返回自己设备的设备描述符

目的：这一步主要是让主机知道你的 USB 设备的基础属性，比如支持的传输数据长度，电流负荷多少，支持那个 USB 版本，以后方便电脑找驱动的 PID，VID。

步骤 2：（配置描述符）

分配地址：主机知道设备的数据长度和电流大小后，然后就是给设备分配一个属于它的地址；

主机：给设备一个地址后，就开始询问**设备**的具体配置（配置描述符）。 首先发送一个试探性的设备配置请求 Get_configuration（要求固定返回 9 个设备**配置字**）指令包；

设备：接到指令包后就开始发送 9 字节的设备配置字，其中包括设备的配置字的总长度；

主机：这样主机就知道设备的配置到底有多长，然后再发一次设备配置请求指令包；

设备：这时设备就开始上传**所有的配置字**；

主机：这个时侯主机就已经很明白你的工作方式和各种特性，然后就**可以正常工作**了；

目的：配置（以后章节详细说明）要求说明自己的名字什么的，这里还要上传**字符串描述符**；如果是鼠标或者键盘还要上传**报告描述符。**

----------（第三）正常数据阶段----------

这个时侯你已经被主机正式接受并且注册了，你可以通过自己写测试驱动或通用驱动与电脑进行通讯了。

对于 USB 的工作我这里做个比方：

主机好比一个公司，你就是 USB 设备，要进入公司首先要面试（枚举），你到了面试现场（第一次插入设备），面试官首先了解到你的外表，性别已经你要应聘的岗位（设备描述符），然后给你一个号，以后就开始按号叫人，当你被叫到就开始问你的专业知识，性格等（配置描述符），如果你比较合适（通过了枚举）你就会录取了，并且注册一个你的信息到公 司（驱动安装，并且写入注册表）。等你下次来公司，只要把工号（PID，VID）报上，就知道是你来了。

【 初涉 USB，初学者 USB 入门总结——设备固件程序】

主机：主机给设备设置地址，主机会通过固定的通道（point 0）发送一个“设定地址”包；

设备：设备主控接到包后会产生中断，并且把**响应的状态**保存在相应的寄存器中；

目的：我们只要**在中断程序中**判断各个寄存器就能完成主机的任务。

【 初涉 USB，初学者 USB 入门总结——数据包阐述】

步骤一：了解 USB 上传的什么东西，以什么格式传数据；

1） 各种总线（USB 也是一种总线）的数据传输都是以固定的层次协议进行的；

2） 所谓的层次是表达一种依附关系，**上层要依赖于底层，上层以底层为基础，上层只需要关心自己的东西就行了；**

3） 要实现两个机器（机器的范围比较广，可以是电脑，交换机，单片机）的通信总是要有一个**载体**才可以，对于机器当然是**电平高低为载体；**

具体的说：机器甲要告诉机器乙一件事情（比如说一条指令），那么机器甲可以通过一根线

（串行数据总线）连到机器乙的一个 IO 口上，甲发送一个个的高低电平，乙固定时间检测自己的这个 IO 口，然后逐个记录下放到自己的缓冲里，这样乙就收到甲送的数据了。

这里也打个比方：比如人与人进行交流，我们当然是通过说话了，物理层就是空气和传输的声波，数据链路层就是我们说的每一个字，物理层就是空气，负责把我们说的话转换成声波传给对方，数据链路层负责让对方能正确的听到每个字，如果听的不清可以告诉对方重新说一遍。经过上述的两个底层，就可以保证每一位数据可以正确的传到对方那里去。

步骤二：解析数据代表了什么，一般来说，数据都是以一串数为单位，一般称为一个包，机器间传输都是以一个**以包为单位**传出的，就像人们说话都是以一句话为单位输出一样。**每一个包包含有**许多位**数据**，这些数据又分段表示不同的意义。

如图 一，这是一个 USB 令牌阶段的包：（数据包 ➔ 数据位 ➔每段数据位的意义）

1. Sync 是同步数据（相当于说话时先打个招呼，告诉对方要跟他说话了）；
2. PID 是包标示（告诉对方这个包是干什么用的）；
3. ADDR 是对方的地址（叫对方的名字）；
4. ENDP 是用端点几通讯（----------）；
5. CRC5 是校验位（判断这个包是否在传输中出错），EOP 是包结束。

Sync	PID	ADDR	ENDP	CRC5	EOP

图一

USB **的数据包**又分为三种，一个是**令牌包**，一个是**数据包**，另一个是**握手包**。

一、每一次的 USB 通讯事务处理都是**以令牌包开头**，告诉对方要跟谁说话，这句话是干嘛的。

二、如果要求有数据传输，则下一步就是数据包，

三、另外如果要求对方要有反馈，则会发出握手包。

四、**令牌包**又简单的包括 OUT，IN，STEP 三种类型；

（1）.OUT 是用于主机告诉设备主机要向 USB 设备发送数据，

（2）.IN 是用于主机告诉设备要上传数据，

（3）.而 STEUP 是用于主机向 USB 设备发送配置信息，在枚举过程中会用到。

- 另外数据包和握手包的具体格式什么的，可以参照详细的协议。
- 可以看到在所以的通讯过程中，主机都是发起者，不管是主机发送数据到 USB 设备还是 USB 设备发送数据到主机，都必须受主机控制。

图二为一次事务的过程：

令牌阶段 ——》 数据阶段 ——》 握手阶段

图二

这个过程可以这样描述，甲和乙对话，甲是老板，乙是职员。第一节已经讲过了，乙面试就是枚举，在这个过程中，甲多段的发送 STEP 令牌包给乙，乙收到后如果要反馈数据，就发数据包给甲，甲正确接收后，跟甲握握手，表示这次对话成功。

乙被录取后，甲会分派任务（OUT），这时甲对乙说有任务给你（令牌阶段），然后乙就开始听，甲说你的任务就是记录数据并且上报（这段话就是数据包），乙说好的（握手包）。乙开始正式工作，并且记录数据。过了一段时间，甲开始要求提交数据（IN），乙把数据报告给甲（数据阶段），甲说好（握手成功）。这里乙不能主动的去向老板汇报，**只能**被动的干活。

【 初涉 USB，初学者 USB 入门总结——USB 通讯设备快速开发（固件，驱动】

经过上述三节的描述，对 USB 应该已经有了初步的认识，其中具体的协议（比如各个描述符的定义什么的）这里不做描述了。下面我以一个实例来详细说明快速开发 USB 设备的步骤。

一、设定规划

凡事预则立，不预则废，所以开发一个小小的 USB 也要稍微规划一下，比如想象要实现什么功能，传输的数据协议什么的。

二、固件编程

固件编程说白了就是写单片机程序，要实现 USB 一般可以使用带 USB 功能的单片机，再就是加一个专用的 USB 芯片。这里以内部集成 USB 功能单片机为例：

固件的 USB 开发一般就是**先使能 USB，使能 USB 时钟，使能各个 USB 控制中断（挂起，复位，标准请求，写入，写出等）然后 USB 就能正常工作了**，这时候不如不写别的东西，电脑就可以检测出有 USB 设备插入了，具体的反应是在设备管理器里会发现闪了一下说明发现了新的 USB 设备，接下来电脑会发送各种标准请求，因为这个时候你的程序还没写完整，对这些请求不会有反应，所以电脑不可能识别出是什么东西。

接下来的工作就是 **在中断中 响应**电脑传来的各种标准请求。当必要的请求都被正确的响应的话，这个时候如果电脑里有正确的驱动，电脑就会去加载这个驱动，如果是第一次插入这个设备，还要把驱动安装一下，然后设备就进入正常工作了，电脑会显示“这个 USB 已经成功安装并可以应用了”。

【端点（endpoint）的概念】

一般一个 USB 设备都会有**数个端点**，**端点**就是一个**数据缓冲控制区（FIFO）**，每一个缓冲区相当于有一个出口一个进口的池子，数据通过进口进入到池子，然后你再在固件里去用这些数据。固件往电脑写数据，也是把数据先放到池子里，然后打开出口，就可以干自己的事情，不用一个个的把数据发出了，池子的出口自动把数据流出。

一般的端口 0 是用来做标准请求响应用的，也就是在枚举阶段用到。我一般把端口 1 定义为出（OUT），端口 2 定义为入（IN）（注意，这个 OUT 和 IN 是相对与电脑的，也就是说 OUT 是数据从电脑出去到设备，IN 是设备的数据进入电脑）。**这些定义**也是在标准请求中去告诉电脑的。接下来就可以实现与电脑的通讯了，你把数据放到相应的池子里就行了。

下面就可以自己定义通讯的数据格式了。**比如**控制开发板上的 8 个 LED 的第一个灯亮，那么上位机发送数据 0x55，0x01，0x80，0xaa。我们就可以规定第一个数据是启示位，遇到这个表明开始一次控制指令，0x01 表示这个是控制灯亮暗的指令，0x80 表示 LED 的控制数据，最高位是 1，表示第一个亮，其他位是 0，表示都暗。最后一个数据是 0xaa，表示这是结束。其实所谓的**数据协议**不过就是自己定义的一套让通讯双方都能正确理解对方的数据格式。

端点（endpoint）：位于 USB 设备或主机上的一个数据缓冲区，用来存放和发送 USB 的各种数据，每一个端点都有惟一的确定地址。

上行、下行：设备到主机为上行，主机到设备为下行

A. 数据在 USB 线里传送是由低位到高位发送的。

B. USB 数据是由二进制数字串构成的，首先数字串构成域（有七种），域再构成包，包再构成事务（IN、OUT、SETUP），事务最后构成传输（中断传输、并行传输、批量传输和控制传输）。

上位机：是指人可以直接发出操控命令的**计算机**，一般是 PC，屏幕上显示各种信号变化（液压，水位，温度等）。

下位机：是直接控制设备获取设备状况的计算机，一般是 PLC/单片机之类的。

1. 上位机发出的命令首先给下位机，
2. 下位机再根据此命令解释成相应时序信号直接控制相应设备。下位机不时读取设备状态数据（一般为模拟量），转换成数字信号反馈给上位机。

➢ 简言之如此，实际情况千差万别，但万变不离其宗：**上下位机都需要编程，都有专门的开发系统。**

二、USB HID 类可采用的通信管道

所有的 HID 设备通过 USB 的控制管道（默认管道，即端点 0）和中断管道与主机通信。

表 1、USB HID 规范定义的 HID 设备可用端点

管道	要求	说明
控制(端点 0)	必须	传输 USB 描述符、类请求代码以及供查询的消息数据等
中断输入	必须	传输从设备到主机的输入数据
中断输出	可选	传输从主机到设备的输出数据

控制管道主要用于以下 3 个方面：

- 接收/响应 USB 主机的控制请示及相关的类数据
- 在 USB 主机查询时传输数据（如响应 Get_Report 请求等）
- 接收 USB 主机的数据

中断管道主要用于以下两个方面：

- USB 主机接收 USB 设备的异步传输数据
- USB 主机发送有实时性要求的数据给 USB 设备

从 USB 主机到 USB 设备的中断输出数据传输是可选的，当不支持中断输出数据传输时，USB 主机通过控制管道将数据传输给 USB 设备。

HID 设备的描述符除了 5 个 USB 的**标准**描述符（设备描述符、配置描述符、接口描述符、端点描述符、字符串描述符，还包括 3 个 HID 设备类**特定**描述符：HID 描述符、报告描述符、实体描述符。

除了 HID 的三个特定描述符组成对 HID 设备的解释外，5 个**标准描述符**中与 HID 设备有关的部分有：

1. 设备描述符中 bDeviceClass、bDeviceSubClass 和 bDeviceProtocol 三个字段的值必须为零。
2. 接口描述符中 bInterfaceClass 的值必须为 0x03;

bInterfaceSubClass 的值为 0 或 1，为 1 表示 HID 设备符是一个启动设备（Boot Device，一般对 PC 机而言才有意义，意思是 BIOS 启动时能识别并使用您的 HID 设备，且只有标准鼠标或键盘类设备才能成为 Boot Device。如果为 0 则只有在操作系统启动后才能识别并使用您的 HID 设备），bInterfaceProtocol 的取值含义如下表所示：

表 2、HID 接口描述符中 bInterfaceProtocol 的含义

bInterfaceProtocol 的取值（十进制）	含义
0	NONE

1	键盘
2	鼠标
3~255	保留

USB 的重要关键字 ：

1、端点：位于 USB 设备或主机上的一个数据缓冲区，用来存放和发送 USB 的各种数据，每一个端点都有惟一的确定地址，有不同的传输特性（如输入端、输出端点、配置端点、批量传输端点）

2、数据是如何在 USB 传输线里面传送的 ？

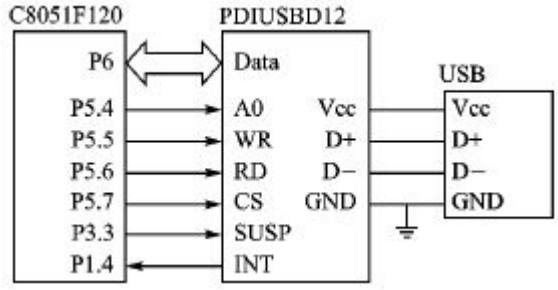
答： 数据在 USB 线里传送是由低位到高位发送的。

USB 固件库

USB 设备（DEVICE）从来只是被动触发，**USB 主机（HOST）掌握主动权**，发送什么数据，什么时候发送，是给设备数据还是从设备请求数据，都是由 **USB 主机完成**的，USB 设备只是配合主机完成设备的枚举、数据方向和大小。根据数据特性再决定该不该回复该如何回复、该不该接收该如何接收这些动作。

HID 接口固件设计与实现

该设备采用 C8051F120 微控制器和 PDIUSBD12 芯片来实现，如图 3 所示。



USB OTG：由于 USB 是主从模式，设备与设备之间、主机与主机之间不能互连，为了解决这个问题，扩大 USB 的使用范围，就出现了 USB OTG(On The Go)。

转载：http://www.usr.cc/thread-51423-1-1.html

作者：追风

一、USB 的“JoyStickMouse”例程结构分析

1、例程的结构

（1）底层结构

包括 5 个文件：usb_core.c（USB 总线数据处理的核心文件），usb_init.c，usb_int.c（用于端点数据输入输入中断处理），usb_mem.c（用于缓冲区操作），usb_regs.c（用于寄存器操作）。它们都包含了头文件“usb_lib.h”。在这个头文件中，又有以下定义：

```
#include "usb_type.h"

#include "usb_regs.h"

#include "usb_def.h"

#include "usb_core.h"

#include "usb_init.h"

#include "usb_mem.h"

#include "usb_int.h"
```

usb_lib.h 中又包含了七个头文件，其中 usb_type.h 中主要是用 typedef 为 stm32 支持的数据类型取一些新的名称。usb_def.h 中主要是定义一些相关的数据类型。

还有一个未包含在 usb_lib.h 中的头文件，usb_conf.h 用于 USB 设备的配置。

（2）上层结构

上层结构总共 5 个文件：hw_config.c（用于 USB 硬件配置）、usb_pwr.c（用于 USB 连接、断开操作）、usb_istr.c（直接处理 USB 中断）、usb_prop.c（用于上层协议处理，比如 HID 协议，大容量存储设备协议）、usb_desc.c（具体设备的相关描述符定义和处理）。

可见，ST 的 USB 操作库结构十分清晰明了，我先不准备直接阅读源代码。而是先利用 MDK 的软件模拟器仿真执行，先了解一下设备初始化的流程。

2、设备初始化所做的工作

（1）Set_System(void)

这个是 main 函数中首先调用的函数，它位于 hw_config.c 文件中。它的主要功能是初始化时钟系统、使能相关的外围设备电源。

配置了 JoyStickMouse 所用到的 5 个按键，并且配置了两个 EXTI 中断，一个是用于把 USB 从挂起模式唤醒，还有一个用途未知。

（2）USB_Interrupts_Config();

这个是 main 函数中调用的第二个函数，它也位于 hw_config.c 文件中。主要功能是配置 USB 所用到的中断。

跟踪到代码中，主要设配置了 USB 低优先级中断和唤醒中断，又有一个 EXTI 中断功能未知。

（3）Set_USBClock()

这个是 main 函数中调用的第三个函数，它也位于 hw_config.c 文件中。它的功能是配置和使能 USB 时钟。

（4）USB_Init(void)

这个是 main 函数中调用的第四个函数，它也位于 usb_init.c 文件中。它初始化了三个全局指针，指向 DEVICE_INFO、USER_STANDARD_REQUESTS 和 DEVICE_PROP 结构体。

后面两个是函数指针结构体，里面都是 USB 请求实现、功能实现的函数指针。

```
void USB_Init(void)

{


pInformation = &Device_Info;


pInformation->ControlState = 2;


pProperty = &Device_Property;


pUser_Standard_Requests = &User_Standard_Requests;
```

```
/* Initialize devices one by one */


pProperty->Init();

}
```

这三个结构体都是与具体设备枚举和功能实现相关的，定义在 usb_prop.c 和 usb_desc.c 文件中。

```
DEVICE_PROP Device_Property =


{


Joystick_init,


Joystick_Reset,


Joystick_Status_In,


Joystick_Status_Out,


Joystick_Data_Setup,


Joystick_NoData_Setup,


Joystick_Get_Interface_Setting,


Joystick_GetDeviceDescriptor,


Joystick_GetConfigDescriptor,


Joystick_GetStringDescriptor,
```

```
0,

0x40 /*MAX PACKET SIZE*/

};

USER_STANDARD_REQUESTS User_Standard_Requests =

{

Joystick_GetConfiguration,

Joystick_SetConfiguration,

Joystick_GetInterface,

Joystick_SetInterface,

Joystick_GetStatus,

Joystick_ClearFeature,

Joystick_SetEndPointFeature,

Joystick_SetDeviceFeature,

Joystick_SetDeviceAddress

};
```

Usb_init()函数调用 pProperty->Init()（实质上就是 Joystick_init）完成设备的初始化。

上层程序调用下次函数是常规性的操作。而下层函数(usb_init 相对于 usb_prop 是输入底层操作文件）调用上层文件函数我们称之为回调。

回调函数的意义在于同一种操作模式、提供不同的回调函数则可以实现不同的功能。Windows 中处理消息，好像也用到了这种模式。

回调函数的实现方法是函数指针数组。这是指针的高级应用。

这是函数的代码：

```
void Joystick_init(void)

{/* Update the serial number string descriptor with the data from the
unique
ID*/



Get_SerialNum();
//获取设备序列号，转变为 unicode 字符串


pInformation->Current_Configuration = 0;


/* Connect the device */


PowerOn();
//连接 USB 设备，实质是能让主机检测到了。


/* USB interrupts initialization */


_SetISTR(0);
/* clear pending interrupts */


wInterrupt_Mask = IMR_MSK;


_SetCNTR(wInterrupt_Mask); /* set interrupts mask */


bDeviceState = UNCONNECTED;
```

}

实质上，代码执行到这里，开发板已经可以响应主机发来的数据了。但我还是先把 main（）函数的代码看完吧。

（5）SysTick_Config();

这个函数调用主要是为程序中用到的精确延时作配置。

3、进入主循环

进入主循环的工作就两个：

Joystick_Send(JoyState())。

JoyState()用来获取按键的状态。

Joystick_Send(JoyState())用来把按键状态发到主机。当然这里真正的发送工作并不是由该代码完成的。它的工作只是将数据写入 IN 端点缓冲区，主机的 IN 令牌包来的时候，SIE 负责把它返回给主机。

主要代码如下：

```
UserToPMABufferCopy(Mouse_Buffer, GetEPTxAddr(ENDP1), 4);
//从用户复制四个字节到端点 1 缓冲区，控制端点的输入缓冲区。

SetEPTxValid(ENDP1); /* enable endpoint for transmission */
```

4、中断处理过程大致理解

（1）usb_istr()函数中的中断处理简单分析

有用的代码大概以下几段，首先是处理复位的代码，调用设备结构中的复位处理函数。

```
wIstr = _GetISTR();

if (wIstr & ISTR_RESET & wInterrupt_Mask)
```

```
{

_SetISTR((u16)CLR_RESET); //清复位中断

Device_Property.Reset();

}
```

处理唤醒的代码：

```
if (wIstr & ISTR_WKUP & wInterrupt_Mask)

{

_SetISTR((u16)CLR_WKUP);

Resume(RESUME_EXTERNAL);

}
```

处理总线挂起的代码：

```
if (wIstr & ISTR_SUSP & wInterrupt_Mask)

{

if (fSuspendEnabled) /* check if SUSPEND is possible */

{

Suspend();
```

```
}

else

{

/* if not possible then resume after xx ms */


Resume(RESUME_LATER);

}

/* clear of the ISTR bit must be done after setting of CNTR_FSUSP */

_SetISTR((u16)CLR_SUSP);

}
```

处理端点传输完成的代码，这段是最重要的，它调用底层 usb_int.c（）文件中的 CTR_LP（）函数来处理端点数据传输完成中断。

```
if (wIstr & ISTR_CTR & wInterrupt_Mask)

{

CTR_LP(); /* servicing of the endpoint correct transfer interrupt */

}
```

二、STM32 处理器的 USB 接口

1、接口模块的内部结构

在书上有一个很好的 USB 内部接口模块内部结构图，比较好的解释了各个模块之间的关系，我这里试着用我自己的理解阐述一下吧。

首先在总线端（与 D+、D-相连的那一端），通过模拟收发器与 SIE 连接。SIE 使用 48MHz 的专用时钟。

与 SIE 相关的的有三大块：CPU 内部控制、中断和端点控制寄存器，挂起定时器（这个好像是 USB 协议的要求，总线在一定时间内没有活动，SIE 模块能够进入 SUSPEND 状态以节约电能），还有包缓冲区接口模块。

说到包缓冲区接口模块，这个对应的含义是，USB 设备应该提供 USB 包缓冲区。这块缓冲区同时受到 SIE 和 CPU 核心的控制，用于 CPU 与 SIE 共享达到数据传输的目的。

所以 CPU 通过 APB1 总线接口访问，SIE 通过包缓冲区接口模块访问，中间通过 Arbiter 来协调访问。

当然我们关注的中心点是控制、中断和端点控制寄存器。我们通过这些寄存器来获取总线传输的状态，控制各个端点的状态，并可以产生中断来让 CPU 处理当前的 USB 事件。

CPU 可以通过 APB1 总线接口来访问这些寄存器。它们使用的都是 PCLK1 时钟。

2、USB 模块的寄存器认识

（1）

控制寄存器 CNTR

传输完成中断允许位。CTRM，1 有效，如果 SIE 置位传输完成标志，则相应的数据传	包缓冲区溢出中断允许位	错误中断允许位	唤醒中断允许位。WKUPM。1 有效，如果唤醒请求标志位置位，则产生唤醒中断。	挂起中断允许位。SUSPM，1 有效，当总线挂起标志置位时，发生挂起中断。	复位中断允许位。RESETM。1 有效，软件强制复位和总线复位信号，都能触发复位中断。	帧首中断允许位	期望帧首中断允许位。ESOFM。它的含义是没有收到帧首信号，允许发生中

输完成中断发生。 第 15 位							断。 第 8 位
			向主机发送的唤醒请求，RESUME。1 有效，主机收到该信号，将唤醒设备。这个由软件置位。 第 4 位	强制挂起控制，FSUSP。1 有效。与由于总线无活动引起挂起的效果相同。	低功耗模式。前提是先进入挂起状态。由软件设置，一般又硬件复位（被唤醒后自动清零）。	断电模式控制位。PDWN。此位为 1 时，USB 模块关闭。	强制复位控制。FRES。与总线上的复位信号产生相同的效果。也能产生复位中断. 第 0 位。

（2）

中断状态寄存器 ISTR

这个寄存器主要是反映 USB 模块当前的状态的。第 15-8 为与控制寄存器的中断允许是意义对应的。相应的标志位置位，且中断未屏蔽，则向 CPU 发出对应的中断。

CTR 标志，数据传输完成后硬件置 1.	PMAOVR 标志	ERR 标志	WKUP 请求，总线检测到主机唤醒请求时由硬件置位。	SUSP 请求标志位。	RESET 请求标志位。	SOF 帧首标志	ESOF，期待帧首标志。
		DIR 传输方向，此位由硬件控制。IN 时为 0，OUT 为 1. 第 4 位。		发生数据传输的端点的地址。			

（3） USB 设备地址寄存器

第 7 位，EF，USB 模块允许位。如果 EF=0，则 USB 模块将停止工作。

第 6-0 位。USB 当前使用的地址。复位时为 0.

（4）

端点状态和配置寄存器，8 个寄存器，支持 8 个双向端点和 16 个单向端点。

CTR_RX,正确接收标志位。 第 15 位。	DTOG_RX，用于检测的数据翻转位。一般由硬件自动设置，软件写 1 可使其手动翻转。	STAT_RX，占据两位。 00 表示该端点不可用，无回应。 01 表示响应 STALL 10 响应 NAK 11 表示端点有效，可接收数据。	
SETUP 标志。收到 SETUP 令牌包时置位。用户收到数据后需检查次位。 第 11 位。	EP_TYPE，两位，表示端点类型。 00 表示批量端点。 01 表示控制端点 10 表示等时端点。 11 表示中断端点。		EP_KIND，端点特殊类型。在 EP_TYPE=01 时，表示设备期望主机的 0 字节状态包。
CTR_TX。 正确发送标志。主机的 IN 包之后。 第 7 位。	DTOG_TX，用于检测的数据翻转位。一般由硬件自动设置，软件写 1 可使其手动翻转。	STAT_TX，占据两位。 00 表示该端点不可用，无回应。 01 表示响应 STALL 10 响应 NAK 11 表示端点有效，可发送数据。	
端点地址：EA【3：0】，表明该寄存器对应的端点号码。比如 1、2 号寄存器都可以对应端点 1（在双缓冲情况下）。 第 3-0 位。			

（5）

端点描述符表相关寄存器

首先有一个描述符表地址寄存器，指明了包缓冲区内端点描述符表的地址。

每一个端点都对应一个描述附表。描述符表也在包缓冲区内。每个端点寄存器对应的描述符表的地址可根据公式计算。

单缓冲、双向的端点描述符表有四项，每项占据两个字节：分别是端点 n 的发送缓冲区地址、发送字节数、接收缓冲区地址、接收字节数。

了解 USB 相关寄存器的知识以后，接下来就可以分析“JoyStickMouse”详细的工作过程了。

三、USB 的“JoyStickMouse”工作过程详细分析

1、初始化过程叙述

从 main（）函数开始

（1）Set_System(void)的工作过程

由于这些代码都是采用库代码，所以我主要分析每个代码具体做了什么工作。有些常用、类似的代码这里就不列出来了。

先将 RCC 部分复位，系统使用内部振荡 HSI，8MHz——RCC_DeInit();。

使能 HSE——RCC_HSEConfig(RCC_HSE_ON);

设置 HCLK = SYSCLK——RCC_HCLKConfig(RCC_SYSCLK_Div1);

设置 PCLK2，PCLK1——RCC_PCLK2Config(RCC_HCLK_Div1);

设置 PLL，使能 PLL——PLL 采用 HSE，输出=HSE X 9；

RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);

系统时钟采用 PLL 输出——

RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

使能 PWR 控制，目的是为了控制 CPU 的低功耗模式；

将所有输入口初始化为模拟输入——GPIO_AINConfig();

使能 USB 上拉控制 GPIO 端口的时钟，这个端口设置为低电平时，USB 外设会被集线器检测到，并报告给主机，这也是设备枚举的开始；

将这个端口的模式设置为开漏输出；

初始化上下左右四个按键为上下拉输入；

配置 GPIOG8 为 EXTI8 中断输入引脚，这个是在外部按键输入引起中断。

配置 EXTI18 中断。这个是发生 USB 唤醒事件时用。

```
EXTI_InitStructure.EXTI_Line = EXTI_Line18; // USB resume from suspend
mode

EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;

EXTI_InitStructure.EXTI_LineCmd = ENABLE;

EXTI_Init(&EXTI_InitStructure);
```

（2）USB_Interrupts_Config(void)的工作过程

设置向量表位置在 FLASH 起始位置——

```
NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x00);
```

设置优先级分组，1 位用于抢占组级别。其余用于子优先级——

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
```

接下来配置、使能了三个中断，包括 USB 低优先级中断、USB 唤醒中断（EXTI18）、和 EXTI8（按键控制）中断。

它的优先级设置有些问题，明明只有一位用于抢占优先级。它把 EXTI8 的抢占优先级设为 2。结果在调试时发现，它的抢占优先级仍然是 0。

（3）Set_USBClock()的工作过程

这个代码就两句话：

```
RCC_USBCLKConfig(RCC_USBCLKSource_PLLCLK_1Div5);

RCC_APB1PeriphClockCmd(RCC_APB1Periph_USB, ENABLE);
```

作用是设置并使能 USB 时钟，从 RCC 输出可以看到，USB 时钟是 48MHz。

（4）USB_Init()的工作过程

```
void USB_Init(void)
{

pInformation = &Device_Info;

pInformation->ControlState = 2;

pProperty = &Device_Property;
//这个是设备本身支持的属性和方法

pUser_Standard_Requests = &User_Standard_Requests; //这个是主机请求的实现方法。

pProperty->Init();
//回调设备的初始化例程。
}
```

这个主要是初始化了三个全局结构体指针，pInformation 表明当前连接的状态和信息，pProperty 表明设备支持的方法，pUser_Standard_Requests 是主机请求实现的函数指针数组。

Device_Info 是一个结构体，包括 11 个成员变量。这里是将它的 ControlState 设为 2，意义现在还不十分明了。

```
typedef struct _DEVICE_INFO

{


u8 USBbmRequestType;
/* bmRequestType */


u8 USBbRequest;
/* bRequest */


u16_u8 USBwValues;
/* wValue */


u16_u8 USBwIndexs;
/* wIndex */


u16_u8 USBwLengths;
/* wLength */


u8 ControlState;
/* of type CONTROL_STATE */


u8 Current_Feature;


u8 Current_Configuration;
/* Selected configuration */


u8 Current_Interface;
/* Selected interface of current configuration */


u8 Current_AlternateSetting;/* Selected Alternate Setting of current
```

interface*/

ENDPOINT_INFO Ctrl_Info;
//端点信息结构体

}DEVICE_INFO;

最后调用 pProperty->Init()，实质就是调用 Joystick_init(void)。

在这个函数中，首先获取设备版本，并转换为 Unicode 存入版本号字符串。

——Get_SerialNum();

设备当前配置置为 0。然后调用 PowerOn()，这个函数实质上将 D+上拉，此时 USB 设备就能被集线器检测到了。因此分析进入下一个流程。

2、进入设备检测状态

（1）在 PowerOn()中执行的情况。

在 USB_init（）中调用 PowerOn()，而它先调用 USB_Cable_Config(ENABLE)，这个函数实质上将 USB 连接控制线设置为低电平，然后设备就可以检测到设备了。

当集线器报告设备连接状态，并收到主机指令后，会复位 USB 总线，这需要一定的时间（这段时间内设备应该准备好处理复位指令）。但是现在设备初始化程序将继续往下进行，因为它还没有使能复位中断。

wRegVal = CNTR_FRES;

_SetCNTR(wRegVal);
//这句话实际上使能了 USB 模块的电源，因为上电复位时，CNTR 寄存器的断电控制为 PDWN 位是 1，模块是断电的。

这句话虽然将强制复位 USB 模块，但由于复位中断允许位没有使能，不会引起复位中断，而间接上由使 PDWN=0，模块开始工作。

_SetCNTR 是一个宏，将 wRegVal 赋值给 CNTR 寄存器，此时所有的中断被屏蔽。

再接下来两句指令又将清除复位信号。

然后清除所有的状态位。——_SetISTR(0);

接下来是很关键的两句话：

```
wInterrupt_Mask=CNTR_RESETM| CNTR_SUSPM | CNTR_WKUPM;

_SetCNTR(wInterrupt_Mask);
```

后面一个语句执行后，复位中断已经被允许，而此时集线器多半已经开始复位端口了。或者说稍微有限延迟，设备固件还能继续初始化一些部件，但已经不会影响整个工作流程了。

所以接下来，分析直接进入复位中断。

（2）复位中断的处理。

当复位中断允许、且总线被集线器复位的时候，固件程序进入 USB_LP 中断。

中断程序直接调用 USB_Istr(void)程序。

接下来讲对中断位进行判断：

```
if (wIstr & ISTR_RESET & wInterrupt_Mask)

{

_SetISTR((u16)CLR_RESET);
//先清除复位中断位


Device_Property.Reset();
//进入设备定义的复位过程。实际上是调用 JoyStick_Reset（）函数进行处理。

}
```

（3）JoyStick_Reset（）函数的处理。

这里将一句句来分析：

```
void Joystick_Reset(void)

{


pInformation->Current_Configuration = 0;
//当前配置为 0


pInformation->Current_Interface = 0;/当前接口为 0


pInformation->Current_Feature = Joystick_ConfigDescriptor[7];



//需要总线供电


SetBTABLE(BTABLE_ADDRESS); //设置包缓冲区地址。


SetEPType(ENDP0, EP_CONTROL);
//端点 0 为控制端点


SetEPTxStatus(ENDP0, EP_TX_STALL);
//端点状态为发送无效，也就是主机 IN 令牌包来的时候， 回送一个 STALL。



SetEPRxAddr(ENDP0, ENDP0_RXADDR); //设置端点 0 描述符表，包括接收缓冲区地址、最大允许接收的字节数、发送缓冲区地址三个量。


SetEPTxAddr(ENDP0, ENDP0_TXADDR); //这是发送缓冲区地址


Clear_Status_Out(ENDP0);
//清除 EP_KIND 的 STATUS_OUT 位，如果改位被设置，在控制模式下只对 0 字节数据包相应。其它的都返回 STALL。主要用于控制传输的状态过程。
```

```
SetEPRxCount(ENDP0, Device_Property.MaxPacketSize); //接收缓冲区支持
64 个字节。


SetEPRxValid(ENDP0);
//使能端点 0 的接收，因为很快就要接收 SETUP 令牌包后面跟着的数据包了。


SetEPType(ENDP1, EP_INTERRUPT);
//端点 1 为中断端点。


SetEPTxAddr(ENDP1, ENDP1_TXADDR); //设置发送缓冲区地址。


SetEPTxCount(ENDP1, 4);
//每次发送四个字节


SetEPRxStatus(ENDP1, EP_RX_DIS);
//接收禁止，只发送 Mouse 信息，而不从主机接收。


SetEPTxStatus(ENDP1, EP_TX_NAK); //现在发送端点还不允许发送数据。


bDeviceState = ATTACHED;
//连接状态改为已经连接，默认地址状态。


SetDeviceAddress(0); //地址默认为 0.

}
```

复位中断执行完成后，开发板的 USB 接口能够以默认地址对主机来的数据包进行响应了。这个阶段的分析到此结束，下一个阶段就是正式分析代码实现的枚举过程了。

四、USB 的“JoyStickMouse”工作过程详细分析

1、枚举第一步：获取设备的描述符

从 USB_init（）开始

（1）先要允许数据传输完成中断

在 poweron（）函数后面紧跟着几句话：

```
PowerOn();
//这句执行完，设备被主机检测到，并且能够响应复位中断了。

_SetISTR(0);
/* clear pending interrupts */

wInterrupt_Mask = IMR_MSK;

_SetCNTR(wInterrupt_Mask); /* set interrupts mask */

//以上这两句话将允许所有的 USB 中断

bDeviceState = UNCONNECTED;
//设备状态置位为未连接状态。这里我不太理解。这时候即使复位中断未发生，
最起码设备已经算是连接入总线了，为什么这个状态还要设置为
“未连接”呢？
```

（2）主机获取描述符

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。

```
_SetISTR((u16)CLR_CTR); /*首先清除传输完成标志 */

EPindex = (u8)(wIstr & ISTR_EP_ID); //获取数据传输针对的端点号。

if (EPindex == 0)
//如果是端点 0，这里的确是端点 0


{


SaveRState = _GetEPRxStatus(ENDP0); //保存端点 0 状态，原本是有效状态。


SaveTState = _GetEPTxStatus(ENDP0);


_SetEPRxStatus(ENDP0, EP_RX_NAK); //在本次数据处理好之前，对主机发来的数据包以 NAK 回应


_SetEPTxStatus(ENDP0, EP_TX_NAK);


if ((wIstr & ISTR_DIR) == 0) //如果是 IN 令牌，数据被取走


{


_ClearEP_CTR_TX(ENDP0);


In0_Process();
//调用该程序处理固件数据输出后的工作。


_SetEPRxStatus(ENDP0, SaveRState);


_SetEPTxStatus(ENDP0, SaveTState);
```

```
return;

}

Else
//DIR=1 时，要么是 SETUP 包，要么是 OUT 包。

{
//这里先分析 SETUP 包。

wEPVal = _GetENDPOINT(ENDP0);
//获取整个端点 0 状态

if ((wEPVal & EP_CTR_TX) != 0)
//这种情况一般不太可能，

{
//如果出现表示同时 TX 和 RX 同时置位。

}

else if ((wEPVal &EP_SETUP) != 0)
//我们的程序会执行到这里

{

_ClearEP_CTR_RX(ENDP0);

Setup0_Process();

//主要是调用该程序来处理主机请求。
```

```
_SetEPRxStatus(ENDP0, SaveRState);

_SetEPTxStatus(ENDP0, SaveTState);


return;

}

else if ((wEPVal & EP_CTR_RX) != 0) //暂时不执行的代码先删除掉。

{

}

}

}/* if(EPindex == 0) */
```

后面处理其他端点的代码就先不看了。

```
}/* while(...) */
```

（3）Setup0_Process()函数的执行分析

这个函数执行的时候，主机发来的请求数据包已经存在于 RxADDR 缓冲区了。大部分的标志位已经清除，除了 SETUP 位，这个味将由下一个令牌包自动清除。

进入处理函数：

```
pBuf.b = PMAAddr + (u8 *)(_GetEPRxAddr(ENDP0) * 2); //这是取得端点 0 接收缓冲区的起始地址。
```

PMAAddr 是包缓冲区起始地址，_GetEPRxAddr(ENDP0)获得端点 0 描述符表里的接收缓冲区地址，为什么要乘以 2 呢？大概因为描述符表里地址项为 16 位，使用的是相对偏移。

```
if (pInformation->ControlState != PAUSE)

{

pInformation->USBbmRequestType = *pBuf.b++; //请求类型，表明方向和接收对象（设备、接口还是端点）此时为 80，表明设备到主机

pInformation->USBbRequest = *pBuf.b++; /* 请求代码，第一次时应该是 6，表明主机要获取设备描述符。 */

pBuf.w++;

pInformation->USBwValue = ByteSwap(*pBuf.w++); /* wValue */

pBuf.w++;
//我觉得这里可能有些问题。

pInformation->USBwIndex
= ByteSwap(*pBuf.w++); /* wIndex */

pBuf.w++;

pInformation->USBwLength = *pBuf.w; /* wLength */

}

pInformation->ControlState = SETTING_UP;

if (pInformation->USBwLength == 0)
```

```
{

NoData_Setup0();


}

else

{

Data_Setup0();
//这次是有数据传输的，所以有进入该该函数。

}

return Post0_Process();
```

（4）Data_Setup0()函数的执行分析

```
CopyRoutine = NULL; //这是一个函数指针，由用户提供。

wOffset = 0;

if (Request_No == GET_DESCRIPTOR) //如果是获取设备描述符

{

if(Type_Recipient==(STANDARD_REQUEST| EVICE_RECIPIENT))

{
```

```
u8 wValue1 = pInformation->USBwValue1;

if (wValue1 == DEVICE_DESCRIPTOR)

{

CopyRoutine = pProperty->GetDeviceDescriptor;

} //获取设备描述符的操作由用户提供。

if (CopyRoutine)

{

pInformation->Ctrl_Info.Usb_wOffset = wOffset;

pInformation->Ctrl_Info.CopyData = CopyRoutine;

(*CopyRoutine)(0); //这个函数这里调用的目的只是设置了 pInformation 中
需要写入的描述符的长度。

Result = USB_SUCCESS;

}

if (ValBit(pInformation->USBbmRequestType, 7))
//此时为 80
```

```
{
//上面这个语句主要是判断传输方向。如果为 1，则是设备到主机

vu32 wLength = pInformation->USBwLength; 这个一般是 64

if (pInformation->Ctrl_Info.Usb_wLength > wLength)

{
//设备描述符长度 18

pInformation->Ctrl_Info.Usb_wLength = wLength;

}
//有些细节暂时先放着

pInformation->Ctrl_Info.PacketSize = pProperty->MaxPacketSize;


DataStageIn();
//最主要是调用这个函数完成描述符的输出准备

}
```

（5）DataStageIn()函数的执行分析

以下是主要执行代码：

```
DataBuffer = (*pEPinfo->CopyData)(Length); //这个是取得用户描述符缓冲区的地址。这里共 18 个字节

UserToPMABufferCopy(DataBuffer, GetEPTxAddr(ENDP0), Length);//这个函数将设备描述符复制到用户的发送缓冲区。
```

```
SetEPTxCount(ENDP0, Length);
//设置发送字节的数目、18

pEPinfo->Usb_wLength -= Length; 等于 0

pEPinfo->Usb_wOffset += Length; 偏移到 18

vSetEPTxStatus(EP_TX_VALID); //使能端点发送，只要主机的 IN 令牌包一来，
SIE 就会将描述符返回给主机。

USB_StatusOut();/* 这个实际上是使接收也有效，主机可取消 IN。 */

Expect_Status_Out:

pInformation->ControlState = ControlState;
```

（6）执行流程返回到 CTR_LP(void)

```
_SetEPRxStatus(ENDP0, SaveRState);

_SetEPTxStatus(ENDP0, SaveTState);

//由于 vSetEPTxStatus(EP_TX_VALID)实际改变了 SaveTState，所以此时端点
发送已经使能。

return;
```

（7）主机的 IN 令牌包

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。

主要是调用 In0_Process()完成剩下的工作。

（7）追踪进入函数 In0_Process()

此时实际上设备返回描述符已经成功了。

这一次还是调用 DataStageIn()函数，但是目的只是期待主机的 0 状态字节输出了。

```
if ((ControlState == IN_DATA) || (ControlState == LAST_IN_DATA))

{
```

第一次取设备描述符只取一次。

```
DataStageIn();
```

//此次调用后，当前状态变成 WAIT_STATUS_OUT，表明设备等待状态过程，主机输出 0 字节。

```
/* ControlState may be changed outside the function */

ControlState = pInformation->ControlState;
```

}返回时调用 Post0_Process(void)函数,这个函数没做什么事。

（8）进入状态过程

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（）。

由于此时状态为 WAIT_STATUS_OUT，所以执行以下这段。

```
else if (ControlState == WAIT_STATUS_OUT)
```

```
{

(*pProperty->Process_Status_OUT)();
//这是个空函数，什么也不做。


ControlState = STALLED;
//状态转为 STALLED。


}
```

获取设备描述符后，主机再一次复位设备。设备又进入初始状态。

五、USB 的“JoyStickMouse”工作过程详细分析

1、枚举第二步：设置地址

（1）重新从复位状态开始

在第一次获取设备描述符后，程序使端点 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，设备端 EP0 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 所指向的缓冲区。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（），所做的工作仍然是先填充 pInformation 结构，获取请求特征码、请求代码和数据长度。

由于设置地址不会携带数据，所以接下来调用 NoData_Setup0()。执行以下代码：

```
else if (RequestNo == SET_ADDRESS)

{

Result = USB_SUCCESS;

}
```

说明设置地址没有做任何工作。

```
ControlState = WAIT_STATUS_IN;/* After no data stage SETUP */

USB_StatusIn(); //这句话是一个关键，它是一个宏，实际是准备好发送 0 字节的状态数据包。因为地址设置没有数据过程，建立阶段后直接进入状态阶段，主机发 IN 令牌包，设备返回 0 字节数据包，主机再 ACK。
```

它对应的宏是这样的：

```
#define USB_StatusIn() SendOLengthData() //准备发送 0 字节数据

#define SendOLengthData() { _SetEPTxCount(ENDP0, 0); \

vSetEPTxStatus(EP_TX_VALID); \ //设置发送有效，发送字节数为 0

}
```

（3）设置地址的状态阶段：

而前面把状态设置为 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，进入中断。

本次中断由 IN0_Process（）函数来处理，追踪进入，它执行以下代码：

```
else if (ControlState == WAIT_STATUS_IN)
```

```
{

if ((pInformation->USBbRequest == SET_ADDRESS) &&
(Type_Recipient==(STANDARD_REQUEST|DEVICE_RECIPIENT)))

{

SetDeviceAddress(pInformation->USBwValue0);

pUser_Standard_Requests->User_SetDeviceAddress(); //这个函数就一个赋
值语句，bDeviceState = ADDRESSED。

}

(*pProperty->Process_Status_IN)(); //这是一个空函数。

ControlState = STALLED;

}
```

执行设置地址操作、采用新地址后，把设备的状态改为 STALLED。而在处理的出口中调用 Post0_Process()函数，这个所做的工作是：

```
SetEPRxCount(ENDP0, Device_Property.MaxPacketSize);
//将端点 0 的缓冲区大小设置为 64 字节

if (pInformation->ControlState == STALLED)

{

vSetEPRxStatus(EP_RX_STALL);
```

```
vSetEPTxStatus(EP_TX_STALL);

}
```

将端点 0 的发送和接收都设置为：STALL，这种状态下只接受 SETUP 令牌包。

2、枚举第三步：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（）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。由调用 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()函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

本次操作会产生数据发送完成中断，由 In0_Process(void)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_OUT;


vSetEPTxStatus(EP_TX_STALL);
//转入状态阶段。
```

有可能的话：

```
SendOLengthData();


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(void)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。

3、对配置描述符、字符串描述符获取过程进行简单跟踪，过程就不再一一叙述了。

4、主机设置配置。

建立阶段：主机发 SETUP 包、发请求数据包（DATA0 包）、用户 ACK。

进入 CTR 中断，用户调用 Setup0_Process()函数进行处理，取得请求数据后，由于没有数据传输阶段，该函数调用 NoData_Setup0()函数进行处理。

判断为设置配置后，调用 Standard_SetInterface()函数将设备状态结构体的当前配置改为主机数据中的配置参数。同时调用用户的设置配置函数，将设备状态改为“configured”。

退出时，将控制传输状态改为：ControlState = WAIT_STATUS_IN，进入状态阶段。设备期待主机的 IN 令牌包，返回状态数据。

状态阶段：主机发 IN 令牌、设备返回 0[size=12p]Setup0_Process()函数进行处理，取得请求数据后，由于没有数据传输阶段，该函数调用 NoData_Setup0()函数进行处理。

设置空闲时一个类特殊请求，其特征码为 0x21，2 表示类请求而不是标准请求，1 表示接收对象是接口而不是设备。

USB 的底层并不支持类特殊请求，它将调用上层函数提供的函数：

```
if (Result != USB_SUCCESS)

{

Result = (*pProperty->Class_NoData_Setup)(RequestNo); //这里就是调用用户提供的类特殊请求的处理函数。结果发现用户提供的类特殊请求（针对无数据情况）只支持 SET_PROTOCOL。针对有数据情况只支持：GET_PROTOCOL。

if ((Type_Recipient==(CLASS_REQUEST | INTERFACE_RECIPIENT))

&& (RequestNo == SET_PROTOCOL))

{

return Joystick_SetProtocol();

}

}
```

6、主机获取报告描述符

建立阶段：主机发 SETUP 包、发请求数据包（DATA0 包）、用户 ACK。

进入 CTR 中断，获取描述符是一个标准请求，但是报告描述符并不是需要通用实现的，所以在底层函数中没有实现。跟踪 Setup0_Process(void)——进入 Data_Setup(void)函数，它是这么处理的：

```
if (Request_No == GET_DESCRIPTOR)

{

if(Type_Recipient==(STANDARD_REQUEST| EVICE_RECIPIENT))

{

u8 wValue1 = pInformation->USBwValue1;

if (wValue1 == DEVICE_DESCRIPTOR)

{

CopyRoutine = pProperty->GetDeviceDescriptor;

}

else if (wValue1 == CONFIG_DESCRIPTOR)

{

CopyRoutine = pProperty->GetConfigDescriptor;

}

else if (wValue1 == STRING_DESCRIPTOR)
```

```
{

CopyRoutine = pProperty->GetStringDescriptor;

}
/* End of GET_DESCRIPTOR */

}

}
```

可见核心函数只支持设备描述符、配置描述符以及字符串描述符。最终该函数将调用：

```
Result= (*pProperty->Class_Data_Setup)(pInformation->USBbRequest);
```

调用用户的类特殊实现来获取报告描述符，同时 HID 类描述符也是通过这种方式取得的。

7、主机从中断端点读取鼠标操作数据

主机会轮询设备，设备数据的准备在主函数中，用 Joystick_Send(JoyState()) 函数来实现。

```
Mouse_Buffer[1] = X;

Mouse_Buffer[2] = Y;

/*copy mouse position info in ENDP1 Tx Packet Memory Area*/

UserToPMABufferCopy(Mouse_Buffer, GetEPTxAddr(ENDP1), 4);

/* enable endpoint for transmission */
```

```
SetEPTxValid(ENDP1);
```

使能端点 1 的发送，当主机的 IN 令牌包来的时候，SIE 将数据返回给主机。同时产生 CTR 中断。

在中断处理程序中，执行下列代码：

```
if ((wEPVal & EP_CTR_TX) != 0)

{

/* clear int flag */

_ClearEP_CTR_TX(EPindex);

(*pEpInt_IN[EPindex-1])();

} /* if((wEPVal & EP_CTR_TX) != 0) */
```

这是在函数指针数组中调用函数，跟踪进入：发现这个函数什么也没有做。

经过对程序执行过程的跟踪和分析，我现在对 USB 设备 HID 类的工作有了大概的了解，对 ST 的 USB 库的工作也有了初步的概念。把所有文件的源代码粗略地浏览了一遍，心里大概有了些底。但现在我还不准备阅读源代码，我先把例程在智林开发板上移植好，再详细的阅读一遍源代码。