# USB的“JoyStickMouse”源代码分析 01

## 一、c源文件和头文件

### 1、c文件的组织

我们一般用 C 语言编程的时候都比较讲究模块化、层次化，以及数据和操作分开的原则。

模块化最明显的表现是我们把常用的、某一个具体功能实现封装在一个函数中，我们所要操作的数据以参数的方式提供给函数，函数通过代码处理过后，再把结果回馈个调用者。

层次化表现在比较复杂的功能实现都要分几层来实现，有时候也是为了提高可移植性。比如文件系统的代码实现大致要分这么几层：

磁盘操作层实现磁盘扇区的读取、写入、控制；

文件分配管理函数和目录项操作函数；

文件的打开、读取、写入函数，目录的建立、删除函数等；

用户应用操作函数，如文件的查找、具体数据的写入等等。

一个具体 USB 功能的实现也能分成几层，比如我们的 Stm32 JoyStickMouse，我们也可以这样分层：

硬件操作层：寄存器操作和内存操作。

协议通用层：设备枚举的控制传输实现，对所有类协议都一样。

具体协议层：比如 HID 协议的描述符，类特定请求实现。

这样一划分，一个具体的功能实现就需要比较多的 C 源文件。

## 2、c文件和h文件

一个 c 文件一般是几个函数组成，这几个函数可能一部分存在依存关系，而有一些函数要对外引出，而同时它还可能要引用外部函数。解决这个相互关联的是“包含头文件”。

一个 c 文件很多时候都存在一个与它同名的“h 文件”，对 c 文件的意义主要有以下几个：

（1）提供常量定义，#define。

（2）提供一些类型定义，为了实现 c 文件数据类型的编译器无关性，一般都要将该编译器提供的数据类型重新定义，用 tpyedef 来实现。

（3）结构体、联合体、枚举类型的定义。

（4）带参数的宏定义。

（5）外部函数声明。

（6）外部变量声明。

（7）本身实现函数的声明：起始这个声明对 c 文件本身意义不大，主要是其它 c 文件要使用本 c 文件实现的函数时，包含同名“h 文件”，意义更清楚一些。

## 3、h文件的相互包含

头文件对 c 源文件的意义比较清晰。但实际有时候“头文件”也会存在依存关系，相互包含。

比如有个头文件使用数据类型“u8”，那么它必须先包含定义这个数据类型的头文件。

有时候为了让 c 文件包含的头文件数目不至于过多，一般采用这样的方法：用一个 h 文件包含所有相关的头文件，然后每个 c 文件包含这个头文件就行了。这个以 usb 的库函数组织最为典型。

编程时，用到库函数的源文件只要包含“stm32f10x_lib.h”就行了，USB 操作库函数也采用了类似的方式。

# 二、USB函数库分析

## 1、usb_regs.h

（1）寄存器定义

```
#define CNTR ((volatile unsigned *)(RegBase + 0x40))
```

我觉得这是 c 语言指针的魅力所在，通过这样的定义可以直接操作寄存器。*CNTR = Value。特别是含有多个寄存器的设备，可以将首地址结构体指针化，通过指针间接访问结构体，实际上就访问了各个寄存器。

（2）标志位、屏蔽位定义

```
#define ISTR_CTR (0x8000) /* Correct TRansfer (clear-only bit) */

#define CLR_CTR (~ISTR_CTR) /* clear Correct TRansfer bit */
```

有了这个定义，我们可以以比较明了的方式对相关位进行操作。

（3）寄存器操作宏定义

```
#define _SetCNTR(wRegValue) (*CNTR = (u16)wRegValue)
```

这样定义以后，用户对寄存器的操作调用宏就行了，意义清晰。

```
#define _SetEPTxStatus(bEpNum,wState) {\

register u16 _wRegVal; \

_wRegVal = _GetENDPOINT(bEpNum) & EPTX_DTOGMASK;\

/* toggle first bit ? */ \

if((EPTX_DTOG1 & wState)!= 0) \

_wRegVal ^= EPTX_DTOG1; \
```

```
/* toggle second bit ? */ \
if((EPTX_DTOG2 & wState)!= 0) \
_wRegVal ^= EPTX_DTOG2; \
_SetENDPOINT(bEpNum, _wRegVal); \
} /* _SetEPTxStatus */
```

宏可以代表一个变量，也可以是一个表达式，也可以是一个代码块。

（4）声明外部变量

```
extern volatile u16 wIstr; /* ISTR register last read value */
```

（5）对外声明同名 c 文件实现的函数

```
u16 GetCNTR(void);
```

它在这里使用了“u16”类型定义，所以必须先定义。

## 2、usb_regs.c

主要是调用宏，实现寄存器操作，但我看上层函数也很少调用这些函数，直接就使用宏了。

## 3、usb_mem.h和usb_mem.c

（1）usb_mem.h

这个头文件很简单，只声明了两个函数，供其它源文件使用。

void UserToPMABufferCopy(u8 *pbUsrBuf, u16 wPMABufAddr, u16 wNBytes); 用户数据复制到端点对应的 USB 包缓冲区。

```
void PMAToUserBufferCopy(u8 *pbUsrBuf, u16 wPMABufAddr, u16 wNBytes);
```

（2）usb_mem.c

上述两个函数的实现。我这里将详细分析一个函数的代码。

```
void UserToPMABufferCopy(
u8 *pbUsrBuf, //用户提供的数据缓冲区指针。
u16 wPMABufAddr, //端点地址，这是包缓冲区内相对地址
u16 wNBytes) //本次复制的字节数
{
u32 n = (wNBytes + 1) >> 1; /*用户提供的是字节个数，而包缓冲区每次处理一个字、两个字节，这里是进行转换，+1 的目的是防止用户给出奇数。比如用户参数为 9，则实际需要 5 个字。 */
u32 i, temp1, temp2;
u16 *pdwVal;
pdwVal = (u16 *)(wPMABufAddr * 2 + PMAAddr); // wPMABufAddr 实际是端点描述符表寄存器里对应的相对地址，两个字节对齐。而 stm32 内部实际是 4 个字节对齐。举个例子：端点 0 描述附表，在包缓冲区相对偏移 0 起始，占据 8 字节，分别为“0018,0008，0058,0000”。第 0-1 字节存储发送缓冲区的相对地址，第 4-5 字节存储接收缓冲区的相对地址。
```

比如现在要将接收缓冲区的内容复制到用户缓冲区，首先要取得这个“0058”数值，获取的方法是 缓冲区相对偏移 RxADDR =“0058” =*（u16*）（包缓冲区起始地址 + 4*2）。这里 4*2 的意思是“0058”的相对偏移实际是第 8 个字节，而不是相对的偏移 4 字节。

然后获取接收缓冲区实际地址。 缓冲区实际地址 = 缓冲区起始地址 + “0058”*2。这里乘以 2 的意思跟上述是一样的。

```
for (i = n; i != 0; i--)
{
temp1 = (u16) * pbUsrBuf; //取得低字节
pbUsrBuf++; //指向高字节
temp2 = temp1 | (u16) * pbUsrBuf << 8; //高低组合成一个字
```

*pdwVal++ = temp2; //写入包缓冲区的发送缓冲区。

pdwVal++; //四字节对齐，跳过两个字节

pbUsrBuf++; //指向下一个字节。

}

}

# 4、usb_init.c和usb_init.h

（1）usb_init.h

主要是声明了函数 ust_init（）。

声明了一些外部变量。

extern USER_STANDARD_REQUESTS *pUser_Standard_Requests;

//有几个很重要的指针，在这个文件里作外部声明。

extern u16 SaveState ;

extern u16 wInterrupt_Mask;

（3）usb_init.c

DEVICE_INFO Device_Info;//这个最重要的设备信息机构体在这里定义。

函数实现：

void USB_Init(void)

{

pInformation = &Device_Info; // pInformation 是在 usb 控制传输处理核心 usb_core.c 中最重要的指针。

pInformation->ControlState = 2; //2 实际上代表 IN_DATA 状态，实际我觉得设置为 0 最好，代表 WAIT_SETUP 状态。

pProperty = &Device_Property; //这个结构体由用户提供，是一个函数指针结构体，大部分成员都是上层协议需要实现的操作函数。

pUser_Standard_Requests = &User_Standard_Requests; //这个结构体跟上一个类似，主要是用户对标准请求的实现。

/* Initialize devices one by one */

pProperty->Init(); //调用用户提供的初始化函数。

}

## 5、usb_int.c和usb_int.h

（1）usb_int.h

主要是声明了函数 void CTR_LP(void)，这是数据传输完成处理的核心处理部分。

（2）usb_int.c

这个文件主要是实现了函数 void CTR_LP(void)。

开头：

```
u16 SaveRState;

u16 SaveTState; //这是中断处理时用于保存当时端点的状态

extern void (*pEpInt_IN[7])(void); /* Handles IN interrupts */

extern void (*pEpInt_OUT[7])(void); /* Handles OUT interrupts */
```

这两个函数指针数组的定义在用户层 usb_prop.c 文件中。

```
void CTR_LP(void)

{

u32 wEPVal = 0;

while (((wIstr = _GetISTR()) & ISTR_CTR) != 0)

{
```

```
_SetISTR((u16)CLR_CTR); /* clear CTR flag */
```

EPindex = (u8)(wIstr & ISTR_EP_ID); //获取发生中断的端点号，分成 0 和非 0 端点两种情况。

```
if (EPindex == 0)
{
SaveRState = _GetEPRxStatus(ENDP0);
SaveTState = _GetEPTxStatus(ENDP0);
_SetEPRxStatus(ENDP0, EP_RX_NAK);
_SetEPTxStatus(ENDP0, EP_TX_NAK);

if ((wIstr & ISTR_DIR) == 0)
{
_ClearEP_CTR_TX(ENDP0);
```

In0_Process(); //取得方向标志，如果为 0 表示主机要“IN”数据。调用 In0_Process()来处理。

```
_SetEPRxStatus(ENDP0, SaveRState);
_SetEPTxStatus(ENDP0, SaveTState);
return;
}
```

Else //DIR==1，有两种情况，可能是主要要“OUT”，也可能是在“SETUP”。

```
{
wEPVal = _GetENDPOINT(ENDP0);
if ((wEPVal &EP_SETUP) != 0) //SETUP 的情况
{
```

```
_ClearEP_CTR_RX(ENDP0);
```

Setup0_Process(); //调用这个函数处理控制传输过程，这个函数的理解是理解 USB 工作最关键的。大部分 usb_core.c 文件里面的函数都是为它服务的。

```
_SetEPRxStatus(ENDP0, SaveRState);

_SetEPTxStatus(ENDP0, SaveTState);

return;

}



else if ((wEPVal & EP_CTR_RX) != 0)//OUT 的情况

{

_ClearEP_CTR_RX(ENDP0);
```

Out0_Process(); //调用这个函数处理用户输出。

```
_SetEPRxStatus(ENDP0, SaveRState);

_SetEPTxStatus(ENDP0, SaveTState);

return;

}

}

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

Else //这是非 0 端点的处理。在 JoyStickMouse 例程中几乎没什么作用。就是一个空架子。

```
{

wEPVal = _GetENDPOINT(EPindex);

if ((wEPVal & EP_CTR_RX) != 0)

{
```

```
_ClearEP_CTR_RX(EPindex);
(*pEpInt_OUT[EPindex-1])();

} /* if((wEPVal & EP_CTR_RX) */

if ((wEPVal & EP_CTR_TX) != 0)
{
_ClearEP_CTR_TX(EPindex);
(*pEpInt_IN[EPindex-1])();
} /* if((wEPVal & EP_CTR_TX) != 0) */
}/* if(EPindex == 0) else */
}/* while(...) */
}
```

# 6、usb_def.h

主要是定义了与 USB 标志请求相关的一些常量。

```
typedef enum _RECIPIENT_TYPE
{
DEVICE_RECIPIENT, /* Recipient device */
INTERFACE_RECIPIENT, /* Recipient interface */
ENDPOINT_RECIPIENT, /* Recipient endpoint */
OTHER_RECIPIENT
```

} RECIPIENT_TYPE;//请求发往的对象，可以是设备、接口、端点以及其它（这个需要用户特别实现）。

```
typedef enum _STANDARD_REQUESTS
{
GET_STATUS = 0, //这个可以发给设备、接口和端点。
CLEAR_FEATURE, //这个可以发给设备、接口和端点
RESERVED1,
SET_FEATURE, //这个可以发给设备、接口和端点
RESERVED2,
SET_ADDRESS,
GET_DESCRIPTOR,//其余的全部都是发往设备处理。
SET_DESCRIPTOR,
GET_CONFIGURATION,
SET_CONFIGURATION,
GET_INTERFACE,//这个发往接口
SET_INTERFACE,//这个也是发往接口。
TOTAL_sREQUEST, /* Total number of Standard request */
SYNCH_FRAME = 12
} STANDARD_REQUESTS; //标志请求总共 13 种，USB 实现 11 种。

typedef enum _DESCRIPTOR_TYPE
{
```

```
    DEVICE_DESCRIPTOR = 1,

    CONFIG_DESCRIPTOR,

    STRING_DESCRIPTOR,

    INTERFACE_DESCRIPTOR,

    ENDPOINT_DESCRIPTOR

    } DESCRIPTOR_TYPE; //描述符类型，这里都是标志描述符，类特殊描述符由用户程序支持。

    typedef enum _FEATURE_SELECTOR

    {

    ENDPOINT_STALL,

    DEVICE_REMOTE_WAKEUP

    } FEATURE_SELECTOR;//这个是可以设置的特性，Get_Feature（）和 Set_Feature（）请求所用。



    #define REQUEST_TYPE 0x60 /* 请求发送方屏蔽位，可能是标准请求、类请求和厂商请求。 */

    #define STANDARD_REQUEST 0x00 /* Standard request */

    #define CLASS_REQUEST 0x20 /* Class request */

    #define VENDOR_REQUEST 0x40 /* Vendor request */



    #define RECIPIENT 0x1F /* 请求接收对象的屏蔽位。 */
```

# 7、usb_core.h

一些核心处理结构体就是在这个头文件中定义的。

typedef enum _CONTROL_STATE

{

WAIT_SETUP, /* 0 */初始状态是等待 SETUP

SETTING_UP, /* 1 */收到 SETUP 后是建立中。

IN_DATA, /* 2 */

OUT_DATA, /* 3 */

LAST_IN_DATA, /* 4 */

LAST_OUT_DATA, /* 5 */

WAIT_STATUS_IN, /* 7 */数据阶段是 OUT，则状态阶段主机发 IN。

WAIT_STATUS_OUT, /* 8 */

STALLED, /* 9 */用户请求不支持，等待下一个 SETUP。

PAUSE /* 10 */这个可能表示出错了。

} CONTROL_STATE; /* 控制传输阶段对主机信号的处理，是以状态机类似的处理方式，这个就是设备可能的状态。*/

typedef struct _ENDPOINT_INFO

{//端点传输信息结构体，前面本来有一段很长的英文注释，重点讲解 CopyData 的作用。实际它不是用于复制的，主要是指明数据长度、返回数据缓冲指针的。

u16 Usb_wLength; //还有多少数据需要操作。

u16 Usb_wOffset; //当前数据缓冲的偏移

```
u16 PacketSize; //端点支持的包长度。
u8 *(*CopyData)(u16 Length); //这是一个函数指针定义。
}ENDPOINT_INFO;

typedef struct _DEVICE_INFO
{
u8 USBbmRequestType; /* bmRequestType */
u8 USBbRequest; /* bRequest */
u16_u8 USBwValues; /* wValue */
u16_u8 USBwIndexs; /* wIndex */
u16_u8 USBwLengths; /* wLength */
```

以上部分从 SETUP 包后面跟的数据包中获得。

```
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;/ */

ENDPOINT_INFO Ctrl_Info;
}DEVICE_INFO;
```

这个就是控制传输最重要的结构体定义。

```
typedef struct _DEVICE_PROP
```

```
{
void (*Init)(void); /* Initialize the device */
void (*Reset)(void); /* Reset routine of this device */

void (*Process_Status_IN)(void);
void (*Process_Status_OUT)(void);//用户对状态过程的特殊处理
RESULT (*Class_Data_Setup)(u8 RequestNo);
RESULT (*Class_NoData_Setup)(u8 RequestNo); //有些类特殊请求，usb_core.c 并不会支持，而交由上层用户处理。
RESULT(*Class_Get_Interface_Setting)(u8 Interface,u8 lternateSetting);

u8* (*GetDeviceDescriptor)(u16 Length);
u8* (*GetConfigDescriptor)(u16 Length);
u8* (*GetStringDescriptor)(u16 Length); //描述符都是与具体功能设备相关的，上层用户要提供给底层“usb_core.c”统一的操作模式。
u8* RxEP_buffer;
u8 MaxPacketSize;
}DEVICE_PROP;
```

对四个最重要函数的声明：

```
u8 Setup0_Process(void);
u8 Post0_Process(void);
u8 Out0_Process(void);
```

```
u8 In0_Process(void);
```

对四个最重要结构体的声明：

```
extern DEVICE_PROP Device_Property;

extern USER_STANDARD_REQUESTS User_Standard_Requests;

extern DEVICE Device_Table;

extern DEVICE_INFO Device_Info;
```

这四个结构体前三个由用户定义和实现，第四个结构体由“usb_init.c”定义和初始化，主要在“usb_core.c”中起作用。

今天的分析就先到这儿，明天再详细分析一下“usb_core.c”。

# USB的“JoyStickMouse”源代码分析 02

## 三、USB函 数库分析-核心处理函数usb_core.c

以下是几个个重要的函数：

```
uint8_t Setup0_Process(void);

uint8_t Post0_Process(void);

uint8_t Out0_Process(void);

uint8_t In0_Process(void);

static void DataStageOut(void);

static void DataStageIn(void);

static void NoData_Setup0(void);

static void Data_Setup0(void);
```

## 1、Setup0_Process(void)的 处理过程

```
u8 Setup0_Process(void)

{

   pBuf.b = PMAAddr + (u8 *)(_GetEPRxAddr(ENDP0) * 2);

//在 SETUP 中断 时，它后面所跟随的请求数据包，已经存入了端点 0 的接收缓冲区，这个表达式是取得该地 址。

   if (pInformation->ControlState != PAUSE)

   {

      pInformation->USBbmRequestType = *pBuf.b++; /请求 特征吗，一个字节，表明数据方向、发送者、请求接收对象*/

      pInformation->USBbRequest = *pBuf.b++; /*具体 请求*/
```

```
            pBuf.w++;   /* 后面 两个字节为空，跳过去*/
            pInformation->USBwValue = ByteSwap(*pBuf.w++); /*wValue */
            pBuf.w++;   /*后面 两个字节为空，跳过去*/
            pInformation->USBwIndex   = ByteSwap(*pBuf.w++); /*wIndex */
            pBuf.w++;   /*后面 两个字节为空，跳过去*/
            pInformation->USBwLength = *pBuf.w; /*wLength */
        }

        pInformation->ControlState = SETTING_UP; //设置 状态“正在 SETUP”
        if (pInformation->USBwLength == 0)
        {
            NoData_Setup0();   //像设置地址、设置配置这些请求是没有数据过程的，调用该函数处理。
        }
        else
        {
            Data_Setup0();//像获取描述符，设置报告这些是带数据 的。
        }
        return Post0_Process();
    }
```

## 2、NoData_Setup0 (void)的处理过程

```
if(Type_Recipient== (STANDARD_REQUEST | DEVICE_RECIPIENT))
```

如 果是标准请求 STANDARD_REQUEST，且请求对象为设备 DEVICE_RECIPIENT， 则在这个代码块里进行处理。

这 个代码块里处理三个标准请求：设置配置、设置地址和设置设备特性（好像是远程唤醒特性）。都是调用相应的标准设置函数实现的。比如：

```
Result = Standard_SetConfiguration()
```

第二个代码块：else if (Type_Recipient == (STANDARD_REQUEST | INTERFACE_RECIPIENT))

这 个代码块下面的请求是发往接口的：只支持设置接口命令。

第三个代码块：else if (Type_Recipient == (STANDARD_REQUEST | ENDPOINT_RECIPIENT))

第 三个代码块的请求是发往端点的：包括设置特性和清除特性两个请求。

如果 请求不属于以上部分，则交由下面处理：

```
if (Result != USB_SUCCESS)
{   //交由用户提供的类处理函数 Class_NoData_Setup 处 理。
    Result = (*pProperty->Class_NoData_Setup)(RequestNo);
    if (Result == USB_NOT_READY)
    {     //如果用户层也不支持这个请求，则 SETUP 失败。
        ControlState = PAUSE;
        goto exit_NoData_Setup0;
```

```
        }
    }
```

如 果请求被成功执行：

```
    ControlState = WAIT_STATUS_IN;/* 进入状 态过程，等待主机的 IN 指令，然后返回一个 0 字节的 状态数据包。*/
    USB_StatusIn();
```

这 个函数出口时，ControlState 可能是三种状态。

Stalled：表 明本次请求失败，遇到 IN、OUT 包 均不响应了，直到下一个 SETUP 到来。

PAUSE：USB 设备 不再接受请求，枚举失败。

WAIT_STATUS_IN：本 次处理成功，期待主机的“IN 包”，进入状态过程。实际上是在“IN 包” 的处理过程中，真正处理设置地址的请求。

从 这个函数回到 Setup0_Process(void)后，还要调用 Post0_Process(void)。 这里 Post0_Process(void)的 主要作用是如果处理状态变为，则将端点状态设置为“stalled”，不再响应“IN” 和“OUT”包，只响应“SETUP”包。

# 3、Data_Setup0 (void)的处理过程

如 果请求数据的长度不等于 0，则进入 Data_Setup0 进 行处理。

```
    if (Request_No == GET_DESCRIPTOR)
    {
        if(Type_Recipient==(STANDARD_REQUEST| EVICE_RECIPIENT))
```

```
    {
        u8 wValue1 = pInformation->USBwValue1;
        if (wValue1 == DEVICE_DESCRIPTOR)
        {
            CopyRoutine = pProperty->GetDeviceDescriptor;
        }
```

这 个代码块是请求描述符的，标准请求三种情况：设备描述符、配置描述符和字符串描述符。因 为这些描述符都有具体应用和用户提供，所以它需要调用用户的相应函数 pProperty->GetDeviceDescriptor 等 等。

当 然这个函数并不实际承担数据转移的任务，它只是获取描述符长度（还剩多少字节要传输）、对当前数据传输进行定位（偏移到多少了）。

```
    else if ((Request_No = GET_STATUS) && (pInformation->USBwValue = 0)
                    && (pInformation->USBwLength == 0x0002)
                    && (pInformation->USBwIndex1 == 0))
```

这 个代码块实现获取状态的请求，返回数据要求 2 个字节。获取 状态可以分别针对设备、接口和端点。

接下 来两个代码块是获取配置和获取接口的。这里就不详细分析了。

```
    if (CopyRoutine)   //如果 是标准请求，且参数正确，执行这个代码块。
    {
        pInformation->Ctrl_Info.Usb_wOffset = wOffset;   //初始偏移为 0
        pInformation->Ctrl_Info.CopyData = CopyRoutine;
```

```
    (*CopyRoutine)(0);   //这个 实际获得需要传输的数据的长度。
    Result = USB_SUCCESS;
}
Else   //如果是类特殊请求，则交由用户处理。
{
  Result = (*pProperty->Class_Data_Setup)(pInformation->USBbRequest);
    if (Result == USB_NOT_READY)
    {   //如果 用户也不支持，通信失败。
        pInformation->ControlState = PAUSE;
        return;
    }
}
```

接 下来是确定数据阶段的传输方向，主机可能是要发送数据：如 Set_Descripter， 也可能是要取得数据，如：Get_Descripter。

```
if (ValBit(pInformation->USBbmRequestType, 7))
{
      vu32 wLength = pInformation->USBwLength; //主机 要获取的长度。

    if (pInformation->Ctrl_Info.Usb_wLength > wLength)
    {     //如果 实际长度超过了需求长度，把实际长度减少为主机需求长度
        pInformation->Ctrl_Info.Usb_wLength = wLength;
    }
```

```
            Else if (pInformation->Ctrl_Info.Usb_wLength < pInformation->USBwLength)
//实际能 传输数目要小于主机需求长度
        {
            if (pInformation->Ctrl_Info.Usb_wLength< Property->MaxPacketSize)
                {
                    Data_Mul_MaxPacketSize = FALSE;
                }
            Else if((pInformation->Ctrl_Info.Usb_wLength% pProperty->MaxPacketSize)
== 0)
                {//如果实际长度是最大数据包长的整数倍，要传输 0 字节 数据包。
                    Data_Mul_MaxPacketSize = TRUE;
                }
            } //大多数情况下，主机指定长度应该与实际传输长度要对应。
            pInformation->Ctrl_Info.PacketSize = pProperty->MaxPacketSize;
            DataStageIn(); //如果数据阶段是“IN”， 则现在就要准备好数据，设置端点
TX 状态。
        }
        else
        {   //如果数据阶段是“OUT”则 主要在“Out0_Process()”函数中处理。这里只需
要改变状态，使 能端点接收就行了。
            pInformation->ControlState = OUT_DATA;
            vSetEPRxStatus(EP_RX_VALID); /* enable for next data reception */
        }
```

## 4、DataStageIn()的 处理过程

这 个函数主要是将用户缓冲区的数据（以描述符为最典型），复制到控制端点 0“TxADDR”所指向的端点数据输出缓冲区。

```
void DataStageIn(void)
{
   ENDPOINT_INFO *pEPinfo = &pInformation->Ctrl_Info;
   u32save_wLength = pEPinfo->Usb_wLength;//剩下 的需要传输的长度。
   u32 ControlState = pInformation->ControlState;
   u8 *DataBuffer;
   u32 Length;
   if ((save_wLength == 0) && (ControlState == LAST_IN_DATA))
   {
       if(Data_Mul_MaxPacketSize == TRUE)
       {
           Send0LengthData();   //发 0 字节数 据包。
           ControlState = LAST_IN_DATA;
           Data_Mul_MaxPacketSize = FALSE;
       }//0 字节数据包也发了，进入状态阶段。
       else
       {   //已经传 输完成，进入状态阶段。
           ControlState = WAIT_STATUS_OUT;
           vSetEPTxStatus(EP_TX_STALL);
       }
```

```
            goto Expect_Status_Out;

        }

        Length = pEPinfo->PacketSize;

        ControlState =(save_wLength <= Length) ? LAST_IN_DATA : IN_DATA;   //这里 根据剩余要发送的字节数与每次能发送的最大字节数相比，确认是否最后一次发送。

        if (Length > save_wLength)

        {

            Length = save_wLength;   //length 现在 为实际传输字节数。可能是用户缓冲区还剩余的字节数，也可能就等于数据包长。

        }//为了使概念更清晰一些，我这里举出具体数据。比如主机索取 255 个 字节，描述符实际长度为 129 字节，包长为 64 个字 节。判断后的状态为“IN_DATA”.

    则 到了这儿 Length = 64，也就是本次传输 64 个字 节。

        DataBuffer =(*pEPinfo->CopyData)(Length); //第 一次传输时，偏移为 0，DataBuffer 指 向缓冲区开头。

         UserToPMABufferCopy(DataBuffer, GetEPTxAddr(ENDP0), Length);

    //复 制 64 个字节到端点发送缓冲区。只要主机的“IN”一 来，数据马上返回给主机。

        SetEPTxCount(ENDP0, Length);



        pEPinfo->Usb_wLength -= Length;//此 时剩余长度为 65

        pEPinfo->Usb_wOffset += Length;//偏 移指向 64

        vSetEPTxStatus(EP_TX_VALID);

        USB_StatusOut();//这里是指用户可以取消“IN 数据 过程”，这个是在“Out0_Process()”中完成的。

    Expect_Status_Out:

        pInformation->ControlState = ControlState;
```

```
}
```

## 5、In0_Process()的 处理过程

数 据过程从上面蓝色的文字给出的状态开始：

在 主机的“IN”令牌包发出后，首先取走了“64 个字 节”，然后进入中断，并调用 In0_Process()。

```
if ((ControlState == IN_DATA) || (ControlState == LAST_IN_DATA))
{
    DataStageIn(); //继 续调用它准备下面的数据。
    //第二次调用时，用 户缓冲区指针指向 64，实际传输长度 64,剩余 长度变为 1，偏移指向 128，状 态还是“IN_DATA”。
```

第 三次调用时，状态变成“LAST_DATA”，缓冲区指向 128， 实际传输长度只有 1 个字节。再次发生“IN”的 时候，就会转入状态阶段。

```
    ControlState = pInformation->ControlState;
}
```

## 6、Out0_Process()的 处理过程

```
if((ControlState == OUT_DATA) || (ControlState == LAST_OUT_DATA))
{
    DataStageOut();
    ControlState = pInformation->ControlState; /* may be changed outside the function */
```

```
}
```

这个 函数最主要就是调用进行数据处理，将端点接收缓冲区收到的数据复制到用户缓冲区。

# 7、DataStageOut()的 处理过程

```
save_rLength = pEPinfo->Usb_rLength;

if (pEPinfo->CopyData && save_rLength)

{

    Length = pEPinfo->PacketSize;   //初始 化为数据包长。

    if (Length > save_rLength)

    {       //实际 不需要这么大，这长度改为实际长度。

        Length = save_rLength;

    }   //如果实 际长度大于等于包长，则复制一个包长。

    Buffer = (*pEPinfo->CopyData)(Length); //定位 用户缓冲区。

    pEPinfo->Usb_rLength -= Length;

    pEPinfo->Usb_rOffset += Length;

    PMAToUserBufferCopy(Buffer, GetEPRxAddr(ENDP0), Length);

}



if (pEPinfo->Usb_rLength >= pEPinfo->PacketSize)

{     //根据 剩余长度确定控制传输状态。

    pInformation->ControlState = OUT_DATA;

}

else
```

```
{       if (pEPinfo->Usb_rLength > 0)
    {       //还有数 据但是小于一个包。则下一次为最后一次传输。
        pInformation->ControlState = LAST_OUT_DATA;
    }
    else if (pEPinfo->Usb_rLength == 0)
    {   //没有数 据了，期待状态过程。
        pInformation->ControlState = WAIT_STATUS_IN;
        USB_StatusIn();   //先把 0 字节 数据包准备好。
    }
}
```

通过 把这几个核心函数详细的看一遍，我对于 USB 控制传输阶段的流程感觉已经比较熟悉 了。

# 8、其 它处理函数

主 要包括：

```
RESULT Standard_SetEndPointFeature(void);
RESULT Standard_SetDeviceFeature(void);

u8 *Standard_GetConfiguration(u16 Length);
RESULT Standard_SetConfiguration(void);
u8 *Standard_GetInterface(u16 Length);
RESULT Standard_SetInterface(void);
u8 *Standard_GetDescriptorData(u16 Length, PONE_DESCRIPTOR pDesc);
```

```
u8 *Standard_GetStatus(u16 Length);

RESULT Standard_ClearFeature(void);

void SetDeviceAddress(u8);
```

代 码都不多，我就简单的浏览了一遍，就不详细分析了。

# USB的“JoyStickMouse”源代码分析 03

## 四、USB应 用层主要函数分析

### 1、用 户协议的主要描述：usb_desc.c

为了使枚举过程更清晰，对于控制 传输阶段描述符的使用更形象，我在核心处理函数中加入一些调试语句，向串口输出数据，得到“JoyStickMouse” 的枚举过程完整数据。这样对于学习标准请求和控制传输很直观。

调试函数以下列形式出现，如果不 需要后，定义 usb_debug 为 0 就行了。

```
#if usb_debug
    Uart_PutString("setup 中断\r\n" );
    Uart_PutHex ( pInformation->USBbmRequestType );
    Uart_PutHex ( pInformation->USBbRequest );
    Uart_PutHex ( pInformation->USBwValue0 );
    Uart_PutHex ( pInformation->USBwValue1 );

    Uart_PutHex ( pInformation->USBwIndex0 );
    Uart_PutHex ( pInformation->USBwIndex1 );
    Uart_PutHex ( pInformation->USBwLength1 );
    Uart_PutHex ( pInformation->USBwLength0 );
    Uart_PutString("\r\n" );
#endif
```

进入 USB 鼠标实现过程!

总线复位中断

总线复位中断

setup 中断　　//80 表示数据输入、主机请求发往设备。06 是获取描述符

80 0600 0100 00 40 00 // 00 01Value 高字节 01 表示设备，请求 0x40 字节。

获取设备描述符

设备准备发送 12 字节 12 0100 02 00 00 00 4083 04 10 57 00 02 01 02 03 01　　// 40 表示端点支持的包长 64 字节

IN 令牌 04 中断　　//04 表示这是“LAST_IN_DATA”，一次性传输完。02 表示需 要分成多次传输。

OUT 状态中断

总线复位中断

setup 中断　　// 00 表示 主机标准请求到设备，数据输出。

00 050200 00 00 00 00

设置地址//地址 00 02

IN 状态中断

setup 中断

80 06 00 01 00 00 12 00

获取设备描述符　　// 以新地址 02 获取描述符。

设备准备发送 12 字节 12 01 00 02 00 00 0040 83 04 10 57 00 02 01 02 03 01　　// 00 00 00 代表设备类、设备子类、协议。

IN 令牌 04 中断

OUT 状态中断

setup 中断

80 06 00 0200 00 09 00

获取配置描述符 // 00 22 表示配置集合有 34 个字节。

设备准备发送 09 字节 09 02 22 0001 0100 E0 32

IN 令牌 04 中断　//一个接口、配置号为 1。E0 表示自供电、支持远程唤醒。32 表示 50 个电流单位，100mA。

OUT 状态中断

setup 中断

80 06 0003 00 00 FF 00

获取字符串描述符　// 03 表示获取字符串描述符，00 表示获取语言 ID。

设备准备发送 04 字节 04 03 09 04

IN 令牌 04 中断　　// 04 09 表示英语。

OUT 状态中断

setup 中断

80 06 0303 09 04FF 00

获取字符串描述符　//以用户提供的语言 ID 获取 设备序列号字符串（03）。

设备准备发送 1A 字节 1A03 53 00 54 00 4D 00 33 00 32 00 31 00 30 00 00 00 00 00 00 00 00 00 00 00 00

IN 令牌 04 中断

OUT 状态中断

setup 中断

80 06 00 02 00 00 FF 00

获取配置集合：包括蓝色的接口描述符、红色 HID 类描述符、蓝色端点描述符。//0103 01 02 后面 3 个字节表示接口类、接口子类、接口协议。

设备准备发送 22 字节 09 02 22 00 01 01 00 E0 32 09 04 00 00 0103 01 02 00 09 2100 01 00 01224A00 07 05 81 0304 00 20

IN 令牌 04 中断 // 22 表示报告描述符类型。4A 表 示 76 个字节。

OUT 状态中断 //81 03 表示使用 1 号端点输入、中断类型,20 表 示轮询时间要求 32ms。

setup 中断

80 06 00 06 00 00 0A 00

setup 中断

80 06 00 03 00 00 FF 00

获取字符串描述符

设备准备发送 04 字节 04 03 09 04

IN 令牌 04 中断

OUT 状态中断

setup 中断

80 06 0203 09 04FF 00

获取字符串描述符　//又获取字符串描述符，这次是产品名称 字符串。

设备准备发送 1E 字节 1E 03 53 00 54 00 4D 00 33 00 32 00 20 00 4A 00 6F 00 79 00 73 00 74 00 69 00 63 00 6B 00

IN 令牌 04 中断

OUT 状态中断

setup 中断

80 06 00 03 00 00 FF 00

获取字符串描述符

设备准备发送 04 字节 04 03 09 04

IN 令牌 04 中断

OUT 状态中断

setup 中断

80 06 02 03 09 04 FF 00

获取字符串描述符　//重复获取，为什么这么做呢？

设备准备发送 1E 字节 1E 03 53 00 54 00 4D 00 33 00 32 00 20 00 4A 00 6F 00 79 00 73 00 74 00 69 00 63 00 6B 00

IN 令牌 04 中断

OUT 状态中断

setup 中断

80 06 000100 00 12 00

获取设备描述符　　//又获取设备描述符，新地址下第二次。

设备准备发送 12 字节 12 01 00 02 00 00 00 40 83 04 10 57 00 02 01 02 03 01

IN 令牌 04 中断

OUT 状态中断

setup 中断

80 06 00 0200 00 09 00

获取配置描述符　//又获取配置描述符，新地址下第二次。

设备准备发送 09 字节 09 02 22 00 01 01 00 E0 32

IN 令牌 04 中断

OUT 状态中断

setup 中断

80 06 00 0200 00 22 00

获取配置描述符集合。　//又获取配置描述集合，新地址下第二次

设备准备发送 22 字节 09 02 22 00 01 01 00 E0 32 09 04 00 00 01 03 01 02 00 09 21 00 01 00 01 22 4A 00 07 05 81 03 04 00 20

IN 令牌 04 中断

OUT 状态中断

setup 中断

00 090100 00 00 00 00

设置配置　//设置配置，用户的配置就有效了。

IN 状态中断

setup 中断//21 表示类请求，发送到接口。

210A00 00 00 00 00 00　　//0A 表示 Set_Idle 请求，设备无反应，故主机忽略。

setup 中断

81 06 00 2200 00 8A00　　//这里大小应该是 00 4A 吧？

获取报告描述符

设备准备发送 40 字节 05 01 09 02 A1 01 09 01 A1 00 05 09 19 01 29 03 15 00 25 01 95 03 75 01 81 02 95 01 75 05 81 03 05 01 09 30 09 31 09 38 15 81 25 7F 75 08 95 03 81 06 C0 09 3C 05 FF 09 01 15 00 25 01 75 01 95

IN 令牌 02 中断　　//02 表 示后面还有数据传输。

设备准备发送 0A 字节 02 B1 22 75 06 95 01 B1 01 C0

IN 令牌 04 中断

OUT 状态中断

## 2、用 户协议的主要实现：usb_prop.c

主 要是实现两个函数指针结构体。

```
DEVICE_PROP Device_Property =
    {
        Joystick_init,    //用户 初始化，清除复位、开启中断、连接电缆。
```

```
        Joystick_Reset,    // 使能 端口 0
        Joystick_Status_In,
        Joystick_Status_Out,    // 两 个空函数。
        Joystick_Data_Setup,    // 实现报告描述符请求、HID 类描述 符请求、协议获取请求
        Joystick_NoData_Setup, // 实现设 置协议请求。
        Joystick_Get_Interface_Setting,    // 有实现，但意义不太明白。
        Joystick_GetDeviceDescriptor,
        Joystick_GetConfigDescriptor,
        Joystick_GetStringDescriptor,    //以 上三个是上层协议为底层描述符请求提供具体数据。
        0,
        0x40 /*MAX PACKET SIZE*/   // 端点支持的包长。
    };
USER_STANDARD_REQUESTS User_Standard_Requests =
    {
        Joystick_GetConfiguration,
        Joystick_SetConfiguration,    //这个函数调用后，设备为可用状态。
        Joystick_GetInterface,
        Joystick_SetInterface,
        Joystick_GetStatus,
        Joystick_ClearFeature,
        Joystick_SetEndPointFeature,
        Joystick_SetDeviceFeature,
```

```
    Joystick_SetDeviceAddress //这个函 数调用后，设备状态为地址状态。

};   // 其 它函数都是空函数。是在控制传输过程中用户提供的回调函数。
```