USB 的"JoyStickMouse”例程分析

发布: 2010-10-25 21:49 | 作者: AVR 侠 | 来源: PowerAVR 电子 DIY 网

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，则相应的数据传输完成中断发生。第15位	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。

1.jpg

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。			

2.jpg

（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。

<table>
<tr><td>CTR_RX.正确接收标志位。

第15位。</td><td>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。</td><td colspan="2">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。</td></tr>
<tr><td>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。</td><td colspan="2">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。</td><td>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。</td></tr>
<tr><td>CTR_TX。
正确发送标志。主机的IN包之后。
第7位。</td><td>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。</td><td colspan="2">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。</td></tr>
<tr><td colspan="4">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。</td></tr>
</table>

3.jpg

（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

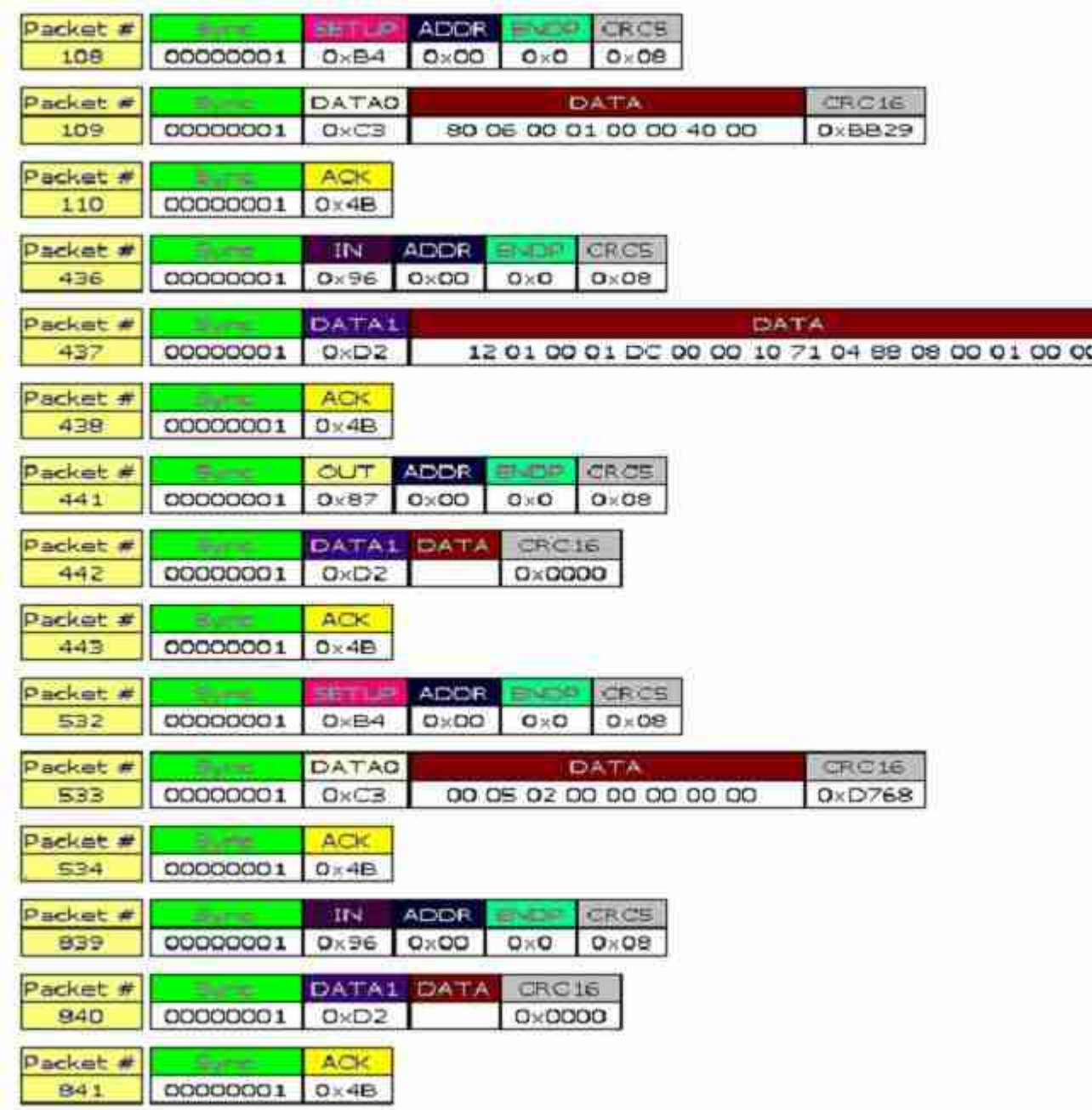
主机收到 18 个字节的描述符后，进入状态事务过程，此过程的令牌包为 OUT，字节数为 0.只需要用户回一个 ACK。

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。



5.jpg

上图很好地描述了枚举阶段"获取描述符"和"设置地址"两个阶段主机和设备数据交换的过程。

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() Send0LengthData() //准备发送 0 字节数据
#define Send0LengthData() { _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：

```
Send0LengthData();
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 字节 DATA1、主机 ACK。

主机 ACK 之后，设备进入 CTR 中断，调用函数 In0_Process(void)来处理。根据当前控制传输状态，该函数把状态改为“STALLED”，退出时将端点状态改为“STALL”。状态阶段完成。

5、主机类特殊请求：设置空闲

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。

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。