基于STM32 的USB程序开发笔记

以前一直就有打玩 USB 的想法，最近时间充足于是决心打玩 STM32 的 USB，购买的是万利的 STM3210B-LK1 板，琢磨 USB 已有半个多月，在固 件、上位机驱动以及应用程序的访问这三方面终于有所突破，这期间通过网络上授寻了许多相关资料，主要来自 ST 提供的 USB 固件，以及圈圈 (computer00)提供的一些关于 USB 驱动开发的资料，通过这段时间的学习，了解到学习 USB 对于未接触过的朋友来说确实存在许多的知识壁垒，本 着开源精神，在此对 STM32 的 USB 固件程序的编写、DriverStudio + WindowsXP DDK + VC6 驱动开发以及应用程序做了一些介绍，为更好理解，请仔细学习 STM32 USB 的参考手册以及 USB 协议，如果对 VC6 下开发还不是太熟悉或者说不曾学过，那么如果想理解有些问题，就必须学习 VC6 了。本套笔记是基于我编写的程序进行说明的，请配合该程序进行学习。

第一篇：需要准备的一些资料

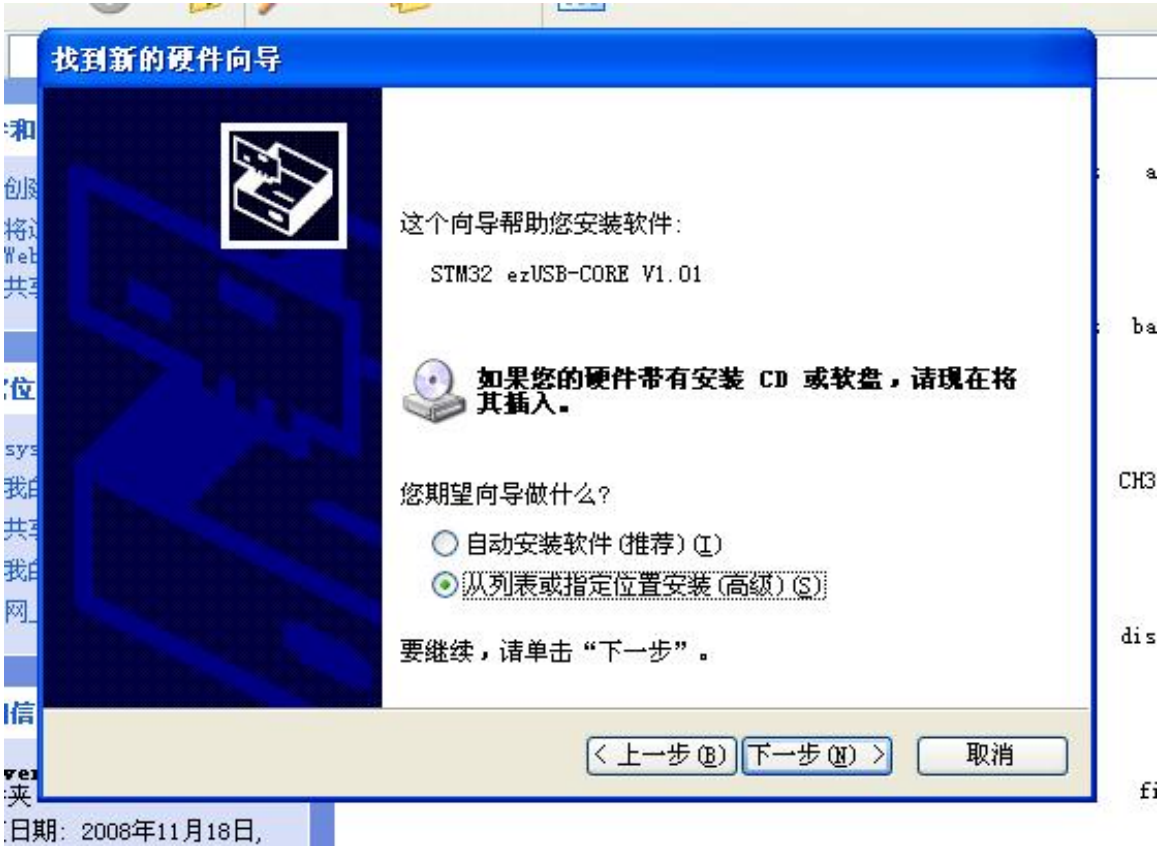
1：STM32 的参考手册，这对于设备底层 USB 的硬件配置以及事件驱动机制的了解尤为重要，你需要了解各个寄存器的功能以及如何操作，比如 CNTR、ISTR、EPnR、DADDR 等等，如果你想学习 USB，这个手册是必须的。

2：USB2.0 协议，这个资料同样必不可少，如果因为英语阅读能力而苦苦寻找中文版的 USB2.0 协议，建议不要这么做，现在网络中的所谓的中文版的 USB2.0 协议不 是官方撰写的，大多数是一些热心朋友自己翻译的，却不是很全面，如果你在为寻找这类的资料而无所获时，建议认真塌实的看看官方英文版的 USB2.0 协议， 官方协议阐述的十分详细，650 多页，一字一句的了解全部协议不太可行,可针对性的重点理解,比如对第 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)。

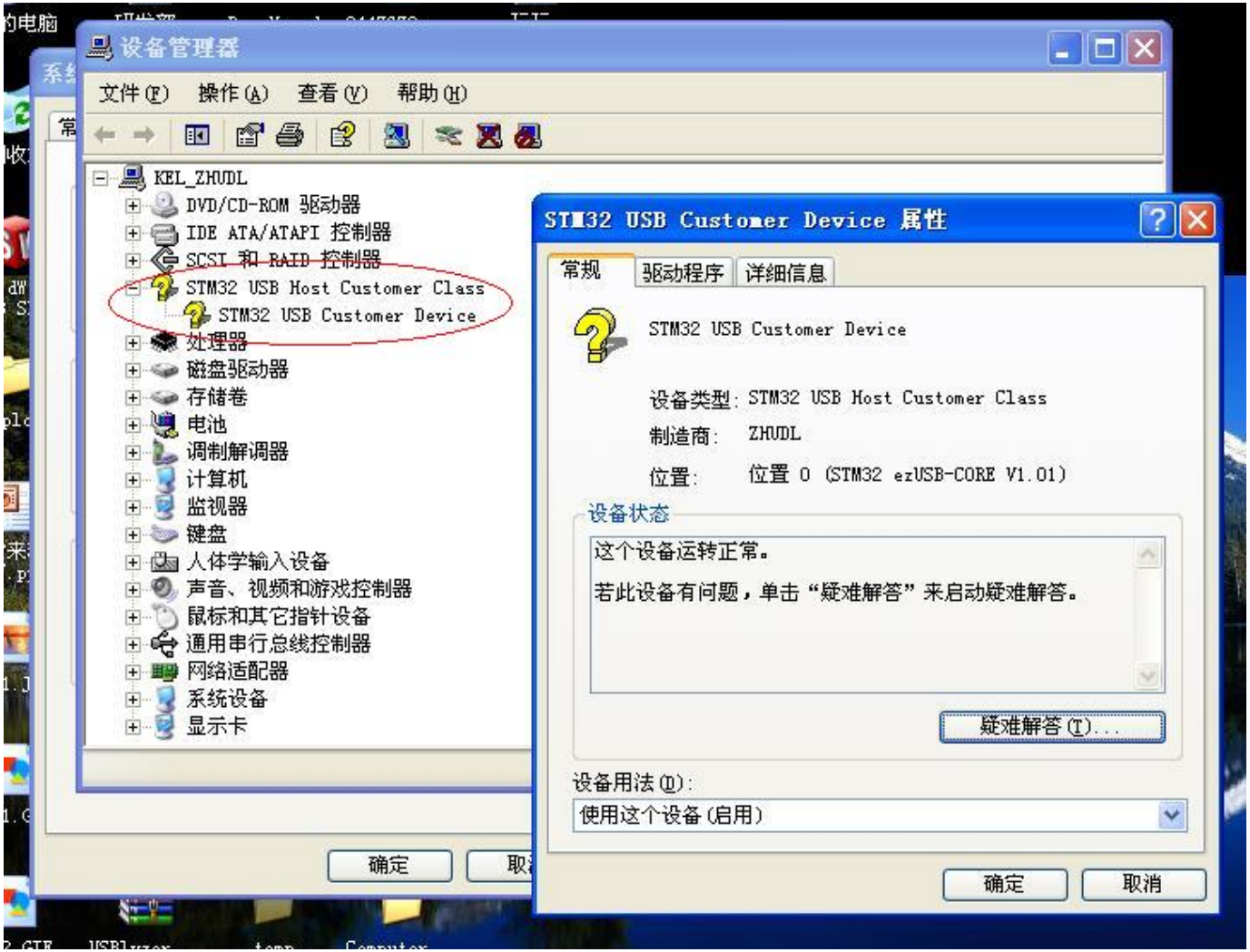
3：ST提供的USB固件库，这个类库较为散乱，但不可不参考

以下是链接包含固件、驱动以及应用程序，固件部分有些功能是不被支持的，如SR_SetDescriptor()、 SR_SynchFrame()等等，在此说明不支持非故意如此，而是还没去更仔细深入编写完善，目前这些不被支持的部分目前不被使用到。
下载链接：http://blog.ednchina.com/lbxxx

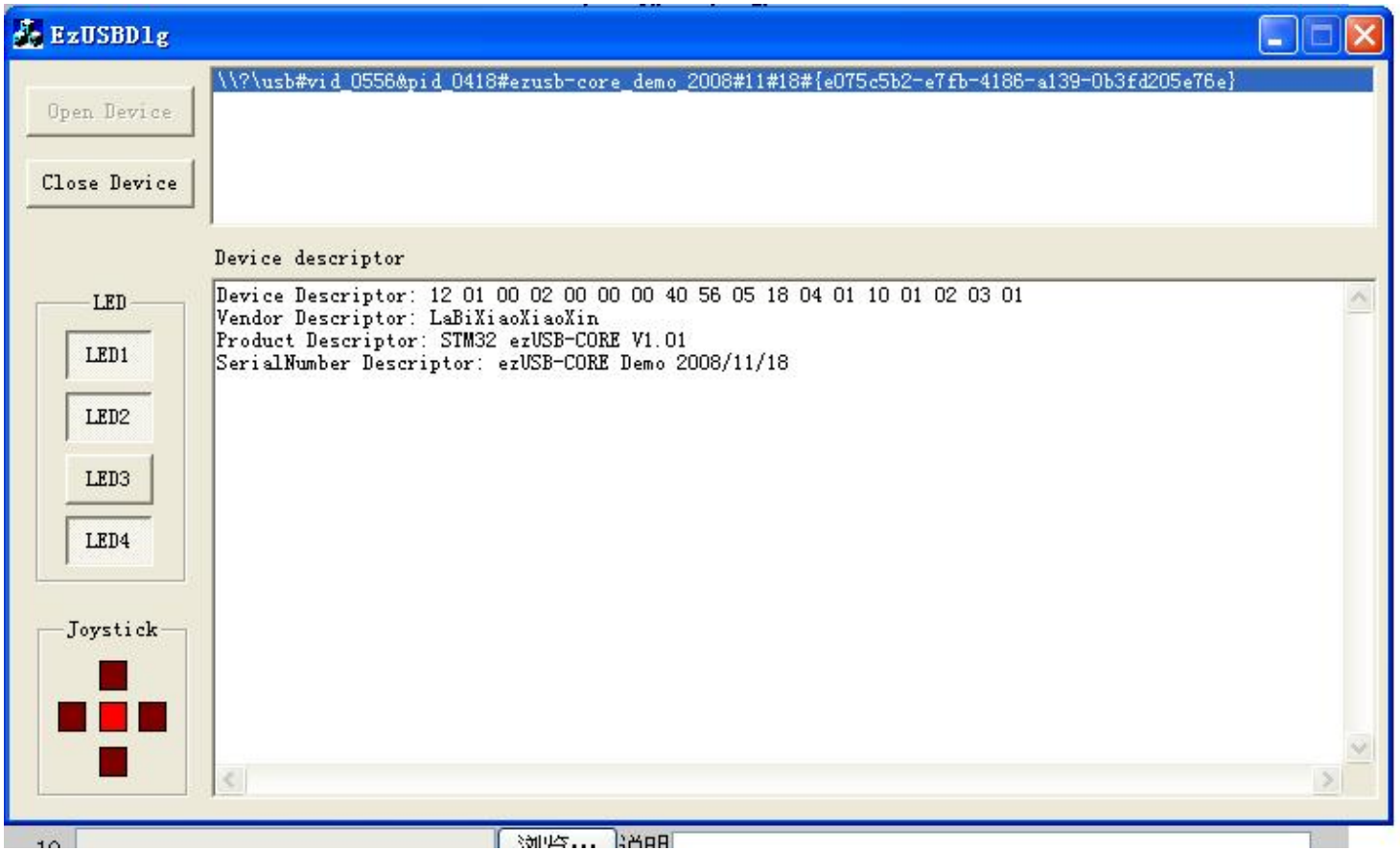
如果你使用的是万利的 STM3210B-LK1 开发板，则可以烧写 hex 文件后直接进行测试。以下一组图片说明的 XP 下驱动的安装过程以及测试软件打开后的情形，仅参考。




设备管理器
找到新的硬件向导
向导正在安装软件，请稍候...
STM32 USB Customer Device
ezUSB.sys
到 C:\WINDOWS\system32\DRIVERS
< 上一步(B)
下一步(N) >
取消


设备管理器
文件(F) 操作(A) 查看(V) 帮助(H)
KEL_ZHUDL
DVD/CD-ROM 驱动器
IDE ATA/ATAPI 控制器
SCSI 和 RAID 控制器
STM32 USB Host Customer Class
STM32 USB Customer Device
处理器
磁盘驱动器
存储卷
电池
调制解调器
计算机
监视器
键盘
人体学输入设备
声音、视频和游戏控制器
鼠标和其它指针设备
通用串行总线控制器
网络适配器
系统设备
显示卡
确定
STM32 USB Customer Device 属性
常规
驱动程序
详细信息
STM32 USB Customer Device
设备类型: STM32 USB Host Customer Class
制造商: ZHUDL
位置: 位置 0 (STM32 ezUSB-CORE V1.01)
设备状态
这个设备运转正常。
若此设备有问题，单击“疑难解答”来启动疑难解答。
疑难解答(T)...
设备用法(D):
使用这个设备(启用)
确定
取消

测试软件：控制万利 STM3210B-LK1 开发板的 4 个 LED，并定时读取 Joystick 状态



第 2 篇：STM32 USB 固件函数的驱动原理

首先需要了解一个概念：
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。
了解这些，再仔细查看 STM32 的参考手册 USB 部分以及 STM32 的中断向量表，从中可以找到两个中断：

```
/*******************************************************************************
* Function Name  : USB_HP_CAN_TX_IRQHandler
* Description    : This function handles USB High Priority or CAN TX interrupts
*                  requests.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void USB_HP_CAN_TX_IRQHandler(void)
{
  USB_HPI();
}

/*******************************************************************************
* Function Name  : USB_LP_CAN_RX0_IRQHandler
* Description    : This function handles USB Low Priority or CAN RX0 interrupts
*                  requests.
* Input          : None
* Output         : None
```

```
* Return         : None
*******************************************************************************/
void USB_LP_CAN_RX0_IRQHandler(void)
{
  USB_LPI();
}
```

即 USB 的高、低优先权中断处理函数，这也是整个 STM32 USB 的事件驱动源，USB_HPI()与 USB_LPI()既而转向 usb_core(.c,.h)进行相关处理。中断传输(interrupt)、控 制传输(control)、大流量传输(bulk)由 USB_LPI()响应，大流量传输(bulk)同样可能响应 USB_HPI()，同步传输 (isochronous)只响应 USB_HPI()。

这样响应 USB 的所有请求只需要关注 usb_core.c 文件中的 USB_LPI()与 USB_HPI()函数。由于本人也是对 USB 刚刚有所了解，因而在本例笔记中 USB_HPI()函数未做任何处理，在此开源希望大 家能完善与纠正错误并能共享喜悦。以下是 USB_LPI()函数：

```
// *******************************************************************************
// Function Name  : USB_LPI.
// Description    : Low Priority Interrupt's service routine.
// Input          :
// Output         :
// Return         :
// *******************************************************************************
void USB_LPI(void)
{
  unsigned short wValISTR = GetISTR();

#if(CNTR_MASK & ISTR_RESET)   // Reset
  if(wValISTR & ISTR_RESET & vwInterruptMask)
```

```
  {
    SetISTR(CLR_RESET);
    INT_ISTR_RESET();
  }
#endif

#if(CNTR_MASK & ISTR_DOVR)   // DMA Over/Underrun
  if(wValISTR & ISTR_DOVR & vwInterruptMask)
  {
    SetISTR(CLR_DOVR);
    INT_ISTR_DOVR();
  }
#endif

#if(CNTR_MASK & ISTR_ERR)   // Error
  if(wValISTR & ISTR_ERR & vwInterruptMask)
  {
    SetISTR(CLR_ERR);
    INT_ISTR_ERROR();
  }
#endif

#if(CNTR_MASK & ISTR_WKUP)    // Wakeup
  if(wValISTR & ISTR_WKUP & vwInterruptMask)
  {
    SetISTR(CLR_WKUP);
```

```
    INT_ISTR_WAKEUP();
  }
#endif

#if(CNTR_MASK & ISTR_SUSP)   // Suspend
  if(wValISTR & ISTR_SUSP & vwInterruptMask)
  {
    INT_ISTR_SUSPEND();
    SetISTR(CLR_SUSP);     // must be done after setting of CNTR_FSUSP
  }
#endif

#if(CNTR_MASK & ISTR_SOF)   // Start Of Frame
  if(wValISTR & ISTR_SOF & vwInterruptMask)
  {
    SetISTR(CLR_SOF);
    INT_ISTR_SOF();
  }
#endif

#if(CNTR_MASK & ISTR_ESOF)   // Expected Start Of Frame
  if(wValISTR & ISTR_ESOF & vwInterruptMask)
  {
    SetISTR(CLR_ESOF);
    INT_ISTR_ESOF();
  }
```

```
#endif

#if(CNTR_MASK & ISTR_CTR)   // Correct Transfer
  if(wValISTR & ISTR_CTR & vwInterruptMask)
  {
    INT_ISTR_CTR();
  }
#endif
}

// ********************************************************************************
// Function Name  : USB_HPI.
// Description    : High Priority Interrupt's service routine.
// Input          :
// Output         :
// Return         :
// ********************************************************************************
void USB_HPI(void)
{

}
```

可 以看出，在USB_LPI()函数中，根据STM32 USB的中断状态寄存器（ISTR）的标志位的状态以及定义的USB控制寄存器中断事件屏蔽码，响应各自的中断事件，比如 INT_ISTR_RESET()响应USB的复位中断，一般可在此函数内进行USB的寄存器的初始化；
INT_ISTR_CTR()响应一次正确的数据 传输中断，故名思意，在完成一次正确的数据传输操作后，就会响应此函数。
具体含义请仔细查阅STM32参考手册。

第 3 篇：STM32 USB 固件函数的一些介绍

STM32 USB 中断事件为以下几种，详细情况可以查看 usb_core(.c/.h):

```
void ISTR_CTR(void);
void ISTR_SOF(void);
void ISTR_ESOF(void);
void ISTR_DOVR(void);
void ISTR_ERROR(void);
void ISTR_RESET(void);
void ISTR_WAKEUP(void);
void ISTR_SUSPEND(void);
```

这些处理函数使能由定义 CNTR_MASK 决定：

```
// CNTR mask control
#define CNTR_MASK   CNTR_CTRM | CNTR_WKUPM | CNTR_SUSPM | CNTR_ERRM |     \
                    CNTR_SOFM | CNTR_ESOFM | CNTR_RESETM | CNTR_DOVRM     \
```

其中着重说明的是 ISTR_RESET()和 ISTR_CTR()函数，ISTR_RESET()主要处理 USB 复位后进行一些初始化任务，ISTR_CTR()则是处理数据正确传输后控制，比如说响应主机。

```
// ****************************************************************************
// Function Name  : INT_ISTR_RESET
// Description    : ISTR Reset Interrupt service routines.
// Input          :
// Output         :
// Return         :
```

```
// *****************************************************************************
void INT_ISTR_RESET(void)
{
  // Set the buffer table address
  SetBTABLE(BASEADDR_BTABLE);

  // Set the endpoint type: ENDP0
  SetEPR_Type(ENDP0, EP_CONTROL);
  Clr_StateOut(ENDP0);

  // Set the endpoint data buffer address: ENDP0 RX
  SetBuffDescTable_RXCount(ENDP0, ENDP0_PACKETSIZE);
  SetBuffDescTable_RXAddr(ENDP0, ENDP0_RXADDR);

  // Set the endpoint data buffer address: ENDP0 TX
  SetBuffDescTable_TXCount(ENDP0, 0);
  SetBuffDescTable_TXAddr(ENDP0, ENDP0_TXADDR);

  // Initialize the RX/TX status: ENDP0
  SetEPR_RXStatus(ENDP0, EP_RX_VALID);
  SetEPR_TXStatus(ENDP0, EP_TX_NAK);

  // Set the endpoint address: ENDP0
  SetEPR_Address(ENDP0, ENDP0);

  // ---------------------------------------------------------------------
```

```
// TODO: Add you code here
// ----------------------------------------------------------------------
// Set the endpoint type: ENDP1
SetEPR_Type(ENDP1, EP_INTERRUPT);
Clr_StateOut(ENDP1);

// Set the endpoint data buffer address: ENDP1 RX
SetBuffDescTable_RXCount(ENDP1, ENDP1_PACKETSIZE);
SetBuffDescTable_RXAddr(ENDP1, ENDP1_RXADDR);

// Set the endpoint data buffer address: ENDP1 TX
SetBuffDescTable_TXCount(ENDP1, 0);
SetBuffDescTable_TXAddr(ENDP1, ENDP1_TXADDR);

// Initialize the RX/TX status: ENDP1
SetEPR_RXStatus(ENDP1, EP_RX_VALID);
SetEPR_TXStatus(ENDP1, EP_TX_DIS);

// Set the endpoint address: ENDP1
SetEPR_Address(ENDP1, ENDP1);



SetEPR_Type(ENDP2, EP_INTERRUPT);
Clr_StateOut(ENDP2);
```

```
// Set the endpoint data buffer address: ENDP2 RX
SetBuffDescTable_RXCount(ENDP2, ENDP2_PACKETSIZE);
SetBuffDescTable_RXAddr(ENDP2, ENDP2_RXADDR);

// Set the endpoint data buffer address: ENDP2 TX
SetBuffDescTable_TXCount(ENDP2, 0);
SetBuffDescTable_TXAddr(ENDP2, ENDP2_TXADDR);

// Initialize the RX/TX status: ENDP2
SetEPR_RXStatus(ENDP2, EP_RX_DIS);
SetEPR_TXStatus(ENDP2, EP_TX_VALID);

// Set the endpoint address: ENDP2
SetEPR_Address(ENDP2, ENDP2);



// ----------------------------------------------------------------
// End of you code
// ----------------------------------------------------------------

SetDADDR(0x0080 | vsDeviceInfo.bDeviceAddress);
vsDeviceInfo.eDeviceState = DS_DEFAULT;
vsDeviceInfo.bCurrentFeature = 0x00;
vsDeviceInfo.bCurrentConfiguration = 0x00;
vsDeviceInfo.bCurrentInterface = 0x00;
```

```
  vsDeviceInfo.bCurrentAlternateSetting = 0x00;
  vsDeviceInfo.uStatusInfo.w = 0x0000;
}
```

在这个 ISTR_CTR()函数中，定义了 EP0、1、2 的传输方式以及各自的缓冲描述符，其中 EP0 是默认端口，负责完成 USB 设备的枚举，一般情况是不需要更改的。其他端点配置则需根据实际应用而决定，如何设置请仔细理解 STM32 的参考手册。

值 得说明的是 STM32 的端点 RX/TX 缓冲描述表是定义在 PMA 中的，他是基于分组缓冲区描述报表寄存器(BTABLE)而定位的，各端点 RX/TX 缓冲 描述表说明是数据存储地址以及大小，这个概念需要了解，ST 提供的固件很含糊，为此，我在 usb_regs.h 文件中进行了重新定义，如下：

```
// USB_IP Packet Memory Area base address
#define PMAAddr  (0x40006000L)

// Buffer Table address register
#define BTABLE  ((volatile unsigned *)(RegBase + 0x50))

// ******************************************************************************
// Packet memory area: Total 512Bytes
// ******************************************************************************
#define BASEADDR_BTABLE        0x0000
// ******************************************************************************
// PMAAddr + BASEADDR_BTABLE + 0x00000000 : EP0_TX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x00000002 : EP0_TX_COUNT
// PMAAddr + BASEADDR_BTABLE + 0x00000004 : EP0_RX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x00000006 : EP0_RX_COUNT
//
// PMAAddr + BASEADDR_BTABLE + 0x00000008 : EP1_TX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x0000000A : EP1_TX_COUNT
```

```
// PMAAddr + BASEADDR_BTABLE + 0x0000000C : EP1_RX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x0000000E : EP1_RX_COUNT
//
// PMAAddr + BASEADDR_BTABLE + 0x00000010 : EP2_TX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x00000012 : EP2_TX_COUNT
// PMAAddr + BASEADDR_BTABLE + 0x00000014 : EP2_RX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x00000016 : EP2_RX_COUNT
//
// PMAAddr + BASEADDR_BTABLE + 0x00000018 : EP3_TX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x0000001A : EP3_TX_COUNT
// PMAAddr + BASEADDR_BTABLE + 0x0000001C : EP3_RX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x0000001E : EP3_RX_COUNT
//
// PMAAddr + BASEADDR_BTABLE + 0x00000020 : EP4_TX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x00000022 : EP4_TX_COUNT
// PMAAddr + BASEADDR_BTABLE + 0x00000024 : EP4_RX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x00000026 : EP4_RX_COUNT
//
// PMAAddr + BASEADDR_BTABLE + 0x00000028 : EP5_TX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x0000002A : EP5_TX_COUNT
// PMAAddr + BASEADDR_BTABLE + 0x0000002C : EP5_RX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x0000002E : EP5_RX_COUNT
//
// PMAAddr + BASEADDR_BTABLE + 0x00000030 : EP6_TX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x00000032 : EP6_TX_COUNT
// PMAAddr + BASEADDR_BTABLE + 0x00000034 : EP6_RX_ADDR
```

```
// PMAAddr + BASEADDR_BTABLE + 0x00000036 : EP6_RX_COUNT
//
// PMAAddr + BASEADDR_BTABLE + 0x00000038 : EP7_TX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x0000003A : EP7_TX_COUNT
// PMAAddr + BASEADDR_BTABLE + 0x0000003C : EP7_RX_ADDR
// PMAAddr + BASEADDR_BTABLE + 0x0000003E : EP7_RX_COUNT
// *****************************************************************************
//
// PMAAddr + BASEADDR_BTABLE + (0x00000040 - 0x000001FF) : assigned to data buffer
//
// *****************************************************************************
#define BASEADDR_DATA   (BASEADDR_BTABLE + 0x00000040)

// ENP0
#define ENDP0_PACKETSIZE      0x40
#define ENDP0_RXADDR          BASEADDR_DATA
#define ENDP0_TXADDR          (ENDP0_RXADDR + ENDP0_PACKETSIZE)

// ENP1
#define ENDP1_PACKETSIZE      0x40
#define ENDP1_RXADDR          (ENDP0_TXADDR + ENDP0_PACKETSIZE)
#define ENDP1_TXADDR          (ENDP1_RXADDR + ENDP1_PACKETSIZE)

// ENP2
#define ENDP2_PACKETSIZE      0x40
#define ENDP2_RXADDR          (ENDP1_TXADDR + ENDP1_PACKETSIZE)
```

```
#define ENDP2_TXADDR          (ENDP2_RXADDR + ENDP2_PACKETSIZE)

// ENP3
#define ENDP3_PACKETSIZE      0x40
#define ENDP3_RXADDR          (ENDP2_TXADDR + ENDP2_PACKETSIZE)
#define ENDP3_TXADDR          (ENDP3_RXADDR + ENDP3_PACKETSIZE)

// ENP4
#define ENDP4_PACKETSIZE      0x40
#define ENDP4_RXADDR          (ENDP3_TXADDR + ENDP3_PACKETSIZE)
#define ENDP4_TXADDR          (ENDP4_RXADDR + ENDP4_PACKETSIZE)

// ENP5
#define ENDP5_PACKETSIZE      0x40
#define ENDP5_RXADDR          (ENDP4_TXADDR + ENDP4_PACKETSIZE)
#define ENDP5_TXADDR          (ENDP5_RXADDR + ENDP5_PACKETSIZE)

// ENP6
#define ENDP6_PACKETSIZE      0x40
#define ENDP6_RXADDR          (ENDP5_TXADDR + ENDP5_PACKETSIZE)
#define ENDP6_TXADDR          (ENDP6_RXADDR + ENDP6_PACKETSIZE)

// ENP7
#define ENDP7_PACKETSIZE      0x40
#define ENDP7_RXADDR          (ENDP6_TXADDR + ENDP6_PACKETSIZE)
#define ENDP7_TXADDR          (ENDP7_RXADDR + ENDP7_PACKETSIZE)
```

这样，一般只要在 PMA 的大小区域内（512Bytes），修改端点 EPnR 的数据包大小就可以了，当然，实际情况可以根据需要进行更改。

```
// *******************************************************************************
// Function Name  : INT_ISTR_CTR
// Description    : ISTR Correct Transfer Interrupt service routine.
// Input          :
// Output         :
// Return         :
// *******************************************************************************
void INT_ISTR_CTR(void)
{
  unsigned short wEPIndex;
  unsigned short wValISTR;
  unsigned short wValENDP;

  while( ((wValISTR=GetISTR()) & ISTR_CTR) != 0 )
  {
    // Get the index number of the endpoints
    wEPIndex =  wValISTR & ISTR_EP_ID;

    if(wEPIndex == 0)
    {
      // Set endpoint0 RX/TX status: NAK (Negative-Acknowlegment)
      SetEPR_RXStatus(ENDP0, EP_RX_NAK);
      SetEPR_TXStatus(ENDP0, EP_TX_NAK);
```

```
 // Transfer direction
if((wValISTR & ISTR_DIR) == 0)
{
  // DIR=0: IN
  // DIR=0 implies that EP_CTR_TX always 1
  ClrEPR_CTR_TX(ENDP0);
  CTR_IN0();
  return;
}
else
{
  // DIR=1: SETUP or OUT
  // DIR=1 implies that CTR_TX or CTR_RX always 1
  wValENDP = GetEPR(ENDP0);
  if((wValENDP & EP_CTR_TX) != 0)
  {
    ClrEPR_CTR_TX(ENDP0);
    CTR_IN0();
    return;
  }
  else if((wValENDP & EP_SETUP) != 0)
  {
    ClrEPR_CTR_RX(ENDP0);
    CTR_SETUP0();
    return;
```

```
    }
    else if((wValENDP & EP_CTR_RX) != 0)
    {
      ClrEPR_CTR_RX(ENDP0);
      CTR_OUT0();
      return;
    }
  }
}
// Other endpoints
else
{
  wValENDP = GetEPR(wEPIndex);

  SetEPR_RXStatus(wEPIndex, EP_RX_NAK);
  SetEPR_TXStatus(wEPIndex, EP_TX_NAK);

  if((wValENDP & EP_CTR_TX) != 0)
  {
    ClrEPR_CTR_TX(wEPIndex);
    switch(wEPIndex)
    {
    case ENDP1: CTR_IN1(); break;
    case ENDP2: CTR_IN2(); break;
    case ENDP3: CTR_IN3(); break;
    case ENDP4: CTR_IN4(); break;
```

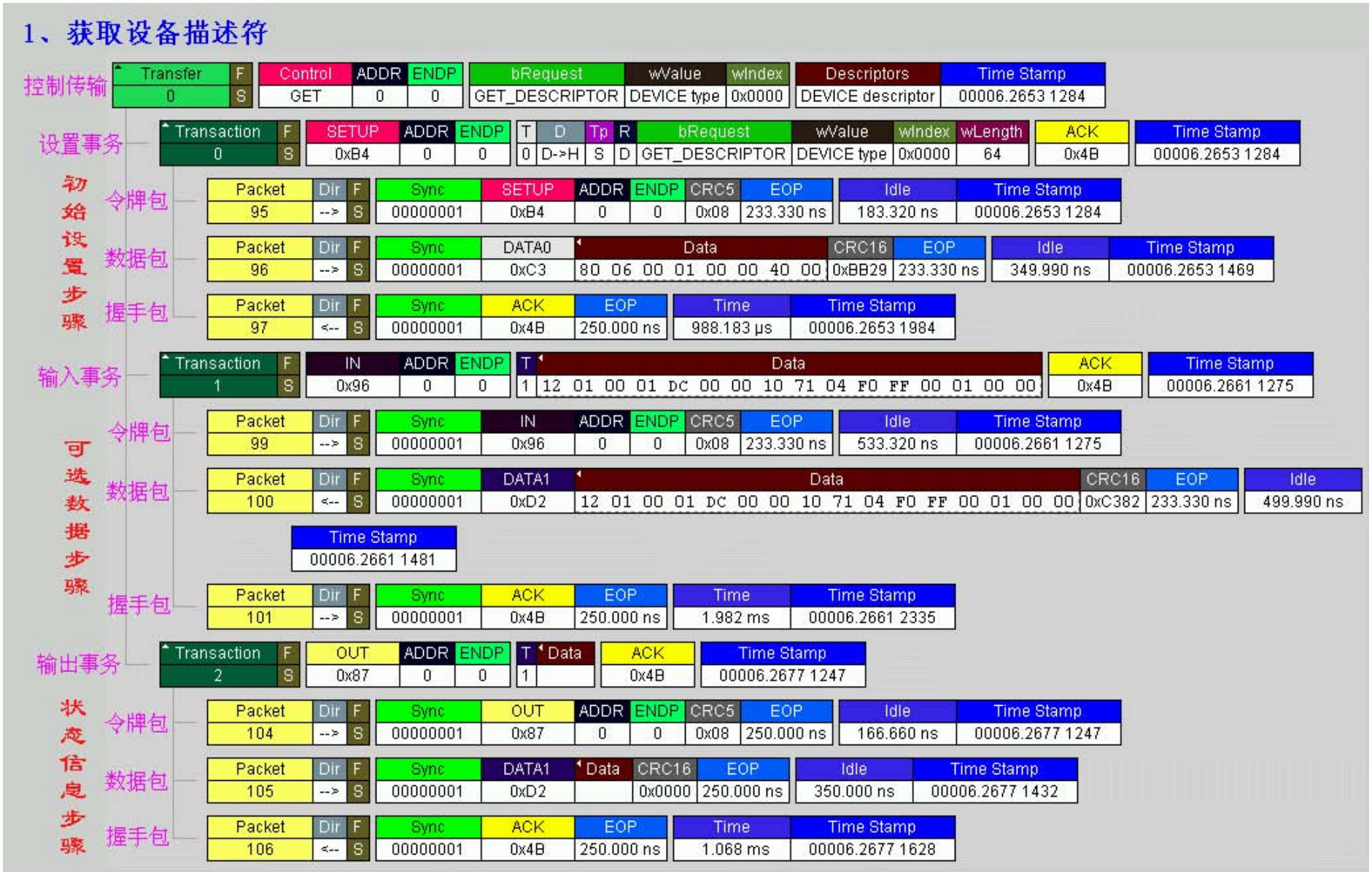
```
        case ENDP5: CTR_IN5(); break;
        case ENDP6: CTR_IN6(); break;
        case ENDP7: CTR_IN7(); break;
        default: break;
        }
      }

      if((wValENDP & EP_CTR_RX) != 0)
      {
        ClrEPR_CTR_RX(wEPIndex);
        switch(wEPIndex)
        {
        case ENDP1: CTR_OUT1(); break;
        case ENDP2: CTR_OUT2(); break;
        case ENDP3: CTR_OUT3(); break;
        case ENDP4: CTR_OUT4(); break;
        case ENDP5: CTR_OUT5(); break;
        case ENDP6: CTR_OUT6(); break;
        case ENDP7: CTR_OUT7(); break;
        default: break;
        }
      }
    }
  }
}
```

INT_ISTR_CTR()函数将各自响应事件提取出来，默认端点 EP0 也是最为复杂的，这个需要查看 STM32 的参考手册以及 USB 协议才能更好了解为何如此。到这里 STM32 USB 里数据传输事件就指向了各个对应的端点。

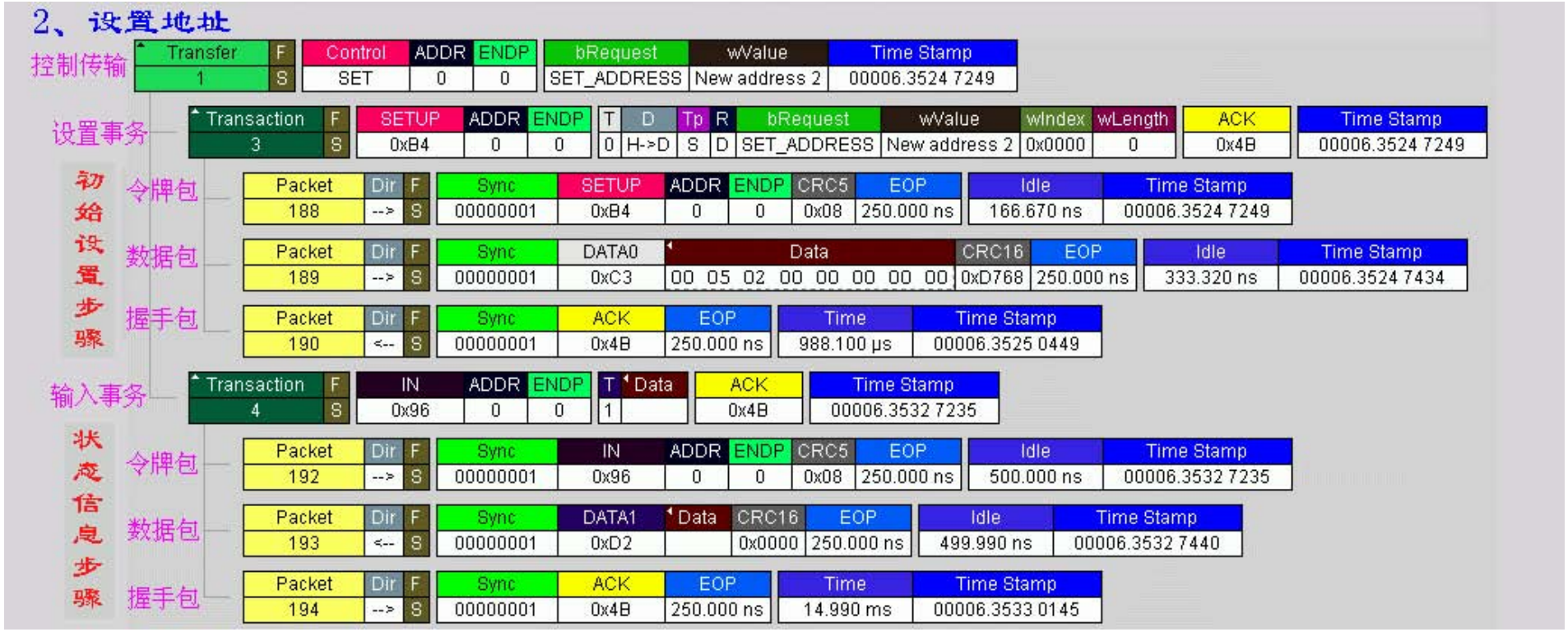
第四篇：USB 设备的枚举（上）

USB 设备能否工作，枚举步骤，用“乡村爱情”里的话说，“必须的！”，网上也有很多资料，圈圈就提供了一份详细的枚举过程，但对 STM32 是怎么响应的没有说明，一会详细道来，先贴上圈圈的提供的那个枚举图示（希望圈圈支持，如果不妥，请与我联系）：

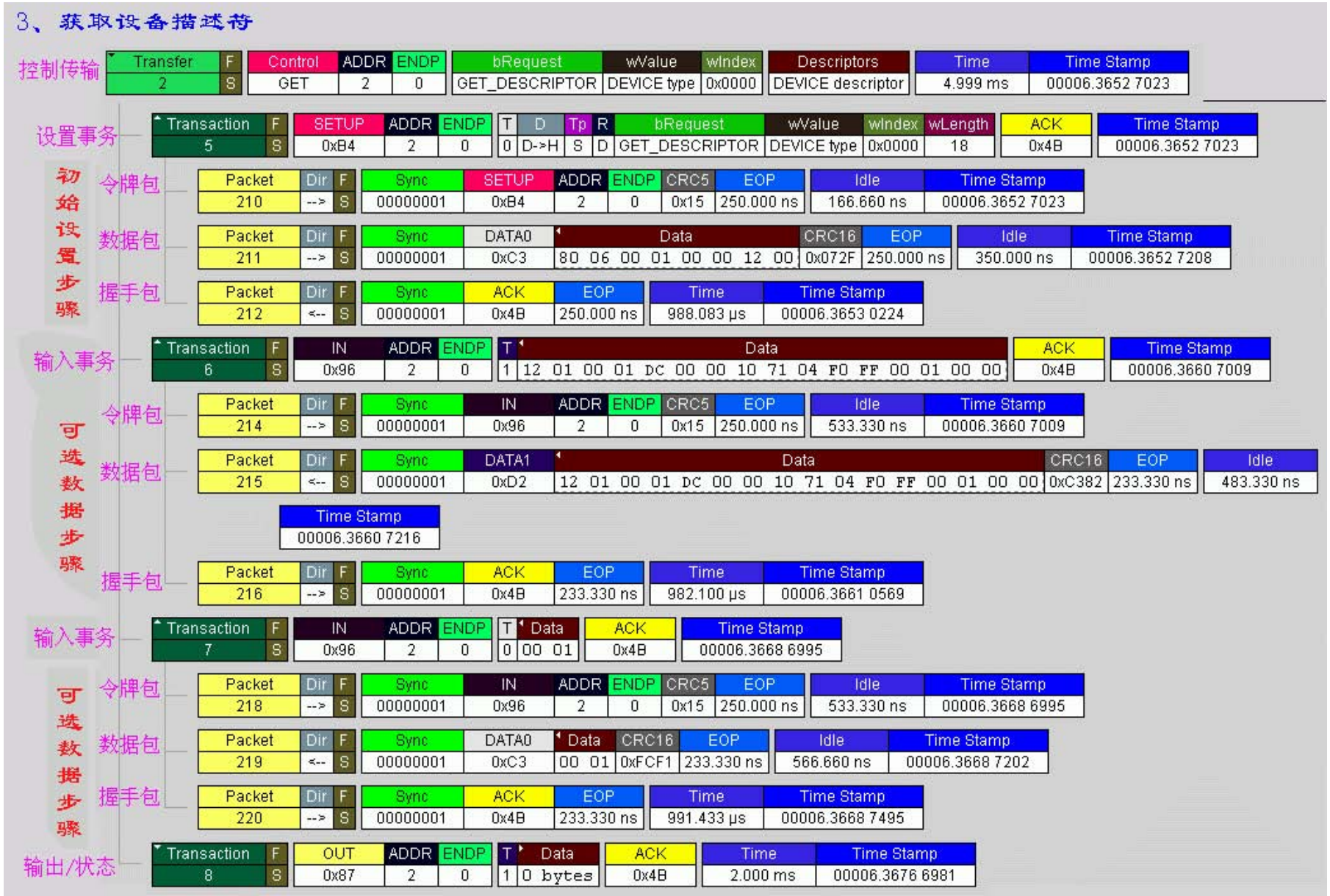
1、获取设备描述符


控制传输
Transfer F 0 S
Control GET
ADDR 0
ENDP 0
bRequest GET_DESCRIPTOR
wValue DEVICE type
wIndex 0x0000
Descriptors DEVICE descriptor
Time Stamp 00006.2653 1284
设置事务
Transaction F 0 S
SETUP 0xB4
ADDR 0
ENDP 0
T 0
D D->H
Tp S
R D
bRequest GET_DESCRIPTOR
wValue DEVICE type
wIndex 0x0000
wLength 64
ACK 0x4B
Time Stamp 00006.2653 1284
初始设置步骤
令牌包
Packet 95
Dir -->
F S
Sync 00000001
SETUP 0xB4
ADDR 0
ENDP 0
CRC5 0x08
EOP 233.330 ns
Idle 183.320 ns
Time Stamp 00006.2653 1284
数据包
Packet 96
Dir -->
F S
Sync 00000001
DATA0 0xC3
Data 80 06 00 01 00 00 40 00
CRC16 0xBB29
EOP 233.330 ns
Idle 349.990 ns
Time Stamp 00006.2653 1469
握手包
Packet 97
Dir <--
F S
Sync 00000001
ACK 0x4B
EOP 250.000 ns
Time 988.183 µs
Time Stamp 00006.2653 1984
输入事务
Transaction F 1 S
IN 0x96
ADDR 0
ENDP 0
T 1
Data 12 01 00 01 DC 00 00 10 71 04 F0 FF 00 01 00 00
ACK 0x4B
Time Stamp 00006.2661 1275
可选数据步骤
令牌包
Packet 99
Dir -->
F S
Sync 00000001
IN 0x96
ADDR 0
ENDP 0
CRC5 0x08
EOP 233.330 ns
Idle 533.320 ns
Time Stamp 00006.2661 1275
数据包
Packet 100
Dir <--
F S
Sync 00000001
DATA1 0xD2
Data 12 01 00 01 DC 00 00 10 71 04 F0 FF 00 01 00 00
CRC16 0xC382
EOP 233.330 ns
Idle 499.990 ns
Time Stamp 00006.2661 1481
握手包
Packet 101
Dir -->
F S
Sync 00000001
ACK 0x4B
EOP 250.000 ns
Time 1.982 ms
Time Stamp 00006.2661 2335
输出事务
Transaction F 2 S
OUT 0x87
ADDR 0
ENDP 0
T 1
Data
ACK 0x4B
Time Stamp 00006.2677 1247
状态信息步骤
令牌包
Packet 104
Dir -->
F S
Sync 00000001
OUT 0x87
ADDR 0
ENDP 0
CRC5 0x08
EOP 250.000 ns
Idle 166.660 ns
Time Stamp 00006.2677 1247
数据包
Packet 105
Dir -->
F S
Sync 00000001
DATA1 0xD2
Data
CRC16 0x0000
EOP 250.000 ns
Idle 350.000 ns
Time Stamp 00006.2677 1432
握手包
Packet 106
Dir <--
F S
Sync 00000001
ACK 0x4B
EOP 250.000 ns
Time 1.068 ms
Time Stamp 00006.2677 1628

2、设置地址


控制传输
Transfer | F | S
1
Control | ADDR | ENDP
SET | 0 | 0
bRequest | wValue | Time Stamp
SET_ADDRESS | New address 2 | 00006.3524 7249
设置事务
Transaction | F | S
3
SETUP | ADDR | ENDP
0xB4 | 0 | 0
T | D | Tp | R | bRequest | wValue | wIndex | wLength
0 | H->D | S | D | SET_ADDRESS | New address 2 | 0x0000 | 0
ACK
0x4B
Time Stamp
00006.3524 7249
初始设置步骤
令牌包
Packet | Dir | F | Sync | SETUP | ADDR | ENDP | CRC5 | EOP | Idle | Time Stamp
188 | --> | S | 00000001 | 0xB4 | 0 | 0 | 0x08 | 250.000 ns | 166.670 ns | 00006.3524 7249
数据包
Packet | Dir | F | Sync | DATA0 | Data | CRC16 | EOP | Idle | Time Stamp
189 | --> | S | 00000001 | 0xC3 | 00 05 02 00 00 00 00 00 | 0xD768 | 250.000 ns | 333.320 ns | 00006.3524 7434
握手包
Packet | Dir | F | Sync | ACK | EOP | Time | Time Stamp
190 | <-- | S | 00000001 | 0x4B | 250.000 ns | 988.100 µs | 00006.3525 0449
输入事务
Transaction | F | S
4
IN | ADDR | ENDP
0x96 | 0 | 0
T | Data
1
ACK
0x4B
Time Stamp
00006.3532 7235
状态信息步骤
令牌包
Packet | Dir | F | Sync | IN | ADDR | ENDP | CRC5 | EOP | Idle | Time Stamp
192 | --> | S | 00000001 | 0x96 | 0 | 0 | 0x08 | 250.000 ns | 500.000 ns | 00006.3532 7235
数据包
Packet | Dir | F | Sync | DATA1 | Data | CRC16 | EOP | Idle | Time Stamp
193 | <-- | S | 00000001 | 0xD2 | | 0x0000 | 250.000 ns | 499.990 ns | 00006.3532 7440
握手包
Packet | Dir | F | Sync | ACK | EOP | Time | Time Stamp
194 | --> | S | 00000001 | 0x4B | 250.000 ns | 14.990 ms | 00006.3533 0145

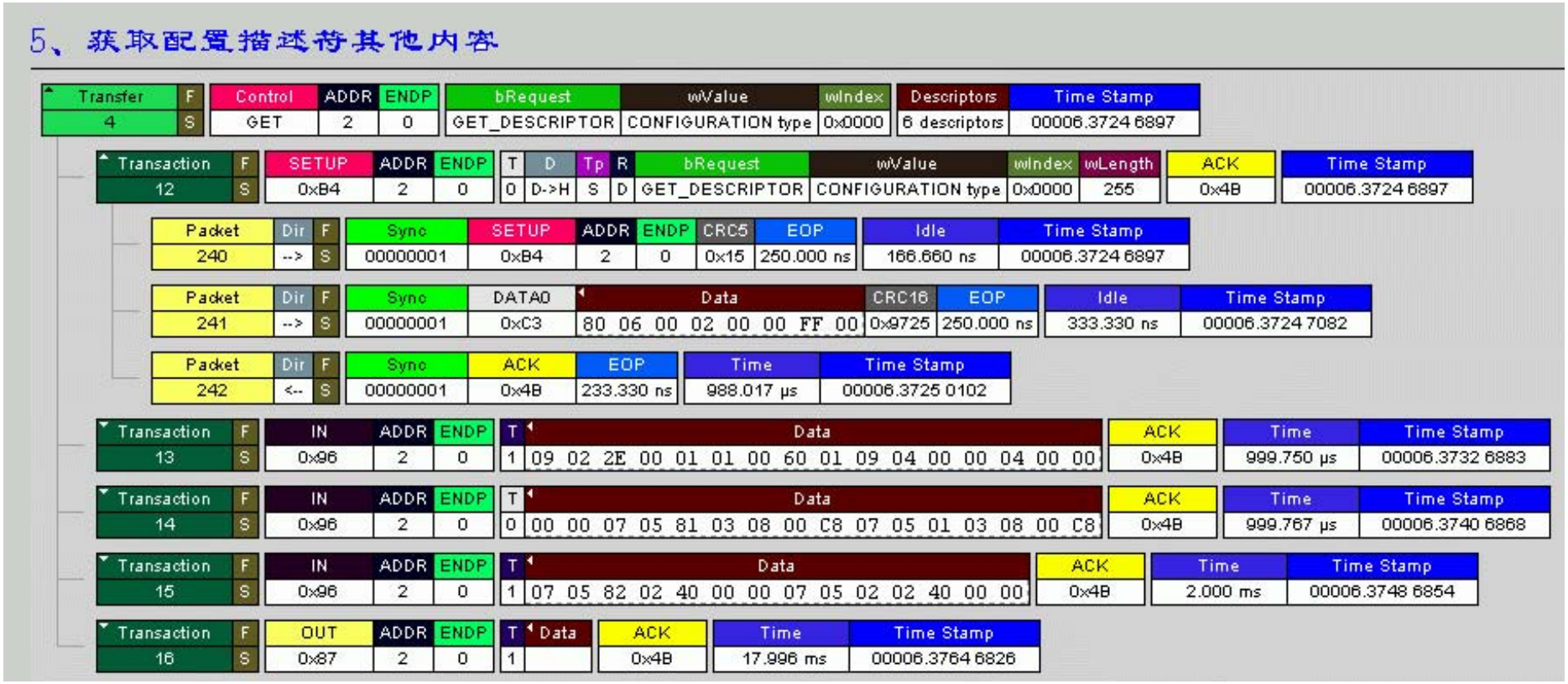
3、获取设备描述符


控制传输
Transfer F 2 S
Control ADDR ENDP GET 2 0
bRequest wValue wIndex GET_DESCRIPTOR DEVICE type 0x0000
Descriptors DEVICE descriptor
Time Time Stamp 4.999 ms 00006.3652 7023
设置事务
Transaction F 5 S
SETUP ADDR ENDP 0xB4 2 0
T D Tp R bRequest wValue wIndex wLength 0 D->H S D GET_DESCRIPTOR DEVICE type 0x0000 18
ACK 0x4B
Time Stamp 00006.3652 7023
初始设置步骤
令牌包
Packet Dir F 210 --> S
Sync SETUP ADDR ENDP CRC5 EOP 00000001 0xB4 2 0 0x15 250.000 ns
Idle Time Stamp 166.660 ns 00006.3652 7023
数据包
Packet Dir F 211 --> S
Sync DATA0 Data CRC16 EOP 00000001 0xC3 80 06 00 01 00 00 12 00 0x072F 250.000 ns
Idle Time Stamp 350.000 ns 00006.3652 7208
握手包
Packet Dir F 212 <-- S
Sync ACK EOP 00000001 0x4B 250.000 ns
Time Time Stamp 988.083 µs 00006.3653 0224
输入事务
Transaction F 6 S
IN ADDR ENDP 0x96 2 0
T Data 1 12 01 00 01 DC 00 00 10 71 04 F0 FF 00 01 00 00
ACK 0x4B
Time Stamp 00006.3660 7009
可选数据步骤
令牌包
Packet Dir F 214 --> S
Sync IN ADDR ENDP CRC5 EOP 00000001 0x96 2 0 0x15 250.000 ns
Idle Time Stamp 533.330 ns 00006.3660 7009
数据包
Packet Dir F 215 <-- S
Sync DATA1 Data 00000001 0xD2 12 01 00 01 DC 00 00 10 71 04 F0 FF 00 01 00 00
CRC16 EOP Idle 0xC382 233.330 ns 483.330 ns
Time Stamp 00006.3660 7216
握手包
Packet Dir F 216 --> S
Sync ACK EOP 00000001 0x4B 233.330 ns
Time Time Stamp 982.100 µs 00006.3661 0569
输入事务
Transaction F 7 S
IN ADDR ENDP 0x96 2 0
T Data 0 00 01
ACK 0x4B
Time Stamp 00006.3668 6995
可选数据步骤
令牌包
Packet Dir F 218 --> S
Sync IN ADDR ENDP CRC5 EOP 00000001 0x96 2 0 0x15 250.000 ns
Idle Time Stamp 533.330 ns 00006.3668 6995
数据包
Packet Dir F 219 <-- S
Sync DATA0 Data CRC16 EOP 00000001 0xC3 00 01 0xFCF1 233.330 ns
Idle Time Stamp 566.660 ns 00006.3668 7202
握手包
Packet Dir F 220 --> S
Sync ACK EOP 00000001 0x4B 233.330 ns
Time Time Stamp 991.433 µs 00006.3668 7495
输出/状态
Transaction F 8 S
OUT ADDR ENDP 0x87 2 0
T Data 1 0 bytes
ACK 0x4B
Time Time Stamp 2.000 ms 00006.3676 6981

4、获取配置描述符


控制传输
Transfer F
3 S
Control ADDR ENDP
GET 2 0
bRequest wValue wIndex
GET_DESCRIPTOR CONFIGURATION type 0x0000
Descriptors Time Stamp
CONFIGURATION descriptor 00006.3692 6953
设置事务
Transaction F
9 S
SETUP ADDR ENDP
0xB4 2 0
T D Tp R bRequest wValue wIndex wLength
0 D->H S D GET_DESCRIPTOR CONFIGURATION type 0x0000 9
ACK Time Stamp
0x4B 00006.3692 6953
初始设置步骤
令牌包
Packet Dir F
227 --> S
Sync SETUP ADDR ENDP CRC5 EOP
00000001 0xB4 2 0 0x15 233.330 ns
Idle Time Stamp
183.320 ns 00006.3692 6953
数据包
Packet Dir F
228 --> S
Sync DATA0 Data CRC16 EOP
00000001 0xC3 80 06 00 02 00 00 09 00 0x7520 233.330 ns
Idle Time Stamp
366.660 ns 00006.3692 7138
握手包
Packet Dir F
229 <-- S
Sync ACK EOP
__000001 0x4B 250.000 ns
Time Time Stamp
988.167 µs 00006.3693 0154
输入事务
Transaction F
10 S
IN ADDR ENDP
0x96 2 0
T Data
1 09 02 2E 00 01 01 00 60 01
ACK Time Stamp
0x4B 00006.3700 6944
可选数据步骤
令牌包
Packet Dir F
231 --> S
Sync IN ADDR ENDP CRC5 EOP
00000001 0x96 2 0 0x15 233.330 ns
Idle Time Stamp
550.000 ns 00006.3700 6944
数据包
Packet Dir F
232 <-- S
Sync DATA1 Data CRC16 EOP
00000001 0xD2 09 02 2E 00 01 01 00 60 01 0xA01E 233.330 ns
Idle Time Stamp
483.320 ns 00006.3700 7151
握手包
Packet Dir F
233 --> S
Sync ACK EOP
00000001 0x4B 233.330 ns
Time Time Stamp
986.933 µs 00006.3701 0214
输出事务
Transaction F
11 S
OUT ADDR ENDP
0x87 2 0
T Data
1
ACK Time Stamp
0x4B 00006.3708 6930
状态信息步骤
令牌包
Packet Dir F
235 --> S
Sync OUT ADDR ENDP CRC5 EOP
00000001 0x87 2 0 0x15 233.330 ns
Idle Time Stamp
183.320 ns 00006.3708 6930
数据包
Packet Dir F
236 --> S
Sync DATA1 Data CRC16 EOP
00000001 0xD2 0x0000 250.000 ns
Idle Time Stamp
350.000 ns 00006.3708 7115
握手包
Packet Dir F
237 <-- S
Sync ACK EOP
00000001 0x4B 233.330 ns
Time Time Stamp
1.993 ms 00006.3708 7311

5、获取配置描述符其他内容



Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time Stamp
4	S	GET	2	0	GET_DESCRIPTOR	CONFIGURATION type	0x0000	6 descriptors	00006.3724 6897

Transaction	F	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time Stamp
12	S	0xB4	2	0	0	D->H	S	D	GET_DESCRIPTOR	CONFIGURATION type	0x0000	255	0x4B	00006.3724 6897

Packet	Dir	F	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
240	-->	S	00000001	0xB4	2	0	0x15	250.000 ns	166.660 ns	00006.3724 6897

Packet	Dir	F	Sync	DATA0	Data	CRC16	EOP	Idle	Time Stamp
241	-->	S	00000001	0xC3	80 06 00 02 00 00 FF 00	0x9725	250.000 ns	333.330 ns	00006.3724 7082

Packet	Dir	F	Sync	ACK	EOP	Time	Time Stamp
242	<--	S	00000001	0x4B	233.330 ns	988.017 µs	00006.3725 0102

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
13	S	0x96	2	0	1	09 02 2E 00 01 01 00 60 01 09 04 00 00 04 00 00	0x4B	999.750 µs	00006.3732 6883

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
14	S	0x96	2	0	0	00 00 07 05 81 03 08 00 C8 07 05 01 03 08 00 C8	0x4B	999.767 µs	00006.3740 6868

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
15	S	0x96	2	0	1	07 05 82 02 40 00 00 07 05 02 02 40 00 00	0x4B	2.000 ms	00006.3748 6854

Transaction	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
16	S	0x87	2	0	1		0x4B	17.996 ms	00006.3764 6826

6、获取设备和配置描述符

Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time Stamp
5	S	GET	2	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE descriptor	00006.3908 6573

Transaction	F	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time Stamp
17	S	0xB4	2	0	0	D->H	S	D	GET_DESCRIPTOR	DEVICE type	0x0000	18	0x4B	00006.3908 6573

Packet	Dir	F	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
278	-->	S	00000001	0xB4	2	0	0x15	250.000 ns	166.660 ns	00006.3908 6573

Packet	Dir	F	Sync	DATA0	Data	CRC16	EOP	Idle	Time Stamp
279	-->	S	00000001	0xC3	80 06 00 01 00 00 12 00	0x072F	250.000 ns	350.000 ns	00006.3908 6758

Packet	Dir	F	Sync	ACK	EOP	Time	Time Stamp
280	<--	S	00000001	0x4B	233.330 ns	988.067 µs	00006.3908 7274

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
18	S	0x96	2	0	1	12 01 00 01 DC 00 00 10 71 04 F0 FF 00 01 00 00	0x4B	999.767 µs	00006.3916 6558

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
19	S	0x96	2	0	0	00 01	0x4B	999.767 µs	00006.3924 6544

Transaction	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
20	S	0x87	2	0	1		0x4B	2.000 ms	00006.3932 6530

Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time Stamp
6	S	GET	2	0	GET_DESCRIPTOR	CONFIGURATION type	0x0000	6 descriptors	00006.3948 6502

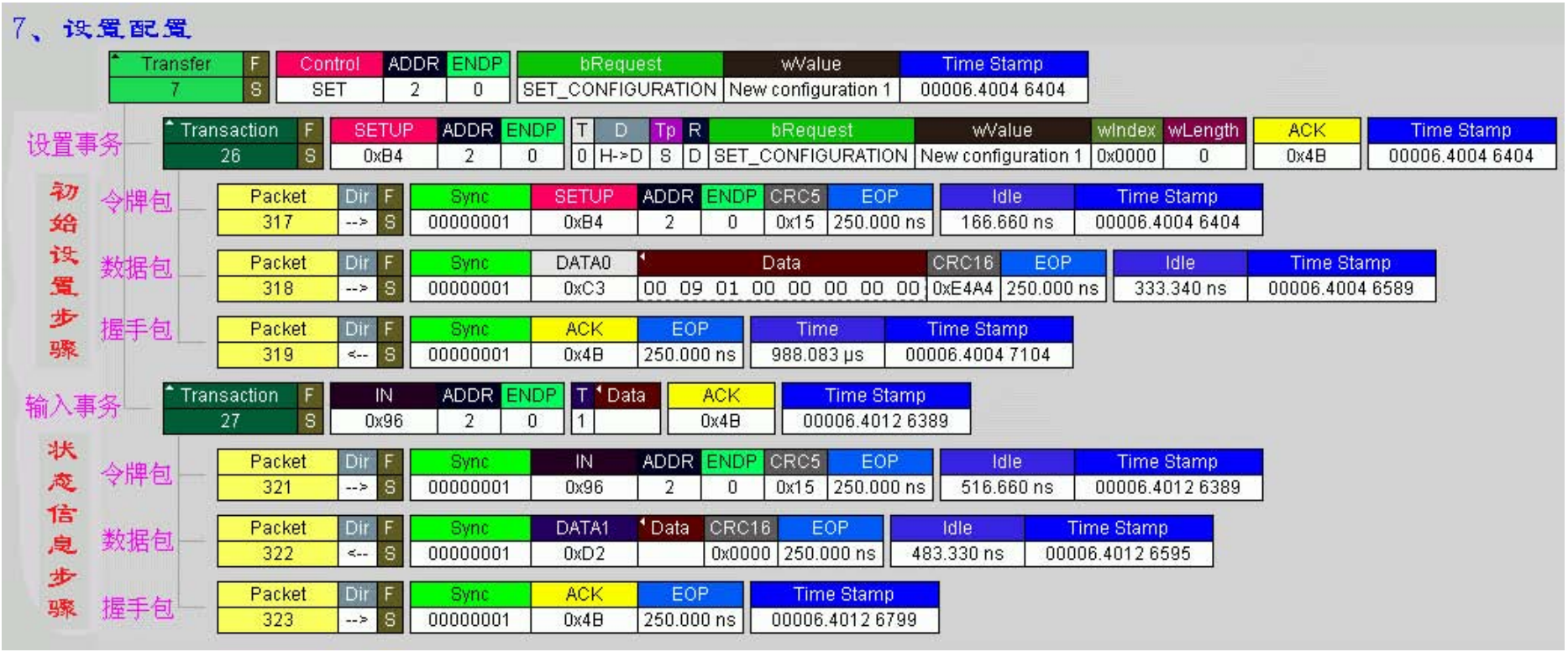
Transaction	F	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time	Time Stamp
21	S	0xB4	2	0	0	D->H	S	D	GET_DESCRIPTOR	CONFIGURATION type	0x0000	137	0x4B	999.767 µs	00006.3948 6502

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
22	S	0x96	2	0	1	09 02 2E 00 01 01 00 60 01 09 04 00 00 04 00 00	0x4B	999.850 µs	00006.3956 6488

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
23	S	0x96	2	0	0	00 00 07 05 81 03 08 00 C8 07 05 01 03 08 00 C8	0x4B	999.683 µs	00006.3964 6479

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
24	S	0x96	2	0	1	07 05 82 02 40 00 00 07 05 02 02 40 00 00	0x4B	2.000 ms	00006.3972 6460

Transaction	F	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
25	S	0x87	2	0	1		0x4B	2.000 ms	00006.3988 6432



说明枚举过程之前，首先说明一个变量，定义在 usb_core.c 中：

```
volatile DEVICE_INFO vsDeviceInfo;
```

看意思就知道他的作用，DEVICE_INFO 是个结构，定义在 usb_type.h 中：

```
// ***************************************************************************
```

```
// DEVICE_INFO
// *****************************************************************************
typedef struct _DEVICE_INFO
{
  unsigned char bDeviceAddress;

  unsigned char bCurrentFeature;
  unsigned char bCurrentConfiguration;
  unsigned char bCurrentInterface;
  unsigned char bCurrentAlternateSetting;

  WORD_2BYTE    uStatusInfo;

  DEVICE_STATE  eDeviceState;
  RESUME_STATE  eResumeState;
  CONTROL_STATE eControlState;

  SETUP_DATA    SetupData;

  TRANSFER_INFO TransInfo;
}
DEVICE_INFO,
*PDEVICE_INFO;
```

在 枚举过程中，就是如何处理好 SETUP 事件，如果 STM32 USB 接收到正确的 SETUP 事件，将响应函数 CTR_SETUP0()，SETUP 事件是特殊的 OUT 事件，数据方向 Host->Device，SETUP 事件数据长度固定为 8，数据定义在 DEVICE_INFO.SetupData，其数据结构是(定义在

usb_type.h 中)：

```
typedef struct _SETUP_DATA
{
  unsigned char bmRequestType;            // request type
  unsigned char bRequest;                 // request code

  WORD_2BYTE wValue;
  WORD_2BYTE wIndex;
  WORD_2BYTE wLength;
}
SETUP_DATA,
*PSETUP_DATA;
```

WORD_2BYTE 是定义的一个共用体：

```
typedef union _WORD_2BYTE
{
  unsigned short w;
  struct
  {
    unsigned char LSB;
    unsigned char MSB;
  }b;
}
WORD_2BYTE;
```

为什么将 SETUP 数据结构中的 wValue,wIndex,wLength 如此定义？

1：USB 协议中所有数据传输都是依照低位在先的原则
2：高地位字节可能功能复用
这样在后续的程序编写中就变得十分方便，ST 提供的 USB 固件方法同样如此，但这方面的处理让人有些摸不着头脑，详情可参阅。至于具体的 SETUP 数据结构含义如何，还是要具备基本知识：了解 USB 协议

CTR_SETUP0() 函数将 SETUP 数据提取出来，SETUP 数据结构有 0 长度和非 0 长度的数据结构，详细参阅 USB2.0 官方协议第 9 章。在这将两种区别开来分别执行 SETUP0_NoData()和 SETUP0_Data()函数，并返回结果，根据返回结果再响应 USB 主机

```
// *****************************************************************************
// Function Name  : CTR_SETUP0
// Description    :
// Input          :
// Output         :
// Return         :
// *****************************************************************************
void CTR_SETUP0(void)
{
  RESULT eResult;

  BufferCopy_PMAToUser( (unsigned char *)&vsDeviceInfo.SetupData,
                        GetBuffDescTable_RXAddr(ENDP0),
                        GetBuffDescTable_RXCount(ENDP0));

  if(vsDeviceInfo.SetupData.wLength.w == 0)
  {
    eResult = SETUP0_NoData();
  }
```

```
  else
  {
    eResult = SETUP0_Data();
  }

  switch(eResult)
  {
  case RESULT_SUCCESS:

    break;

  case RESULT_LASTDATA:

    break;

  case RESULT_ERROR:
  case RESULT_UNSUPPORT:
    SetEPR_RXStatus(ENDP0, EP_RX_VALID);
    SetEPR_TXStatus(ENDP0, EP_TX_STALL);
    break;
  }
}
```

SETUP0_Data() 和 SETUP0_NoData()函数支持的所有 USB 请求类型只有罗列的这些，有多少种组合都定义在 USB 协议中，程序根据请求代码，再去执行对应函 数，这样做的目的就是让程序结构明了。其中注释为"// done"的部分表明此部分功能已完成。对于未完成部分，希望大家在交流中完善。

```
// *******************************************************************************
```

```
// Routine Groups: SETUP_Data
// ***************************************************************************
RESULT SETUP0_Data(void)
{
  // SetupData.bRequest: request code
  switch(vsDeviceInfo.SetupData.bRequest)
  {
  case SR_GET_STATUS:         return SR_GetStatus();                  // done
  case SR_GET_DESCRIPTOR:     return SR_GetDescriptor();              // done
  case SR_SET_DESCRIPTOR:     return SR_SetDescriptor();              // unsupport
  case SR_GET_CONFIGURATION:  return SR_GetConfiguration();           // done
  case SR_GET_INTERFACE:      return SR_GetInterface();               // unsupport
  case SR_SYNCH_FRAME:        return SR_SynchFrame();                 // unsupport

  default: return RESULT_UNSUPPORT;
  }
}

// ***************************************************************************
// Routine Groups: SETUP_NoData
// ***************************************************************************
RESULT SETUP0_NoData(void)
{
  // SetupData.bRequest: request code
  switch(vsDeviceInfo.SetupData.bRequest)
  {
```

```
  case SR_CLEAR_FEATURE:      return SR_ClearFeature();        // unsupport
  case SR_SET_FEATURE:        return SR_SetFeature();          // unsupport
  case SR_SET_ADDRESS:        return SR_SetAddress();          // done
  case SR_SET_CONFIGURATION:  return SR_SetConfiguration();    // done
  case SR_SET_INTERFACE:      return SR_SetInterface();        // unsupport

  default: return RESULT_UNSUPPORT;
  }
}
```

第五篇：USB 设备的枚举（下）

SETUP 事件正确接收后，根据该事件提供的请求类型进行对主机的响应。SETUP 数据结构的 wLength 字段说明的是请求返回或者提供的数据长度。

如果判断出的请求信息错误或者说不被支持，STM32 USB 设备需要中断此次请求：

```
    SetEPR_RXStatus(ENDP0, EP_RX_VALID);
    SetEPR_TXStatus(ENDP0, EP_TX_STALL);
```

正确获取到请求信息后，如果 wLength 为 0，设备需要发送一个 0 长度数据包以响应主机：

```
// *****************************************************************************
// Function Name  : SETUP0_Trans0Data
// Description    :
// Input          :
// Output         :
```

```
// Return         :
// *****************************************************************************
RESULT SETUP0_Trans0Data(void)
{
  // Send 0-length data frame as ACK to host
  SetBuffDescTable_TXCount(ENDP0, 0);
  SetEPR_RXStatus(ENDP0, EP_RX_NAK);
  SetEPR_TXStatus(ENDP0, EP_TX_VALID);

  return RESULT_SUCCESS;
}
```

如果 wLength 不为 0，设备则需要根据请求的数据长度发送数据包以响应主机：

```
// *****************************************************************************
// Function Name  : SETUP0_TransData
// Description    :
// Input          :
// Output         :
// Return         :
// *****************************************************************************
RESULT SETUP0_TransData(void)
{
  unsigned short wLength = vsDeviceInfo.TransInfo.wLength;
  unsigned short wOffset = vsDeviceInfo.TransInfo.wOffset;
  unsigned short wMaxSize = vsDeviceInfo.TransInfo.wPacketSize;

  if(wLength)
```

```
  {
    if(wLength > wMaxSize)
    {
      wLength = wMaxSize;
    }

    // Copy the transfer buffer to the endpoint0's buffer
    BufferCopy_UserToPMA( vsDeviceInfo.TransInfo.pBuffer+wOffset,   // transfer buffer
                          GetBuffDescTable_TXAddr(ENDP0),           // endpoint 0 TX address
                          wLength);

    SetBuffDescTable_TXCount(ENDP0, wLength);
    SetEPR_RXStatus(ENDP0, EP_RX_NAK);
    SetEPR_TXStatus(ENDP0, EP_TX_VALID);

    // Update the data lengths
    vsDeviceInfo.TransInfo.wLength -= wLength;
    vsDeviceInfo.TransInfo.wOffset += wLength;

    return RESULT_LASTDATA;
  }

  return RESULT_SUCCESS;
}
```

如果发送的数据长度大于端点设置的最大数据包长度，数据将分割为若干次发送，记录发送数据的状态包含在结构体 TRANSFER_INFO 中：

```
// ********************************************************************************
// TRANSFER_INFO
// ********************************************************************************
typedef struct _TRANSFER_INFO
{
  unsigned short wLength;                 // total lengths data will be transmit
  unsigned short wOffset;                 // number of data be transmited
  unsigned short wPacketSize;             // endpoints packet max size
  unsigned char* pBuffer;                 // address of data buffer
}
TRANSFER_INFO,
*PTRANSFER_INFO;
```

TRANSFER_INFO.wLength 记录发送的数据长度，如果非 0，表示有数据需要被发送。
TRANSFER_INFO.wOffset 记录已发送的数据长度，用以确定数据缓冲 TRANSFER_INFO.pBuffer 的偏移量。

需 要了解的一点：USB 主机向 USB 设备正确发送一请求后（这部分的处理由硬件完成），USB 主机将间隔若干次的向 USB 设备索取响应数据，STM32 USB TX 状态为 NAK 说明不响应 USB 主机，USB 主机在超时后退出此次请求；TX 状态为 STLL 说明中断此次请求，USB 主机将无条件退出请求；TX 状态为 VALID 说明设备已准备好数据发送，USB 主机将从 USB 设备读取数据。
以非 0 长度数据请求的 GET_DESCRIPTOR 请求为例的响应过程：
CTR_SETUP0()->SETUP0_Data()->SR_GetDescriptor()->SETUP0_TransData()

```
RESULT SR_GetDescriptor(void)
{
  // RequestType: device->host, standard request and device recipient
  if(vsDeviceInfo.SetupData.bmRequestType == RT_D2H_STANDARD_DEVICE)
  {
```

```
    // SetupData.wValue.b.MSB: descriptor type
    // SetupData.wValue.b.LSB: descriptor index
    switch(vsDeviceInfo.SetupData.wValue.b.MSB)
    {
    case DESCRIPTOR_DEVICE:               return SR_GetDescriptor_Device();
    case DESCRIPTOR_CONFIG:               return SR_GetDescriptor_Config();
    case DESCRIPTOR_STRING:               return SR_GetDescriptor_String();

    default:  return RESULT_UNSUPPORT;
    }
  }

  return RESULT_UNSUPPORT;
}
```

GET_DESCRIPTOR 请求属于 USB 协议中的标准请求（standard request）并且数据方向为设备至主机（device->host），分设备描述符、配置描述符、字符串描述符三种。已设备描述符为例：

```
RESULT SR_GetDescriptor_Device(void)
{
  // Assigned the device descriptor to the transfer
  vsDeviceInfo.TransInfo.wOffset = 0;
  vsDeviceInfo.TransInfo.wPacketSize = ENDP0_PACKETSIZE;
  vsDeviceInfo.TransInfo.pBuffer = DescBuffer_Device.pBuff;
  vsDeviceInfo.TransInfo.wLength = DescBuffer_Device.wLen;
  vsDeviceInfo.eControlState = CS_GET_DESCRIPTOR;

  if(vsDeviceInfo.TransInfo.wLength > vsDeviceInfo.SetupData.wLength.w)
```

```
  {
    vsDeviceInfo.TransInfo.wLength = vsDeviceInfo.SetupData.wLength.w;
  }

  return SETUP0_TransData();
}
```

这里说明了发送数据的长度、缓冲、偏移、端点包大小以及当前的控制状态，并说明了如果发送的数据长度超出请求的数据长度，则将舍弃超出的部分。数据配置好后，调用 SETUP0_TransData()进行数据发送。

在 USB 主机查询到 USB 设备准备就绪后，将读取出这些数据，完成后，USB 设备将产生 IN 事件，此时将响应 CTR_IN0()函数：

```
// *****************************************************************************
// Function Name  : CTR_IN
// Description    :
// Input          :
// Output         :
// Return         :
// *****************************************************************************
void CTR_IN0(void)
{
  switch(vsDeviceInfo.eControlState)
  {
  case CS_GET_DESCRIPTOR:
    if(SETUP0_TransData() == RESULT_SUCCESS)
    {
      SetEPR_TXStatus(ENDP0, EP_TX_NAK);
      SetEPR_RXStatus(ENDP0, EP_RX_VALID);
    }
```

```
    break;

  case CS_SET_ADDRESS:
    SetEPR_TXStatus(ENDP0, EP_TX_NAK);
    SetEPR_RXStatus(ENDP0, EP_RX_VALID);

    SetDADDR(0x0080 | vsDeviceInfo.bDeviceAddress);
    vsDeviceInfo.eDeviceState = DS_ADDRESSED;
    break;

  case CS_SET_CONFIGURATION:
    SetEPR_TXStatus(ENDP0, EP_TX_NAK);
    SetEPR_RXStatus(ENDP0, EP_RX_VALID);

    vsDeviceInfo.eDeviceState = DS_CONFIGURED;
    break;

  default:
    break;
  }
}
```

再 这如果响应GET_DESCRIPTOR请求发送的数据如果全部发送完毕,SETUP0_TransData()返回RESULT_SUCCESS,并设 置TX状态为NAK;否则返回RESULT_LASTDATA，将继续发送剩余的数据直到数据全部被发送。至此，整个的GET_DESCRIPTOR请求 过程完成。

0长度的数据请求在发送0长度数据响应后，因为不存在可能还未传送的数据，因而IN事件后直接结束此次请求。

在数据方向为USB主机->USB设备时，如果正确接收到数据，将响应CTR_OUT0()函数，处理过程类同CTR_IN0()函数。

在USB设备的枚举过程中，USB的一些描述符数据结构需要了解，具体在USB协议中有详细的说明，在usb_desc(.c/.h)文件中，定义

了这些结构，这些结构是特定的：

设 备描述符：长度、格式固定，其中 VENDOR_ID 与 PRODUCT_ID 决定上位机驱动的识别。设备分属类别决定了设备的性质，如果为自定义 USB 设 备，设备分属类别值为 0，同时上位机驱动必须配合编写；如果为标准 USB 设备，则必须使用这些标准设备的驱动、数据结构等等，条件是你必须了解这些标准设 备的一些信息，好处是省去一些麻烦的驱动编写。

```
const unsigned char cbDescriptor_Device[DESC_SIZE_DEVICE] =
{
  DESC_SIZE_DEVICE,      // bLength: 18
  DESCRIPTOR_DEVICE,     // descriptor type

  0x00,              // bcdUSB LSB: USB release number -> USB2.0
  0x02,        // bcdUSB MSB: USB release number -> USB2.0

  0x00,          // bDeviceClass:    Class information in the interface descriptors
  0x00,        // bDeviceSubClass:
  0x00,        // bDeviceProtocol:
  0x40,        // bMaxPacketSize0:  LowS(8), FullS(8,16,32,64), HighS(64)

  LOWORD(VENDOR_ID),        // idVendor LSB:
  HIWORD(VENDOR_ID),        // idVendor MSB:

  LOWORD(PRODUCT_ID),       // idProduct LSB:
  HIWORD(PRODUCT_ID),       // idProduct MSB:

  LOWORD(DEVICE_VERSION),   // bcdDevice LSB:
  HIWORD(DEVICE_VERSION),   // bcdDevice MSB:
```

```
  0x01,        // iManufacturer: Index of string descriptor describing manufacturer
  0x02,        // iProduct: Index of string descriptor describing product
  0x03,        // iSerialNumber: Index of string descriptor describing the device serial number

  0x01         // bNumConfigurations: number of configurations
};
```

配 置描述符：前 9 个字节格式固定，后面紧跟的各种描述结构跟实际配置有关，每增加一种描述结构，该描述结构的第一字节说明了结构的长度，第二直接说明了结构 的类型。在配置描述符中一般包含配置描述、接口描述、端点描述，如果需要同样可增加自定义的描述。使用标准 USB 设备类别时，配置描述符的结构也必须满足 此类标准设备的数据结构。

```
const unsigned char cbDescriptor_Config[DESC_SIZE_CONFIG] =
{
  // Descriptor of configuration
  0x09,                    // lengths
  DESCRIPTOR_CONFIG,       // descriptor type

  DESC_SIZE_CONFIG,        // Total configuration descriptor lengths LSB
  0x00,                    // Total configuration descriptor lengths MSB

  0x01,        // bNumInterfaces: Total number of interfaces
  0x01,        // bConfigurationValue: Configuration value
  0x00,        // iConfiguration: Index of string descriptor describing the configuration

  0xA0,        // bmAttributes: bus powered
                        // bit 4...0 : Reserved, set to 0
```

```
                        // bit 5     : Remote wakeup (1:yes)
                        // bit 6     : Self power (1:yes)
                        // bit 7     : Reserved, set to 1

0x32,       // bMaxPower: this current is used for detecting Vbus = 100mA


// Descriptor of interface
0x09,
DESCRIPTOR_INTERFACE,

0x00,       // bInterfaceNumber: Number of Interface
0x00,       // bAlternateSetting: Alternate setting

0x02,       // bNumEndpoints: Number of endpoints except EP0
0x00,       // bInterfaceClass:
0x00,       // bInterfaceSubClass:
0x00,       // nInterfaceProtocol:

0x00,       // iInterface: Index of string descriptor describing the interface


// Descriptor of endpoint1 OUT
0x07,
DESCRIPTOR_ENDPOINT,
```

```
0x01,        // bEndpointAddress
             // bit 3...0 : the endpoint number
             // bit 6...4 : reserved
             // bit 7     : 0(OUT), 1(IN)

0x03,        // bmAttributes
             // bit 1...0 : Transfer type
             //                 00(CONTROL), 01(ISOCHRONOUS), 10(BULK), 11(INTERRUPT)
             // bit 3...2 : Synchronization type
             //                 00(No Synch), 01(Asynchronous), 10(Adaptive), 11(Synchronous)
             // bit 5...4 : Endpoint Usage type
             //                 00(data), 01(Feedback), 10(Implicit feedback data endpoint), 11(Reserved)
             // bit 7...6 : Reserved, must be zero

0x40,        // packet size LSB
0x00,        // packet size MSB

0x20,         // polling interval time: 32ms

// Descriptor of endpoint2 IN
0x07,
DESCRIPTOR_ENDPOINT,

0x82,        // bEndpointAddress
             // bit 3...0 : the endpoint number
             // bit 6...4 : reserved
```

```
                // bit 7     : 0(OUT), 1(IN)

  0x03,         // bmAttributes
                // bit 1...0 : Transfer type
                //                00(CONTROL), 01(ISOCHRONOUS), 10(BULK), 11(INTERRUPT)
                // bit 3...2 : Synchronization type
                //                00(No Synch), 01(Asynchronous), 10(Adaptive), 11(Synchronous)
                // bit 5...4 : Endpoint Usage type
                //                00(data), 01(Feedback), 10(Implicit feedback data endpoint), 11(Reserved)
                // bit 7...6 : Reserved, must be zero

  0x40,         // packet size LSB
  0x00,         // packet size MSB

  0x20          // polling interval time: 32ms
};
```

字符串描述符：定义了与设备有关的一些信息，常见的为以下四种，如果有需要，同样可以定义自己的字符串描述符。

```
const unsigned char cbDescriptor_StringLangID[DESC_SIZE_STRING_LANGID] =
{
  DESC_SIZE_STRING_LANGID,  // bLength
  DESCRIPTOR_STRING,        // bDescriptorType = String Descriptor

  0x09,                                  // LangID LSB:
  0x04                                  // LangID MSB: 0x0409(U.S. English)
};
```

```
const unsigned char cbDescriptor_StringVendor[DESC_SIZE_STRING_VENDOR] =
{
  DESC_SIZE_STRING_VENDOR,  // bLength
  DESCRIPTOR_STRING,        // bDescriptorType = String Descriptor

  // String: "LaBiXiaoXiaoXin"
  'L',0, 'a',0, 'B',0, 'i',0, 'X',0, 'i',0, 'a',0, 'o',0,
   'X',0, 'i',0, 'a',0, 'o',0, 'X',0, 'i',0, 'n',0
};

const unsigned char cbDescriptor_StringProduct[DESC_SIZE_STRING_PRODUCT] =
{
  DESC_SIZE_STRING_PRODUCT, // bLength
  DESCRIPTOR_STRING,        // bDescriptorType = String Descriptor

    // String: "STM32 ezUSB-CORE V1.01"
  'S',0, 'T',0, 'M',0, '3',0, '2',0, ' ',0, 'e',0, 'z',0, 'U',0, 'S',0, 'B',0,
  '-',0, 'C',0, 'O',0, 'R',0, 'E',0, ' ',0, 'V',0, '1',0, '.',0, '0',0, '1',0
};

const unsigned char cbDescriptor_StringSerial[DESC_SIZE_STRING_SERIAL] =
{
  DESC_SIZE_STRING_SERIAL,  // bLength
  DESCRIPTOR_STRING,        // bDescriptorType = String Descriptor
```

```
    // String: "ezUSB-CORE Demo 2008/11/18"
  'e',0, 'z',0, 'U',0, 'S',0, 'B',0, '-',0, 'C',0, 'O',0, 'R',0, 'E',0, ' ',0,
  'D',0, 'e',0, 'm',0, 'o',0, ' ',0, '2',0, '0',0, '0',0, '8',0, '/',0, '1',0, '1',0, '/',0, '1',0, '8',0
};
```

了 解这些描述符的用法以及作用，最好的方法的是编写自定义的 USB 上位机驱动以及应用程序，这样你可以深刻了解 USB 设备与主机间的数据交换方式以及实现手段。

第六篇：XP 下 USB 驱动开发的初步准备工作

必须的先决条件：

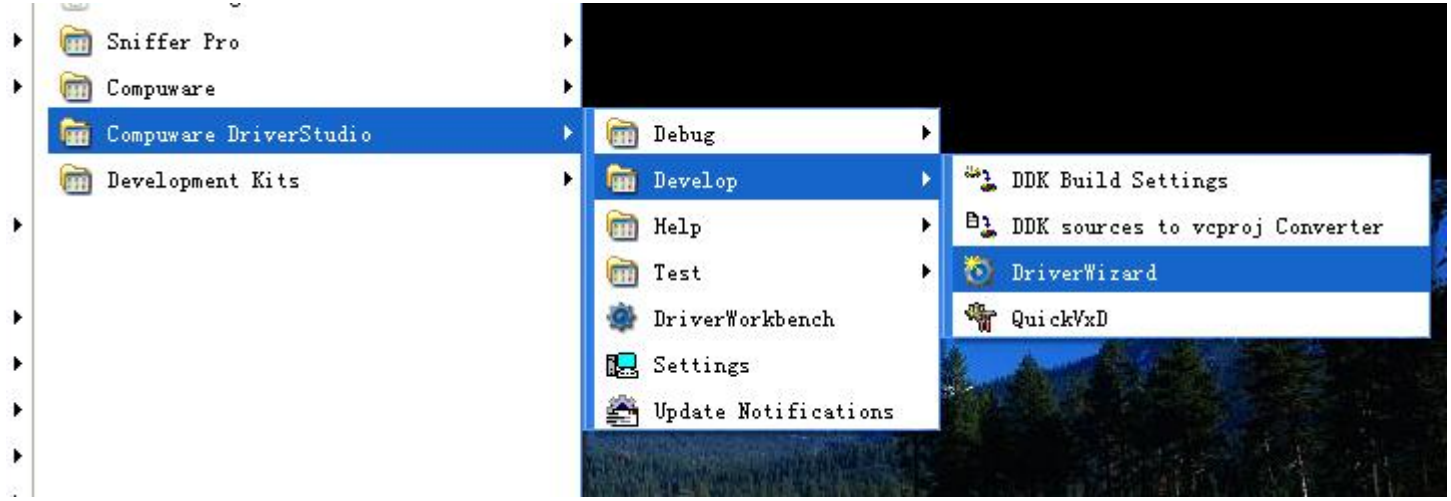
1：XP DDK（Driver Development Kits），可从 MS 网站下载。（Windows 2000 下请使用 Windows 2000 DDK），具备后安装 DDK，如果你觉得只需要利用 DDK 就可以开发驱动，那么接下来的内容完全可以不看，在这讨论的是利用 DriverStudio 的 DriverWinziard 生成的驱动框架。因为纯粹利用 DDK 开发驱动将是项十分艰巨的工作，需要你了解太多的系统知识，开发全部基于 C 语言，而且底 层驱动处理稍微不当，就容易让你系统直接挂了（WINDOWS 著名的蓝屏）。DDK 提供了一些驱动代码，有兴趣的朋友可以参考参考。

2：Compuware DriverStudio V3.2：Compuware 出品，是进行驱动开发最常见的平台，他封装了大部分设备驱动所必须的基本框架，以 C++形式生成 VC6.0 或者 VS2002、2003、2005 工程，用户一般只需要对该工程进行一些修改就可以完成最终目的。

3：代码开发环境 VC6.0，这个大家都知道

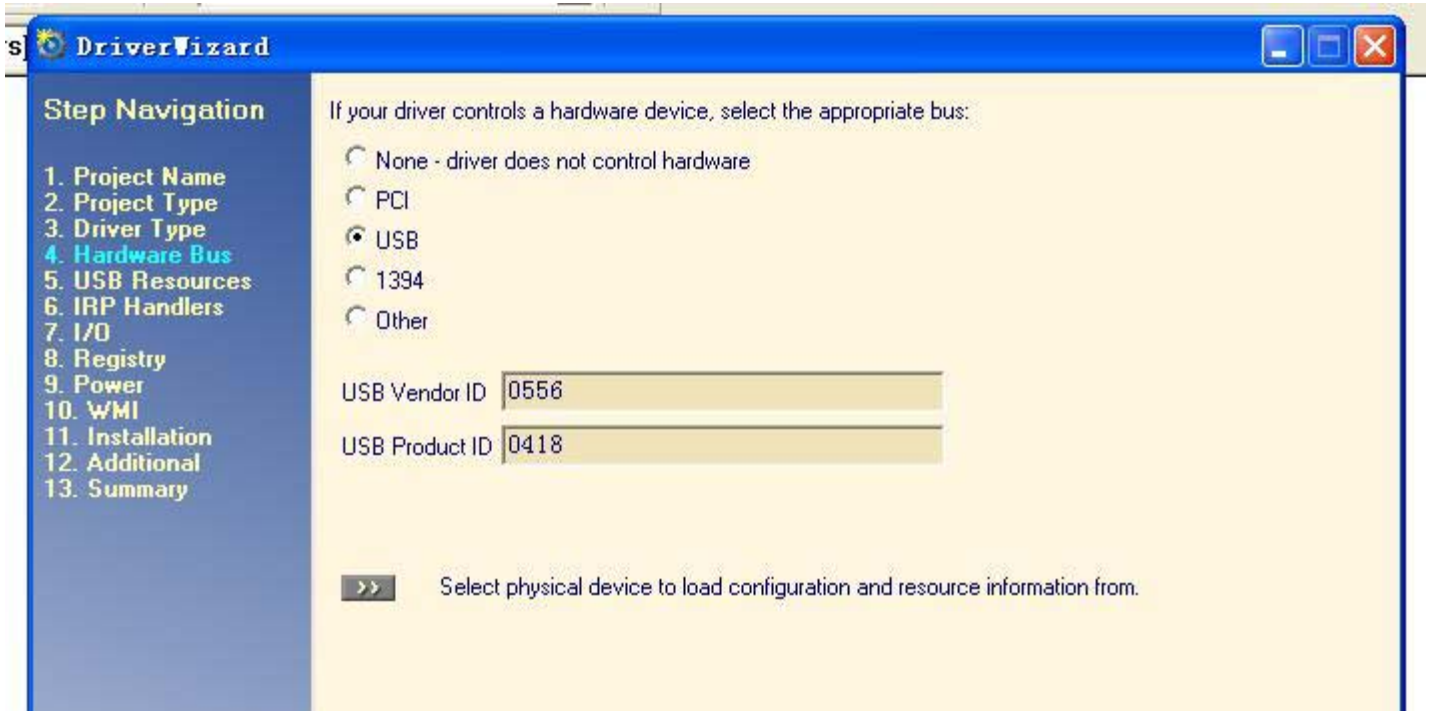
安装了 Windows XP DDK：

安装了 DriverStudio：

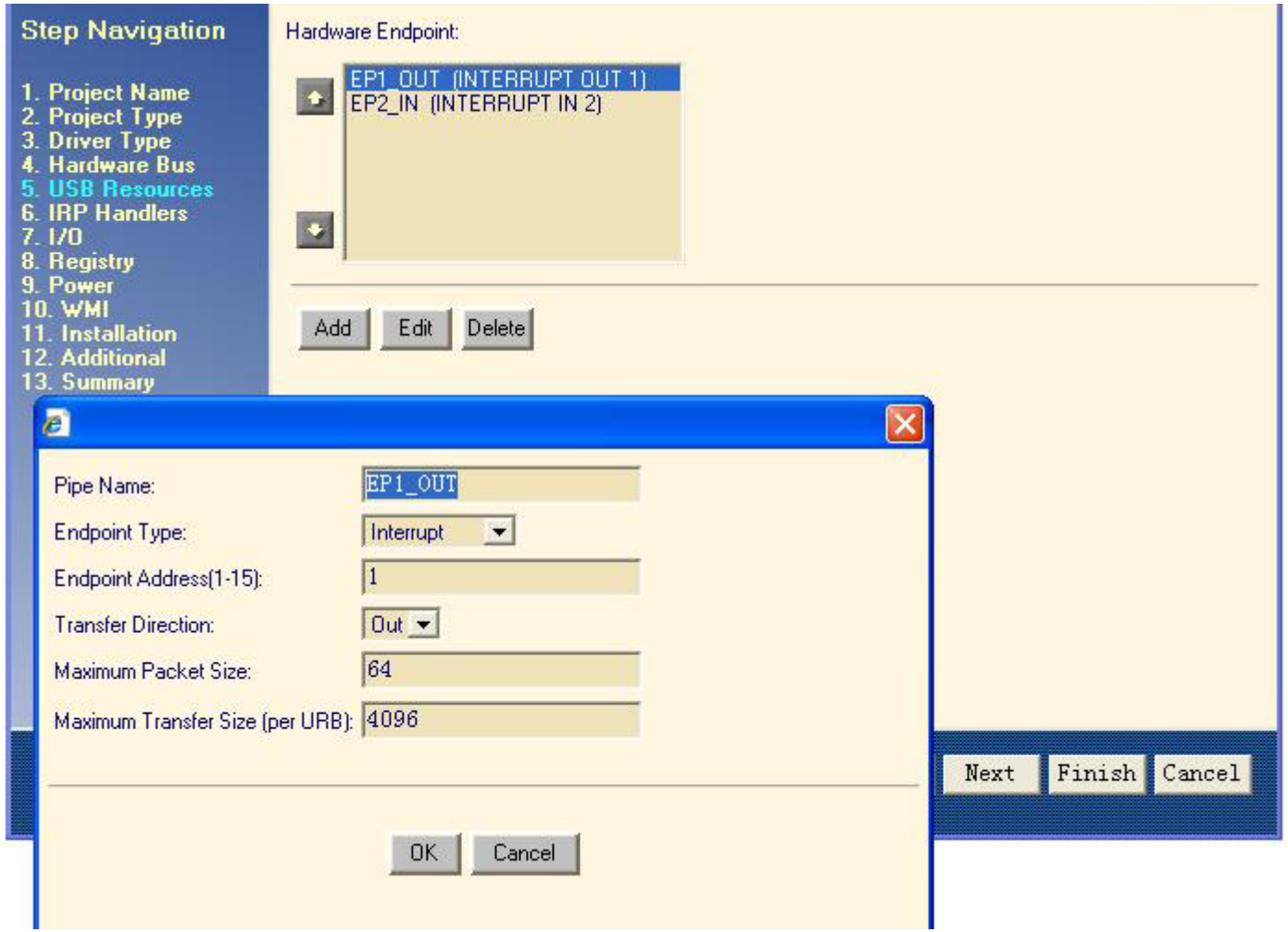


之后，DriverStudio 以插件形式嵌入到 VC6：

这样，就可以进行驱动开发了，首先打开 DriverWizard 生成需要设计的驱动框架，以 USB 驱动为例需要特殊说明的是：
这里填写的 Vendor ID 和 Product ID 必须与 USB 设备固件程序里设备描述符里的 Vendor ID 和 Product ID 一致，这两个 ID 用以寻找配对的驱动。

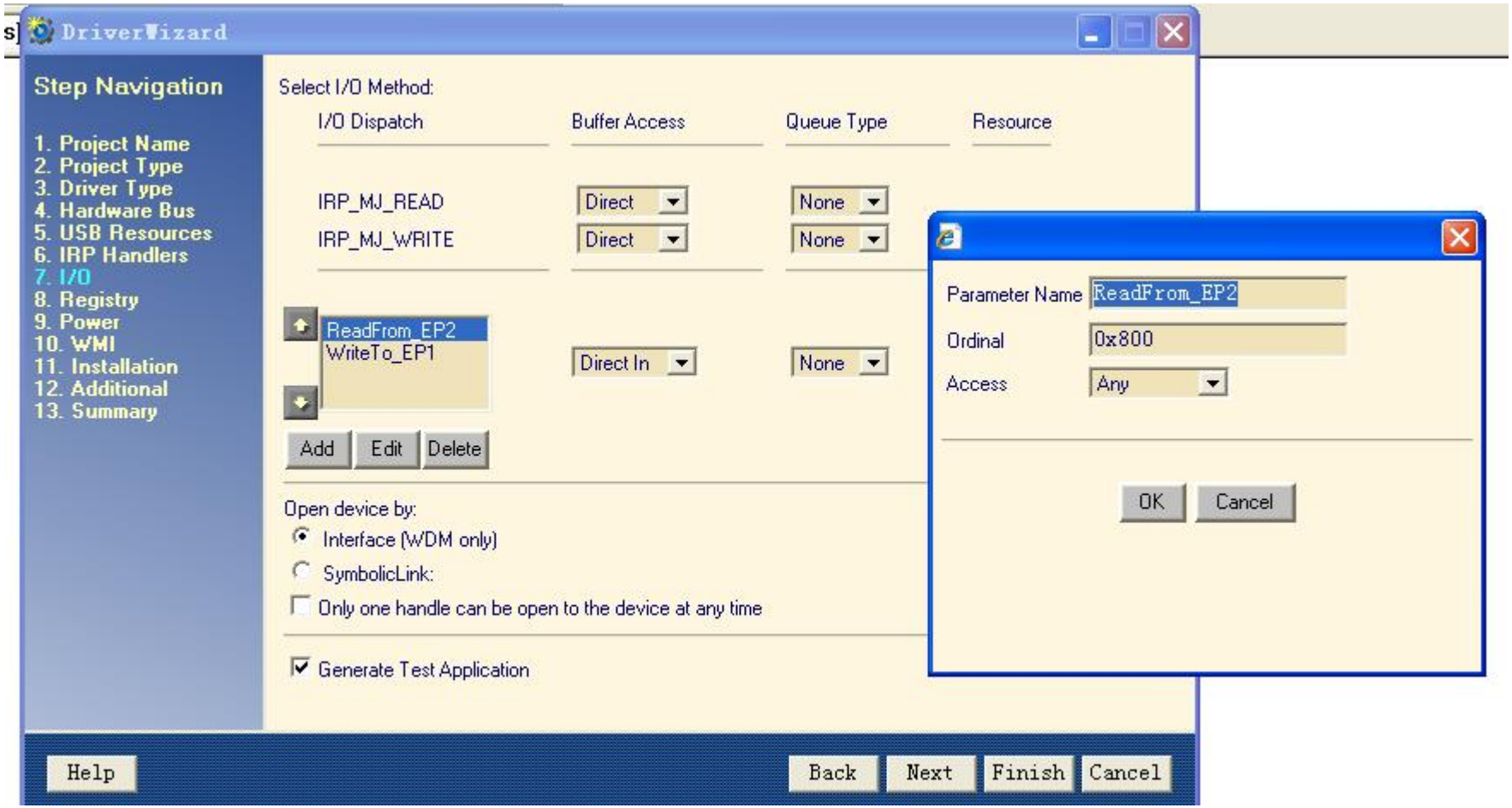


这里添加的是 USB 端点特性，在生成的程序框架中，管道名称(Pipe Name)将作为程序的内部变量成员，派属 KUsbPipe 类，端点操作函数都集成在此类中。在这设置 USB 设备 Enpoint 1 为接收端口，Enpoint 2 为发送端口。

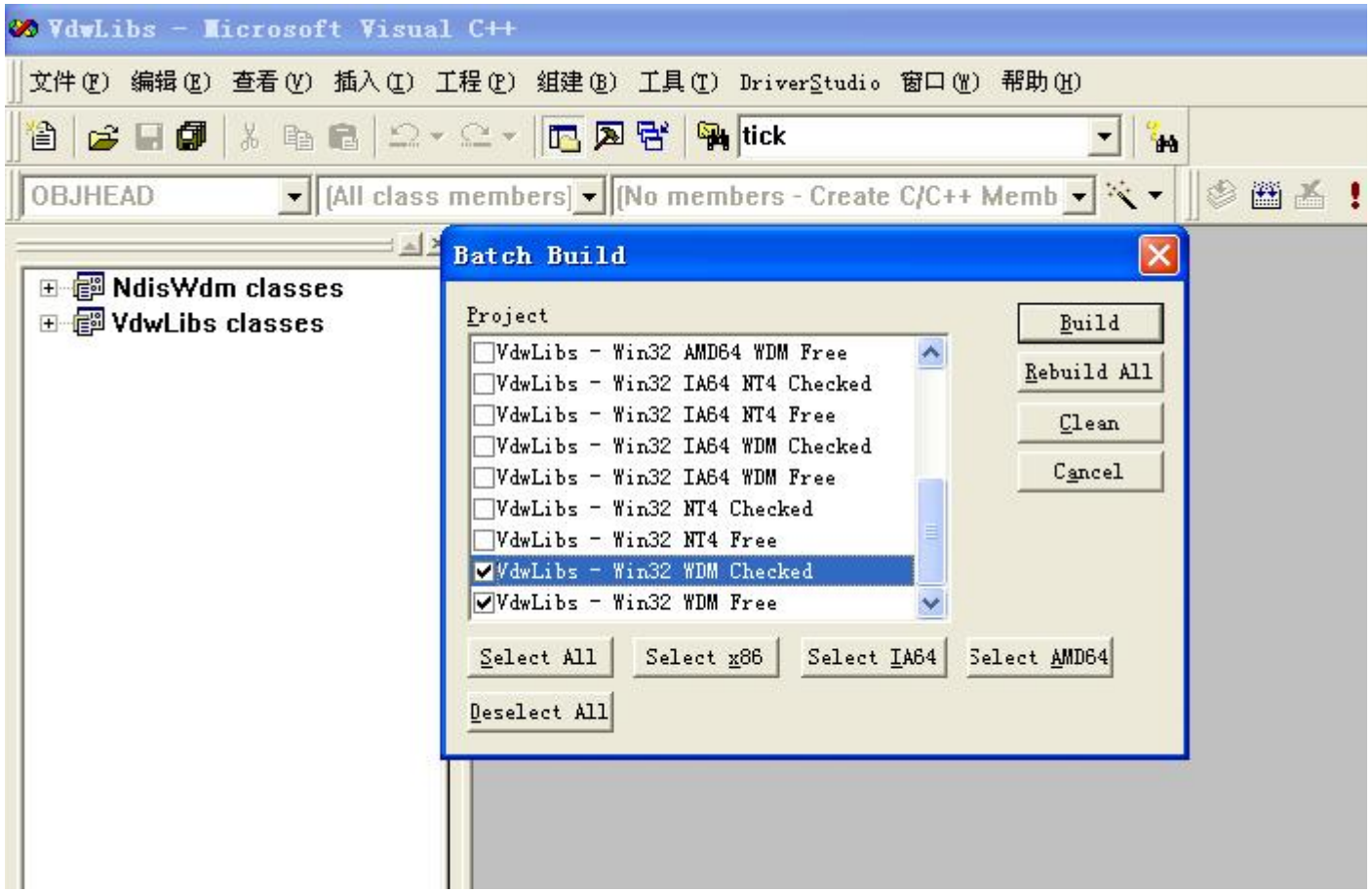


这 里添加 USB 的控制操作方式，在 NT 平台下应用程序控制设备只能通过 ReadFile()/WriteFile()和 DeviceIoControl() 两做方式，执行 ReadFile()/WriteFile()将响应 IRP_MJ_READ/IRP_MJ_WRITE 请求，在这添加了 ReadFrom_EP2 和 WriteTo_EP1 两个 IRP_MJ_DEVICE_CONTROL 请求代码，在执行 DeviceIoControl() 时，可以根据请求这两个请求代码进行区别，DriverWizard 生成的框架中将增添两个函数：

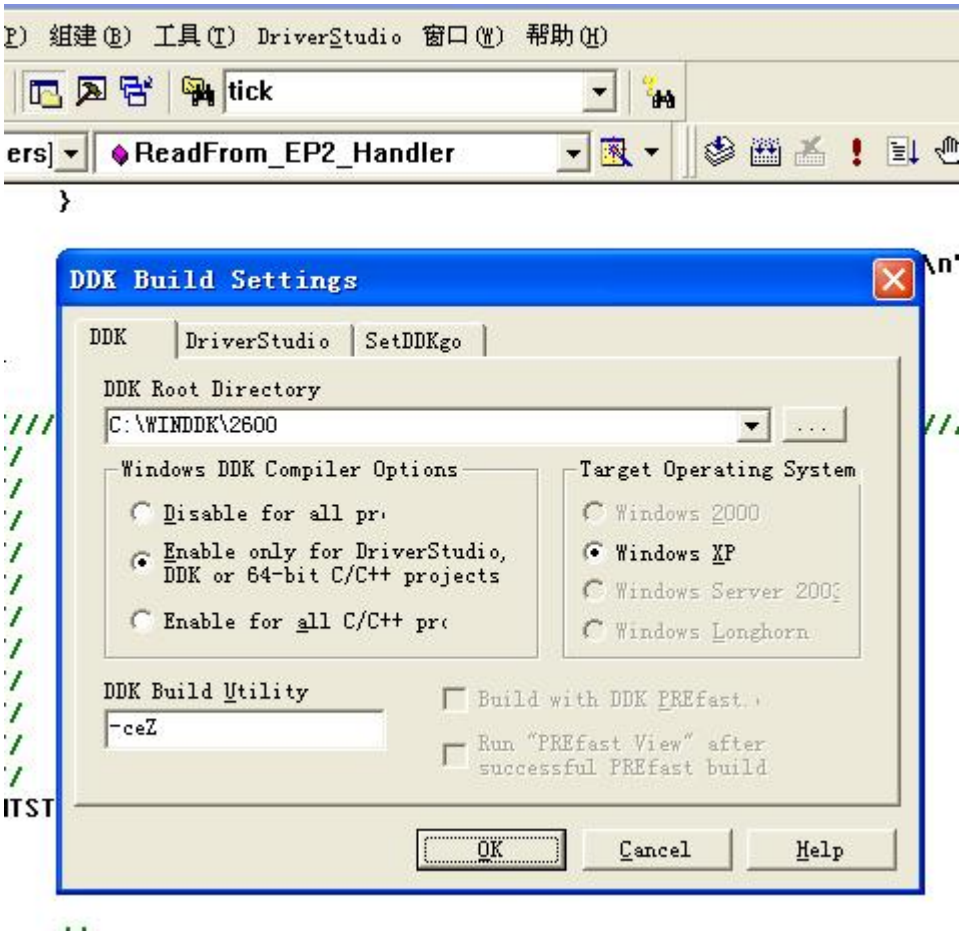
```
NTSTATUS ezUSBDevice::ReadFrom_EP2_Handler(KIrp I);
NTSTATUS ezUSBDevice::WriteTo_EP1_Handler(KIrp I);
```



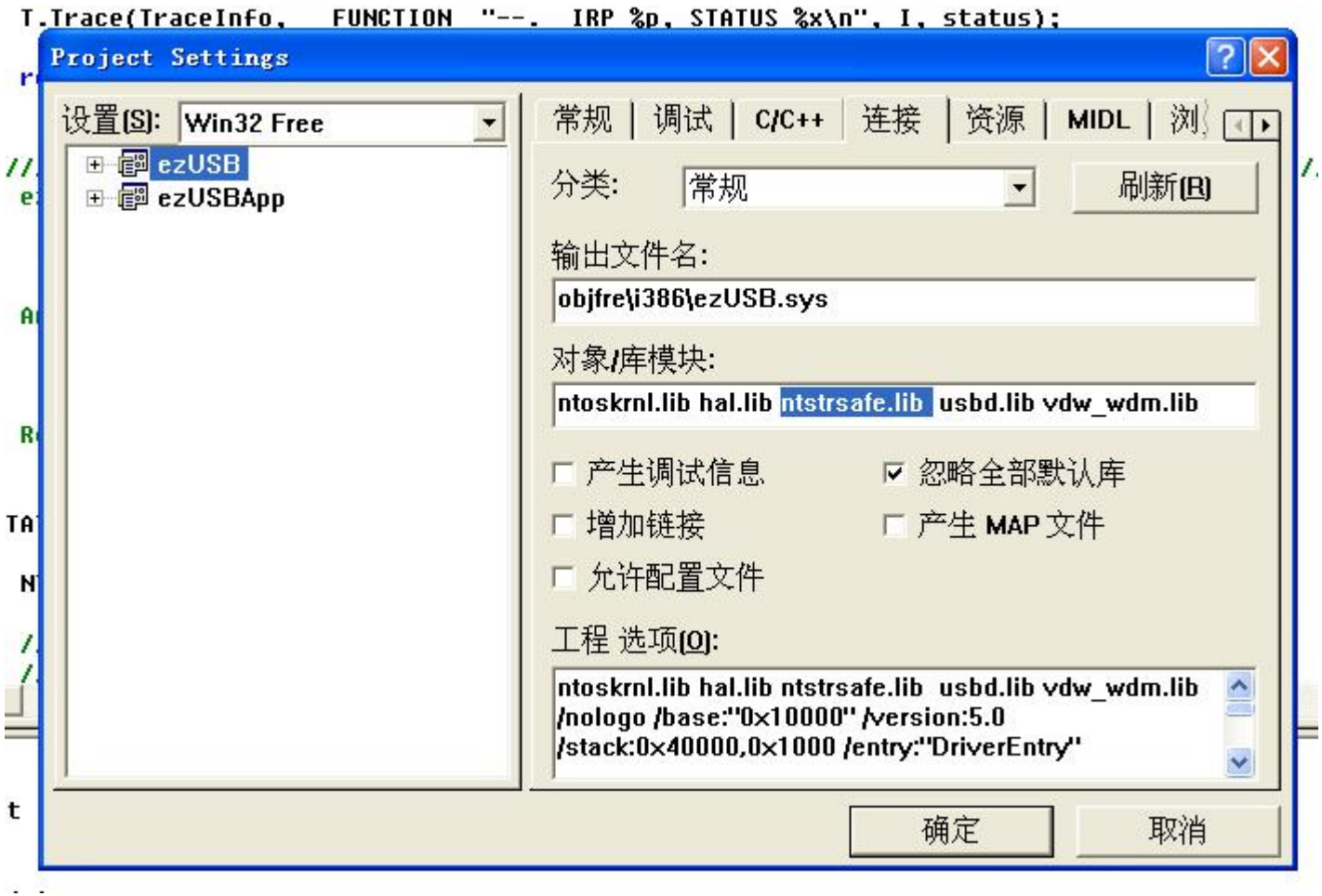
如 果 DriverStudio 首次安装后，请先利用 VC6 打开\Compuware\DriverStudio\DriverWorks\source \VdwLibs.dsw 工程，然后按照以下方法编译：VC6 主菜单->Bulid->Batch Bulid，按图示设置后点击 Rebuild All，编译成功后关闭此项目：



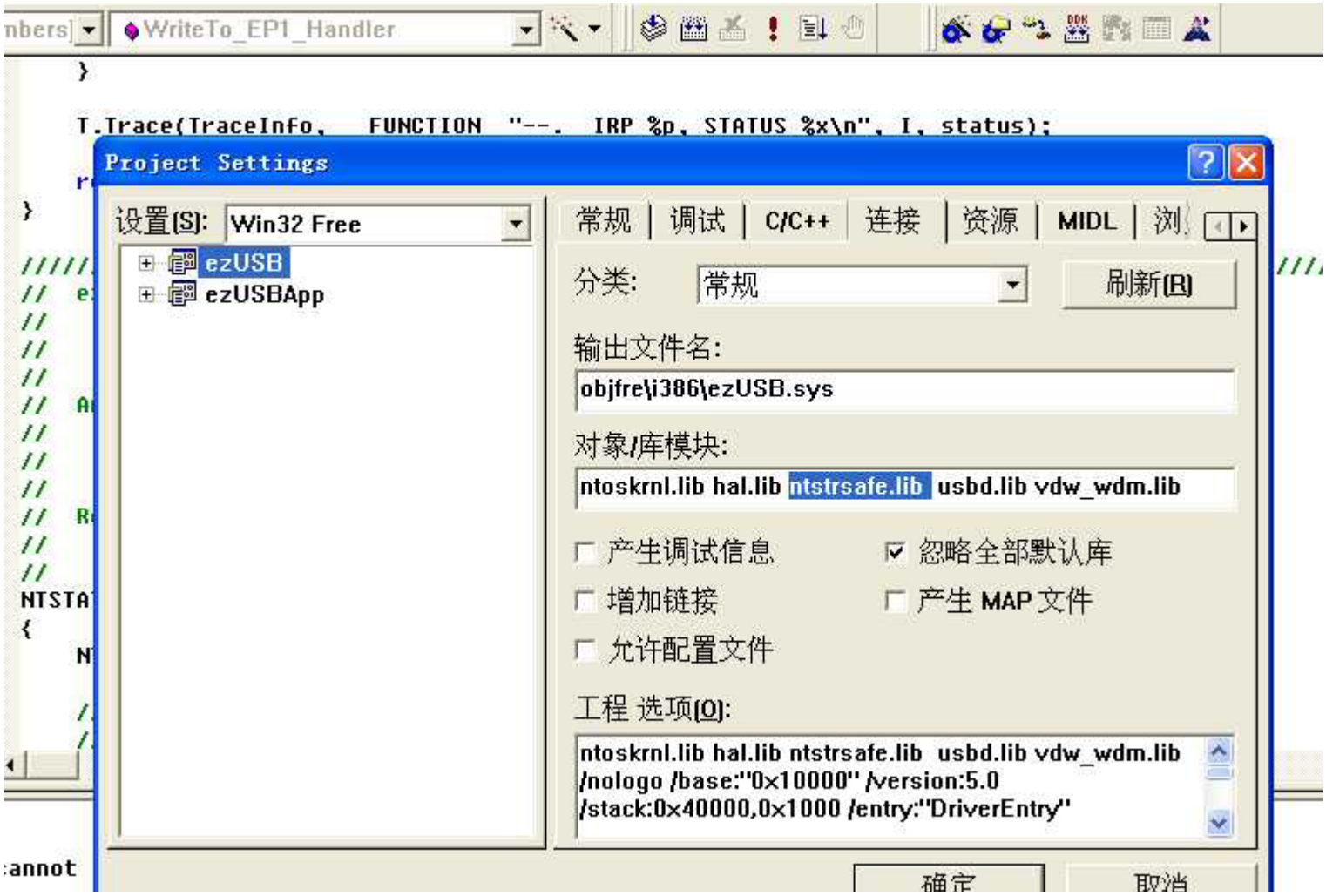
DriverWizard 框架生成完成后，就可以在设定的路径下找到你设置的工程，至此就可以使用 VC6 打开工程。打开后请先打开 VC6 中 DriverStudio 插件：VC6 主 菜单->DriverStudio->DDK Build Settings，选择 DDK 目录：



至此就可以编译该工程了，如果提示找不到库:ntstrsafe.lib，请删除此库：



到此，对利用 DriverStudio 进行 USB 驱动开发的开发环境的设置做了一些着重的介绍，具体工程如出现一些特殊情况请利用网络资源收集，下篇将介绍 USB 应用程序与驱动之间的数据交换。



第七篇：XP 下 USB 驱动开发的最终完成

这是我进行的唯一一次驱动开发，对 DDK 以及 DriverStudio 知之甚少，驱动代码部分不做阐述，在这我将 STM32-USB 驱动-应用程序串联起来说明。

在 VC6 环境下，连接 USB 驱动部分我写了个类 CUSBAPI 来封装该操作，在 USBAPI.h 文件中：

```
#define FILE_DEVICE_EZUSB  0x8000

#define EZUSB_IOCTL(index) \
    CTL_CODE(FILE_DEVICE_EZUSB, index, METHOD_BUFFERED, FILE_READ_DATA)

#define ReadFrom_EP2 \
    CTL_CODE(FILE_DEVICE_EZUSB, 0x800, METHOD_IN_DIRECT, FILE_ANY_ACCESS)
#define WriteTo_EP1 \
    CTL_CODE(FILE_DEVICE_EZUSB, 0x801, METHOD_OUT_DIRECT, FILE_ANY_ACCESS)
```

这部分定义的是 DeviceIoControl()函数所需要的 I/O 控制代码，此定义在 DriverWizard 生成的 interface.h 文件中，在这可包含 interface.h 也可以复制过来进行定义。

```
typedef struct _NODE_ENUDEVICEINTERFACE
{
    CHAR    pDeviceInterfaceSymbolicName[MAX_PATH];        // Index for this node

    struct _NODE_ENUDEVICEINTERFACE *pNext;
}
NODE_ENUMDI,
```

*PNODE_ENUMDI;
这个链表用以存储USB设备的接口名称，需要另外说明的一点，在USBAPI.c文件中定义了：
DEFINE_GUID(GUID_DEVICEINTERFACE,
    0xE075C5B2, 0xE7FB, 0x4186, 0xA1, 0x39, 0x0B, 0x3F, 0xD2, 0x05, 0xE7, 0x6E);
需要通过这个GUID_DEVICEINTERFACE取得该接口名称，有了该接口名称后就可以通过CreateFile()获取该接口的句柄，进而可以通过ReadFile()/WriteFile()或者DeviceIoControl()读写USB设备。获取一个设备可能有多个接口，在该类中建立了一个循环链表用以存储该信息。

```
typedef struct _STRUCT_IO
{
    HWND hTargetWnd;

    HANDLE hDevice;

    DWORD dwIoControlCode;

    PCHAR pInBuffer;
    DWORD dwInSize;

    PCHAR pOutBuffer;
    DWORD dwOutSize;

    LPDWORD lpBytesReturned;
}
STRUCT_IO,
*PSTRUCT_IO;
```

这个结构中的 hTargetWnd 定义了消息对象的窗口句柄，用以向该窗口发送读写数据完成的消息；hDevice 即 USB 设备接口的句柄；其他的含义很明了，就不说明了。

```
#define CORESTATUS_SUCCESS                          0x0000L
#define CORESTATUS_DESTROY                          0x0001L
#define CORESTATUS_READWRITE_EVENT_ERROR        0x0002L
#define CORESTATUS_READWRITE_THREAD_ERROR        0x0003L
#define CORESTATUS_IOCONTROL_EVENT_ERROR        0x0004L
#define CORESTATUS_IOCONTROL_THREAD_ERROR        0x0005L
```

这些是定义的类状态，应用程序可以获取这些状态。

```
#define MSG_READWRITE_COMPLETION                WM_USER+0x0010
#define MSG_IOCONTROL_COMPLETION                WM_USER+0x0011
```

这些定义的是自定义消息码，应用程序识别此消息码可得知读写操作已完成。

```
#define ERROR_HANDLE_WINDOW                         0x1000L
#define ERROR_HANDLE_DEVICE                         0x1001L
#define ERROR_BUFFER_LENGTH                         0x1002L
#define ERROR_BUFFER_ISNULL                         0x1003L
#define ERROR_READWRITE_BUSY                    0x1004L
#define ERROR_IOCONTROL_BUSY                    0x1005L
```

这些定义的进行读写操作时，进行的一些参数检查并返回的状态。

下面这些是类成员函数以及变量，USBAPI 类内建立了两个独立线程，这样在对 USB 设备进行读写时，就不会堵塞应用程序的窗口线程，读写操作完成后由消息 MSG_READWRITE_COMPLETION 和 MSG_IOCONTROL_COMPLETION 通知应用程序。详细代码请参考源程序。

```
// ************************************************************************************************************
```

```
// Class members definition
// ********************************************************************************************************
class CUSBAPI
{
public:
    CUSBAPI();
    virtual ~CUSBAPI();

public:

    DWORD EnumDeviceInterface(LPGUID pGUID);
    HANDLE OpenDeviceInterface(PCHAR pDeviceInterfaceSymbolicName);

    DWORD Execute_ReadFile(
            HWND    hWnd,
            HANDLE    hDevice,
            PCHAR    pInBuffer,
            DWORD    dwInSize,
            LPDWORD lpBytesReturned
            );

    DWORD Execute_WriteFile(
            HWND    hWnd,
            HANDLE    hDevice,
            PCHAR    pOutBuffer,
            DWORD    dwOutSize,
```

```
            LPDWORD lpBytesReturned
            );

    DWORD Execute_IoControl(
            HWND    hWnd,
            HANDLE    hDevice,
            DWORD    dwIoControlCode,
            PCHAR    pInBuffer,
            DWORD    dwInSize,
            PCHAR    pOutBuffer,
            DWORD    dwOutSize,
            LPDWORD lpBytesReturned
            );

    BOOL Node_HeadCreate(VOID);
    VOID Node_HeadDelete(VOID);
    VOID Node_RemoveAll(VOID);
    BOOL Node_Append(PCHAR pDeviceInterfaceSymbolicName);
    VOID Node_Remove(PCHAR pDeviceInterfaceSymbolicName);
    PNODE_ENUMDI Node_Find(PCHAR pDeviceInterfaceSymbolicName);

public:
    GUID GUID_Device;

    PNODE_ENUMDI pEnumDeviceNode;
    PNODE_ENUMDI pEnumDeviceHead;
```

```
    HANDLE hEvent_ReadWrite;
    HANDLE hEvent_IoControl;

    HANDLE hThread_ReadWrite;
    HANDLE hThread_IoControl;

    DWORD dwThreadID_ReadWrite;
    DWORD dwThreadID_IoControl;

    STRUCT_IO ReadWrite;
    STRUCT_IO IoControl;

    BOOL bExecuting_IoControl;
    BOOL bExecuting_ReadWrite;

    DWORD dwCoreStatus;
};
```

在应用程序中定义变量：CUSBAPI ezUSB;
读写操作函数也就三种：
Execute_IoControl()、Execute_ReadFile()、Execute_WriteFile()

执行这些函数后，将与 USB 的驱动程序挂钩，分别响应：
Execute_IoControl() -> NTSTATUS ezUSBDevice::DeviceControl(KIrp I)
Execute_ReadFile() -> NTSTATUS ezUSBDevice::Read(KIrp I)
Execute_WriteFile() -> NTSTATUS ezUSBDevice::Write(KIrp I)

其中 NTSTATUS ezUSBDevice::DeviceControl(KIrp I)根据 I.IoctlCode()区别类型，按照此示例说明:
ReadFrom_EP2->Execute_IoControl()->NTSTATUS ezUSBDevice::ReadFrom_EP2_Handler(KIrp I)
WriteTo_EP1->Execute_IoControl()->NTSTATUS ezUSBDevice::WriteTo_EP1_Handler(KIrp I)

这样应用程序与 USB 驱动之间就建立了通讯渠道，在驱动函数中：
NTSTATUS ezUSBDevice::DeviceControl(KIrp I)
NTSTATUS ezUSBDevice::Read(KIrp I)
NTSTATUS ezUSBDevice::Write(KIrp I)
执行一些操作就可以与 STM32 的 USB 设备进行通讯了，此时就需要很好的掌握 DriverStudio 封装的各种类库了。
DriverStudio 向导生成的框架，一般就只需要更改这三个函数接口，当然，对于 DriverStudio 向导的一个 BUG 不可不知：

```
    // Initialize each Pipe object
    EP1_OUT.Initialize(m_Lower, 1, 64);
    EP2_IN.Initialize(m_Lower, 82, 64);
```

Initialize()函数第二参数是端点地址，在这是 16 进制表示，这里需要补上 0x:

```
    // Initialize each Pipe object
    EP1_OUT.Initialize(m_Lower, 0x01, 64);
    EP2_IN.Initialize(m_Lower, 0x82, 64);
```

忘记此处修改的后果是，执行 EP2_IN 操作将会使系统直接蓝屏。
这三个函数接口中涉及到读写操作方式，比如说 buffer 或者 direct io，具体有什么区别，请从网络搜寻。

USB 驱动负责底层通过端口地址及方式与 STM32 连接后，

EP1_OUT：USB 主机向 USB 设备发送数据，void CTR_OUT1(void)函数响应
EP2_IN：USB 主机请求 USB 设备发送数据，void CTR_IN2(void)函数响应
示例中 CTR_OUT1()接收 2Bytes 数据，CTR_IN2()发送 2Bytes 数据，分别控制 LED1-4 和定时获取 Joystick 的状态:

```
void CTR_OUT1(void)
{
  unsigned short portc;
  unsigned short wCount;

  wCount = GetBuffDescTable_RXCount(ENDP1);

  if(wCount == 2)
  {
    //portc = GPIO_ReadInputData(GPIOC);

    BufferCopy_PMAToUser((unsigned char *)&portc, GetBuffDescTable_RXAddr(ENDP1), 2);

    GPIO_Write(GPIOC, (GPIO_ReadInputData(GPIOC)&0xFF0F)|(portc&0x00F0));

  }

  SetEPR_RXStatus(ENDP1,EP_RX_VALID);
  SetEPR_TXStatus(ENDP1,EP_TX_STALL);
}

void CTR_IN2(void)
```

```
{
  unsigned short portd = GPIO_ReadInputData(GPIOD) & 0xF800;  // 11-15

    // Copy the transfer buffer to the endpoint0's buffer
    BufferCopy_UserToPMA( (unsigned char *)&portd,    // transfer buffer
                          GetBuffDescTable_TXAddr(ENDP2),           // endpoint 0 TX address
                          2);

  SetBuffDescTable_TXCount(ENDP2, 2);
  SetEPR_RXStatus(ENDP2,EP_RX_DIS);
  SetEPR_TXStatus(ENDP2,EP_TX_VALID);
}
```

至此，整个基于 STM32 的 USB 开发过程的介绍大致说了一遍，详细情况请参考源代码，这篇学习笔记到此结束了，水平有限，错误难免！谢谢这段时间关心与支持的朋友们，欢迎朋友们一起探讨学习。

zhudlmax@126.com

http://blog.ednchina.com/lbxxx

2008-11-25 小新.上海