第6章

USB转串口

在这个USB流行的年代，串口在PC上已经开始慢慢地退出历史舞台了。但是很多设备开发者，都喜欢使用串口与计算机进行通信，因为串口使用起来简单方便。那么矛盾就出现了，新装的计算机（尤其是笔记本）通常都没有串口，怎么跟设备的串口进行连接？有没有什么办法可以在计算机上增加串口呢？回答是肯定的，可以通过使用PCI卡设计一个或者多个串口。当然，通过USB口也可以模拟出串口设备，因为USB也是一种总线，总线就可以连接不同的设备。本章讲述的就是如何设计一个USB转串口的设备。

6.1 串口家族历史

通用异步串行通信口（简称串口或者COM口）是一种比较古老的串行通信口，在前几年的PC上，几乎是必备的接口。通常一台台式计算机上有一到两个串口。在圈圈开始玩计算机的那个年代，PC上的串口使用的都是9针（DB9）公头。而在更古老一些的计算机上，串口使用的是25针（DB25）的公头。像圈圈2002年装的台式机，就有2个DB9的串口，另外还有一个25孔的DB25母头打印机接口。DB25的打印机接口在现在的新计算机中几乎没有了，新的打印机基本上都是USB接口。以前还有串口鼠标（估计很多年纪比较小的朋友都没玩过这个，圈圈也没玩过），到圈圈玩计算机那个时代，基本上都是PS2接口的了，键盘也是使用的PS2接口。前段时间听一位朋友说，他陪他的朋友去装计算机，结果机箱背后有一堆USB口（好象是8个，前面面板上还有6个USB口）和几个音频口，像PS2口、串口、并口都没有了。当然网口、接显示器用的VGA口是有的，不过它们并没有集成在主板上，是以子卡的方式插在主板上的。

6.2 串口接头的引脚分布及功能

像串口这样的DB头，如何区分公母以及引脚编号呢？接头中的电气接点是一根根小金属棒（针）的，就是公头，电气接点是一个个孔的，就是母头，读者自己好好体会一下吧。至于引脚编号，在接口中的引脚旁边通常用小数字标注了，仔细观察就可

以找到这些小数字了。如果没有数字，则可以按以下方法记忆：将 DB 头插接面对准自己，宽的一面在上，对于公头，上面一排从左往右数刚好就是 1～5 脚；而对于母头，从右往左数是 1～5 脚。公头的 1 脚在左边，母头的 1 脚在右边。下面一排的顺序跟上面一排是一样的，为 6～9 脚。

在 PC 串口的 DB9 的 9 根引脚中，最常用的就是 2 脚(RXD，数据接收引脚)、3 脚(TXD，数据发送引脚)和 5 脚(GND，地)。另外还有几根不常用，只有在使用硬件流控制时才使用。在测试串口线路是否能够正常工作时，常使用自收发法：将串口的 TXD 和 RXD 引脚连接在一起，然后往串口发送数据，看自己能否收到。这时发送在 TXD 上的数据将被 RXD 引脚接收回去，如果串口工作正常，应该能够接收到自己发送的数据。如果不正常，则应该重点检查串口连接线是否正确，当然也可能是串口已经损坏了。

在我们的学习板上，有一个 DB9 母头，它的 2 脚是 TXD，3 脚是 RXD，刚好跟计算机端的 DB9 头的 2 脚和 3 脚交叉。它是可以直接跟计算机的 DB9 公头相连的，或者使用串口直连线连接。有些设备的串口虽然使用的也是 DB9 母头，但它没有将 2 脚跟 3 脚交叉，这样就不能直接与计算机的串口相连，而应该使用交叉线连接。如果要把学习板上的串口改装成计算机上的串口那样，必须自己找一个转接头，或者使用一条两端都是公头的交叉串口线。但是，圈圈在电子市场上转了几圈，也没有买到这样的串口线。只好买了一个两头都是公头、直连型的转接头，再买了一条一端是公头另一端是母头的串口交叉线，这样就可以得到一个引脚分布跟计算机端一样的串口了。

计算机的串口也叫做 RS－232 接口，因为它使用的是 RS－232 电平规范。RS－232 电平是一种负逻辑电平，负电平表示逻辑 1，正电平表示逻辑 0，这跟 TTL 电平刚好是相反的。TTL 跟 RS－232 电平之间的转换常使用 MAX232 芯片。

6.3　USB 转串口的实现方法

要实现 USB 转串口，有两种可行的方法：

①使用用户自定义 USB 设备，然后开发其驱动程序，由驱动程序生成串口；

②使用 USB 协议规定的 CDC 类中的抽象控制模型(abstract control model)子类中的通用 AT 命令(common AT commands)协议。

使用方法①需要用户自行开发驱动程序，比较麻烦，但是灵活性较强；使用方法②不需要用户自己开发驱动程序，只需要提供一个安装驱动的 inf 文件即可，但是灵活性不强，会受到一些限制。本章将使用方法②中所说的 CDC 类来实现，对于安装驱动所需要的 inf 文件，本章不会详细介绍如何编写，仅介绍如何在原来的基础上修改它。

由于使用 AT 命令的设备软件通常是通过串口通信的，因而也就会打开串口设

备。为了能够让这些较老的软件在USB环境中还能够继续使用，在CDC协议中就增加了抽象控制模型子类中的AT命令的协议，它可以增加一个虚拟串口设备。实际的串口数据通过非0端点的批量管道来传输，而对于设置波特率、数据位格式等，则通过控制端点0发送类请求来实现。要实现USB数据与串口数据之间的相互转换，只需要将从USB批量输出端点接收的数据发送到串口的TXD，将从串口RXD接收的数据发送到USB批量输入端点即可。由于USB的数据发送和接收是按数据包进行的，所以必须开一个缓冲区来保存这些数据；否则，串口接收的数据可能来不及从USB发送出去，从而导致数据丢失。

如果要深入理解CDC协议，需要读者自行阅读USB CDC协议文档。本章仅介绍如何实现该设备，并不会对USB CDC协议作非常详细的讲解。其实，在CDC协议中有实现这个USB转串口所需要的描述符的例子。

为了加快数据传输的速度，避免数据丢失，本程序决定使用D12的端点2作为批量传输，因为它的缓冲区有64字节，并且有双缓冲机制。在使用双接口实现带鼠标USB键盘程序中，我们已经使用过端点2了，因此本实例就用它来修改成USB转串口。将UsbKeyboardWithMouse(TwoInterfaces)复制一份，改名为UsbToUart。

6.4 设备描述符

要修改一个新的USB程序，当然首先就要修改设备描述符。与前面几个设备不同，本设备必须在设备描述符中指定设备的类型，即设备类bDeviceClass字段必须指定为0x02(通信设备类的类代码)；否则，会由于在配置集合中有两个接口，而被系统认为是一个USB复合设备，从而导致设备工作不正常。当指定了设备类型为通信设备类后，子类和所使用的协议都必须指定为0。厂商ID号不变，产品ID号继续使用前面的生成方式，设置为0x0007。最终的设备描述符代码如下：

```
//USB设备描述符的定义
code uint8 DeviceDescriptor[0x12] =   //设备描述符为18字节
{
//bLength字段。设备描述符的长度为18(0x12)字节
 0x12,
//bDescriptorType字段。设备描述符的编号为0x01
 0x01,
//bcdUSB字段。这里设置版本为USB1.1，即0x0110
//由于是小端结构，所以低字节在先，即0x10、0x01
 0x10,
 0x01,
//bDeviceClass字段。本设备必须在设备描述符中指定设备的类型，否则，由于在配置集合
```

中有

```
    //两个接口,就会被系统认为是一个 USB 复合设备,从而导致设备工作不正常
    //0x02 为通信设备类的类代码
     0x02,
    //bDeviceSubClass 字段。必须为 0
     0x00,
    //bDeviceProtocol 字段。必须为 0
     0x00,
    //bMaxPacketSize0 字段。PDIUSBD12 的端点 0 大小的 16 字节
     0x10,
    //idVender 字段。厂商 ID 号,这里取 0x8888,仅供实验用
    //实际产品不能随便使用厂商 ID 号,必须跟 USB 协会申请厂商 ID 号,注意小端模式,低字节在先
     0x88,
     0x88,
    //idProduct 字段。产品 ID 号,由于是第七个实验,这里取 0x0007;注意小端模式,低字节在先
     0x07,
     0x00,
    //bcdDevice 字段。取 1.0 版,即 0x0100;小端模式,低字节在先
     0x00,
     0x01,
    //iManufacturer 字段。厂商字符串的索引值,为了方便记忆和管理,字符串索引就从 1 开始
     0x01,
    //iProduct 字段。产品字符串的索引值;刚刚用了 1,这里就取 2 吧;注意字符串索引值不要使用相同的值
     0x02,
    //iSerialNumber 字段。设备的序列号字符串索引值;这里取 3 就可以了
     0x03,
    //bNumConfigurations 字段。该设备所具有的配置数;只需要一种配置就行了,因此该值设置为 1
     0x01
    };
    ////////////////////////设备描述符完毕////////////////////////////////
```

6.5　字符串描述符

字符串描述符可以根据自己的喜好来填写,这里就不再重复讲述了。具体可以

参看前面的章节以及源代码。

6.6　配置描述符集合

USB 转串口的配置描述符集合稍微复杂一些，因为在 CDC 类的协议中定义了一些类特殊接口描述符。此外，它还具有两个接口，CDC 类接口和数据类接口。

6.6.1　配置描述符

由于原来的程序也是两个接口的，本设备也是两个接口的，所以配置描述符不用修改，直接使用原来的就行了。

6.6.2　CDC 接口描述符

在 CDC 设备中，必须要有一个 CDC 接口，以供数据类接口依附。CDC 接口使用标准的接口描述符，它有一个中断输入端点，用来报告一些状态。但是在协议中似乎并没有说明该端点如何使用，所以圈圈暂时也没去理会这个端点了。要实现虚拟串口的功能，必须在该接口描述符中指定使用 CDC 接口类以及 Abstract Control Model 子类和 Common AT Commands 协议。实际实现的 CDC 接口描述符如下(在原来的接口 0 上修改)：

```
/**********************CDC 类接口描述符**************************/
 //bLength 字段。接口描述符的长度为 9 字节
 0x09,

 //bDescriptorType 字段。接口描述符的编号为 0x04
 0x04,

 //bInterfaceNumber 字段。该接口的编号，第一个接口，编号为 0
 0x00,

 //bAlternateSetting 字段。该接口的备用编号，为 0
 0x00,

 //bNumEndpoints 字段。非 0 端点的数目。CDC 接口只使用一个中断输入端点
 0x01,

 //bInterfaceClass 字段。该接口所使用的类。CDC 接口类代码为 0x02
 0x02,

 //bInterfaceSubClass 字段。该接口所使用的子类。要实现 USB 转串口，就必须
 //使用 Abstract Control Model(抽象控制模型)子类。它的编号为 0x02
 0x02,

 //bInterfaceProtocol 字段。使用 Common AT Commands(通用 AT 命令)协议。该协议的编号为 0x01
```

```
0x01,
//iConfiguration字段。该接口的字符串索引值。这里没有,为0
0x00,
```

6.6.3 类特殊接口描述符——功能描述符

在CDC类中,不再具有HID描述符和报告描述符了,因而在程序中将原来的HID描述符、报告描述符以及获取报告描述符处理等代码删除。取而代之的是叫做功能描述符(functional descriptors)的类特殊接口描述符(class-specific interface descriptor),它们主要用来描述接口的功能。功能描述符放在CDC接口(主接口)之后,功能描述符完毕之后就是主接口的端点描述符,再接下来是其他接口以及它们的端点描述符。

功能描述符的第一字节为该功能描述符的长度bFunctionLength,第二字节为该描述符的类型bDescriptorType,固定为0x24(CS_INTERFACE的编码),第三字节为描述符子类型bDescriptorSubtype,可以选择具体的功能描述符。剩余的数据与所选的描述符子类有关。

在抽象控制模型中需要用到的功能描述符有:Header Functional Descriptor(0x00)、Call Management Functional Descriptor(0x01)、Abstract Control Management Functional Descriptor(0x02)和Union Functional Descriptor(0x06)。后面括号中的数字表示该描述符子类的编号,它们将出现在功能描述符的bDescriptorSubtype字段中。

功能描述符的第一个描述符必须是Header Functional Descriptor,总共有5字节,前3字节分别为bFunctionLength、bDescriptorType和bDescriptorSubtype三个字段。后面2字节为USB通信设备协议的版本号,我们所参考的协议版本为1.1版,所以取0x0110。

Call Management Functional Descriptor有5字节,前面3字节的意义已经介绍过了,这里只说后面2字节。第四字节为bmCapabilities,它描述设备的能力,只有最低两位D0和D1有意义,其余位为保留值0。D0为0,表示设备自己不处理调用管理(call management),D0为1,表示自己处理。当D0为0时,D1将被忽略。D1表示调用管理通过数据类接口还是通信类接口,0表示仅通过通信类接口,1表示可通过数据类接口。在这里,我们将D0和D1都设置为0,即设备自己不处理调用管理。第五字节为bDataInterface,表示选择用来做调用管理的数据类接口编号,由于我们不使用数据类接口做调用管理,因而该字段设置为0。

Abstract Control Management Functional Descriptor有4字节,第四字节为bmCapabilities,描述设备的能力。其D4~D7位为保留位,设置为0。D0位表示是否支持以下请求:Set_Comm_Feature、Clear_Comm_Feature、Get_Comm_Feature为1表示支持。D1位表示是否支持Set_Line_Coding、Set_Control_Line_State、Get_

Line_Coding 请求和 Serial_State 通知，为 1 表示支持。D2 表示是否支持 Send_Break，为 1 表示支持。D3 表示是否支持 Network_Connection 通知，为 1 表示支持。本实例仅将 D1 设置为 1，即支持 Set_Line_Coding、Set_Control_Line_State、Get_Line_Coding 请求和 Serial_State 通知，其他请求和通知不支持。

Union Functional Descriptor 至少有 5 字节，它描述一组接口之间的关系可以被当作为一个功能单元来看待。这些接口中有一个作为主接口(master)，其他的作为从接口(slave)，可以有多个从接口。第四字节为主接口编号，第五字节为第一从接口编号，第六字节为第二从接口编号，以此类推。本实例只有一个从接口——数据类接口。

最终实现的功能描述符代码部分如下：

```
/**********功能描述符**********/
/********** Header Functional Descriptor ********/
//bFunctionLength 字段。该描述符长度为 5 字节
0x05,
//bDescriptorType 字段。描述符类型为类特殊接口(CS_INTERFACE),
//编号为 0x24
0x24,
//bDescriptorSubtype 字段。描述符子类为 Header Functional Descriptor,编号为 0x00
0x00,
//bcdCDC 字段。CDC 版本号，为 0x0110(低字节在先)
0x10,
0x01,
/******* Call Management Functional Descriptor ********/
//bFunctionLength 字段。该描述符长度为 5 字节
0x05,
//bDescriptorType 字段。描述符类型为类特殊接口(CS_INTERFACE),编号为 0x24
0x24,
//bDescriptorSubtype 字段。描述符子类为 Call Management functional descriptor,编号为 0x01
0x01,
//bmCapabilities 字段。设备自己不管理 call management
0x00,
//bDataInterface 字段。没有数据类接口用作 call management
0x00,
/*** Abstract Control Management Functional Descriptor ***/
//bFunctionLength 字段。该描述符长度为 4 字节
0x04,
//bDescriptorType 字段。描述符类型为类特殊接口(CS_INTERFACE),编号为 0x24
0x24,
```

```
    //bDescriptorSubtype 字段。描述符子类为 Abstract Control Management functional de-
scriptor,
    //编号为 0x02
    0x02,

    //bmCapabilities 字段。支持 Set_Control_Line_State、Get_Line_Coding 请求和 Serial_
State 通知
    0x02,

    /****Union Functional Descriptor****/
    //bFunctionLength 字段。该描述符长度为 5 字节
    0x05,

    //bDescriptorType 字段。描述符类型为类特殊接口(CS_INTERFACE),编号为 0x24
    0x24,

    //bDescriptorSubtype 字段。描述符子类为 Union functional descriptor,编号为 0x06
    0x06,

    //MasterInterface 字段。这里是前面编号为 0 的 CDC 接口
    0x00,

    //SlaveInterface 字段,这里是接下来编号为 1 的数据类接口
    0x01,
```

6.6.4 接口0(CDC接口)的端点描述符

接口0只有一个中断输入端点,这里使用端点1。原来的程序中,接口0也有一个中断输入端点1,因而原来的输入端点描述符不用做任何修改,可以直接使用。原来的输出端点1在这里不使用,删除之。具体的代码这里就不贴出了,可以参看光盘中的源代码或者前面的USB键盘章节。

6.6.5 数据类接口的接口描述符

CDC类接口(接口0)是负责管理整个设备的,而真正的串口数据传输是在数据类接口中进行的。这里只使用一个数据类接口,编号为1(在前面的Union Functional Descriptor中指定了编号为1的接口作为从接口,指的就是本接口)。

CDC协议中定义了数据接口的类代码为0x0A,接口子类代码和接口协议都为0。该接口要使用一对批量传输端点,因而端点数量为2。

数据类接口的接口描述符代码如下(在原来的接口1上修改,同时删除多余的HID描述符):

```
    /********** 接口1(数据接口)的接口描述符 **********/
    //bLength 字段。接口描述符的长度为 9 字节
    0x09,

    //bDescriptorType 字段。接口描述符的编号为 0x04
    0x04,
```

```
//bInterfaceNumber 字段。该接口的编号,第二个接口,编号为 1
0x01,

//bAlternateSetting 字段。该接口的备用编号为 0
0x00,

//bNumEndpoints 字段。非 0 端点的数目。该设备需要使用一对批量端点,设置为 2
0x02,

//bInterfaceClass 字段。该接口所使用的类。数据类接口的代码为 0x0A
0x0A,

//bInterfaceSubClass 字段。该接口所使用的子类为 0
0x00,

//bInterfaceProtocol 字段。该接口所使用的协议为 0
0x00,

//iConfiguration 字段。该接口的字符串索引值。这里没有,为 0
0x00,
```

6.6.6　接口 1(数据类接口)的端点描述符

接口 1 使用了一对批量端点:批量输入端点 2 和批量输出端点 2。批量输入端点用来从 USB 口返回串口数据,批量输出端点用来接收从 USB 口发来的"串口"数据。

将原来的中断输入端点 2 的描述符的 bmAttributes 字段由 0x03 改为 0x02,即,将端点的传输类型由中断传输改成了批量传输。查询时间(bInterval 字段)在这里已经没有意义了,可以设置为 0。然后再复制一份批量输入端点 2 的代码修改为批量输出端点 2,只需要将端点地址由 0x82 改成 0x02 即可。最终修改好的两个批量端点的描述符如下:

```
/*****************接口 1(数据类接口)的端点描述符 ************/
/*****************批量输入端点 2 描述符 ********************/
//bLength 字段。端点描述符长度为 7 字节
0x07,

//bDescriptorType 字段。端点描述符编号为 0x05
0x05,

//bEndpointAddress 字段。端点的地址。这里使用 D12 的输入端点 2
//D7 位表示数据方向,输入端点 D7 为 1,所以输入端点 2 的地址为 0x82
0x82,

//bmAttributes 字段。D1～D0 为端点传输类型选择
//该端点为批量端点,批量端点的编号为 0x02,其他位保留为 0
0x02,

//wMaxPacketSize 字段。该端点的最大包长。端点 2 的最大包长为 64 字节
//注意低字节在先。
```

```
    0x40,
    0x00,

    //bInterval 字段。端点查询的时间,这里对批量端点无效
    0x00,

    /*****************批量输出端点 2 描述符 ********************/
    //bLength 字段。端点描述符长度为 7 字节
    0x07,

    //bDescriptorType 字段。端点描述符编号为 0x05
    0x05,

    //bEndpointAddress 字段。端点的地址。这里使用 D12 的输出端点 2
    //D7 位表示数据方向,输出端点 D7 为 0,所以输出端点 2 的地址为 0x02
    0x02,

    //bmAttributes 字段。D1～D0 为端点传输类型选择
    //该端点为批量端点,批量端点的编号为 0x02,其他位保留为 0
    0x02,

    //wMaxPacketSize 字段。该端点的最大包长。端点 2 的最大包长为 64 字节。注意低字节在先
    0x40,
    0x00,

    //bInterval 字段。端点查询的时间,这里对批量端点无效
    0x00
```

6.6.7　修改好描述符后的测试

经过一番辛苦,终于将各种描述符修改好了。但是我们还有一些发送到接口的类请求以及对端点的数据处理还未完成,不过,为了马上享受一下修改的成果,先将程序烧入进去试一试。打开调试信息的宏定义,连接好串口,烧写程序后运行。

从串口返回的调试信息可以看出,主机在请求了设备描述符、设置地址、获取配置描述符、获取字符串描述符之后就停了下来,然后弹出了发现新硬件的对话框(见图 6.6.1),而并没有出现类请求和设置配置的请求,因为这些请求是在设备驱动程序中实现的,必须要安装了设备驱动程序之后才会发出。

图 6.6.1　发现新硬件

此时会弹出一个“找到新的硬件向导”的对话框,提示我们需要安装驱动。选择

"从列表或指定位置安装(高级)(S)",然后单击"下一步",如图 6.6.2 所示。这时会出现一个新的对话框,选择驱动所在的位置,如图 6.6.3 所示。单击对话框中的"浏

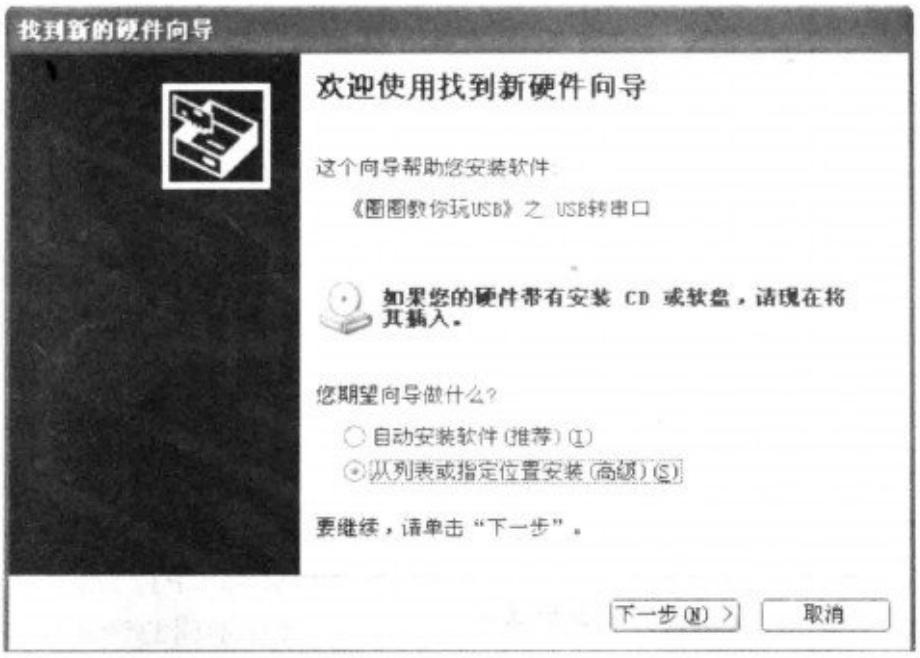

图 6.6.2　驱动安装步骤 1

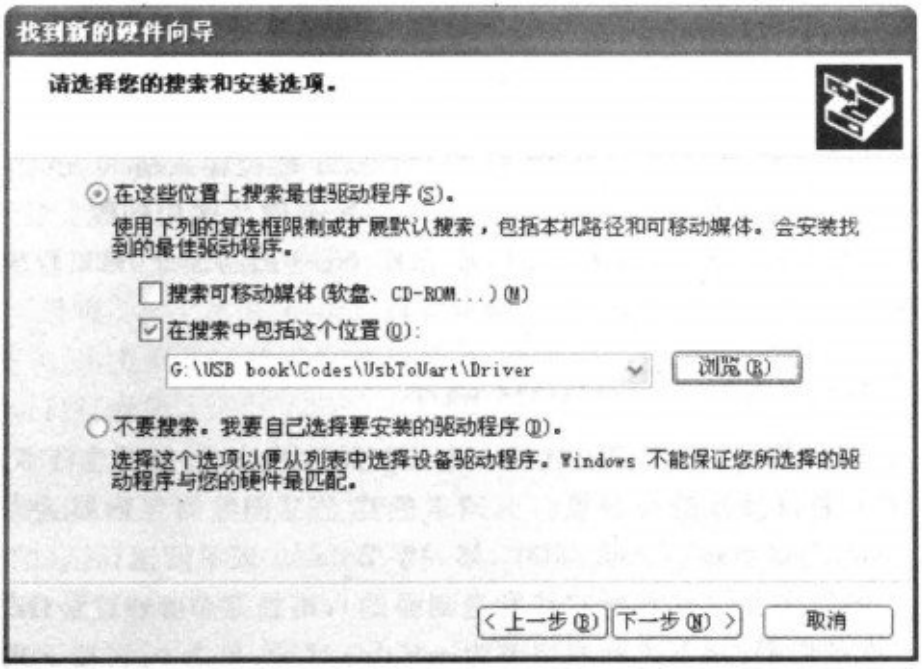

图 6.6.3　驱动安装步骤 2

览"按钮,指定光盘中 USB 转串口程序目录下的 inf 文件的位置,它在光盘目录\Codes\UsbToUart\Driver 下。然后单击"下一步",等待搜索。当搜索到指定的 inf

文件后，就会提示开始安装驱动，正在复制文件。此时学习板会收到设置配置的请求，然后再收到一个类输入请求，但是我们并没有实现这个请求的处理，因而驱动程序就一直在等待这个请求的返回。一段时间后，驱动程序检测到超时，就放弃了这个输入请求，又另外发了一个输出请求，但是我们的设备依然不认识这个请求，又是等待超时。最后，显示驱动安装成功，设备已经可以使用了。以上描述的过程在串口上显示的信息如下：

```
USB端点0输出中断。
读端点0缓冲区8字节。
0xA1 0x21 0x00 0x00 0x00 0x00 0x07 0x00
USB类输入请求：
USB端点0输出中断。
读端点0缓冲区8字节。
0x21 0x22 0x00 0x00 0x00 0x00 0x00 0x00
USB类输出请求：未知请求。
```

然后再打开设备管理器，可以看到在“端口（COM和LPT）”增加了一个串口“电脑圈圈做的USB转串口(COM10)”（后面的串口号在不同的计算机上可能不一样），再双击它可以打开该设备属性，里面有我们在inf文件中所设置的制造商、驱动程序提供商等信息。

6.7 类请求的实现

在前面讲到，主机发送了一个bRequest为0x21的类输入请求，还有一个bRequest为0x22的类输出请求，那么这两个请求到底是做什么用的呢？在USB CDC协议中可以找到答案，0x21是GET_LINE_CODING的请求，而0x22是SET_CONTROL_LINE_STATE的请求。

6.7.1 GET_LINE_CODING请求

GET_LINE_CODING请求是主机获取串口属性的请求，包括波特率设置、停止位位数、校验类型以及数据位位数。该请求的建立过程数据包内容为0xA1 0x21 0x00 0x00 0x00 0x00 0x07 0x00。其中，第一字节0xA1表示发送到接口的类输入请求（不过我们的程序并没有判断它是发送到哪的）；第二字节0x21为GET_LINE_CODING请求的编码；接下来的两字节为wValue字段，值为0；再接下来的两个字节为wIndex字段，它是指明发送到哪个接口的，这里为发送到接口0；最后两字节表示请求返回数据的长度wLength，为0x0007字节。协议中给出的请求结构如表6.7.1所列，后面的Data是要由设备返回的。

表 6.7.1　GET_LINE_CODING 请求的结构

bmRequestType	bRequest	wValue	wIndex	WLength	Data
10100001B	GET_LINE_CODING	Zero	Interface	Size of Structure	Line Coding Structure

那么这个 Line Coding 的格式到底是怎样的呢？在文档中同样有定义，如表 6.7.2 所列。

表 6.7.2　Line Coding 结构体

偏移量	域	大小/字节	取　值	描　述
0	dwDTERate	4	数值	Data terminal rate, in bits per second
4	bCharFormat	1	数值	Stop bits 0：1 Stop bit 1：1.5 Stop bit 2：2 Stop bit
5	bParityType	1	数值	Parity 0：None　　3：Mark 1：Odd　　4：Space 2：Even
6	bDataBits	1	数值	Data bits(5,6,7,8,or16)

第一个域为 dwDTERate(4 字节的整数，注意大小端问题)，表示数据终端的速率，单位为 b/s，对于串口，就是我们所说的波特率，例如 9 600，115 200 等。第二个域为 bCharFormat，表示使用多少个停止位，0 为 1 个停止位，1 为 1.5 个停止位，2 为 2 个停止位。第三个域为 bParityType，表示所使用的校验方式，0 为无校验，1 为奇校验，2 为偶校验，后面两种 Mark 和 Space 很少用。第四个域为 bDataBit，表示数据位位数，可选择 5、6、7、8 或者 16。通常只使用 8 位数据位、1 位停止位，以及无奇偶校验的模式，本实例程序为了简单化处理，只支持该模式。如果要支持更多的模式，需要读者自行增加。

同样，为了避免移植时的大小端模式、对齐和填充问题，本 Line Coding 不使用结构体，而使用数组。当主机发送 GET_LINE_CODING 请求时，就将该数组的值返回；当主机发送 SET_LINE_CODING 时，就将主机发送过来的值保存在这里面，并同时按照参数设置好相关属性。

6.7.2　SERIAL_STATE 通知

本来该通知是用来返回串口状态的，但是圈圈在实际使用中，发现主机从来未发送过该请求，并且学习板上也没有相关串口的流控制引脚，因此这里并不对它进行处理，只是简单地发送一个 0 长度的数据包。如果对该请求感兴趣，可以参看 CDC 协

议中对该请求的说明。

6.7.3 SET_CONTROL_LINE_STATE 请求

该请求没有数据输出阶段，其中，wValue 字段的 D0 位表示 DTR，D1 位表示 RTS。但是板上的串口并没有这两个引脚，所以对该请求只是简单地返回一个 0 长度的状态过程数据包即可。

6.7.4 SET_LINE_CODING 请求

SET_LINE_CODING 请求用来设置串口的属性，跟 GET_LINE_CODING 是相对的，后面所使用的 Line Coding 格式也一样。该请求将在控制传输的数据过程发送 7 字节的 Line Coding 数据。但是我们之前的程序都没有处理过在数据阶段输出数据，因此在这里需要修改端点 0 输出中断函数末尾部分对普通数据的处理。为此增加一个 UsbEp0DataOut()函数来负责处理这些数据，以避免端点 0 输出中断处理函数变得十分冗长(事实上它现在已经十分冗长了，这很不好，感兴趣的读者可以考虑将内部的代码改成函数来实现，以使结构更清晰。圈圈把这个函数弄成这样，主要是为了偷懒，不想增加一大堆函数)。

在 UsbEp0DataOut()函数中，判断前面的请求是否为 SET_LINE_CODING，如果是，则读回 7 字节的数据放入到 Line Coding 中，然后根据它们设置串口的波特率。本实例仅支持修改波特率，其他参数被忽略掉，并将它们修改回固定的 1 停止位、无校验、8 数据位的格式。具体的实现代码如下：

```
/*******************************************************
函数功能:USB 端点 0 数据过程数据处理函数
入口参数:无
返    回:无
备    注:该函数用来处理 0 端点控制传输的数据或状态过程
*******************************************************/
void UsbEp0DataOut(void)
{
 //由于本程序中只有一个请求输出数据,所以可以直接使用 if 语句判断条件,
 //如果有很多请求的话,使用 if 语句就不方便了,而应该使用 switch 语句散转
 if((bmRequestType == 0x21)&&(bRequest == SET_LINE_CODING))
 {
  uint32 BitRate;
  uint8 Length;

  //读回 7 字节的 LineCoding 值
  Length = D12ReadEndpointBuffer(0,7,LineCoding);
  D12ClearBuffer();                    //清除缓冲区
```

```
    if(Length == 7)                    //如果长度正确
    {
     //从 LineCoding 计算设置的波特率
     BitRate = LineCoding[3];
     BitRate = (BitRate<<8) + LineCoding[2];
     BitRate = (BitRate<<8) + LineCoding[1];
     BitRate = (BitRate<<8) + LineCoding[0];
    #ifdef DEBUG0
     Prints("波特率设置为:");
     PrintLongInt(BitRate);
     Prints("bps\r\n");
    #endif
     //设置串口的波特率
     BitRate = UartSetBitRate(BitRate);
     //将 LineCoding 的值设置为实际的设置值
     LineCoding[0] = BitRate&0xFF;
     LineCoding[1] = (BitRate>>8)&0xFF;
     LineCoding[2] = (BitRate>>16)&0xFF;
     LineCoding[3] = (BitRate>>24)&0xFF;
     //由于只支持 1 位停止位,无校验,8 位数据位,所以固定这些数据
     LineCoding[4] = 0x00;
     LineCoding[5] = 0x00;
     LineCoding[6] = 0x08;
    }
    //返回 0 长度的状态数据包
    D12WriteEndpointBuffer(1,0,0);
   }
   else                                //其他请求的数据过程或者状态过程
   {
    D12ReadEndpointBuffer(0,16,Buffer);
    D12ClearBuffer();
   }
  }
  ////////////////////////End of function////////////////////////////
```

其中用到了 UartSetBitRate()函数,该函数负责设置串口波特率,返回值为实际设置的波特率。由于定时器溢出时间的离散性,导致了波特率的离散性,从而实际设置的波特率与期望的波特率可能有一定的偏差。通过 UartSetBitRate()函数的返回值,可以计算出期望波特率与实际波特率之间的偏差。为了获得更宽的波特率,使用了定时器 2 来作为波特率发生器。同时为了使该函数方便移植到没有定时器 2 的 51 单片机上,也保留了使用定时器 1 作为波特率的代码。在 UART.C 文件中,定义

一个宏#define USE_T2将使用定时器2作为波特率发生器，删除该宏将使用定时器1作为波特率发生器。具体的代码请参看本实例的UART.c文件。

6.7.5 实现类请求后的测试

实现了这几个类请求后，又忍不住想测试一下看看结果了……等一下，我的学习板去哪儿了？晕，原来是昨天摆在桌面上太乱，被老婆收起来了……将程序编译下载到学习板中，打开串口调试助手，上电运行。

串口显示的设置配置以及之后的调试信息如图6.7.1所示。

图6.7.1 实现类请求后的调试信息

从串口返回的信息可以看到，成功实现了SET_CONTROL_LINE_STATE和GET_LINE_CODING请求。然后再打开另外一个串口调试助手，选择虚拟串口，以9 600的波特率打开它，可以看到串口返回的调试信息(部分)，如图6.7.2所示。

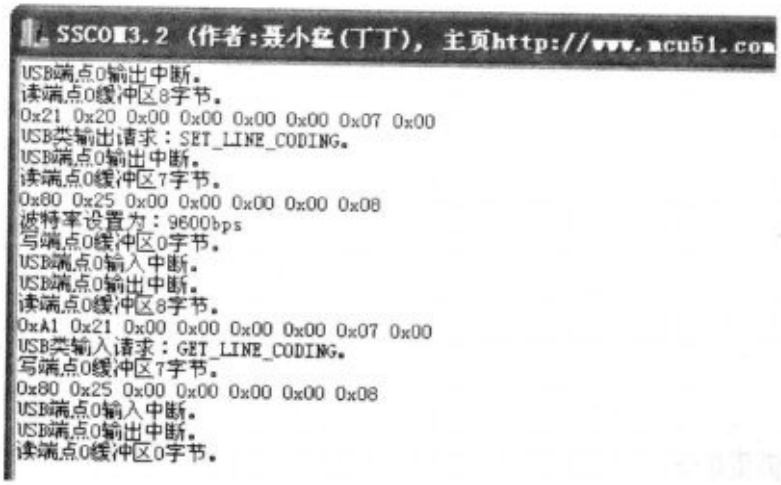

图6.7.2 使用串口调试助手打开时的部分调试信息

调试信息显示了几回GET_LINE_CODING、SET_CONTROL_LINE_STATE以及

SET_LINE_CODING 请求。为什么会发送这么多次，其原因圈圈也不清楚，这要看主机端的程序是怎么写的了。其中 SET_LINE_CODING 请求的数据阶段过程数据为 0x80 0x25 0x00 0x00 0x00 0x00 0x08，即波特率为 9600，1 位停止位，无校验，8 位数据位。

此时将新打开的串口调试助手（即自己制作的虚拟串口）显示选择为 HEX 显示，然后按一下学习板上的 KEY2，可以看到串口接收到了数据。而调试信息也显示往端点 2 发送了两次数据，每次为 5 字节，如图 6.7.3 所示。那么这些数据从哪里来的呢？还记得我们的程序是用什么改来的吗？对了，是带鼠标的键盘程序。这里还并没有修改端点 2 的输入输出处理，按 KEY2 实际上是通过端点 2 返回鼠标的报告，第一字节 0x01 为报告 ID，第二字节 0x00 为按键情况，第三字节 0xF6 为 X 轴往左移动 10 个单位（KYE2 键的功能）。此时不要往虚拟串口发送数据，因为我们还没有对端点 2 的输出中断做相关处理呢，如果发送一个数据，那么调试信息就会一直显示端点 2 输出中断。

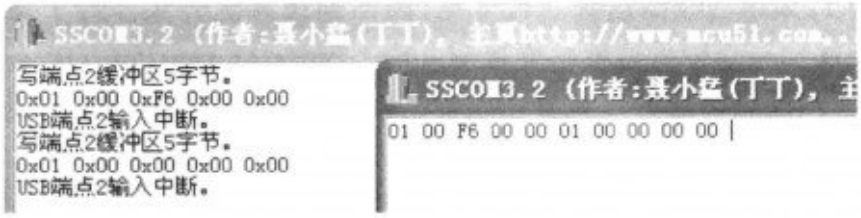

图 6.7.3　按下 KEY2 时的调试信息（左）和接收的数据（右）

6.8　对串口数据的处理

对于大部分 USB 设备的开发来说，主要的工作在枚举以及相关请求方面，真正的数据传输是比较简单的，本虚拟串口也是如此。

要实现串口数据到 USB 数据之间的互换，只要将串口接收到的数据发送到端点 2，将端点 2 接收到的数据发送到串口即可。不过，这里还存在着一个数据溢出的问题。如果串口中接收到的数据来不及通过 USB 口发送出去，就会导致数据丢失，因为在串口端是没有流控制的。对于 USB 端则没有这个问题，如果数据未处理完，D12 会自动应答 NAK，从而进行流控制。那么如何解决这个问题呢？可以开一个循环缓冲区，在串口接收中断处理函数中将数据接收下来；然后在主循环中，检测到端点 2 输入缓冲区空闲时，就将数据写入到端点 2 中。为了防止主循环长期不检查串口是否收到数据，在主循环中对输出端点 2 接收到的数据每次只发送 1 字节到串口。这样可以保证一定时间内就会查询串口是否接收到数据，并将数据写入到输入端点 2。由于端点 2 的最大包长描述为 64 字节，所以可能会在输出端点 2 一次性收到 64 字节的数据，因此需要使用一个 64 字节的缓冲区来保存这些数据。主循环每循环一

次就发送1字节数据到串口(如果有的话),当数据发送完毕后才能再次去读输出端点2的数据。考虑到8952的RAM只有256字节,串口接收的循环缓冲区也使用64字节,理论上该缓冲区越大,就可以保存越多的数据,不过在这里64字节的缓冲区已经足够了。也许你所使用的单片机内部RAM比较少(例如8951),那么可以考虑在端点描述符中将端点2的最大包长改小一些,例如16字节。这样两个缓冲区大小就可以设置为16字节了。

通过以上分析,在程序中增加两个64字节的串口数据缓冲区和USB端点2数据缓冲区(UartBuffer[64]和UsbEp2 Buffer[64]),并增加两个计数器变量(UartByteCount和UsbEp2ByteCount)。另外,先收到的数据要先发送(FIFO)。由于串口接收数据缓冲区是循环缓冲,所以还需要知道当前缓冲区中的取数据点和写入数据点,故增加两个变量(UartBufferOutputPoint和UartBufferInputPoint)。每次从缓冲区中取数据时,从UartBufferOutputPoint位置取,取完后加1,如果已经到达缓冲区末尾,则回到缓冲区开头;每次往缓冲区写数据时,写到UartBufferOutputPoint位置,写完后加1,如果已经到达缓冲区末尾,则回到缓冲区开头。对于输出端点2的数据,也需要增加一个输出位置的变量UsbEp2 BufferOutputPoint,以便知道当前发送的数据位置。注意将这些计数器和位置值在USB总线复位处理中初始化为0。

将原来程序中的端点1输出处理部分读数据及控制LED部分删除(当然不删也可以,它除了占一点代码空间外,不影响程序运行),将main函数中对按键的处理部分删除,发送鼠标报告函数SendMouseReport()和发送键盘报告函数SendKeyboardReport()也删除,将KEY.C从工程中移除,初始化按键的函数调用也删除。因为这些代码我们已经不再需要它们了。LED用不上,也删除之。由于串口要用作发送数据用,所以不能再用它来输出调试信息了,在config.h中将DEBUG0和DEBUG1的定义删除。

然后是修改串口中断处理函数,将接收中断处理部分进行修改,如下所示(它将把串口接收到的数据写入到循环缓冲区中,并增加输入位置的值以及接收字节计数器的值):

```
if(RI)                          //收到数据
 {
  RI = 0;                       //清中断请求
  //从SBUF读回1字节数据保存在缓冲区中
  UartBuffer[UartBufferInputPoint] = SBUF;
  //将输入位置下移
  UartBufferInputPoint ++ ;
  //如果已经到达缓冲区末尾,则切换到缓冲区开头
  if(UartBufferInputPoint >= BUF_LEN)
  {
   UartBufferInputPoint = 0;
```

```
    }
    //接收字节数加 1
    UartByteCount ++ ;
  }
```

然后在端点 2 输出中断处理中添加对输出数据的处理。当缓冲区 UsbEp2Buffer 中还有数据未发送时，就不读端点中的数据，直接返回。当缓冲区 UsbEp2Buffer 空后，才读回端点 2 输出的数据。修改后的中断处理函数代码如下：

```
/*****************************************************
函数功能：端点 2 输出中断处理函数
入口参数：无
返　　回：无
备　　注：无
*****************************************************/
void UsbEp2Out(void)
{
#ifdef DEBUG0
 Prints("USB 端点 2 输出中断。\r\n");
#endif
 //如果缓冲区中的数据还未通过串口发送完毕，则暂时不处理该中断，直接返回
 if(UsbEp2ByteCount!  = 0) return;
//读最后接收状态，这将清除端点 2 输出的中断标志位
 D12ReadEndpointLastStatus(4);
 //读取端点 2 的数据。返回值为实际读到的数据字节数
 UsbEp2ByteCount = D12ReadEndpointBuffer(4,BUF_LEN,UsbEp2Buffer);
 //清除端点缓冲区
 D12ClearBuffer();
//输出位置设为 0
 UsbEp2BufferOutputPoint = 0;
}
//////////////////////End of function////////////////////////////
```

至此，两方面接收数据的处理都完成了，剩下来的就是对数据的发送处理。在 main 函数中的设置配置判断后，增加如下代码：

```
  if(ConfigValue!  = 0)        //如果已经设置为非 0 的配置，则可以返回和发送串口数据
  {
   if(Ep2InIsBusy == 0)        //如果端点 2 空闲，则发送串口数据到端点 2
   {
    SendUartDataToEp2();       //调用函数将缓冲区数据发送到端点 2
   }
```

```
    if(UsbEp2ByteCount!  = 0)  //端点 2 接收缓冲区中还有数据未发送,则发送到串口
    {
     //发送 1 字节到串口
     UartPutChar(UsbEp2Buffer[UsbEp2BufferOutputPoint]);
     UsbEp2BufferOutputPoint ++ ;  //发送位置后移 1
     UsbEp2ByteCount -- ;        //计数值减 1
    }
  }
```

由上面的代码可以判断,当设置为非 0 配置后,就可以返回和发送串口数据。如果端点 2 处于空闲状态,那么就调用函数 SendUartDataToEp2()将缓冲区的数据发送到端点 2。如果端点 2 接收缓冲区中还有数据未发送完毕,则调用 UartPutChar()函数将数据发送到串口,并修改相应的计数值和取数据的位置。

发送串口数据到端点 2 的函数 SendUartDataToEp2()代码如下所示。由于在串口中断处理中修改字节计数值,为了防止在主程序中取值和修改时不同步,暂时关闭串口中断再做相关处理。由于使用的是循环缓冲,如果数据跨越缓冲区边界(不连续),就不好发送了。这里使用了一个小技巧,就是检查需要发送的数据如果跨越边界,就先只发送前面连续的一部分,剩余的部分留待下一次发送。

```
/*************************************************************
函数功能:将串口缓冲区中的数据发送到端点 2 的函数
入口参数:无
返　　回:无
备　　注:无
*************************************************************/
void SendUartDataToEp2(void)
{
 uint8 Len;

 //暂时禁止串行中断,防止 UartByteCount 在中断中修改而导致不同步
 ES = 0;
 //将串口缓冲区接收到的字节数复制出来
 Len = UartByteCount;
 //检查长度是否为 0,如果没有收到数据,则不需要处理,直接返回
 if(Len == 0)
 {
  ES = 1;                    //记得打开串口中断
  return;
 }
 //检查 Len 字节个数据是否跨越了缓冲区边界,如果跨越了,那么本次只发送跨越边界之前的数据
 //剩余的数据留待下次发送;否则,可以一次发送全部
```

```
if((Len + UartBufferOutputPoint)>BUF_LEN)
{
 Len = BUF_LEN - UartBufferOutputPoint;
}
//修改缓冲区数据字节数
UartByteCount -= Len;
//至此可以打开串口中断了
ES = 1;
//将数据写入到端点 2 输入缓冲区
D12WriteEndpointBuffer(5,Len,UartBuffer + UartBufferOutputPoint);
//修改输出数据的位置
UartBufferOutputPoint += Len;
//如果已经到达缓冲区末尾,则设置回开头
if(UartBufferOutputPoint>= BUF_LEN)
{
 UartBufferOutputPoint = 0;
}
//设置端点 2 输入忙
Ep2InIsBusy = 1;
}
//////////////////////End of function//////////////////////////
```

将上面的程序再次编译,并下载后运行,这时已经实现双向收发功能了。具体怎么测试的呢?首先学习板连接到计算机真正的串口上,用串口调试助手打开这个真正的串口;然后连接学习板的USB线,这时会增加一个虚拟串口,使用另外一个串口调试助手打开它。此时虚拟串口发送的数据将到达学习板的串口上,从而被真正的串口接收到,显示在调试助手上。而从真正的串口发送的数据,将到达学习板的串口上,然后再被学习板发到USB上,最终到达虚拟串口,显示在打开虚拟串口的调试助手上。这好比两个串口A和B连在了一起,A发送数据B收到,B发送数据A收到。

但是经过快速测试(自动发送)后,发现从虚拟串口发送数据出去时,会延迟一次发送。例如,先发送字符X,再发送字符Y后,才显示X,需要另外再发送一次,才会显示Y。这是什么原因导致的呢?这是因为D12的端点2有个双缓冲机制。在我们的程序中,并没有检测双缓冲区是否都满了,而是简单地清除中断。如果快速发送时,程序还未处理D12端点2中的数据,主机又发送了输出数据,那么D12就会将接收到的数据放在另一个缓冲区中。此时D12的两个缓冲区中都含有有效的数据,但是我们的程序只读了一个缓冲区的数据就清除了中断,所以剩余的数据就没有读取。当主机下一次发送数据时,才引起一次新的中断,这时才读出前一次的数据,而本次发送的数据又未被读到,从而总是有一个数据包延迟;而从虚拟串口返回数据则没有这个问题,因为只要缓冲区中有数据,主机请求数据输入时就会取走。我们的程序每

写一次就设置端点忙了，事实上，只写一次数据是不忙的，因为双缓冲区中还有另外一个缓冲区空闲。像这样，输入端点2实际上是当作单缓冲区来使用了，浪费了一个缓冲区没用，当然如果处理速度够快，这也不会带来什么坏处。

那么怎么解决这个问题呢？很简单，在接收时，读取完数据包后，如果发现双缓冲区中某个缓冲依然是满的，那么就先不清除中断，直接返回；发送时也类似，如果两个缓冲区都满了才设置端点忙。那如何获取端点的双缓冲是否满的信息呢？这要用到D12的另一个命令：读端点状态(read endpoint status)，命令代码为0x80～0x85，每个端点对应一个。使用该命令可以从D12中读回1字节，其中的D5、D6位用来表示端点的状态，某位为1时，表示对应的端点处于满状态。只要将读回的结果跟0x60作按"位与"操作，如果结果为0x60，那么说明两个缓冲都满了。实现该命令的函数代码如下：

```
/*******************************************************
函数功能：读取 D12 端点状态函数
入口参数：Endp  端点号
返    回：端点状态寄存器的值
备    注：无
*******************************************************/
uint8 D12ReadEndpointStatus(uint8 Endp)
{
 D12WriteCommand(0x80 + Endp);          //读取端点状态命令
 return D12ReadByte();
}
//////////////////////End of function//////////////////////////////
```

在端点2输出中断处理中，将原来的清除端点缓冲区的代码删除，并在函数返回前增加如下代码：

```
//当两个缓冲区中都没有数据时，才能清除中断标志
if(! (D12ReadEndpointStatus(4)&0x60))
{
//读最后发送状态，这将清除端点 2 输入的中断标志位
 D12ReadEndpointLastStatus(4);
}
```

在SendUartDataToEp2()函数中，修改设置端点忙的语句如下：

```
//只有两个缓冲区都满时，才设置端点 2 输入忙
if((D12ReadEndpointStatus(5)&0x60) == 0x60)
{
 Ep2InIsBusy = 1;
}
```

然后再对程序编译，并下载到学习板中运行。使用两个串口调试助手对发，测试结果一切正常。圈圈使用该串口调试助手所支持的最高波特率 256 000 b/s 测试，发送了 100 多 KB 的数据，没有一个丢失。

6.9　安装驱动用的 inf 文件

虽然 Winows 2000/XP 自带了这个 CDC 类的驱动，但是要安装本设备，还需要提供一个 inf 文件，否则系统会提示找不到驱动。本 inf 文件也不是圈圈自己写的，而是在网上下载，然后根据自己的需要来修改的。说实话，圈圈对这个 inf 文件也不太熟悉，只能拿别人现成的来改改，自己重新写有些难度。在这里，仅简单地介绍这个 inf 文件中的内容以及如何去修改它们，一般的读者掌握到这一层就够了。inf 文件中的内容如下：

```
;--------------------inf 文件开始----------------------
; Windows USB CDC Setup File
; Copyright (c) 2000 Microsoft Corporation
; Copyright (c) 2006 Recursion Co., Ltd.

[Version]
Signature = "$Windows NT$"

;类选择为端口
Class = Ports

;使用的安装类 GUID。该 GUID 类的设备为"端口（COM 和 LPT）"，可以在注册表
;HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class 中找到它
;在设备管理器中我们可以看到最后生成的设备被放在了"端口（COM 和 LPT）"之下，
;并且打开设备的属性可以看到设备的类型为"端口（COM 和 LPT）"
ClassGuid = {4D36E978 - E325 - 11CE - BFC1 - 08002BE10318}

;驱动程序的提供商，它将显示在设备属性的驱动程序标签页中的驱动程序提供商中
;驱动程序提供商名称 COMPANY 在该文件最后被定义，它是一个字符串
Provider = %COMPANY%

;使用 layout.inf 文件
LayoutFile = layout.inf

;驱动程序的日期和版本号。驱动程序安装器据此来判断驱动程序的新旧
;它们会显示在设备属性的驱动程序标签页中
DriverVer = 08/04/2008,1.0.0.1

[Manufacturer]
;制造商名称。会在设备属性窗口的常规标签的制造商中显示，MFGNAME 在该文件最后被定义
%MFGNAME%  = ManufName

[DestinationDirs]
;目标文件夹的位置。12 为 system32 目录
```

```
DefaultDestDir = 12

[ManufName]
;这里设置显示设备的名称以及匹配的 ID 号
;Modem3 是在后面定义的一个字符串,修改它可以显示不同的设备名称
;后面的 USB\VID_8888&PID_0007 表示该驱动所匹配的设备,需要根据自己的设备设置
;我们的设备 VID 为 8888,PID 为 0007
%Modem3% = Modem3, USB\VID_8888&PID_0007

;-----------------------------------------------------------
;  Windows 2000/XP Sections
;-----------------------------------------------------------

[Modem3.nt]
CopyFiles = USBModemCopyFileSection
AddReg = Modem3.nt.AddReg

[USBModemCopyFileSection]
;需要复制 usbser.sys 文件,该文件是 Windows 2000/XP 自带的
usbser.sys,,,0x20

[Modem3.nt.AddReg]
;增加注册表项
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"

[Modem3.nt.Services]
;增加驱动服务
AddService = usbser, 0x00000002, DriverService

[DriverService]
DisplayName = %SERVICE%
ServiceType = 1
StartType = 3
ErrorControl = 1
ServiceBinary = %12%\usbser.sys

;-----------------------------------------------------------
;  String Definitions
;-----------------------------------------------------------
;这些是根据自己需要定义的字符串,可以按照自己的需要来修改它们,
;它们只是一些供显示用的字符串,没有实际的意义,用户可以随便修改
[Strings]
;公司名称
COMPANY = "电脑圈圈的家当"
;制造商名称
MFGNAME = "电脑圈圈"
;设备名称,它将显示在设备管理器中
Modem3 = "电脑圈圈做的 USB 转串口"
```

```
;服务名称
SERVICE = "USB2UART Driver"
;-------------------------inf 文件结束----------------------
```

在 inf 文件中，被分成一个个节，每个节用[ ]扩起来。以分号开头的表示该行为注释。为了方便大家理解，圈圈在该文件中增加了一些注释(中文部分)。一般来说，需要自己修改的部分只有 VID、PID 以及最后定义的字符串部分。VID 和 PID 必须要跟自己的设备 ID 一致，否则将提示找不到驱动。本实验的 VID 为 0x8888，PID 为 0x0007，该设备是连接在 USB 总线上的，所以在 inf 文件中指定的匹配 ID 就是 USB\VID_8888&PID_0007。

至于后面的字符串，只是显示给使用者看的，不改也无所谓。但是显示别人的东西总是不好，例如你可以把该设备在设备管理中显示的名字改得炫一点(如“圈圈的超级串口”或者“我的垃圾串口”等)。制造商和驱动程序提供商也是可以随意改的，圈圈懒得敲那么多字了，还是用图片来说明吧，如图 6.9.1～图 6.9.3 所示。

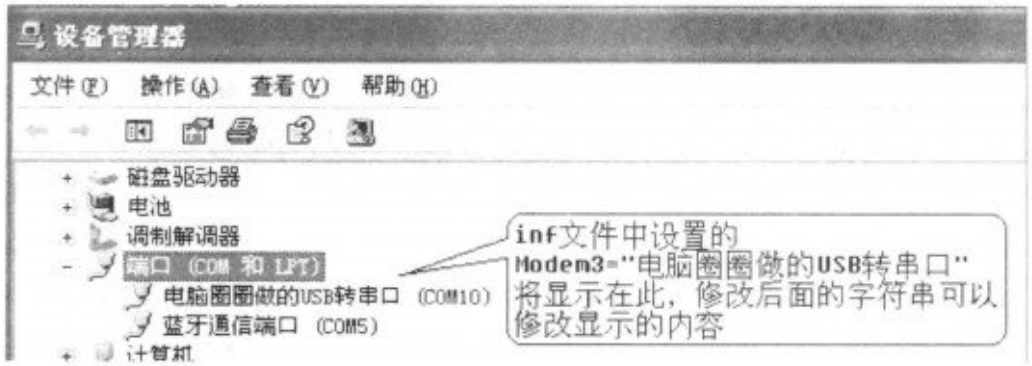

图 6.9.1　设备管理器中显示的设备名称

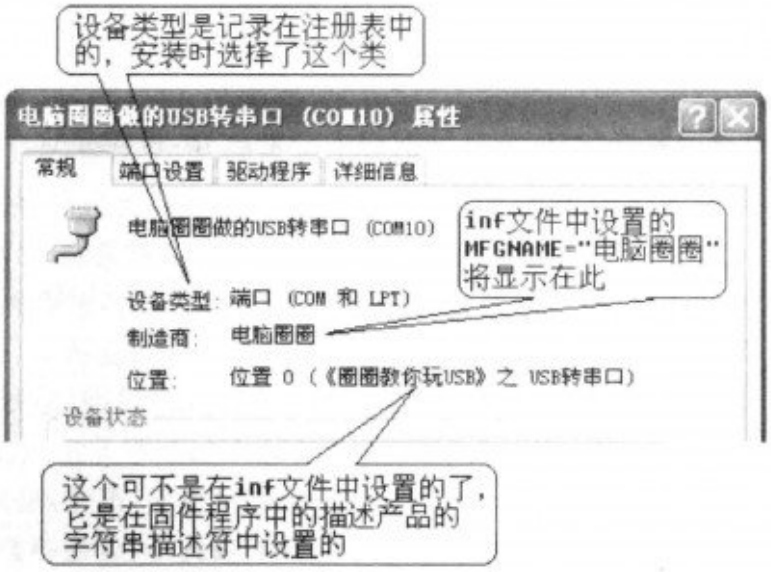

图 6.9.2　设备属性中的常规选项卡

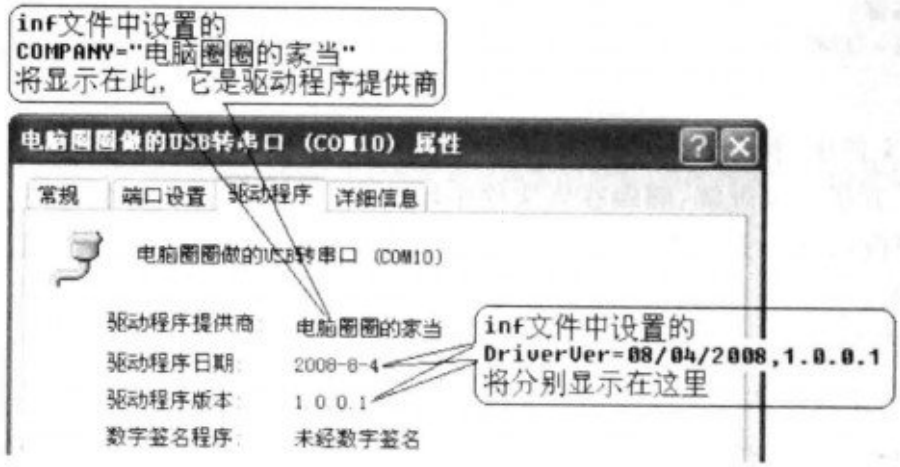

图 6.9.3　设备属性中的驱动程序选项卡

6.10　本章小结

本章所设计的 USB 转串口还是很好用的，像圈圈现在所使用的计算机就没有串口，那么那些通过串口显示的调试信息是怎么来的呢？对了，就是使用本学习板实现的 USB 转串口。圈圈手头有 2 块 USB 学习板，一块做 USB 转串口用，另一块做实验用。用这个虚拟串口还可以通过串口给 STC 单片机下载程序，而有些市场上买的 USB 转串口线则不行。不过需要注意的是，使用该方法增加的串口是 WDM 驱动产生的，而非真正的物理端口的那种串口，所以一些直接访问 PC 端口的老串口软件或许不能使用该串口。开发应用程序时也要注意，不要直接使用古老的端口号方式来访问串口，而应该使用较新的 WMD 方式访问。文中只是挑了一些必要的请求来实现，如果要深入了解，还需要读者自行研究 USB CDC 的协议文档。圈圈曾尝试在同一个设备上实现两个串口，但是不管是使用两套相同的接口还是仅使用两个数据接口，多次试验都以蓝屏告终，最终只好很无奈地放弃，可能是这个驱动并不支持这样的方式。