【博客藏经阁丛书】

圈圈教你玩USB

刘荣 编著
[网名 电脑圈圈]

北京航空航天大学出版社

CD-ROM INCLUDED + PCB

策划编辑：胡晓柏
封面设计：runsign

教你玩USB

内容特色

本书是作者专门为广大USB入门和提高者而编写的USB入门级教材。作者在经过长期、大量的USB学习和开发后，按照自己的理解以及汉语的思维方式编写了本书，同时将USB学习和开发过程中总结的一些经验和方法溶入了书中。

本书的特点是以实例的方式，一步步讲解USB设备与驱动程序和应用程序开发的详细过程及步骤，同时在光盘中附带了完整的源代码（C/C++语言）。为了照顾初学者，书中语句尽量做到通俗易懂，少用专业词汇。

读者对象

USB设备与驱动设计入门和提高者，以及所有对圈圈的支持者。

圈圈的防伪LOGO

随书光盘内含学习板的电路图以及所有实验的完整源代码包，并包含原创的《21ic电工之歌.mp3》：

21ic电工之歌

曲：电脑圈圈　词：程序匠人

——谨以此歌献给21ic的网友暨流浪的电工们

上架建议：单片机/嵌入式系统

ISBN 978-7-81124-600-1

9 787811 246001 >

定价：39.00元(含光盘+PCB板)

TP334
31

TP334
31

【博客藏经阁丛书】

附光盘

圈圈教你玩 USB

刘　荣[网名　电脑圈圈]　编著

北京航空航天大学出版社

内 容 简 介

通过 U 盘、USB 鼠标、USB 键盘、USB MIDI 键盘、USB 转串口、自定义的 USB HID 设备和自定义的 USB 设备等几个具体的 USB 例子，一步步讲解 USB 设备及驱动程序和应用程序开发的详细过程和步骤。最后两章介绍 USB WDM 驱动开发，并给出一个简单的 USB 驱动和 USB 上层过滤驱动的实例。

本书附带的光盘中有 USB 学习板的电路图以及所有实验的完整源代码包(C/C++语言)。

本书的读者对象主要是 USB 设备与驱动设计的初学者和提高者，以及所有对圈圈的支持者。

图书在版编目(CIP)数据

圈圈教你玩 USB/刘荣编著. —北京 :北京航空航天大学出版社，2009.1

ISBN 978-7-81124-600-1

Ⅰ. 圈… Ⅱ. 刘… Ⅲ. 电子计算机—接口 Ⅳ. TP334

中国版本图书馆 CIP 数据核字(2008)第 159685 号

© 2009，北京航空航天大学出版社，版权所有。

未经本书出版者书面许可，任何单位和个人不得以任何形式或手段复制本书及其所附光盘内容。侵权必究。

圈圈教你玩 USB

刘 荣[网名 电脑圈圈] 编著

责任编辑 张冀青

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(100191) 发行部电话:010-82317024 传真:010-82328026

http://www.buaapress.com.cn E-mail:emsbook@gmail.com

涿州市新华印刷有限公司印装 各地书店经销

*

开本:787 mm×960 mm 1/16 印张:20.5 字数:459 千字

2009 年 1 月第 1 版 2009 年 1 月第 1 次印刷 印数:5 000 册

ISBN 978-7-81124-600-1 定价:39.00 元(含光盘+PCB 板)

前言

USB 作为一种计算机总线技术，在如今的个人计算机上已经是必不可少的了。看看我们身边的计算机外部设备，有多少是通过 USB 口与计算机连接的？鼠标、键盘、游戏手柄、打印机、扫描仪、MP3、数码相机、U 盘、移动硬盘及移动光驱等，另外，还有一些我们不太常见的 USB 设备，例如一些具有 USB 口的仪表仪器、开发用的调试器、烧录机、USB 网卡、USB 耳机、USB 话筒及 USB 电话，甚至一些移动电话（手机）也具备 USB 口。总之，只要是与计算机通信的外部设备，似乎都可以用 USB 来连接，这足见 USB 之强大。

USB 之所以使用得如此广泛，是因为它具有连接简单、速度快、可扩展性强、支持热插拔操作和标准统一等特点。由于 USB 协议详细地规定了各种参数以及数据结构、格式，因而使得各厂生产出来的设备都能够很好地相互兼容。不过，这却给 USB 设备开发者带来了一些麻烦。USB 设备开发者首先要很清楚 USB 协议才能开发出符合协议的 USB 设备，然而 USB 协议本身是一个比较复杂、庞大的系统，再加上众多的子类协议，使得很多设计者望而生畏，或者时间上不允许。所以有一些小公司将 USB 部分外包或者使用别人现成的 USB 模块来加快开发进度。如果你会开发 USB 相关设备的话，可能会获得不少机会哦☺。不过，虽然圈圈（就是笔者我啦，读者一定要牢记这点，不然把你弄晕了我可不负责）会一点 USB，但目前还是穷光蛋一个☹。当然，你也可以学圈圈这样，写本书出来忽悠忽悠。

虽然 USB 这么好用，但是教你如何设计 USB 设备的书在市面上却是少见，大部分都是对 USB 协议翻译，只有少量的内容是实际开发的内容和代码。圈圈从 2003 年年底（大二）开始学习 USB，花了约一年的时间（主要是利用课余时间）阅读了一些 USB 书籍和文档后，才真正开始动手做自己的第一个 USB 设备：一个基于 AT89C52＋PDIUSBD12 的假 U 盘。圈圈自认自己资质不差，却让一个小小的 USB 困扰了这么久，我想除了自己的原因之外，还跟 USB 协议本身的复杂性和没有一些好的书籍有很大关系。在整个 USB 学习过程中主要都是靠自己慢慢摸索，走了一些弯路。现在回过头再来看看，如果当初能有一个整体的认识，按照合理的步骤来操作，就不会走这么多弯路了。因此圈圈意识到，迫切需要写一本能让 USB 初学者少走弯路、快速入门和上手的书籍。圈圈写这本书，是真心地想要更多的人能够学会 USB，会开发 USB 设备。本书并不是对 USB 协议简单地进行翻译，而是尽量用圈圈自己的语言来进行描述。所以，本书的一大特点就是语言有些口语化，逻辑不是太严密，有些语句也可能存在

着错误。但我想这样读起来也许会更轻松些，太严谨、太严密的描述，可能会显得有些乏味。另外，有些地方可能会存在一些重复性描述，主要是想方便理解和加深印象。

本书通过U盘、USB鼠标、USB键盘、USB MIDI键盘、USB转串口、自定义的USB HID设备和自定义的USB设备等几个具体的USB例子，来介绍USB设备设计的具体流程。提到自定义的USB设备，就不可避免地要提到自己开发USB的驱动程序。本书最后两章简单地介绍了USB WDM驱动的开发，并给出一个简单的USB驱动和一个USB下层过滤驱动的实例。驱动程序开发更深层次的研究不属本书范畴，留给读者自行深究。另外，本书也包括了一些圈圈在USB的学习和实际设计过程中总结的一些电路设计和程序设计的经验及方法，希望能够帮助大家更好地学习和设计USB系统。

本书主要面向的读者是刚接触USB开发的电子设计人员，需要有一定的电子技术、计算机技术和C语言基础。对于已经熟悉USB的读者意义不大，应以官方的数据手册和文档为主。本书为了让读者快速入门，可能会对一些模型做简化处理，也可能有一些地方是圈圈本身理解偏差或者错误的，当本书与官方的协议和文档不一致或者冲突时，以官方文档为准。

书中的实例，都是基于圈圈设计的USB学习板之上的。本书附带的光盘中有该学习板的电路图(pdf格式)以及所有实验的完整源代码包，电路图和实验的代码包也可以到圈圈的USB小组或者博客中下载。

作为一个人或一本书，出错之处在所难免，如果大家发现有错误的地方，请告诉圈圈一声。你可以在圈圈的USB专区里给圈圈留言，也可以去个人博客里留言。圈圈的个人博客地址：http://blog.ednchina.com/computer00/或http://computer00.21ic.org。注意是电脑圈圈，是两个数字0，而不是字母O。如果你记不住这些地址也没关系，直接去网上搜索“电脑圈圈的USB专区”或者“电脑圈圈”，也可找到它们。

在此感谢北京航空航天大学出版社嵌入式系统事业部主任胡晓柏先生对出版本书的关心和支持；感谢同学以及网友对本书的建议和支持；感谢家人对我的支持和理解，尤其是要感谢老婆，为了写书，少了很多陪她的时间，谢谢她的支持、理解和爱；同时也要感谢购买本书的读者——你。

电脑圈圈

2009年1月

于广州

目 录

第 1 章　USB 概述及协议基础

1.1　USB 是什么 …… 1
1.2　USB 的特点 …… 1
1.3　USB 的拓扑结构 …… 2
1.4　USB 的电气特性 …… 5
1.5　USB 的线缆、插头及插座 …… 6
1.6　USB 的插入检测机制 …… 8
1.7　USB 的描述符及其之间的关系 …… 10
1.8　USB 设备的枚举过程 …… 11
1.9　USB 的包结构及传输过程 …… 13
　1.9.1　USB 的包结构及包的分类 …… 13
　1.9.2　令牌包 …… 15
　1.9.3　数据包 …… 16
　1.9.4　握手包 …… 17
　1.9.5　特殊包 …… 17
　1.9.6　如何处理数据包 …… 18
1.10　USB 的四种传输类型 …… 19
　1.10.1　USB 事务 …… 19
　1.10.2　批量传输 …… 19
　1.10.3　中断传输 …… 21
　1.10.4　等时传输 …… 22
　1.10.5　控制传输 …… 22
　1.10.6　端点类型与传输类型的关系 …… 24

1.10.7 传输类型与端点支持的最大包长 …… 24
1.11 本章小结 …… 24

第 2 章 硬件系统设计

2.1 方案及芯片的选定 …… 25
2.2 D12 引脚功能说明 …… 26
2.3 D12 与 AT89S52 的连接 …… 30
2.4 串口部分电路 …… 32
2.5 按键部分 …… 33
2.6 指示灯部分 …… 34
2.7 IDE 接口部分 …… 34
2.8 单片机部分 …… 35
2.9 元件安装 …… 35
2.10 电路调试 …… 38
2.11 测试程序的编写和调试 …… 39
2.11.1 建立一个工程 …… 40
2.11.2 为工程添加源文件 …… 41
2.11.3 KEIL 工具栏及仿真介绍 …… 43
2.11.4 按键驱动的编写 …… 46
2.11.5 串口驱动的编写 …… 52
2.11.6 PDIUSBD12 读写函数及读 ID 的实现 …… 57
2.12 本章小结 …… 62

第 3 章 USB 鼠标的实现

3.1 USB 鼠标工程的建立 …… 63
3.2 USB 的断开与连接 …… 63
3.3 USB 中断的处理 …… 67
3.4 读取从主机发送到端点 0 的数据 …… 68
3.5 USB 标准请求 …… 75
3.5.1 USB 标准设备请求的结构 …… 75
3.5.2 GET_DESCRIPTOR 请求 …… 77
3.5.3 SET_ADDRESS 请求 …… 78
3.5.4 SET_CONFIGURATION 请求 …… 78
3.6 设备描述符的实现 …… 79

3.7 设备描述符的返回 …… 83
3.8 设置地址请求的处理 …… 93
3.9 配置描述符集合的结构 …… 95
3.9.1 配置描述符的结构 …… 95
3.9.2 接口描述符的结构 …… 96
3.9.3 端点描述符的结构 …… 97
3.9.4 HID 描述符的结构 …… 98
3.10 配置描述符集合的实现以及返回 …… 99
3.11 字符串及语言 ID 请求的实现 …… 103
3.12 设置配置请求的实现 …… 109
3.13 报告描述符的结构及实现 …… 112
3.14 报告的返回 …… 118
3.15 Bus Hound 工具的简介 …… 122
3.16 本章小结 …… 124

第 4 章 USB 键盘的实现

4.1 USB 键盘工程的建立 …… 125
4.2 设备描述符的实现 …… 125
4.3 配置描述符集合的实现 …… 126
4.3.1 配置描述符 …… 126
4.3.2 接口描述符 …… 127
4.3.3 HID 描述符 …… 127
4.3.4 端点描述符 …… 127
4.4 字符串描述符 …… 130
4.5 报告描述符 …… 130
4.6 输入和输出报告的实现 …… 133
4.7 USB 键盘实例的测试 …… 136
4.8 再谈 USB HID 的报告描述符 …… 138
4.9 带鼠标功能的 USB 键盘(方法一) …… 140
4.10 带鼠标功能的 USB 键盘(方法二) …… 146
4.11 多媒体 USB 键盘 …… 154
4.12 本章小结 …… 160

第 5 章 用户自定义的 USB HID 设备

5.1 MyUsbHid 工程的建立 …… 161
5.2 描述符的修改 …… 161
5.3 报告的实现 …… 163
5.4 对用户自定义的 USB HID 设备的访问 …… 165
5.5 访问 HID 设备时所用到的相关函数 …… 166
5.5.1 获取 HID 设备的接口类 GUID 的函数 …… 166
5.5.2 获取指定类的所有设备信息集合的函数 …… 166
5.5.3 从设备信息集合中获取一个设备接口信息的函数 …… 167
5.5.4 获取指定设备接口详细信息的函数 …… 168
5.5.5 打开设备的函数 …… 169
5.5.6 获取 HID 设备属性的函数 …… 170
5.5.7 从设备读取数据的函数 …… 170
5.5.8 往设备写数据的函数 …… 171
5.5.9 通过控制端点 0 读取报告的函数 …… 171
5.5.10 通过控制端点 0 发送报告的函数 …… 171
5.5.11 关闭句柄的函数 …… 172
5.5.12 需要包含的库文件 …… 172
5.6 访问 USB HID 设备的上位机软件的实现 …… 172
5.6.1 上位机程序编写的思路 …… 172
5.6.2 查找及打开 HID 设备的代码 …… 173
5.6.3 读输入报告线程的代码 …… 178
5.6.4 写输出报告的代码(发送 LED 的状态) …… 181
5.6.5 写输出报告线程的代码 …… 183
5.6.6 线程的创建以及设备插拔事件的注册 …… 184
5.6.7 对设备状态改变事件的处理 …… 186
5.7 软件界面以及使用方法 …… 187
5.8 本章小结 …… 188

第 6 章 USB 转串口

6.1 串口家族历史 …… 189
6.2 串口接头的引脚分布及功能 …… 189
6.3 USB 转串口的实现方法 …… 190
6.4 设备描述符 …… 191
6.5 字符串描述符 …… 192

6.6 配置描述符集合 …… 192
6.6.1 配置描述符 …… 193
6.6.2 CDC 接口描述符 …… 193
6.6.3 类特殊接口描述符——功能描述符 …… 193
6.6.4 接口 0(CDC 接口)的端点描述符 …… 196
6.6.5 数据类接口的接口描述符 …… 196
6.6.6 接口 1(数据类接口)的端点描述符 …… 197
6.6.7 修改好描述符后的测试 …… 198
6.7 类请求的实现 …… 200
6.7.1 GET_LINE_CODING 请求 …… 200
6.7.2 SERIAL_STATE 通知 …… 201
6.7.3 SET_CONTROL_LINE_STATE 请求 …… 202
6.7.4 SET_LINE_CODING 请求 …… 202
6.7.5 实现类请求后的测试 …… 204
6.8 对串口数据的处理 …… 205
6.9 安装驱动用的 inf 文件 …… 211
6.10 本章小结 …… 214

第 7 章 USB MIDI 键盘

7.1 MIDI 简介 …… 216
7.2 MIDI 的工作原理 …… 217
7.3 USB MIDI 设备的数据流模型 …… 217
7.4 设备描述符 …… 218
7.5 配置描述符集合 …… 218
7.5.1 配置描述符 …… 218
7.5.2 音频控制接口描述符 …… 218
7.5.3 类特殊音频控制接口描述符 …… 219
7.5.4 MIDI 流接口描述符 …… 220
7.5.5 类特殊 MIDI 流接口描述符 …… 220
7.5.6 端点描述符和类特殊端点描述符 …… 225
7.5.7 字符串描述符 …… 227
7.6 修改好描述符后的测试 …… 227
7.7 USB MIDI 键盘的数据返回 …… 228
7.8 USB MIDI 键盘的使用 …… 230
7.9 单片机自动弹奏的实现 …… 232
7.10 本章小结 …… 233

第 8 章 U 盘

8.1 USB 大容量存储设备 …… 234
8.2 设备描述符 …… 234
8.3 字符串描述符 …… 235
8.4 配置描述符集合 …… 235
8.4.1 配置描述符 …… 235
8.4.2 接口描述符 …… 235
8.4.3 端点描述符 …… 236
8.5 测 试 …… 236
8.6 类特殊请求 …… 237
8.6.1 Get Max LUN 请求 …… 237
8.6.2 Bulk-Only Mass Storage Reset 请求 …… 238
8.7 仅批量传输协议的数据流模型 …… 239
8.7.1 命令块封包 CBW 的结构 …… 239
8.7.2 命令状态封包 CSW 的结构 …… 240
8.7.3 对批量数据的处理 …… 240
8.8 SCSI 命令集和 UFI 命令集 …… 241
8.8.1 查询命令 INQUIRY …… 241
8.8.2 读格式化容量命令 READ FORMAT CAPACITIES …… 243
8.8.3 读容量命令 READ CAPACITY …… 244
8.8.4 READ(10)命令 …… 245
8.8.5 WRITE(10)命令 …… 246
8.8.6 REQUEST SENSE 命令 …… 247
8.8.7 TEST UNIT READY 命令 …… 248
8.9 FAT 文件系统 …… 248
8.9.1 关于 DBR …… 249
8.9.2 关于 FAT 表 …… 251
8.9.3 关于目录项 …… 252
8.10 模拟一个 FAT16 文件系统 …… 253
8.11 实验结果 …… 254
8.12 IDE 转 USB 的实现 …… 256
8.13 本章小结 …… 257

第 9 章 自定义 USB 设备及驱动开发

9.1 用户自定义 USB 设备 …… 258

9.1.1 设备描述符 …… 258
9.1.2 配置描述符集合 …… 258
9.1.3 字符串描述符 …… 259
9.1.4 数据的处理 …… 259
9.2 驱动程序开发简介 …… 259
9.3 WDM 驱动开发编程环境的建立 …… 259
9.4 创建一个 USB WDM 驱动程序 …… 262
9.5 对工程的编译 …… 270
9.6 关于 inf 文件 …… 272
9.7 驱动程序的修改 …… 272
9.7.1 Read(KIrp I)函数 …… 273
9.7.2 Write(KIrp I)函数 …… 276
9.7.3 EP1_READ_Handler(KIrp I)函数 …… 277
9.7.4 EP1_WRITE_Handler(KIrp I)函数 …… 279
9.7.5 EP2_READ_Handler(KIrp I)函数 …… 280
9.7.6 EP2_WRITE_Handler(KIrp I)函数 …… 280
9.8 驱动的安装及安装后的信息 …… 280
9.9 应用程序对驱动的访问 …… 284
9.10 测试软件的使用 …… 286
9.11 本章小结 …… 287

第 10 章 USB 过滤驱动开发

10.1 过滤驱动简介 …… 289
10.2 使用 DS 创建一个下层过滤驱动 …… 290
10.3 过滤驱动代码的修改 …… 294
10.4 过滤驱动的安装 …… 300
10.5 过滤驱动的卸载 …… 302
10.6 驱动程序测试 …… 303
10.7 本章小结 …… 306
附 录 第 3 章实例的完整调试信息 …… 307
参考文献 …… 314
后 记 …… 315

第1章 USB概述及协议基础

本章首先概要地介绍一下USB系统，包括USB的出现、特点、结构及一些基本概念等，让USB设计者对USB系统有一个总体上的认识。

1.1 USB是什么

USB是什么呢？一种说法USB是You SB的意思。另一种说法是USB其实是美国的弟弟，因为美国叫USA，USB当然是他的弟弟了（由此看出，美国叫USA也不是随便叫的，而是经过一番深思熟虑的，不管是USB、USC还是US×，都得做他的小弟了）。

那么USB到底是什么呢？其实USB是通用串行总线（Universal Serial Bus）的缩写，它已经有10多年的历史了。它的出现主要是为了简化个人计算机与外围设备的连接，增加易用性。在比较老的个人计算机中，基本上是要关掉计算机（往往还要打开机箱）之后才能连接或者更换设备，连接好之后还要在硬件上分配资源，然后重新启动计算机设备才能正常工作。这个过程是非常不方便的，如果设备能够在计算机运行过程中随意地接入，并且立刻就能正常投入工作，那么这样的特性叫做即插即用PnP（Plug and Play）。这一特性将使计算机变得更易用、更大众化。USB正是为解决这个问题而诞生的，它支持热插拔，并且是即插即用的；另外，它还具有很强的可扩展性，速度也很快。现在，USB设备已经十分普及，只要跟计算机打过交道的人，几乎都知道USB。

USB协议出现过的版本有USB1.0、USB1.1、USB2.0等。由于USB是主从模式的结构，设备与设备之间、主机与主机之间不能互连。为了解决这个问题，扩大USB的应用范围，又出现了USB OTG（On The Go）。USB OTG的做法是：同一个设备，在不同的场合下可以在主机与从机之间切换。

1.2 USB的特点

在USB1.0和USB1.1版本中，只支持1.5 Mb/s的低速（low-speed）模式和12 Mb/s的全速（full-speed）模式。在USB2.0中，又加入了速度更快（480 Mb/s）的高速（high-speed）模

式。目前，USB3.0协议正在制订当中，据说速度会比USB2.0的高速模式还要快10倍，即达到5 Gb/s左右。

值得注意的是，USB2.0并不是高速设备的代名词，因为USB2.0协议对设备的高速模式并不是强制的，而是可选的。例如PDIUSBD12，它是符合USB2.0协议的，但是不支持高速模式，只支持12 Mb/s的全速模式。因此，在选择USB芯片时要注意，不要光看到支持USB2.0协议就认为一定具有高速模式。作为一个专业的USB设计人员，一定要搞清楚这个关系。

USB具有很多优点，例如即插即用PnP，容易使用，方便携带，传输速度快，可扩展性强，标准统一，价格便宜等。目前流行的USB设备主要有移动硬盘、数码相机、MP3、U盘、USB鼠标、普通键盘、游戏杆、USB MIDI键盘、USB摄像头、USB打印机、USB扫描仪、USB声卡、USB话筒、USB网卡、USB显示器、USB电话，以及具有USB口的各种仪表仪器等。只要是连到计算机上的外设，就基本上可以通过USB来实现，足见USB系统的强大。

然而USB也有一些缺点，例如传输距离短，开发、调试难度大等。当然，它还有一个更大的缺点，那就是找出它的缺点是件非常令人头疼的事情。圈圈为了列出它的几个缺点而头疼了数月之久……

要开发USB，一个网站是开发者必须要知道的，那就是USB开发者论坛，网址是Http://www.usb.org/。该网站是公布USB的相关协议和标准的官方网站，大家需要相关协议和文档时，可以去该网站下载。另外，还有一个网站<USB专区>小组http://group.ednchina.com/93/，大家有什么问题也可以去那里交流。那里有很多搞USB的朋友，也有很多资源和代码下载。

现在USB技术已经很流行了，就像以前的串口一样。以前的电子工程师不会搞串口通信就有点落伍了，而现在的电子工程师如果不会搞USB通信就有点落伍了。

图1.2.1 USB的Logo

图1.2.1所示是USB的Logo(标志)，看到它你就应该想起USB。

1.3 USB的拓扑结构

USB是一种主从结构的系统。主机叫做Host，从机叫做Device(也叫做设备)。

通常所说的主机具有一个或者多个USB主控制器(host controller)和根集线器(root hub)。主控制器主要负责数据处理，而根集线器则提供一个连接主控制器与设备之间的接口和通路。另外，还有一类特殊的USB设备——USB集线器(USB hub)，它可以对原有的USB口在数量上进行扩展，就可以获得更多的USB口。

注意：集线器只能扩展出更多接口的USB口，而不能扩展出更多的带宽。带宽是共享一个USB主控制器的。

通常，PC上有多个USB主控制器和多个USB口。每个主控制器下有一个根集线器，根集线器通常具有一个或者几个USB口。当你有多个不同的USB设备都需要较大的数据带宽时，可以考虑将它们分别接到不同的主控制器的根集线器上，以避免带宽不足。

在Windows的设备管理器里，可以看到USB主控制器和USB根集线器。右击“我的电脑”，在弹出的菜单中选择“属性”，在“属性”对话框中选择“硬件”选项卡，再单击“设备管理器”按钮，打开“设备管理器”对话框（以后经常要到设备管理器中查看一个设备的属性，例如驱动程序是否安装好，查看驱动程序信息、设备的PID和VID等，请记住查看设备管理器的方法）。在“设备管理器”对话框中找到“通用串行总线控制器”并展开之，即可看到USB主控制器和根集线器。随便选择一个根集线器，双击之，在弹出的窗口中选择“电源”标签页，就可以看到该根集线器下的端口信息了。

图1.3.1和图1.3.2是圈圈的计算机中设备管理器以及根集线器的情况。从图中可以看到，总共有7个USB主控制器和7个USB根集线器。所选择查看的那个根集线器具有两个USB口，一个正在使用中，需要电流为100 mA（它就是圈圈正在使用的鼠标），另一个是空闲的。

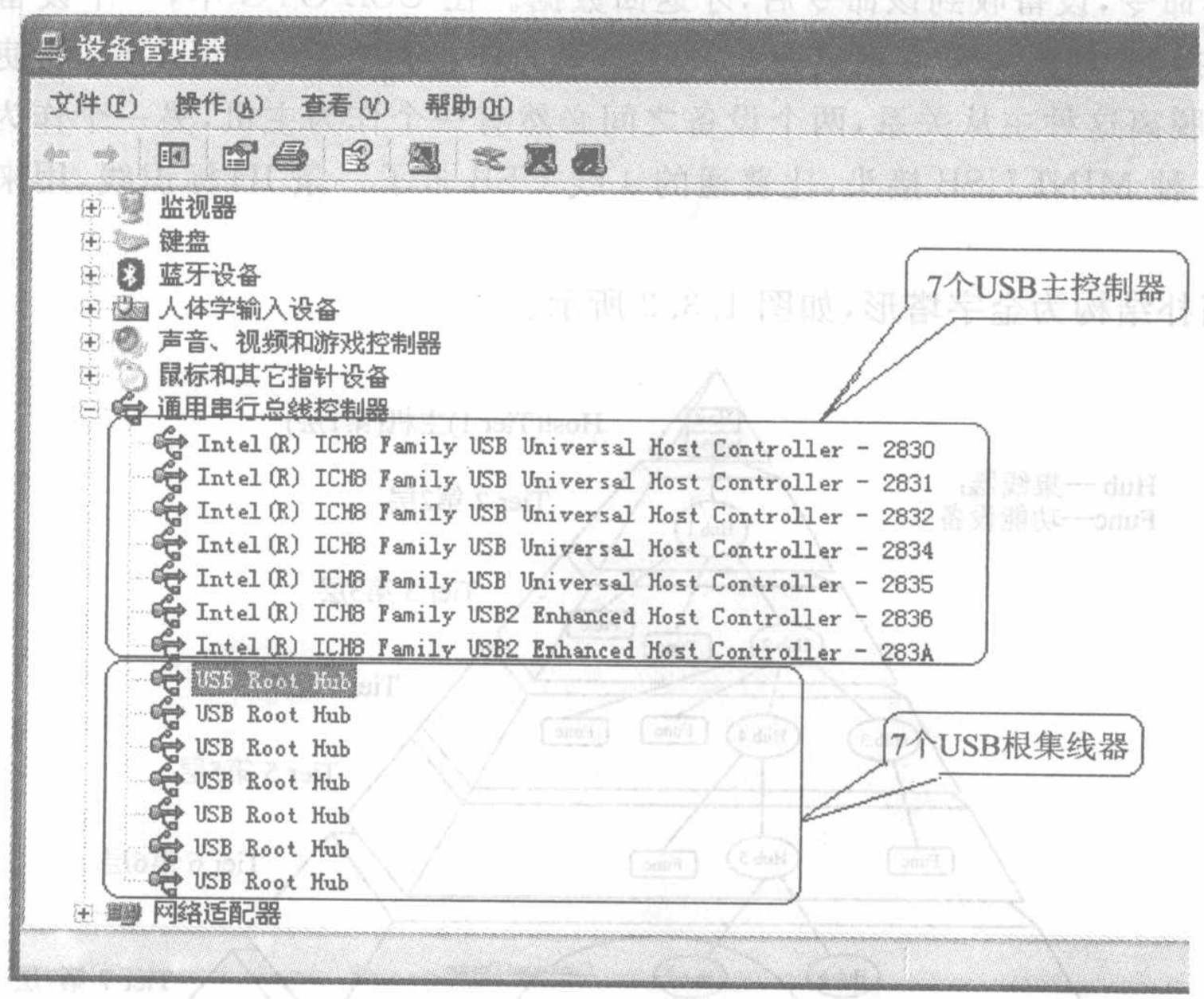


图1.3.1 设备管理器中的主控制器和根集线器

USB的数据交换只能发生在主机与设备之间，主机与主机、设备与设备之间不能直接互连和交换数据。为了在物理上区分主机和设备，使用了不同的插头和插座，具体在1.5节中会讲到。所有的数据传输都由主机主动发起，而设备只是被动地负责应答。例如，在读数据时，

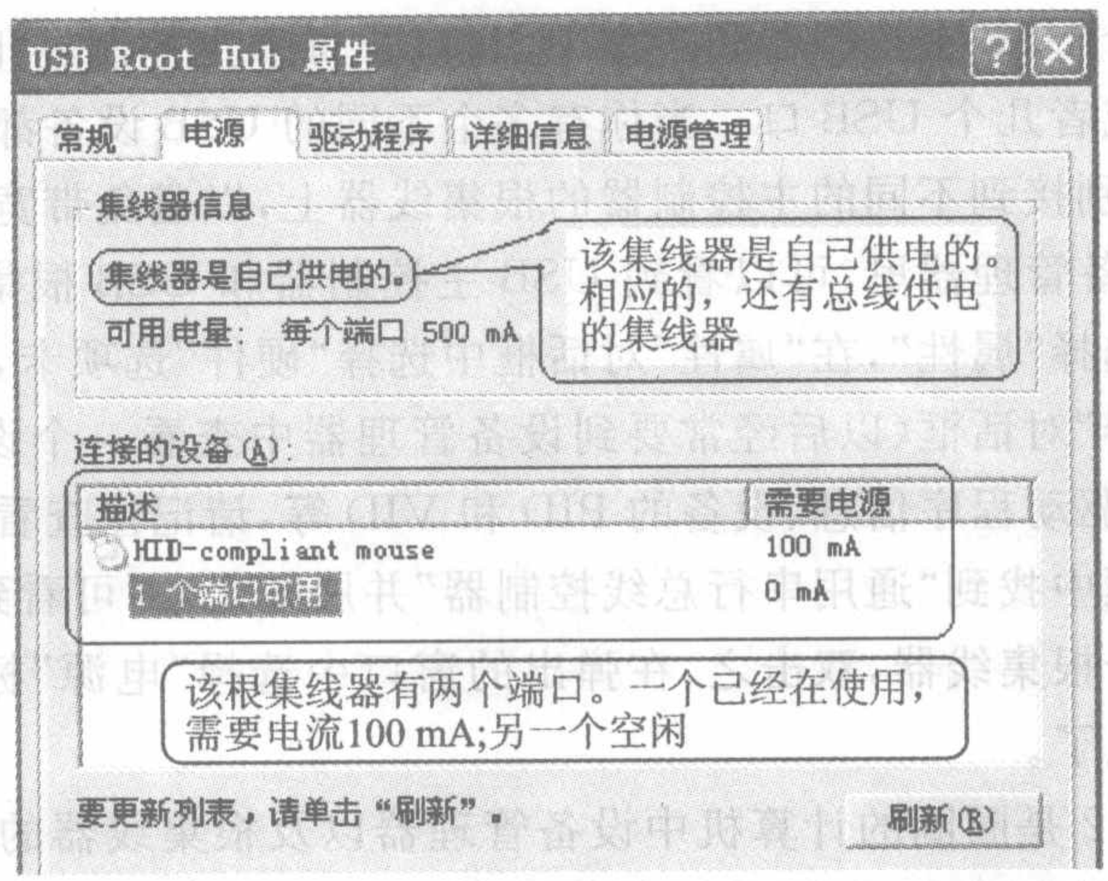


图 1.3.2 一个根集线器的属性

USB 先发出读命令，设备收到该命令后，才返回数据。在 USB OTG 中，一个设备可以在从机与主机之间切换，这样就可以实现设备与设备之间的连接，大大增加了 USB 的使用范围。但这时依然没有脱离这种主从关系，两个设备之间必然有一个作为主机，另一个作为从机。USB OTG 增加了一种 MINI USB 接头，比普通的 4 线 USB 多了一条 ID 标识线，用来表明它是主机还是设备。

USB 的拓扑结构为金字塔形，如图 1.3.3 所示。

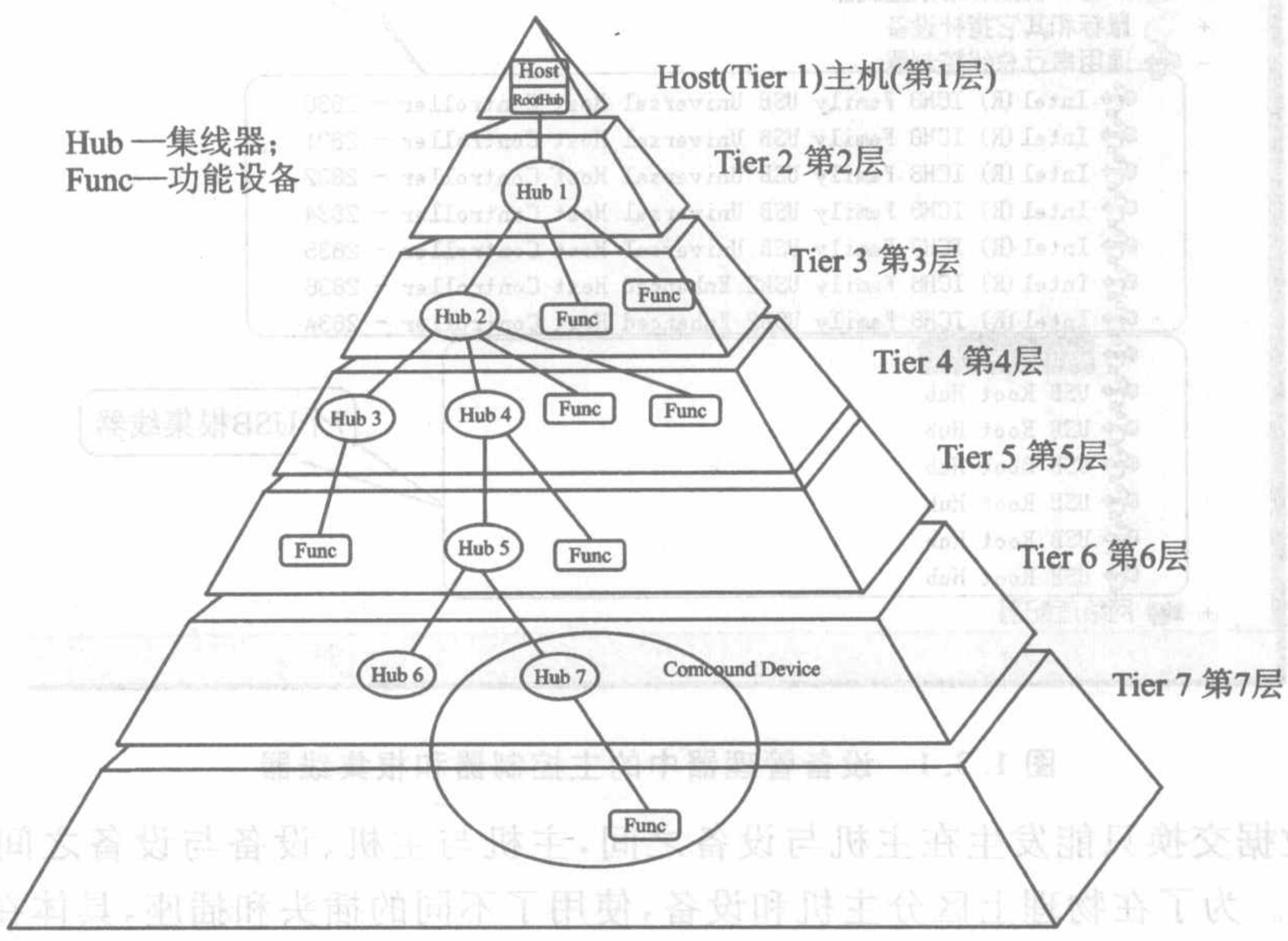


图 1.3.3 USB 的金字塔形拓扑结构

塔顶为USB主控制器和根集线器，下面接USB集线器，USB集线器将一个USB口扩展为多个USB口，多个USB口又可以通过集线器扩展出更多的接口。但USB协议中对集线器的层数是有限制的，USB1.1规定最多为4层，USB2.0规定最多为6层。理论上，一个USB主控制器最多可接127个设备，这是因为协议规定每个USB设备具有一个7 bit的地址(取值范围为0～127，而地址0是保留给未初始化的设备使用的)。实际上，通常是不会连接127个设备的。所说的一个USB主控制器可以连接多个USB设备，并不是直接简单地将多个设备并联或者串联，而是要由集线器负责端口扩展，才能连接更多的设备。在PC上，也有一个或者多个(视主板上的USB主控制器的个数而定)集线器，它就是前面提到的根集线器，直接连在USB主控制器上。

一个完整的USB数据传输过程如下：首先由USB主控制器发出命令和数据，通过根集线器，再通过下面的集线器(如果有)发给USB设备；设备对接收到的数据进行处理后，返回一些信息或者数据，它首先到达其上一层的集线器，上层的集线器再交给更上层的集线器，一直到USB主控制器为止；最终，USB主控制器将数据交给计算机的CPU处理。在标准的PC上，USB主控制器是挂接在PCI总线上的。在Windows中，由各种USB功能驱动程序负责产生和管理USB功能设备(FDO)。这就是我们最终所看到的实际设备。我们的应用程序可以通过Windows提供的一些API函数来访问USB设备，例如CreateFile()，ReadFile()，DeviceIOControl()等。

1.4 USB的电气特性

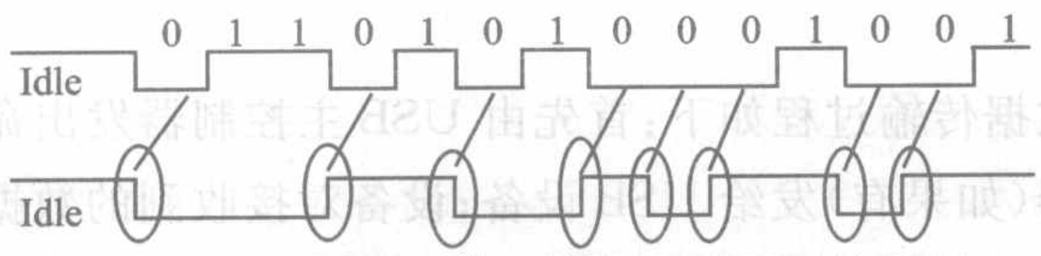
标准的USB连接线使用4芯电缆：5 V电源线(V_{BUS})、差分数据线负(D−)、差分数据线正(D+)及地(GND)。在USB OTG中，又增加了一种MINI USB接头，使用的是5条线，比标准的USB多了一条身份识别(ID)线。USB使用的是差分传输模式，因而有2条数据线，分别是D+和D−。在USB的低速和全速模式中，采用的是电压传输模式；而在高速模式下，则是电流传输模式。关于具体的各种电气参数，请参看USB协议。

USB2.0支持3种传输速度：低速模式(1.5 Mb/s)、全速模式(12 Mb/s)以及高速模式(480 Mb/s)。传输速度是指总线上每秒传输的位数，实际的数据速率要比这个速度低一些，因为有很多协议开销，例如同步、令牌、校验、位填充和包间隙等。

USB使用的是NRZI编码方式：当数据为0时，电平翻转；数据为1时，电平不翻转，如图1.4.1所示。为了防止出现长时间电平不变化(这样不利于时钟信号的提取)，在发送数据前要经过位填充(bit stuffing)处理。位填充处理的过程是这样的：当遇到连续6个数据1时，就强制插入一个数据0。经过位填充后的数据，由串行接口引擎(SIE)将数据串行化和NRZI

编码后，发送到USB的差分数据线上。在接收端，刚好是一个相反的过程。接收端采样数据线，由SIE将数据并行化（反串行化），然后去掉位填充（反位填充），恢复出原来的数据。通常，我们使用现成的USB芯片，如位填充、串行化、反串行化、CRC校验等处理过程，芯片内部的硬件已经帮我们做好了，因此可以不用关心这些细节，只需要对这个过程有个了解就可以了。当然，如果是自己设计一个USB核，或者用软件来模拟USB口，那么这些过程就必须自己来做了。

USB协议规定：设备在未配置之前，可以从V_{BUS}上最多获取100 mA的电流；在配置之后，最多可从V_{BUS}上获取500 mA的电流。V_{BUS}是5 V的电压，具体的参数请参看USB协议。



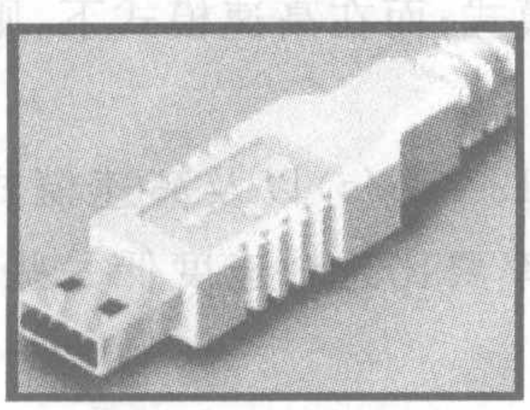
注：第一行是原始数据，第二行是NRZI编码后的数据，
由波形看出，在每个数据0出现时，电平要发生翻转。

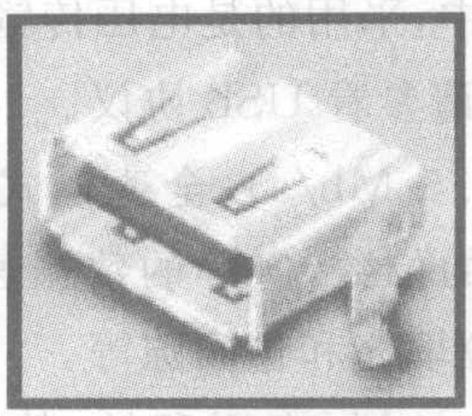
图1.4.1 NRZI编码示意图

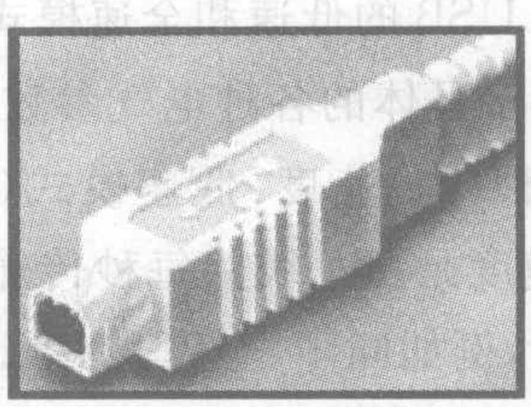
1.5 USB的线缆、插头及插座

USB是一个非常严格的标准协议，对线缆、插头和插座等有很严格的规范要求，这对防止将插头插错及提高易用性提供了保障。

在最初的协议版本中，规定USB接头有4条线：电源、D－、D＋和地线。我们暂且把这样的接头叫做标准的USB接头吧。标准的USB连接器有A型和B型。其中每一型又分为插头和插座，因而总共有4种连接器：A型插头、A型插座、B型插头和B型插座，分别如图1.5.1所示。


(a) USB A型插头


(b) USB A型插座


(c) USB B型插头

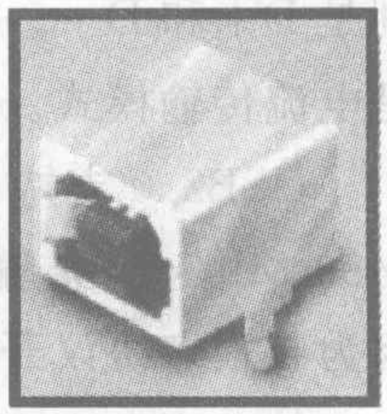

(d) USB B型插座

图1.5.1 USB插头和插座

平时常见的装在计算机上的那种USB插座就是A型USB插座，而相应的插头就是A型

插头(例如U盘或者USB连接线插到计算机上那一端的插头);而固定在设备(例如打印机)上面的插座,就是B型插座(比较四方的),相应的插头就是B型插头。也许你还见过一头方一头扁的USB连接线,没错,扁的那头就叫做A型插头,而方的那头就叫做B型插头;相应的被插的那两个插座,就分别是A型插座和B型插座。A型插头是插不进B型插座的,而B型插头也是插不进A型插座的。

后来USB OTG出现了,又增加了MINI USB接头。MINI USB接头有5条线,多了一条ID线,用来标识身份(主机还是从机)。MINI USB也分为A型和B型,还增加了一个AB型插座(不是血型呀,别搞错了,没有O型的☺)。既然它叫做MINI USB,那么当然它就是很小的了,主要用在便携式设备上,例如MP3、手机、数码相机等。USB是一主多从结构,即同一时刻只能有一台主机。两个设备之间是无法直接进行通信的,必须要经过一个USB主机。而USB OTG的出现,解决了这个矛盾:一个设备可以在某种场合下,改变身份,以主机的形式出现。因而就出现AB型的MINI USB插座,不管是A型MINI USB插头,还是B型MINI USB插头,都插得进去。依靠里面多出的那条ID线来识别它的身份是主机还是从机。这样,两个USB设备就可以直接连接起来进行数据传输了,其中一个作为主机,另一个作为从机。MP3上用的那种MINI USB插座,就是B型的MINI USB插座。

USB总共有4条线,分别是1脚V_{BUS}、2脚D−、3脚D+以及4脚GND。对于一个具体的插头或者插座,应该怎样辨认这些引脚呢?可以参照下面的方法来识别:

- 对于A型插头,可以这样看:将A型插头朝上(插头在上,线在下),4个触点露出那面对准自己,然后从左往右数,分别对应的就是引脚1、2、3、4。
- 对于B型插头,也是将插头面朝上,然后将两个有倒角的面对着自己,这时右边那个倒角位置的就是1引脚了(长一点的那个),顺时针数一圈,分别对应的就是引脚1、2、3、4。
- 对于插座,刚好是跟插头的引脚1、2、3、4一一对应的,只要找出插头的引脚排布,插座的也就知道了。

不知道你在按照上面的方法做时,有没有留意到这个USB插头里的4个触点(其实叫做触棒似乎更合适点,因为它们是长长的小棒)中,有2个是比另外2个要长一些的?没错,这不是你眼花,也不是视觉错误,这是为了支持热插拔而专门设计的硬件结构,长的2个分别是V_{BUS}引脚和GND引脚。当USB插入时,先接通GND和V_{BUS},而后接通数据线。拔下时,先断开数据线再断开V_{BUS}和GND。这就保证了在插拔过程中,不会出现有数据信号而无电源的情况。如果数据线早于电源线接通,则可能会让芯片I/O引脚电压比电源电压高,从而导致我们常说的芯片闩锁(latch up)现象。芯片一旦进入闩锁状态,轻则不能正常工作,重则使芯片过流、过热而烧毁。闩锁是一种类似可控硅的效应,要解除闩锁状态,必须断开电源重新

上电。

对于USB电缆的参数，通常许多人不会关心(除非你是生产USB线缆的)，买现成的就行了。通常选购短而粗、看上去比较结实的USB电缆性能比较好，长而细的电缆性能往往不行。全速模式和高速模式下需要使用带屏蔽的双绞电缆线，而低速模式则可以不使用屏蔽和双绞。此外，USB协议规定，USB低速电缆长度不得超过3 m，而全速电缆长度不得超过5 m。USB标准规定了信号线的颜色，其中V_{BUS}为红色，D－为白色，D＋为绿色，GND为黑色。不过，圈圈见过很多USB线缆并没有遵循标准来设计，所以大家在使用时要小心，用万用表测量一下比较可靠。

1.6 USB的插入检测机制

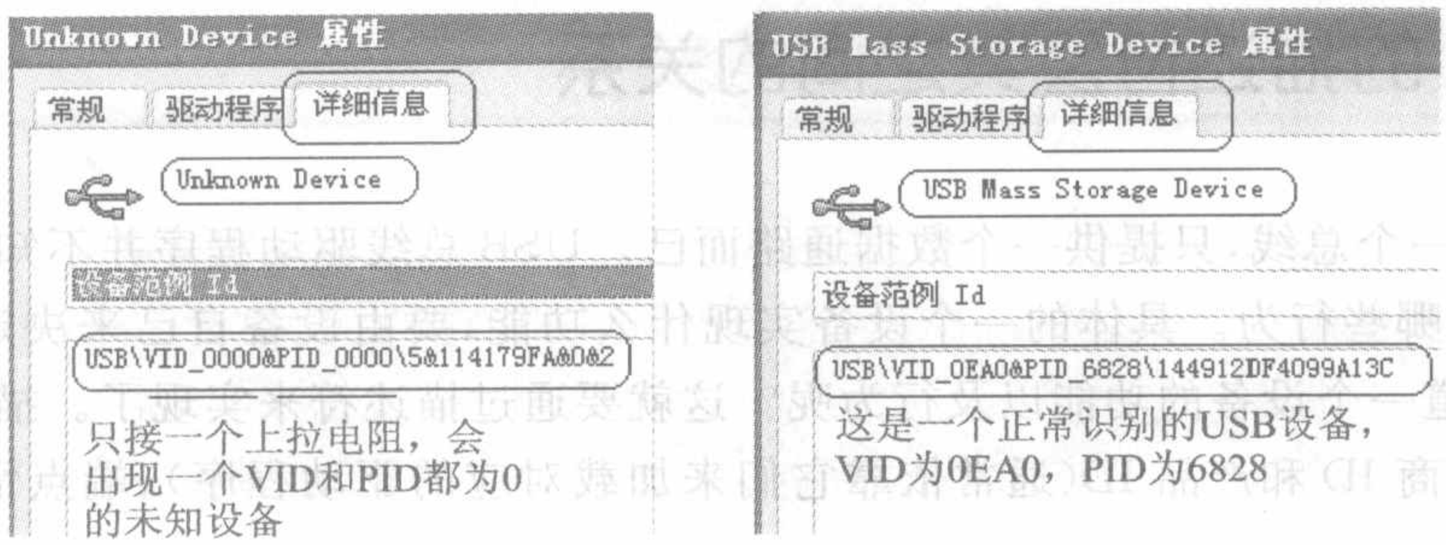
USB主机是如何检测到设备插入的呢？这要从USB集线器接收端的接口说起。

在USB集线器的每个下游端口的D＋和D－上，分别接了一个15 kΩ的下拉电阻到地。这样，当集线器的端口悬空(即没有设备插入)时，输入端就被这两个下拉电阻拉到了低电平。而在USB设备端，在D＋或者D－上接了一个1.5 kΩ的上拉电阻到3.3 V的电源。1.5 kΩ的上拉电阻是接在D＋还是D－上，由设备的速度来决定。对于全速设备和高速设备，上拉电阻是接在D＋上的，而低速设备的上拉电阻则是接在D－上。我们可以这样记忆：速度快的，上拉电阻接正的；速度慢的，上拉电阻接负的。

当设备插入到集线器时，接了上拉电阻的那条数据线的电压由1.5 kΩ的上拉电阻和15 kΩ的下拉电阻分压决定，结果大概在3 V左右。这对集线器的接收端来说，是一个高电平信号。集线器检测到这个状态后，它就报告给USB主控制器(或者通过它上一层的集线器报告给USB主控制器)，这样就检测到设备的插入了。集线器根据检测到被拉高的数据线是D＋还是D－来判断插入的是什么速度类型的设备。USB高速设备先是被识别为全速设备，然后通过集线器和设备两者的确认，再切换到高速模式下。在高速模式下，是电流传输模式，这时要将D＋上的上拉电阻断开。

一个简单的实验：用一个10 kΩ的上拉电阻接在USB的＋5 V和D＋(或者D－)上，Windows也会提示发现新硬件，但是无法找到驱动程序。这是因为D＋或者D－被拉高，集线器认为有设备插入了，它就报告给了主机；但是主机获取数据却没有响应，就会得到一个无法识别的USB设备。这时设备管理器显示一个未知USB设备，并且其VID和PID都为0，如图1.6.1所示。图1.6.1(a)是没有枚举成功时的VID和PID，都是0，图1.6.1(b)是枚举成功时的VID和PID，这两个值就是设备在设备描述符中规定的VID和PID。根据这个特性，可以简单地判断设备是否枚举成功。如果在调试时，发现VID和PID都为0，那么很可能设备

什么都没返回。



(a) 枚举不成功　　(b) 枚举成功

图 1.6.1　枚举成功和枚举不成功时设备的详细信息对照

一些 USB 芯片内部已经集成了一个 1.5 kΩ 的上拉电阻，并且有些还具有软连接(soft connect)功能，例如 PIDUSBD12。这个功能其实是通过开关来控制 1.5 kΩ 的上拉电阻是否跟 3.3 V 电源接通来实现的。如果你所使用的 USB 芯片没有内部上拉电阻，那么你需要外接一个 1.5 kΩ 的上拉电阻到 D+或者 D−(具体接哪儿要看设备的速度)。如果你还想实现软连接功能的话，则需要用一个开关来控制这个电阻是否连接，通常我们使用三极管或者场效应管来控制。

图 1.6.2 是一个实现软连接的简单电路。这是圈圈在基于 AT89C51SND1C 芯片的硬盘 MP3 上使用的电路，因为该芯片的 USB 口是没有内部上拉电阻的。J4 是一个跳线，用于直接连通上拉电阻。因为这个芯片是用 USB 口进行 ISP 下载的，但是在运行 ISP 程序时，没有程序负责将上拉电阻接入，所以只好用一个跳线，手动强制接通上拉电阻。正常使用时，J4 拔下，由用户自己的程序来控制上拉电阻的接通和断开。

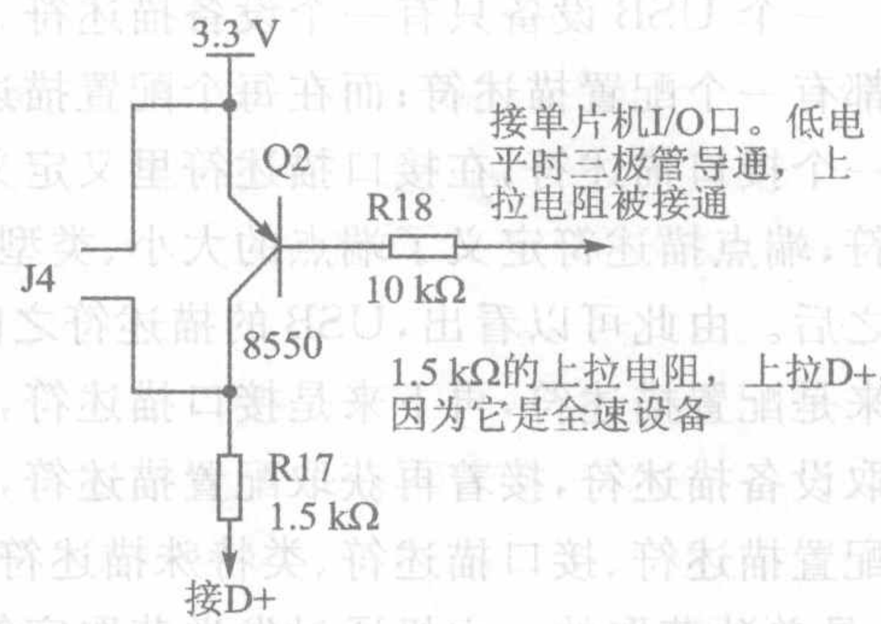


图 1.6.2　一个简单的上拉电阻控制图

这样设计有个好处，当需要连接 USB 时，才将上拉电阻连通。如果一开始就将上拉电阻接通了，可能设备还在执行初始化程序，但是主机已经检测到有设备插入了，从而发送请求，结果设备还未完成初始化工作而无法响应，导致设备不能被正常识别。另外，还有一个功能就是可以实现设备的二次枚举。当设备插入后，它先被识别成一个设备，该设备负责从主机下载固件到设备的 RAM 内，然后设备将上拉电阻断开(模拟拔下，但是设备并没断电)，接着重新连接上拉电阻。当主机检测到新设备插入，重新识别设备，这时设备使用的已经是刚下载的新固件了。Cypress 公司有一些 USB 芯片支持这样的功能。具有这种特性的 USB 设备在开发和产品固件更新等方面是很方便的，无需找专门的

烧录器来重新烧写固件，只需要更改主机端的固件程序（一个文件）即可。

1.7 USB的描述符及其之间的关系

USB只是一个总线，只提供一个数据通路而已。USB总线驱动程序并不知道一个设备具体如何操作，有哪些行为。具体的一个设备实现什么功能，要由设备自己来决定。那么，USB主机是如何知道一个设备的功能以及行为呢？这就要通过描述符来实现了。描述符中记录了设备的类型、厂商ID和产品ID（通常依靠它们来加载对应的驱动程序）、端点情况、版本号等众多信息。

USB1.1协议定义的标准描述符有设备描述符（Device Descriptor）、配置描述符（Configuration Descriptor）、接口描述符（Interface Descriptor）、端点描述符（Endpoint Descriptor）及字符串描述符（String Descriptor）。USB2.0协议中又增加了两个新的标准描述符：Device Qualifier Descriptor和Other Speed Configuration Descriptor。为了降低入门难度，本书主要以USB1.1的全速设备为例，所以对后面两个描述符不作介绍，如果对此感兴趣的，可自己下载USB2.0协议来参考。其实只要USB1.1的全速设备会做了，再去搞高速的，是很容易的。另外还有一些特殊的描述符，例如类特殊描述符（如HID描述符和音频接口描述符）、厂商自定义的描述符等。

一个USB设备只有一个设备描述符。设备描述符里决定了该设备有多少种配置，每种配置都有一个配置描述符；而在每个配置描述符中又定义了该配置里有多少个接口，每个接口都有一个接口描述符；在接口描述符里又定义了该接口有多少个端点，每个端点都有一个端点描述符；端点描述符定义了端点的大小、类型等。如果有类特殊描述符，它跟在相应的接口描述符之后。由此可以看出，USB的描述符之间的关系是一层一层的，最上一层是设备描述符，接下来是配置描述符，再下来是接口描述符，最下面是端点描述符。在主机获取描述符时，首先获取设备描述符，接着再获取配置描述符，然后根据配置描述符中的配置集合的总长度，一次将配置描述符、接口描述符、类特殊描述符（如果有）、端点描述符一次读回。对于字符串描述符，是单独获取的。主机通过发送获取字符串描述符的请求以及描述符的索引号、语言ID来获取对应的字符串描述符。

- 设备描述符主要记录的信息有：设备所使用的USB协议版本号、设备类型、端点0的最大包大小、厂商ID（VID）和产品ID（PID）、设备版本号、厂商字符串索引、产品字符串索引、设备序列号索引、可能的配置数等。
- 配置描述符主要记录的信息有：配置所包含的接口数、配置的编号、供电方式、是否支持远程唤醒、电流需求量等。
- 接口描述符主要记录的信息有：接口的编号、接口的端点数、接口所使用的类、子类、协议等。

- 端点描述符主要记录的信息有：端点号及方向、端点的传输类型、最大包长度、查寻时间间隔等。
- 字符串描述符主要是提供一些方便人们阅读的信息，它不是必需的。

说了半天，也许你还没搞清楚到底设备、配置、接口、端点等这些是什么东西。不要急，这些东西的确是有点晕人。特别是刚接触时，这么多的内容很容易让人搞混，或者似乎是懂了，然后再想想，似乎又没懂……这里所说的设备，就是一个实实在在的USB设备，例如一个USB鼠标。设备有一个设备地址，USB主机依靠这个设备地址来访问设备。而在设备内部还会分得更细。它会分出一些端点出来，例如端点0、端点1等。就是说，如果USB主机要和USB设备通信，光有设备地址是不够的，还需要一个端点地址。有了设备地址和端点地址，就能准确地对端点发送和读取数据了。好比你要去找8号教学楼的808教室，8号楼就是设备地址，而808教室就是端点地址。而配置和接口，是为了更方便地管理端点而抽象出来的概念。一个设备可以有多个配置，但是同一时刻只能有一个配置有效。每个配置下又可以有多个接口。当我们需要不同的功能时，只要选择不同的配置即可。拿刚才的教学楼来说，我们可以把它分成两个配置：平时上课用和期末考试用。考试用时，全部的教室都拿来作考场（即该配置下只有一个接口，接口下有很多端点——教室）；而平时上课用时，分成两类（即该配置下有两个接口，每个接口下有一些端点——教室）：教师休息室和上课的课室。教师休息室和课室是不能共用的（这在USB中也是如此，同一个端点号不能出现在同一个配置下的两个或多个不同的接口中）。但是平时用来做课室或者休息室的教室，考试时都可以拿来作考场（这在USB中也是如此，同一个端点号可用在不同的配置中）。具有多个接口并由接口来实现功能的设备把它叫做USB复合设备，例如一个USB音频设备，它具有一个音频控制接口，另外还可能具有一到多个音频流或MIDI流接口。在主机端会把USB复合设备的每个接口当作一个功能设备来看待。像常见的USB鼠标、U盘等，通常是单一的设备，即一个设备下只有一个配置描述符、一个接口描述符。

总结如下：由端点构成一个接口（或者反过来说，接口是端点的集合），由接口又构成一个配置（反过来说，配置是接口的集合），再由配置构成一个设备（设备是配置的集合）。学习USB，一定要把这些关系理清楚了，才能按照需要构造出一个合格的USB设备。如果一个设备的各种描述符成功返回了，那么可以说已经成功了大半。相反，只要描述符出现一点问题，哪怕只是一个bit的错误，都可能造成设备无法识别或者无法正常工作。关于各种描述符的细节到具体的USB实例中再作详细介绍。

1.8 USB设备的枚举过程

USB主机在检测到USB设备插入后，就要对设备进行枚举了。为什么要枚举呢？枚举就是从设备读取各种描述符信息，这样主机就可以根据这些信息来加载合适的驱动程序，从而

知道设备是什么样的设备，如何进行通信等。调试USB设备，很重要的一点就是USB的枚举过程，只要枚举成功了，剩下的工作就不多了。

在说枚举之前，先介绍一下USB的一种传输模式——控制传输。这种传输在USB中是非常重要的，它要保证数据的正确性，在设备的枚举过程中都是使用控制传输。控制传输分为三个过程：建立过程、可选的数据过程及状态过程。

建立(setup)过程都是由USB主机发起的。它开始于一个SETUP令牌包，后面紧跟一个DATA0数据包，接着就是数据过程。如果是控制读传输，那么数据过程就是输入数据；如果是控制写传输，那么数据过程是输出数据。如果在建立过程中，指定了数据长度为0，则没有数据过程。数据过程之后是状态过程。状态过程刚好与数据过程的数据传输方向相反：如果是控制读传输，则状态过程是一个输出数据包；如果是控制写传输，则状态过程是一个输入数据包。状态过程用来确认所有的数据是否都已经正确传输完成。

下面介绍枚举的详细过程。

① USB主机检测到USB设备插入后，就会先对设备复位。USB设备在总线复位后其地址为0，这样主机就可以通过地址0和那些刚刚插入的设备通信。USB主机往地址为0的设备的端点0发送获取设备描述符的标准请求(这是一个控制传输的建立过程)。设备收到该请求后，会按照主机请求的参数，在数据过程将设备描述符返回给主机。主机在成功获取到一个数据包的设备描述符并且确认没有错误后，就会返回一个0长度的确认数据包(状态过程)给设备，从而进入到接下来的设置地址阶段。这里需要注意的是，第一次主机只会读取一个数据包的设备描述符。标准的设备描述有18字节，有些USB设备的端点0大小不足18字节(但至少具有8字节)，在这种情况下，USB主机也是只发送一次数据输入请求，多余的数据将不会再次请求。因此，如果当设备端点0大小不足18字节时，就需要注意到这个问题。也就是说在第一次获取设备描述符时，只需要返回一次数据即可，不要再等主机继续获取剩余数据(如果还有)，因为主机不会这么干的。当主机成功获取到设备描述符的前8字节之后(USB协议规定端点0最大包长至少要有8字节)，它就知道端点0的最大包长度了，因为端点0最大包长度刚好在设备描述符的第八字节处。

② 主机对设备又一次复位。这时就进入到了设置地址阶段。USB主机往地址为0的设备的端点0发出一个设置地址的请求(控制传输的建立过程)，新的设备地址包含在建立过程的数据包中。具体的地址由USB主机负责管理，主机会分配一个唯一的地址给刚接入的设备。USB设备在收到这个建立过程之后，就直接进入到状态过程，因为这个控制传输没有数据过程。设备等待主机请求状态返回(一个输入令牌包)，收到输入令牌包后，设备就返回0长度的状态数据包。如果主机确认该状态包已经正确收到，就会发送应答包ACK给设备，设备在收到这个ACK之后，就要启用新的设备地址了。这样设备就分配到了一个唯一的设备地址，以后主机就通过它来访问该设备。需要注意的是，像D12这样的USB接口芯片，会自动等待状态过程主机的ACK之后才启用新地址，所以要在返回0长度的状态包之前，将地址写

到D12芯片的地址寄存器中，D12芯片等主机返回ACK后，才会使用新的地址。

③ 主机再次获取设备描述符。这次跟第一次有点不一样，首先是主机不再使用地址0来访问设备，而是新的设备地址；另外，这次需要获取全部的18字节的设备描述符。如果你的端点0最大包长小于18字节，那就会有多次请求数据输入（即发送多个IN令牌包）。

④ 主机获取配置描述符。配置描述符总共为9字节。主机在获取到配置描述符后，根据配置描述符中所描述的配置集合总长度，获取配置集合。获取配置描述符和获取配置描述符集合的请求是差不多的，只是指定的长度不一样。有些主机干脆不单独获取配置描述符，而是直接使用最大长度来获取配置描述符集合，因为设备实际返回的数据可以少于指定的字节数。配置集合包括配置描述符、接口描述符、类特殊描述符（如果有）、端点描述符等。接口描述符、类特殊描述符、端点描述符是不能单独获取的，必须跟随配置描述符以一个集合的方式一并返回。

如果有字符串描述符，还要获取字符串描述符。另外，像HID设备还有报告描述符等，它们是单独获取的。我们可以使用BUS Hound（一个非常好的数据包监听软件，后面会介绍）查看数据包或者通过串口返回信息来查看具体的请求，从而在程序中增加对它们的响应代码。主机请求什么，你的程序就要响应什么。

1.9 USB的包结构及传输过程

圈圈在前面说了一大堆关于数据包、令牌包、应答包之类的东西，也许让你摸不着头脑；或者是具体的数据传输过程是怎样的，让你云里雾里。的确，这些东西在没有彻底弄清楚之前，很是叫人犯晕。下面就来详细地说说数据包的结构以及它们传输的过程。

1.9.1 USB的包结构及包的分类

USB是串行总线，所以数据是一位一位地在数据线上传送的。既然是一位一位地传送，就存在着一个数据位先后的问题。USB使用的是LSB在前的方式，即先出来的是最低位数据，接下来是次低位，最后是最高位（MSB）。一个包，又被分成了很多个域（field），而LSB、MSB就是以域为单位来划分的。

前面说过，USB数据在发送到总线上之前，要先经过位填充，再经过NRZI编码。在这里讨论时，所用的数据都是原始的数据，即没有经过位填充和NRZI编码的原始数据。以后也是如此，凡是没有明确说明是位填充或NRZI编码过的数据，默认为原始的数据。另外还有一个数据传输方向的问题，因为在USB系统中，主机处于主导地位，所以把从设备到主机的数据叫做输入，从主机到设备的数据叫做输出。

USB总线上传输数据是以包为基本单位的。一个包被分成不同的域。根据不同类型的包，所包含的域是不一样的。但是不同的包有个共同的特点，就是都要以同步域开始，紧跟着

一个包标识符 PID(Packet Identifier),最终以包结束符 EOP(End Of Packet)来结束这个包。

同步域是用来告诉 USB 的串行接口引擎数据要开始传输了,请做好准备。除此之外,同步域还可以用来同步主机端和设备端的数据时钟,因为同步域是以一串 0 开始的,而 0 在 USB 总线上就被编码为电平翻转,结果就是每个数据位都发生电平变化,这让串行接口引擎很容易就能恢复出采样时钟信号;对于全速设备和低速设备,同步域使用的是 00000001(二进制数,总线上的发送顺序);对于高速设备,同步域使用的是 31 个 0,后面跟 1 个 1(需要注意的是,这是对发送端的要求,接收端解码时,0 的个数可以少于这个数)。

图 1.9.1 是一个全速或者低速 USB 数据包的同步域经过 NRZI 编码后的波形。这个波形有 7 次电平翻转,即对应着 7 个 0,最后一个电平不翻转,即对应着 1 个 1。当串行接口引擎检测到一个位的数据未发生翻转后(即收到数据 1),就认为包标识符 PID 开始了,如图 1.9.1 中的 PID0 和 PID1,就是包标识符的最低两位。

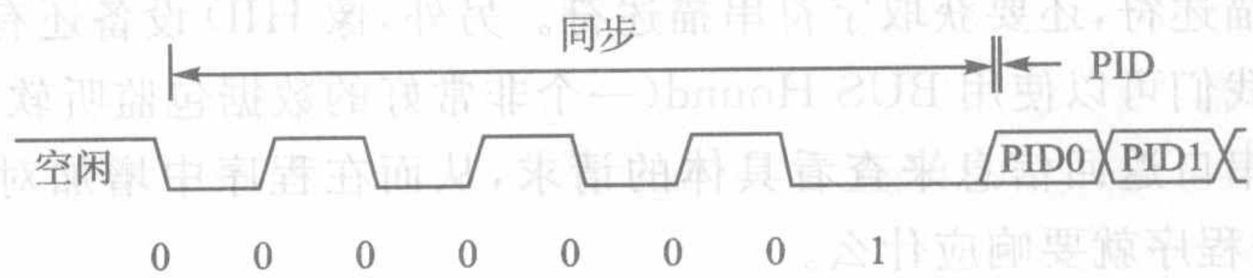


图 1.9.1 全速设备和低速设备的同步域

包结束符 EOP,对于高速设备和全速/低速设备也是不一样的。全速/低速设备的 EOP 是一个大约为 2 个数据位宽度的单端 0(SE0)信号。SE0 的意思就是,D+和 D-同时都保持为低电平。由于 USB 使用的是差分数据线,通常都是一高一低的,而 SE0 不同,是一种都为低特殊的状态。SE0 用来表示一些特殊的意义,例如包结束、复位信号等。前面提到 USB 集线器对 USB 设备进行复位的操作,就是通过将总线设置为 SE0 状态大约 10 ms 来实现的。对于高速设备的 EOP,使用故意的位填充错误来表示。那么如何判断一个位填充错误是真的位填充错误还是包结束呢?这个由 CRC 校验来判断。如果 CRC 校验正确,则说明这个位填充错误是 EOP;否则,说明传输出错。具体的定义请参看 USB 协议,这里只要知道有 EOP 这么一个东西就行了。

包标识符 PID 是用来标识一个包的类型的。它总共有 8 位,其中 USB 协议使用的只有 4 位(PID0~PID3),另外 4 位(PID4~PID7)是 PID0~PID3 的取反,用来校验 PID。USB 协议规定了 4 类包,分别是:令牌包(token packet, PID1~0 为 01)、数据包(data packet,PID1~0 为 11)、握手包(handshake packet,PID1~0 为 10)和特殊包(special packet,PID1~0 为 00)。不同类的包又分成几种具体的包。

表 1.9.1 是 USB2.0 协议中规定的各种 PID,其中有些是在 USB1.1 协议中没有的,用 * 号标出。

表 1.9.1 USB2.0 中定义的各种 PID

PID 类型	PID 名	PID[3:0]	说 明
令牌类	OUT	0001B	通知设备将要输出数据
	IN	1001B	通知设备将要输入数据
	SOF	0101B	通知设备这是一个帧起始包
	SETUP	1101B	通知设备将要开始一个控制传输
数据类	DATA0	0011B	不同类的数据包
	DATA1	1011B	
	DATA2 *	0111B	
	MDATA *	1111B	
握手类	ACK	0010B	确认
	NAK	1010B	不确认
	STALL	1110B	挂起
	NYET *	0110B	未准备好
特殊类	PRE	1100B	前导(这是一个令牌包)
	ERR *	1100B	错误(这是一个握手包)
	SPLIT *	1000B	分裂事务(这是一个令牌包)
	PING *	0100B	PING 测试(这是一个令牌包)
	—	0000B	保留,未使用

1.9.2 令牌包

令牌包用来启动一次 USB 传输。因为 USB 是主从结构的拓扑结构,所以所有的数据传输都是由主机发起的,设备只能被动地接听数据(唯一的例外是支持远程唤醒的设备能够主动改变总线的状态让集线器感知到设备的唤醒信号,但是这个过程并不传送数据,只是改变一下总线的状态)。这就需要主机发送一个令牌来通知哪个设备进行响应,如何响应。

令牌包有 4 种,分别为输出(OUT)、输入(IN)、建立(SETUP)和帧起始(SOF Start Of Frame)。

- 输出令牌包用来通知设备将要输出一个数据包。
- 输入令牌包用来通知设备返回一个数据包。
- 建立令牌包只用在控制传输中,它跟输出令牌包的作用一样,也是通知设备将要输出一个数据包,两者的区别在于:SETUP 令牌包后只使用 DATA0 数据包,且只能发到设备的控制端点,并且设备必须要接收,而 OUT 令牌包没有这些限制。

➢ 帧起始包在每帧(或微帧)开始时发送,它以广播的形式发送,所有 USB 全速设备和高速设备都可以接收到 SOF 包。USB 全速设备每毫秒产生一个帧,而高速设备每 125 μs 产生一个微帧。USB 主机会对当前帧号进行计数,在每次帧开始时(或者微帧开始时,每毫秒有 8 个微帧,这 8 个微帧的帧号是一样的,即相同的 SOF)通过 SOF 包发送帧号。SOF 中的帧号是 11 位的。在 4 个令牌包中,只有 SOF 令牌包之后不跟随数据传输,其他的都有数据传输。图 1.9.2 是 SOF 令牌包的结构。

每个令牌包,最后都有一个 CRC5 的校验,它只校验 PID 之后的数据,不包括 PID 本身,因为 PID 本身已经有 4 个取反的位进行校验了。

同步域	8位包标识PID	11位帧号	5位CRC5校验	EOP

图 1.9.2 SOF 令牌包的结构图

图 1.9.3 是 OUT、IN、SETUP 令牌包的结构。它们具有同样的结构:同步域、包标识域、地址域、端点域、CRC5 校验域和包结束。其中,地址域是要访问的设备的地址,端点域是要访问的端点号(还记得前面说过的教学楼模型吗?回忆一下地址和端点的概念);CRC5 校验只计算 PID 之后的地址域和端点域,而不包括 PID。前面说过,数据在总线上传输时,每个域的 LSB 在前,请记住这一点。例如,7 位地址在总线上传输的先后就是 A0、A1、A2、A3、A4、A5、A6。

同步域	8位包标识PID	7位地址	4位端点号	5位CRC5校验	EOP

图 1.9.3 OUT、IN、SETUP 令牌包结构图

1.9.3 数据包

顾名思义,数据包就是用来传输数据的。在 USB1.1 协议中,只有两种数据包:DATA0 包和 DATA1 包。在 USB2.0 中又增加了 DATA2 和 MDATA 包,主要用在高速分裂事务和高速高带宽同步传输中。

数据包都具有同样的结构:一个同步域,后面跟整数字节的数据,然后是 CRC16 校验,最后是包结束符 EOP,如图 1.9.4 所示。

同步域	8位包标识PID	字节0	字节1	…	字节N	16位CRC16校验	EOP

图 1.9.4 数据包的结构

之所以有不同类型的数据包,是用在当握手包出错时纠错。下面以 DATA0 包和 DATA1 包的切换为例进行具体的解释。

主机和设备都会维护自己的一个数据包类型切换机制:当数据包成功发送或者接收时,数

据包类型切换。当检测到对方所使用的数据包类型不对时，USB系统认为这发生了一个错误，并试图从错误中恢复。数据包类型不匹配主要发生在握手包被损坏的情形。当一端已经正确接收到数据并返回确认信号时，确认信号却在传输过程中被损坏。这时另一端就无法知道刚刚发送的数据是否已经成功，这时它只好保持自己的数据包的类型不变。如果对方下一次使用的数据包类型跟自己的不一致，则说明它刚刚已经成功接收到数据包了（因为它已经做了数据包切换，只有正确接收才会如此）；如果对方下一次使用的数据包类型跟自己的一致，则说明对方没有切换数据包类型，也就是说，刚刚的数据包没有发送成功，这是上一次的重试操作。

1.9.4 握手包

握手包用来表示一个传输是否被对方确认。握手包只有同步域、PID和EOP，是最简单的一种数据包，如图1.9.5所示。

同步域	包标识PID	EOP

图1.9.5 握手包结构

握手包有ACK、NAK、STALL和NYET。

- ACK表示正确接收数据，并且有足够的空间来容纳数据。主机和设备都可以用ACK来确认，而NAK、STALL、NYET只有设备能够返回，主机不能使用这些握手包。
- NAK表示没有数据需要返回，或者数据正确接收但是没有足够的空间来容纳它们。当主机收到NAK时，知道设备还未准备好，主机会在以后合适的时机进行重试传输。
- STALL表示设备无法执行这个请求，或者端点已经被挂起了，它表示一种错误的状态。设备返回STALL后，需要主机进行干预才能解除这种STALL状态。
- NYET只在USB2.0的高速设备输出事务中使用，它表示设备本次数据成功接收，但是没有足够的空间来接收下一次数据。主机在下一次输出数据时，将先使用PING令牌包来试探设备是否有空间接收数据，以避免不必要的带宽浪费。

需要注意的是，返回NAK并不表示数据出错，只是说明设备暂时没有数据传输或者暂时没有能力接收数据。当USB主机或者设备检测到数据出错时（如CRC校验错、PID校验错、位填充错等），将什么都不返回。这时等待接收握手包的一方就会收不到握手包从而等待超时。

1.9.5 特殊包

特殊包是一些在特殊场合使用的包。总共有4种：PRE、ERR、SPLIT和PING。其中PRE、SPLIT、PING是令牌包，ERR是握手包。ERR、SPLIT、PING三个是在USB2.0协议中新增的。

PRE是通知集线器打开其低速端口的一种前导包。PRE只使用在全速模式中。平时，为了防止全速信号使低速设备误动作，集线器是没有将全速信号传送给低速设备的。只有当收到PRE令牌包时，才打开其低速端口。PRE令牌包与握手包的结构一样，只有同步域、PID和EOP。当需要传送低速事务时，主机首先发送一个PRE令牌包（以全速模式发送）。对于

全速设备，将会忽略这个令牌包。集线器在收到这个令牌包后，打开其连接了低速设备的端口。接着，主机就会以低速模式给低速设备发送令牌包、数据包等。

PING令牌包与OUT令牌包具有一样的结构，但是PING令牌包后并不发送数据，而是等待设备返回ACK或者NAK，以判断设备是否能够传送数据。在USB1.1中，是没有PING令牌包的。只有在USB2.0高速环境中才会使用PING令牌包，它只被使用在批量传输和控制传输的输出事务中。直接使用OUT令牌包发送数据时，不管设备是否有空间接收数据，都会在OUT令牌包之后跟着发送一个数据包，如果设备没有空间接收数据，就返回一个NAK。这样的结果就是浪费了总线带宽，白白传送了数据。在高速设备中增加了这个PING机制，主机先用PING令牌包试试设备是否有空间接收数据，而不用事先把数据发送出去。在全速模式下，有时会遇到一个很有趣的现象，就是下位机程序慢了一点点处理完数据，结果传输速度却下降了很多。这就是前面所说的OUT过程直接发送数据导致的，也就是说，虽然程序只慢了一点，但是却丢弃了整个数据包。

SPLIT令牌包是高速事务分裂令牌包，通知集线器将高速数据包转化为全速或者低速数据包发送给其下面的端口。ERR握手包是在分裂事务中表示错误使用。由于高速分裂事务过程比较复杂，主要属于集线器的功能，在此就不详述了，感兴趣的读者可以阅读USB2.0协议相关部分。

1.9.6 如何处理数据包

这么多类型的包以及传输过程，那我们该怎么去处理呢？其实，如果使用现成的USB接口芯片，很多过程，USB接口芯片都已经处理好了，可以不用太关心这些细节，只要知道有这么一个过程就行了。

一般的USB接口芯片会完成如CRC校验、位填充、PID识别、数据包切换、握手等协议的处理。

当USB接口芯片正确接收到数据时，如果有空间保存，则它将数据保存并返回ACK，同时，设置一个标志表示已经正确接收到数据；如果没有空间保存数据，则自动会返回NAK。

收到输入请求时，如果有数据需要发送，则发送数据，并等待接收ACK。只有当数据成功发送出去（即接收到应答信号ACK）之后，它才设置标志，表示数据已成功发送；如果无数据需要发送，则它自动返回NAK。

通常只需要根据芯片提供的一些标志，准备要发送的数据到端点，或者从端点读取接收到的数据即可。所要发送和接收的数据是指数据包中的数据，至于同步域、包标识、地址、端点、CRC等是看不到的，在BUS Hound中抓到数据也是如此，仅是数据包；并且，BUS Hound中只能看到成功传输的数据，即只有ACK确认过的数据包。在USB接口芯片中，通过一些标志可以知道是哪个端点接收或者成功发送了数据。另外，由于控制传输比较特殊，SETUP包也会有相应的标志供我们使用。

1.10 USB的四种传输类型

虽然USB定义了数据在总线上传输的基本单位是包，但是我们还不能随意地使用包来传输数据，必须按照一定的关系把这些不同的包组织成事务(transaction)才能传输数据。

1.10.1 USB事务

那么事务是什么呢？事务通常由两个或者三个包组成：令牌包、数据包和握手包。

- 令牌包用来启动一个事务，总是由主机发送；
- 数据包传送数据，可以从主机到设备，也可以从设备到主机，方向由令牌包来指定；
- 握手包的发送者通常为数据接收者，当数据接收正确后，发送握手包。设备也可以使用NAK握手包来表示数据还未准备好。

USB协议规定了4种传输类型：批量传输、等时传输(也有的翻译为同步传输)中断传输和控制传输。其中，批量传输、等时传输、中断传输每传输一次数据都是一个事务；控制传输包括三个过程，建立过程和状态过程分别是一个事务，数据过程则可能包含多个事务。

1.10.2 批量传输

批量传输是最容易理解的，这里首先拿它来开刀。

批量传输使用批量事务(bulk transaction)传输数据。一次批量事务有三个阶段：令牌包阶段、数据包阶段和握手包阶段。还记得前面讲过的各种包吗？这里的每个阶段都是一个独立的包。批量传输分为批量读和批量写(记住，输入还是输出是以主机为参考的)，批量读使用批量输入事务，批量写使用批量输出事务。

批量传输没有规定数据包中数据的意义和结构，具体的数据结构要由设备自己定义。批量传输通常用在数据量大、对数据的实时性要求不高的场合，例如USB打印机、扫描仪、大容量存储设备等。

首先介绍批量输出事务。主机先发出一个OUT令牌包，这个令牌包中包含了设备地址、端点号。然后，再发送一个DATA包(具体是什么类型的DATA包，要看数据切换位)，这时地址和端点匹配的设备就会收下这个数据包。然后主机切换到接收模式，等待设备返回握手包。设备解码令牌包、数据包都准确无误，并且有足够的缓冲区来保存数据后，就会使用ACK握手包或者NYET握手包来应答主机(只有高速模式才有NYET握手包，它表示本次数据成功接收，但是没有能力接收下一次传输)。如果没有足够的缓冲区来保存数据，那么它就会返回一个NAK握手包，告诉主机目前没有缓冲区可用，主机会在稍后的时间重试该批量输出事务。如果设备检测到数据正确，但是端点处于挂起状态，则返回一个STALL握手包。如果设备检测到有错误(例如校验错误、位填充错误等)，则不做任何响应，让主机等待超时。

再来看看批量输入事务。主机首先发出一个IN令牌包，同样，IN令牌包中包含了设备地址和端点号。然后主机切换到接收数据状态，等待设备返回数据。如果设备检测到错误，那么不做任何响应，主机等待超时。如果此时有地址和端点匹配的设备，并且没有检测到错误，则该设备要做出响应：如果设备有数据需要返回，那么它把一个数据包放到总线上（具体的数据包类型要看数据切换位）；如果设备没有数据需要返回，则它直接使用NAK握手包来响应主机；如果该端点处于挂起状态，设备会返回一个STALL握手包。如果主机接收到设备发送的数据包并解码正确后，使用ACK握手包应答设备。如果主机检测到错误，则不做任何响应，设备会检测到超时。USB协议规定，不允许主机使用NAK握手包来拒绝接收数据包（否则的话，设备会在下面想：你真是U"SB"啊！既然你没空间接收数据，你还请求我返回数据给你干啥！浪费表情……）。主机在收到NAK握手包后，知道设备暂时无数据返回，主机会在稍后的时间里重试该输入事务。

另外，在USB2.0高速设备中增加了一个PING令牌包，它不发出数据，直接等待设备的握手包。因此PING事务只有令牌包和握手包。

图1.10.1是一个批量事务的流程图。下面对该流程图进行简单地解释，后面几种事务的流程图表示方法是差不多的，就不再一一解释了。

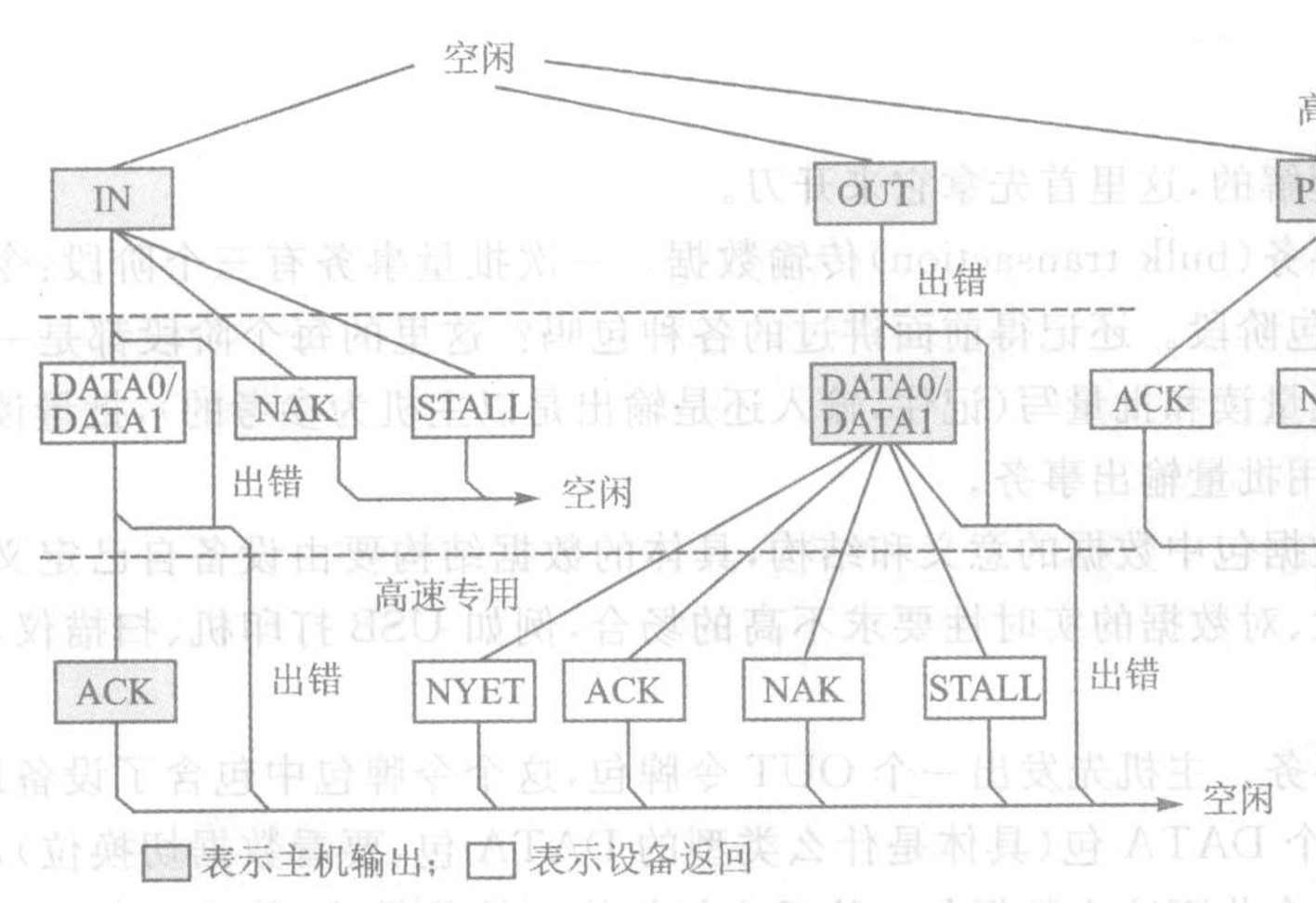


图1.10.1 批量事务流程示意图

平时无数据传输时，总线处于空闲状态。当需要传输一次事务时，主机发送一个令牌包，它可以是OUT令牌包、IN令牌包或者是PING令牌包，如图1.10.1中第一行所示。其中PING令牌包是USB2.0高速模式输出特有的，全速模式和低速模式没有这个令牌包。

如果设备解码令牌包时出错，则直接进入空闲状态。

令牌阶段之后是数据阶段或者握手阶段。对于批量输入事务，则由设备返回数据，或者返

回应答包 NAK 握手包或 STALL 握手包。这由设备的状态来决定。对于批量输出事务，则主机在令牌包后面再发送一个数据包。

PING 令牌用来探测设备是否有空间接收数据，它没有数据阶段，只有握手阶段，设备根据实际的情况返回握手包。ACK 握手包表示有空间接收数据，NAK 握手包表示无空间接收，STALL 握手包表示端点挂起。

最后是握手阶段。对于批量输入事务，如果主机接收设备返回的数据正确，则由主机返回 ACK 握手包；否则数据错误，主机什么也不返回。主机必须要能够接收数据，不能用 NAK 握手包回应设备。对于批量输出事务，如果设备能够接收数据，则返回 ACK 握手包；如果设备没有空间接收数据包，则返回 NAK 握手包；如果设备端点挂起，则返回 STALL 握手包；如果设备检测到传输错误，则什么都不回应，直接进入到空闲状态。

图 1.10.2 和图 1.10.3 更详细地给出一个传输正确的批量输入事务和批量输出事务的数据包图(传送了 2 字节数据)。

图 1.10.2 一次正确的批量输入事务

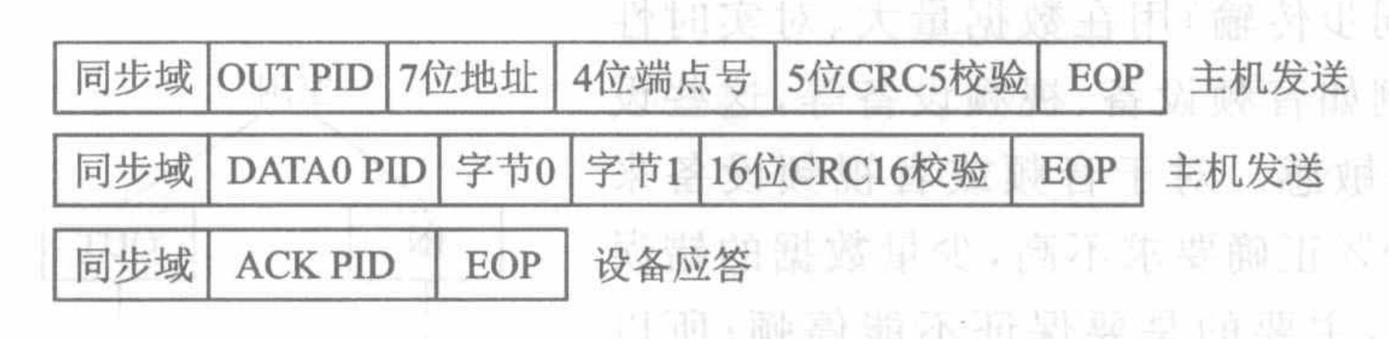


图 1.10.3 一次正确的批量输出事务

1.10.3 中断传输

中断传输是一种保证查询频率的传输。中断端点在端点描述符中要报告它的查询间隔，主机会保证在小于这个时间间隔的范围内安排一次传输。

这里所说的中断，跟我们硬件上的中断是不一样的。它不是由设备主动地发出一个中断请求，而是由主机保证在不大于某个时间间隔内安排一次传输。中断传输通常用在数据量不大，但是对时间要求较严格的设备中，例如人机接口设备(HID)中的鼠标、键盘、轨迹球等。中断传输也可以用来不断地检测某个状态，当条件满足后再使用批量传输来传送大量的数据。

除了在对端点查询的策略上不一样之外，中断传输和批量传输的结构基本上是一样的，只是中断传输中没有 PING 和 NYET 两种包。中断传输使用中断事务(interrupt transaction)，中断事务的流程图如图 1.10.4 所示，它和图 1.10.1 是差不多的。

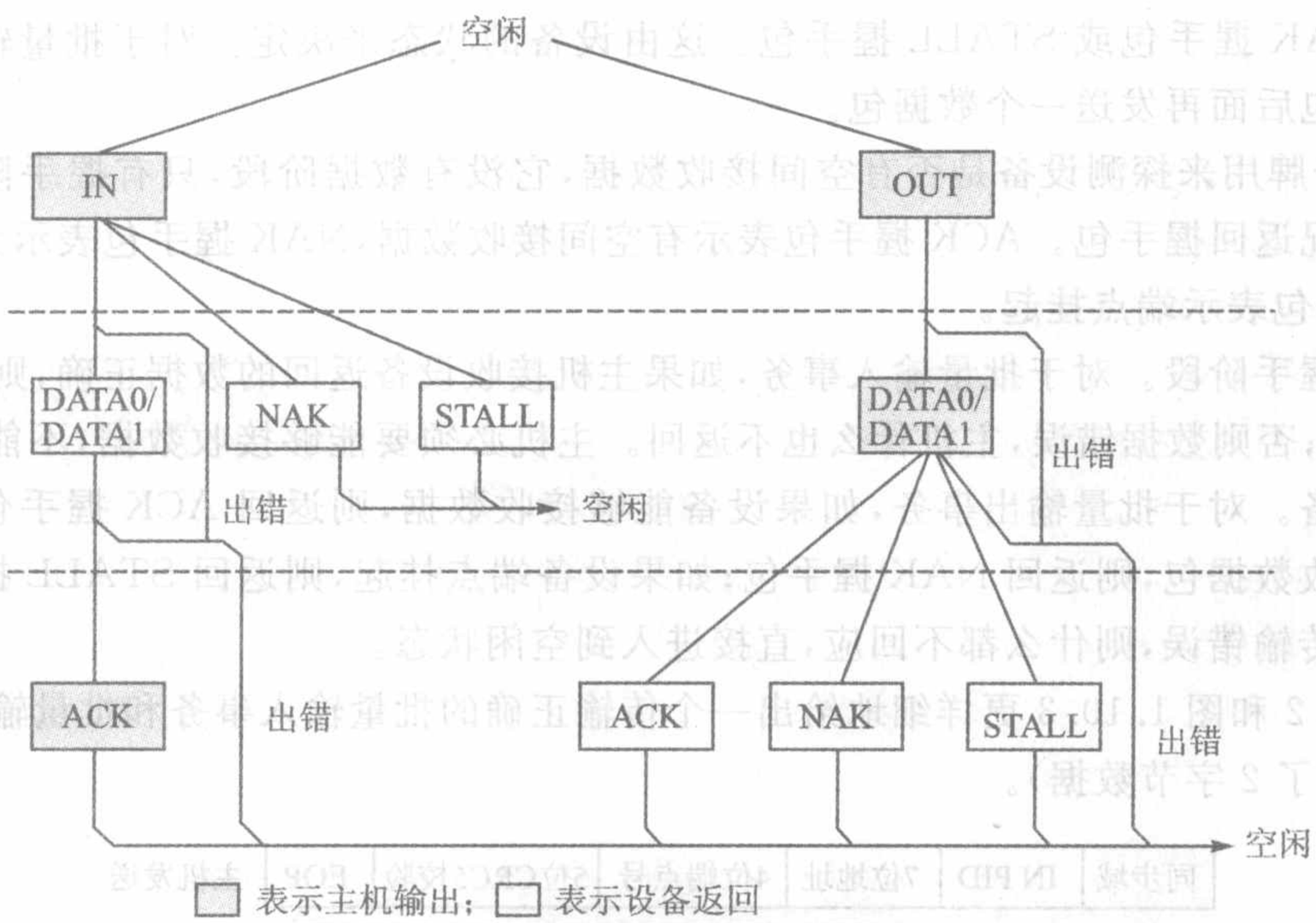


图 1.10.4 中断事务流程图

1.10.4 等时传输

等时传输(同步传输)用在数据量大、对实时性要求高的场合，例如音频设备、视频设备等，这些设备对数据延迟很敏感。对于音频或者视频设备来说，对数据的100%正确要求不高，少量数据的错误还是能够容忍的，主要的是要保证不能停顿；所以等时传输是不保证数据100%正确的。当数据错误时，并不进行重传操作。因此等时传输也就没有应答包。数据是否正确，可以由数据包的CRC校验来确认。至于出错的数据如何处理，由软件来决定。等时传输使用等时事务(isochronous transaction)来传输数据。图1.10.5是等时事务的流程图。

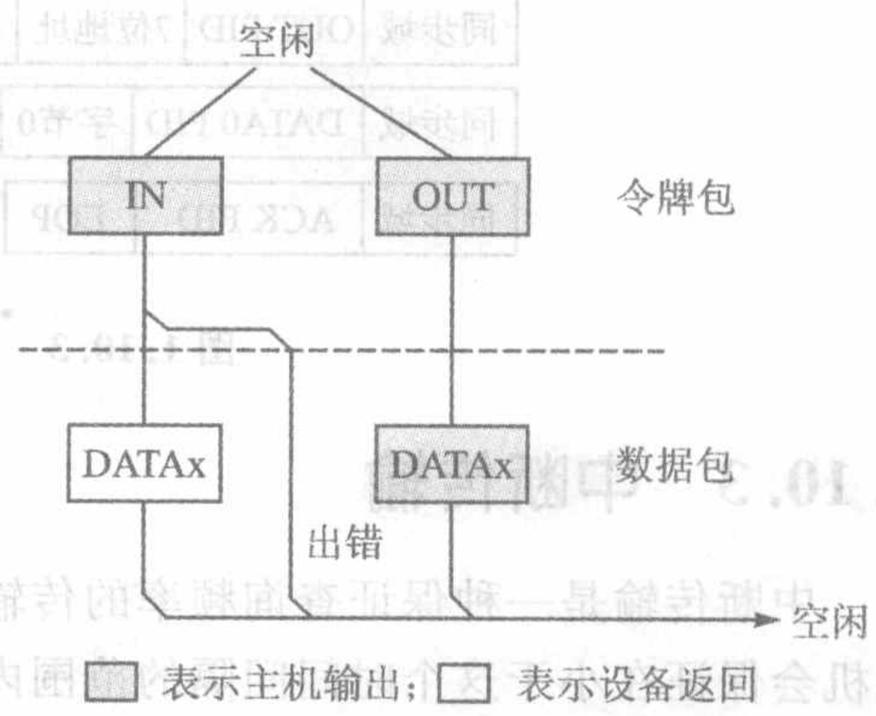


图 1.10.5 等时事务流程图

1.10.5 控制传输

控制传输与前面三种传输相比，要稍微复杂一些。前面在介绍设备的枚举过程时，就提到过控制传输。

控制传输分为三个过程：第一个过程是建立过程；第二个过程是可选的数据过程；第三个过程是状态过程。

建立过程使用一个建立事务。建立事务是一个输出数据的过程，与批量传输的输出事务相比，有几处不一样：首先是令牌包不一样，建立过程使用SETUP令牌包；其次是数据包类型，SETUP只能使用DATA0包；最后是握手包，设备只能使用ACK来应答（除非出错了，不应答），而不能使用NAK或者STALL来应答，即设备必须要接收建立事务的数据。图1.10.6是建立事务的流程图。

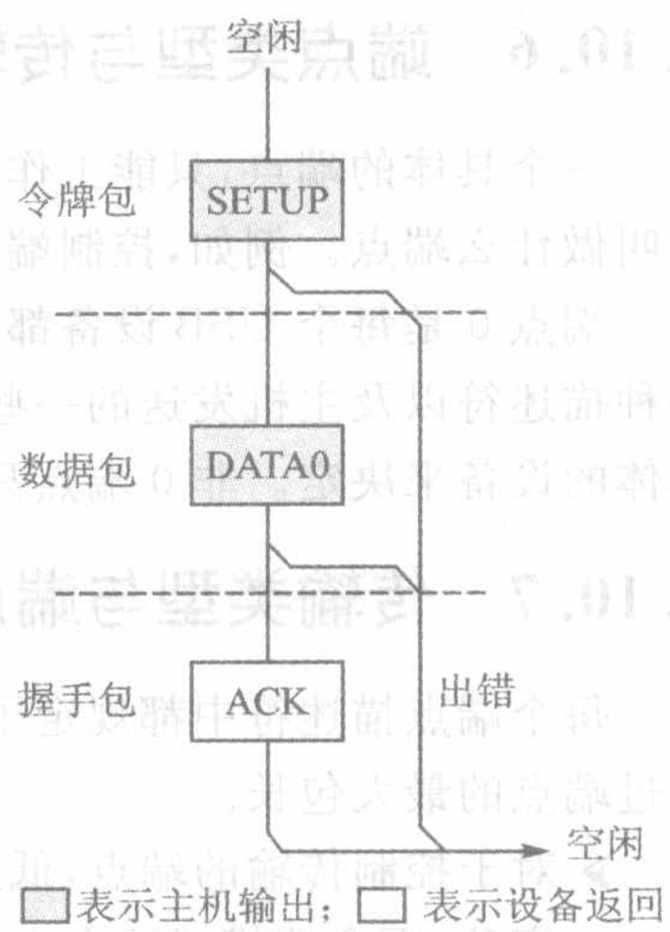


图1.10.6 建立事务的流程图

数据过程是可选的，即一个控制传输可能没有数据过程。如果有，一个数据过程可以包含一笔或者多笔数据事务。控制传输所使用的数据事务与批量传输中的批量事务是一样的。要注意的是，在数据过程中，所有的数据事务必须是同一个传输方向的。也就是说，在控制读传输中，数据过程中的所有数据事务都必须是输入的；在控制写传输中，数据过程中的所有数据事务都必须是输出的。一旦数据传输方向发生改变，就会认为进入到了状态过程。数据过程的第一个数据包必须是DATA1包，然后每次正确传输一个数据包后就在DATA0和DATA1之间交替。

状态过程也是一笔批量事务，它的传输方向刚好跟前面的数据阶段相反，即控制写传输在状态过程使用一个批量输入事务；控制读传输在状态过程使用一个批量输出事务。状态过程只使用DATA1包。

控制传输之所以要弄得这么复杂，是因为它要保证数据传输过程的数据完整性。设备枚举过程中各种描述符的获取以及设置地址、设置配置等，都是通过控制传输来实现的。关于USB协议中定义的控制传输所使用的各种标准请求的数据结构和请求命令，将会在后面的实例中具体、详细地分析。

图1.10.7是几种控制传输的实例。

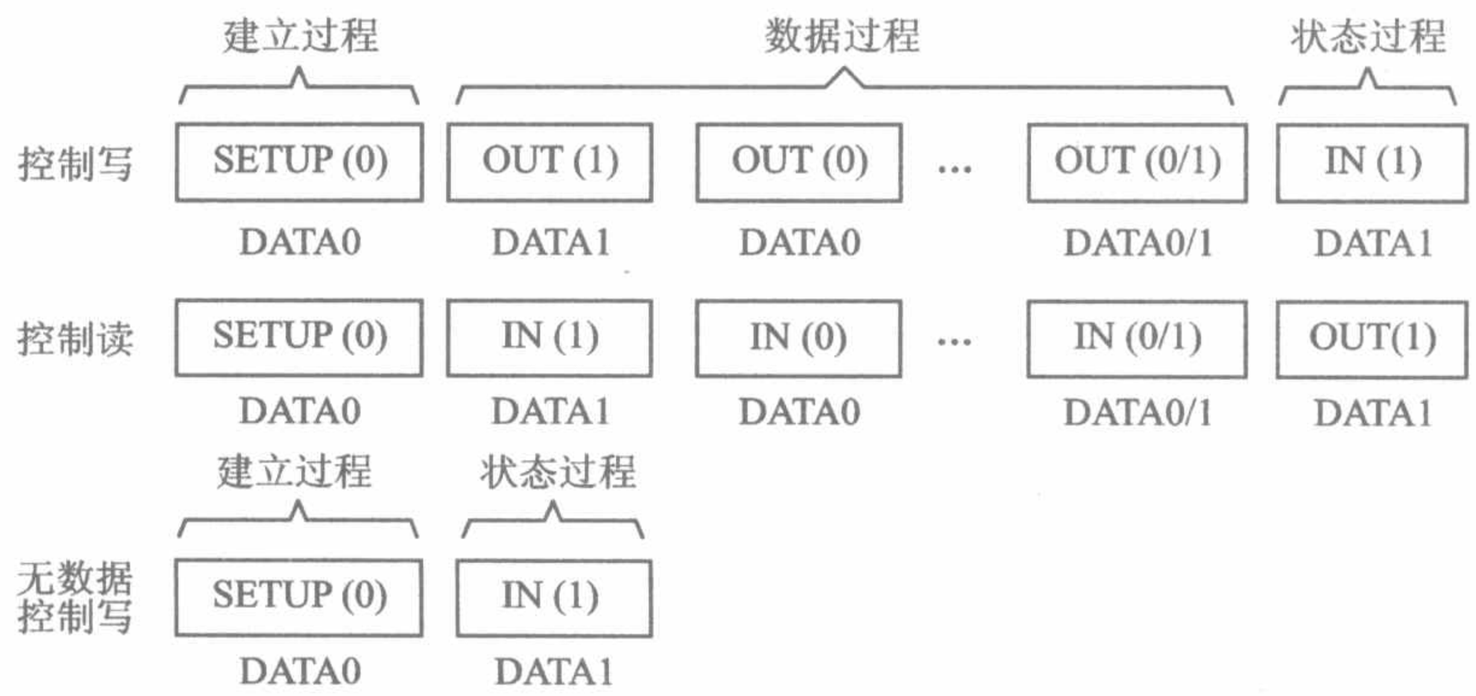


图1.10.7 几种控制传输的实例

1.10.6 端点类型与传输类型的关系

一个具体的端点，只能工作在一种传输模式下。通常，我们把工作在什么模式下的端点，就叫做什么端点。例如，控制端点、批量端点等。

端点0是每个USB设备都必须具备的默认控制端点，它一上电就存在并且可用。设备的各种描述符以及主机发送的一些命令，都是通过端点0传输的。其他端点是可选的，需要根据具体的设备来决定。非0端点只有在Set Config之后才能使用。

1.10.7 传输类型与端点支持的最大包长

每个端点描述符中都规定了端点所支持的最大数据包长。主机每次发送数据包，都不能超过端点的最大包长。

- 对于控制传输的端点，低速模式最大包长固定为8字节，高速模式最大包长固定为64字节，而全速模式可在8、16、32、64字节中选择。
- 对于等时传输的端点，全速模式最大包长上限为1 023字节，高速模式最大包长上限为1 024字节，低速模式不支持等时传输。
- 对于中断传输的端点，低速模式最大包长上限为8字节，全速模式最大包长上限为64字节，高速模式最大包长上限为1 024字节。
- 对于批量传输的端点，高速模式固定为512字节，全速模式最大包长可在8、16、32、64字节中选择，低速模式不支持批量传输。

1.11 本章小结

本章简单介绍了USB的一些基本概念，包括电气特性、数据包结构等知识，对于USB开发者来说，了解这些基本知识是必要的。

第2章 硬件系统设计

本章主要介绍USB学习板硬件系统的分析和设计，包括芯片的选择、电路的设计等内容。USB学习板是圈圈为学习USB而挑选的一款硬件比较简单、价格比较便宜的USB实验板。它使用的是PDIUSBD12加AT89S52的架构，具有键盘、串口、IDE口和LED等资源。目前已成功在该学习板上实现了具有USB鼠标、USB键盘、USB MIDI键盘、假U盘、USB转串口、USB转IDE和自定义的USB HID设备、自定义的USB设备等功能的设备。

本书主要针对的是USB系统软件的设计，所以在硬件设计上做了一些简化。例如：键盘只有8个按键；鼠标没有位移传感器，是用按键来模拟鼠标的移动。这些设计与USB的关系不大，所以没有包含这些内容。如果要做成真正的键盘、鼠标之类的，还需要读者自己设计这部分电路和程序。

2.1 方案及芯片的选定

为了能够实现不同的USB设备类，需要选择一款通用的USB接口芯片。既要考虑价格，又要考虑资料是否丰富以及购买是否方便，圈圈最终决定使用PDIUSBD12芯片。PDIUSBD12这个名字比较长，简写为D12。

PDIUSBD12芯片是荷兰菲利普(Philips)半导体公司的一颗通用USB接口芯片。它支持USB2.0的全速模式(12 Mb/s)，具有软连接(soft connect)功能和数据流指示灯(good link)；它使用8位并行的数据口与MCU连接，数字引脚兼容5 V逻辑电平；内置3.3 V稳压器，可直接使用5 V电源供电，也可使用3.3 V电源供电；内置PLL(锁相环)，外部使用6 MHz的晶体作为时钟源。除了端点0，它还有两个额外的端点，每个端点都具有输入和输出端点。其中，端点0和端点1支持最大包长16字节；端点2普通模式下支持最大包长64字节，端点2还支持等时传输模式；单向时支持最大包长128字节，双向时支持最大包长64字节。PDIUSBD12有SO28和TSSOP28两种封装。SO28封装的货较少，所以我们的板子上使用的是TSSOP28封装的。它的体积和引脚间距都比较小，焊接时稍微麻烦一些，如果焊接较熟练，也是很容易焊好的。

至于MCU，圈圈选择了容易购买、价格便宜的STC89C52RC单片机。也有很多人问过

我，为什么要选择这个 MCU，为什么不用 AVR，ARM 等？选择这个 MCU，主要有三个方面的原因：大多数初学者都是用 51 入门的，而且国内有丰富的 51 资料；功能强大、易用的开发环境 Keil C51；价格便宜、容易购买。我们主要的目的是学习 USB，而不是学习使用单片机，为了降低入门难度，就选择了常见的 51 处理器。另外，初学者也可以选择常用的 AT89S52 单片机，但需配 ISP 下载线，它的引脚跟 STC89C52RC 是完全兼容的，FLASH 和 RAM 也都是一样，可以直接替换 STC89C52RC。当然，如果没有这些芯片，使用 AT89C52 也是可以的，不过它不支持在线编程功能，需要从板上拔下来放到编程器上去烧写。这个很麻烦，尤其是在调试程序时，影响心情。

确定了两个主要的芯片后，剩下的事情就很简单了。只要按照芯片数据手册提供的引脚功能，把它们连接起来就可以了。当然，只有这两个主芯片还是不够的，我们还需要一些辅助功能，例如按键、串口、LED 和扩展接口等。

2.2　D12 引脚功能说明

要正确地使用一个芯片，首先要阅读它的数据手册（datasheet）。数据手册是芯片厂商提供给用户使用该芯片的技术文档，通常包括芯片功能简介、方框图、内部工作原理、寄存器排布、控制命令、引脚分布、电路图和封装等各种重要信息。数据手册通常可从芯片公司的网站下载，或者在一些技术论坛也会提供下载。用该芯片的型号加 pdf 作为关键字，使用搜索引擎搜索通常也可以搜索到数据手册。初学者一定要学会去查找和阅读数据手册，不要怕麻烦。拿到数据手册后，先大概浏览一遍，看自己需要的信息在哪里。实际使用时，再对需要的信息细读。

下面介绍 D12 的引脚分布。图 2.2.1 就是从 D12 的数据手册中复制来的引脚分布图。

图 2.2.1　D12 的引脚分布图

通常，从这些引脚名上可以看出很多有用的信息。例如，图 2.2.1 中 DATA〈0〉～DATA〈7〉表示数据口，而 GND 则表示地线。ALE（Address Latch Enable）也是很常用的，是地址锁存使能；CS（Chip Select）表示片选，后面加个 N 就表示低电平有效（有时也会在标号上加横杠或者在前面加斜杠），即低电平时选中该芯片；INT 表示中断请求信号；RD 表示读选通信号；WR 表示写选通信号；RESET 表示复位。这些引脚后面的 N 都表示它们是低电平有效的。XTAL1 和 XTAL2 是接晶体的。这些符号都是常用

的符号，即使没有看后面的引脚描述，也应该能从它字面上看出它是干什么用的。这对我们阅读数据手册、设计电路都是很有用的。

至于其他几个，就不是很常见了，需要看后面的引脚描述才知道具体的意思。当然，对于前面提到的几个常用的符号，也应该在引脚描述表中读一下，看看是否有特殊的地方。SUSPEND是挂起的意思，这个是USB特有的。当总线上3 ms没有数据活动时，设备要进入挂起状态，这个引脚就是用来表示这种状态。CLKOUT是时钟输出(clock output)，它是主时钟经过可编程的配置分频而得到的一个时钟输出信号，可以供给其他需要时钟源的电路使用。GL就是前面提到过的Good Link灯。V_{DD}接5 V电源，$V_{OUT3.3}$是芯片内部3.3 V稳压器的输出引脚。A0是一个地址引脚。

在数据手册中可以找到引脚描述(pin description)。下面对D12数据手册中的引脚描述一一解释。以下描述中黑体部分表示从数据手册中复制而来，第一项是引脚名，第二项是引脚编号，第三项是引脚的类型，第四项是该引脚的注释。后面的非黑体部分是圈圈对黑体部分的翻译和增加的解释。

DATA〈0〉,1,IO2,Bit 0 of bidirectional data. Slew-rate controlled.

类型IO2在数据手册中的注释有说明，它是有2 mA驱动能力的输入、输出双向数据口。DATA〈0〉是双向数据的Bit0，并且是带压摆率控制的。带压摆率控制的输出口，芯片在设计时就限制了电压的变化速度，数据翻转时，边沿不会很陡峭，从而可以减少数字噪声。对于其他几个DATA口，都是一样的，这里就不啰唆了。

GND,5,P,Ground.

类型P表示它是一个电源引脚。该引脚接电源地。

ALE,10,I,Address Latch Enable. The falling edge is used to close the latch of the address information in a multiplexed address/ data bus. Permanently tied LOW for separate address/ data bus configuration.

输入引脚。地址锁存使能引脚。在地址/数据总线复用的系统中，ALE的下降沿将地址信息锁存。在地址/数据总线分离的系统中，将ALE引脚固定为低电平。在地址/数据总线复用的系统中，地址线和数据线是由DATA引脚分时复用的。由于D12只有一个位的地址，所以实际的地址只由DATA〈0〉决定。先从DATA〈0〉发送地址信号(这时DATA引脚充当地址线使用)，然后由ALE引脚产生一个下降沿，将D0锁存到地址寄存器中去，之后，DATA引脚当数据口使用。

CS_N,11,I,Chip Select(Active LOW).

输入，片选信号(低电平有效)。

SUSPEND,12,IOD4,Device is in Suspend state.

输入,漏极开路输出(可吸入4 mA电流),该引脚表示设备处于挂起状态。

CLKOUT,13,O2,Programmable Output Clock (slew-rate controlled).

输出引脚,最大电流2 mA。可编程时钟输出(带压摆率控制)。

INT_N,14,OD4,Interrupt(Active LOW).

漏极开路输出,最大可吸入4 mA电流。中断请求信号(低电平有效)。由于它是漏极开路输出的,所以不能输出高电平,在使用时需要接上拉电阻。

RD_N,15,I,Read Strobe(Active LOW).

输入,读选通信号(低电平有效)。

WR_N,16,I,Read Strobe(Active LOW).

输入,写选通信号(低电平有效)。

DMREQ,17,O4,DMA Request.

输出(4 mA),DMA请求信号。

DMACK_N,18,I,DMA Acknowledge(Active LOW).

输入,DMA应答信号(低电平有效)。

EOT_N,19,I,End of DMA Transfer (Active LOW). Double up as V_{BUS} sensing. EOT_N is only valid when asserted together with DMACK_N and either RD_N or WR_N.

输入,DMA传输终止信号(低电平有效)。此外,还作为V_{BUS}检测用,只有当该引脚为高电平时,D12内部的上拉电阻才能被连接。EOT_N只有当DMACK_N有效并且RD_N或WR_N中的一个有效时,才有效。

RESET_N,20,I,Reset (Active LOW and asynchronous). Built-in Power-on reset circuit present on chip, so pin can be tied HIGH to V_{CC}.

输入,复位引脚(低电平有效,并且是异步的,即复位不需要时钟信号)。D12芯片内置上电复位电路。该引脚可以直接连接到高电平V_{CC}。

GL_N,21,OD8,GoodLink LED indicator (Active LOW).

漏极开路输出,最大吸入8 mA电流。Good Link发光二极管指示灯(低电平有效)。当连接正常时(未进入挂起状态),该引脚输出低电平,点亮LED。当总线上有数据传输时,LED会闪烁。

XTAL1,22,I,Crystal Connection 1 (6 MHz).

输入,6 MHz晶体连接端1。

XTAL2,23,O,Crystal Connection 2 (6 MHz). If external clock signal, instead of crystal, is connected to XTAL1, then XTAL2 should be floated.

输出,6 MHz 晶体连接端 2。如果使用外部时钟信号代替晶体,则外部信号连接到 XTAL1,而 XTAL2 悬空。

V_{CC},24,P,Voltage supply (4.0 — 5.5 V). To operate the IC at 3.3 V, supply 3.3 V to both V_{CC} and $V_{OUT3.3}$ pins.

电源引脚,供电电压范围 4.0～5.5 V。如果芯片使用 3.3 V 电源供电,则将 3.3 V 同时连接到 V_{CC} 和 $V_{OUT3.3}$ 引脚。

D—,25,A,USB D— data line.

USB D—数据线。虽然它是数据线,但事实上它是一个模拟信号的 I/O 口。

D+,26,A,USB D+ data line.

USB D+数据线。虽然它是数据线,但事实上它是一个模拟信号的 I/O 口。

$V_{OUT3.3}$,27,P,3.3 V regulated output. To operate the IC at 3.3 V, supply a 3.3 V to both V_{CC} and $V_{OUT3.3}$ pins.

3.3 V 稳压器输出端。如果芯片使用 3.3 V 电源供电,则将 3.3 V 同时连接到 V_{CC} 和 $V_{OUT3.3}$ 引脚。

A0,28,I,Address bit. A0 = 1 selects command instruction; A0 = 0 selects the data phase. This bit is a don't care in a multiplexed address and data bus configuration and should be tied HIGH.

输入,地址位。A0 为高表示命令,A0 为低表示数据。在地址/数据总线复用的系统中,该引脚是无用的,应该固定连接到高电平。

图 2.2.2 是数据手册中推荐的一个与 80C51 连接的线路图。

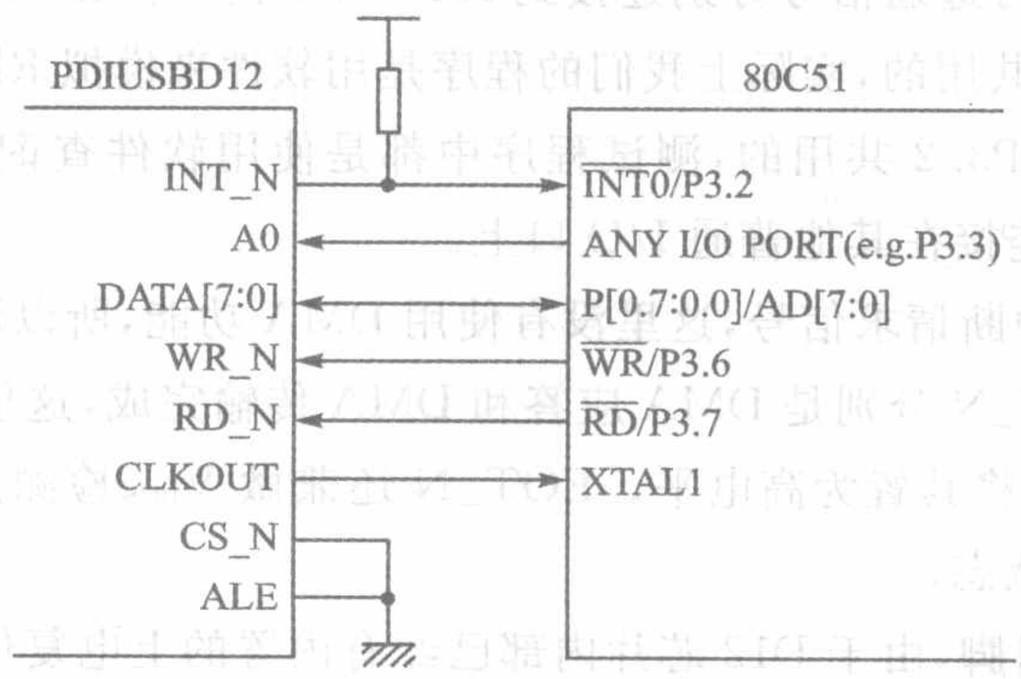


图 2.2.2 数据手册推荐的连接图

2.3 D12 与 AT89S52 的连接

将 8 根 DATA 引脚与 AT89S52 的 P0 口相连。由于 AT89S52 的 P0 口在作为普通 I/O 使用时，不能输出高电平，需要接上拉电阻。从前面的描述中可知，DATA 口的驱动能力为 2 mA，所以接了上拉电阻后，DATA 口为 0 时电流不能超过 2 mA。这里使用的是 5 V 的电源，当电流为 2 mA 时，电阻为 5 V/2 mA＝2.5 kΩ。因此选择上拉电阻大于 2.5 kΩ 就可以了。当然也不能太大，太大就会使上升沿太慢。板子实际选取 10 kΩ 的排阻来做上拉电阻。如果没有 10 kΩ 的排阻，也可以使用 2.7 kΩ、3.3 kΩ、4.7 kΩ 等排阻来代替。注意，排阻有个公共端(1 脚)，千万别搞反了。以前的助学活动中就曾遇到不少网友将排阻装反，从而导致不能正常工作。

为了让大家学习如何使用 I/O 口来模拟时序，圈圈决定使用软件模拟时序的方式来访问 D12。这里使用独立的地址线 A0(用一个 I/O 口来模拟 A0)，所以不需要使用 ALE 引脚，直接将 ALE 接地。因为数据总线上只有 D12 这个器件，所以片选信号 CS_N 直接接地，即该芯片一直是被选中的。

SUSPEND 引脚是双向的，当芯片处于挂起状态时，其输出为高。当发送远程唤醒命令时，需要外部将其拉低。由于可以通过程序来检测芯片是否挂起，所以在硬件连接上不需要用到这个引脚，直接把它连接到低电平即可。

CLKOUT 是时钟输出信号，我们没有用到，悬空即可。

INT_N 是中断请求信号，漏极开路输出的，需要外接上拉电阻。把它接到单片机的 INT0 上，既可以软件查询，也可以使用中断。由于 AT89S52 的 INT0 内部有上拉电阻，所以就不需要再接一个额外的上拉电阻了。

RD_N 和 WR_N 读写选通信号分别连接到 AT89S52 的 RD 和 WR 上。AT89S52 的 RD 和 WR 是与 P3.6、P3.7 共用的，实际上我们的程序是用软件来模拟 RD 和 WR 信号的。前面说的 INT0 也是，它是与 P3.2 共用的，测试程序中都是使用软件查询的方式，并没有使用中断。也可以将这些引脚连接在其他普通 I/O 口上。

DMREQ 是 DMA 中断请求信号，这里没有使用 DMA 功能，所以该引脚悬空即可。

DMACK_N 和 EOT_N 分别是 DMA 应答和 DMA 传输完成，这里没有使用 DMA 功能，直接用 1 kΩ 的上拉电阻将其置为高电平。EOT_N 还兼做 V_{BUS} 检测用，必须要接高电平，芯片才会进入正常的工作状态。

RESET_N 是复位引脚，由于 D12 芯片内部已经有内置的上电复位电路，所以不需要连接额外的上电复位电路，直接将该引脚通过 1 kΩ 的电路上拉到高电平。

GL_N是芯片工作的指示灯。当芯片处于活动状态时，GL_N输出低电平。接一个发光二极管(图2.3.1中的LED11)可以用来指示芯片的工作状态和USB数据传输情况。发光二极管经1 kΩ限流电阻连接到电源。

XTAL1和XTAL2接6 MHz晶体。像这样的晶体振荡电路，通常还需要外接两个22 pF左右的起振电容到地(图2.3.1中的C6和C15)。

D+和D−是USB差分数据线，分别串联一个22 Ω的阻抗匹配电阻(图2.3.1中的R2和R4)后接到USB插头上。

A0前面说过了，用一个I/O口来模拟，把它连接到AT89S52的T0引脚上，T0引脚是与P3.4复用的，我们实际使用的是P3.4这个普通I/O，与计数器0是没有关系的。

D12部分的实际的连接图如图2.3.1所示。其中，USB PORT为USB A型插头。板上电源直接从USB总线上获取。USB PORT的1引脚即为USB的5 V电源，通过1 Ω保护电阻R7给整板供电。R7是0603的贴片电阻，承受的功率较小，如果板子发生过流问题，可以让R7冒烟烧坏，从而避免烧坏USB主机。板子发生过流的原因有很多，例如PCB本身工艺问题导致电源和地线短路(所以拿到新PCB最好先检查一下再焊接，如果等到焊好后再发现问题就不好办了，尤其是那几个插座下面。圈圈就遇到过DIP40插座下面的焊盘有搭地的毛刺，费了好大劲才将插座取下然后重新装上)、焊接时搭锡，或者粗心将芯片装反、滤波电容装反等。

对于电源部分，通常是芯片的每个电源引脚都要连接一个0.1 μF左右的电源滤波电容到地，这样可以给芯片提供一个干净的电源。当然，如果芯片需要的电流较大时，还可能会放置容量较大的电容。在画原理图时，这些电容有时可能会放在一起，但是在PCB上千万别放在一堆，应该靠近各个芯片放置，否则就没有意义了。这好比自来水系统，虽然在自来水厂或者房顶上修建有水池，但是经过一条细长的自来水管后，一下想要大量的水就不行了。所以，虽然有了自来水系统，但是基本上大家都还会再配一个水桶。电路系统中的电源主滤波电容部分就好比水池，电源走线就像自来水管，而各个芯片旁边的小电源滤波电容就好比水桶，用来保存短时间的大量用“水”。如果某个芯片或模块的电流较大，除了使用较粗的“自来水管”(电源走线粗)之外，还需要使用一个大点的容器，例如“浴缸”。另外，通常小容量的电容具有较好的高频特性，主要是ESL(等效串联电感)较小。大容量电容由于其体积和结构的原因，ESL通常较大，高频特性不够好，所以在滤除高频信号时，常使用小容量电容。经常在电源滤波电路中看到大电容并联小电容，这里的小电容就是用来滤高频信号的。

D12的V_{CC}引脚直接接到5 V电源，并接一个0.1 μF的陶瓷介质电容到地(图2.3.1中的C1)。$V_{OUT3.3}$输出端这里未使用，接一个0.1 μF的电容到地即可(图2.3.1中的C5)。

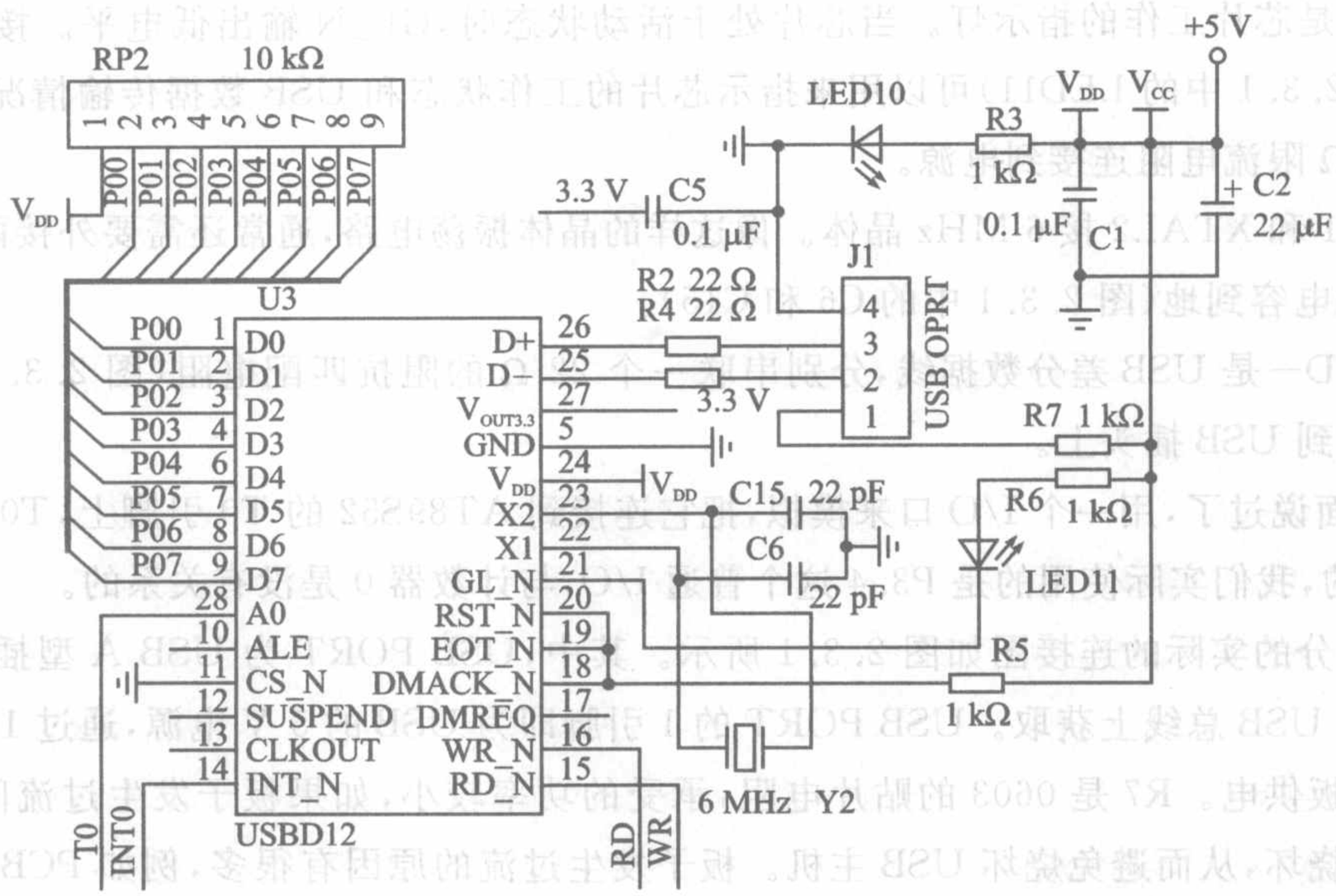


图 2.3.1　PDIUSBD12 部分原理图

2.4　串口部分电路

由于 USB 协议规定了操作是有时间限制的，所以在调试 USB 系统时，不能像普通的单片机系统那样单步调试，也就不好观察各种寄存器和变量的值。为此在电路中增加一个 RS－232串口，通过串口返回一些信息来方便我们的调试。另外，这个串口还可以用来做 USB 转串口的实验。

AT89S52 自带了异步串口，不过它是 TTL 电平的。而计算机的串口是 232 电平，这两者是不一样的，在互连时需要一个电平转换芯片。这里使用大家都很熟悉的 MAX232 芯片，电路连接如图 2.4.1 所示。

MAX232 的连接图是该芯片数据手册中提供的典型应用图。MAX232 是一个电荷泵器件，它采用开关电容技术将 5 V 电压升压和获取负电压。图 2.4.1 中的电容 C10、C11、C12、C13 就是升压和产生负压用的电容。不同的版本有不同的电容取值，有些是 0.1 μF，有些是 1 μF，还有些是 10 μF 的。通常，我们使用 1 μF 的电容就可以正常工作了。如果使用 0.1 μF 的电容，则驱动能力会弱些，不能驱动某些串口，尤其是当波特率较高时。

图 2.4.1 中 TXD 和 RXD 分别连接到 AT89S52 的 RXD 和 RXD 引脚。J6 和 J7 是跳线，可以将 TXD、RXD 跟 MAX232 芯片断开。这两个引脚同时还被用在 IDE 硬盘接口上，当使

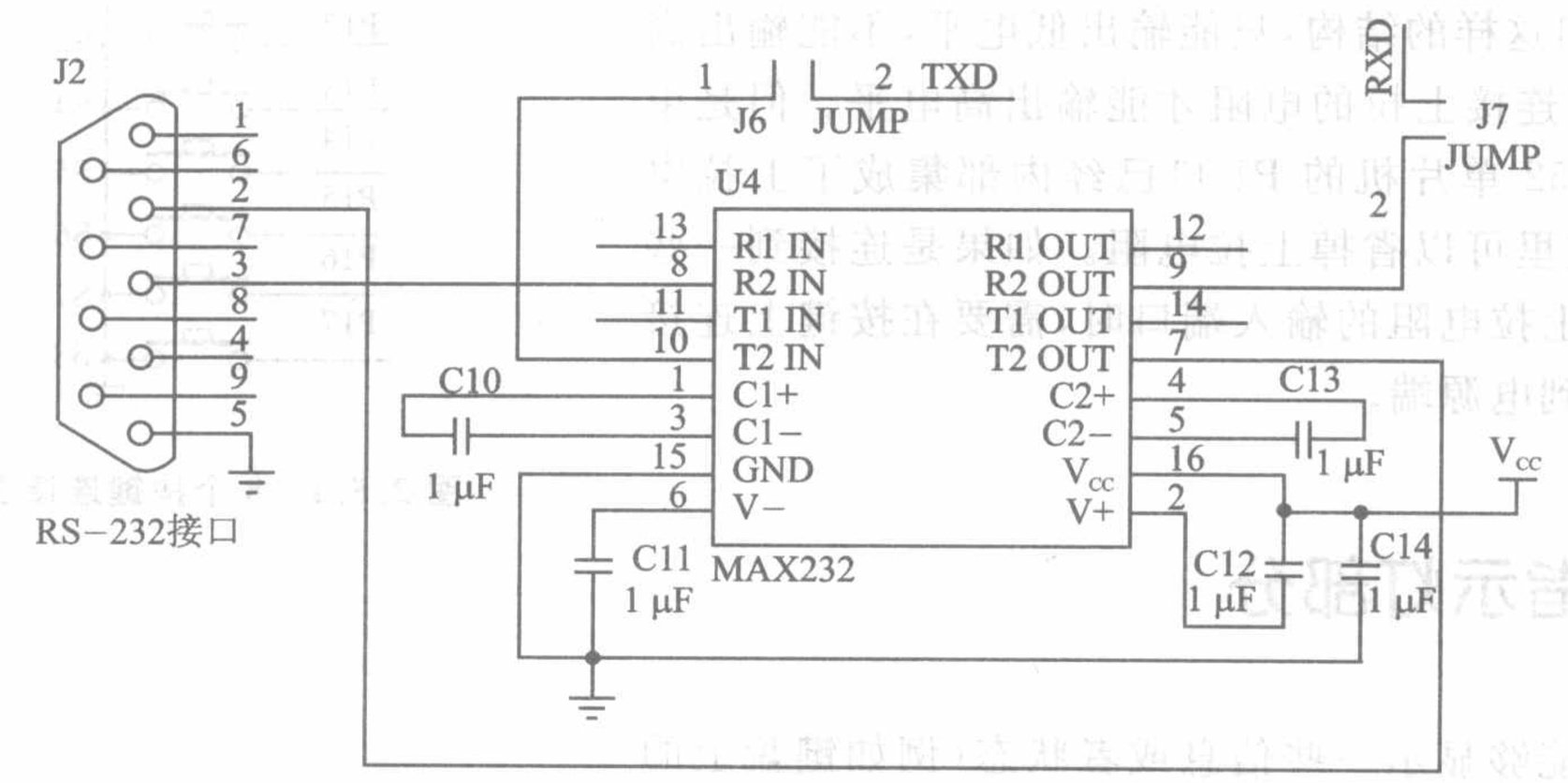


图 2.4.1　MAX232 电平转换部分电路

用 IDE 功能时，需要将两个跳线帽取下。平时正常使用时，两个跳线帽连通 MCU 和 MX232。另外，两个跳线在 PCB 板子上是靠在一起的，跳线帽可以换个方向插，可以把 MAX232 的9 脚和 10 脚连接起来，MCU 的 RXD 和 TXD 连接起来。这样连接时，串口发送的数据经过 MAX232 芯片后，又回到了串口，也就是说，我们可以做自收发测试。在这种连接模式下，使用串口调试助手，向串口发送数据，应该能够看到刚刚发送出去的数据才对；否则，就说明串口通路存在问题，需要检查串口线路。而 MCU 的 RXD 和 TXD 连接起来，MCU 自己发送的数据被自己接收，这在做 USB 转串口实验时也可以用到，相当于一个自收发过程。

串口接头使用的是 DB9 母头，而计算机端的串口则是 DB9 公头，刚好可以插上。在板子上，232 信号端的 TXD 和 RXD 是经过交叉的，使用串口直连线连接计算机的 DB9 和学习板的 DB9 即可。在做 USB 转串口时，如果要得到一个像计算机端那样的 DB9 公头接口，需要使用一条两头都是 DB9 公头的交叉串口线。不过这种串口线似乎不好买，圈圈问了几家都没有。最后只好使用一条一端是 DB9 公头，另一端是 DB9 母头的交叉串口线，再连接一个两头都是公头 DB9 的直连转接头。

2.5　按键部分

一些实验需要有输入装置，例如 USB 键盘、鼠标等，因此板上设计了 8 个轻触按键，电路如图 2.5.1 所示。按键部分电路很简单，就是使用 8 个轻触开关，连接在 P1 口和地之间。当

开关按下时，对应的I/O口变为低电平，程序通过不断地扫描这些I/O口，就可以判断出具体的按键动作了。像图2.5.1这样的结构，只能输出低电平，不能输出高电平，需要连接上拉的电阻才能输出高电平。但是由于AT89S52单片机的P1口已经内部集成了上拉电阻，所以这里可以省掉上拉电阻。如果是连接到一些没有内部上拉电阻的输入端口时，需要在按键上连接上拉电阻到电源端。

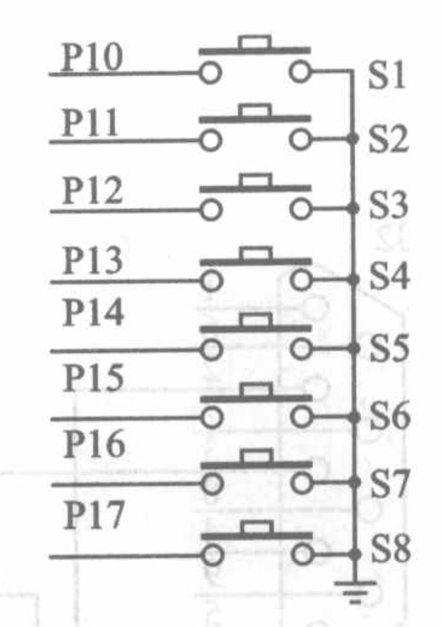


图2.5.1 8个按键连接图

2.6 指示灯部分

为了能够显示一些信息或者状态(例如键盘上的大写键锁定灯、数字小键盘灯等)，使用了8个发光二极管作为指示灯，电路如图2.6.1所示。8个LED通过1 kΩ的限流电阻直接连接到P2口上。普通LED的管压降在1.8 V左右，V_{DD}为5 V，可以计算出工作电流为(5 V−1.8 V)/1 kΩ=3.2 mA。一般的LED工作在这个电流下就有足够的亮度了，尤其是红色的，比较显眼。8个限流电阻使用了一个排阻，公共端连接到5 V的V_{DD}。

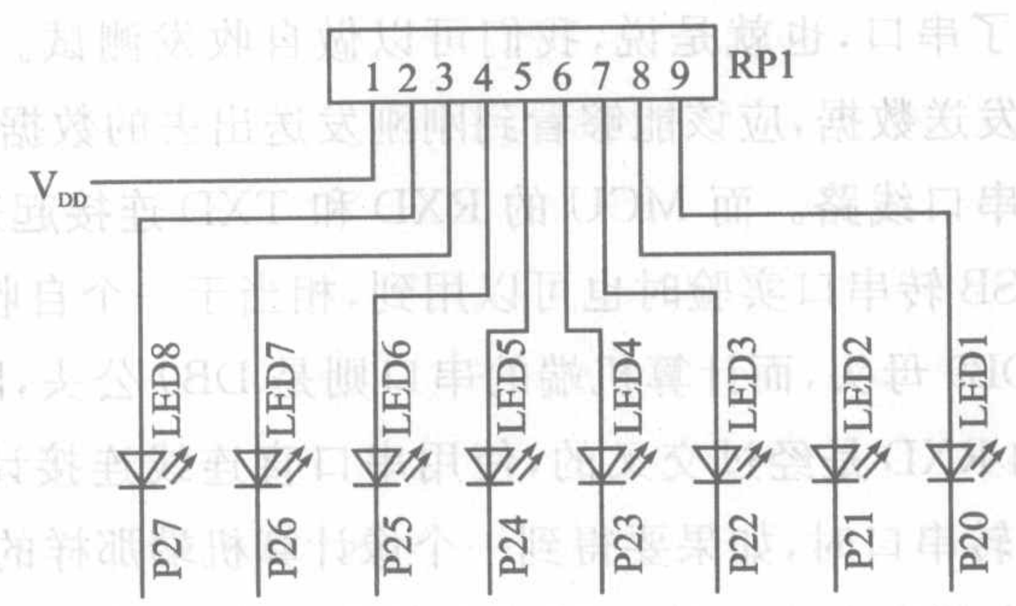


图2.6.1 指示灯连接图

2.7 IDE接口部分

由于IDE接口需要的I/O口较多，MCU的I/O口有限，所以IDE同按键、LED、串口的

I/O口进行了复用，并用一片74HC573进行了I/O口扩展。因此在使用IDE接口时，不能使用键盘、LED和串口。IDE接口部分内容比较多，如果对此感兴趣可自行查阅IDE相关资料。

2.8 单片机部分

AT89S52单片机与AT89C51或者AT89C52单片机的引脚是完全一样的，功能也几乎一样。这些芯片的最小系统电路在很多单片机教材和网站上都能找到，这里就不详述了。为了方便获得标准的串口波特率，使用了22.1184 MHz的晶体。另外，为了能够对I/O进行扩展，将单片机所有引脚都用排针引出。AT89S52单片机支持在线编程，为了方便下载程序，板上有一个ISP下载线接口(J4)。如果使用STC89C52，则是直接通过串口ISP，不需要用到J4。

完整的电路图可在光盘中找到，或者去USB专区小组或圈圈的个人博客中下载。

2.9 元件安装

由于D12芯片的封装是贴片的TSSOP28，所以引脚比较密，焊接需要一点技巧。另外，MAX232、74HC573、电阻、电容都是贴片的。有些朋友反映，贴片的元件不好焊，体积太小了。其实，当你习惯了使用贴片元件之后，你会爱上贴片元件的。贴片元件焊接起来很方便，尤其是电阻、电容，不用将引脚插过去，再翻到板子背面焊接。只要把板子摆在桌面上，然后拿着镊子夹着电阻、电容放到焊盘上焊接就OK了。焊完也不用剪引脚，因而速度很快。贴片的元件拆起来也比直插的方便得多，拆了之后也不用清理过孔，要知道，一些直插的元件，拆下来后再清理过孔是很痛苦的，尤其是当引脚很多时。

在动手焊接前，最好先检查板子，看是否有开路、短路的地方。一旦焊好后再来排查，就比较麻烦了。

焊接像D12这样引脚很密的贴片元件，一般使用拖焊。焊接前，首先要准备几样东西：干净的松香、焊锡、镊子、烙铁。注意烙铁的温度不要太高，也不能太低。烙铁温度太高时，助焊剂碳化太快，焊锡氧化也快；温度太低，焊锡流动性不好。能够让焊锡很好地流动，又不让松香狂冒烟时的烙铁温度是最合适的。这时用烙铁碰松香，松香会慢慢地冒烟，而不是剧烈的沸腾。先将D12放到电路板上，将引脚跟焊盘对整齐，用手指压住不让它动。注意引脚顺序，千万别搞反了。跟焊盘对整齐很重要，如果有错位，很可能会让焊盘碰到其他引脚。因此宁可多花些时间来慢慢对齐，也不要急着焊接。一旦焊上之后，再弄下来就比较麻烦了，特别是对于

没有丰富的焊接经验的人，可能还会把焊盘弄掉。当确认焊盘对整齐后，用镊子夹一小块松香，放到引脚上，用烙铁慢慢加热将其涂在引脚上，等松香凝固后，芯片就被暂时固定在板上了。然后松开手，再仔细检查下引脚是否真的对整齐了。注意，动作要轻，因为松香很脆，动作大点可能就将芯片弄掉了。确认对整齐后，再在另一边也加一点松香固定。然后用烙铁加一点焊锡依次将4个角的引脚焊上，以固定芯片。焊接时不用担心两个或者多个脚连在一起了，这很容易用烙铁把多余的焊锡拖掉。然后根据实际引脚的多少，估计需要多少焊锡，加上焊锡后，将焊锡熔化后从一端往另一端拖动。注意拖动的速度不要太快，要等焊锡充分熔化，缩成光滑的小球才开始移动。烙铁要带着焊锡往一边拖动，不要用力压住焊盘拖动，不然会把引脚弄弯，甚至弄掉焊盘。如果焊锡和松香都比较合适，通常拖过去就焊好了，或者只剩最后有几个脚有多余的焊锡连在了一起。这时将烙铁头上多余的焊锡在镊子或螺丝刀之类的东西上抹掉，然后再去芯片上把多余的焊锡一点点弄下来。注意操作过程中，芯片的引脚都要有松香，否则焊锡的流动性不好，容易导致粘连。当发现两个引脚之间的焊锡已经很少，但还是连在一起时，可以加一小块松香上去，再同时焊接两个引脚试试。通常这么一处理，两个引脚之间的焊锡就自动分开了。

至于像MAX232、74HC573这样的贴片，由于引脚间距比较大，拖焊反而不好。通常是一个引脚一个引脚地焊接。当然，要拖焊也是可以的。

至于贴片电阻、电容，一般也是先固定，再焊接。先在电阻或电容一端的焊盘上焊一小点锡(通常，圈圈喜欢一次把全部的贴片电阻、电容的一端都处理完，然后再把元件一个个装上去)，然后用镊子夹住元件，摆正后，用烙铁将一端焊住。待元件一端固定后，再用烙铁在另一端焊接。这时左手就不用拿镊子了，而是拿焊锡丝，加上合适的焊锡。烙铁要先放到焊盘上加热，待焊盘热后再加焊锡。注意焊接时间和力度，焊接太久可能会让对面的焊锡也熔化了，从而导致元件移位。

在USB专区小组中，有圈圈写的USB学习板安装必看和另一位网友做的贴片元件焊接视频教程，焊接前可以先看看这两篇文章，地址分别是http://group.ednchina.com/93/7005.aspx和http://group.ednchina.com/93/7842.aspx。

图2.9.1是焊接好的USB学习板的照片，挺漂亮的一个小板子，呵呵。作为对比，图2.9.2是圈圈以前用万用板搭的USB学习板，上面看起来还算整洁，背面是一堆密密麻麻的飞线……D12是用细铜丝将引脚引出，做了一个转接头，上面用松香封住，以免不小心碰断了细铜丝。圈圈没钱去做板，只好条件艰苦点了。虽然它看上去不是太好看，可是工作起来却很正常。满足了！

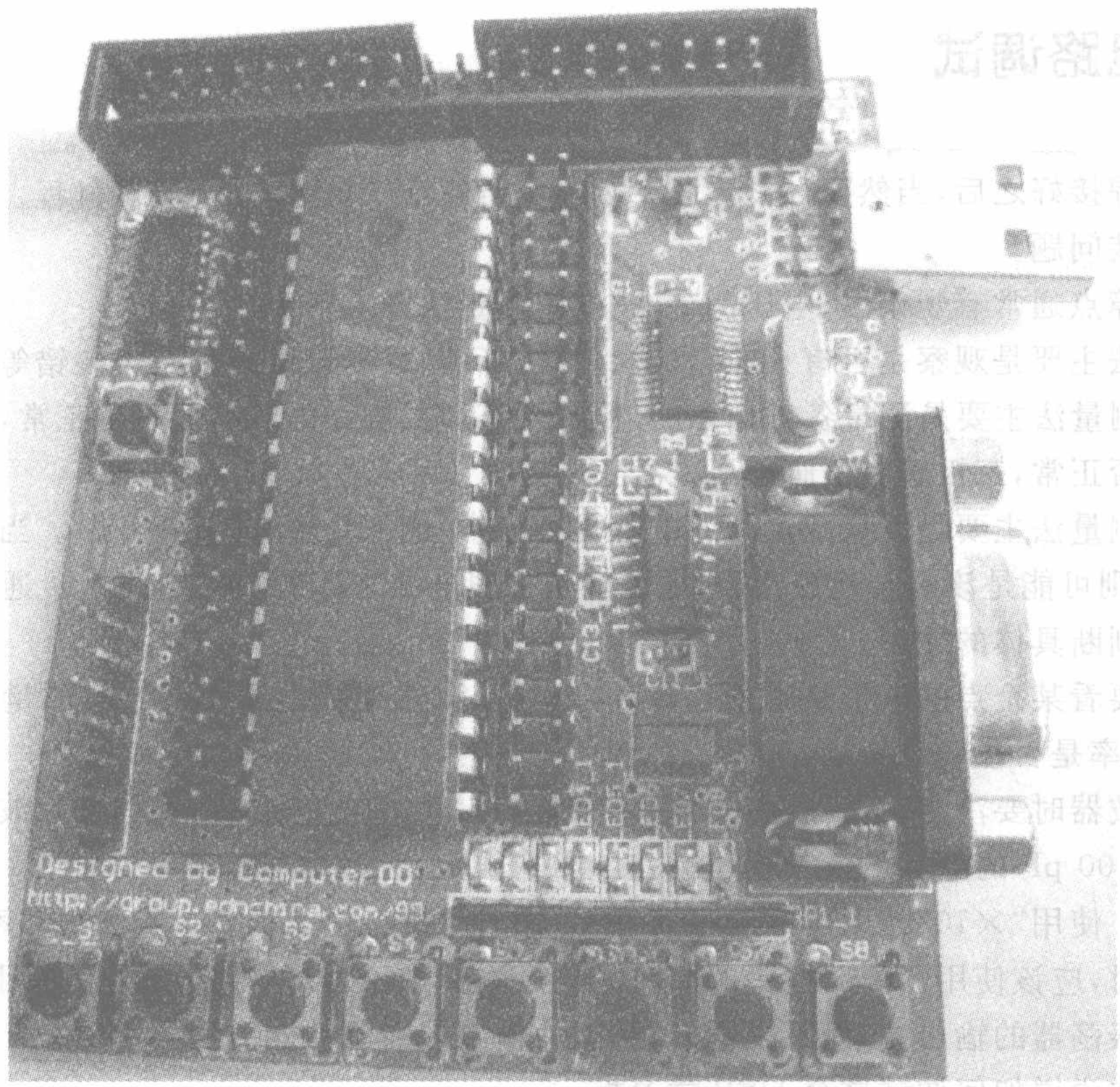


图 2.9.1　USB 学习板实物图

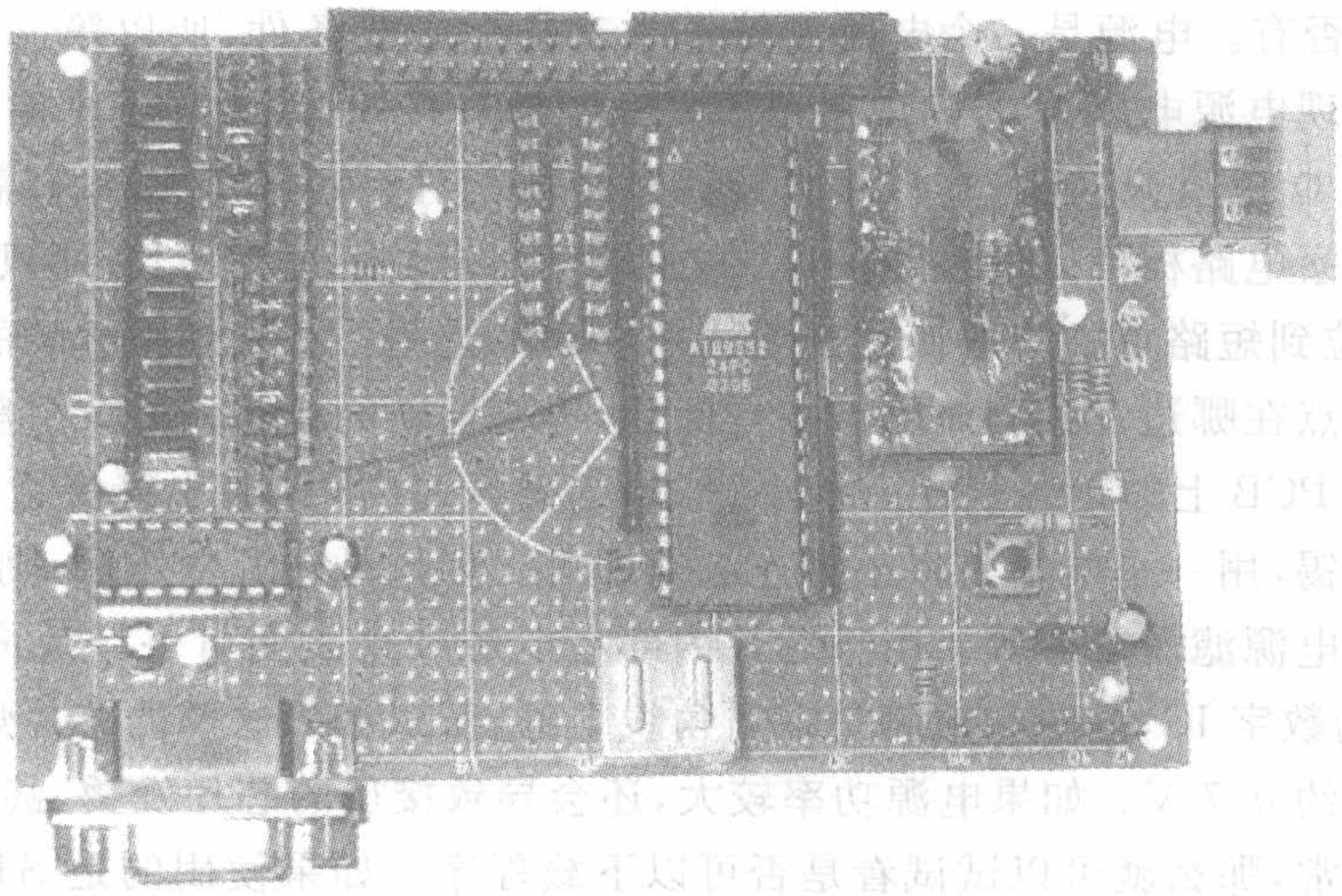


图 2.9.2　圈圈以前用万用板搭的 D12 学习板

2.10 电路调试

电路板焊接好之后，当然要试试看各个功能是否正常了，这个就是调试过程。调试主要解决设计和安装问题。

定位故障点通常有观察法、电压测量法、电阻测量法和波形测量法等。

- 观察法主要是观察是否有火花、发烫、烧焦、断裂、松动以及元件是否装错等问题。
- 电压测量法主要是测量各工作点的电压是否正常，例如电源电压是否正常，各I/O口电平是否正常，三极管各引脚电压是否正常等。
- 电阻测量法主要是测量两点间的电阻是否正常，例如是否有短路、开路。当某点电压为0时，则可能是该点与地短路了，也可能是它跟电源之间的通路断路了。通过电阻测量可以判断具体的原因。
- 当需要看某个点的波形是否正确时，需要用示波器来观察波形，例如，测量晶体是否起振，频率是否正确等。

使用示波器时要注意示波器探头对电路的影响。使用探头的“×1”档时，示波器探头大约相当于一个100 pF的电容，这在频率较高时，影响很大，会使信号幅度衰减很多，甚至导致振荡电路停振。使用“×10”档则会改善很多，因为输入阻抗会增大很多。当测量频率较高、内阻较大的信号时，应该使用示波器探头的“×10”档。测量晶体振荡电路时，也是使用“×10”档并将探头接在振荡器的输出端，而不是输入端，因为输出端的驱动能力强一些。

下面具体说说如何调试这个USB学习板。

将板子接通电源，应该可以看到电源指示灯LED10亮。如果LED10不亮，用电压表测量5 V电源电压是否有。电源是一个电路系统正常工作的必要条件，所以第一步要保证电源电压正常。如果发现电源电压为0，则说明可能是电源没有接通，或者板子上有地方短路了。如果是板子短路，1 Ω电阻R7压降应该很大或者被烧坏。对于短路现象，我们需要找到短路点。通常通过仔细观察电路板可以发现短路点，但如果仔细看了还是找不到，那只能一步步缩小故障范围，最终定位到短路点。一般使用割断法缩小故障范围。将有短路故障的线路割断，再用万用表测试短路点在哪边。找到短路点并排除后，再将切断的线路恢复。一般，切断线路的方法是使用刀子将PCB上的走线割开一条小缝，当需要恢复时，先将小缝旁边铜皮上的阻焊层(绿油)刮掉并上锡，用一小堆焊锡将小缝连起来即可。如果是电源电压低，则可能是有元件装反了，例如芯片、电源滤波电容等。由于芯片内部在电源和地之间有一个反向的二极管，而像74HC573这样的数字IC，电源和地脚是在对角线上的，所以如果这样的芯片焊反，就会使电源电压被钳位在大约0.7 V。如果电源功率较大，还会导致接反的芯片发烫，甚至冒烟烧毁。

如果电源正常，那么就可以试试看是否可以下载程序。如果使用的是STC89C52，则通过串口进行ISP。先连接好串口，然后打开STC的ISP软件，打开一个需要下载HEX文件，选

择好对应的串口，然后单击下载。等软件提示给板子上电时，插上 USB 插头，给板子供电，这时提示正在下载了。

如果按照上面的操作不能进行下载，则按照下面的方法进行检查。

首先要确认串口线是否是好的，是交叉线还是直连线。检验串口是否正常工作的方法一般是做自收发测试：用镊子或者螺丝刀（找不到这些东西用铅笔头的石墨也可以）将串口的 2 脚和 3 脚连起来（串口里面写了小数字的，仔细观察），然后在 PC 端用串口调试助手或者超级终端发送字符，观察发送出去的数据是否能够被自己收到。如果能够收到，则说明串口工作正常。

然后再将串口连接到学习板上，并给板子上电。将串口经过 MAX232 之后的 TXD 和 RXD 连接起来（参看前面串口部分的跳线设置），再做一次自收发测试，就可以判断 MAX232 芯片是否工作正常了。如果前面的操作可以收到自己发送的数据，而经过 MAX232 之后收不到，则说明 MAX232 芯片没有正常工作或者使用的串口线类型不对。通过测量电压，可以找出一些不对劲的地方。对于串口的 TTL 电平端来说，TXD 引脚空闲时应为高电平。如果测量时发现它为低电平，则说明可能是对地短路了。而 TXD 经过 MAX232 之后，应该变为负电压，大约在 −3～−12 V 之间，如果电压不够，则可能是 MAX232 没有工作或者升压电容不够，也可能是负载过重，将串口线拔下可以排除负载的影响。

如果以上检查都没问题，但还是不能下载，则可能是单片机本身没有正常工作起来。首先应该检查电源引脚、复位引脚电压是否正常。测量复位引脚电压，当按住复位开关时，应该是高电平（5 V 左右）；松开复位开关，应该为低电平（0 V 左右）。如果松开复位开关后电压比 0 V大很多，则可能是上电自动复位电容装反了，检查之。另外，晶体的两个引脚应该在芯片电源电压的 1/2 左右。51 核的单片机有个特性，就是复位后，所有 I/O 口都处于高电平状态。因此按住复位开关后，测量 52 的各 I/O 口，应该都为高电平。如果某 I/O 为低电平且没有对“地”短路，那么很可能是晶体没有起振，从而导致单片机不能正常复位。

如果能够下载程序，则说明单片机部分已经工作起来了，需要写一个测试程序来做进一步的测试，例如测试 LED、按键、D12、串口通信等。

2.11 测试程序的编写和调试

本学习板使用的是 51 内核的单片机，所以可以使用业界很流行的 Keil C 来开发。Keil C 是一款支持汇编和 C 语言的集成开发环境，自带文本编辑器，具有纯软件模拟仿真调试功能。推荐大家使用 Keil UV3，因为它的界面更友好，功能更强大，例如支持右键插入头文件，支持秒表功能等。另外，在它的工程窗口中还有一个 Templates 窗口，在这里可以设置一些常用的代码和注释，通过鼠标双击就可以将代码插入到光标处，很方便。例如圈圈就把函数头的格式、文件头的格式放在了这里面，每次写程序时，直接双击需要的代码，然后再做少量修改即

可，不用每次都去复制。如果安装 Keil UV3 后 51 编译器的路径不对，自己修改一下 toos.ini 文件即可。

以下以 Keil UV3 开发环境为例，具体讲解如何创建工程、编译、调试等。

2.11.1 建立一个工程

建立一个工程，首先要启动 UV3，然后在打开界面上选择菜单项 Project→New Project，在弹出的对话框中选择保存路径和输入需要创建的工程名，在这里把它命名为 TestBoard，如图 2.11.1 所示。

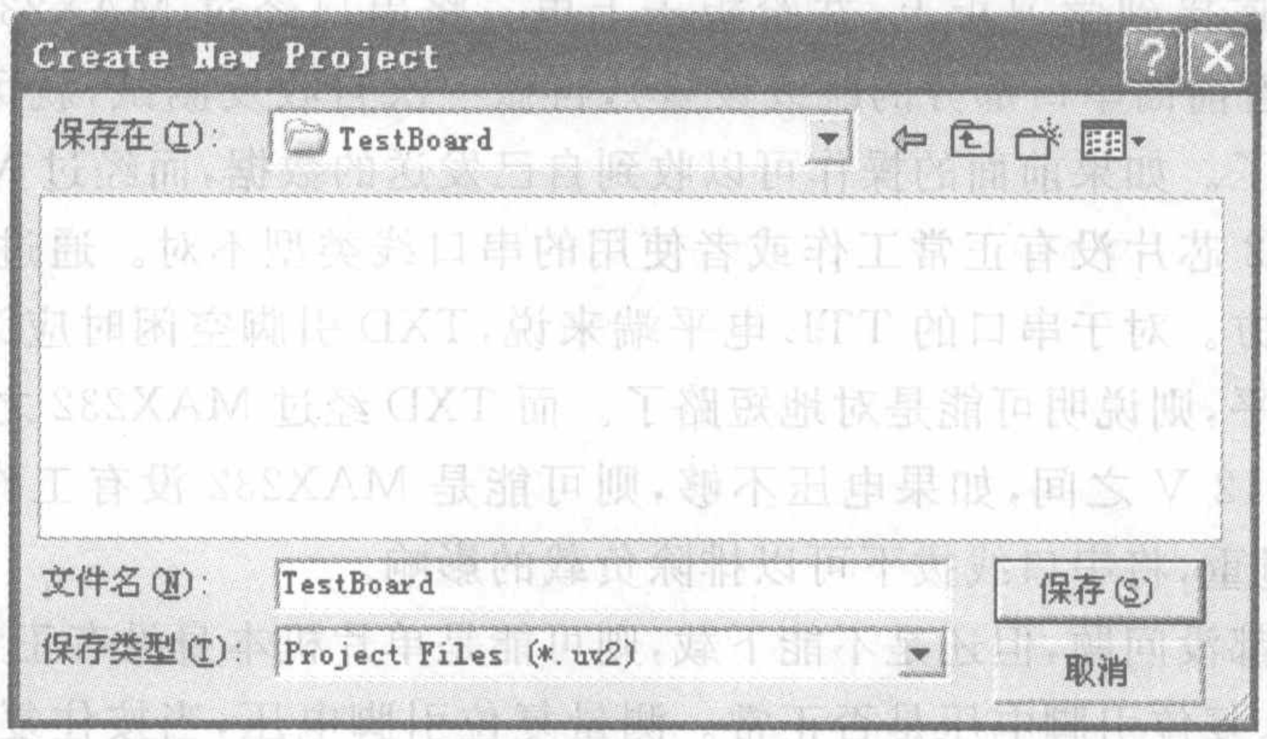


图 2.11.1 创建一个新的工程

保存结束后会弹出一个选择器件的窗口。在这里，需要选择一款合适的器件，否则编译出来的代码可能不能正常工作。由于 STC89C52 单片机在列表中没有，我们选择 Atmel 公司的 AT89S52 来代替（当然，如果本来使用的就是 AT89S52，那更应该选 AT89S52 了）。选择芯片之后，会在窗口右边列出该芯片的一些基本特性，例如 I/O 口数量、定时/计数器、中断源、FLASH、RAM 等，如图 2.11.2 所示。

单击图 2.11.2 中的“确定”按钮，会弹出另一个对话框（见图 2.11.3），询问你是否复制标准的 8051 启动代码到工程目录并添加到工程中。一般选择“是”，即需要添加启动代码。如果选择否，编译器会自动插入一段我们无法修改的启动代码。一般情况下，这对我们的工程也没什么影响，但有些特殊要求（例如不要清空 RAM 时）是必须修改启动代码，这时就必须添加启动代码。

至此，一个工程就已经建立好了，并且已经有了启动代码。在工程窗口中，点击左下边的 Files 标签，就可以看到这个工程的结构了。在 Source Group1 下，有一个 STARTUP.A51 的文件，它就是刚刚所添加的启动代码。它是一个汇编文件，主要做一些复位后的初始化工作，例如设置堆栈，初始化 RAM 为 0 等。

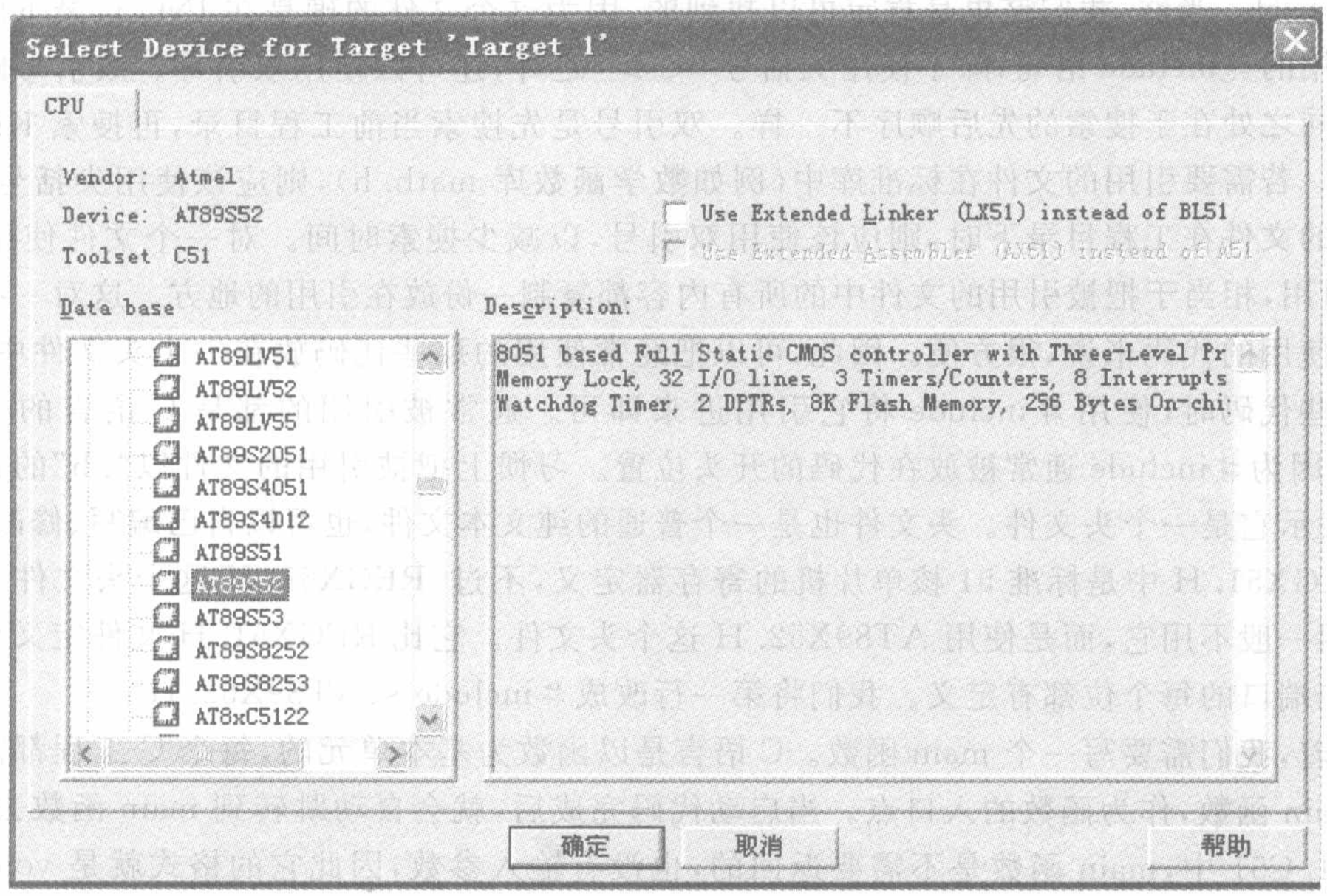


图 2.11.2　为工程选择一个合适的器件

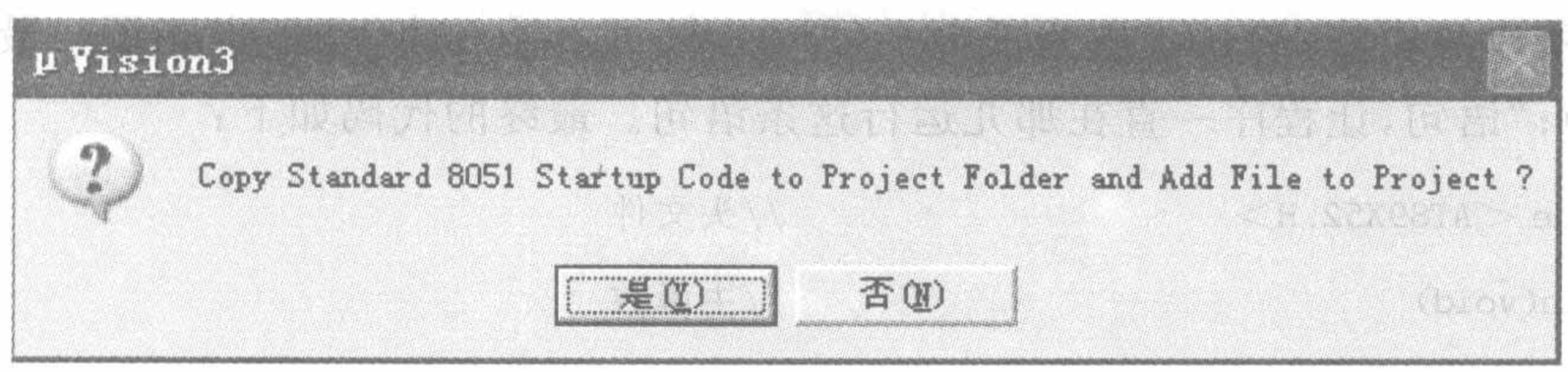


图 2.11.3　是否增加启动代码的对话框

2.11.2　为工程添加源文件

在 UV3 主界面上选择菜单项 File→New，或者直接点击工具栏上的新建文件的按钮，可以创建一个新的文本文件。在这个新的文本文件的空白处，鼠标右击，在弹出的对话框中选择“Insert ‘#include <REGX51.H>’”，这将为我们添加一条包含器件头文件的代码，该语句是跟我们所选择的器件自动相关联的。#include <REGX51.H>这条代码告诉编译器，将引用 KEIL 安装目录下的 INC 目录（或其子目录）下的 REGX51.H 文件。如果在 INC 目录下找不到 REGX51.H 文件，则编译器会到当前工程目录（在这里，就是 TestBoard 这个工程所在的目录）下去查找 REGX51.H 文件。如果还是找不到，则编译器报错，显示找不到头文件

REGX51.H。当然，我们这里是肯定可以找到的，因为这个文件的确是在INC目录下的。对文件引用的#include语句，除了使用尖括号"< >"之外，还可以使用双引号。双引号跟尖括号的不同之处在于搜索的先后顺序不一样。双引号是先搜索当前工程目录，再搜索KEIL安装目录。若需要引用的文件在标准库中(例如数学函数库math.h)，则应该使用尖括号；若需要引用的文件在工程目录下时，则应该使用双引号，以减少搜索时间。对一个文件使用#include引用，相当于把被引用的文件中的所有内容都复制一份放在引用的地方。这对一些经常要重复使用的代码来说，很方便。因此，可以把经常使用的那些代码放在一个头文件中，需要使用这些代码时，使用#include将它引用进来即可。通常被引用的都是C语言的头文件(*.h)，因为#include通常被放在代码的开头位置。习惯上把被引用的文件以".h"的扩展名保存，表示它是一个头文件。头文件也是一个普通的纯文本文件，也可以自己编写、修改它。

REGX51.H中是标准51核单片机的寄存器定义，不过，REGX51.H这个头文件不太好用，圈圈一般不用它，而是使用AT89X52.H这个头文件。它比REGX51.H文件定义得更详细，例如端口的每个位都有定义。我们将第一行改成#include <AT89X52.H>。

接着，我们需要写一个main函数。C语言是以函数为基本单元的，每个C工程都需要有一个main函数，作为函数的入口点。当启动代码完成后，就会自动跳转到main函数去运行。在KEIL C51中，main函数是不需要返回的，也没有输入参数，因此它的格式就是void main(void)。当进入main函数之后，先是一些初始化工作，然后是设置一个死循环，剩余的工作都在死循环中做。在初始化代码中，增加一条让8个LED都亮的语句。因为8个LED是接在P2口上的，所以只要将P2口设置为低电平(即0)，就可以让8个LED点亮了。最后，加入一条"while(1);"语句，让程序一直在那儿运行这条语句。最终的代码如下：

```
#include <AT89X52.H>                    //头文件
void main(void)                         //主函数
{
 P2 = 0;                                //点亮8个LED
 while(1);                              //死循环
}
```

这里将文件名设置为main.c。光有C源文件还不够，还需要把它添加到工程中去。在工程窗口中的Files窗口中，右击Target 1下的Source Group 1，在弹出的菜单中选择Add Files to Group 'Source Group 1'，弹出一个增加文件的对话框，在文件选项下拉列表中选择C Source file，然后找到main.c文件，双击(或者选中后单击Add按钮)，此时已将main.c文件添加到工程中。如果还有其他文件需要添加，依次添加即可。然后关闭添加文件的窗口，可以看到在Source Group 1下面多了一个main.c文件。

然后单击"编译"(Build target)按钮，或者选择菜单项Project→Build target，即可完成对工程的编译和链接，编译产生的报告如图2.11.4所示。正确的程序，编译后应该是没有错误

和警告的，如图 2.11.4 最后一行所示，0 个错误和 0 个警告。错误是告诉编程者，程序无法完成编译；而警告则轻一些，程序能够正确完成编译，但是可能会存在潜在的错误。另外，通过编译报告还可以了解到存储器使用的情况。如图 2.11.4 所示，data=9.0 表示使用了 9 字节的内部 RAM，之所以有小数点，是因为 51 有位变量，可能会使用非整数字节的 RAM。xdata=0 表示没有使用外部 RAM。code=20 表示使用了 20 字节的代码，即程序需要 20 字节的 FLASH。看一个程序需要多大的 FLASH，应该以这里报告的 code 数量为准，而不是 HEX 文件的大小。因为 HEX 文件是用 ASCII 码来存储数据的，并且有地址信息、校验信息等。通常 HEX 文件大小是实际代码大小的 2.5 倍左右。

```
Build target 'Target 1'
assembling STARTUP.A51...
compiling Main.c...
linking...
Program Size: data=9.0 xdata=0 code=20
"TestBoard" - 0 Error(s), 0 Warning(s).
```

图 2.11.4 输出窗口的编译报告

2.11.3 KEIL 工具栏及仿真介绍

首先来看看编译工具栏，如图 2.11.5 所示。通常检查单个文件是否有语法错误时，使用第一个按钮。当文件较多时，使用第二个按钮要比第三个快一些，因为它只会编译被修改过的文件。但是有时编译器可能没有发现文件已经被修改了，这时使用第二个按钮编译时，就会得到不对的结果。这时第三个按钮就派上用场了，它不管文件有没有被修改过，都会全部编译。最后一个按钮很少用，只有在编译时发现有错误或者不小心点到编译按钮时，才会使用它来停止编译。

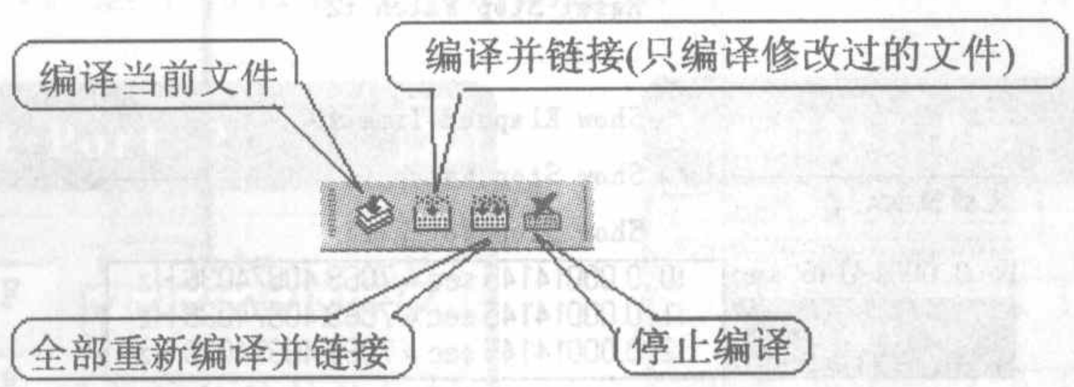


图 2.11.5 编译工具栏

再看看仿真调试工具栏。仿真调试工具栏有两个，一个只有在仿真调试状态时才可见。单击“开始/停止调试”按钮，即可进入到仿真调试状态。如果要停止调试，再单击一次该按钮即可。进入调试状态之后，就可以看到调试工具栏了。通常是在需要查看运行结果的地方，放

置一个断点，然后全速运行，等执行到断点时，程序会自动停下来，然后再使用单步运行来调试。

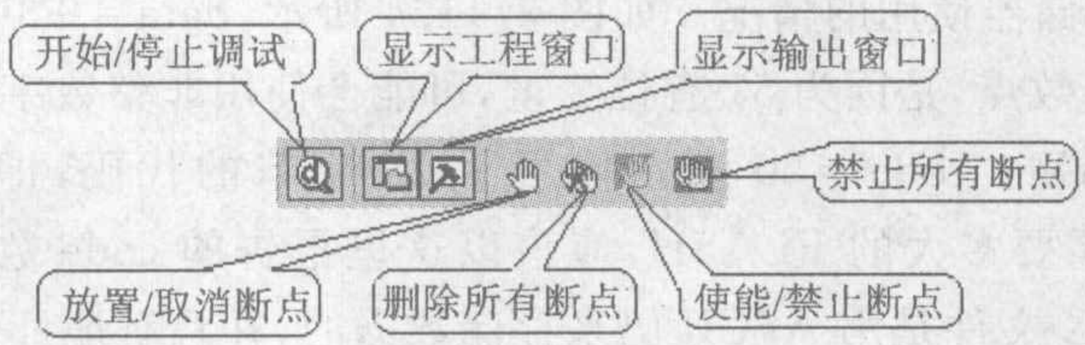


图 2.11.6　仿真及断点工具栏

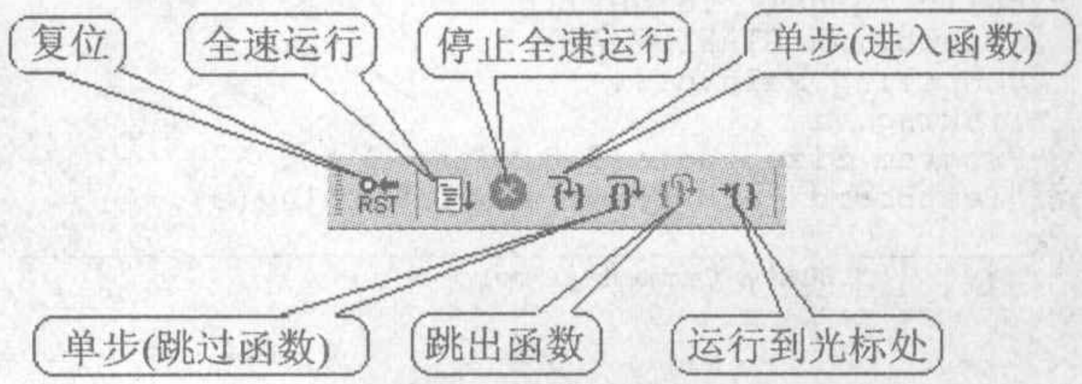


图 2.11.7　调试工具栏

进入到调试状态之后，Peripherals 菜单下就会多出几个子菜单，在这里可以查看中断、I/O口、串口、定时器等状态。注意，要让这些状态能够实时地显示，需要将 View 菜单中的 Periodic Window Update 勾选上。另外，UV3 还有一个秒表功能，使用它可以查看运行时间，这在写软件延时、定时器参数设置时很有用。秒表功能在 KEIL 的最下方的状态栏上，如图 2.11.8所示。将鼠标停留在秒表上，可以显示秒表的时间，以及对应的频率。右击秒表，可以选择显示哪个秒表以及清除秒表的计时。

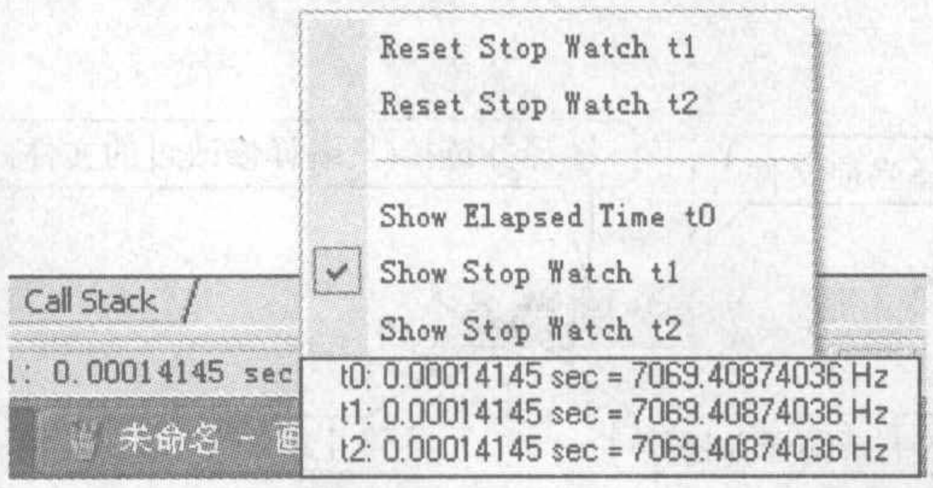


图 2.11.8　UV3 的秒表功能

在调试之前，还需要对工程选项进行一些设置。在工程窗口中，右击 Target 1，在弹出的菜单中选择“Options for Target ‘Target 1’”，这时会出现一个对话框。在这个对话框中，有很多个选项卡。其中，Device 选项卡可选择器件，跟前面介绍的那个选择器件的页面一样；

Target 选项卡主要设置时钟频率以及存储器情况。在这里，将时钟频率 Xtal(MHz)设置为 22.1184，因为板子用的是这个频率的晶体。旁边的 Use On-chip ROM 勾选上，其他使用默认值，如图 2.11.9 所示。Output 选项卡中，将 Create HEX File 勾选上，这样编译器才会生成一个 HEX 文件。其他几个选项卡中的内容不用改动，使用默认值即可。

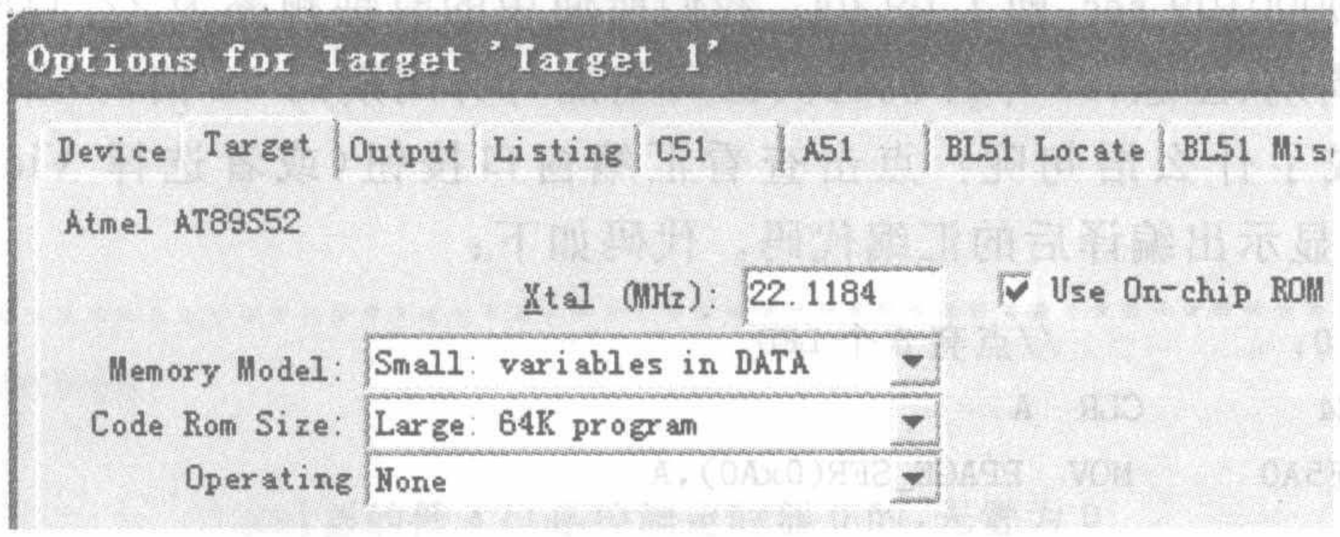


图 2.11.9 Target 标签选项的设置

将工程选项设置好之后，就可以开始仿真程序了。单击“开始调试”按钮，进入到调试状态。这时程序会自动运行到 main 函数，并停在“P2=0;”这条代码处。这时观察秒表，显示值为 0.00021105 sec，表示程序从开始执行到这条语句的时间为 0.000 211 05 s。这些主要是启动代码执行的时间。然后右击秒表，将秒表 1 清 0。然后将 Peripherals 菜单下的 I/O-Ports 中的 Port 2 勾选上，这样就可以看到 P2 口的每个引脚电平了，打勾的表示高电平 1，没打勾的表示低电平 0。显示有两行，上面一行表示程序设置的状态，而下面一行表示引脚实际的电平状态，如图 2.11.10 所示。下面一行的值可以修改，来模拟外部电平输入。注意：51 单片机在输出低电平时，是强驱动的，这时不能强行将一个高电平加在端口上。在仿真中也是如此，如果上面一行没打勾，而将下面一行打勾时，就会弹出一个出错的对话框。输出为高电平时，是弱驱动，可以直接将低电平加在端口上。因此当我们需要将 I/O 口作为输入口使用时，应该将对应的端口设置为 1。这个特性就是所谓的准双向 I/O 口，51 单片机的特色之一。

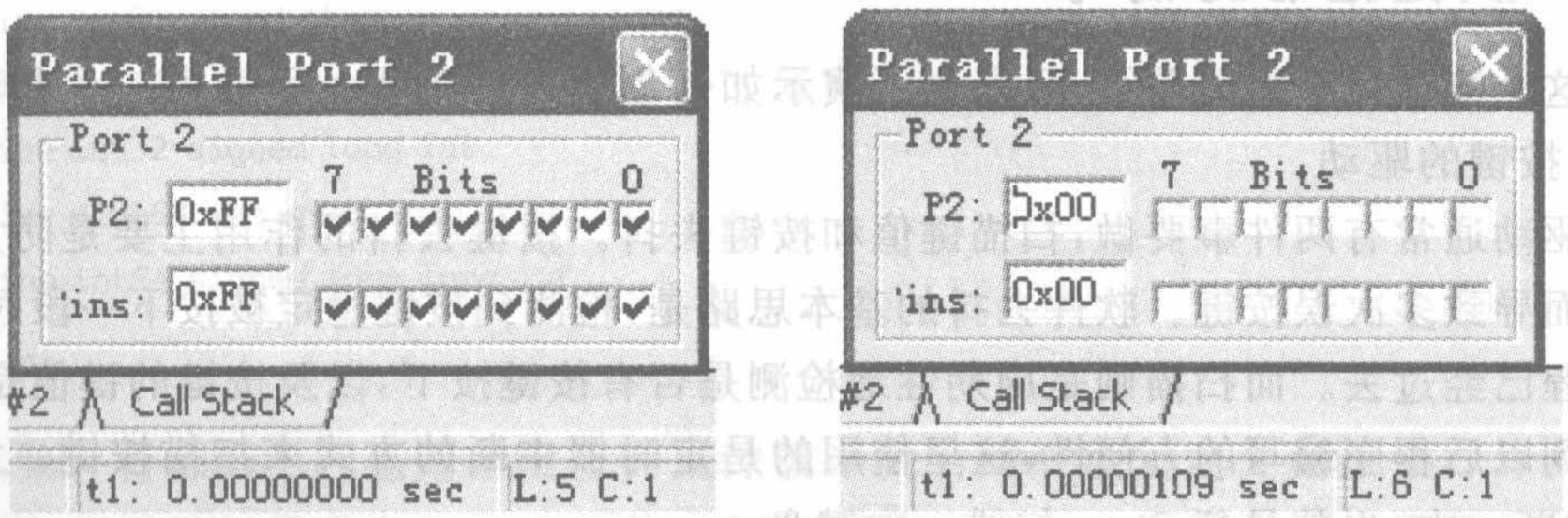


图 2.11.10 仿真时端口及秒表状态的变化

51单片机在复位后，4组I/O口均为高电平，所以这时看到的I/O口是全部打勾的。然后选择单步运行，或者按F10或F11键，执行“P2=0;”这条语句，可以看到：P2的所有端口都变成不打勾状态了，即输出全为0。如果将这个程序下载到USB学习板中，运行后8个连接在P2口上的LED应该全部点亮。图2.11.10是运行前后I/O口状态的对比，这时观察一下秒表，时间为0.00000109 sec，即1.09 μs。我们所使用的时钟频率为22.1184 MHz，1.09 μs相当于2个机器周期(注意，51单片机的机器周期是时钟周期的12倍)。那么这条P2=0语句，到底被编译成了什么语句呢？点击查看汇编窗口按钮(或者选择View→Disassembly Window)，就可以显示出编译后的汇编代码。代码如下：

```
    5:  P2 = 0;              //点亮8个LED
C:0x000F    E4        CLR  A
C:0x0010    F5A0      MOV  PPAGE_SFR(0xA0),A
```

由代码可知，它生成了“CLR A”和“MOVE P2,A”两条指令，其中0xA0这个地址刚好是P2的地址。这两条指令都是单机器周期的，所以P2=0这条C语句执行的时间就是2个机器周期。由此可以看出，一条C语句并不是对应着一条汇编指令的，通常是一条C语句对应着多条汇编指令，特别是对于较复杂的表达式和算术运算时。当然，有时一条C语句并不产生汇编代码，也可能刚好是一条C语句对应着一条汇编指令。通过查看汇编代码，可以计算出运行时间。但是这需要知道每条指令执行的时间和整个执行过程，当指令比较多或者有很多循环和分支时，计算起来就没那么容易了。圈圈是懒人一个，肯定不会去一个个计算了，所以就使用仿真的办法来决定运行时间。嘿嘿！

有时觉得自己写的代码没问题，但是运行结果就是不对时，可以考虑看看生成的汇编代码是怎样的，通常会有意外的收获。例如，有些代码被删除(优化)掉了。当然，看汇编代码时要注意，编译器有时会帮你做一些优化工作，并不按照所写的代码来生成代码(这在程序分支时经常会遇到)，但最终的运行结果可能是正确的。

2.11.4 按键驱动的编写

前面这个测试程序太简单了，只是为了演示如何创建一个工程。下面做一个相对复杂一些的程序：按键的驱动。

按键驱动通常有两件事要做：扫描键值和按键去抖。按键去抖的作用主要是防止开关按下时抖动而导致多次误按键。软件去抖的基本思路是，检测到按键稳定被按下一段时间后，认为抖动过程已经过去。而扫描则是周期性地检测是否有按键按下，以及按键的键值是多少。

考虑到以后程序编写的方便性，这里使用的是定时器中断的方式来扫描按键。定时器每隔5 ms中断一次，也就是每5 ms扫描一次键盘。

新建立一个key.c和key.h的文件，将key.c添加到工程中。再新建一个config.h的头文件，这个头文件用来保存一些基本的配置信息。

我们先写 key.c 文件。由于要使用定时器中断做按键扫描，所以要先初始化好一个定时器。这里选择定时器0作为按键扫描的定时器，而定时器1留作串口波特率发生器。初始化定时器就是设置定时器的工作模式，启动定时器等。代码如下：

```
/*****************************************************************
函数功能:定时器0初始化,用来做按键扫描
入口参数:无
返　　回:无
备　　注:无
*****************************************************************/
void InitTimer0(void)
{
 TMOD&=0xF0;          //定时器的低4位是控制定时器0的,先置为0
 TMOD|=0x01;          //然后再将最低位设置为1,最终选择16位定时器模式
 ET0=1;               //允许定时器0中断
 TR0=1;               //启动定时器0
}
/*****************************************************************/
```

扫描按键，需要一些变量来保存一些状态及最后的扫描结果，因此增加几个全局变量。为了方便书写和移植，将数据类型关键字做了重新定义，这些定义被放在 MyType.h 中，新建一个 MyType.h 文件，放入以下代码：

```
#ifndef __MY_TYPE_H__
#define __MY_TYPE_H__

#define uint8 unsigned char
#define uint16 unsigned short int
#define uint32 unsigned long int
#define int8 signed char
#define int16 signed short int
#define int32 signed long int
#define uint64 unsigned long long int
#define int64 signed long long int

#endif
```

其中，第一行和第二行是用来防止头文件被多次引用时重复定义而报错。#ifndef 用于编译时检查，意思是如果没有定义后面的__MY_TYPE_H__，则需要编译下面的内容，直到遇到#else 或者#endif。假设这个头文件被同一个 C 文件包含了两次，那么在第一次检查时，__MY_TYPE_H__是没有被定义的，因而就编译了中间那部分。在第二次引用检查时，由

于此时__MY_TYPE_H__已经被定义了(因为前面编译时有一句#define __MY_TYPE_H__)，所以就会直接跳过这部分代码而不编译它们。__MY_TYPE_H__是一个唯一的标志，它跟变量命名规则是一样的，通常写成两个下划线加文件名(文件名的“.”换成一个下划线)加两个下划线的格式。这种用宏定义来控制代码是否被编译的方法是经常要用到的，大家一定要掌握。

刚刚说到全局变量的定义，定义好之后还需要对它们进行初始化。另外，键盘对应的I/O口要设置为输入状态，调用定时器0初始化函数等。因此，专门写了一个对键盘初始化的函数。变量的定义和初始化的代码如下：

```
volatile uint8 idata KeyCurrent,KeyOld,KeyNoChangedTime;
volatile uint8 idata KeyPress;
volatile uint8 idata KeyDown,KeyUp,KeyLast;
volatile uint8 KeyCanChange;

/*****************************************************************
函数功能:键盘初始化
入口参数:无
返    回:无
备    注:无
*****************************************************************/
void InitKeyboard(void)
{
 KeyIO = 0xFF;                    //键盘对应的口设置为输入状态
 KeyPress = 0;                    //无按键按住
 KeyNoChangedTime = 0;            //按键按下状态未改变的时间为0
 KeyOld = 0;                      //上一次按键值为0(没有按键按下)
 KeyCurrent = 0;                  //当前按键值为0(没有按键按下)
 KeyLast = 0;                     //最后一次按键值为0(没有按键按下)
 KeyDown = 0;                     //没有按键按下
 KeyUp = 0;                       //没有按键释放
 InitTimer0();                    //初始化定时器
 KeyCanChange = 1;                //允许键值改变
}
/*****************************************************************/
```

在变量定义中使用了volatile和idata两个特殊的关键字。关键字volatile在嵌入式系统中经常用到。它的意思是告诉编译器，不要优化对这个变量的操作，因为这个变量的值可能会被意外地修改而需要每次都重新读回(例如，在中断中被修改，或者被硬件修改，如各种I/O口、标志寄存器)，或者每次赋值都需要回写(例如，在某个地址要它产生一个波形，就会对它进行一系列的赋值操作。如果定义为普通变量，编译器可能会认为前面的赋值是无用的而只保

留最后一次赋值操作，或者干脆把值放在了工作寄存器中，而不回写到内存，但这并不是我们所需要的）。idata 是 Keil C 中将变量分配到高 128 字节 RAM（只能使用间接寻址）的关键字。编译器会优先使用低 128 字节的 RAM，也就是说，即使加了 idata，也不一定会将变量分配在高 128 字节。但如果变量超过 128 字节，而没有使用 idata，编译器将会报 RAM 不够的错误。

KeyCurrent、KeyOld、KeyLast 和 KeyNoChangedTime 是扫描按键使用的变量，应用程序不直接使用它们。KeyPress、KeyDown、KeyUp、KeyCanChange 是提供给应用程序使用的变量，KeyPress 表示当前被按住不放的键，KeyDown 表示新按下的键，KeyUp 表示新松开的键，KeyCanChange 是应用程序用来控制是否允许新的扫描。当某个键按下时，则 KeyPress 对应的位被置 1，并且 KeyDown 对应的位也置 1；当按键松开后，KeyPress 对应的位为 0，KeyUp 对应的位被置 1。由于定时器 0 是 1 号中断，所以后面使用关键字 interrupt 1 修饰。注意，中断函数不能有入口参数也不能有返回参数。定时器中断服务的程序如下：

```
/*****************************************************************
函数功能:定时器 0 中断处理
入口参数:无
返      回:无
备      注:22.1184 MHz 晶体约 5 ms 中断一次
*****************************************************************/
void Timer0Isr(void) interrupt 1
{
//定时器 0 重装,定时间隔为 5 ms,加 15 是为了修正重装所花费时间
//这个值可以通过软件仿真来确定,在这里设置断点,调整使两次运行时间差刚好为 5 ms 即可
 TH0 = (65536 - Fclk/1000/12 * 5 + 15)/256;
 TL0 = (65536 - Fclk/1000/12 * 5 + 15) % 256;
if(!KeyCanChange)return;              //如果正在处理按键,则不再扫描键盘
//开始键盘扫描,保存按键状态到当前按键情况
//KeyCurrent 总共有 8 bit,当某个开关按下时,对应的位为 1
KeyCurrent = GetKeyValue();           //读取键值,GetKeyValue()其实是个宏,不是函数,
                                      //这里故意写成函数的样子,美观。它的定义在 key.h 文件中
if(KeyCurrent! = KeyOld)              //如果两次值不等,则说明按键情况发生了改变
 {
  KeyNoChangedTime = 0;               //键盘按下时间为 0
  KeyOld = KeyCurrent;                //保存当前按键情况
  return;                             //返回
 }
else
```

```
    {
      KeyNoChangedTime ++ ;                    //按下时间累计
      if(KeyNoChangedTime> = 1)                //如果按下时间足够
      {
        KeyNoChangedTime = 1;
        KeyPress = KeyOld;                     //保存按键
        KeyDown| = (~KeyLast)&(KeyPress);      //求出新按下的键
        KeyUp| = KeyLast&(~KeyPress);          //求出新释放的键
        KeyLast = KeyPress;                    //保存当前按键情况
      }
    }
  }
/*****************************************************************/
```

在 key.h 文件中，我们将所定义的变量和函数进行声明，以便其他文件可以使用 key.c 中的全局变量。注意变量声明时需要在前面增加 extern，并且不要有赋初始值的操作。另外，还需要定义读取按键的宏和按键值的宏，不直接在程序中使用 I/O 口和数值是为了方便程序的移植和增加可读性。key.h 文件的内容如下：

```
#ifndef __KEY_H__
#define __KEY_H__

#include <at89x52.h>
#include "MyType.h"

extern volatile uint8 idata KeyCurrent,KeyOld,KeyNoChangedTime;
extern volatile uint8 idata KeyPress;
extern volatile uint8 idata KeyDown,KeyUp,KeyLast;

extern volatile uint8 KeyCanChange;

void InitKeyboard(void);

#define KeyIO P1
//获取按键值的宏。由于开关按下为 0,所以对结果取反
#define GetKeyValue() (~KeyIO)

#define KEY1 0x01
#define KEY2 0x02
#define KEY3 0x04
#define KEY4 0x08
#define KEY5 0x10
#define KEY6 0x20
#define KEY7 0x40
```

```
#define KEY8 0x80

#endif
```

还有一个 config.h 文件，它里面保存的是一些配置信息。例如，时钟频率、波特率。它的内容如下：

```
#ifndef __CONFIG_H__
#define __CONFIG_H__

#define Fclk        22118400UL          /*使用 22.1184 MHz 晶体*/
#define BitRate     9600UL              /*波特率定义为 9600*/

#endif
```

其中，UL 的意思是无符号长整数。因为默认为整型，Keil C51 中整数是两个字节的，最大只能到 65 535，这里已经超过，因而需要增加 UL，表示长整。在 key.c 开始处增加对 config.h和 key.h 两个头文件的引用(还记得尖括号和双引号的区别吗？这里应该使用双引号)，不然编译会通不过。

有了这个按键驱动，就可以用它来控制 LED 了。在控制 LED 之前，再来写一个 LED.h 文件。在前面那个简单的测试程序中，是直接用 I/O 口来操作 LED 的。这种方法不是很好。当硬件发生改动时，每个操作 LED 的地方都需要修改，一来麻烦，二来容易出错。因此在 LED.h 文件中对 LED 的操作进行了重新定义，代码如下：

```
#ifndef __LED_H__
#define __LED_H__

//全部 LED
#define LEDs   P2

//单个 LED,LED1～LED7
//注意"^"这个操作符，只有跟 sbit 搭配时才表示定义一个位，
//C 语言中，"^"表示异或操作，不要在程序中直接使用"^"来表示某一位
sbit LED1 = LEDs^0;
sbit LED2 = LEDs^1;
sbit LED3 = LEDs^2;
sbit LED4 = LEDs^3;
sbit LED5 = LEDs^4;
sbit LED6 = LEDs^5;
sbit LED7 = LEDs^6;
sbit LED8 = LEDs^7;

//点亮某个 LED
#define OnLed1()   LED1 = 0
```

```
#define OnLed2()   LED2 = 0
#define OnLed3()   LED3 = 0
#define OnLed4()   LED4 = 0
#define OnLed5()   LED5 = 0
#define OnLed6()   LED6 = 0
#define OnLed7()   LED7 = 0
#define OnLed8()   LED8 = 0

//关闭某个 LED
#define OffLed1()   LED1 = 1
#define OffLed2()   LED2 = 1
#define OffLed3()   LED3 = 1
#define OffLed4()   LED4 = 1
#define OffLed5()   LED5 = 1
#define OffLed6()   LED6 = 1
#define OffLed7()   LED7 = 1
#define OffLed8()   LED8 = 1

#endif
```


然后将 main.c 函数修改成如下的样子，就可以用按键来控制 LED 了。按下某个按键，则对应的 LED 会点亮。

```
#include <AT89X52.H>                    //头文件
#include "Key.h"
#include "Led.h"

void main(void)                         //主函数
{
 EA = 1;                                //打开中断
 InitKeyboard();                        //初始化按键
 while(1)                               //死循环
 {
 LEDs = ~KeyPress;                      //将按键结果取反后控制 LED
 }
}
```

2.11.5 串口驱动的编写

前面的 LED 只能表示少量信息，而通过串口，则可以显示大量的信息。这在调试时很有帮助，可以将一些需要的信息通过串口发送到计算机上，把计算机当作一个显示屏来用。另外，在 USB 转串口实验中，串口驱动也必不可少。

在使用串口之前，必须对串口进行初始化，包括设置串口的工作模式、设置波特率等。新建一个 UART.c 和一个 UART.h 文件，并将 UART.c 文件添加到工程中。在 UART.c 中增加一个串口初始化的函数，代码如下（其中，BitRate 是在 config.h 中设置的波特率。记住在文件开始要引用头文件）：

```
#include <at89x52.H>

#include "UART.h"
#include "MyType.h"
#include "config.h"

/*****************************************************************************
函数功能：串口初始化
入口参数：无
返　　回：无
备　　注：无
*****************************************************************************/
void InitUART(void)
{
 EA = 0;                                  //暂时关闭中断
 TMOD& = 0x0F;                            //定时器 1 模式控制在高 4 位
 TMOD| = 0x20;                            //定时器 1 工作在模式 2，自动重装模式
 SCON = 0x50;                             //串口工作在模式 1
 TH1 = 256 - Fclk/(BitRate * 12 * 16);    //计算定时器重装值
 TL1 = 256 - Fclk/(BitRate * 12 * 16);
 PCON| = 0x80;                            //串口波特率加倍
 ES = 1;                                  //串行中断允许
 TR1 = 1;                                 //启动定时器 1
 REN = 1;                                 //允许接收
 EA = 1;                                  //允许中断
}
//////////////////////End of function/////////////////////////////
```

然后，还需要再增加一个串口中断处理函数。由于串口的中断号是 4，因此后面使用 interrupt 4。我们这个串口驱动只发送数据，所以接收到数据就不管它了，只是简单地将中断标志清除即可。而数据发送中断，也比较简单，清除中断标志后再将一个正在发送状态的变量清除即可。正在发送状态的标志需要自己增加一个变量。最后的代码如下：

```
volatile uint8 Sending;
/*****************************************************************************
函数功能：串口中断处理
```

```
入口参数:无
返      回:无
备      注:无
*************************************************************************/
void UartISR(void) interrupt 4
{
 if(RI)              //收到数据
 {
  RI = 0;            //清中断请求
 }
 else                //发送完1字节数据
 {
   TI = 0;
   Sending = 0;      //清正在发送标志
 }
}
//////////////////////End of function//////////////////////////////
```


再增加三个函数，功能分别是发送单个字符；发送一个字符串；以HEX格式发送一个整数。最后一个函数用来发送D12的ID号，后面会用到。这三个函数的代码如下：

```
/*************************************************************************
函数功能:往串口发送1字节数据
入口参数:d  要发送的字节数据
返      回:无
备      注:无
*************************************************************************/
void UartPutChar(uint8 d)
{
 SBUF = d;              //将数据写入到串口缓冲
 Sending = 1;           //设置发送标志
 while(Sending);        //等待发送完毕
}
//////////////////////End of function//////////////////////////////
/*************************************************************************
函数功能:发送一个字符串
入口参数:pd  要发送的字符串指针
返      回:无
备      注:无
*************************************************************************/
```

```
void Prints(uint8 * pd)
{
 while((*pd)!='\0')                    //发送字符串,直到遇到 0 才结束
 {
  UartPutChar(*pd);                    //发送一个字符
  pd++;                                //移动到下一个字符
 }
}
//////////////////////End of function//////////////////////////////
code uint8 HexTable[]={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
/*****************************************************************
函数功能:将短整数按十六进制发送
入口参数:待发送的整数
返　　回:无
备　　注:无
*****************************************************************/
void PrintShortIntHex(uint16 x)
{
 uint8 i;
 uint8 display_buffer[7];
 display_buffer[6]=0;
 display_buffer[0]='0';
 display_buffer[1]='x';
 for(i=5;i>=2;i--)                     //将整数转换为 4 字节的 HEX 值
 {
  display_buffer[i]=HexTable[(x&0xf)];
     x>>=4;
 }
 Prints(display_buffer);
}
//////////////////////End of function//////////////////////////////
```

在 UART.h 文件中增加对 UART.c 文件中的函数的声明。

有了这个串口驱动,就可以将需要显示的信息通过串口发送出来了。在 main.c 文件中,增加对 UART.h 的引用,然后在 main 函数中增加初始化串口的函数调用,之后就可以使用串口了。这里增加显示一个信息头,然后再增加对按键按下、释放操作的显示。具体的代码这里就不给出了,可以参看光盘中的 TestBoard 工程下的 main.c 文件。

编译这个工程,然后开始调试,点击全速运行后,再点击 Keil 的仿真串口按钮(或者选择菜单项 View→Serial Window #1),打开串口对话框,可以显示发送到串口的信息,如

图 2.11.11所示；也可以直接在这个对话框用键盘输入，不过它是没有回显的，即看不到所输入的字符，但是串口是可以接收到这些数据的。在串口中断中也可以设置一个断点，然后再在串口对话框中敲键盘试试，这时可以看到它的确进入串口中断了，并且是串口接收中断。

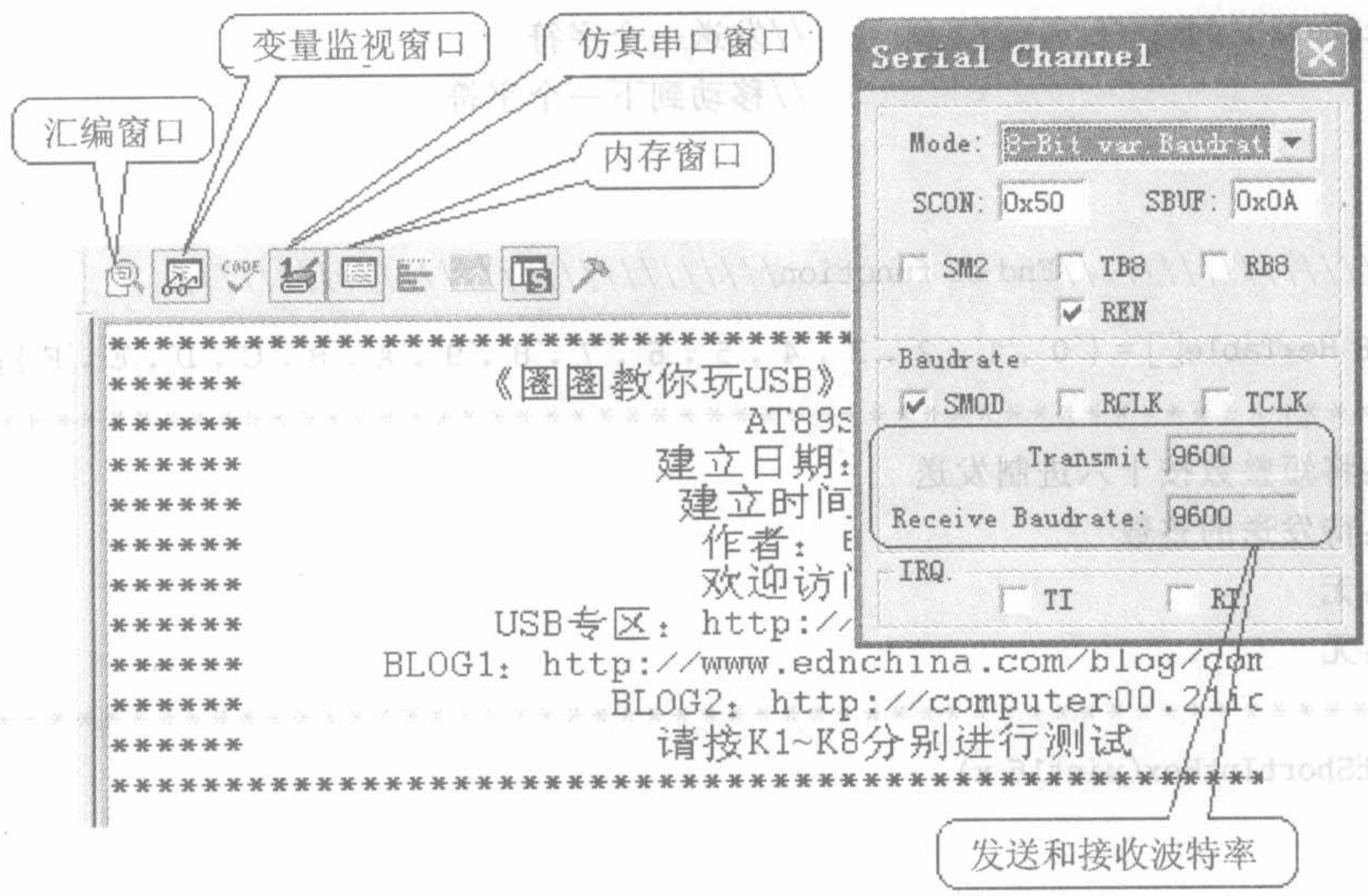


图 2.11.11 仿真串口窗口和串口状态窗口

另外，在 Peripherals 菜单下有个 Serial 的子菜单，点击它可以看到当前串口的设置状态，例如工作模式，收发的波特率等。通过它，可以确认波特率是否设置对了。当然，前提是必须要在工程选项中设置正确的时钟频率。另外，这个窗口也有个 SBUF 的值，它只能看到程序发送出去的串口数据，而不能看到接收到的数据。要看到接收到的数据，应该在串口接收中断处理中将 SBUF 的值赋给一个变量。

从返回的信息可以看到，有个建立日期和建立时间。这两个值不是自己去修改为实际的时间的，而是直接使用编译器提供的内部宏来实现的。编译器提供了__DATE__和__TIME__两个宏。这是两个字符串。当编译程序时，编译器会自动使用当前的系统时间来生成这两个字符串。另外，还有两个系统定义的宏：__FILE__和__LINE__。分别表示当前的文件名和当前所在行号，有时可以用它们来显示一些错误信息，并报告错误发生在哪个文件的哪一行。

在这里顺便提一下，Keil 仿真的变量监视窗口和内存窗口，如图 2.11.12 所示。左边的是变量监视窗口，下面有一行标签可以选择不同的观察对象。Locals 表示局部变量，当调试进入一个函数后，里面的局部变量会自动在此显示。后面的 Watch #1 等可以自己输入需要查看的变量。Name 是变量名，Value 是变量的值，右击它可以选择不同的显示格式。右边的是内存窗口，在这里敲入地址可以显示从该地址开始的内存的值。如果不写前缀，直接敲入地址再回车，显示的就是代码空间 ROM 的值。

使用前缀，可以查看到不同存储空间的内存值。

- 使用前缀 D 加冒号，可以显示直接寻址的 256 字节的内容（包括特殊寄存器区）；
- 使用前缀 I 加冒号，可以显示间接寻址的 256 字节内容（包括前 128 字节 RAM 和高 128 字节 RAM）；
- 使用前缀 X 加冒号，可以显示外部 RAM（XRAM）存储区的数据；
- 使用前缀 C 加冒号，与直接使用地址是一样的，显示的是代码空间 ROM 的值。

图 2.11.12 所显示的就是 ROM 地址 0 的值，前三个地址的值为 020806。其实它是一条跳转指令的机器码，跳转的目的地址是 0806。这条指令写成汇编语句就是 LJMP 0806H。大家知道，地址 0 是复位地址，所以程序一开机运行，就会跳转到 0806 去执行。而在地址 0x0B 处，是定时器 0 中断入口地址，这里的值是 020705，那么应该就是一条 LJMP 0705H 指令，跳转到 0705 处去执行。据此推测，0705 应该就是定时器 0 中断服务函数（即 Timer0Isr 函数）的入口地址。此问题留给读者自己去验证。

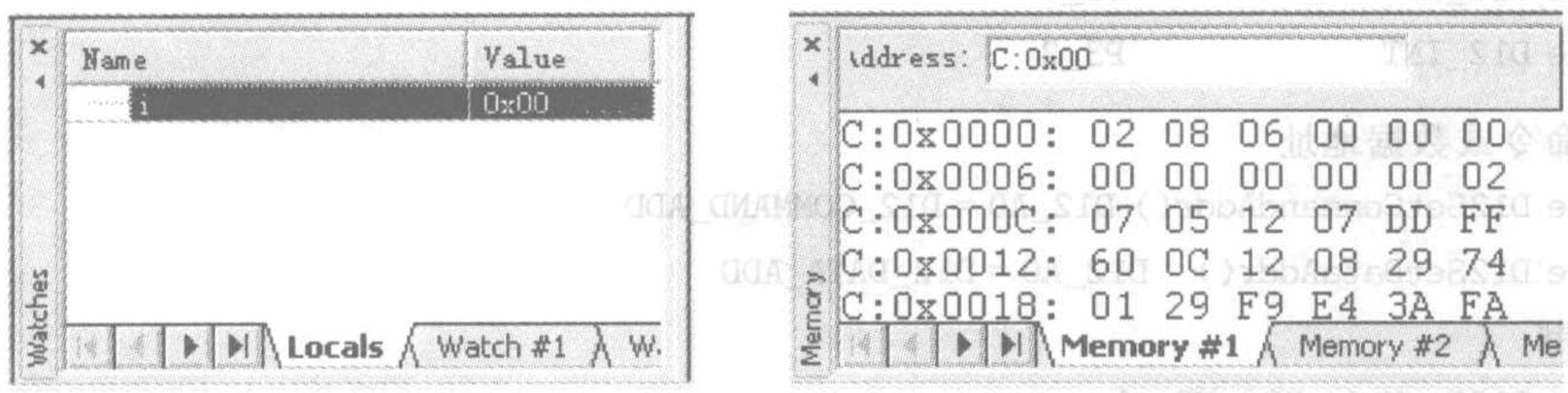


图 2.11.12 变量监视窗口和内存窗口

2.11.6 PDIUSBD12 读写函数及读 ID 的实现

如何知道 D12 是否正常工作了呢？这就需要写个简单的测试程序来测试一下。D12 芯片提供了一条读取 ID 号的命令，如果正确读取到了 ID 号，则说明芯片应该是工作起来了。当然也存在着某条数据线出问题的可能性，毕竟仅这个指令中有些位是 0。不过，这条读 ID 的命令应该是芯片厂商内部使用的，它并没有在数据手册中公布，所以在数据手册中是找不到这条命令的。

要读取 ID 号，首先要实现写命令和读数据的两个基本函数。新建 PDIUSBD12.c 和 PDIUSBD12.h 文件，在 PDIUSBD12.c 文件中增加读写函数，在 PDIUSBD12.h 中增加对函数的声明，然后在 main.c 文件中就可以使用读 ID 的函数了。

那么写命令和读数据的函数该怎么写呢？这就要根据硬件连接和 D12 的数据手册提供的时序图来编写了。

首先，在 PDIUSBD12.h 文件中定义一些硬件相关的宏来代替直接对 I/O 操作。对照电路图可知：D12 的数据口接在 P0 口，A0 接在 P35，WR 接在 P36，RD 接在 P37，INT 接在 P32。

PDIUSBD12.h 文件的内容如下：

```
#ifndef __PDIUSBD12_H__
#define __PDIUSBD12_H__

#include <at89x52.h>
#include "MyType.h"

//命令地址和数据地址
#define D12_COMMAND_ADD     1
#define D12_DATA_ADD        0

//PDIUSBD12 芯片连接引脚
#define D12_DATA            P0
#define D12_A0              P3_5
#define D12_WR              P3_6
#define D12_RD              P3_7
#define D12_INT             P3_2

//选择命令或数据地址
#define D12SetCommandAddr() D12_A0 = D12_COMMAND_ADD
#define D12SetDataAddr()    D12_A0 = D12_DATA_ADD
//WR 控制
#define D12SetWr() D12_WR = 1
#define D12ClrWr() D12_WR = 0
//RD 控制
#define D12SetRd() D12_RD = 1
#define D12ClrRd() D12_RD = 0
//获取中断状态
#define D12GetIntPin()  D12_INT
//读写数据
#define D12GetData() D12_DATA
#define D12SetData(Value) D12_DATA = (Value)

//将数据口设置为输入状态,51 单片机端口写 1 就是为输入状态
#define D12SetPortIn() D12_DATA = 0xFF
//将数据口设置为输出状态,由于 51 单片机是准双向 IO 口,所以不用切换,为空宏
#define D12SetPortOut()
//D12 的读 ID 命令
#define Read_ID  0xFD
//函数声明
void D12WriteCommand(uint8);
```

```
uint8 D12ReadByte(void);
uint16 D12ReadID(void);
#endif
```

下面，按照数据手册提供的时序图来实现写命令和读数据的函数了。D12 的读写时序如图 2.11.13 所示。

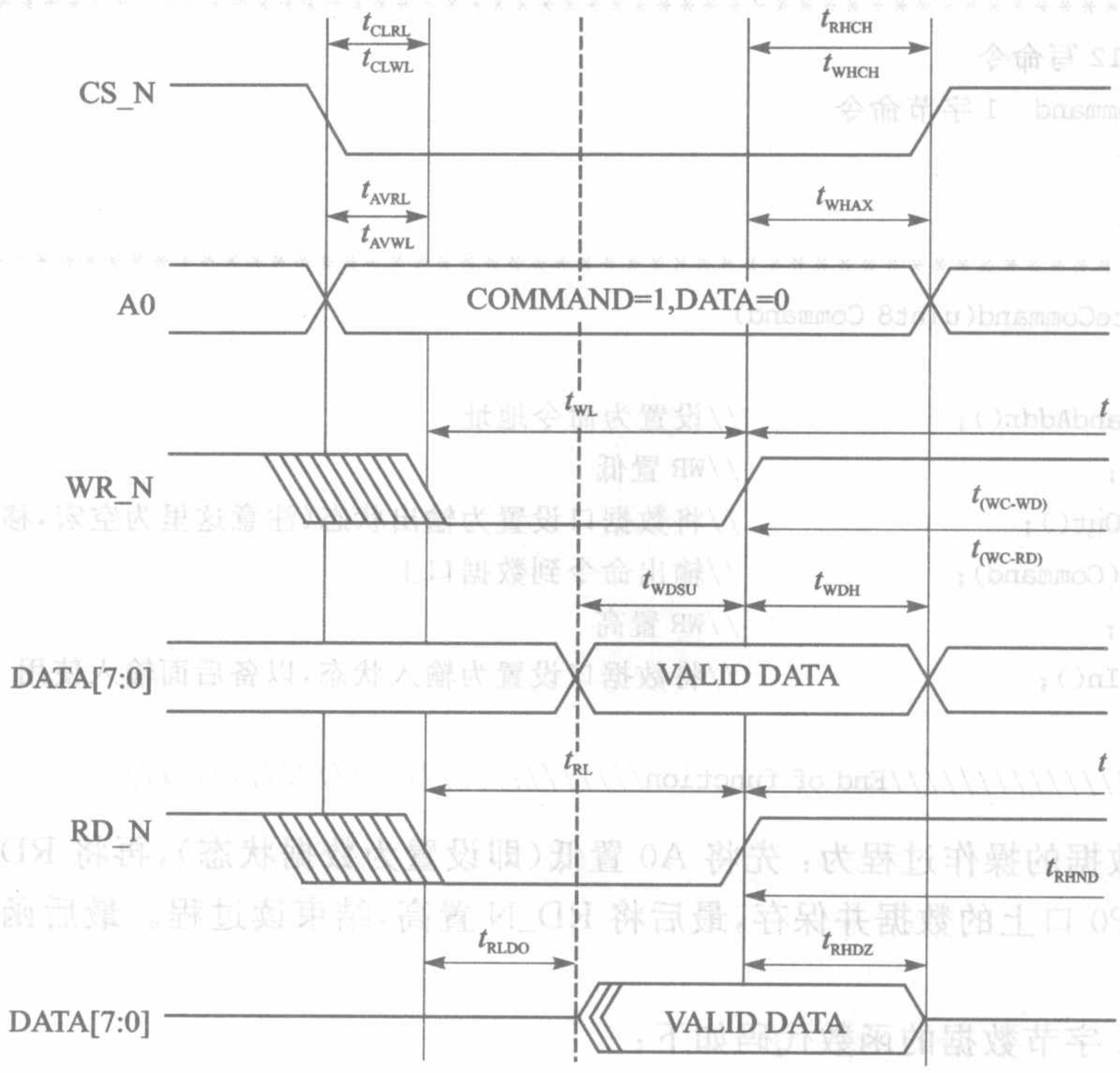


图 2.11.13　D12 的读写时序图

图 2.11.13 中，CS_N 是片选信号。这里表示的意思是，当片选信号为低电平时，下面的操作才有意义。由于板子上是将 CS_N 固定接地的，所以它一直保持为低电平，因此程序不用管 CS_N 引脚。A0 是地址线，用于选择是命令还是数据。A0 为 1 时表示操作的是命令，A0 为 0 时表示操作的是数据。WR_N 是写信号，表示 WR_N 的上升沿将数据写入到芯片中。数据必须在上升沿的前后稳定地保持一段时间（即图中的 t_{WDSU} 和 t_{WDH}）才能可靠写入，这些时间参数可以在前面的交流电气参数表格中找到。我们模拟出来的时序必须要满足这些条件才能正确地操作硬件。RD_N 是读选通信号，在读数据时，应该先将 RD_N 置低，等待 t_{RLDD} 时间后，数据将出现在数据总线 DATA[7:0]上，这时可以读取数据。读取完毕之后，将 RD_N 置高，数据总线上的数据将在 t_{RHDZ} 时间后消失。

按照以上的分析，可以得出写命令的操作过程为：先将 A0 置高(即设置为命令状态)，再将 WR_N 置低，把需要发送的命令放到数据总线上，再将 WR_N 置高。这将产生一个上升沿，从而把数据写入到 D12 中。写完后，须将总线设置为输入状态，以避免总线冲突。如果是写数据的操作，只要将第一步的 A0 设置为低即可(即设置为数据状态)。

最终的写命令函数代码如下：

```
/*****************************************************************
函数功能：D12 写命令
入口参数:Command  1 字节命令
返      回:无
备      注:无
*****************************************************************/
void D12WriteCommand(uint8 Command)
{
 D12SetCommandAddr();              //设置为命令地址
 D12ClrWr();                       //WR 置低
 D12SetPortOut();                  //将数据口设置为输出状态(注意这里为空宏，移植时可能有用)
 D12SetData(Command);              //输出命令到数据口上
 D12SetWr();                       //WR 置高
 D12SetPortIn();                   //将数据口设置为输入状态，以备后面输入使用
}
//////////////////////End of function////////////////////////////
```

读 1 字节数据的操作过程为：先将 A0 置低(即设置为数据状态)，再将 RD_N 置低(表示读数据)，读取 P0 口上的数据并保存，最后将 RD_N 置高，结束读过程。最后函数返回读取到的数据。

最终的读 1 字节数据的函数代码如下：

```
/*****************************************************************
函数功能:读 1 字节 D12 数据
入口参数:无
返      回:读回的 1 字节
备      注:无
*****************************************************************/
uint8 D12ReadByte(void)
{
 uint8 temp;
 D12SetDataAddr();                 //设置为数据地址
 D12ClrRd();                       //RD 置低
 temp = D12GetData();              //读回数据
```

```
 D12SetRd();                          //RD 置高
 return temp;                         //返回读到的数据
}
//////////////////////End of function////////////////////////////
```

有了这两个函数之后，就可以读取 D12 芯片的 ID 号了。

下面编写一个读取 ID 号的函数，调用它之后直接返回 2 字节的 ID 号。读取 ID 号的命令是 0xFD，在前面的头文件中已经有定义了，即 #define Read_ID 0xFD。在程序中直接使用 Read_ID 这个宏即可。

读取 ID 号的函数代码如下：

```
/*****************************************************************
函数功能：读 D12 的 ID
入口参数：无
返　　回：D12 的 ID
备　　注：无
*****************************************************************/
uint16 D12ReadID(void)
{
 uint16 id;
 D12WriteCommand(Read_ID);                //写读 ID 命令
 id = D12ReadByte();                      //读回 ID 号低字节
 id| = ((uint16)D12ReadByte())<<8;        //读回 ID 号高字节
 return id;
}
//////////////////////End of function////////////////////////////
```

接下来在 main.c 文件中增加对 pdiusbd12.h 头文件的引用，再增加如下读取并显示 ID 号的代码，就可以在串口中看到 ID 号了。如果 ID 号正确，那么恭喜你；否则，还需要再检查电路板的焊接情况。

读取并显示 ID 号的函数代码如下：

```
id = D12ReadID();
 Prints("Your D12 chip\´s ID is: ");
 PrintShortIntHex(id);
 if(id == 0x1012)
 {
  Prints(". ID is correct! Congratulations! \r\n\r\n");
 }
 else
```

```
{
 Prints(". ID is incorrect! What a pity! \r\n\r\n");
}
```

如果一切都正确，在串口调试助手上（波特率9600，数据位8位，停止位1位，无硬件流控制）将会显示如图2.11.14所示的信息。其中的日期和时间可能不一样，这个取决于你的编译时间。

图2.11.14 测试程序运行结果显示

2.12 本章小结

本章从最基本的硬件设计说起，到电路板的焊接、调试，程序的编写、调试，以及开发环境的使用等，步骤介绍得很详细。这主要是为了照顾一些初学者，也为了方便圈圈在后面把程序的重点放在USB方面。因为前面所说的这几个基本的测试程序，在后面的每个USB程序中几乎都被用到，例如按键、串口等。虽然本章介绍得比较细，但基本上都是开发过程中要用到的东西，所以请不要嫌圈圈啰唆。

第3章 USB 鼠标的实现

前面两章分别讲述了 USB 协议基础和硬件电路设计、测试程序设计等，本章将以一个实际的例子——USB 鼠标来讲述如何去设计一个 USB 设备。本章将穿插 USB 标准请求、各种标准描述符、报告描述符等重要知识。由于是第一个实例程序，所以本章将会是本书中最长的一章，也是最重要的一章。以后的实例程序都是在此基础上修改而来的。

3.1 USB 鼠标工程的建立

首先将第 2 章中的实例复制一份（整个文件夹复制），将文件夹名改为 UsbMouse。然后进入 UsbMouse 文件夹中，将工程文件名 TestBoard. uv2 改为 UsbMouse. uv2。打开工程，在工程窗口中右击 Target 1，选择 Options for Target Target 1；选择 Target 选项卡，在时钟频率中输入 22. 1184；然后选择 Output 选项卡，在 Name for Executable 文本框中输入 UsbMouse，编译工程。再回到 UsbMouse 文件夹中将原来的所有含有 TestBoard 字样的文件全部删除（当然，留着也无所谓）。

这样就有一个程序的基本框架了，并且含有串口、键盘和 LED 等驱动。然后再修改 main. c 文件，将显示的信息头改一下，并将死循环中的代码删除，只剩下一些初始化代码。

3.2 USB 的断开与连接

在前面的测试程序中，还缺少一个写 1 字节数据的函数。在 PDIUSBD12. c 文件中，增加一个写 1 字节数据的函数，记得还要在相应的头文件中增加函数声明。写 1 字节的函数代码如下：

```
/*****************************************************************
函数功能:写 1 字节 D12 数据
入口参数:Value  要写的 1 字节数据
返    回:无
备    注:无
```

```
*****************************************************************************/
void D12WriteByte(uint8 Value)
{
 D12SetDataAddr();                //设置为数据地址
 D12ClrWr();                      //WR置低
 D12SetPortOut();                 //将数据口设置为输出状态(注意这里为空宏,移植时可能有用)
 D12SetData(Value);               //写出数据
 D12SetWr();                      //WR置高
 D12SetPortIn();                  //将数据口设置为输入状态,以备后面输入使用
}
//////////////////////End of function//////////////////////////////
```

当按下复位开关后,程序重新运行,这时须模拟一个USB拔下的动作,因此在程序的开始处,需要将D12内部的上拉电阻断开。这可以通过D12的设置模式命令来实现。将上拉电阻断开后,需要再延迟一段时间,以便主机确认设备已经断开连接。然后再将D12的上拉电阻连上,这时主机就会检测到设备的插入。下面介绍D12的设置模式命令(Set Mode)。

Set Mode命令的代码是0xF3,它后面跟2字节数据的写入。第一字节是配置字节,第二字节是时钟分频系数。第一字节的详细结构如图3.2.1所示,第二字节的详细结构如图3.2.2所示。

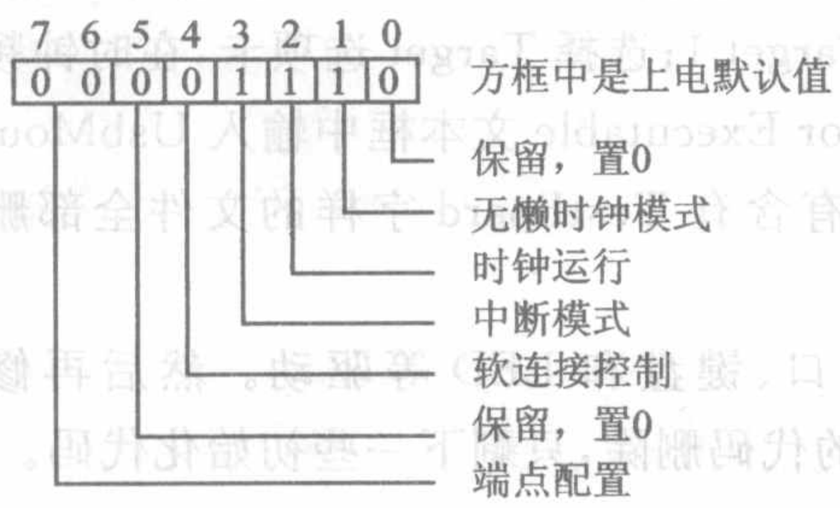


图3.2.1 Set Mode命令的第一字节

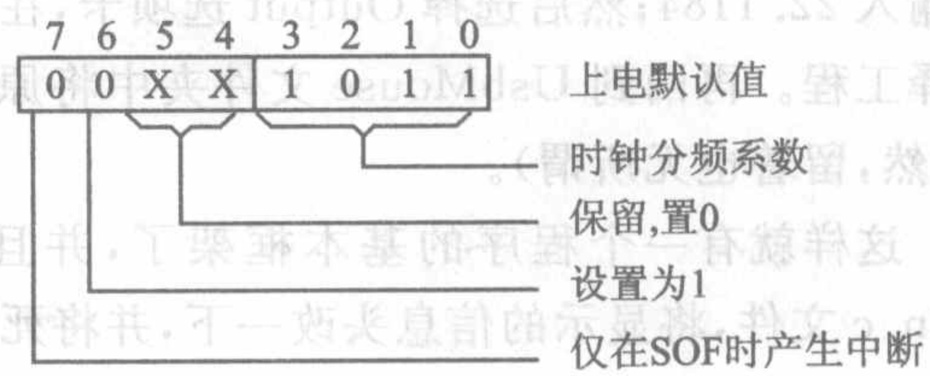


图3.2.2 Set Mode命令的第二字节

Set Mode命令的第一字节各位解释如下:

位0 该位保留,置0。

位1 无懒时钟(低频时钟)模式。该位设置为1表示时钟输出端CLKOUT不会切换到懒时钟模式;该位设置为0表示时钟输出端将在Suspend引脚变高后1 ms切换到懒时钟模式。懒时钟的频率为30×(1±40%) kHz。该位在USB总线复位时不会被改变。

位2 时钟运行。该位为1,表示即使在USB挂起状态下,内部时钟和PLL也会保持运行状态;该位设置为0,表示当时钟不再需要时,内部时钟、晶体振荡器和PLL都将会停止运行。为了能够达到USB协议对总线挂起时严格的电流限制,该位应该设置为0,以节省在挂起状态下的电流消耗。该位在USB总线复位时不会被改变。

位3　中断模式。该位设置为1时，表示所有的错误和NAK都将产生中断请求；该位设置为0时，表示只有当传输正确（对于输出端点，正确接收到数据；对于输入端点，成功发送出数据）时才产生中断请求。该位在USB总线复位时不会被改变。

位4　软连接控制。当该位设置为1，并且V_{BUS}有效（前面说过，V_{BUS}是通过EOT_N检测的）时，就会将上拉电阻连通；当该位设置为0时，上拉电阻被断开。该位在USB总线复位时不会被改变。

位5　该位保留，置0。

位7～6　端点配置选择。可以选择模式0到模式3。这三种具体的模式请参看D12数据手册。模式0为无等时端点，即端点1和端点2都是普通端点，可作中断或批量端点。

Set Mode命令的第二字节各位解释如下：

位3～0　时钟分频系数。假设该值为N，那么CLKOUT端的频率值就是48 MHz除以$N+1$。通过对该值的设置，可以获得不同频率的CLKOUT时钟输出。USB总线复位不会影响该值。

位5～4　保留，置0。

位6　该位必须设置为1。

位7　仅在SOF时产生中断。该位设置为1，只有当帧起始（SOF）时，中断信号才产生。

为了方便调试，我们不考虑节电，时钟都设置为使能状态。中断模式选择为只有正确传输才产生中断，即成功发送出数据或者成功接收到数据后才产生中断。端点配置选择为模式0，即端点1和2都为普通端点，因为这里不需要使用等时传输。由此得出，断开USB连接时该字节的值为0x06，连接USB时该字节的值为0x16。而第二字节，则将分频系数设置为最大，即8分频，从而在CLKOUT端得到6 MHz的时钟频率。通过使用示波器观察该引脚是否为6 MHz，可以检验命令是否正确写入。中断可以在任何时刻产生，不需要仅在SOF时产生，所以第二字节的位7设置为0。因此得出第二字节的值为0x47。

前面说过，在断开USB连接时需要延迟一段时间，这里设置为1 s。因此还需要编写一个延时函数。该函数带一个输入参数，用来决定延时时间。该延时函数用循环语句来实现。通过软件仿真，再调整循环次数，最后得到比较准确的延时时间。为了方便管理，再增加一个UsbCore.c和UsbCore.h文件，大部分跟USB协议相关的代码都放在这里。为了方便调试，增加了一些调试信息，并使用宏来关闭和打开，该宏的定义在config.h中。延时、断开连接、连接的三个函数都放在UsbCore.c中，代码如下：

```
/*****************************************************************
函数功能：延时x毫秒函数
入口参数：x　延时的毫秒数
返　　回：无
备　　注：无
```

```
*****************************************************************/
void DelayXms(uint16 x)
{
 uint16 i;
 uint16 j;
 for(i = 0;i<x;i ++ )
 for(j = 0;j<227;j ++ );                                  //循环语句延时
}
//////////////////////End of function//////////////////////////////
/*****************************************************************
函数功能:USB断开连接函数
入口参数:无
返    回:无
备    注:无
*****************************************************************/
void UsbDisconnect(void)
{
#ifdef DEBUG0
 Prints("断开USB连接。\r\n");
#endif
 D12WriteCommand(D12_SET_MODE);                           //写设置模式命令
 D12WriteByte(0x06);                                      //设置模式的第一字节
 D12WriteByte(0x47);                                      //设置模式的第二字节
 DelayXms(1000);                                          //延迟1 s
}
//////////////////////End of function//////////////////////////////
/*****************************************************************
函数功能:USB连接函数
入口参数:无
返    回:无
备    注:无
*****************************************************************/
void UsbConnect(void)
{
#ifdef DEBUG0
 Prints("连接USB。\r\n");
#endif
 D12WriteCommand(D12_SET_MODE);                           //写设置模式命令
 D12WriteByte(0x16);                                      //设置模式的第一字节
 D12WriteByte(0x47);                                      //设置模式的第二字节
```

```
}
//////////////////////End of function//////////////////////////////
```

进入主函数，完成各种初始化之后，就先调用断开连接函数来断开USB的连接，然后再调用连接USB的函数将上拉电阻接通，这时主机就检测到设备已经插入了。

3.3 USB中断的处理

当D12芯片完成一个操作后，就会产生中断请求信号，以通知CPU来进行相关处理。导致中断发生的事件有USB总线复位、D12进入挂起状态、成功接收到数据和发送完数据等。这里在主程序中一直查询中断引脚电平的状态来判断D12是否有中断发生，从而进行进一步的处理。当然，如果你喜欢的话，也可以改成中断方式，而不用查询。

使用宏#define D12GetIntPin() D12_INT来获取中断引脚状态。由于中断引脚是低电平有效的，所以当它为0时，就表示有中断发生了。知道中断发生后，还须知道具体的中断源是什么，才能对中断进行处理。中断源可以通过读取D12的中断寄存器来获取。读取中断寄存器的命令为Read Interrupt Register，代码为0xF4。发送该命令后可以读取两字节的数据，第一字节中的内容是端点和总线状态的中断，第二字节的内容只有一位有效，是与DMA相关的。由于我们的程序并不使用DMA，所以程序中只保存了中断寄存器的第一字节。第一字节的详细结构如图3.3.1所示。

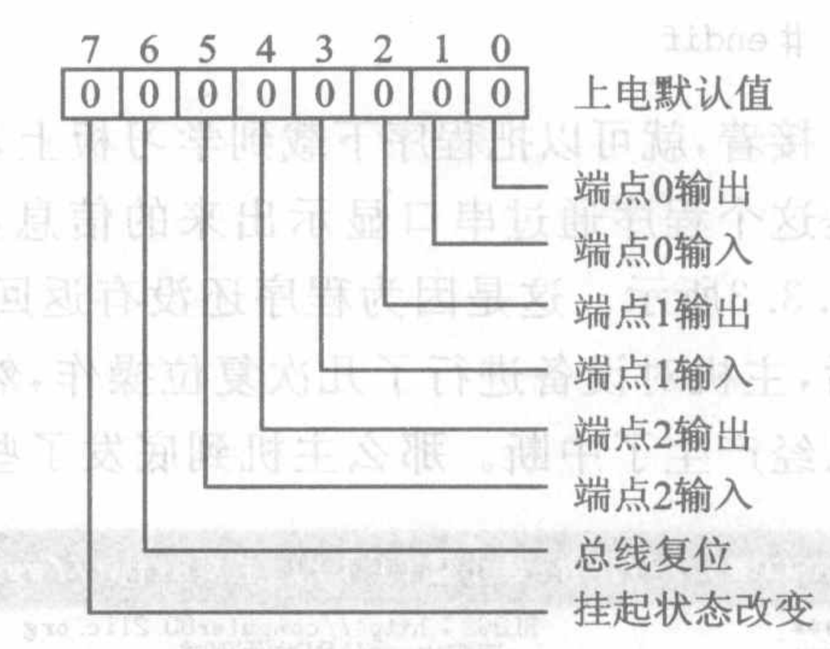


图3.3.1 中断寄存器第一字节结构

如果其中某位的值为1，则表示该中断源发出了中断请求。各端点的中断标志(位0～5)在通过Read Last Transaction Status命令读取端点最后传输状态后，被清零。而另外两位(位6和位7)则在读取本寄存器后，被自动清零。通过判断该寄存器中每一位的值，可以写8个对应的处理函数来处理它们。这8个函数都放在UsbCore.c中，可以只写一个空函数，暂时先不写任何内容(这种编程的方法叫做自顶而下逐步细化)。对中断源处理的代码如下：

```
while(1)                                        //死循环
 {
  if(D12GetIntPin() == 0)                       //如果有中断发生
  {
   D12WriteCommand(READ_INTERRUPT_REGISTER);    //写读中断寄存器的命令
   InterruptSource = D12ReadByte();             //读回第一字节的中断寄存器
   if(InterruptSource&0x80)UsbBusSuspend();     //总线挂起中断处理
```

```
        if(InterruptSource&0x40)UsbBusReset();          //总线复位中断处理
        if(InterruptSource&0x01)UsbEp0Out();            //端点0输出中断处理
        if(InterruptSource&0x02)UsbEp0In();             //端点0输入中断处理
        if(InterruptSource&0x04)UsbEp1Out();            //端点1输出中断处理
        if(InterruptSource&0x08)UsbEp1In();             //端点1输入中断处理
        if(InterruptSource&0x10)UsbEp2Out();            //端点2输出中断处理
        if(InterruptSource&0x20)UsbEp2In();             //端点2输入中断处理
    }
}
```

然后，在每个函数中都写上一句输出调试信息的语句。例如，在总线复位中就可以增加一条显示总线复位的语句：

```
#ifdef DEBUG0
 Prints("USB总线复位。\r\n");
#endif
```

接着，就可以把程序下载到学习板上，再通电测试，看看具体会有哪些中断发生。图3.3.2就是这个程序通过串口显示出来的信息，同时，还弹出了无法识别的USB设备的对话框，如图3.3.3所示。这是因为程序还没有返回描述符。从串口显示的内容可以看出，在连接USB之后，主机对设备进行了几次复位操作，然后就往端点发送数据了，因为我们看到了端点0输出已经产生了中断。那么主机到底发了些什么数据到端点0呢？且看下节分解。

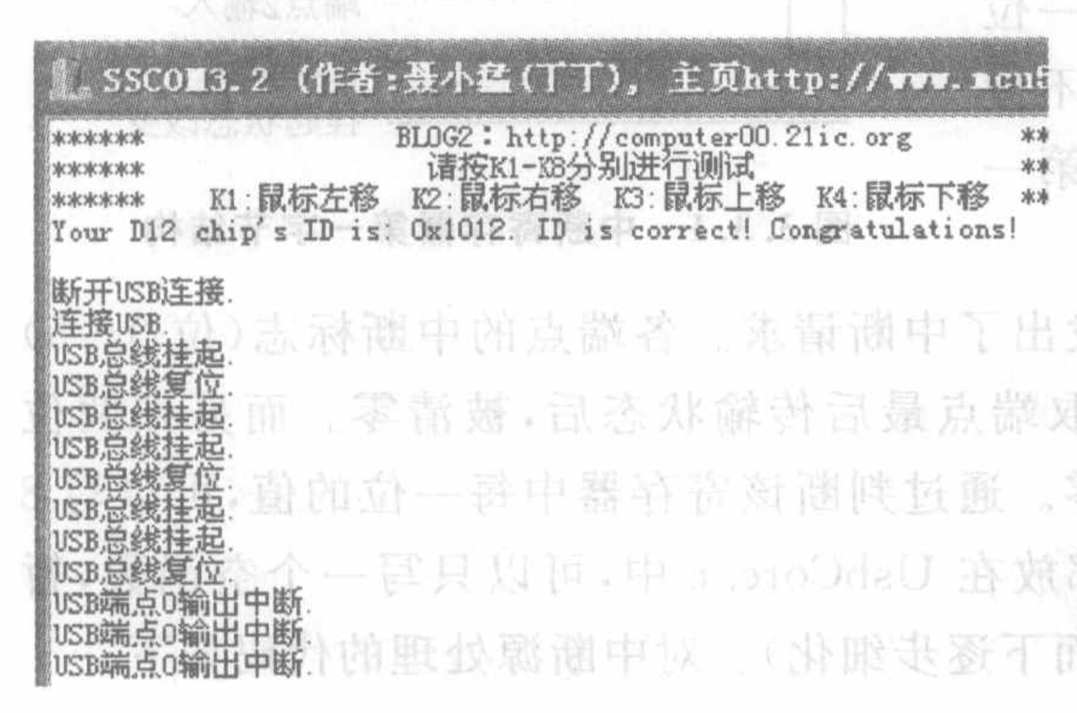


图3.3.2 调试信息输出

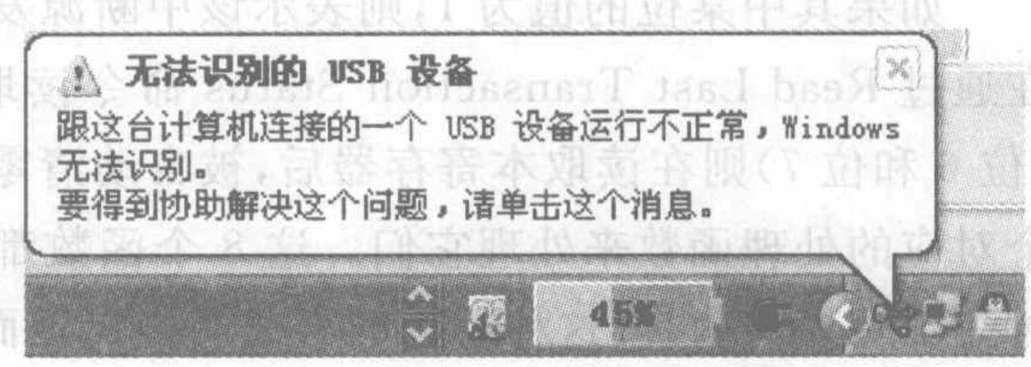


图3.3.3 发现无法识别的USB设备

3.4 读取从主机发送到端点0的数据

前面提到，主机发送了数据到端点0，但这些具体的数据却还没有读出来。因此接下来，就来写一个读端点数据的函数。这个函数有三个入口参数，分别是要读取数据的端点号

Endp、需要读取数据的长度 Len 和存放数据的缓冲区 Buf。该函数返回实际读到的数据长度(字节数)。

读取 D12 的数据缓冲区,使用 D12 的读缓冲(read buffer)命令,它的代码是 0xF0。发送该命令后,就可以连续读数据了。读出的第一字节是保留的,没有意义,不用理会它。第二字节的值是接收到数据的字节数,读取它之后我们就知道缓冲区内实际接收到了多少字节的数据。从第三字节开始是真正的 USB 数据,将它们读出保存到自己的 Buf 中。该函数有个入口参数 Len,它是想要读取的字节数。如果 Len 值比实际接收到的字节数要小,那么就只取前面的 Len 字节。如果实际接收到的数据不足 Len 长度,那么就只读实际接收的数据。函数的返回值是实际读取了多少字节的数据。

但是,D12 有很多个端点,如何决定读的是哪个端点的缓冲呢?这就需要用到 D12 的另外一个命令:选择端点(select endpoint)命令。选择端点命令共有 6 个,分别对应着 3 个端点的输出和输入,命令代码是 0x00~0x05,发送哪个命令就选择了哪个端点。为此,专门写一个选择端点的函数,该函数的入口参数就是要选择操作的端点号。选择端点的函数和最终的读取端点缓冲的函数代码如下:

```
/*****************************************************************
函数功能:选择端点的函数,选择一个端点后才能对它进行数据操作
入口参数:Endp   端点号
返      回:无
备      注:无
*****************************************************************/
void D12SelectEndpoint(uint8 Endp)
{
 D12WriteCommand(0x00 + Endp);                    //选择端点的命令
}
//////////////////////////End of function////////////////////////////

/*****************************************************************
函数功能:读取端点缓冲区函数
入口参数:Endp   端点号; Len   需要读取的数据长度; Buf   保存数据的缓冲区
返      回:实际读到的数据长度
备      注:无
*****************************************************************/
uint8 D12ReadEndpointBuffer(uint8 Endp, uint8 Len, uint8 * Buf)
{
 uint8 i,j;
 D12SelectEndpoint(Endp);                         //选择要操作的端点缓冲
 D12WriteCommand(D12_READ_BUFFER);                //发送读缓冲区的命令
```

```
    D12ReadByte();                          //该字节数据是保留的,不用
    j = D12ReadByte();                      //这里才是实际的接收到的数据长度
    if(j>Len)                               //如果要读的字节数比实际接收到的数据长
    {
     j = Len;                               //则只读指定的长度数据
    }
   #ifdef DEBUG1                            //如果定义了 DEBUG1,则需要显示调试信息
    Prints("读端点");
    PrintLongInt(Endp/2);                   //端点号。由于 D12 特殊的端点组织形式,
                                            //这里的 0 和 1 分别表示端点 0 的输出和输入;
                                            //而 2、3 分别表示端点 1 的输出和输入;
                                            //3、4 分别表示端点 2 的输出和输入
                                            //因此要除以 2 才显示对应的端点
    Prints("缓冲区");
    PrintLongInt(j);                        //实际读取的字节数
    Prints("字节。\r\n");
   #endif
    for(i = 0;i<j;i++)
    {
     //这里不直接调用读 1 字节的函数,而直接在这里模拟时序,可以节省时间
     D12ClrRd();                            //RD 置低
      *(Buf + i) = D12GetData();            //读 1 字节数据
     D12SetRd();                            //RD 置高
   #ifdef DEBUG1
     PrintHex( *(Buf + i));                 //如果需要显示调试信息,则显示读到的数据
     if(((i + 1) % 16) == 0)Prints("\r\n");  //每 16 字节换行一次
   #endif
    }
   #ifdef DEBUG1
    if((j % 16)! = 0)Prints("\r\n");        //换行
   #endif
    return j;                               //返回实际读取的字节数
   }
   ////////////////////////End of function////////////////////////////
```

在上面的代码中,输出调试信息使用了按十进制格式发送无符号长整数和按十六进制格式发送字节的函数,需要在 UART.c 中增加以下代码(记得在 UART.h 中做相关声明):

```
/*****************************************************************
函数功能:将整数按十进制字符串发送
```

```
入口参数:x 待显示的整数
返      回:无
备      注:无
*************************************************************************/
void PrintLongInt(uint32 x)
{
 int8 i;
 uint8 display_buffer[10];

 for(i=9;i>=0;i--)                                  //求出每位
  {
   display_buffer[i]='0'+x%10;
   x/=10;
  }
 for(i=0;i<9;i++)                                   //去掉前面多余的0
  {
   if(display_buffer[i]!='0')break;
  }
 for(;i<10;i++)UartPutChar(display_buffer[i]);      //通过串口发送
}
//////////////////////End of function//////////////////////////
/*************************************************************************
函数功能:发送1字节的数据
入口参数:x 待发送的数据
返      回:无
备      注:无
*************************************************************************/
void Printc(uint8 x)
{
 Sending=1;
 SBUF=x;
 while(Sending);
}
//////////////////////End of function//////////////////////////
/*************************************************************************
函数功能:以HEX格式发送1字节的数据
入口参数:x 待发送的数据
返      回:无
备      注:无
```

```
*****************************************************************/
void PrintHex(uint8 x)
{
 Printc('0');
 Printc('x');
 Printc(HexTable[x>>4]);
 Printc(HexTable[x&0xf]);
 Printc(' ');
}
//////////////////////End of function//////////////////////////////
```

然后在端点0输出中断处理函数中调用读取端点缓冲的函数，以读取主机发送过来的数据。但是只读取数据还不够，还有些额外的工作要做，例如清除中断标志（否则就会一直提示中断发生）、清除数据缓冲区（否则就不能再接收数据）等。

在读中断寄存器时提到过，清除端点中断标志要使用读最后传输状态命令（read last transaction status register），该命令代码为0x40～0x45，分别对应着3个端点的输出和输入。发送该命令后可以读1字节数据，数据的内容为该端点传输的最后状态，详细的结构如图3.4.1所示。

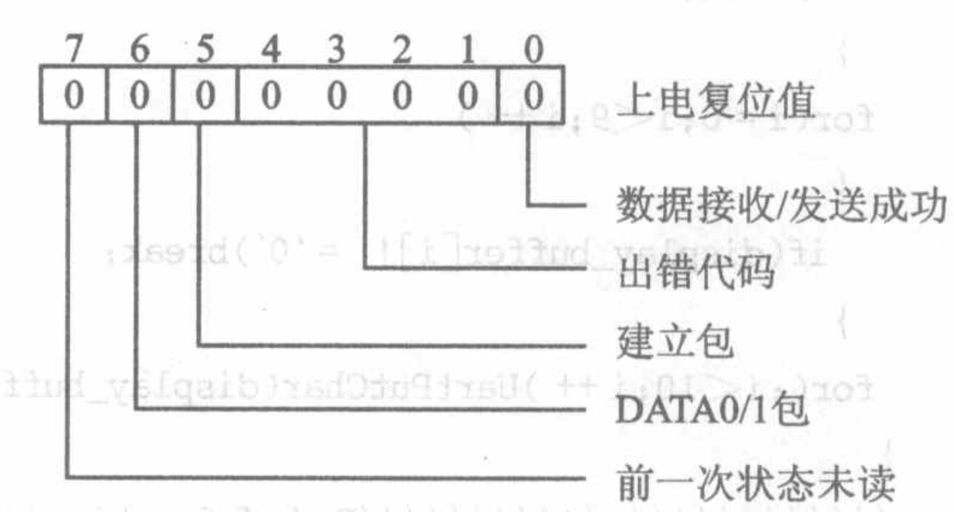


图3.4.1 最后传输状态寄存器

- 位0：该位为1表示数据成功接收或发送。
- 位1～4：出错代码。具体的出错代码请看数据手册。
- 位5：如果该位为1，则表示收到的是建立（setup）过程的数据包。
- 位6：该位为0表示收到的是DATA0数据包，该位为1表示收到的是DATA1数据包。
- 位7：该位为1表示前一次状态没有读取，前面的状态已经被覆盖。

其中第5位在控制传输中很有用，这让我们知道当前收到的是建立过程的数据包。建立包是控制传输第一个过程的令牌包，地位很特殊，控制端点必须要接收建立过程的数据包。

出错代码可用来调试，知道当前的芯片处于怎么样的状态。

清除数据缓冲区的命令是Clear Buffer，命令代码是0xF2。如果一个端点接收数据后没有使用Clear Buffer命令清除，对于以后发往该端点的数据包（建立过程的数据包除外，设备必须接收它）将使用NAK来应答。因此在读取数据之后，要记得将端点缓冲区清空。然而，对于D12的控制端点，接收到建立包后必须要使用一个特殊的命令，才能让Clear Buffer命令和Validate Buffer命令生效，这个命令就是Acknowledge Setup。这样做的目的是为了保证控制传输建立过程的数据不会丢失，并且接着也不会返回错误的数据，只有等到处理完了这个

建立过程，并发送 Acknowledge Setup 命令之后，才能使用 Clear Buffer 命令和 Validate Buffer 命令。因此程序首先要判断一下，收到的这个数据包是否为建立过程的数据包，如果是，则在发送 Clear Buffer 命令之前，还需要先发送 Acknowledge Setup 命令。Acknowledge Setup 命令对控制输出和输入端点都要发送，因为 Clear Buffer 命令是针对输出端点的，而 Validate Buffer 命令是针对输入端点的。

通常是先读取端点缓冲区后再来清除端点缓冲，因此这里的清除端点缓冲区函数并没有再选择端点，避免多余的操作。在调用该函数之前，一定要确保当前所选择的端点是需要清除的目标端点。例如：下面的 AcknowledgeSetup()函数就是先对输入端点 0 操作，然后再对输出端点 0 操作，以保证在后面使用清除缓冲函数时当前的目标端点是输出端点 0。最后的 ClearBuffer()函数和 AcknowledgeSetup()函数代码如下：

```
/*****************************************************************
函数功能:清除接收端点缓冲区的函数
入口参数:无
返      回:无
备      注:只有使用该函数清除端点缓冲后,该端点才能接收新的数据包(建立过程除外)
*****************************************************************/
void D12ClearBuffer(void)
{
 D12WriteCommand(D12_CLEAR_BUFFER);
}
//////////////////////End of function//////////////////////////

/*****************************************************************
函数功能:应答建立包的函数
入口参数:无
返      回:无
备      注:无
*****************************************************************/
void D12AcknowledgeSetup(void)
{
 D12SelectEndpoint(1); //选择端点 0 输入
 D12WriteCommand(D12_ACKNOWLEDGE_SETUP); //发送应答设置到端点 0 输入
 D12SelectEndpoint(0); //选择端点 0 输出
 D12WriteCommand(D12_ACKNOWLEDGE_SETUP); //发送应答设置到端点 0 输出
}
//////////////////////End of function//////////////////////////
```

进入端点 0 输出中断后，首先读取最后传输状态，然后检查第 5 位是否为 1，如果是 1 则

说明是建立包，这时读取数据后需要调用D12AcknowledgeSetup()函数；如果不是1，则说明只是普通的输出数据包，就不用调用D12AcknowledgeSetup()函数，直接清除缓冲区即可。最后，将端点0输出中断处理函数改成下面的样子：

```
/*****************************************************************
函数功能:端点0输出中断处理函数
入口参数:无
返    回:无
备    注:无
*****************************************************************/
void UsbEp0Out(void)
{
#ifdef DEBUG0
 Prints("USB端点0输出中断。\r\n");
#endif
 //读取端点0输出最后传输状态,该操作清除中断标志
 //并判断第5位是否为1,如果是,则说明是建立包
 if(D12ReadEndpointLastStatus(0)&0x20)
 {
  D12ReadEndpointBuffer(0,16,Buffer);
  D12AcknowledgeSetup();
  D12ClearBuffer();
 }
 else                                    //普通数据输出
 {
  D12ReadEndpointBuffer(0,16,Buffer);
  D12ClearBuffer();
 }
}
//////////////////////End of function/////////////////////////////
```

编译上面的程序，并烧入到学习板中运行。这时通过串口调试助手应该可以看到返回的调试信息，如图3.4.2所示。

从图3.4.2中可以看出，已经成功地接收到了主机发送过来的8字节数据。在第一次接收到数据后，会停顿一段时间。这段时间主机一直在请求输入。但是程序目前还没有返回数据，所以D12一直在回答NAK，即没有数据准备好。结果USB主机经过一段时间的等待之后，终于不耐烦了，发送了一次总线复位，然后又重新输出这个8字节的数据，然后又是等待输入数据……主机总共要重试三次这个操作，但是，结果三次都失败了，没有读取到数据，没办法，主机只好很无奈地放弃了操作。这时，该USB端口上不再有数据活动，从而D12进入了

```
SSCOM3.2 (作者:聂小垒(丁丁), 主页http://www.mcu51.
Your D12 chip's ID is: 0x1012. ID is correct! Congratulations!

断开USB连接.
连接USB.
USB总线挂起.
USB总线复位.
USB总线挂起.
USB总线挂起.
USB总线复位.
USB端点0输出中断.
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
USB总线复位.
USB端点0输出中断.
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
USB总线复位.
USB端点0输出中断.
读端点0缓冲区8字节.
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
USB总线挂起.
```

图 3.4.2 串口显示的调试信息

挂起状态。同时,在计算机端弹出无法识别的 USB 设备对话框。

那么这 8 个字节的数据到底是干什么用的?主机又到底在试图读取什么数据呢?其实这 8 个字节的数据就是 USB 协议所规定的标准请求,主机在试图获取描述符。

3.5 USB 标准请求

USB 协议定义了一个 8 字节的标准设备请求,主要用在设备的枚举过程中。这 8 字节的数据是在控制传输的建立过程通过默认控制端点 0 发出的。在这 8 字节的数据中,包含了数据过程所需要传输数据传输的方向、长度以及数据类型等信息。正是由于 8 字节标准请求的原因,USB 协议规定,端点 0 的最大包长度至少为 8 字节。也就是说,任何一个 USB 设备都能够(而且必须要)接收 8 字节的标准请求。

3.5.1 USB 标准设备请求的结构

正如前面在调试信息中所看到的那样,这 8 个字节只是一些数字,我们不知道它们表示什么意思。要知道这些数据所表示的意思,就需要查看 USB 协议中对这些数据的定义。表 3.5.1就是 USB 协议所规定的标准设备请求的结构。

表 3.5.1 USB标准设备请求的数据结构

偏移量/字节	域	大小/字节	取 值	描 述
0	bmRequestType	1	位图	请求的特性 D7:数据传输方向 0=主机到设备 1=设备到主机 D6～5:请求的类型 0=标准 1=类 2=厂商 3=保留 D4～0:请求的接收者 0=设备 1=接口 2=端点 3=其他 4～31=保留
1	bRequest	1	数值	请求代码
2	wValue	2	数值	该域的意义由具体的请求决定
4	wIndex	2	索引或偏移量	该域的意义由具体的请求决定
6	wLength	2	字节数	数据过程(如果有)所需要传输的字节数

本节只介绍USB协议定义的标准请求,即bmRequestType的D6～5位为00的请求。USB协议定义了11个标准请求(bRequest),其名字和请求代码如表3.5.2所列。

表 3.5.2 标准请求以及代码

bRequest	Value	bRequest	Value
GET_STATUS	0	GET_CONFIGURATION	8
CLEAR_FEATURE	1	SET_CONFIGURATION	9
SET_FEATURE	3	GET_INTERFACE	10
SET_ADDRESS	5	SET_INTERFACE	11
GET_DESCRIPTOR	6	SYNCH_FRAME	12
SET_DESCRIPTOR	7		

不同的请求对于其接收者、wValue和wIndex,其各字段的意义是不一样的。表3.5.3是各个标准请求的结构以及数据过程需要传输的数据。其中第一列有的有多个,主要是最低

5位不同，即表示接收者不同。有的请求只能发送到设备，而有的请求可以发送到设备、接口和端点。常用的几个请求为GET_DESCRIPTOR、SET_ADDRESS和SET_CONFIGURATION。下面详细介绍这几个请求。

表3.5.3 各种标准请求的结构及需要传输的数据

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
00000000B 00000001B 00000010B	CLEAR_FEATURE	特性选择	0 接口号 端点号	0	没有
10000000B	GET_CONFIGURATION	0	0	1	配置值
10000000B	GET_DESCRIPTOR	描述符类型和索引	0或者语言ID	描述符的长度	描述符
10000001B	GEG_INTERFACE	0	接口号	1	备用接口号
10000000B 10000001B 10000010B	GET_STATUS	0	0 接口号 端点号	2	设备、接口或者端点状态
00000000B	SET_ADDRESS	设备地址	0	0	没有
00000000B	SET_CONFIGURATION	配置值	0	0	没有
00000000B	SET_DESCRIPTOR	描述符类型和索引	0或者语言ID	描述符的长度	描述符
00000000B 00000001B 00000010B	SET_FEATURE	特性选择	0 接口号 端点号	0	没有
00000001B	SET_INTERFACE	备用接口号	接口号	0	没有
10000010B	SYNCH_FRAME	0	端点号	2	帧号

3.5.2 GET_DESCRIPTOR请求

GET_DESCRIPTOR(获取描述符)请求是在枚举过程中用得最多的一个请求。主机通过发送获取描述符请求读取设备的各种描述符，从而可以获知设备类型、端点情况等众多重要信息。获取描述符的接收者只能是设备，从bmRequestType的第7位可以看出，它是请求数据输入的。bRequest的值为0x06(GET_DESCRIPTOR)。GET_DESCRIPTOR请求的结构和需要传输的数据如表3.5.4所列。

表3.5.4 GET_DESCRIPTOR请求的结构

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
10000000B (0x80)	GET_DESCRIPTOR (0x06)	描述符类型和索引	0或者语言ID	描述符长度	描述符

表3.5.4中wValue域的第一字节(低字节)表示的是索引号,用来选择同一种描述符(例如字符串描述符和配置描述符)中具体的某个描述符。wValue域的第二字节,表示描述符的类型编号。各种描述符的类型编号定义如表3.5.5所列。wIndex域只在获取字符串描述符中有用,它表示字符串的语言ID号,获取除字符串描述符的其他描述符时,wIndex的值为0。wLength域为请求设备返回数据的字节数,设备实际返回的字节数可以比该域指定的字节数少。设备在收到获取描述符的请求后,应该按照所请求的描述符类型编号,在数据过程中返回相应的描述符。对于全速模式和低速模式,获取描述符的标准请求只有三种:获取设备描述符、获取配置描述符和获取字符串描述符。另外的接口描述符和端点描述符是跟随配置描述符一并返回的,不能单独请求返回(如果单独返回,主机无法确认它们属于哪个配置)。

需要注意的是wValue、wIndex、wLength这三个域都是两字节的,在USB协议中规定,使用的是小端结构,即低字节在先,高字节在后。

表3.5.5 描述符类型及编号

描述符类型	编 号
设备描述符(DEVICE)	1
配置描述符(CONFIGURATION)	2
字符串描述符(STRING)	3
接口描述符(INTERFACE)	4
端点描述符(ENDPOINT)	5

3.5.3 SET_ADDRESS请求

SET_ADDRESS(设置地址)请求是主机请求设备使用指定地址的请求,指定的地址就包含在8字节数据中的wValue字段中。每个连接在同一个主控制器上的USB设备都需要具有一个唯一的设备地址,这样主机才能区分每个不同的设备。当设备复位后,都使用默认的地址0。主机从地址为0的设备获取设备描述符,一旦收到第一次设备描述符之后,主机就会发送设置地址的请求,以尽量减少设备使用公共地址0的时间。设置地址请求是没有数据过程的,因而wLength的值为0。wIndex也用不着,值为0。当设备收到设置地址请求后,就直接进入状态过程,等待主机读取0长度的状态数据包。主机成功读取到状态数据包(用ACK响应设备)后,设备将启用新的地址。这以后的传输中,主机都将使用新的地址与设备进行通信。表3.5.6所列是SET_ADDRESS请求的结构。

表3.5.6 SET_ADDRESS请求的结构

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
00000000B (0x00)	SET_ADDRESS (0x05)	设备地址	0x0000	0x0000	没有

3.5.4 SET_CONFIGURATION请求

SET_CONFIGURATION(设置配置)请求和设置地址请求很类似。区别在于wValue域

的意义。在设置地址请求中，wValue的第一字节(低字节)为设备的地址；而在设置配置请求中，wValue的第一字节为配置的值。当该值与某配置描述符中的配置编号一致时，表示选中该配置。该值通常为1，因为大多数USB设备只有一种配置，配置编号为1；如果该值为0，则会让设备进入设置地址状态。设备只有在收到非0的配置值后，才能启用它的非0端点。SET_CONFIGURATION请求的结构如表3.5.7所列。

表3.5.7 SET_CONFIGURATION请求的结构

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
00000000B (0x00)	SET_CONFIGURATION (0x09)	配置值	0x0000	0x0000	没有

在前面的调试信息中，接收到的数据为0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00，对照标准请求的结构可知，它是一个请求设备描述符的标准请求，请求的数据长度为64字节。

3.6 设备描述符的实现

既然知道主机请求设备返回设备描述符，那么我们就应该在数据过程中返回设备的设备描述符。但是现在还不知道设备描述符是些什么东西，所以也就没办法返回数据。因此接下来，就需要实现一个设备描述符。

已知每个设备都必须有且仅有一个设备描述符，它的结构在USB协议中有详细的定义，如表3.6.1所列。

表3.6.1 设备描述符的结构

偏移量/字节	域	大小/字节	说 明
0	bLength	1	该描述符的长度(18字节)
1	bDescriptorType	1	描述符类型(设备描述符为0x01)
2	bcdUSB	2	本设备所使用的USB协议版本
4	bDeviceClass	1	类代码
5	bDeviceSubClass	1	子类代码
6	bDeviceProtocol	1	设备所使用的协议
7	bMaxPackeSize0	1	端点0最大包长
8	idVender	2	厂商ID
10	idProduct	2	产品ID
12	bcdDevice	2	设备版本号

续表 3.6.1

偏移量/字节	域	大小/字节	说 明
14	iManufacturer	1	描述厂商的字符串的索引
15	iProduct	1	描述产品的字符串的索引
16	iSerialNumber	1	产品序列号字符串的索引
17	bNumConfigurations	1	可能的配置数

bLength 长度为1字节,表示该描述符的长度。设备描述符的长度为18字节,写成十六进制就是0x12。

bDescriptorType 描述符的类型,长度为1字节。具体的取值如表3.5.5所列。设备描述符的编号为0x01。

bcdUSB 该设备所使用的USB协议的版本,长度为2字节。可以取2.0或者1.1等版本号。注意,它是用BCD码来表示的,例如USB2.0协议就是0x0200,而USB1.1协议就是0x0110。前面说过,USB协议中使用的是小端结构,所以实际数据在传输时,是低字节在先的,也就是说,USB2.0协议的bcdUSB拆成两个字节就是0x00和0x20。而USB1.1的bcdUSB拆成两个字节就是0x10和0x01。

bDeviceClass 长度为1字节,是设备所使用的类代码。设备的类代码由USB协会规定,具体的类代码可查阅USB相关文档。对于大多数标准的USB设备类,该字段通常设置为0,而在接口描述符中的bInterfaceClass中指定接口所实现的功能。当bDeviceClass为0时,下面的bDeviceSubClass也必须为0。如果bDeviceClass为0xFF,表示是厂商自定义的设备类。

bDeviceSubClass 长度为1字节,是设备所使用的子类代码。当类代码不为0和0xFF时,子类代码由USB协议规定。当bDeviceClass为0时,bDeviceSubClass也必须为0。

bDeviceProtocol 长度为1字节,是设备所使用的协议,协议代码由USB协会规定。当该字段为0时,表示设备不使用类所定义的协议。当该字段为0xFF时,表示设备使用厂商自定义的协议。bDeviceProtocol必须要结合设备类和设备子类联合使用才有意义,因此当类代码为0时,bDeviceProtocol应该也要为0。

bMaxPackeSize0 长度为1字节,是端点0的最大包长。它的取值可以为8、16、32、64。

idVender 长度为2字节,它是厂商的ID号。该ID号由USB协会分配,不能随意使用。可以跟USB协会申请一个厂商ID号,但是这是需要交"保护费"的。在实验中,选取了一个0x8888作为厂商ID号。这个ID号是随便选的,也许它已经被某个公司购买了。只是拿来学习使用是没关系的,如果要做真正的产品设计,就要注意了,必须要使用公司自己的ID号,以避免侵权。另外,主机通常是靠厂商ID号、产品ID号以及产品序列号来安装和加载驱动的,如果使用了别人的ID号,可能会导致驱动程序安装或加载错误,从而导致设备无法工作。跟bcdUSB一样,要注意小端结构的问题。

idProduct 长度为2字节，是产品ID号。与厂商ID号不一样，它是由生产厂商自己根据产品来编号的，比较自由。例如，本书中的USB鼠标，可以给它分配产品ID为0x0001；USB键盘，可以分配产品ID为0x0002，而厂商ID则使用同一个0x8888。通常主机会根据厂商ID、产品ID以及设备的序列号来加载对应的驱动。有些朋友在修改描述符之后，重新做实验，却没有修改上述三个中的任何一个，结果加载的还是旧驱动，从而工作不正常。如果没有想到这一点，一个劲地在那查找固件的问题，恐怕是找不出问题的。如果不想修改这些ID号和序列号，还有另外一个方法就是去设备管理器里，将原先的驱动卸载，然后扫描新硬件或者拔下重新插上，让它重新安装驱动。不过有时卸载了还是会有残留信息在注册表中，所以还是改ID比较保险。

bcdDevice 长度为2字节，是设备的版本号。当同一个产品升级后（例如修改了固件增加了某些功能），可以通过修改设备的版本号来区别。

iManufacturer 长度为1字节，是描述厂商的字符串的索引值。当该值为0时，表示没有厂商字符串。主机获取设备描述符时，会将索引值放在wValue的第一字节中，用来选择不同的字符串。

iProduct 长度为1字节，是描述产品的字符串的索引值。当该值为0时，表示没有产品字符串。当第一次插上某个USB设备时，会在Windows的右下角弹出一个对话框，显示发现新硬件，并且会显示该设备的名称。其实这里显示的信息就是从产品字符串里获取来的。如果想让它显示出所需要的信息，应该修改产品字符串。

iSerialNumber 长度为1字节，是设备的序列号字符串索引值。最好给你的每个产品指定一个唯一的序列号，好比每个英特尔的奔四处理器都有一个ID号一样。设备序列号可能被主机联合VID和PID用来区别不同的设备，有时同时连接多个具有相同VID、PID以及设备序列号的设备，可能会导致设备无法正确识别。当该值为0时，表示没有序列号字符串。

bNumConfigurations 长度为1字节，表示设备有多少种配置。每种配置都会有一个配置描述符，主机通过发送设置配置来选择某一种配置。大部分的USB设备只有一个配置，即该字段的值为1。

通过上面的描述，你应该知道如何去构造一个设备描述符了吧。下面给出USB鼠标实例的设备描述符。这里使用数组来实现，使用数组的好处是可移植性好，但是可读性稍差点。如果使用结构体，可读性会好一些；但是在不同位数的处理器中移植时，可能存在着字节对齐、大小端问题和结构体填充等问题。

```
//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
//我们不在设备描述符中定义设备类,而在接口描述符中定义设备类,所以该字段的值为 0
 0x00,
//bDeviceSubClass 字段。bDeviceClass 字段为 0 时,该字段也为 0
 0x00,
//bDeviceProtocol 字段。bDeviceClass 字段为 0 时,该字段也为 0
 0x00,
//bMaxPacketSize0 字段。PDIUSBD12 的端点 0 大小为 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,必须向 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,由于是第一个实验,这里取 0x0001 注意小端模式,低字节应该在先
 0x01,
 0x00,
//bcdDevice 字段。因为这个 USB 鼠标刚开始做,就叫它 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
//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,
};
```

3.7 设备描述符的返回

现在设备描述符有了，那怎么将它返回给主机呢？这就要通过控制输入端点0来返回了。在端点0的输出中断处理函数中，先对接收到的建立过程的数据进行判断，如果是获取设备描述符的请求，那么将设备描述符数组的内容写入到端点0输入缓冲区中，并使能端点发送。当主机在下一次发送IN令牌后，D12将会自动将端点0输入缓冲区中的数据返回给主机，这样就实现了获取设备描述符的请求。

要将数据写入到D12的端点0输入缓冲区中，首先要准备一个写数据到端点输入缓冲区的函数。该函数与读取端点缓冲区的函数很类似，也有三个输入参数和一个返回值。输入参数分别是要发送数据的端点号Endp、需要发送数据的长度Len和存放待发送数据的缓冲区Buf。函数返回Len的值。

写数据之前首先要选择端点，这个函数前面已经实现过了，这里直接调用它。写数据到端点缓冲区的命令是Write Buffer，命令代码是0xF0。与读端点缓冲的命令类似，写入的数据第一个字节必须为0，第二字节为需要发送数据的长度。后面写入的才是真正需要发送的数据。

当数据写入到端点缓冲区之后，还需要使用一个命令将端点的发送缓冲区中的数据设置为有效后，数据才会在主机发送IN令牌包后发送出去。这个命令就是Validate Buffer，命令代码为0xFA。实现该命令的函数以及写输入端点缓冲区的函数代码如下：

```
/*****************************************************************
函数功能:使能发送端点缓冲区数据有效的函数
入口参数:无
返      回:无
备      注:只有使用该函数使能发送端点数据有效之后,数据才能发送出去
*****************************************************************/
void D12ValidateBuffer(void)
{
 D12WriteCommand(D12_VALIDATE_BUFFER);
}
//////////////////////End of function//////////////////////////

/*****************************************************************
函数功能:将数据写入端点缓冲区函数
入口参数:Endp  端点号; Len  需要发送的长度; Buf  保存数据的缓冲区
返      回:Len的值
备      注:无
*****************************************************************/
uint8 D12WriteEndpointBuffer(uint8 Endp,uint8 Len,uint8 * Buf)
```

```
{
 uint8 i;
 D12SelectEndpoint(Endp);                    //选择端点
 D12WriteCommand(D12_WRITE_BUFFER);          //写 Write Buffer 命令
 D12WriteByte(0);                            //该字节必须写 0
 D12WriteByte(Len);                          //写需要发送数据的长度

 #ifdef DEBUG1                               //如果定义了 DEBUG1,则需要显示调试信息
 Prints("写端点");
 PrintLongInt(Endp/2);                       //端点号。由于 D12 特殊的端点组织形式,这里的 0 和 1
                                             //分别表示端点 0 的输出和输入;而 2、3 分别表示端点 1
                                             //的输出和输入,3、4 分别表示端点 2 的输出和输入,因
                                             //此要除以 2 才显示对应的端点
 Prints("缓冲区");
 PrintLongInt(Len);                          //写入的字节数
 Prints("字节。\r\n");
 #endif
 SetUsbPortOut();                            //将数据口设置为输出状态(注意这里为空宏,移植时可
                                             //能有用)
 for(i = 0;i<Len;i ++ )
 {
  //这里不直接调用写一字节的函数,而直接在这里模拟时序,可以节省时间
  D12ClrWr();                                //WR 置低
  D12WriteByte( * (Buf + i));                //将数据放到数据线上
  D12SetWr();                                //WR 置高,完成 1 字节写
 #ifdef DEBUG1
  PrintHex( * (Buf + i));                    //如果需要显示调试信息,则显示发送的数据
  if(((i + 1) % 16) == 0)Prints("\r\n");     //每 16 字节换行一次
 #endif
  }
 #ifdef DEBUG1
 if((Len % 16)! = 0)Prints("\r\n");          //换行
 #endif
 D12SetPortIn();                             //数据口切换到输入状态
 D12ValidateBuffer();                        //使端点数据有效
 return Len;                                 //返回 Len
}
//////////////////////End of function////////////////////////////
```

写数据的函数已经准备好了,接下来就要分析建立过程数据包的内容,以确定何时返回设

备描述符。分析数据包的内容主要使用if语句和switch语句来散转，对不同的请求做不同的处理。

在分析数据之前，先申请一些变量，用来保存这些标准请求的不同字段。另外，我们还需要一个用来保存发送位置的指针变量和一个保存剩余字节数的变量，以及一个是否需要返回0长度数据包的标志。变量定义代码如下：

```
//USB设备请求的各字段
uint8  bmRequestType;
uint8  bRequest;
uint16 wValue;
uint16 wIndex;
uint16 wLength;
//当前发送数据的位置
uint8 * pSendData;
//需要发送数据的长度
uint16 SendLength;
//是否需要发送0数据包的标志。在USB控制传输的数据过程中，当返回的数据包字节数少于最大
//包长时，会认为数据过程结束；当请求的字节数比实际需要返回的字节数长，而实际返回的字节数
//又刚好是端点0大小的整数倍时，就需要返回一个0长度的数据包来结束数据过程
//因此这里增加一个标志，供程序决定是否需要返回一个0长度的数据包
uint8 NeedZeroPacket;
```

接着将从缓冲区中读回的数据分别添加到设备请求的各字段中，代码如下：

```
//将缓冲数据添加到设备请求的各字段中
bmRequestType = Buffer[0];
bRequest = Buffer[1];
wValue = Buffer[2] + (((uint16)Buffer[3])<<8);
wIndex = Buffer[4] + (((uint16)Buffer[5])<<8);
wLength = Buffer[6] + (((uint16)Buffer[7])<<8);
```

首先是对bmRequestType的D7位判断，如果它是1，则说明是输入请求；如果是0，则说明是输出请求。再对D6～5位判断，如果是00，则说明是标准请求；如果是01，则说明是类请求；如果是10，则说明是厂商请求。再根据表3.5.3，在标准请求中写上全部的标准请求情况。如果是获取描述符的请求，则还需要对bRequest进行散转，看具体请求的是什么描述符；如果是获取字符串描述符，则还需要对wValue值的低字节散转，看具体是哪个字符串描述符；这个过程内容比较多，用文字描述起来不方便，还是来看看具体的实现代码。

下面的代码判断具体的请求，并根据不同的请求进行相关操作。如果D7位为1，则说明是输入请求。事实上，我们还需要对接收者进行散转，因为不同的请求接收者是不一样的。接

收者在 bmRequestType 的 D4～D0 位中定义。这里为了简化操作，有些就省略了对接收者的判断。例如获取描述符的请求，只根据描述符的类型来区别。

```
if((bmRequestType&0x80) == 0x80)
{
 //根据 bmRequestType 的 D6～5 位散转，D6～5 位表示请求的类型
 //0 为标准请求，1 为类请求，2 为厂商请求
 switch((bmRequestType>>5)&0x03)
 {
  case 0:                                    //标准请求
   #ifdef DEBUG0
    Prints("USB 标准输入请求:");
   #endif
   //USB 协议定义了几个标准输入请求，实现这些标准请求即可
   //请求的代码在 bRequest 中，对不同的请求代码进行散转
   switch(bRequest)
   {
    case GET_CONFIGURATION:                  //获取配置
     #ifdef DEBUG0
      Prints("获取配置。\r\n");
     #endif
    break;
    case GET_DESCRIPTOR:                     //获取描述符
     #ifdef DEBUG0
      Prints("获取描述符。\r\n");
     #endif
    break;
    case GET_INTERFACE:                      //获取接口
     #ifdef DEBUG0
      Prints("获取接口。\r\n");
     #endif
    break;
    case GET_STATUS:                         //获取状态
     #ifdef DEBUG0
      Prints("获取状态。\r\n");
     #endif
    break;
    case SYNCH_FRAME:                        //同步帧
```

```
    #ifdef DEBUG0
     Prints("同步帧。\r\n");
    #endif
   break;
   default:                                    //未定义的标准请求
    #ifdef DEBUG0
     Prints("错误:未定义的标准输入请求。\r\n");
    #endif
   break;
  }
 break;
 case 1:                                       //类请求
  #ifdef DEBUG0
   Prints("USB类输入请求:\r\n");
  #endif
 break;
 case 2:                                       //厂商请求
  #ifdef DEBUG0
   Prints("USB厂商输入请求:\r\n");
  #endif
 break;
 default:                                      //未定义的请求。这里只显示一个报错信息
  #ifdef DEBUG0
   Prints("错误:未定义的输入请求。\r\n");
  #endif
 break;
 }
}
//否则说明是输出请求
else                                           //if(bmRequestType&0x80 == 0x80)之 else
{
 //根据 bmRequestType 的 D6～5 位散转,D6～5 位表示请求的类型
 //0 为标准请求,1 为类请求,2 为厂商请求
 switch((bmRequestType>>5)&0x03)
 {
  case 0:                                      //标准请求
   #ifdef DEBUG0
    Prints("USB标准输出请求:");
```

```
#endif
//USB协议定义了几个标准输出请求,实现这些标准请求即可
//请求的代码在bRequest中,对不同的请求代码进行散转
switch(bRequest)
{
 case CLEAR_FEATURE:                    //清除特性
  #ifdef DEBUG0
   Prints("清除特性。\r\n");
  #endif
 break;

 case SET_ADDRESS:                      //设置地址
 #ifdef DEBUG0
   Prints("设置地址。\r\n");
  #endif
 break;

 case SET_CONFIGURATION:                //设置配置
 #ifdef DEBUG0
   Prints("设置配置。\r\n");
  #endif
 break;

 case SET_DESCRIPTOR:                   //设置描述符
 #ifdef DEBUG0
   Prints("设置描述符。\r\n");
  #endif
 break;

 case SET_FEATURE:                      //设置特性
 #ifdef DEBUG0
   Prints("设置特性。\r\n");
  #endif
 break;

 case SET_INTERFACE:                    //设置接口
 #ifdef DEBUG0
   Prints("设置接口。\r\n");
  #endif
 break;

 default:                               //未定义的标准请求
  #ifdef DEBUG0
```

```
    Prints("错误:未定义的标准输出请求。\r\n");
    #endif
   break;
  }
 break;
 case 1:                                    //类请求
  #ifdef DEBUG0
   Prints("USB类输出请求:\r\n");
  #endif
 break;
 case 2:                                    //厂商请求
  #ifdef DEBUG0
   Prints("USB厂商输出请求:\r\n");
  #endif
 break;
 default:                                   //未定义的请求。这里只显示一个报错信息
  #ifdef DEBUG0
   Prints("错误:未定义的输出请求。\r\n");
  #endif
 break;
}
}
```

以上代码是实现各种请求的散转，接下来，在获取描述符的处理中，增加对具体描述符散转的处理。描述符的类型保存在 wValue 中的高字节中。下面是描述处理散转处理的代码，由于暂时只实现了设备描述符，所以这里仅有对设备描述符的处理。

```
case GET_DESCRIPTOR:                        //获取描述符
 #ifdef DEBUG0
  Prints("获取描述符——");
 #endif
 //对描述符类型进行散转,对于全速设备,标准请求只支持设备、配置、字符串三种描述符
 switch((wValue>>8)&0xFF)
 {
  case DEVICE_DESCRIPTOR:                   //设备描述符
   #ifdef DEBUG0
      Prints("设备描述符。\r\n");
   #endif
   pSendData=DeviceDescriptor;              //需要发送的数据
```

```
//判断请求的字节数是否比实际需要发送的字节数多
//这里请求的是设备描述符,因此数据长度就是 DeviceDescriptor[0]
//如果请求的比实际长,那么只返回实际长度的数据
if(wLength>DeviceDescriptor[0])
{
 SendLength = DeviceDescriptor[0];
 if(SendLength % DeviceDescriptor[7] == 0)          //并且刚好是整数个数据包时
 {
  NeedZeroPacket = 1;                               //需要返回 0 长度的数据包
 }
}
else
{
 SendLength = wLength;
}
//将数据通过 EP0 返回
UsbEp0SendData();
break;

case CONFIGURATION_DESCRIPTOR:                      //配置描述符
 #ifdef DEBUG0
  Prints("配置描述符。\r\n");
 #endif
break;

case STRING_DESCRIPTOR:                             //字符串描述符
 #ifdef DEBUG0
  Prints("字符串描述符。\r\n");
 #endif
break;

default:                                            //其他描述符
 #ifdef DEBUG0
  Prints("其他描述符,描述符代码:");
  PrintHex((wValue>>8)&0xFF);
  Prints("\r\n");
 #endif
break;
}
break;
```


在上面的代码中,用到了一个 UsbEp0SendData()函数。该函数的功能是根据需要发送

数据的长度和位置，将数据写入到端点0去。以后需要往端点0写数据时，先设置好需要发送好数据的位置和长度，再调用这个函数即可。代码如下：

```
/*****************************************************************
函数功能:根据 pData 和 SendLength 将数据发送到端点 0 的函数
入口参数:无
返    回:无
备    注:无
*****************************************************************/
void UsbEp0SendData(void)
{
 //将数据写到端点中去,准备发送写之前要先判断一下需要发送的数据是否比端点 0 最大长度大,
 //如果超过端点大小,则一次只能发送最大包长的数据
 //端点 0 的最大包长在 DeviceDescriptor[7]
 if(SendLength>DeviceDescriptor[7])
 {
  //按最大包长度发送
  D12WriteEndpointBuffer(1,DeviceDescriptor[7],pSendData);
  //发送后剩余字节数减少最大包长
  SendLength- =DeviceDescriptor[7];
  //发送一次后指针位置要调整
  pSendData += DeviceDescriptor[7];
 }
 else
 {
  if(SendLength! = 0)
  {
   //不够最大包长,可以直接发送
   D12WriteEndpointBuffer(1,SendLength,pSendData);
   //发送完毕后,SendLength 长度变为 0
   SendLength = 0;
  }
  else                                        //如果要发送的数据包长度为 0
  {
   if(NeedZeroPacket == 1)                    //如果需要发送 0 长度数据
   {
    D12WriteEndpointBuffer(1,0,pSendData);    //发送 0 长度数据包
    NeedZeroPacket = 0;                       //清需要发送 0 长度数据包标志
   }
  }
```

```
    }
}
//////////////////////////End of function////////////////////////////
```

注意上面的代码中，如果需要发送的字节数比端点0还多时，一次只能发送最大包长的数据。那么剩余的数据怎么办呢？这就要牵涉到端点0输入中断的处理了。当前面的数据成功发送后，端点0输入中断就会产生。因此只要在端点0输入中断中，发送剩下的数据即可。记得还要清除端点0的输入中断，这个可以通过调用D12ReadEndpointLastStatus(1)函数实现。前面提到过，主机第一次获取设备描述符时，只会读一个数据包的设备描述符，但是这里的设备描述符长度比端点0的大小要大，因而也就会发送两次数据包。这会不会有问题呢？答案是不会有问题。因为D12在收到下一个SETUP包之后，会自动将端点0输入缓冲的数据置为无效，也就说是第二次加进去的数据实际上并没有起作用。因此在这里我们可以不对端点0的输入做特殊处理。最后的端点0输入中断处理函数修改如下：


```
/*****************************************************************
函数功能:端点0输入中断处理函数
入口参数:无
返      回:无
备      注:无
*****************************************************************/
void UsbEp0In(void)
{
#ifdef DEBUG0
 Prints("USB端点0输入中断。\r\n");
#endif
 //读最后发送状态,这将清除端点0的中断标志位
 D12ReadEndpointLastStatus(1);
 //发送剩余的字节数
 UsbEp0SendData();
}
//////////////////////////End of function////////////////////////////
```

到此，返回设备描述符的代码已经实现，将程序下载到学习板中运行，应该可以看到对端点0的写入操作。如果主机此时正确收到了设备描述符，应该会接着对总线复位一次，然后发送设置地址的标准请求。那么实际运行的结果如何呢？请看图3.7.1的串口返回的调试信息。从返回的调试信息可以看出，的确已经成功返回了设备描述符，并且也发出了设置地址的标准请求。设置的地址值在wValue的低字节，即图3.7.1中倒数第二行中的0x04。接下来，就要对设置地址这个标准请求进行相关处理了。

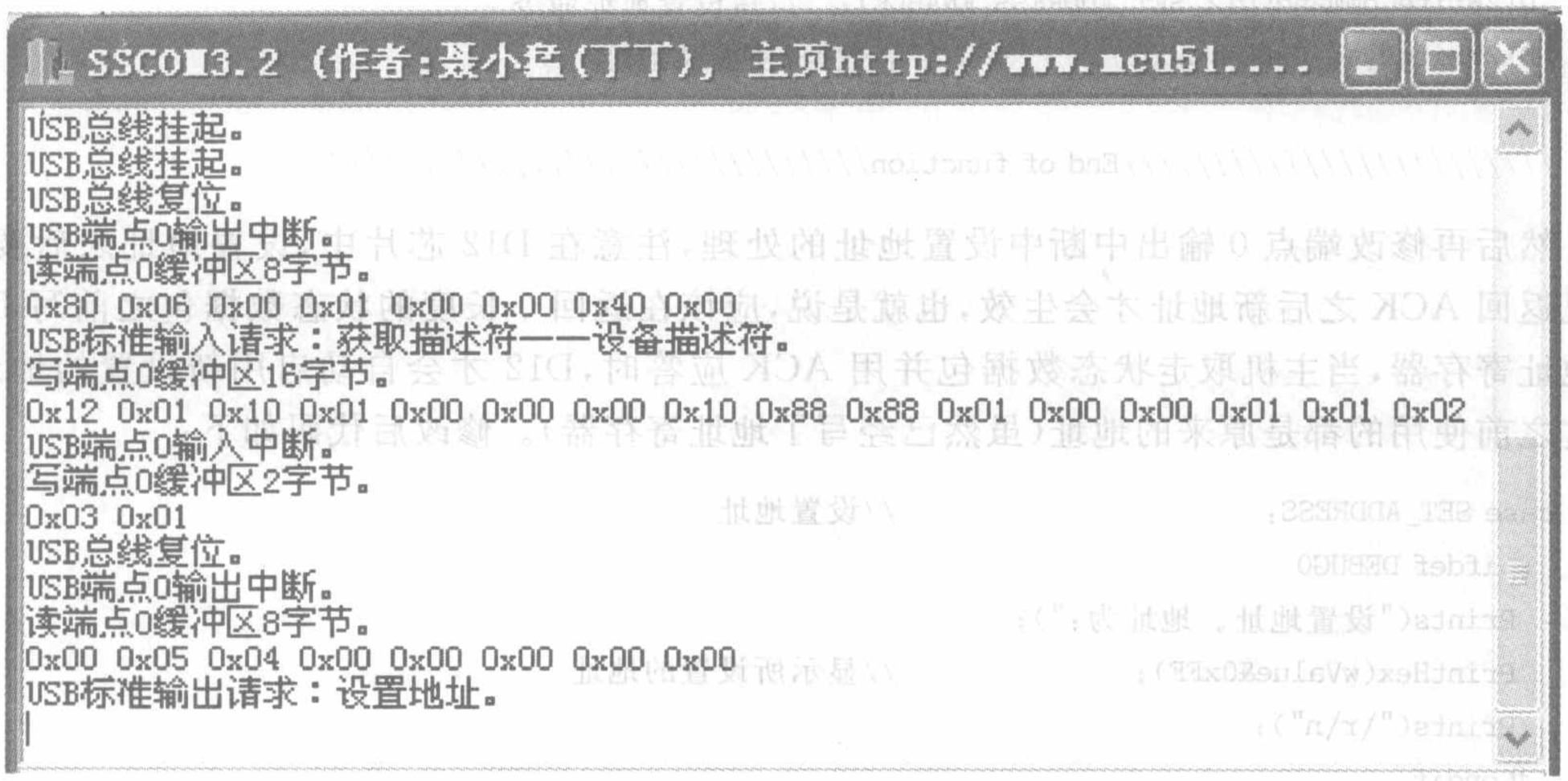


图 3.7.1 返回设备描述符时的调试信息

3.8 设置地址请求的处理

每个 USB 设备都具有一个唯一的设备地址，这个地址是主机在设置地址请求时分配给设备的。设备在收到设置地址请求后，应该返回一个 0 长度的状态数据包(因为设置地址请求是没有数据过程的)，然后等待主机确认这个数据包(即用 ACK 应答设备)。设备在正确接收到状态数据包的 ACK 之后，就开始使用新的设备地址了。

D12 芯片提供了一条设置地址的命令：Set Address/Enable 命令，命令代码为 0xD0。设置地址命令后跟一字节数据写入操作，该字节的 D7 位用来控制设备是否使能，只有当 D7 位设置为 1 时，D12 的普通端点才能通过使能端点(Set Endpoint Enable)命令启用。D6～D0 位是设备的 7 位地址，应该将接收到的标准请求中的地址写到 D6～D0 中。为此，写一个对 D12 写地址的函数，代码如下：

```
/*********************************************************************
函数功能:设置地址函数
入口参数:Addr  要设置的地址值
返      回:无
备      注:无
*********************************************************************/
void D12SetAddress(uint8 Addr)
{
```

```
 D12WriteCommand(D12_SET_ADDRESS_ENABLE);  //写设置地址命令
 D12WriteByte(0x80 | Addr);  //写1字节数据:使能及地址
}
//////////////////////End of function////////////////////////////
```

然后再修改端点0输出中断中设置地址的处理，注意在D12芯片中，设置地址之后要等主机返回ACK之后新地址才会生效，也就是说，应该在返回0长度的状态数据包之前写D12的地址寄存器，当主机取走状态数据包并用ACK应答时，D12才会自动启用新设置的地址。在这之前使用的都是原来的地址(虽然已经写了地址寄存器)。修改后代码如下：

```
case SET_ADDRESS:                          //设置地址
 #ifdef DEBUG0
  Prints("设置地址。地址为:");
  PrintHex(wValue&0xFF);                   //显示所设置的地址
  Prints("\r\n");
 #endif
 D12SetAddress(wValue&0xFF);               //wValue中的低字节是设置的地址值
 //设置地址没有数据过程,直接进入到状态过程,返回一个0长度的数据包
 SendLength = 0;
 NeedZeroPacket = 1;
 //将数据通过EP0返回
 UsbEp0SendData();
 break;
```

代码修改好后，编译下载运行后调试信息显示如图3.8.1所示。这时显示分配的地址跟前面的0x04不一样了，变成了0x14。这没什么关系，地址由主机管理，只要设备能分配到一个唯一的设备地址即可。在地址设置好之后，调试信息显示又接收到了主机的获取设备描述符的请求，这说明设置地址操作已经成功了。因为这时主机已经是使用的新地址来发送请求了，如果不成功，主机就会检测到超时，从而复位总线。这次获取设备描述符将会完整地获取。获取完设备描述符之后，主机接着又发了一个请求：获取配置描述符的请求，请求长度为9字节。标准配置描述符的长度就是9字节的，通常主机第一次先获取9字节长度的配置描述符，然后根据配置描述符中的配置集合长度，再次获取配置描述符。第二次获取配置描述符时，会将配置描述符、接口描述符、类特殊描述符(如果有)、端点描述符等一并读回。因此接下来就是实现配置描述符及接口描述符、类特殊描述符、端点描述符这个大集合——配置描述符集合。

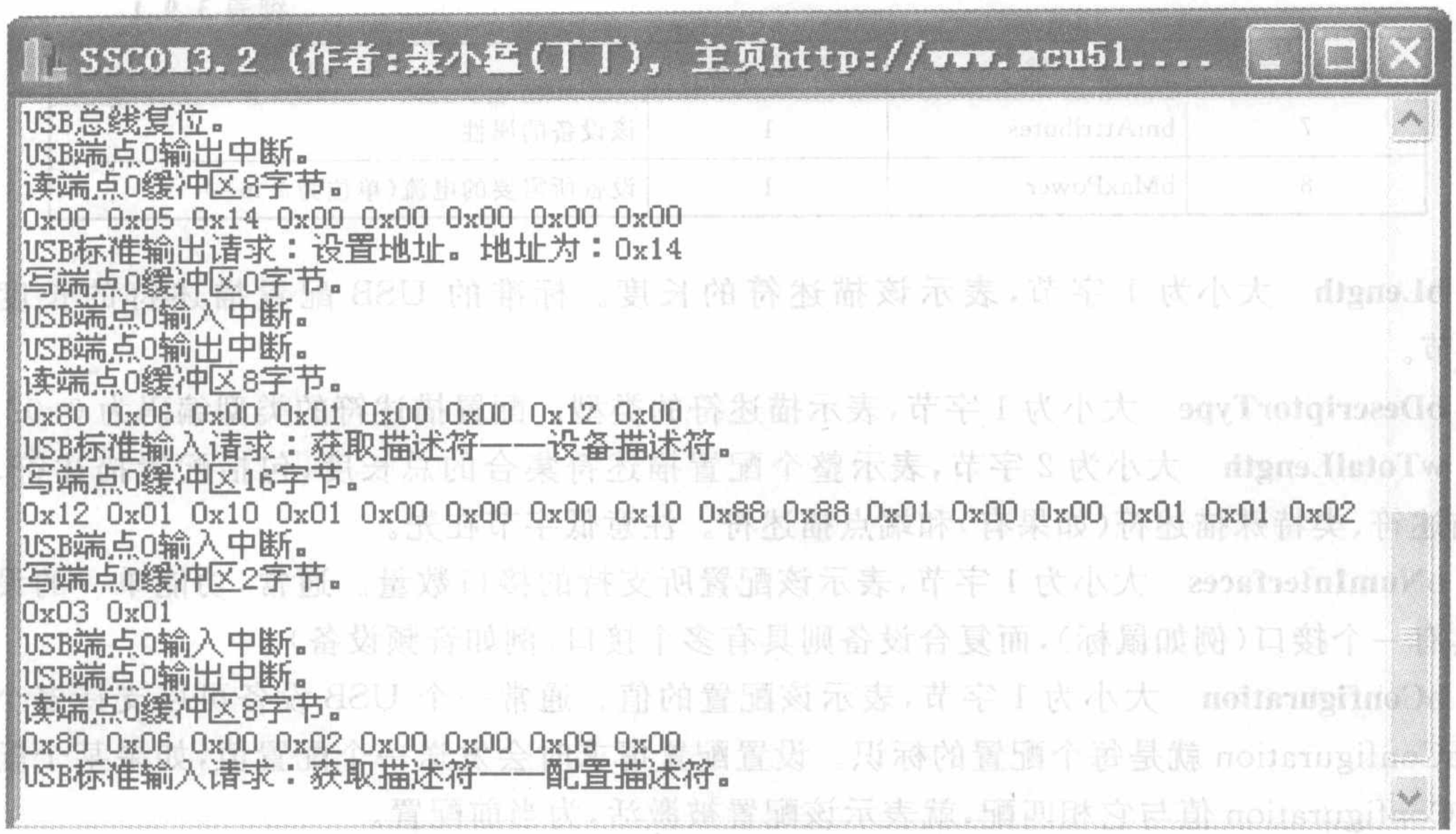


图 3.8.1 实现设置地址后的调试信息显示

3.9 配置描述符集合的结构

每个 USB 设备至少都要有一个配置描述符，在设备描述符中规定了该设备有多少种配置，每种配置都有一个描述符。在本 USB 实例中，只有一个配置描述符。

3.9.1 配置描述符的结构

表 3.9.1 是 USB 协议规定的标准配置描述符的结构。

表 3.9.1 标准配置描述符的结构

偏移量/字节	域	大小/字节	说 明
0	bLength	1	该描述符的长度(9 字节)
1	bDescriptorType	1	描述符类型(配置描述符为 0x02)
2	wTotalLength	2	配置描述符集合总长度
4	bNumInterfaces	1	该配置所支持的接口数
5	bConfigurationValue	1	该配置的值
6	iConfiguration	1	描述该配置的字符串的索引值

续表 3.9.1

偏移量/字节	域	大小/字节	说 明
7	bmAttributes	1	该设备的属性
8	bMaxPower	1	设备所需要的电流(单位为 2 mA)

bLength 大小为 1 字节,表示该描述符的长度。标准的 USB 配置描述符的长度为 9 字节。

bDescriptorType 大小为 1 字节,表示描述符的类型。配置描述符的类型编码为 0x02。

wTotalLength 大小为 2 字节,表示整个配置描述符集合的总长度,包括配置描述符、接口描述符、类特殊描述符(如果有)和端点描述符。注意低字节在先。

bNumInterfaces 大小为 1 字节,表示该配置所支持的接口数量。通常,功能单一的设备只具有一个接口(例如鼠标),而复合设备则具有多个接口(例如音频设备)。

bConfiguration 大小为 1 字节,表示该配置的值。通常一个 USB 设备可以支持多个配置,bConfiguration 就是每个配置的标识。设置配置请求时会发送一个配置值,如果某个配置的 bConfiguration 值与它相匹配,就表示该配置被激活,为当前配置。

iConfiguration 大小为 1 字节,是描述该配置的字符串的索引值。如果该值为 0,则表示没有字符串。

bmAttributes 大小为 1 字节,用来描述设备的一些特性。其中,D7 是保留的,必须要设置为 1。D6 表示供电方式,当 D6 为 1 时,表示设备是自供电的;当 D6 为 0 时,表示设备是总线供电的。D5 表示是否支持远程唤醒,当 D5 为 1 时,支持远程唤醒。D4~D0 保留,设置为 0。

bMaxPower 大小为 1 字节,表示设备需要从总线获取的最大电流量,单位为 2 mA。例如,如果需要 200 mA 的最大电流,则该字节的值为 100。

3.9.2 接口描述符的结构

表 3.9.2 是 USB 协议规定的标准接口描述符的结构。接口描述符不能单独返回,必须附着在配置描述符后一并返回。

表 3.9.2 标准接口描述符的结构

偏移量/字节	域	大小/字节	说 明
0	bLength	1	该描述符的长度(9 字节)
1	bDescriptorType	1	描述符类型(接口描述符为 0x04)
2	bInterfaceNumber	1	该接口的编号(从 0 开始)
3	bAlternateSetting	1	该接口的备用编号
4	bNumEndpoints	1	该接口所使用的端点数

续表 3.9.2

偏移量/字节	域	大小/字节	说 明
5	bInterfaceClass	1	该接口所使用的类
6	bInterfaceSubClass	1	该接口所使用的子类
7	bInterfaceProtocol	1	该接口所使用的协议
8	iConfiguration	1	描述该接口的字符串的索引值

bLength 大小为1字节，表示该描述符的长度。标准的USB接口描述符的长度为9字节。

bDescriptorType 大小为1字节，是描述符的类型。接口描述符的类型编码为0x04。

bInterfaceNumber 大小为1字节，表示该接口的编号。当一个配置具有多个接口时，每个接口的编号都不相同。从0开始依次递增对一个配置的接口进行编号。

bAlternateSetting 大小为1字节，是该接口的备用编号。编号规则与bInterfaceNumber一样，很少会使用该字段，设置为0。

bNumEndpoints 大小为1字节，是该接口所使用的端点数(不包括0端点)。如果该字段为0，则表示没有非0端点，只使用默认的控制端点。

bInterfaceClass、bInterfaceSubClass、bInterfaceProtocol 分别是接口所使用的类、子类以及协议，它们的代码由USB协会定义，跟设备描述符中的意义类似。通常在接口中定义设备的功能，而在设备描述符中将类、子类以及协议字段的值设置为0。

iConfiguration 大小为1字节，是描述该接口的字符串的索引值。如果该值为0，则表示没有字符串。

3.9.3 端点描述符的结构

表3.9.3是USB协议规定的标准端点描述符的结构。端点描述符不能单独返回，必须附着在配置描述符后一并返回。

表 3.9.3 标准端点描述符的结构

偏移量/字节	域	大小/字节	说 明
0	bLength	1	该描述符的长度(7字节)
1	bDescriptorType	1	描述符的类型(端点描述符为0x05)
2	bEndpointAddress	1	该端点的地址
3	bmAttributes	1	该端点的属性
4	wMaxPackeSize	2	该端点支持的最大包长度
6	bInterval	1	端口的查询时间

bLength 大小为1字节，表示该描述符的长度。标准的USB端点描述符的长度为5字节。

bDescriptorType 大小为1字节，表示描述符的类型。端点描述符的类型编码为0x05。

bEndpointAddress 大小为1字节，表示该端点的地址。最高位D7为该端点的传输方向，1为输入(有点像Input的第一个字母)，0为输出(有点像Output的第一个字母)。D3～D0为端点号。D6～D4保留，设为0。

bmAttributes 大小为1字节，是该端点的属性。最低两位D1～D0表示该端点的传输类型，0为控制传输，1为等时传输，2为批量传输，3为中断传输。如果该端点是非等时传输的端点，那么D7～D2为保留值，设为0。如果该端点是等时传输的，则D3～2表示同步的类型，0为无同步，1为异步，2为适配，3为同步；D5～D4表示用途，0为数据端点，1为反馈端点，2为暗含反馈的数据端点，3是保留值。D7～D6保留。

wMaxPackeSize 大小为1字节，是该端点所支持的最大包长度。注意低字节在先。对于全速模式和低速模式，D10～D0表示端点的最大包长，其他位保留为0。对于高速模式，D12～D11为每个帧附加的传输次数，具体请参看USB2.0协议。

bInterval 大小为1字节，表示该端点查询的时间。对于中断端点，表示查询的帧间隔数。对于等时传输以及高速模式的中断、批量传输，该字段的意义请参看USB2.0协议。

3.9.4 HID描述符的结构

我们知道，USB鼠标是属于USB HID类的。通过查看USB HID类的官方文档，HID类的设备在配置描述符中还需要一个HID描述符。它是一个类描述符，应该跟在接口描述符后面。HID描述符的结构如表3.9.4所列。

表3.9.4 HID描述符的结构

偏移量/字节	域	大小/字节	说 明
0	bLength	1	该描述符的长度
1	bDescriptorType	1	描述符类型(HID描述符为0x21)
2	bcdHID	2	HID协议的版本
4	bCountyCode	1	国家代码
5	bNumDescriptors	1	下级描述符的数量
6	bDescriptorType	1	下级描述符的类型
7	wDescriptorLength	2	下级描述符的长度
9	bDescriptorType	1	下级描述符的类型(可选)
10	wDescriptorLength	2	下级描述符的长度(可选)
	…	…	…(可选)

bLength 大小为1字节，是该描述符的总长度。它的大小与该描述符中下级描述符的个数有关。例如，只有一个下级描述符时，总长度为1+1+2+1+1+1+2=9字节。

bDescriptorType 大小为1字节，是该描述符的编号。HID描述符的编号为0x21。

bcdHID 大小为2字节，是该设备所使用的HID协议的版本号。这里参看的HID协议是USB HID1.1协议，因此这里为0x0110。

bCountyCode 大小为1字节，是设备所适用的国家。通常我们的键盘是美式键盘，代码为33，即0x21。

bNumDescriptors 大小为1字节，是下级描述符的数量。该值至少为1，即至少要有一个报告描述符。下级描述符可以是报告描述符或物理描述符。

bDescriptorType 大小为1字节，是下级描述符的类型。报告描述符的编号为0x22，物理描述符编号为0x23。

bDescriptorLength 大小为2字节，是下级描述符的长度。当有多个下级描述符时，bDescritporType和bDescriptorLength交替重复下去。

3.10 配置描述符集合的实现以及返回

通过前面的分析知道了配置描述符集合的结构，接下来就要用代码来实现一个配置描述符集合，并在主机的获取配置描述符请求中返回。构造配置描述符集合的代码如下：

```
//USB配置描述符集合的定义
//配置描述符总长度为(9+9+9+7)字节
code uint8 ConfigurationDescriptor[9+9+9+7]=
{
 /*****************配置描述符*************************/
 //bLength字段。配置描述符的长度为9字节
0x09,

//bDescriptorType字段。配置描述符编号为0x02
0x02,

//wTotalLength字段。配置描述符集合的总长度，包括配置描述符本身、接口描述符、类描述符、端点
//描述符等
sizeof(ConfigurationDescriptor)&0xFF,          //低字节
(sizeof(ConfigurationDescriptor)>>8)&0xFF,     //高字节

//bNumInterfaces字段。该配置包含的接口数，只有一个接口
0x01,

//bConfiguration字段。该配置的值为1
0x01,
```

```
//iConfigurationz 字段。该配置的字符串索引,这里没有,为 0
0x00,

//bmAttributes 字段。该设备的属性。由于我们的板子是总线供电的,并且不想实现远程唤醒的功
//能,所以该字段的值为 0x80
0x80,

//bMaxPower 字段。该设备需要的最大电流量。由于我们的板子需要的电流不到 100 mA,所以这里
//设置为 100 mA;由于每单位电流为 2 mA,所以这里设置为 50(0x32)
0x32,
/*******************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。由于 USB 鼠标只需要一个中断输入端点,因此该值为 1
0x01,

//bInterfaceClass 字段。该接口所使用的类。USB 鼠标是 HID 类,HID 类的编码为 0x03
0x03,

//bInterfaceSubClass 字段。该接口所使用的子类。在 HID1.1 协议中,只规定了一种子类:支持 BIOS
//引导启动的子类。USB 键盘、鼠标属于该子类,子类代码为 0x01
0x01,

//bInterfaceProtocol 字段。如果子类为支持引导启动的子类,则协议可选择鼠标和键盘
//键盘代码为 0x01,鼠标代码为 0x02
0x02,

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

/*******************HID 描述符*********************/
//bLength 字段。本 HID 描述符下只有一个下级描述符,所以长度为 9 字节
0x09,

//bDescriptorType 字段。HID 描述符的编号为 0x21
0x21,

//bcdHID 字段。本协议使用 HID1.1 协议。注意低字节在先
```

```
0x10,
0x01,
//bCountyCode 字段。设备适用的国家代码,这里选择美国,代码 0x21
0x21,
//bNumDescriptors 字段。下级描述符的数目。我们只有一个报告描述符
0x01,
//bDescritporType 字段。下级描述符的类型为报告描述符,编号为 0x22
0x22,
//bDescriptorLength 字段。下级描述符的长度。下级描述符为报告描述符
sizeof(ReportDescriptor)&0xFF,
(sizeof(ReportDescriptor)>>8)&0xFF,
/**********************端点描述符**********************/
//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。我们使用 D12 的输入端点 1
//D7 位表示数据方向,输入端点 D7 为 1,所以输入端点 1 的地址为 0x81
0x81,
//bmAttributes 字段。D1~D0 为端点传输类型选择
//该端点为中断端点。中断端点的编号为 3。其他位保留为 0
0x03,
//wMaxPacketSize 字段。该端点的最大包长。端点 1 的最大包长为 16 字节,注意低字节在先
0x10,
0x00,
//bInterval 字段。端点查询的时间,我们设置为 10 个帧时间,即 10 ms
0x0A
};
//////////////////////////配置描述符集合完毕//////////////////////////
```

程序中,需要知道报告描述符的大小,但是现在还未构造出报告描述符,怎么办呢?没关系,可以随便构造一个"错误"的报告描述符充一下数吧,先让程序运行起来再说。下面是构造的报告描述符代码,只放了一个数据 0 在里面。后面会详细介绍如何构造 USB 报告描述符。

```
//USB 报告描述符
code uint8 ReportDescriptor[] =
{
```

```
 0x00
};
//////////////////////////报告描述符完毕//////////////////////////
```

然后，在获取配置描述符的请求中，增加对配置描述符返回的代码即可。修改端点0输出中断处理函数中获取配置描述符的代码如下：

```
case CONFIGURATION_DESCRIPTOR:  //配置描述符
 #ifdef DEBUG0
  Prints("配置描述符。\r\n");
 #endif
 pSendData = ConfigurationDescriptor;  //需要发送的数据为配置描述符
 //判断请求的字节数是否比实际需要发送的字节数多
 //这里请求的是配置描述符集合，因此数据长度就是
 //ConfigurationDescriptor[3] * 256 + ConfigurationDescriptor[2]
 //如果请求的比实际的长，那么只返回实际长度的数据
 SendLength = ConfigurationDescriptor[3];
 SendLength = SendLength * 256 + ConfigurationDescriptor[2];
 if(wLength>SendLength)
 {
  if(SendLength % DeviceDescriptor[7] == 0)  //并且刚好是整数个数据包时
  {
   NeedZeroPacket = 1;  //需要返回0长度的数据包
  }
 }
 else
 {
  SendLength = wLength;
 }
 //将数据通过EP0返回
 UsbEp0SendData();
break;
```

然后编译代码，下载到学习板中运行，查看串口的返回的调试信息，内容显示如图3.10.1所示。从图中可以看到，程序返回了9字节的配置描述符，其中0x22 0x00表示描述符结合的总长度为34字节。主机接着又发送了获取字符串描述符的请求，并且请求的索引值为0。这个请求不是请求索引为0的字符串，因为没有哪个字符串可以使用索引值0。这个请求是请求语言ID用的。接下来，就要实现返回语言ID以及各字符串的请求了。

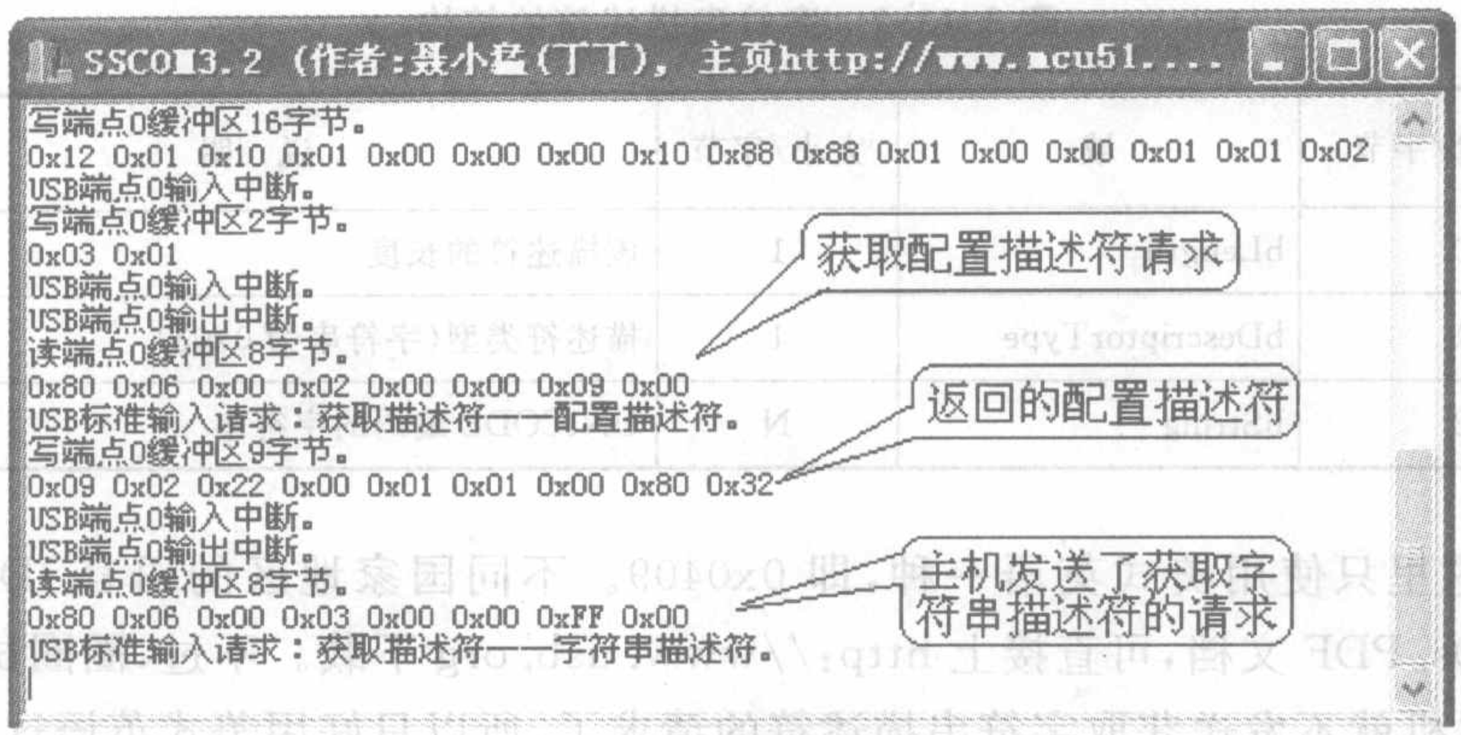


图 3.10.1　返回 9 字节的配置描述符后的调试信息

3.11　字符串及语言 ID 请求的实现

在 USB 协议中，字符串描述符是可选的。当某个描述符中的字符串索引值为非 0 时，就表示它具有那个字符串描述符，注意索引值不能重复。在设备描述符中，申请了 3 个非 0 的索引值，分别是厂商字符串、产品字符串以及产品序列号；其索引值分别为 1，2，3。USB 主机使用获取字符串描述符和索引值来获取对应的字符串。当索引值为 0 时，表示获取语言 ID。语言 ID 是一个描述该设备支持的语言种类的数组，每个 ID 号占 2 字节。

字符串描述符的结构很简单，如表 3.11.1 和表 3.11.2 所列。其中表 3.11.1 为语言 ID 描述符结构，表 3.11.2 为字符串描述符结构。

表 3.11.1　语言 ID 描述符的结构

偏移量/字节	域	大小/字节	说　明
0	bLength	1	该描述符的长度
1	bDescriptorType	1	描述符类型(字符串为 0x03)
2	wLANGID[0]	2	语言 ID 号 0
⋮	⋮	⋮	⋮
2 * n+2	wLANGID[n]	2	语言 ID 号 n

表 3.11.2 字符串描述符的结构

偏移量/字节	域	大小/字节	说 明
0	bLength	1	该描述符的长度
1	bDescriptorType	1	描述符类型(字符串为 0x03)
2	bString	N	UNICODE 编码的字符串

语言 ID,这里只使用美式英语一种,即 0x0409。不同国家地区的语言 ID 号,可以查看 USB_LANGIDs. PDF 文档,可直接上 http://www.usb.org 下载。不过,圈圈试着将该 ID 换成中国的,则主机就不发送获取字符串描述符的请求了,所以只好用美式英语这个了。

字符串描述符中的 bString 字段是使用 UNICODE 编码的字符串。UNICODE 用 2 字节来表示一个字符,如果是英文字符,则直接在 ASCII 码前补 1 字节的 0 扩充为 2 字节的 UNICODE 码。通常在程序中直接用双引号引起来的字符串使用的是 GB 码,它跟 UNICODE 编码没有什么规律。

为了方便使用,圈圈用 Java 脚本写了专门生成字符串描述符的小工具,大家可以直接在里面输入需要的字符串,然后点击"转换"按钮,就可以生成所需要的字符串描述符了。该工具的地址为 http://computer00.21ic.org/user1/2198/archives/2007/42769.html。

这里将厂商字符串设置为"电脑圈圈的 USB 专区 http://group.ednchina.com/93/",产品字符串设置为"《圈圈教你学 USB》之 USB 鼠标",产品序列号嘛,可以随便一点,今天是 2008 年 7 月 7 日,就用"2008-07-07"吧!或者你自己给你的每个产品一个唯一的编号也行,这样做还可以知道客户手头的设备是否是你们公司卖出去的。将上面三个字符串分别填入到刚刚介绍的字符串描述符生成工具中,然后点击"转换"按钮即可获得字符串描述符,分别改名为 ManufacturerStringDescriptor、ProductStringDescriptor、SerialNumberStringDescriptor。最终的语言 ID 以及各字符串描述符代码如下:

```
/*********************语言 ID 的定义*******************/
code uint8 LanguageId[4] =
{
 0x04,                    //本描述符的长度
 0x03,                    //字符串描述符
 //0x0409 为美式英语的 ID
 0x09,
 0x04
};
//////////////////////语言 ID 完毕/////////////////////////////////
```

```
/*********************************************************************/
/*********        本转换结果来自 Http://computer00.21ic.org        **********/
/*********                作者:电脑圈圈                            **********/
/*********                欢迎大家使用                             **********/
/*********              版权所有,盗版请写明出处                    **********/
/*********************************************************************/

//http://computer00.21ic.org/user1/2198/archives/2007/42769.html
//字符串"电脑圈圈的 USB 专区 Http://group.ednchina.com/93/"的 UNICODE 编码
//8 位小端格式
code uint8 ManufacturerStringDescriptor[82] = {
82,             //该描述符的长度为 82 字节
0x03,           //字符串描述符的类型编码为 0x03
0x35, 0x75,     //电
0x11, 0x81,     //脑
0x08, 0x57,     //圈
0x08, 0x57,     //圈
0x84, 0x76,     //的
0x55, 0x00,     //U
0x53, 0x00,     //S
0x42, 0x00,     //B
0x13, 0x4e,     //专
0x3a, 0x53,     //区
0x20, 0x00,     //
0x48, 0x00,     //H
0x74, 0x00,     //t
0x74, 0x00,     //t
0x70, 0x00,     //p
0x3a, 0x00,     //:
0x2f, 0x00,     ///
0x2f, 0x00,     ///
0x67, 0x00,     //g
0x72, 0x00,     //r
0x6f, 0x00,     //o
0x75, 0x00,     //u
0x70, 0x00,     //p
0x2e, 0x00,     //.
0x65, 0x00,     //e
0x64, 0x00,     //d
0x6e, 0x00,     //n
```

```
    0x63, 0x00,          //c
    0x68, 0x00,          //h
    0x69, 0x00,          //i
    0x6e, 0x00,          //n
    0x61, 0x00,          //a
    0x2e, 0x00,          //.
    0x63, 0x00,          //c
    0x6f, 0x00,          //o
    0x6d, 0x00,          //m
    0x2f, 0x00,          ///
    0x39, 0x00,          //9
    0x33, 0x00,          //3
    0x2f, 0x00           ///
    };
    ////////////////////////厂商字符串结束////////////////////////////
    //字符串"《圈圈教你玩 USB》之 USB 鼠标"的 UNICODE 编码
    //8 位小端格式
    code uint8 ProductStringDescriptor[34] = {
    34,                  //该描述符的长度为 34 字节
    0x03,                //字符串描述符的类型编码为 0x03
    0x0a, 0x30,          //《
    0x08, 0x57,          //圈
    0x08, 0x57,          //圈
    0x59, 0x65,          //教
    0x60, 0x4f,          //你
    0xa9, 0x73,          //玩
    0x55, 0x00,          //U
    0x53, 0x00,          //S
    0x42, 0x00,          //B
    0x0b, 0x30,          //》
    0x4b, 0x4e,          //之
    0x55, 0x00,          //U
    0x53, 0x00,          //S
    0x42, 0x00,          //B
    0x20, 0x9f,          //鼠
    0x07, 0x68           //标
    };
    ////////////////////////产品字符串结束////////////////////////////
    //字符串"2008 - 07 - 07"的 UNICODE 编码
```

```
//8位小端格式
code uint8 SerialNumberStringDescriptor[22] = {
22,                     //该描述符的长度为22字节
0x03,                   //字符串描述符的类型编码为0x03
0x32, 0x00,             //2
0x30, 0x00,             //0
0x30, 0x00,             //0
0x38, 0x00,             //8
0x2d, 0x00,             //-
0x30, 0x00,             //0
0x37, 0x00,             //7
0x2d, 0x00,             //-
0x30, 0x00,             //0
0x37, 0x00              //7
};
//////////////////////产品序列号字符串结束//////////////////////////
```

接下来，再回到端点0输出中断处理的函数中，增加对获取描述符的相关处理。该处理过程主要是对wValue的低字节进行散转，以判断返回哪个具体的字符串描述符。代码如下：

```
case STRING_DESCRIPTOR:  //字符串描述符
 #ifdef DEBUG0
  Prints("字符串描述符");
 #endif
 switch(wValue&0xFF)  //根据wValue的低字节(索引值)散转
 {
  case 0:  //获取语言ID
   #ifdef DEBUG0
   Prints("(语言ID)。\r\n");
  #endif
  pSendData = LanguageId;
  SendLength = LanguageId[0];
 break;

 case 1:  //厂商字符串的索引值为1,所以这里为厂商字符串
 #ifdef DEBUG0
   Prints("(厂商描述)。\r\n");
  #endif
  pSendData = ManufacturerStringDescriptor;
  SendLength = ManufacturerStringDescriptor[0];
 break;
```

```
case 2:  //产品字符串的索引值为2,所以这里为产品字符串
 #ifdef DEBUG0
  Prints("(产品描述)。\r\n");
 #endif
 pSendData = ProductStringDescriptor;
 SendLength = ProductStringDescriptor[0];
break;

case 3:  //产品序列号的索引值为3,所以这里为产品序列号
 #ifdef DEBUG0
  Prints("(产品序列号)。\r\n");
 #endif
 pSendData = SerialNumberStringDescriptor;
 SendLength = SerialNumberStringDescriptor[0];
 break;

 default :
  #ifdef DEBUG0
   Prints("(未知的索引值)。\r\n");
   #endif
  //对于未知索引值的请求,返回一个0长度的包
  SendLength = 0;
  NeedZeroPacket = 1;
 break;
}
//判断请求的字节数是否比实际需要发送的字节数多
//如果请求的比实际的长,那么只返回实际长度的数据
if(wLength>SendLength)
{
 if(SendLength % DeviceDescriptor[7] == 0)  //并且刚好是整数个数据包时
 {
  NeedZeroPacket = 1;  //需要返回0长度的数据包
 }
}
else
{
 SendLength = wLength;
}
//将数据通过EP0返回
UsbEp0SendData();
break;
```

将程序编译、烧入到学习板中，运行后会在右下角显示发现新硬件的对话框，通过串口调试助手可以观察到返回的调试信息。由于调试信息太长，这里就不给出了(见附录)。调试信息显示主机在获取语言ID后，接着获取了产品序列号。然后再一次获取配置描述符，这次使用的长度为0xFF，所以这次程序返回了全部的配置描述符集合。然后再次获取语言ID和产品描述字符串。接着又获取了设备描述符和配置描述符集合，最后发送了设置配置的请求，如图3.11.1所示。

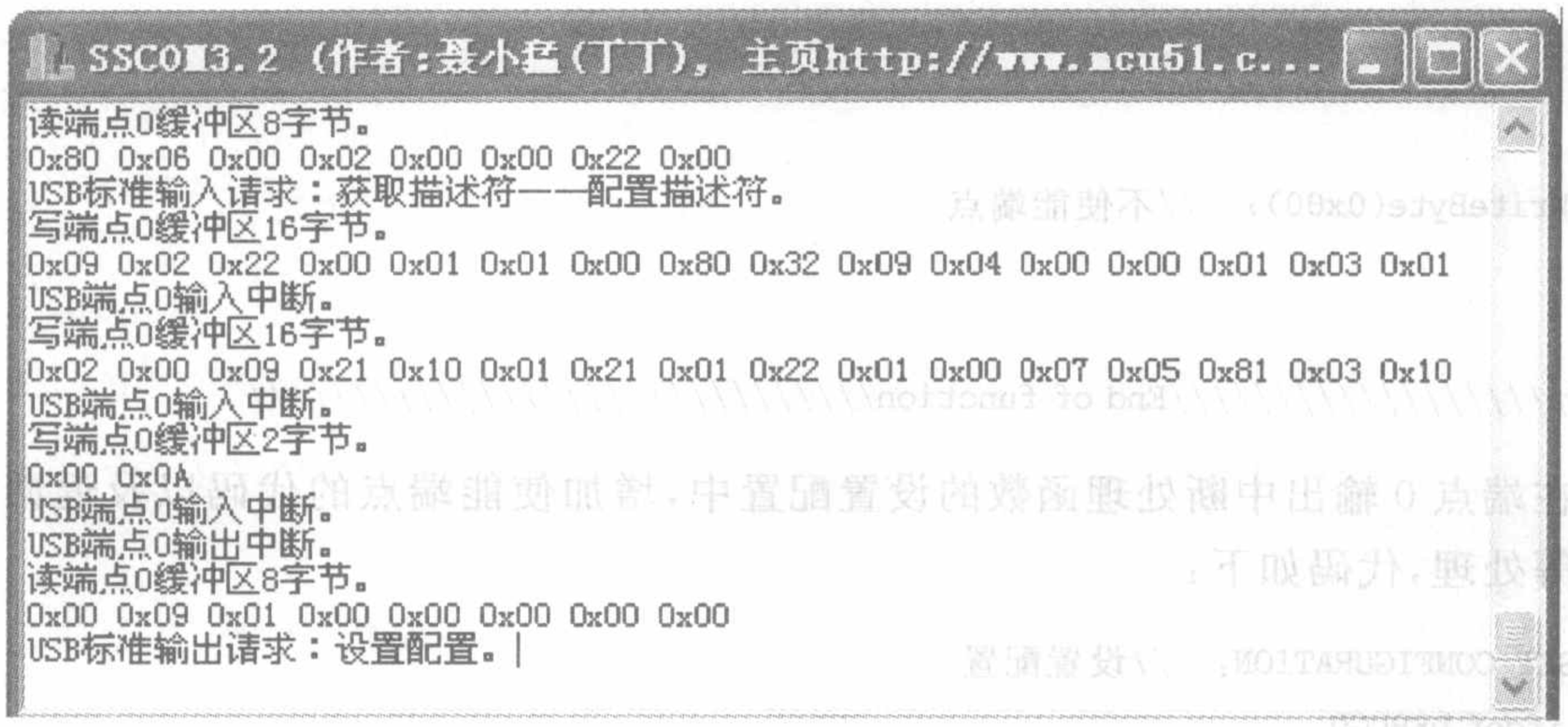


图3.11.1 主机最后发送了设置配置的请求

3.12 设置配置请求的实现

设置配置请求的实现比较简单，它是一个输出请求，只要根据所请求的配置值，使能相应的端点即可。由于我们的鼠标只有一个配置，所以连配置值都可以忽略，直接使能端点，然后返回一个0长度的状态数据包即可。注意，只有收到非0的配置值之后才可以使能非0端点，否则要禁用非0端点。

使能D12的非0端点，要用到D12的SetEndpointEable的命令，命令代码为0xD8，后面跟1字节的数据写入。该字节的最低位D0为1时，使能端点。其他位保留，为0。该命令只有在设置地址命令时使能了设备(D7为1)后才能起作用。这一步在设置地址时我们已经做过了。实现使能端点的函数代码如下：

```
/*********************************************************************
函数功能:使能端点函数
入口参数:Enable   是否使能。0值为不使能,非0值为使能
返      回:无
备      注:无
```

```
*****************************************************************************/
void D12SetEndpointEnable(uint8 Enable)
{
 D12WriteCommand(D12_SET_ENDPOINT_ENABLE);
 if(Enable! = 0)
 {
  D12WriteByte(0x01);  //D0 为 1 使能端点
 }
 else
 {
  D12WriteByte(0x00);  //不使能端点
 }
}
//////////////////////End of function////////////////////////////
```

然后在端点0输出中断处理函数的设置配置中，增加使能端点的代码以及返回0长度状态数据包等处理，代码如下：


```
case SET_CONFIGURATION:  //设置配置
    #ifdef DEBUG0
     Prints("设置配置。\r\n");
    #endif
    //使能非 0 端点。非 0 端点只有在设置为非 0 的配置后才能使能
    //wValue 的低字节为配置的值，如果该值为非 0，才能使能非 0 端点
    D12SetEndpointEnable(wValue&0xFF);
    //返回一个 0 长度的状态数据包
    SendLength = 0;
    NeedZeroPacket = 1;
    //将数据通过 EP0 返回
    UsbEp0SendData();
break;
```

然后编译程序，下载运行，可以看到返回的调试信息，如图3.12.1所示。

可以看到，主机发送了一个bRequest为0x0A的类输出请求。那么这个请求是什么意思呢？这就需要查看HID协议的文档了。在HID协议的文档中，定义了一些类请求，例如Set_Idle、Get_Idle、Get_Report、Set_Report等。其中，编码为0x0A的请求就是Set_Idle请求。这个请求告诉设备，在没有新的事件发生时，不要从中断端点返回数据。对于我们的USB鼠标来说，收到这个请求可以什么都不用干，直接返回一个0长度的状态数据包即可。修改端点0输出中断函数中的类输出请求部分代码如下：

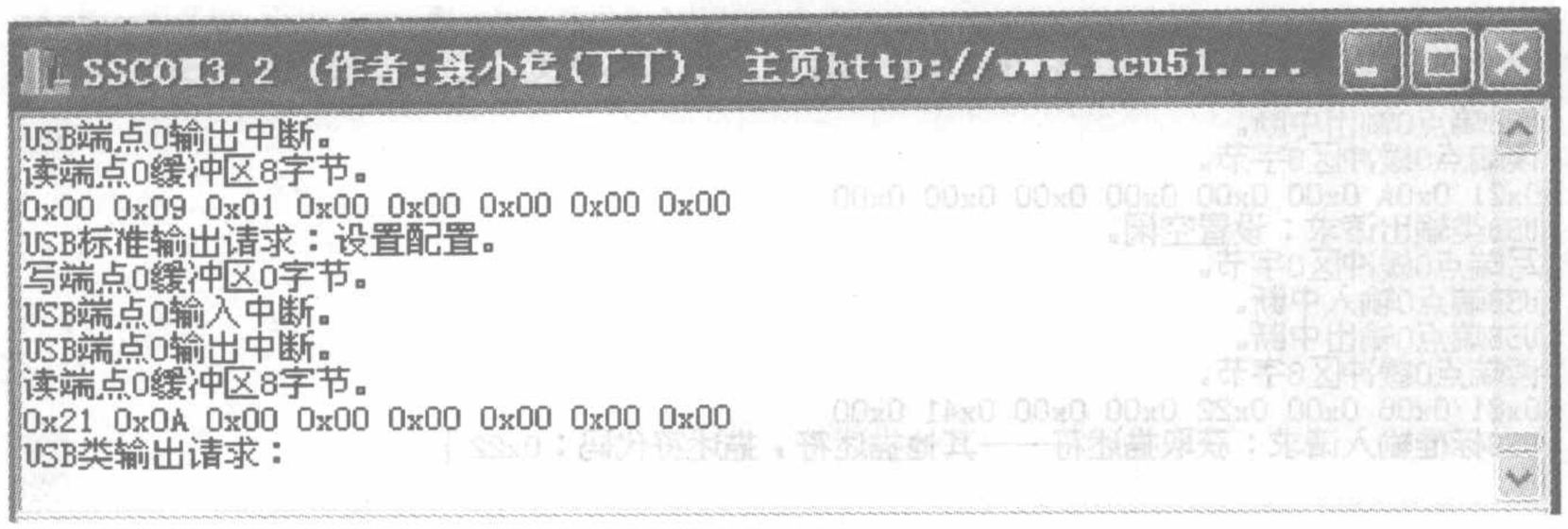


图 3.12.1 设置配置后的调试信息

```
case 1:  //类请求
  #ifdef DEBUG0
   Prints("USB 类输出请求:");
  #endif
  switch(bRequest)
  {
   case SET_IDLE:
    #ifdef DEBUG0
     Prints("设置空闲。\r\n");
    #endif
    //只需要返回一个 0 长度的数据包即可
    SendLength = 0;
    NeedZeroPacket = 1;
    //将数据通过 EP0 返回
    UsbEp0SendData();
   break;
   default:
    #ifdef DEBUG0
     Prints("未知请求。\r\n");
    #endif
   break;
  }
 break;
```

然后再编译，下载，程序运行后返回的调试信息如图 3.12.2 所示。

从调试信息中看到，主机发送了一个获取描述符的请求，接收者是接口(bmRequestType 的 D4～D0 位值为 1)，请求的描述符代码为 0x22。这个描述符代码在前面已经介绍过了，它是 HID 协议定义的报告描述符。为了简化代码，这里并没有判断请求的接收者，仅靠描述符

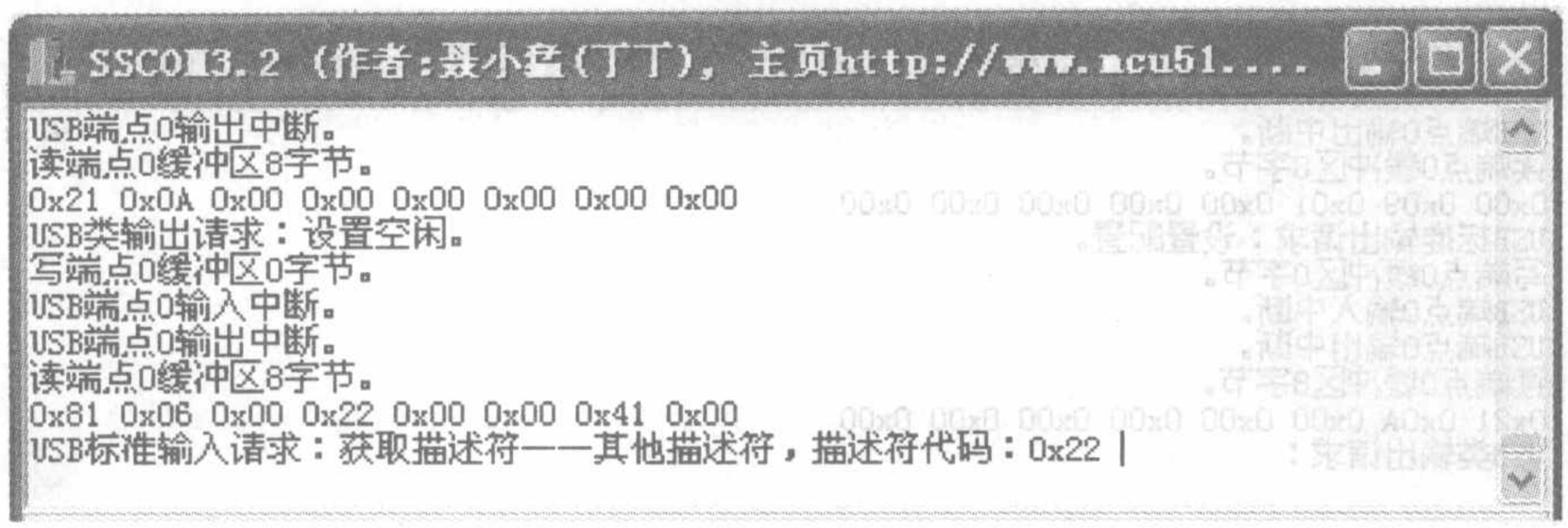


图 3.12.2 设置空闲后返回的调试信息

代码来区分。但是这里显示的请求长度怎么是0x0041呢？我们暂时定义的报告描述符不是只有1字节吗？这个问题圈圈也搞不懂，先不管它吧！反正它请求的长度比实际的报告描述符长度大就够了，因为程序不会返回过多的描述符数据。接下来就要实现这个报告描述符了，报告描述符是最复杂，也是最让人头疼的一个描述符。关于报告描述符，圈圈不可能花大篇幅去细讲它了，只能根据需要，选一些出来讲解。详细、完整的报告描述符资料可以参看USB HID协议以及USB用途表的文档，里面有很多实例。

3.13 报告描述符的结构及实现

USB HID设备是通过报告(report)来传送数据的，报告有输入报告和输出报告。

输入报告是USB设备发送给主机的，例如：USB鼠标将鼠标移动和鼠标点击等信息返回给计算机，键盘将按键数据返回给计算机等。

输出报告是主机发送给USB设备的，例如：键盘上的数字键盘锁定灯和大写字母锁定灯的控制等。

报告里面包含的是所要传送的数据，数量为整数字节，被划分成一个个域。通常，输入报告是通过中断输入端点返回的，而输出报告有点区别，当没有中断输出端点时，可以通过控制输出端点0发送，当有中断输出端点时，通过中断输出端点发出。当然，不管设备是否具有中断输出端点(中断输入端点是必须要的)，主机都可以通过获取报告和设置报告的请求从端点0来获取或者发送报告。

而报告描述符(report descriptor)，是用来描述一个报告的结构以及该报告里面的数据是用来干什么用的。通过报告描述符，USB主机就可以分析出报告里面的数据所表示的意义。报告描述符与普通描述符一样，都是通过控制输入端点0来返回，主机使用获取报告描述符请求来获取报告描述符，注意这个请求是发送到接口的，而不是到设备。一个报告描述符可以描述多个报告，不同的报告通过报告ID来识别。报告ID放在报告的最前面，即第一个字节。当报告描述符中没有规定报告ID时，报告中就没有ID字段，开始就是数据。

那么报告描述符的结构是怎样的？又是怎样来定义这些报告数据的意义的呢？报告描述符与前面所遇到的描述符结构不一样，它并没有描述符长度和描述符类型等信息，而是由一个个条目(item)组成的。通常，在写报告描述符时，一个条目占据一行，这样看起来清晰一些。

HID协议中规定了两种条目：短条目和长条目。长条目很少使用，这里不介绍，只介绍短条目的结构。

短条目由1字节的前缀后面跟上可选的数据字节组成。可选的数据字节可以为0字节、1字节、2字节或者4字节。实际所使用的条目，大部分是只有1字节可选数据的，少数会使用0字节或2字节数据。条目的前缀结构如图3.13.1所示。

7 6 5 4	3 2	1 0
bTag	bType	bSize

图3.13.1 条目前缀的结构

前缀的最低两位D1和D0为bSize，用来表示后面所跟数据的字节数，0为0字节，1为1字节，2为2字节，3为4字节。D3和D2位为bType，表示条目的类型，0为主(main)条目，1为全局(global)条目，2为局部(local)条目，3是保留值。bTag表示该条目的功能，具体请参看HID协议以及HID用途表。

主条目总共有5个，分别为Input(输入)、Output(输出)、Feature(特性)、Collection(集合)和End Collection(关集合)。主条目用来定义或者分组报告的数据域，例如，可以使用输入主条目将输入报告划分为不同的数据域，以及指定该域的属性。对于Input、Output、Feature三个主条目，后面跟的第一字节数据每个位的数据表示一种属性，例如：位0表示该数据域是变量还是常量，位1表示是数组还是单一变量，位2表示是相对值还是绝对值等。

全局条目主要用来选择用途页，定义数据域的长度、数量、报告ID等。全局条目在出现后对接下来的所有主条目都有效，除非遇到另外一个全局条目来改变它。常用的全局条目有：Usage Page(用途页)、Logical Minimum(逻辑最小值)、Logical Maximum(逻辑最大值)、Physical Minimum(物理最小值)、Physical Maximum(物理最大值)、Report Size(数据域大小)、Report Count(数据域数量)和Report ID(报告ID)。其中，Report Size用来描述某个数据域有多少个位；Report Count用来描述这样的数据域有多少个；Logical Minimum和Logical Maximum用来描述数据域的取值范围。

局部条目用来定义控制的特性，例如，该数据域的用途、用途最小值、用途最大值等。局部条目只在局部有效，遇到一个主条目后，它的效用就结束了。常用的局部条目有：Usage(用途)、Usage Minimum(用途最小值)和Usage Maximum(用途最大值)。

各种条目的功能值在此就不一一列举了，大家可以对照协议来设计报告描述符，也可以使用USB官方网站提供的HID描述符工具来生成；还可以使用现成的报告描述符进行修改，例如，在HID协议以及用途表文档中，就有很多现成的例子。

下面给出USB鼠标描述符的实际代码，并在代码中增加注释来说明这个报告描述符的意义。

```
//USB 报告描述符的定义
code uint8 ReportDescriptor[] =
{
 //每行开始的第一字节为该条目的前缀,前缀的格式为:
 //D7～D4:bTag;D3～D2:bType;D1～D0:bSize。以下分别对每个条目注释
 //这是一个全局(bType 为 1)条目,选择用途页为普通桌面 Generic Desktop Page(0x01)
 //后面跟一字节数据(bSize 为 1),后面的字节数就不注释了,自己根据 bSize 来判断
 0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
 //这是一个局部(bType 为 2)条目,说明接下来的应用集合用途用于鼠标
 0x09, 0x02,                    // USAGE (Mouse)
 //这是一个主条目(bType 为 0)条目,开集合,后面跟的数据 0x01 表示该集合是一个应用集合
 //它的性质在前面由用途页和用途定义为普通桌面用的鼠标
 0xa1, 0x01,                    // COLLECTION (Application)
 //这是一个局部条目。说明用途为指针集合
 0x09, 0x01,                    // USAGE (Pointer)
 //这是一个主条目,开集合,后面跟的数据 0x00 表示该集合是一个物理集合
 //用途由前面的局部条目定义为指针集合
 0xa1, 0x00,                    // COLLECTION (Physical)
 //这是一个全局条目,选择用途页为按键(Button Page(0x09))
 0x05, 0x09,                    // USAGE_PAGE (Button)
 //这是一个局部条目,说明用途的最小值为 1。实际上是鼠标左键
 0x19, 0x01,                    // USAGE_MINIMUM (Button 1)
 //这是一个局部条目,说明用途的最小值为 3。实际上是鼠标中键
 0x29, 0x03,                    // USAGE_MAXIMUM (Button 3)
 //这是一个全局条目,说明返回的数据的逻辑值(就是我们返回的数据域的值啦)最小为 0
 //因为这里用"位"来表示一个数据域,因此最小为 0,最大为 1
 0x15, 0x00,                    // LOGICAL_MINIMUM (0)
 //这是一个全局条目,说明逻辑值最大为 1
 0x25, 0x01,                    // LOGICAL_MAXIMUM (1)
 //这是一个全局条目,说明数据域的数量为三个
 0x95, 0x03,                    // REPORT_COUNT (3)
 //这是一个全局条目,说明每个数据域的长度为 1 个位
 0x75, 0x01,                    // REPORT_SIZE (1)
 //这是一个主条目,说明有 3 个长度为 1 位的数据域(数量和长度由前面的两个全局条目所定义)
 //用来作为输入,属性为:Data,Var,Abs。Data 表示这些数据可以变动;Var 表示这些数据域是独立的变量,
 //即每个域表示一个意思;Abs 表示绝对值
 //这样定义的结果就是,第一个数据域位 0 表示按键 1(左键)是否按下,第二个数据域位 1 表示
 //按键 2(右键)是否按下,第三个数据域位 2 表示按键 3(中键)是否按下
 0x81, 0x02,                    // INPUT (Data,Var,Abs)
```

```
//这是一个全局条目,说明数据域数量为1个
0x95, 0x01,                    // REPORT_COUNT (1)
//这是一个全局条目,说明每个数据域的长度为5位
0x75, 0x05,                    // REPORT_SIZE (5)
//这是一个主条目,输入用。由前面两个全局条目可知,长度为5位,数量为1个
//它的属性为常量(即返回的数据一直是0)
//这个只是为了凑齐1字节(前面用了3个位)而填充的一些数据而已,所以它是没有实际用途的
0x81, 0x03,                    // INPUT (Cnst,Var,Abs)
//这是一个全局条目,选择用途页为普通桌面 Generic Desktop Page(0x01)
0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
//这是一个局部条目,说明用途为X轴
0x09, 0x30,                    // USAGE (X)
//这是一个局部条目,说明用途为Y轴
0x09, 0x31,                    // USAGE (Y)
//这是一个局部条目,说明用途为滚轮
0x09, 0x38,                    // USAGE (Wheel)
//下面两个为全局条目,说明返回的逻辑最小和最大值。因为鼠标指针移动时,通常是用相对值来
//表示的,相对值的意思就是,当指针移动时,只发送移动量。
//往右移动时,X值为正;往下移动时,Y值为正。对于滚轮,当滚轮往上滚时,值为正
0x15, 0x81,                    // LOGICAL_MINIMUM (-127)
0x25, 0x7f,                    // LOGICAL_MAXIMUM (127)
//这是一个全局条目,说明数据域的长度为8位
0x75, 0x08,                    // REPORT_SIZE (8)
//这是一个全局条目,说明数据域的个数为3个
0x95, 0x03,                    // REPORT_COUNT (3)
//这是一个主条目。它说明这三个8位的数据域是输入用的,属性为:Data,Var,Rel
//Data说明数据是可以变的,Var说明这些数据域是独立的,即第一个8位表示X轴
//第二个8位表示Y轴,第三个8位表示滚轮。Rel表示这些值是相对值
0x81, 0x06,                    // INPUT (Data,Var,Rel)
//下面这两个主条目用来关闭前面的集合用
//因为开了两个集合,所以要关两次。bSize为0,所以后面没数据
0xc0,                          // END_COLLECTION
0xc0                           // END_COLLECTION
};
//////////////////////////报告描述符完毕//////////////////////////
```

通过上面的报告描述符的定义可知,返回的输入报告具有4字节。第一字节的低3位用来表示按键是否按下,高5位为常数0,无用;第二字节表示X轴的改变量;第三字节表示Y轴的改变量;第四字节表示滚轮的改变量。在中断输入端点1中应该要按照上面的格式返回实

际的鼠标数据。

接下来需要到获取描述符的处理中，增加对获取报告描述符请求的处理代码。在获取描述符的 switch 散转代码中，增加以下代码：

```
case REPORT_DESCRIPTOR:  //报告描述符
 #ifdef DEBUG0
  Prints("报告描述符。\r\n");
 #endif
 pSendData = ReportDescriptor;  //需要发送的数据为报告描述符
 SendLength = sizeof(ReportDescriptor);  //需要返回的数据长度
 //判断请求的字节数是否比实际需要发送的字节数多
 //如果请求的比实际的长，那么只返回实际长度的数据
 if(wLength>SendLength)
 {
  if(SendLength % DeviceDescriptor[7] == 0)  //并且刚好是整数个数据包时
  {
   NeedZeroPacket = 1;  //需要返回 0 长度的数据包
  }
 }
 else
 {
  SendLength = wLength;
 }
 //将数据通过 EP0 返回
 UsbEp0SendData();
break;
```

再次编译、下载程序到学习板中运行，出现调试信息如图 3.13.2 所示，可以看到设备成功返回了 0x34 字节的报告描述符。接下来就没有动静了，这说明主机已经不再发送请求，这样枚举过程就算顺利完成了(不容易啊!)。接下来主机应该就会一直在查询端点 1，以读取鼠标返回的报告，在这里看不到动作，是因为端点 1 还没有成功发送数据。接下来的工作就是扫描鼠标的动作，然后将数据通过端点 1 返回。不知道你是否留意主机请求的报告描述符长度？这里是 0x74，比实际的 0x34 大 0x40 字节，而前面的报告描述符长度为 0x01 字节时，它是 0x41，也比实际的大 0x40 字节。为什么要大 0x40 字节？这个问题圈圈也没想明白，哪位朋友要是知情的话，记得告诉圈圈一声哦。

在进行下一步工作之前，先来炫耀一下我们工作的成果吧。打开设备管理器，然后展开“人体学输入设备”，看看里面是否多出了一个“USB 人体学输入设备”，再展开“鼠标和其他指针设备”，看看里面是否多出了一个“HID-compliant mouse”。如果是，那么恭喜你，成功了。

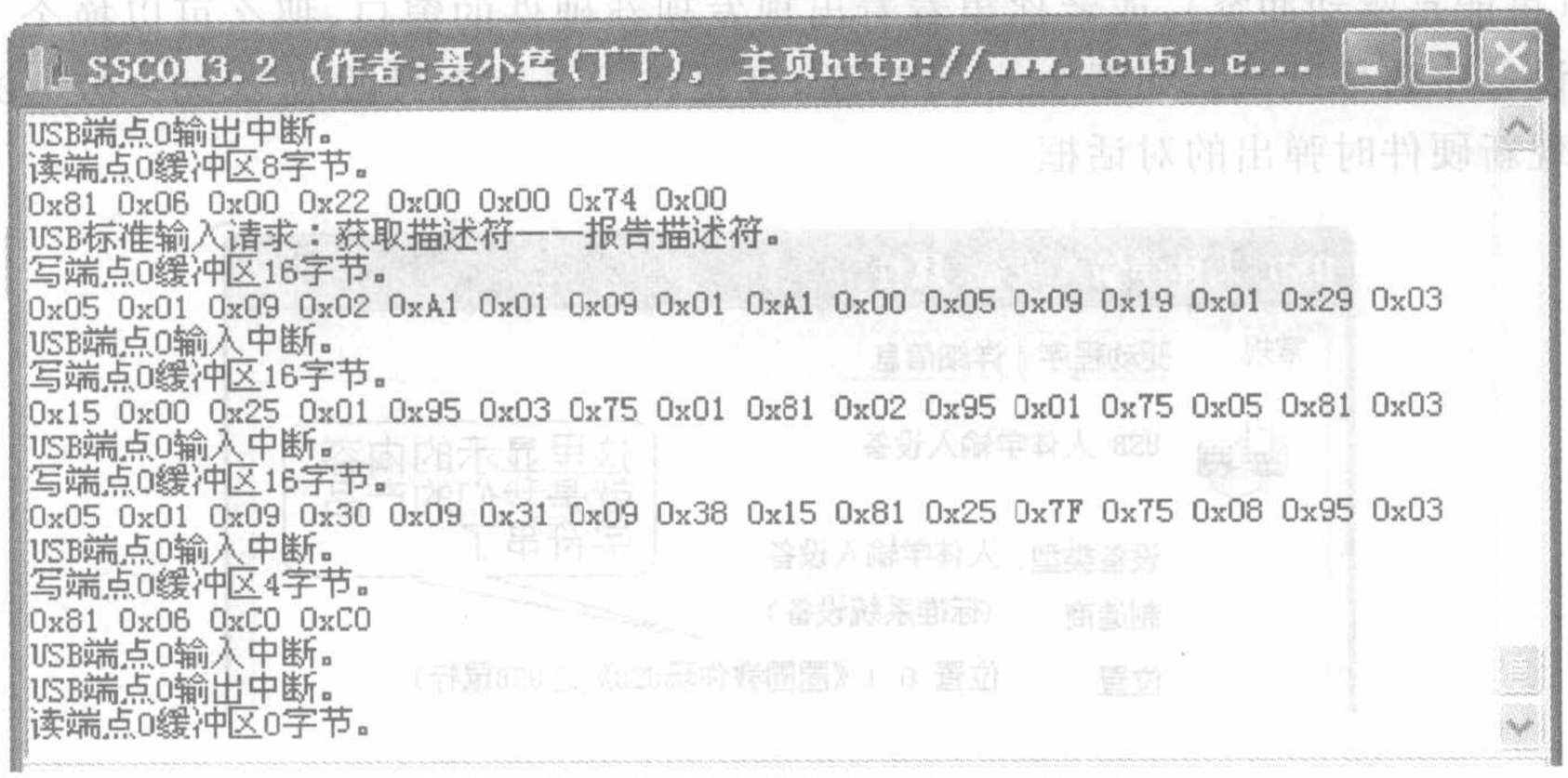


图 3.13.2 返回报告描述符时的调试信息

圈圈的计算机中显示的内容如图 3.13.3 所示。然后再双击“USB 人体学输入设备”，在常规标签页中，有一个“位置：”，里面应该显示“位置 0（《圈圈教你玩 USB》之 USB 鼠标）”，如图 3.13.4 所示。然后再切换到“详细信息”标签页，在下拉列表中选择“设备范例 ID”，可以看到这里其实就是我们所定义的厂商 ID、产品 ID 以及设备序列号，如图 3.13.5 所示。再选择“硬件 ID”，可以看到除了厂商 ID、产品 ID 外还有我们定义的版本号：0100。如果你的设备工

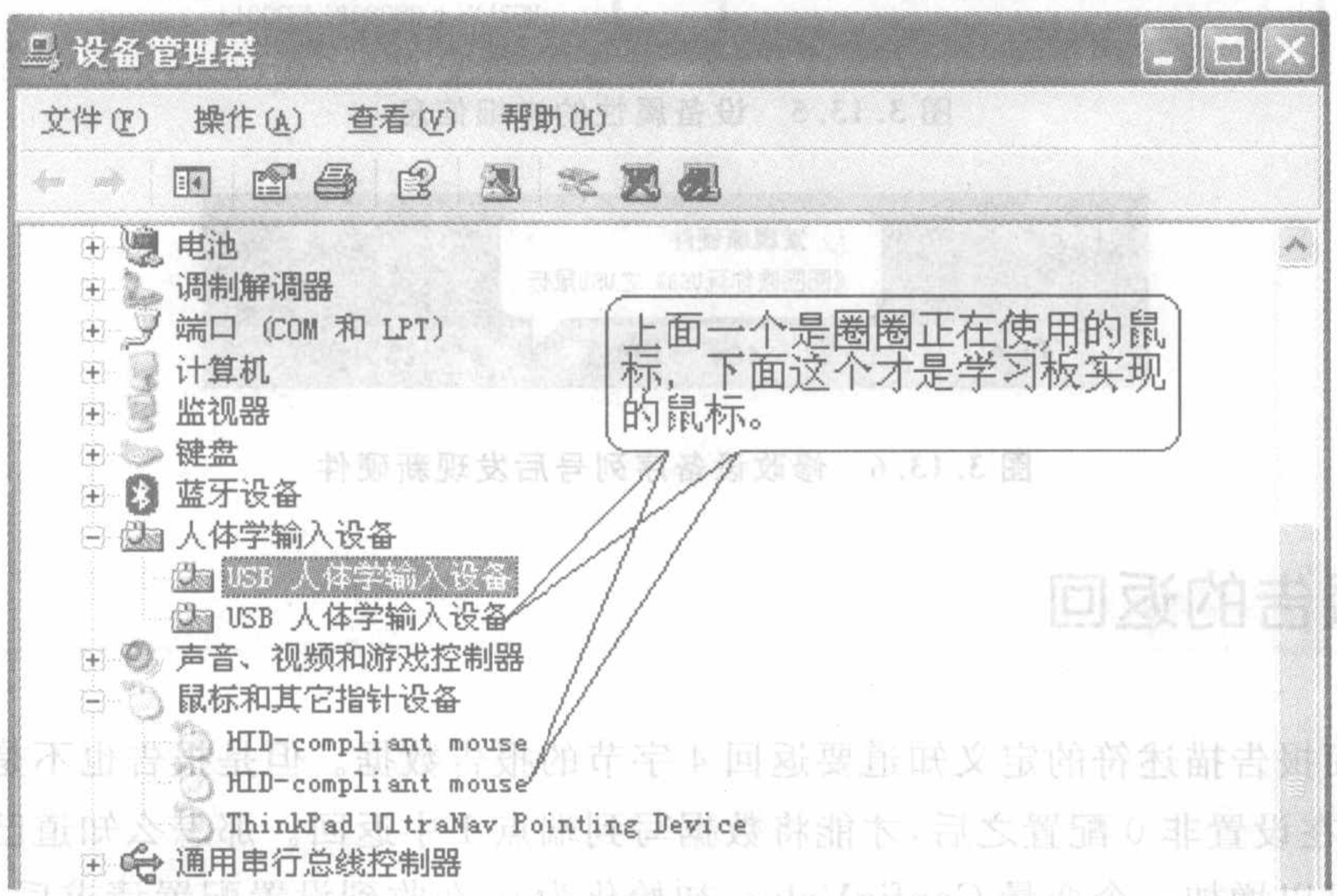


图 3.13.3 设备管理器中出现的新设备

作不正常(可能是驱动冲突),或者你想看看出现发现新硬件的窗口,那么可以换个 VID、PID 或者修改设备序列号试试。如图 3.13.6 所示,就是将设备序列号改成 2008-07-09 后重新运行时发现新硬件时弹出的对话框。

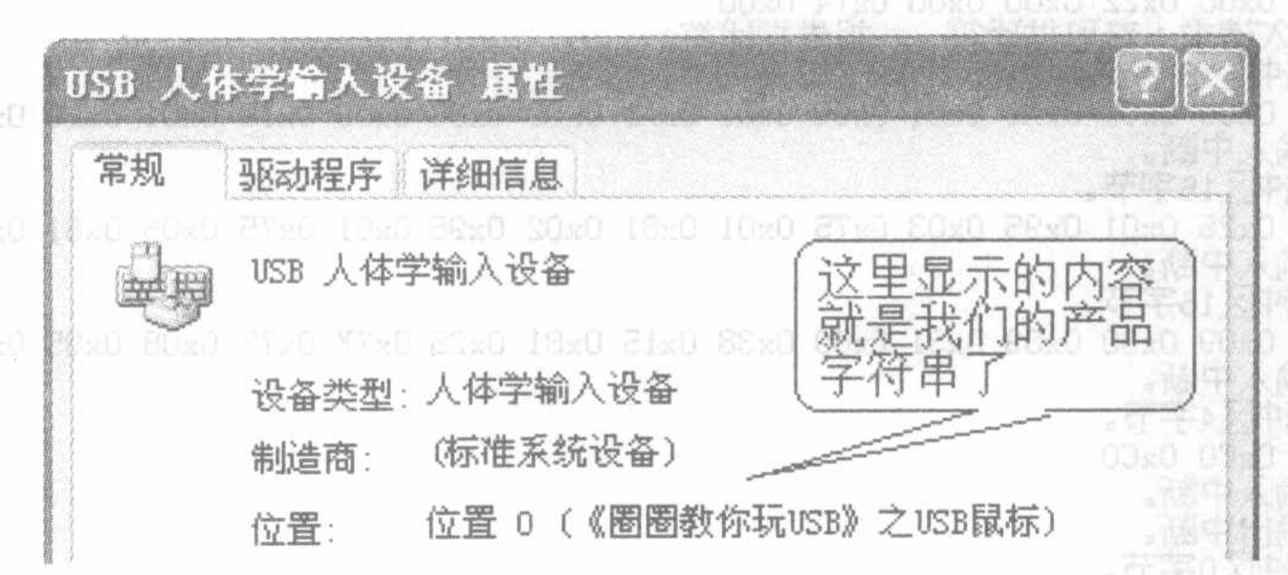


图 3.13.4 常规选项中显示的产品字符串

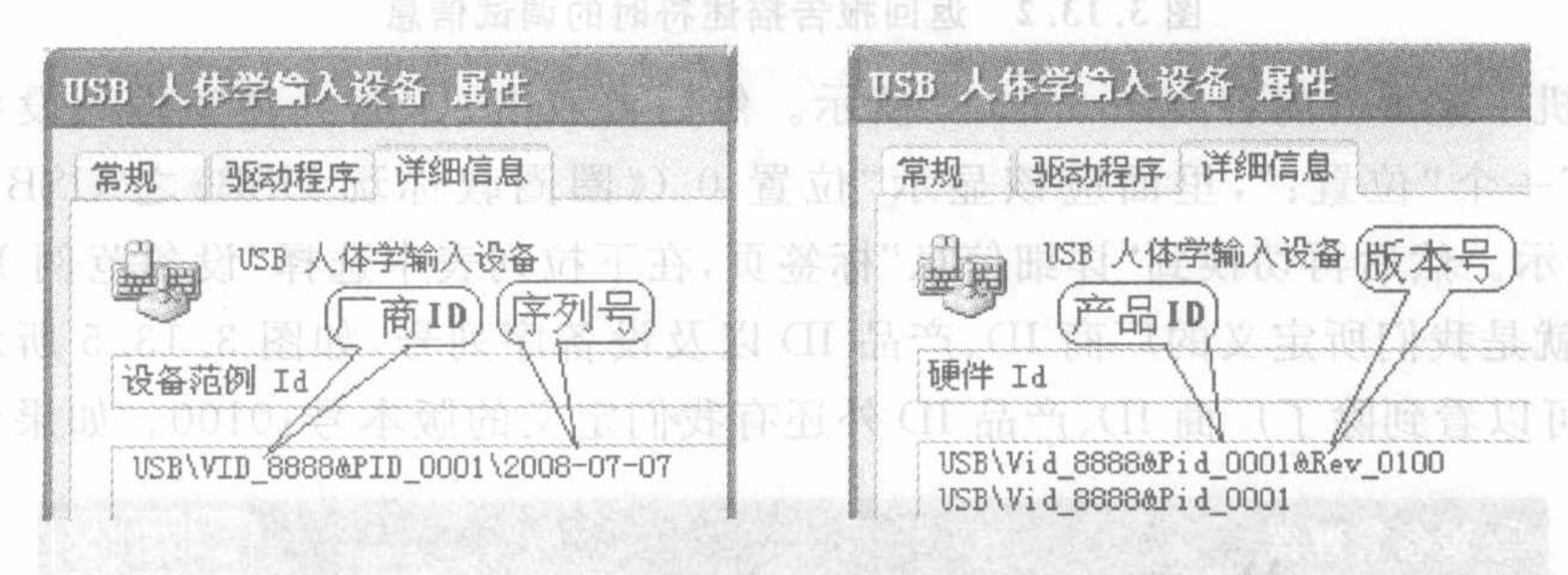


图 3.13.5 设备属性的详细信息

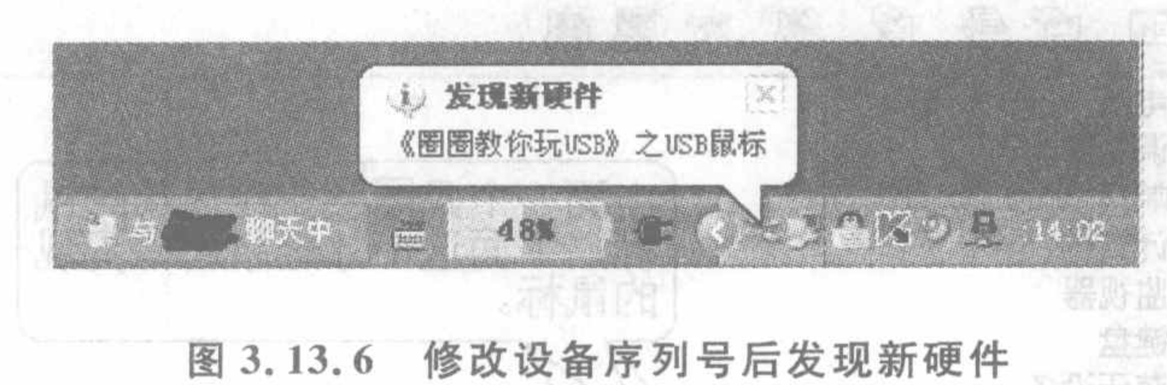


图 3.13.6 修改设备序列号后发现新硬件

3.14 报告的返回

通过前面报告描述符的定义知道要返回 4 字节的报告数据。但是报告也不是随时都能够返回的,只有在设置非 0 配置之后,才能将数据写到端点 1 中返回。那怎么知道已经进行设备配置了呢?可以增加一个变量 ConfigValue,初始化为 0,在收到设置配置请求后,将配置值赋给它,如果是非 0 的配置,就可以返回报告数据了。

发送到端点 1 去之前,需要检查端点 1 是否处于忙状态,即里面是否还有数据未发送出

去。为此，增加一个Ep1InIsBusy的标志，来决定端点1输入缓冲是否空闲。如果缓冲区空闲，则可以发送数据，将数据写入缓冲区后设置Ep1InIsBusy为真（即端点忙）。然后在端点1输入中断处理中（此时数据已经发送完毕），将Ep1InIsBusy设置为假（即端点已空闲）。在设备复位处理中，将Ep1InIsBusy设置为假，即设备复位后，端点1处于空闲状态。端点1输入中断的处理函数和总线复位处理的函数修改如下（记得清除端点1输入中断）：

```
/*****************************************************************
函数功能：端点1输入中断处理函数
入口参数：无
返　　回：无
备　　注：无
*****************************************************************/
void UsbEp1In(void)
{
#ifdef DEBUG0
 Prints("USB端点1输入中断。\r\n");
#endif
 //读最后发送状态，这将清除端点1输入的中断标志位
 D12ReadEndpointLastStatus(3);
 //端点1输入处于空闲状态
 Ep1InIsBusy = 0;
}
//////////////////////End of function//////////////////////////
/*****************************************************************
函数功能：总线复位中断处理函数
入口参数：无
返　　回：无
备　　注：无
*****************************************************************/
void UsbBusReset(void)
{
#ifdef DEBUG0
 Prints("USB总线复位。\r\n");
#endif
 Ep1InIsBusy = 0;  //复位后端点1输入缓冲区空闲
}
//////////////////////End of function//////////////////////////
```

然后，在main函数的主循环中（在判断是否有中断发生后），增加判断是否返回报告的代

码，如下所示：

```
    if(ConfigValue! = 0)                    //如果已经设置为非0的配置，则可以返回报告数据
     {
    LEDs = ~KeyPress;                       //利用板上8个LED显示按键状态，按下时亮
    if(!Ep1InIsBusy)                        //如果端点1输入没有处于忙状态，则可以发送数据
      {
       KeyCanChange = 0;                    //禁止按键扫描
       if(KeyUp||KeyDown||KeyPress)         //如果有按键事件发生，则返回报告
       {
        SendReport();
       }
       KeyCanChange = 1;                    //允许按键扫描
      }
    }
```

其中，SendReport()函数根据当前的按键情况来返回报告。在写这个函数之前，先约定好8个按键的使用：KEY1为光标左移，KEY2为光标右移，KEY3为光标上移，KEY4为光标下移，KEY5为滚轮下滚，KEY6为滚轮上滚，KEY7为鼠标左键，KEY8为鼠标右键。光标左移时，X轴为负；光标右移时，X轴为正；光标下移时，Y轴为正；光标上移时，Y轴为负；滚轮下滚时，为负；滚轮上滚时，为正；左键或者右键按下时，对应的位为1。在该函数中，先判断是否需要返回报告，只有当光标或者滚轮滚动(K1～K6按住)或者左、右键(KEY7、KEY8)状态发生变化时，才需要返回报告。对于移动和滚动，每次返回为1个单位，如果觉得移动速度慢，可以增大该值，以提高灵敏度。使用完KeyUp和KeyDown之后，记得对它们清零。最终的发送报告的函数代码如下：

```
/*********************************************************************
函数功能：根据按键情况返回报告的函数
入口参数：无
返    回：无
备    注：无
*********************************************************************/
void SendReport(void)
{
 //需要返回的4字节报告的缓冲
 //Buf[0]的D0就是左键，D1就是右键，D2就是中键(这里没有)
 //Buf[1]为X轴，Buf[2]为Y轴，Buf[3]为滚轮
 uint8 Buf[4] = {0,0,0,0};

 //我们不需要KEY1～KEY6按键改变的信息，所以先将它们清零
```

```
KeyUp &= ~(KEY1|KEY2|KEY3|KEY4|KEY5|KEY6);
KeyDown &= ~(KEY1|KEY2|KEY3|KEY4|KEY5|KEY6);
//如果有按键按住,并且不是 KEY7、KEY8(左、右键)
//或者 KEY7、KEY8 任何一个键有变动,则需要返回报告
if((KeyPress&(~(KEY7|KEY8)))||KeyUp||KeyDown)
{
 if(KeyPress & KEY1)            //如果 KEY1 按住,则光标需要左移,即 X 轴为负值
 {
  Buf[1] = -1;                  //这里一次往左移动一个单位
 }
 if(KeyPress & KEY2)            //如果 KEY2 按住,则光标需要右移,即 X 轴为正值
 {
  Buf[1] = 1;                   //这里一次往右移动一个单位
 }
 if(KeyPress & KEY3)            //如果 KEY3 按住,则光标需要上移,即 Y 轴为负值
 {
  Buf[2] = -1;                  //这里一次往上移动一个单位
 }
 if(KeyPress & KEY4)            //如果 KEY4 按住,则光标需要下移,即 Y 轴为正值
 {
  Buf[2] = 1;                   //这里一次往下移动一个单位
 }
 if(KeyPress & KEY5)            //如果 KEY5 按住,则滚轮下滚,即滚轮值为负
 {
  Buf[3] = -1;                  //这里一次往下滚动一个单位
 }
 if(KeyPress & KEY6)            //如果 KEY6 按住,则滚轮上滚,既滚轮值为正
 {
  Buf[3] = 1;                   //这里一次往上滚动一个单位
 }
 if(KeyPress & KEY7)            //鼠标左键
 {
  Buf[0]| = 0x01;               //D0 为鼠标左键
 }
 if(KeyPress & KEY8)            //鼠标右键
 {
  Buf[0]| = 0x02;               //D1 为鼠标右键
 }
//报告准备好了,通过端点 1 返回,长度为 4 字节
```

```
    D12WriteEndpointBuffer(3,4,Buf);
    Ep1InIsBusy = 1;                    //设置端点忙标志
  }
  //记得清除 KeyUp 和 KeyDown
  KeyUp = 0;
  KeyDown = 0;
}
//////////////////////End of function////////////////////////////
```

至此，这个 USB 实例就算完成了！但是，这只是一个实现鼠标基本功能的例子而已，离实际的产品还有一段距离。例如，位移传感器的数据采集，还有其他的一些标准请求、类请求等，都还没有相关的代码进行处理。读者可以自己动手将程序进一步完善，例如，增加请求中接收者的判断，增加对其他请求的响应等。对于其他的请求，很少用到，所以不管是输入请求还是输出请求，只要简单地返回 0 长度的数据包应该就可以了。附录是本实例完成后通过串口调试助手返回的全部调试信息和分别按下 KEY1～KEY8 后的数据信息。当程序调试完毕，不再需要这些调试信息时，可以将 config. h 文件中的 DEBUG0 和 DEBUG1 删除掉，这样就不会通过串口发送调试信息了。正是由于发送调试信息的原因，使得鼠标移动速度较慢，去掉调式信息后，鼠标移动速度会快很多。

3.15 Bus Hound 工具的简介

在 USB 的设计、调试中，监听数据传输十分重要。在前面的调试中，就是使用串口来显示接收和发送的数据，但有时仅靠这样的调试手段还是不够的，如果是读写端点程序本来就有问题，这时显示的信息就不可靠了。另外，有时还可以通过分析一个实际的 USB 设备的数据来帮助自己的产品设计，例如描述符的构造、数据传输等。

监听 USB 数据可以使用 USB 总线分析仪，但是这样的设备比较昂贵，大部分自学 USB 的朋友都没有这样的设备。那么有没有一些软件，安装在计算机上就可以捕捉 USB 口上传输的数据呢？有的，Bus Hound 就是这样一款能够监听总线数据的软件。不过，它只能看到传输成功的数据包，对于令牌包和应答包是看不到的，传输出错的数据包（例如设备 NAK 应答或者无应答）也是看不到的。另外，对于 USB 总线，在 Windows XP 环境下，只能看到设置地址后的数据包，而在 Windows 2000 下，设置地址前的数据也能看到。实际上，Bus Hound 使用的是过滤驱动程序的方法，在设备的驱动程序中插入一个过滤驱动，从而捕捉设备的数据。虽然 Bus Hound 在使用上比起 USB 总线分析仪差了很多，但也足以满足很多应用场合了。另外，它除了能够捕捉 USB 总线的数据之外，还能捕捉如鼠标、键盘、硬盘、串口等众多设备的数据。

下面以 Bus Hound 5.0 为例介绍它的使用方法。该软件可以去网上下载，然后直接安装

就可以了。安装后记得要重新启动计算机才有效。

启动 Bus Hound 5.0 后，最上面一行是 6 个大按钮，可以选择不同的界面，各个按键的功能如图 3.15.1 所示。

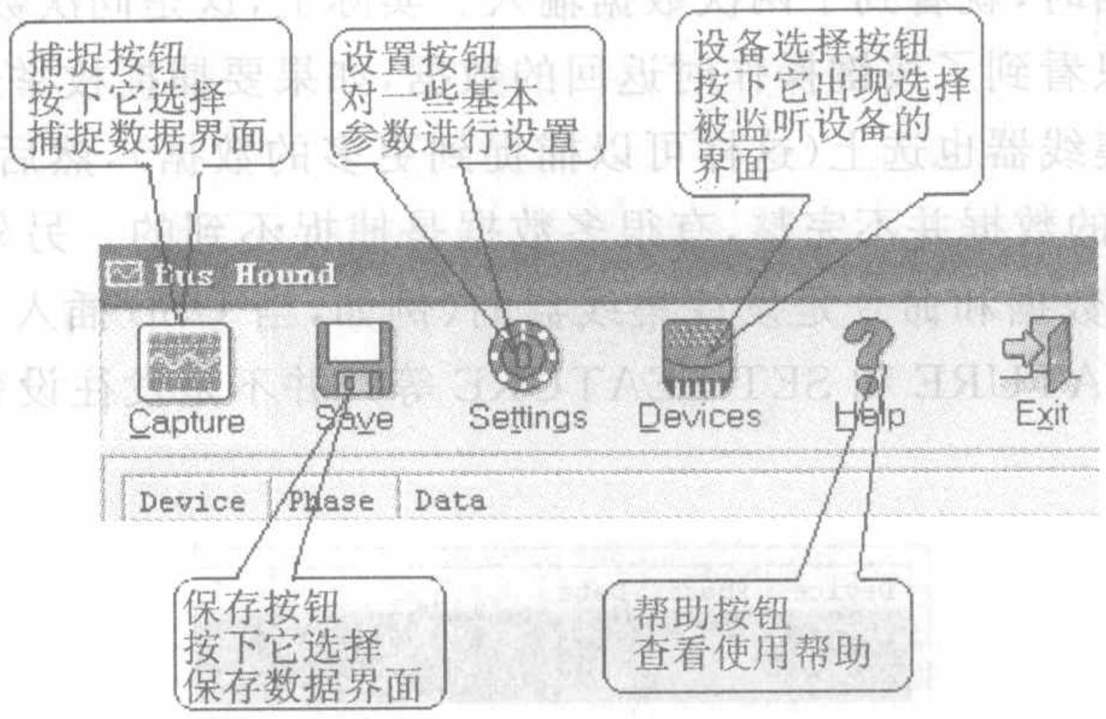


图 3.15.1 Bus Hound 5.0 的界面选择按钮

在开始捕捉数据之前，首先要对一些基本参数进行设置。单击 Settings 按钮弹出相应对话框，然后在 Limits 分组中将 Buffer Size 设置为 5000(即 5 MB)，将 Max Phase 设置为 10 240(即 10 KB)。Buffer Size 是 Bus Hound 捕捉数据用的缓冲区；Max Phase 为 Bus Hound 每个阶段能够显示数据的最大字节数。当实际传输的字节数比该值大时，数据显示被截断。下面的 Stop When 是停止监听的条件，可以根据自己的需要选择条件，或者在下面填入匹配的字符串或者数据，当数据匹配时自动停止监听。右边上面的蓝色框中选择需要监听的数据类型，对于 USB，通常选择 CTL、DI、DO、USTS 就够了。右边下面的蓝色框中选择需要显示哪些列，根据自己的需要选择即可。

然后再单击 Devices 按钮，在在弹出的界面中选择需要被监听数据的设备。这里显示的设备是一层层的关系，如图 3.15.2 所示，就是我们的 USB 鼠标实例所在的位置。将“USB 人体学输入设备”和“HID-compliant mouse”勾选上，以便对它们的数据进行观察。

图 3.15.2 设备选择中显示的 USB HID 鼠标

设备选好之后，再单击 Capture 按钮，切换到捕捉数据的界面，单击界面上的 Run 按钮，

开始捕捉数据。按一下KEY1，可以看到两次数据输入(DI)，如图3.15.3所示。这里为什么有两次数据输入呢？因为我们选择了两个设备，数据返回时，首先到达"USB人体学输入设备"，由该设备驱动将数据包装后(在前面加上一个报告ID 0)再发送到HID-compliant mouse设备。因此在查看数据时，就看到了两次数据输入。实际上，这是同次数据在不同层次的驱动传输而已。这里我们只看到了按键操作时返回的数据，如果要捕捉枚举过程的数据，可以将设备上面的主控制器和集线器也选上(这样可以捕捉到更多的数据)，然后重新连接设备。需要注意的是，这里捕捉到的数据并不完整，有很多数据是捕捉不到的。另外，当选择了集线器和主控制器之后，有一些数据和命令是发往集线器的(例如，当USB插入和拔下时显示的GET STATUS、CLEAR FEATURE和SET FEATURE等)，并不是发往设备，设备不会接收到这些请求。

Device	Phase	Data
27.1	DI	00 ff 00 00
28	DI	00 00 ff 00 00

图3.15.3 按下KEY1后返回的数据

3.16 本章小结

本章从建立工程开始，一步步介绍了USB鼠标实例的实现，对D12芯片的命令使用、USB标准请求的格式、USB各种描述符以及描述符的返回等作了详细介绍。本鼠标是在Windows XP环境下开发的，也许不同的操作系统在枚举时发送的请求顺序不一样，或者增加、减少了某些请求，但只要按照文中的开发思路来操做，就一定可以把USB鼠标做出来。

为了让读者能够容易接受，该实例只要求实现基本功能，对一些模型尽量简单化(还有很多请求并没有去实现，如Get_Report等)。如果需要实现更完整、更详细的功能，还要靠读者自己去研读D12数据手册、USB协议、USB HID协议、USB用途表等文档。本文的作用主要是让大家对USB的通信模型有个整体上的感性认识，这对学习整个USB协议是很有帮助的，否则就是盲人摸象。

第4章 USB 键盘的实现

本章通过对第 3 章中 USB 鼠标实例的修改，来完成 USB 键盘的实现。USB 键盘与 USB 鼠标的结构非常类似，都是 HID 设备，只需要修改各种描述符以及中断端点的数据处理即可。另外，还介绍了带鼠标的 USB 键盘和多媒体 USB 键盘的实现。

4.1 USB 键盘工程的建立

参照第 3 章中 USB 鼠标工程的建立方法，将 USB 鼠标文件夹复制一份，修改为 UsbKeyboard。

4.2 设备描述符的实现

USB 键盘跟 USB 鼠标的设备描述符几乎是一样的，只需要修改产品 ID 号(iProduct 字段)即可。之所以要修改产品 ID 号，是为了让操作系统能够跟前面的 USB 鼠标区别开来。设备描述符其他部分为什么不用修改，相信读者心中已经有了答案，如果还不是很清楚，那么请返回到第 3 章的设备描述符部分。下面是 USB 键盘设备描述符的定义代码。

```
//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。我们不在设备描述符中定义设备类,而是在接口描述符中定义设备类,所以
//该字段的值为 0
```

```
 0x00,
//bDeviceSubClass 字段。bDeviceClass 字段为 0 时,该字段也为 0
 0x00,
//bDeviceProtocol 字段。bDeviceClass 字段为 0 时,该字段也为 0
 0x00,
//bMaxPacketSize0 字段。D12 的端点 0 大小为 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,必须向
//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,由于是第二个实验,这里取 0x0002
//注意小端模式,低字节在先
 0x02,
 0x00,
//bcdDevice 字段。这个 USB 键盘刚开始做,就叫它 1.0 版吧! 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 开始吧
 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//////////////////////////////
```

4.3　配置描述符集合的实现

配置描述符集合包括配置描述符、接口描述符、HID 描述符和端点描述符。

4.3.1　配置描述符

USB 键盘的配置描述符与 USB 鼠标的配置描述符除了长度不一样之外,其他完全一样;

但是因为长度是用 sizeof 自动测量出来的，所以整个配置描述符的代码都不用修改。

4.3.2 接口描述符

本 USB 键盘设置了两个中断端点：一个中断输入端点和一个中断输出端点。中断输入端点用来返回报告，例如，什么键按下了；中断输出端点用来设置 LED 的状态，例如，大写字母锁定 LED，数字小键盘锁定 LED 等。在接口描述符中，bNumEndpoints 字段用来定义非 0 端点的数量，所以 bNumEndpoints 要修改成 2。USB 键盘所使用的协议是键盘，所以 bInterface-Protocol字段要修改为 1，表示使用键盘协议（如果不支持引导时使用，那么子类和协议都可以设置为 0）。

4.3.3 HID 描述符

HID 描述符与 USB 鼠标的 HID 描述符除了下级报告描述符长度不一样之外，其他都一样。但因为下级报告描述符的长度是用 sizeof 自动测量出来的，所以也不用修改。

4.3.4 端点描述符

对于输入端点的描述符不用修改，最后再增加一个输出端点的描述符。新增加的输出端点的属性与输入端点的属性是一样的，只是端点地址（bEndpointAddress 字段）不一样。这里使用端点 1 输出，输出端点的 D7 位为 0，所以 bEndpointAddress 字段为 0x01。

最终修改好的配置描述符集合代码如下：

```
//USB 配置描述符集合的定义，配置描述符总长度为(9+9+9+7+7)字节
code uint8 ConfigurationDescriptor[9+9+9+7+7]=
{
 /*****************配置描述符*************************/
 //bLength 字段。配置描述符的长度为 9 字节
0x09,
//bDescriptorType 字段。配置描述符编号为 0x02
0x02,
//wTotalLength 字段。配置描述符集合的总长度，包括配置描述符、接口描述符、类描述符和端点
//描述符等
sizeof(ConfigurationDescriptor)&0xFF,                    //低字节
(sizeof(ConfigurationDescriptor)>>8)&0xFF,               //高字节
//bNumInterfaces 字段。该配置包含的接口数，只有一个接口
0x01,
//bConfiguration 字段。该配置的值为 1
0x01,
//iConfigurationz 字段。该配置的字符串索引。这里没有，为 0
```

```
0x00,
//bmAttributes 字段。该设备的属性
//由于板子是总线供电的,并且我们不想实现远程唤醒的功能,所以该字段的值为 0x80
0x80,
//bMaxPower 字段。该设备需要的最大电流量。由于板子需要的电流不到 100 mA,所以这里设置为 100 mA
//由于每单位电流为 2 mA,所以这里设置为 50(0x32)
0x32,
/*******************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
//本 USB 键盘需要 2 个中断端点(一个输入一个输出),因此该值为 2
0x02,
//bInterfaceClass 字段。该接口所使用的类。USB 键盘是 HID 类,HID 类的编码为 0x03
0x03,
//bInterfaceSubClass 字段。该接口所使用的子类
//在 HID1.1 协议中,只规定了一种子类:支持 BIOS 引导启动的子类
//USB 键盘、鼠标属于该子类,子类代码为 0x01
0x01,
//bInterfaceProtocol 字段。如果子类为支持引导启动的子类,则协议可选择鼠标和键盘
//键盘代码为 0x01,鼠标代码为 0x02
0x01,
//iConfiguration 字段。该接口的字符串索引值。这里没有,为 0
0x00,
/*******************HID 描述符*********************/
//bLength 字段。本 HID 描述符下只有一个下级描述符,所以长度为 9 字节
0x09,
//bDescriptorType 字段。HID 描述符的编号为 0x21
0x21,
//bcdHID 字段。本协议使用的 HID1.1 协议。注意低字节在先
0x10,
0x01,
```

```
//bCountyCode 字段。设备适用的国家代码,这里选择美国,代码 0x21
0x21,

//bNumDescriptors 字段。下级描述符的数目。我们只有一个报告描述符
0x01,

//bDescritporType 字段。下级描述符的类型,为报告描述符,编号为 0x22
0x22,

//bDescriptorLength 字段。下级描述符的长度。下级描述符为报告描述符
sizeof(ReportDescriptor)&0xFF,
(sizeof(ReportDescriptor)>>8)&0xFF,

/**********************输入端点描述符**********************/
//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。我们使用 D12 的输入端点 1
//D7 位表示数据方向,输入端点 D7 为 1,所以输入端点 1 的地址为 0x81
0x81,

//bmAttributes 字段。D1 和 D0 为端点传输类型选择
//该端点为中断端点。中断端点的编号为 3。其他位保留为 0
0x03,

//wMaxPacketSize 字段。该端点的最大包长。端点 1 的最大包长为 16 字节,注意低字节在先
0x10,
0x00,

//bInterval 字段。端点查询的时间,设置为 10 个帧时间,即 10 ms
0x0A,

/**********************输出端点描述符**********************/
//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。我们使用 D12 的输出端点 1
//D7 位表示数据方向,输出端点 D7 为 0,所以输出端点 1 的地址为 0x01
0x01,

//bmAttributes 字段。D1 和 D0 为端点传输类型选择
//该端点为中断端点。中断端点的编号为 3。其他位保留为 0
```

```
0x03,
//wMaxPacketSize 字段。该端点的最大包长。端点 1 的最大包长为 16 字节
//注意低字节在先
0x10,
0x00,
//bInterval 字段。端点查询的时间,我们设置为 10 个帧时间,即 10 ms
0x0A
};
//////////////////////////配置描述符集合完毕//////////////////////////
```

4.4 字符串描述符

厂商字符串不用变,只需要修改产品字符串和产品序列号即可。将产品字符串改为"《圈圈教你玩 USB》之 USB 键盘",产品序列号改为"2008-07-12",然后用第 3 章介绍的字符串描述符工具直接产生字符串描述符。由于字符串描述符不怎么重要,这里就不再给出相关代码了,请参看光盘中的代码。

4.5 报告描述符

USB 键盘与 USB 鼠标描述符之间差别大一点的就是报告描述符了。键盘通常支持多个按键同时按下,所以输入报告中通常有一个数组,该数组可以同时返回多个按键值。而像 Shift、Ctrl、Alt 等特殊键,要用位图来表示(就像鼠标按键那样)。另外,键盘还有一些 LED,例如大写字母锁定灯、数字小键盘锁定灯等,因此键盘还需要一个输出报告来控制这些 LED。参考 HID 协议中的键盘报告描述的例子,我们的报告描述符实现代码如下:

```
//USB 报告描述符的定义
code uint8 ReportDescriptor[] =
{
 //每行开始的第一字节为该条目的前缀,前缀的格式为:
 //D7～D4:bTag。D3～D2:bType;D1～D0:bSize。以下分别对每个条目注释

 //这是一个全局(bType 为 1)条目,将用途页选择为普通桌面页
 //后面跟一字节数据(bSize 为 1),后面的字节数就不注释了,自己根据 bSize 来判断
 0x05, 0x01,            // USAGE_PAGE (Generic Desktop)

 //这是一个局部(bType 为 2)条目,说明接下来的集合用途用于键盘
 0x09, 0x06,            // USAGE (Keyboard)
```

```
//这是一个主条目(bType为0)条目,开集合,后面跟的数据0x01表示该集合是一个应用集合
//它的性质在前面由用途页和用途定义为普通桌面用的键盘
0xa1, 0x01,             // COLLECTION (Application)

//这是一个全局条目,选择用途页为键盘(Keyboard/Keypad(0x07))
0x05, 0x07,             // USAGE_PAGE (Keyboard/Keypad)

//这是一个局部条目,说明用途的最小值为0xe0。实际上是键盘左Ctrl键
//具体的用途值可在HID用途表中查看
0x19, 0xe0,             // USAGE_MINIMUM (Keyboard LeftControl)

//这是一个局部条目,说明用途的最小值为0xe7。实际上是键盘右GUI(WIN)键
0x29, 0xe7,             // USAGE_MAXIMUM (Keyboard Right GUI)

//这是一个全局条目,说明返回的数据的逻辑值(就是我们返回的数据域的值)最小为0
//因为这里用位来表示一个数据域,因此最小为0,最大为1
0x15, 0x00,             // LOGICAL_MINIMUM (0)

//这是一个全局条目,说明逻辑值最大为1
0x25, 0x01,             // LOGICAL_MAXIMUM (1)

//这是一个全局条目,说明数据域的数量为8个
0x95, 0x08,             // REPORT_COUNT (8)

//这是一个全局条目,说明每个数据域的长度为1个位
0x75, 0x01,             // REPORT_SIZE (1)

//这是一个主条目,说明有8个长度为1位的数据域(数量和长度由前面的两个全局条目所定义)用来
//作为输入,属性为:Data,Var,Abs
//Data表示这些数据可以变动;Var表示这些数据域是独立的,每个域表示一个意思;Abs表示绝对值
//这样定义的结果就是,当某个域的值为1时,就表示对应的键按下
//位0就对应着用途最小值0xe0,位7对应着用途最大值0xe7
0x81, 0x02,             // INPUT (Data,Var,Abs)

//这是一个全局条目,说明数据域数量为1个
0x95, 0x01,             // REPORT_COUNT (1)

//这是一个全局条目,说明每个数据域的长度为8位
0x75, 0x08,             // REPORT_SIZE (8)

//这是一个主条目,输入用,由前面两个全局条目可知,长度为8位,数量为1个
//它的属性为常量(即返回的数据一直是0),该字节是保留字节(保留给OEM使用)
0x81, 0x03,             // INPUT (Cnst,Var,Abs)

//这是一个全局条目。定义位域数量为6个
0x95, 0x06,             // REPORT_COUNT (6)

//这是一个全局条目。定义每个位域长度为8位
```

```
//其实这里这个条目不要也是可以的,因为在前面已经有一个定义长度为 8 位的全局条目了
0x75, 0x08,             // REPORT_SIZE (8)

//这是一个全局条目,定义逻辑最小值为 0
//这里这个全局条目也可以不要,因为前面已经有一个定义逻辑最小值为 0 的全局条目了
0x15, 0x00,             // LOGICAL_MINIMUM (0)

//这是一个全局条目,定义逻辑最大值为 255
0x25, 0xFF,             // LOGICAL_MAXIMUM (255)

//这是一个全局条目,选择用途页为键盘。前面已经选择过用途页为键盘了,所以该条目不要也可以
0x05, 0x07,             // USAGE_PAGE (Keyboard/Keypad)

//这是一个局部条目,定义用途最小值为 0(0 表示没有键按下)
0x19, 0x00,             // USAGE_MINIMUM (Reserved (no event indicated))

//这是一个局部条目,定义用途最大值为 0x65
0x29, 0x65,             // USAGE_MAXIMUM (Keyboard Application)

//这是一个主条目。它说明这 6 个 8 位的数据域是输入用的,属性为:Data,Ary,Abs
//Data 说明数据是可以变的;Ary 说明这些数据域是一个数组,即每个 8 位都可以表示某个键值,
//如果按下的键太多(例如超过这里定义的长度或者键盘本身无法扫描出按键情况时),
//则这些数据返回全 1(二进制),表示按键无效。Abs 表示这些值是绝对值
0x81, 0x00,             // INPUT (Data,Ary,Abs)

//以下为输出报告的描述
//逻辑最小值前面已经有定义为 0 了,这里可以省略
//这是一个全局条目,说明逻辑值最大为 1
0x25, 0x01,             // LOGICAL_MAXIMUM (1)

//这是一个全局条目,说明数据域数量为 5 个
0x95, 0x05,             // REPORT_COUNT (5)

//这是一个全局条目,说明数据域的长度为 1 位
0x75, 0x01,             // REPORT_SIZE (1)

//这是一个全局条目,说明使用的用途页为指示灯(LED)
0x05, 0x08,             // USAGE_PAGE (LEDs)

//这是一个局部条目,说明用途最小值为数字键盘灯
0x19, 0x01,             // USAGE_MINIMUM (Num Lock)

//这是一个局部条目,说明用途最大值为 Kana 灯
0x29, 0x05,             // USAGE_MAXIMUM (Kana)

//这是一个主条目。定义输出数据,即前面定义的 5 个 LED
0x91, 0x02,             // OUTPUT (Data,Var,Abs)

//这是一个全局条目。定义位域数量为 1 个
```

```
0x95, 0x01,             // REPORT_COUNT (1)
//这是一个全局条目。定义位域长度为3位
0x75, 0x03,             // REPORT_SIZE (3)
//这是一个主条目,定义输出常量,前面用了5位,所以这里需要3个位来凑成1字节
0x91, 0x03,             // OUTPUT (Cnst,Var,Abs)
//下面这个主条目用来关闭前面的集合。bSize为0,所以后面没数据
0xc0                    // END_COLLECTION
};
//////////////////////////报告描述符完毕//////////////////////////
```

通过上面的报告描述符的定义,我们知道返回的输入报告具有8字节。第一字节的8个位用来表示特殊键是否按下(例如Shift、Alt等键)。第二字节为保留值,值为常量0。第三到第八字节是一个普通键键值的数组,当没有键按下时,全部6字节值都为0。当只有一个普通键按下时,这6字节中的第一字节值即为该按键的键值(具体的键值请看HID的用途表文档),当有多个普通键同时按下时,则同时返回这些键的键值。如果按下的键太多,则这6字节都为0xFF(不能返回0x00,这样会让操作系统认为所有键都已经释放)。至于键值在数组中的先后顺序是无所谓的,操作系统会负责检查是否有新键按下。我们应该在中断端点1中按照前面的格式返回实际的键盘数据。另外,报告中还定义了一字节的输出报告,是用来控制LED情况的。只使用了低7位,高1位是保留值0。当某位的值为1时,则表示对应的LED要点亮。操作系统会负责同步各个键盘之间的LED,例如你有两块键盘,一块的数字键盘灯亮时,另一块也会跟着亮。键盘本身不需要判断各种LED应该何时亮,它只是等待主机发送报告给它,然后根据报告值来点亮相应的LED。我们在端点1输出中断中读出这1字节的输出报告,然后对它取反(因为学习板上的LED是低电平时亮),直接发送到LED上。这样main函数中按键点亮LED的代码就不需要了。

4.6 输入和输出报告的实现

由于这里的LED需要用作键盘指示灯,因此在main函数中用LED来显示按键按下的代码需要删除。另外,在判断按键事件时,只有当按键情况发生改变时,才需要返回报告,按键一直按住时不需要报告(在USB鼠标实例中,按住方向键不放也返回报告,是为了产生按住移动这样一种效果。在实际的鼠标中,也是只有状态发生改变时才返回报告的)。因此在判断按键事件的代码中,将最后的KeyPress条件去掉。判断是否需要发送报告的代码修改后如下:

```
if(ConfigValue!=0)              //如果已经设置为非0的配置,则可以返回报告数据
{
 if(!Ep1InIsBusy)               //如果端点1输入没有处于忙状态,则可以发送数据
```

```
    {
     KeyCanChange = 0;                    //禁止按键扫描
     if(KeyUp||KeyDown)                   //如果有按键事件发生
     {
      SendReport();                       //则返回报告
     }
     KeyCanChange = 1;                    //允许按键扫描
    }
  }
```

然后再修改 SendReport()函数，该函数负责发送具体的报告。由于学习板上的按键有限，因此这里只选择几个按键来实现。KEY1～KEY3 用作特殊键，分别为左 Ctrl、左 Shift、左 Alt 键。KEY4～KEY6 键为数字小键盘上的 1～3 键，当数字小键盘功能未打开时，就分别对应着 End、↓、Page Down 这三个键。KEY7 为 Caps Lock 键，KEY8 为 Num Lock 键。对于特殊键按下时，在第一字节中将相应的位设置为 1 即可；对于普通键，在后面的 6 字节中填入键值即可。具体的键值可以在 HID 用途表文档中查找。修改 SendReport()函数后的代码如下：


```
/*********************************************************************
函数功能:根据按键情况返回报告的函数
入口参数:无
返　　回:无
备　　注:无
*********************************************************************/
void SendReport(void)
{
 //需要返回的 8 字节报告的缓冲
 //通过报告描述符的定义及 HID 用途表文档可知,Buf[0]的 D0 是左 Ctrl 键,D1 是左 Shift 键,
 //D2 是左 Alt 键,D3 是左 GUI(即 Window 键),D4 是右 Ctrl,D5 是右 Shift,D6 是右 Alt,D7 是右 GUI 键
 //Buf[1]保留,值为 0。Buf[2]～Buf[7]为键值,最多可以有 6 个
 //由于这里普通键最多只有 5 个,因此不会超过 6 个。对于实际的键盘,如果按键数太多时,
 //后面的 6 个字节都为 0xFF,表示按下的键太多,无法正确返回
 uint8 Buf[8] = {0,0,0,0,0,0,0,0};
 //由于需要返回多个按键,所以需要增加一个变量来保存当前的位置
 uint8 i = 2;
 //根据不同的按键设置输入报告
 if(KeyPress & KEY1)                     //如果 KEY1 按住
 {
  Buf[0]| = 0x01;                        //KEY1 为左 Ctrl 键
```

```
}
if(KeyPress & KEY2)                    //如果 KEY2 按住
{
 Buf[0]| = 0x02;                       //KEY2 为左 Shift 键
}
if(KeyPress & KEY3)                    //如果 KEY3 按住
{
 Buf[0]| = 0x04;                       //KEY3 为左 Alt 键
}
if(KeyPress & KEY4)                    //如果 KEY4 按住
{
 Buf[i] = 0x59;                        //KEY4 为数字小键盘 1 键
 i ++ ;                                //切换到下个位置
}
if(KeyPress & KEY5)                    //如果 KEY5 按住
{
 Buf[i] = 0x5A;                        //KEY5 数字小键盘 2 键
 i ++ ;                                //切换到下个位置
}
if(KeyPress & KEY6)                    //如果 KEY6 按住
{
 Buf[i] = 0x5B;                        //KEY6 为数字小键盘 3 键
 i ++ ;                                //切换到下个位置
}
if(KeyPress & KEY7)                    //如果 KEY7 按住
{
 Buf[i] = 0x39;                        //KEY7 为大/小写切换键
 i ++ ;                                //切换到下个位置
}
if(KeyPress & KEY8)                    //如果 KEY8 按住
{
 Buf[i] = 0x53;                        //KEY8 为数字小键盘功能切换键
}
//报告准备好了,通过端点 1 返回,长度为 8 字节
D12WriteEndpointBuffer(3,8,Buf);
Ep1InIsBusy = 1;                       //设置端点忙标志
//记得清除 KeyUp 和 KeyDown
KeyUp = 0;
KeyDown = 0;
```

```
}
//////////////////////End of function////////////////////////////
```

然后在端点 1 输出中断处理中，增加对输出报告的处理。输出报告只有 1 字节，表示当前 LED 的状态。输出报告的某位为 1 时，对应的 LED 亮。将其取反后，即可直接驱动 LED 显示。中断处理中记得清除中断和清除缓冲区，端点 1 输出中断的处理函数修改后代码如下：

```
/*********************************************************************
函数功能:端点 1 输出中断处理函数
入口参数:无
返　　回:无
备　　注:无
*********************************************************************/
void UsbEp1Out(void)
{
 uint8 Buf[1];              //用来保存 1 字节的输出报告,控制 LED
#ifdef DEBUG0
 Prints("USB 端点 1 输出中断。\r\n");
#endif
 //读端点最后状态,这将清除端点 1 输出的中断标志位
 D12ReadEndpointLastStatus(2);
 //从端点 1 输出缓冲读回 1 字节数据
 D12ReadEndpointBuffer(2,1,Buf);
 //清除端点缓冲区
 D12ClearBuffer();
 //输出报告 1 字节为 LED 状态,某位为 1 时,表示 LED 亮
 LEDs = ~Buf[0];
}
//////////////////////End of function////////////////////////////
```

4.7　USB 键盘实例的测试

将上述修改过的程序进行编译，下载到学习板中进行测试。通过串口调试助手显示的调试信息可以看到，该程序顺利地完成了枚举过程，并弹出发现新硬件的窗口，如图 4.7.1 所示。最后，主机通过端点 1 发出了一个设置 LED 状态的报告，具体的值与当前的键盘 LED 所处的状态有关。再次进入到设备管理器中可以看到多了一个“USB 人体学输入设备”和一个“HID Keyboard Device”，如图 4.7.2 所示。双击“USB 人体学输入设备”，可以看到它的属性，在位置一栏有显示我们设置的产品字符串“《圈圈教你玩 USB》之 USB 键盘”，如图 4.7.3 所示。

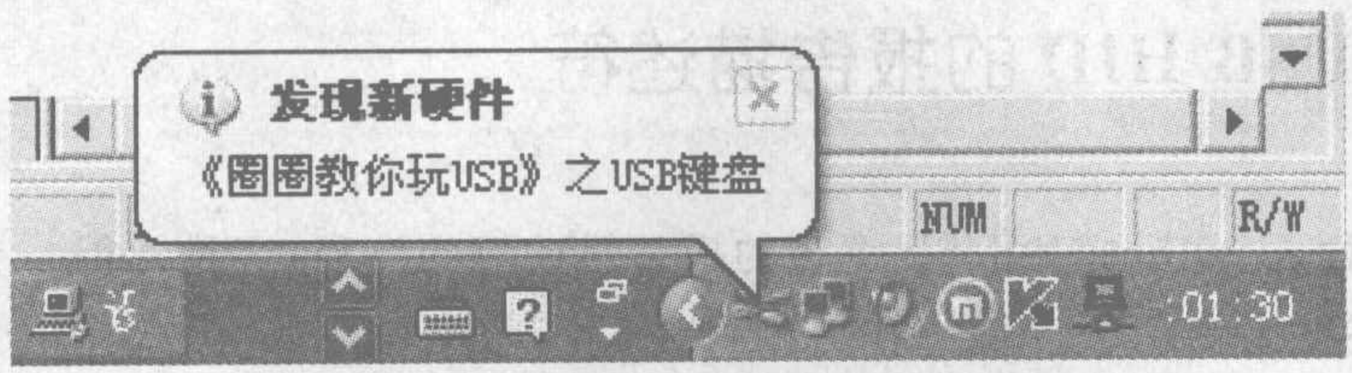


图 4.7.1　发现 USB 键盘新硬件

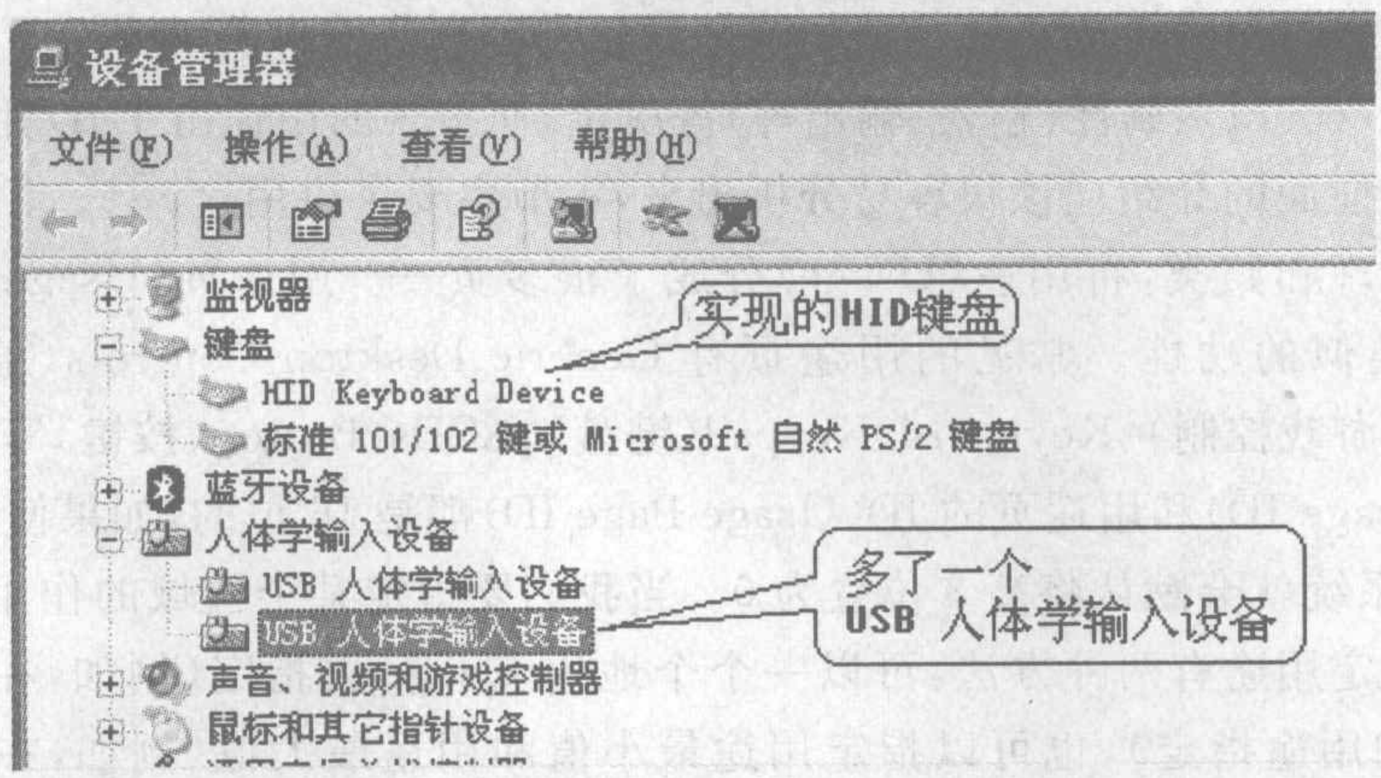


图 4.7.2　设备管理器中显示的 USB 键盘

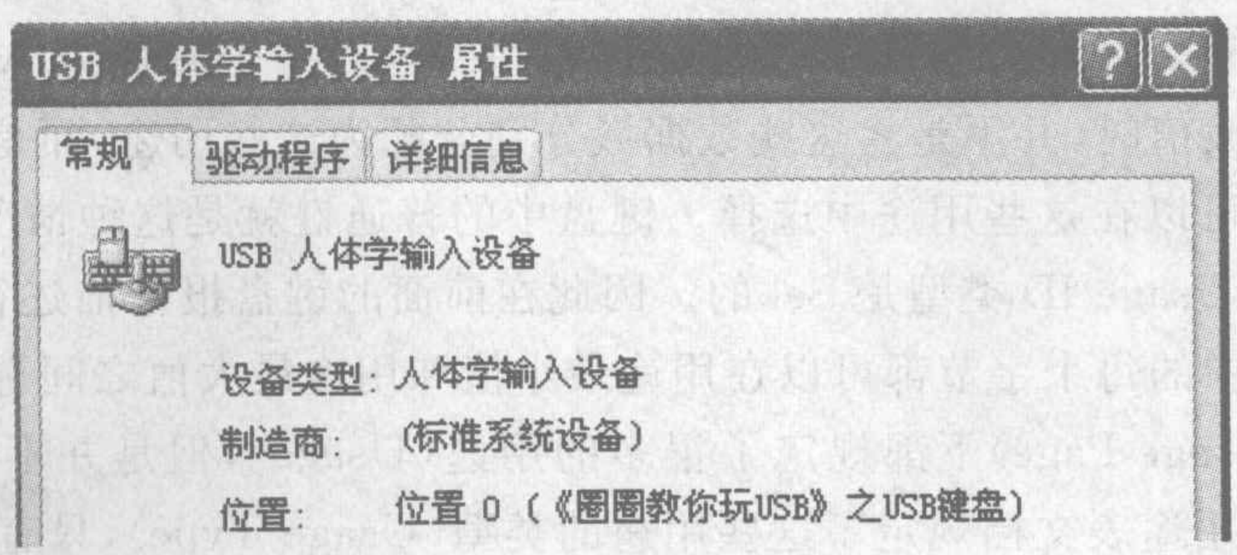


图 4.7.3　USB 键盘的人体学输入设备属性

然后打开一个记事本文件(当然,带编辑窗口的其他程序也行),就可以用我们设计的 USB 键盘输入了。打开数字小键盘(按 KEY8 键,打开时 LED1 会亮),然后按 KEY4～KEY6 可输入数字 1、2、3。按 KEY7 可切换大小写字母状态,大写字母状态时,LED2 会亮。同时按下 KEY1 和 KEY2 可切换输入法,总之,这些键跟平时用的键盘上的键没什么两样。

感觉怎样?简单吧。是的,实现一个 USB 键盘就是这么简单。因此,不要被 USB 庞大而复杂的协议给吓住了,要实现某个具体的设备还是不算太难的。

4.8 再谈 USB HID 的报告描述符

也许看了前面 USB 鼠标和 USB 键盘的报告描述符的注释，还是觉得一头雾水。的确，这个 HID 报告描述符比较复杂，牵涉到的内容比较多，单 HID 协议有 70 多页，而 HID 用途表又有 170 多页。在本书中，圈圈只能摘录其中的要点来说明，帮助读者能够在整体上有个认识，不然的话，就成文档翻译了。详细的内容还请读者自行参考相关文档。

一个报告描述符主要是为了描述报告的结构以及用途。报告的结构主要由报告的字段的长度(size)、数量(count)、属性(输入、输出等)等决定，而报告的用途由 HID 用途表文档规定。报告的结构通过前面的介绍应该很容易分析出来，本节主要讲讲用途。

为了方便管理和归类，将用途(Usage)分成了很多页——用途页(Usage Page)。每页中的用途具有相类似的功能。常见的用途页有 Generic Desktop Controls(通用桌面控制)、Game Controls(游戏控制)、Keyboard/Keypad(键盘)、LEDs、Button(按键)等。

用途 ID(Usage ID)和用途页的 ID(Usage Page ID)都是 16 位的，如果使用带 1 字节数据的短条目，那么系统就会默认将高 8 位置为 0。当我们要指定某个位域的作用时，要指定用途页以及用途。指定用途有两种方法：可以一个个地用 Usage 去指定(例如，在 USB 鼠标中对 X、Y 以及滚轮的用途指定)，也可以指定用途最小值和用途最大值(例如，在 USB 键盘中对 LED 的指定)。这样，每个数据域就会分配到对应的用途。但是这里有一个例外，那就是用途类型(Usage Type)为 Sel(Selectors)的用途 ID，它的数据域属性为 Ary(数组)。一个 Ary 可以有一个或者多个元素，每个元素都具有相同的用途。当对一个属性为 Ary 的数据域指定用途最小值和用途最大值时，并不是将这些数据域分别分配为这些用途，而是这些数据域的每个数组元素的返回值可以在这些用途中选择。键盘中的普通键就是这种情况，HID 用途表中规定了这些普通键的 Usage ID 类型是 Sel 的。因此在前面的键盘报告描述符中，定义了 6 字节的数组。这 6 字节中的每个字节都可以在用途最小值和用途最大值之间选择一个 ID 返回。

每个用途页(Usage Page)下都规定了很多的用途(Usage)，但是并不是每个用途都可以随意使用的。HID 用途表文档规定了这些用途的类型(Usage Type)，只有类型符合的地方才可以使用，例如不能用控制类的用途来开集合，也不能用开集合的用途来定义数据域。有三类基本的用途类型：集合(Collections)、控制(Controls)和数据(Data)。

集合类用途用来定义一个集合，例如本实例的报告描述符的开始，就选择了通用桌面用途页中的键盘用途来开集合。每个用途页中的用途 ID0 是保留值，而 0x01～0x1F(不一定全用)用来定义集合用途。集合用途有 CA(应用集合)、CL(逻辑集合)、CP(物理集合)等。物理集合通常包含轴信息，例如指针设备的 *X* 轴、*Y* 轴等。系统软件通过查找最外层的应用集合用途来决定该设备的类型，如果是系统控制设备(例如鼠标、键盘、操纵杆等)，那么就会加载相应的驱动来产生一个标准输入设备。我们常用的通用桌面用途页中，就定义了几种应用集合的

用途：Mouse(0x02)、Joystick(0x04)、Game Pad(0x05)、Keyboard(0x06)、Keypad(0x07)、Multi-axis Controller(0x08)。在前面的USB鼠标报告描述符中，就使用了Mouse(0x02)，而在本章的USB键盘报告描述符中，则使用了Keyboard(0x06)。一个带有指针设备的键盘，则可以说明两个应用集合，一个用于键盘，另一个用于鼠标。

控制用途类用来定义一个数据域作为控制用。控制用途类分为LC(Linear Control，线性控制，例如音量旋钮或者滑块)、OOC(On/Off Control，开/关控制，例如键盘上的LED)、MC(Momentary Control，例如鼠标的按键)、OSC(One Shot Control，单次射击控制，例如游戏手柄上的单发键)、RTC(Re-trigger Control，重触发控制，例如游戏手柄上的连发键)。

数据用途类用来定义一个数据域作为数据用，分为Sel(选择器)、SV(静态值)、SF(静态标志)、DV(动态值)、DF(动态标志)。前面已经讲过，Sel是供数据域选择的一种选项。数据用途类中比较常见的是DV，例如各种轴的变化值。

当我们构造一个HID报告描述符时，首先要决定它的主要功能是什么，例如键盘、鼠标或者操纵杆等。因为系统软件会通过最外层的应用集合用途来判断它是什么样的设备，所以需要利用这些信息来打开一个应用集合。确定功能后，再到HID用途表中去查找，看哪个用途页中具有所需要的用途，并且用途类型为CA(应用集合)。

以USB鼠标的报告描述符为例，需要一个用途为Mouse的最外层应用集合。通过查看HID用途表，知道在Generic Desktop Page (0x01)中有一个用途类型为CA的Mouse(0x02)用途。因此在USB鼠标报告描述符中，开头就先指定了使用用途页Generic Desktop Page (0x01)，然后又指定了该用途页下的Mouse(0x02)用途，接着使用开集合的主条目打开了一个应用集合。这样系统软件在分析这个报告描述符时，就知道最外层的一个应用集合是用作鼠标的，操作系统就会增加一个HID鼠标设备。然后在该应用集合中，增加了一个用途为Point的物理集合。Point也是Generic Desktop Page中的一个开物理集合用的用途，是指针设备集合。由于鼠标上有三个按键，因此需要用到按键(Button)的功能。在HID用途表中，专门有一个按键页：Button Page (0x09)，选择了按键用途1～3，分别为左键、右键、中键。另外，鼠标还有X轴、Y轴以及滚轮，这些用途是在Generic Desktop页中定义的，所以接着又切换到了Generic Desktop页，将三个宽度为8位的数据段分别定义为X轴、Y轴、滚轮。最后，使用关闭集合的主条目将两个集合关闭。这样一个USB鼠标的HID报告描述符就构造好了。

再以USB键盘为例，在开头就选择了Generic Desktop Page (0x01)，然后选择了Keyboard(0x06)用途，接着用开集合的主条目打开了一个用途为键盘的应用集合。对于键盘，有一个专门的Keyboard/Keypad Page (0x07)，所以接着就选择了Keyboard/Keypad这个用途页。在这个用途页中，大部分普通键的用途类型是Sel的，有些特殊键(例如Shift、Ctrl、Alt等)则是DF的。对于是DF类型的，需要用位来表示按键是否按下，所以第一字节就定义了8个长度为1位的数据域，分别用来表示8个特殊键。而对于是Sel类型的，需要用数组来表

示，数组中的每个元素可表示某个键是否按下。这里定义了一个6字节的数组，因而最多可以同时有6个普通键按下。键盘通常还有指示灯(LED)，用来指示键盘当前的状态，例如大写字母锁定、数字小键盘锁定等。指示灯有一个专门的用途页：LED Page (0x08)。通过选择LED页，定义了5个宽度为1位的输出数据域，每个位对应着一个LED，用途分别为Num Lock、Caps Lock、Scroll Lock、Compose和Kana。当某个功能使能时，对应的位被设置为1。由于协议规定报告必须是整数字节，所以后面增加了3位的填充。最后，使用关集合将集合关闭。这样一个USB键盘报告描述符就构造好了。

总之，根据实际的用途需要，去HID报告表文档中查找用途ID以及它所在的用途页，然后按照规则来构造报告描述符就可以了。

4.9 带鼠标功能的USB键盘(方法一)

圈圈在网上浏览时，有好几次看到有人问能不能在一个设备中同时实现USB鼠标和键盘的功能，以及如何实现。据圈圈所知，至少有两种方法能够实现：

① 只使用一个接口，但是使用两个应用集合和两个报告。

② 使用两个接口，一个接口实现键盘功能，另一个接口实现鼠标功能。

方法①只需要一个输入端点和一个输出端点就够了，而方法②还需要另外一个额外的非0端点，因为同一个配置下的不同接口，必须使用不同的端点。

本节就是使用方法①，通过修改报告描述符来实现。通过前面的分析，知道一个报告描述符中可以具有多个外层应用集合，而系统软件就是通过分析外层应用集合的功能来增加不同的设备和驱动的。因此在这里，只需要实现用途分别为键盘和鼠标的两个应用集合即可。同时，每个应用集合里还需要增加一个报告ID。以区分返回数据的作用。报告ID是报告输入或者输出时的第一个字节，当没有定义报告ID时，报告前面就没有报告ID。对于返回的没有报告ID的报告，人体学输入设备驱动会自动增加一个报告ID 0；而应用程序在发送数据出去时，也要带一个值为0的报告ID。人体学输入设备驱动会自动去掉这个值为0的报告ID，只发送数据出去。关于这点会在后面的自定义HID设备中会讲到。

注意：报告ID一旦定义，输入报告和输出报告的第一字节都是报告ID。例如本例中的键盘，从端点1读输出报告数据时要读2字节的数据，第一字节为报告ID，第二字节才是真正的报告数据。

上面是实现的基本思路，下面就按照上面的思路一步步来创建这个设备。首先将前面的USB键盘复制一份，改名为UsbKeyboardWithMouse。由于设备不一样了，所以应该在设备描述符中修改一下产品ID。这里是第三个实验，产品ID就用0x0003。产品字符串描述符改为“《圈圈教你玩USB》之带鼠标的USB键盘”，设备序列号字符串描述符改为“2008-07-15”。将第3章中的USB鼠标程序的报告描述符复制到本实例的报告描述符的最后，然后分

别在两个开集合的条目后增加一个报告ID的条目，报告ID分别为1和2(报告ID 0是保留的，不能使用)。修改后的报告描述符代码如下(由于跟前面的代码大部分重复，所以这中间省略了大部分重复的内容)：

```
//USB报告描述符的定义
//该报告描述符定义了两个顶层应用集合——键盘和鼠标
code uint8 ReportDescriptor[] =
{
/*********************USB键盘部分报告描述符*********************/
 //这是一个全局(bType为1)条目，用途页选择为普通桌面
 0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)

 //这是一个局部(bType为2)条目，说明接下来的集合用途用于键盘
 0x09, 0x06,                    // USAGE (Keyboard)

 //这是一个主条目(bType为0)，开集合，后面跟的数据0x01表示该集合是一个应用集合
 //它的性质在前面由用途页和用途定义为普通桌面用的键盘
 0xa1, 0x01,                    // COLLECTION (Application)

 //报告ID，这里定义键盘报告的ID为1(报告ID 0是保留的)
 0x85, 0x01,                    // Report ID (1)
 ⋮                              //此处省略部分键盘报告描述符

//下面这个主条目用来关闭前面的集合。键盘报告描述符应用集合在此关闭
 0xc0,                          // END_COLLECTION

/*********************USB鼠标部分报告描述符*********************/
 //这是一个全局(bType为1)条目，选择用途页为普通桌面Generic Desktop Page(0x01)
 0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)

 //这是一个局部(bType为2)条目，说明接下来的应用集合用途用于鼠标
 0x09, 0x02,                    // USAGE (Mouse)

 //这是一个主条目(bType为0)，开集合，后面跟的数据0x01表示该集合是一个应用集合
 //它的性质在前面由用途页和用途定义为普通桌面用的鼠标
 0xa1, 0x01,                    // COLLECTION (Application)

 //报告ID，这里定义鼠标报告的ID为2
 0x85, 0x02,                    //Report ID (2)
 ⋮                              //此处省略部分鼠标报告描述符

 //鼠标应用集合在此处关闭
 0xc0                           // END_COLLECTION
};
//////////////////////////报告描述符完毕//////////////////////////
```

接下来，就要修改返回报告的函数了。由于学习板上按键数量有限，因此这里决定使用KEY1来选择KEY2～KEY8的功能；当KEY1按住时，KEY2～KEY8用作键盘输入；当KEY1松开时，KEY2～KEY8用作鼠标输入。当用作键盘时，将按键情况填入到8个字节的报告中，前面还要增加1字节的报告ID，然后通过端点1将9字节的报告ID及报告返回。当用作鼠标时，根据按键情况设置4字节的报告，前面还要增加1字节的报告ID，然后通过端点1将5字节的报告ID及报告返回。由于最多要用到9字节的缓冲，因此在SendReport()函数中将Buf数组的长度改为9字节。填充报告数据时注意，实际的报告是从第二字节开始的。修改好的SendReport()函数代码如下：

```
/*****************************************************************
函数功能:根据按键情况返回报告的函数
入口参数:无
返      回:无
备      注:无
*****************************************************************/
void SendReport(void)
{
 //需要返回的9字节报告的缓冲(1字节报告ID加键盘8字节报告)
 //而鼠标报告只有4字节,加上1字节报告ID,总共5字节,9字节够了
 uint8 Buf[9]={0,0,0,0,0,0,0,0,0};
 //由于需要返回多个按键,所以需要增加一个变量来保存当前的位置
 //由于报告ID占用第一字节,所以普通按键从第四字节开始存放
 uint8 i=3;
 //我们用KEY1键来选择剩余的7个键是键盘功能还是鼠标功能
 //当KEY1按住时,剩余7个键为键盘功能,这7个键的功能跟键盘实例的一样
 //当KEY1松开时,剩余7个键为鼠标功能,功能分别为
 //KEY2:光标左移;KEY3:光标右移;KEY4:光标上移;KEY5:光标下移
 //KEY6:鼠标左键;KEY7:鼠标中键;KEY8:鼠标右键
 //根据不同的按键设置输入报告
 if(KeyPress & KEY1)              //如果KEY1按住,则为键盘功能
 {
  Buf[0]=0x01;                    //第一字节为报告ID,键盘报告ID为1
  if(KeyPress & KEY2)             //如果KEY2按住
  {
   Buf[1]|=0x02;                  //KEY2为左Shift键
  }
  if(KeyPress & KEY3)             //如果KEY3按住
```

```
{
 Buf[1]| = 0x04;                    //KEY3 为左 Alt 键
}
if(KeyPress & KEY4)                 //如果 KEY4 按住
{
 Buf[i] = 0x59;                     //KEY4 为数字小键盘 1 键
 i ++ ;                             //切换到下个位置
}
if(KeyPress & KEY5)                 //如果 KEY5 按住
{
 Buf[i] = 0x5A;                     //KEY5 数字小键盘 2 键
 i ++ ;                             //切换到下个位置
}
if(KeyPress & KEY6)                 //如果 KEY6 按住
{
 Buf[i] = 0x5B;                     //KEY6 为数字小键盘 3 键
 i ++ ;                             //切换到下个位置
}
if(KeyPress & KEY7)                 //如果 KEY7 按住
{
 Buf[i] = 0x39;                     //KEY7 为大/小写切换键
 i ++ ;                             //切换到下个位置
}
if(KeyPress & KEY8)                 //如果 KEY8 按住
{
 Buf[i] = 0x53;                     //KEY8 为数字小键盘功能切换键
}
 //报告准备好了,通过端点 1 返回,长度为 9 字节
 D12WriteEndpointBuffer(3,9,Buf);
}
else                                //KEY1 松开,为鼠标功能
{
 Buf[0] = 0x02;                     //第一字节为报告 ID,鼠标报告 ID 为 2
 if(KeyDown & KEY2)                 //如果 KEY2 按下
 {
  Buf[2] = - 10;                    //KEY2 为鼠标左移,按一次移动 10 个单位
 }
 if(KeyDown & KEY3)                 //如果 KEY3 按下
 {
```

```
 Buf[2] = 10;                    //KEY3 为鼠标右移,按一次移动 10 个单位
}
if(KeyDown & KEY4)               //如果 KEY4 按下
{
 Buf[3] = -10;                   //KEY4 为鼠标上移,按一次移动 10 个单位
}
if(KeyDown & KEY5)               //如果 KEY5 按下
{
 Buf[3] = 10;                    //KEY5 为鼠标下移,按一次移动 10 个单位
}
if(KeyPress & KEY6)              //如果 KEY6 按住
{
 Buf[1]| = 0x01;                 //KEY6 为鼠标左键
}
if(KeyPress & KEY7)              //如果 KEY7 按住
{
 Buf[1]| = 0x04;                 //KEY7 为鼠标中键
}
if(KeyPress & KEY8)              //如果 KEY8 按住
{
 Buf[1]| = 0x02;                 //KEY8 为鼠标右键
}
//报告准备好后通过端点 1 返回,长度为 5 字节
D12WriteEndpointBuffer(3,5,Buf);
}
Ep1InIsBusy = 1;                 //设置端点忙标志
//记得清除 KeyUp 和 KeyDown
KeyUp = 0;
KeyDown = 0;
}
//////////////////////////End of function////////////////////////////
```

还要记得修改输出报告的处理,因为输出报告前面也增加了 1 字节的报告 ID。这里只定义了一个输出报告,即键盘的 LED 灯。原来的 LED 状态在第一字节中,现在变成了在第二字节中,并且第一字节为报告 ID1。在端点 1 输出中断处理中,修改输出报告的处理函数如下:

```
/*****************************************************************
函数功能:端点 1 输出中断处理函数
入口参数:无
返      回:无
```

```
备      注:无
*************************************************************************/
void UsbEp1Out(void)
{
 uint8 Buf[2];                    //用来保存 2 字节的输出报告,控制 LED
#ifdef DEBUG0
 Prints("USB 端点 1 输出中断。\r\n");
#endif
 //读端点最后状态,这将清除端点 1 输出的中断标志位
 D12ReadEndpointLastStatus(2);
 //从端点 1 输出缓冲读回 2 字节数据
 D12ReadEndpointBuffer(2,2,Buf);
 //清除端点缓冲区
 D12ClearBuffer();
 //注意输出报告也增加了 1 字节的报告 ID
 //第 1 字节为报告 ID,第 2 字节为 LED 状态,某位为 1 时,表示 LED 亮
 //我们定义的键盘的报告 ID 为 1,所以这里判断报告 ID 是否为 1
 if(Buf[0] == 0x01)               //报告 ID 为 1,即键盘的输出报告
 {
  LEDs = ~Buf[1];
 }
}
//////////////////////End of function////////////////////////////
```

另外,为了便于使用者确认 KEY1 是否被按住,使用 LED8 作为指示。当 KEY1 按住时,LED8 点亮。这个处理过程放在 main 函数的判断按键情况之前。

将修改好的程序编译,并下载到学习板中去运行,会提示发现新硬件。等提示新硬件已经安装完毕并可以使用时,就可以按键测试了。不按住 KEY1 时,按动 KEY2~KEY5 应该能够使光标移动(每次移动为 10 个单位,这里没有实现按住不放连续移动的功能),KEY6 为鼠标左键,KEY7 为鼠标中键,KEY8 为鼠标右键,按下 KEY8 应该能弹出快捷菜单。然后按住 KEY1,这时 LED8 应该亮,表示进入键盘状态。按 KEY7 应该能在大/小写字母之间切换,按 KEY8 应该能在数字小键盘/编辑键之间切换。当处于数字小键盘功能时(LED1 亮),KEY4、KEY5、KEY6 应该分别能输入数字 1、2、3。注意操作时,应该在 KEY2~KEY8 都松开的状态下才按下或松开 KEY1,否则,就可能会出现按键一直按住的情况(例如鼠标左键一直按住、键盘一直按住等)。这是因为 KEY1 状态改变后,就切换到了另外一种功能,此时再松开其他键,发送的就不是刚刚那个功能的按键弹起事件。

再来看看设备管理器里,是不是多了一个键盘和一个鼠标设备,如图 4.9.1 所示。

可以打开 BUS Hound 的设备列表来看看这个鼠标和键盘的关系,如图 4.9.2 所示。从

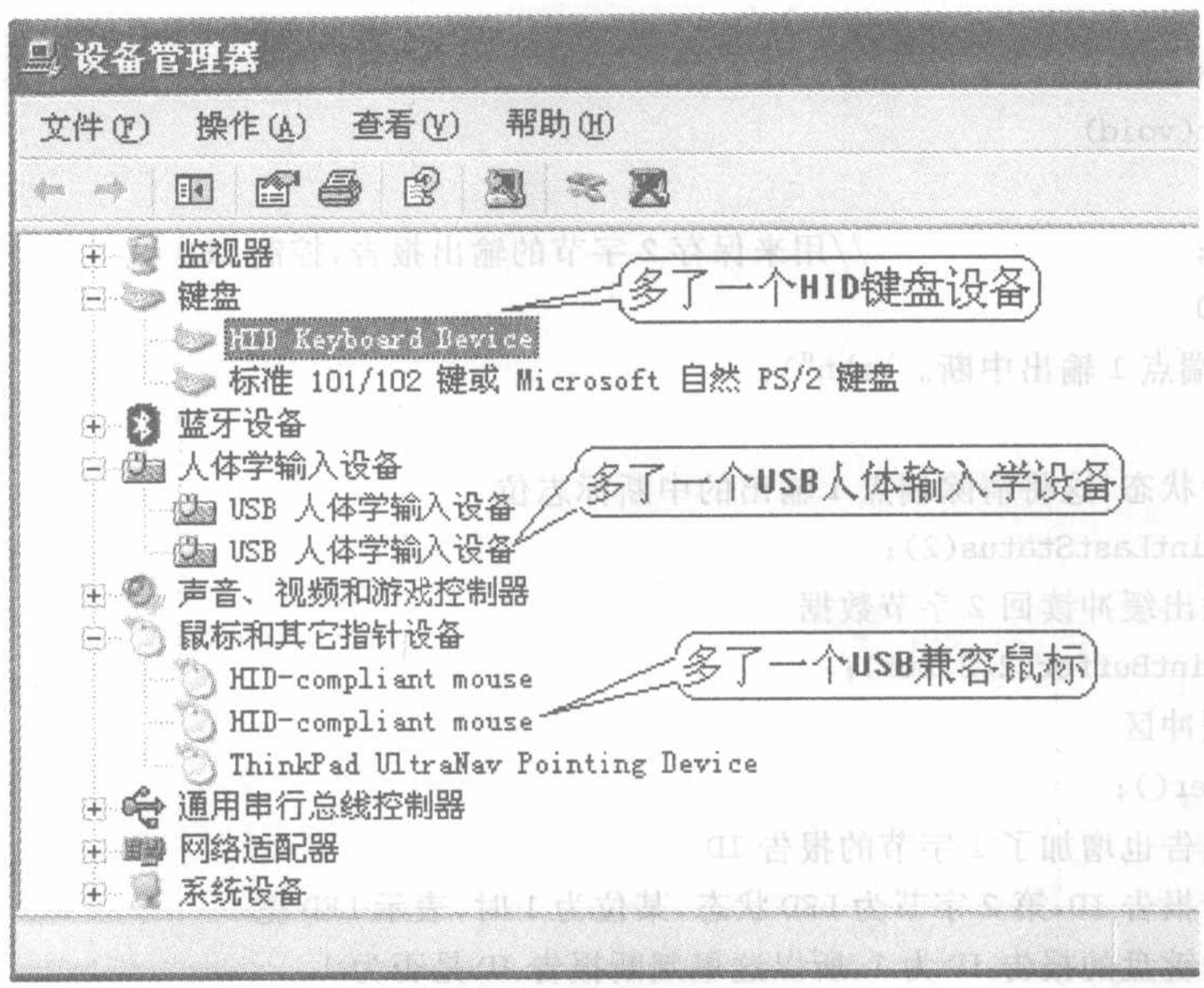


图 4.9.1 设备管理器

图 4.9.2 中可以看出，这里的键盘和鼠标都是由“USB 人体学输入设备”生成的。因为这里是通过不同的应用集合来实现的，但是这些集合都同属于一个 HID 设备。系统软件通过对这个 HID 设备报告描述符的分析，知道它有一个鼠标应用集合和一个键盘应用集合，因而就会分别增加一个鼠标设备和一个键盘设备。按照这个逻辑推测，一个 HID 设备应该可以使用更多的应用集合(每个应用集合使用一个报告 ID)，从而实现更多的标准设备。而使用两个接口的方法实现时，应该会增加两个“USB 人体学输入设备”，因为每个接口都可以看作一个设备，这样的设备叫做 USB 复合设备。

图 4.9.2 带鼠标的 USB 键盘在 BUS Hound 的设备列表中的显示

4.10 带鼠标功能的 USB 键盘(方法二)

之前用一个接口两个应用集合的方法设计了一个带鼠标功能的键盘，本节将介绍另一种

设计方法——使用两个接口来实现。

将前面代码实例复制一份，改名为 UsbKeyboardWithMous(TwoInterfaces)。首先要修改的是设备描述符的产品 ID 号，这里使用 0x0004。然后要修改配置描述符，因为这里使用了两个接口，所以要修改 bNumInterfaces 字段为 0x02。使用两个接口后，每个接口实现一个 HID 设备。这样每个接口各需要一个报告描述符。这里将原来的报告描述符拆成两个：一个叫 KeyboardReportDescriptor，另一个叫 MouseReportDescriptor。这里的报告 ID 其实是可以省略的，为了减少报告发送和接收的处理代码，保留了报告描述符中的报告 ID。但是将鼠标的报告 ID 也改为 1，之所以可以使用同一个报告 ID，是因为这里的两个报告属于不同的接口，可以认为是无关的。报告描述符改了之后，需要在配置描述符集合的 HID 描述符中修改下级描述符的长度。然后，将第 3 章中的 USB 鼠标实例中的配置描述符集合里的接口描述符、HID 描述符、端点描述符复制到本程序的配置描述符集合的最后。后面复制进来的就是第二个接口，修改接口描述符的 bInterfaceNumber 字段为 0x01(接口是从 0 开始编号的，第一个接口的编号就是 0，第二个接口的编号就是 1)。跟前面一样，还需要修改该接口中的 HID 描述符中的下级描述符长度，改为 MouseReportDescriptor 的长度。端点 1 在前面的键盘接口中已经使用了，所以这里的鼠标接口应该使用端点 2，修改端点描述符中的 bEndpointAddress 字段为 0x82，即使用输入端点 2。端点 2 的最大包长为 64 字节，所以修改 bmAttributes 字段为 0x40(当然，如果不改，使用原来的 16 也可以)。配置描述符集合增加了几个描述符，记得修改配置描述符集合的总长度。由于代码改动不大，这里就不再贴出代码了，可以参看光盘中的源代码。图 4.10.1 是该配置集合的一个示意图，帮助大家理解该配置集合的结构。此外，还需要修改一下产品描述的字符串以及产品序列号。

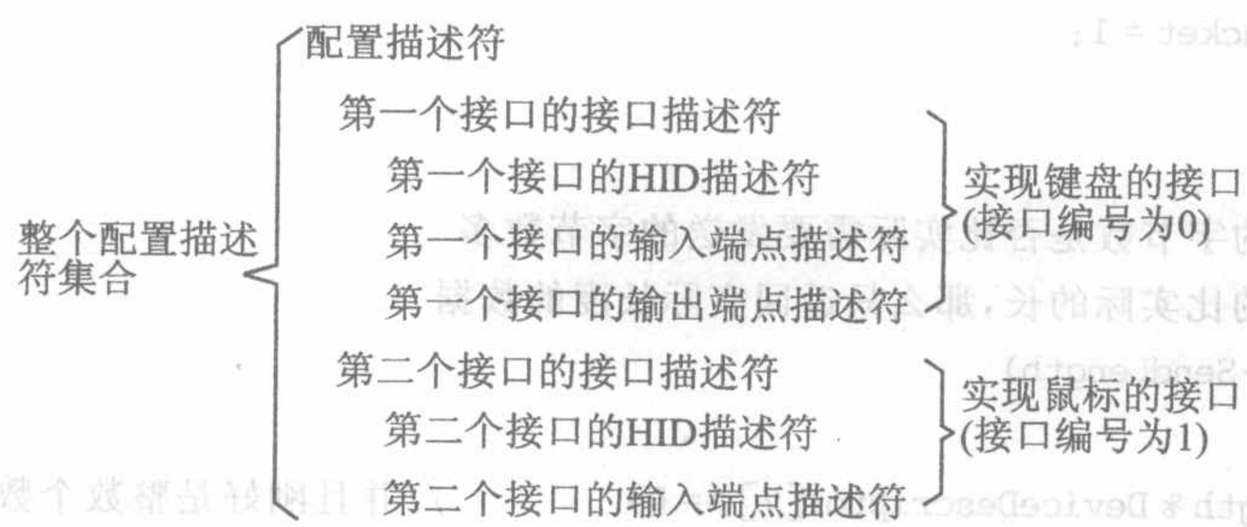


图 4.10.1 具有两个接口的配置描述符集合结构图

现在有两个报告描述符，那我们的程序怎么知道主机请求的是哪个呢？还记得在说获取报告描述符时，请求是发送到哪里的吗？对，是发送到接口的。而在 wIndex 这个字段中，保存的就是所请求的报告描述符所在的接口号。如果 wIndex 为 0，那么就是请求接口 0 的报告描述符，因此就返回键盘的报告描述符 KeyboardReportDescriptor；如果 wIndex 为 1，那么就是请求接口 1 的报告描述符，因此就返回鼠标的报告描述符 MouseReportDescriptor。如果不

是这两个值中的一个，那么说明该请求是非法的，返回一个0长度的数据包(实际上，对于这样的错误，设备应该将端点0挂起，表示出现了一个错误。但这里为了简化处理，就返回一个0长度的数据包)。对获取报告描述符的处理代码修改后如下：

```
case REPORT_DESCRIPTOR:  //报告描述符
 #ifdef DEBUG0
  Prints("报告描述符。\r\n");
 #endif
 //获取报告描述符时，wIndex中保存的是请求的接口号
 //这里有两个接口，接口0用来实现键盘，接口1实现鼠标
 switch(wIndex)
 {
  case 0:    //发送到接口0的请求
   pSendData = KeyboardReportDescriptor;              //需要发送的数据为报告描述符
   SendLength = sizeof(KeyboardReportDescriptor);     //需要返回的数据长度
  break;

  case 1:    //发送到接口1的请求
   pSendData = MouseReportDescriptor;                 //需要发送的数据为报告描述符
   SendLength = sizeof(MouseReportDescriptor);        //需要返回的数据长度
  break;

  default :                                           //其他为未定义的接口，返回0长度数据包
   SendLength = 0;
   NeedZeroPacket = 1;
  break;
 }
 //判断请求的字节数是否比实际需要发送的字节数多
 //如果请求的比实际的长，那么只返回实际长度的数据
 if(wLength>SendLength)
 {
  if(SendLength % DeviceDescriptor[7] == 0)           //并且刚好是整数个数据包时
  {
   NeedZeroPacket = 1;                                //需要返回0长度的数据包
  }
 }
 else
 {
  SendLength = wLength;
 }
```

```
    //将数据通过 EP0 返回
    UsbEp0SendData();
  break;
```

对于按键功能的定义，还是跟 4.10 节中的一样，使用 KEY1 来选择功能。但是我们不再到 SendReport()函数中去判断该发送怎样的报告，而是在 main 函数中直接判断 KEY1 的状态，然后再决定是发送鼠标报告还是键盘报告。因此将原来的 SendReport()函数改成两个，即发送键盘报告的函数 SendKeyboardReport()和发送鼠标报告的函数 SendMouseReport()。

需要注意的是，发送鼠标报告不再是通过端点 1 了，而是变成了端点 2。另外，鼠标报告的报告 ID 也换了，由原来的 2 换成了 1。仿照输入端点 1 的处理方式，增加一个输入端点 2 的忙标志 Ep2InIsBusy，当往端点写入数据后，设置端点忙(Ep2InIsBusy=1)，然后在输入端点 2 的中断处理中将标志清零。记得还要在复位处理中增加对 Ep2InIsBusy 清零的标志，复位后端点是空闲的。至于输出报告，由于依然使用报告 ID 为 1，所以这部分处理不用改动。main 函数中对按键的处理以及发送报告、端点 2 中断处理的函数代码如下：

```
//KEY1 按住时，为键盘功能，松开时，为鼠标功能。
if(KeyPress & KEY1)
{
 OnLed8();
 //如果端点 1 输入没有处于忙状态，则可以发送数据
 if(!Ep1InIsBusy)
 {
  KeyCanChange = 0;                    //禁止按键扫描
  if(KeyUp||KeyDown)                   //如果有按键事件发生
  {
   SendKeyboardReport();               //则返回报告
  }
  KeyCanChange = 1;                    //允许按键扫描
 }
}
else
{
 OffLed8();
 //如果端点 2 输入没有处于忙状态，则可以发送数据
 if(!Ep2InIsBusy)
 {
  KeyCanChange = 0;                    //禁止按键扫描
  if(KeyUp||KeyDown)                   //如果有按键事件发生
  {
```

```
        SendMouseReport();                    //则返回报告
      }
      KeyCanChange = 1;                       //允许按键扫描
    }
  }
/*****************************************************************
函数功能:根据按键情况返回键盘报告的函数
入口参数:无
返    回:无
备    注:无
*****************************************************************/
void SendKeyboardReport(void)
{
  //需要返回的9字节报告的缓冲(1字节报告ID加键盘8字节报告)
  uint8 Buf[9] = {0,0,0,0,0,0,0,0,0};
  //由于需要返回多个按键,所以需要增加一个变量来保存当前的位置
  //由于报告ID占用第一字节,所以普通按键从第四字节开始存放
  uint8 i = 3;
  //用KEY1键来选择剩余的7个键是键盘功能还是鼠标功能
  //当KEY1按住时,剩余7个键为键盘功能,这7个键的功能跟键盘实例的一样
  //根据不同的按键设置输入报告
  Buf[0] = 0x01;                  //第一字节为报告ID,报告ID为1
  if(KeyPress & KEY2)             //如果KEY2按住
  {
   Buf[1]| = 0x02;                //KEY2为左Shift键
  }
  if(KeyPress & KEY3)             //如果KEY3按住
  {
   Buf[1]| = 0x04;                //KEY3为左Alt键
  }
  if(KeyPress & KEY4)             //如果KEY4按住
  {
   Buf[i] = 0x59;                 //KEY4为数字小键盘1键
   i ++ ;                         //切换到下个位置
  }
  if(KeyPress & KEY5)             //如果KEY5按住
  {
   Buf[i] = 0x5A;                 //KEY5数字小键盘2键
   i ++ ;                         //切换到下个位置
```

```
}
if(KeyPress & KEY6)                //如果 KEY6 按住
{
 Buf[i] = 0x5B;                    //KEY6 为数字小键盘 3 键
 i ++ ;                            //切换到下个位置
}
if(KeyPress & KEY7)                //如果 KEY7 按住
{
 Buf[i] = 0x39;                    //KEY7 为大/小写切换键
 i ++ ;                            //切换到下个位置
}
if(KeyPress & KEY8)                //如果 KEY8 按住
{
 Buf[i] = 0x53;                    //KEY8 为数字小键盘功能切换键
}
//报告准备好了,通过端点 1 返回,长度为 9 字节
D12WriteEndpointBuffer(3,9,Buf);
Ep1InIsBusy = 1;                   //设置端点 1 输入忙标志
//记得清除 KeyUp 和 KeyDown
KeyUp = 0;
KeyDown = 0;
}
//////////////////////End of function//////////////////////////////
/*****************************************************************
函数功能:根据按键情况返回鼠标报告的函数
入口参数:无
返      回:无
备      注:无
*****************************************************************/
void SendMouseReport(void)
{
 //需要返回的 5 字节报告的缓冲(1 字节报告 ID 加键盘 4 字节报告)
 uint8 Buf[5] = {0,0,0,0,0};
 //当 KEY1 松开时,剩余 7 个键为鼠标功能,功能分别为
 //KEY2:光标左移;KEY3:光标右移;KEY4:光标上移;KEY5:光标下移
 //KEY6:鼠标左键;KEY7:鼠标中键;KEY8:鼠标右键
 //根据不同的按键设置输入报告。注意,此处报告 ID 要跟报告描述符中的一致
 Buf[0] = 0x01;                    //第一字节为报告 ID,报告 ID 为 1
 if(KeyDown & KEY2)                //如果 KEY2 按下
```

```
{
 Buf[2] = -10;                    //KEY2 为鼠标左移,按一次移动 10 个单位
}
if(KeyDown & KEY3)                //如果 KEY3 按下
{
 Buf[2] = 10;                     //KEY3 为鼠标右移,按一次移动 10 个单位
}
if(KeyDown & KEY4)                //如果 KEY4 按下
{
 Buf[3] = -10;                    //KEY4 为鼠标上移,按一次移动 10 个单位
}
if(KeyDown & KEY5)                //如果 KEY5 按下
{
 Buf[3] = 10;                     //KEY5 为鼠标下移,按一次移动 10 个单位
}
if(KeyPress & KEY6)               //如果 KEY6 按住
{
 Buf[1]| = 0x01;                  //KEY6 为鼠标左键
}
if(KeyPress & KEY7)               //如果 KEY7 按住
{
 Buf[1]| = 0x04;                  //KEY7 为鼠标中键
}
if(KeyPress & KEY8)               //如果 KEY8 按住
{
 Buf[1]| = 0x02;                  //KEY8 为鼠标右键
}
 //注意,此处要通过端点 2 返回
 //报告准备好了,通过端点 2 返回,长度为 5 字节
 D12WriteEndpointBuffer(5,5,Buf);
 Ep2InIsBusy = 1;                 //设置端点 2 输入忙标志
 //记得清除 KeyUp 和 KeyDown
 KeyUp = 0;
 KeyDown = 0;
}
//////////////////////End of function////////////////////////////
/*****************************************************************
函数功能:端点 2 输入中断处理函数
入口参数:无
返      回:无
```

```
备　注:无
*****************************************************************/
void UsbEp2In(void)
{
#ifdef DEBUG0
 Prints("USB 端点 2 输入中断。\r\n");
#endif
 //读最后发送状态,这将清除端点 2 输入的中断标志位
 D12ReadEndpointLastStatus(5);
 //端点 2 输入处于空闲状态
 Ep2InIsBusy = 0;
}
//////////////////////End of function//////////////////////////
```

对修改后的程序编译,并下载到学习板中运行,就会提示发现新硬件,如图 4.10.2 所示。事实上,这是一个 USB 复合设备(USB Composite Device),如图 4.10.3 所示。它会生成两个 USB 人体学输入设备。使用多个接口的方法可以将不同功能的两个或多个 USB 设备复合到一起(当然需要有足够的端点才行)。

该实例的使用方法跟 4.10 节完全一样。

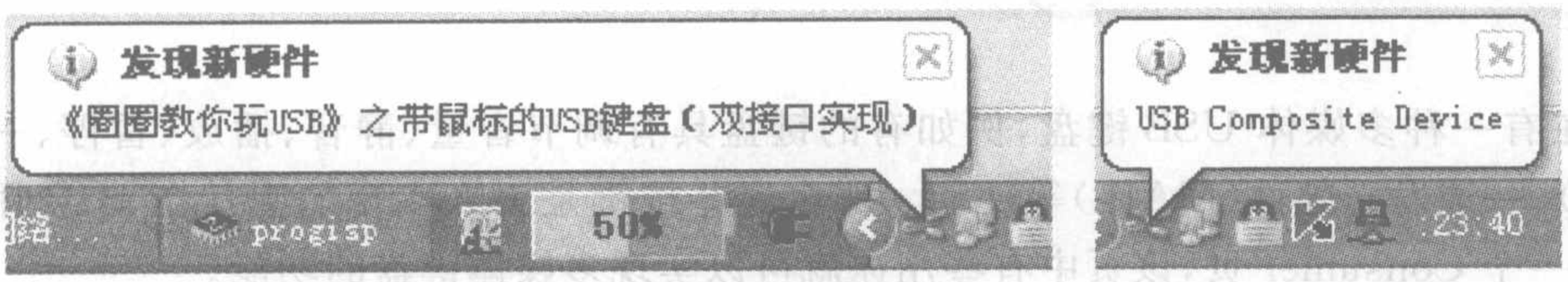


图 4.10.2　发现新硬件

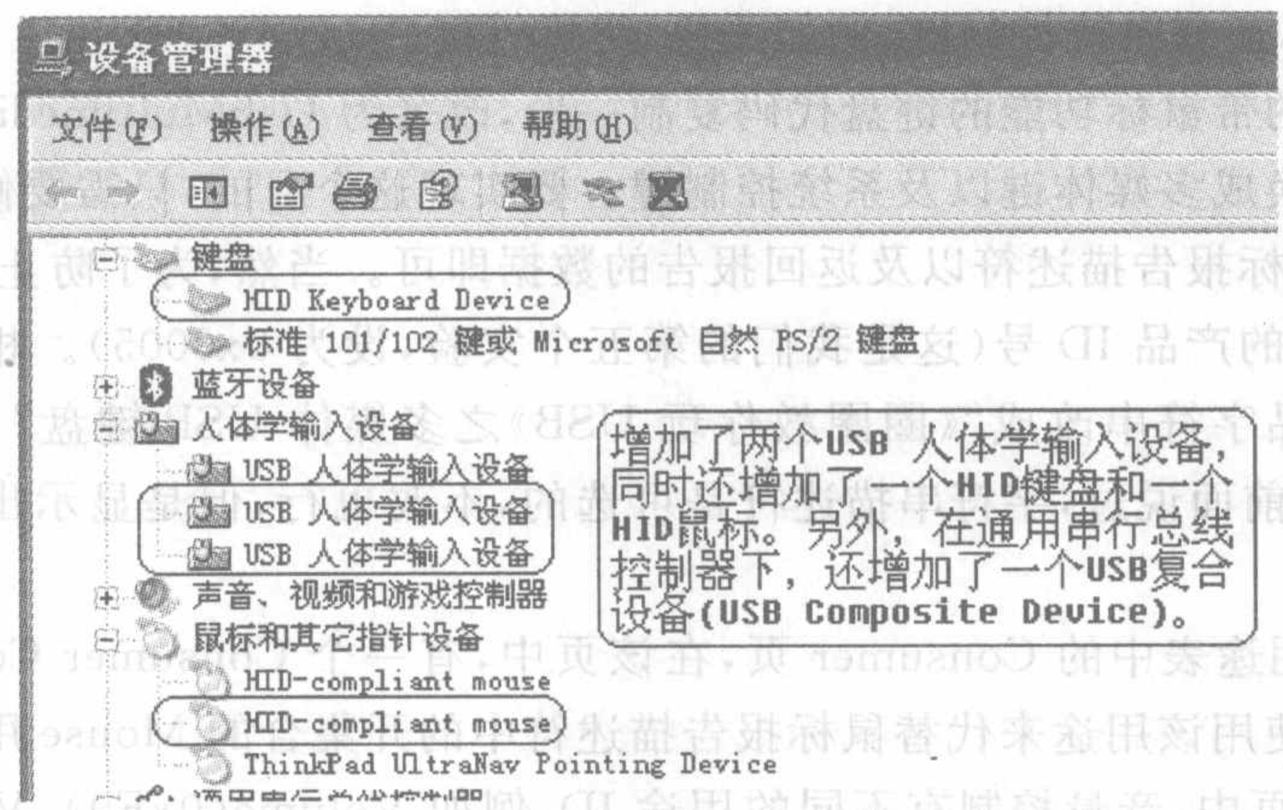


图 4.10.3　设备管理器

再看看设备管理器，里面增加了两个USB人体学输入设备（印证了前面的想法），同时还增加了一个HID Keyboard Device和一个HID-compliant mouse设备。另外，在通用串行总线控制器下，还增加了一个USB Composite Device的设备。

那么这些设备之间的关系是怎样的呢？通过BUS Hound的设备列表可以看出来。由图4.10.4可知，系统生成了一个USB Composite Device，然后再由该设备产生了两个USB人体学输入设备（即我们实现的两个接口），再在USB人体学输入设备分别生成了HID Keyboard Device和HID-compliant mouse设备。弄清楚这些设备之间的关系，对于USB以及驱动程序的设计很有必要。例如一个过滤驱动程序，你需要知道安装在哪个设备之上。

图4.10.4 Bus Hound下显示的复合设备

4.11 多媒体USB键盘

现在有一种多媒体USB键盘，例如有的键盘具有调节音量、静音、播放、暂停、一键上网（WWW）、一键发送邮件（MAIL）等功能。那么这些按键是怎么实现的呢？在HID用途表文档中，有一个Consumer页，该页中有些用途就可以实现多媒体键盘的功能。

本节将实现一个具有多媒体按键的USB键盘（音量降低、音量提升、音量静音、播放/停止、打开IE），同时，还有系统控制键，例如关机、待机等键。

将4.10节中的带鼠标功能的键盘代码复制一份，改名为UsbMultimediaKeyboard。这里把鼠标功能去掉，换成多媒体键以及系统控制键。要实现这个目的，只需要修改报告描述符中的后半部分——鼠标报告描述符以及返回报告的数据即可。当然，为了防止设备冲突，还需要修改设备描述符中的产品ID号（这是我们的第五个实验，设为0x0005）。相关字符串描述符也要改，例如将产品字符串改成“《圈圈教你玩USB》之多媒体USB键盘”，设备序列号改成“2008-08-15”。前面说过，字符串描述符是可选的，不改也行，但是显示出来不对就不大好看了。

先找到HID用途表中的Consumer页，在该页中，有一个Consumer Control（0x01）的开应用集合的用途，使用该用途来代替鼠标报告描述符中的开集合的Mouse用途。

在Consumer页中，音量控制有不同的用途ID，例如Volume（0xE0）、Volume Decrement（0xEA）等。其中，Volume的用途类型是LC（即线性控制）的，可以使用相对值，也可以使用

绝对值。使用相对值时，负数表示音量降低，正数表示音量提升；使用绝对值时，返回的值就是音量的值。而 Volume Decrement、Volume Increment 的用途类型是 RTC(即可重触发)的，它用一个位来表示操作，当该位为1时，就会按照一定的速度重复触发这个事件。例如，当某个位的用途为 Volume Increment(音量提升)时，如果该位的返回值为1，那么音量就会提升；如果该位的值一直保持为1(即按键按住)，那么音量就会按照一定的速度自动提升，直到按键松开为止。本键盘的音量控制将使用 Volume Increment 和 Volume Decrement 方式。播放/停止(Play/Pause)键的用途类型为 OSC，这样的类型只在数据由0往1跳变时才触发事件。打开 IE 的用途为 AC Home(0x0223)，但是文档中给出的用途类型为 Sel，也就是应该像键盘中的普通按键那样，定义一个数组(数组的长度可以为1字节)，然后在数组中返回该 ID 值。但是圈圈发现，直接把它当作 OSC 类型来用也可以。另外，Mute(静音)用途也是一样的，虽然文档中说明的类型是 OOC，但是把它当作 OSC 类型来用也可以。在 Consumer 页中，还有很多用途，读者可以找自己感兴趣的来试试，但是并不是所有的用途都支持的，有的描述可能也不大一样。

以下是圈圈试过的几个用途：

- AL Consumer Control Configuration(0x183)可以打开 Windows Media Player；
- AL Email Reader(0x18A)可以打开发送邮件的 Outlook(应该是键盘上的 MAIL 键)；
- AL Calculator(0x192)可以打开计算器；
- AL Local Machine Browser(0x194)可以打开“我的电脑”(应该是键盘上的 My Computer 键)；
- AC Search(0x221)可以打开搜索文件窗口(应该是键盘上的 Search 键)。

需要注意的是，这些 ID 都是2字节的，所以条目前缀的 bSize 要改成2；因此用途条目的前缀就是 0x0A，而单字节的则是 0x09。在写用途 ID 时要注意小端结构，低字节在先。

系统控制按键在 Generic Desktop 页中，这里使用了两个键：系统待机(System Sleep，0x82)和系统关机(System Power Down，0x81)。

经过上面的分析，设计出的用户控制报告代码如下：

```
/*******************用户控制设备部分报告描述符*********************/
//这是一个全局(bType 为 1)条目，选择用途页为 Consumer Page (0x0C)
0x05, 0x0c,                          // USAGE_PAGE (Consumer Page)

//这是一个局部(bType 为 2)条目，说明接下来的应用集合用于用户控制
0x09, 0x01,                          // USAGE (Consumer Control)

//这是一个主条目(bType 为 0)，开集合，后面跟的数据 0x01 表示该集合是一个应用集合
//它的性质在前面由用途页和用途定义为用户控制
0xa1, 0x01,                          // COLLECTION (Application)

//报告 ID，这里定义用户控制报告的 ID 为 2
```

```
0x85, 0x02,                    //Report ID (2)
//这是一个局部条目。说明用途为音量降低
0x09, 0xea,                    // USAGE (Volume Decrement)
//这是一个局部条目。说明用途为音量提升
0x09, 0xe9,                    // USAGE (Volume Increment)
//这是一个局部条目。说明用途为静音
0x09, 0xe2,                    // USAGE (Mute)
//这是一个局部条目。说明用途为播放/暂停
0x09, 0xcd,                    // USAGE (Play/Pause)
//这是一个局部条目(注意后面带2字节数据,bSize=2)。
//说明用途为打开主页(实际上是浏览器),即有些键盘上的WWW键
//另外,还有很多功能键也可以在Consumer页中找到,大家可以找来试试
//如果要增加按键的话,则要修改报告字段的数量,同时还要修改报告返回
//不过,文档中写着这些用途的类型是sel,也就是说,
//应该用数组来返回(就像键盘普通键那样),但是把它当作OSC来用也可以
//也许是Windows驱动也支持这种方式吧
//不知道键盘的普通按键,是否也可以通过这样的方式来返回呢?
//感兴趣的读者可以试试看,不过这样搞的话,按键一多就需要很多字段了
//因为一个按键就要分配一个字段。注意小端结构,低字节在先
0x0a, 0x23, 0x02,              //USAGE (AC Home(0x0223))
//这是一个全局条目,说明返回的数据的逻辑值最小为0
//因为这里用位来表示一个数据域,因此最小为0,最大为1
0x15, 0x00,                    // LOGICAL_MINIMUM (0)
//这是一个全局条目,说明逻辑值最大为1
0x25, 0x01,                    // LOGICAL_MAXIMUM (1)
//这是一个全局条目,说明数据域的数量为5个(即前面的5个用途)
0x95, 0x05,                    // REPORT_COUNT (5)
//这是一个全局条目,说明每个数据域的长度为1位
0x75, 0x01,                    // REPORT_SIZE (1)
//这是一个主条目,说明有5个长度为1位的数据域(数量和长度由前面的两个全局条目所定义)
//用来作为输入,属性为:Data、Var、Abs。Data表示这些数据可以变动,
//Var表示这些数据域是独立的,每个域表示一个意思。Abs表示绝对值
//这样定义的结果就是,第一个数据域位0表示音量降低,
//第二个数据域位1表示音量提升,第三个数据域位2表示静音开关,
//第四个数据域位3表示播放/暂停,第五个数据域位4表示打开网页
0x81, 0x02,                    // INPUT (Data,Var,Abs)
```

```
//这是一个全局条目,选择用途页为普通桌面 Generic Desktop Page(0x01)
0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)

//这是一个局部条目,说明用途为系统待机
0x09, 0x82,                    // USAGE (System Sleep)

//这是一个局部条目,说明用途为系统关机
0x09, 0x81,                    // USAGE (System Power Down)

//这是一个全局条目,说明数据域的长度为 1 位
0x75, 0x01,                    // REPORT_SIZE (1)

//这是一个全局条目,说明数据域的个数为 2 个
0x95, 0x02,                    // REPORT_COUNT (2)

//这是一个主条目,说明有 2 个长度为 1 位的数据域(数量和长度由前面的两个全局条目所定义)
//用来作为输入,属性为:Data、Var、Abs
//Data 表示这些数据可以变动,Var 表示这些数据域是独立的,每个域表示一个意思。Abs 表示绝对值
//这样定义的结果就是,第一个数据域表示系统待机,第二个数据域表示系统关机
0x81, 0x02,                    // INPUT (Data,Var,Abs)

//这是一个全局条目,说明数据域数量为 1 个
0x95, 0x01,                    // REPORT_COUNT (1)

//这是一个全局条目,说明每个数据域的长度为 1 位
0x75, 0x01,                    // REPORT_SIZE (1)

//这是一个主条目,输入用,由前面两个全局条目可知,长度为 1 位,数量为 1 个
//它的属性为常量(即返回的数据一直是 0)
//这个只是为了凑齐一个字节(前面用了 7 个位)而填充的一些数据而已,所以它没有实际用途
0x81, 0x03,                    // INPUT (Cnst,Var,Abs)

//下面这个主条目用来关闭前面的集合
0xc0                           // END_COLLECTION
/**********************用户控制设备部分报告结束***************/
```

以下描述不包括第一字节报告 ID。通过上面的报告描述符的定义,我们知道返回的输入报告具有 1 字节。其中,位 0 对应音量降低,位 1 对应音量提升,位 2 对应静音控制,位 3 对应播放/停止,位 4 对应打开网页,位 5 对应系统待机,位 6 对应系统关机,位 7 为常量 0。

接着修改返回报告函数 SendReport(),普通键盘功能报告部分不变,将发送鼠标报告的部分改成发送用户控制设备的报告,代码如下:

```
else                        //KEY1 松开,用户控制设备
 {
  Buf[0] = 0x02;            //第一字节为报告 ID,用户控制设备报告 ID 为 2
  if(KeyDown & KEY2)        //如果 KEY2 按下
```

```
{
 Buf[1]| = 0x01;          //KEY2 为音量降低
}
if(KeyDown & KEY3)        //如果 KEY3 按下
{
 Buf[1]| = 0x02;          //KEY3 为音量提升
}
if(KeyDown & KEY4)        //如果 KEY4 按下
{
 Buf[1]| = 0x04;          //KEY4 为静音开关
}
if(KeyDown & KEY5)        //如果 KEY5 按下
{
 Buf[1] = 0x08;           //KEY5 为播放/停止
}
if(KeyPress & KEY6)       //如果 KEY6 按住
{
 Buf[1]| = 0x10;          //KEY6 为打开网页
}
if(KeyPress & KEY7)       //如果 KEY7 按住
{
 Buf[1]| = 0x20;          //KEY7 为系统待机
}
if(KeyPress & KEY8)       //如果 KEY8 按住
{
 Buf[1]| = 0x40;          //KEY8 为系统关机
}
//报告准备好了,通过端点 1 返回,长度为 3 字节
D12WriteEndpointBuffer(3,2,Buf);
}
```

将工程编译,并下载到学习板中,就会弹出发现新硬件的对话框。等驱动程序安装完毕,就可以使用该多媒体键盘了。双击任务栏右下角的小喇叭音量控制图标,打开主音量控制对话框。然后在KEY1不按下的情况下,按下KEY2可以降低音量,按下KEY3可以提升音量,如果按住不放,还可以自动降低或者提升;按下KEY4切换静音状态(注意主音量下的"全部静音"前的小勾);打开Windows Media Player(如果你将前面的AC Home用途改成AL Consumer Control Configuration用途,那么就可以用KEY6键直接打开它了),或者打开"千千静听"(在在圈圈的桌面上,"千千静听"已经被改名为"圈圈静音"啦),随便打开个歌曲,然后按KEY5键,看是否能够控制播放/暂停?再按KEY6键,熟悉的IE是否被打开了(如果是圈圈

的话，会赶紧乘机去网上灌灌水，哈哈）？小心别按到 KEY7 和 KEY8 了，KEY7 是让你的系统进入待机状态的，而 KEY8 则是让你的计算机关机的。这两个键的作用还跟在电源选项中的设置有关。在桌面的空白位置右击，在弹出的菜单中选择“属性”，然后点击“屏幕保护程序”标签页，点击“电源”按钮，再点击“高级”标签页，在下面有个“电源按钮”的分组框，里面有按下电源按钮和睡眠按钮时的选项。在这里作相关选择后，再按下 KEY7 或 KEY8 就会执行相关的操作。当 KEY1 按下后，功能跟前面的键盘是一样的。

这个多媒体键盘还是挺好玩的，可以根据自己的爱好来选择几个功能键，或者再增加些按键来实现更多的功能；甚至还可以做成 USB 红外遥控器。这只要再增加一个一体化的红外遥控接收头以及相关处理代码即可实现。使用这样的遥控器，可以实现鼠标移动、键盘按键、音量调节、播放控制等功能。

再来看看设备管理器中的情况，打开设备管理器，展开“键盘”及“人体学输入设备”，可以看到增加了一个键盘设备和两个人体学输入设备（一个为“USB HID 人体学输入设备”，另一个为“符合 HID 标准的用户控制设备”），如图 4.11.1 所示。根据前面介绍的知识，我们只用了一个接口，那么应该只有一个 USB 设备，然后再在该设备上产生出键盘设备以及用户控制设备。通过 Bus Hound 中的设备列表可以验证这个想法，如图 4.11.2 所示。那么为什么“符合 HID 标准的用户控制设备”也放在“人体学输入设备”下呢？这只是一个设备类型归类而已，并不表示它们之间的层次关系。

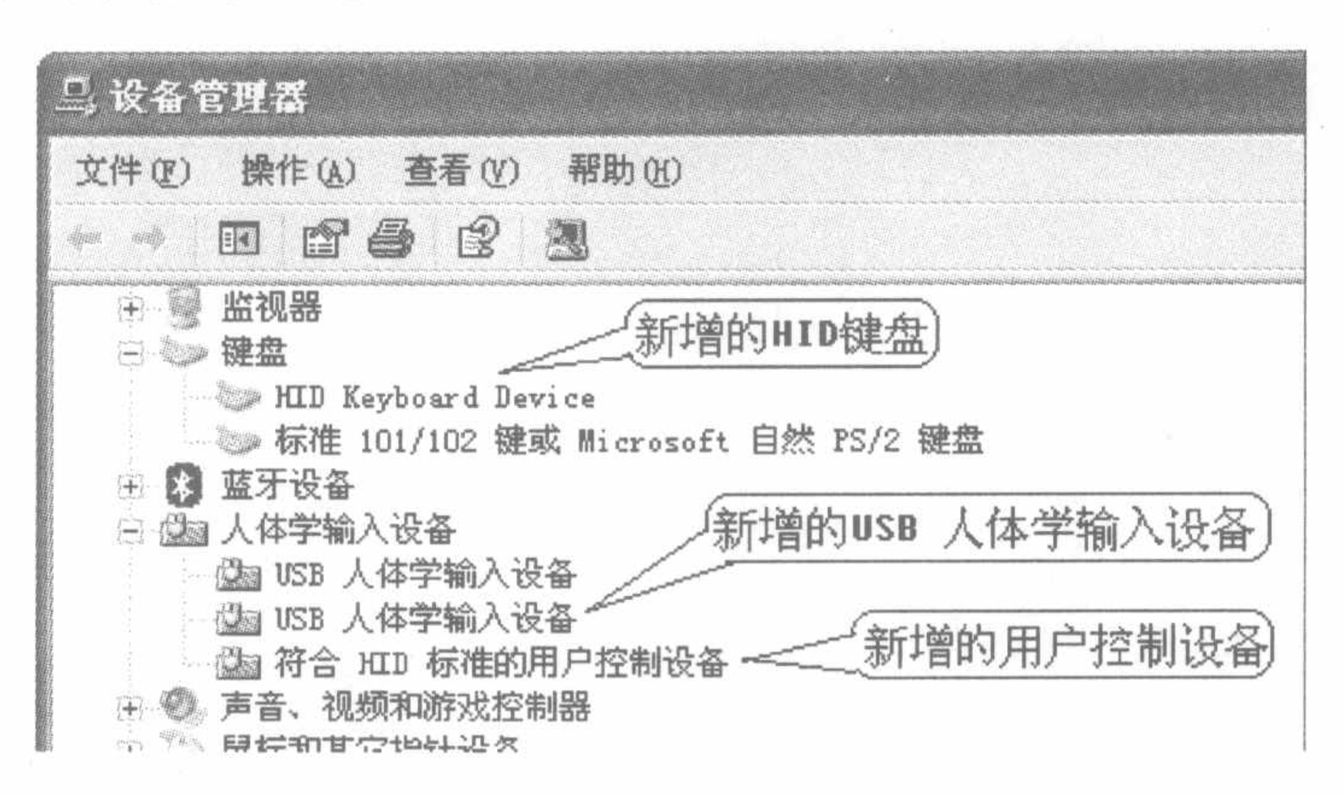


图 4.11.1 设备管理器

图 4.11.2 Bus Hound 设备列表中显示的设备关系

4.12 本章小结

本章的重点是各种描述符的修改，报告的返回，着重介绍了HID报告描述符的构造以及HID用途表。HID用途表在设计HID设备时十分重要，对它了解后，设计一个HID设备的报告描述符就很容易了，而且还可以设计出很多花样。可以上http://www.usb.org去下载一个HID报告描述符生成的工具，使用它可以更方便、更快地构造出一个报告描述符。但是这也是要你在理解了报告描述符的结构以及HID用途之后，才会使用的。这个软件以自然语言的方式提供一些供选择的条目（例如选择用途页、用途、开集合、各种主条目等），然后直接生成相应的数据，从而缩短获取各种条目前缀编码以及各种用途ID号的时间。

第5章

用户自定义的 USB HID 设备

俗话说得好，打铁要趁热。本章将继续讲述 HID 设备。与前两章不同的是，本章的 HID 设备是用户自定义的，也就是说，它不是标准的系统设备。在 Windows 下，标准的系统设备通常是操作系统独占的，应用程序无法直接访问这些设备的驱动程序。而用户自定义的 HID 设备，Windows 操作系统自身并不会访问它们。在 Windows 操作系统下，自带了 HID 设备的驱动程序，因而无需用户自己开发驱动程序。不过，HID 设备也有其固有的缺点，那就是数据只能使用中断或控制传输。由于对中断传输查询的时间间隔最小为一个帧（或者微帧），因而 HID 设备速度受到了限制。对于一些数据量较少的场合（例如按键输入、LED 显示，甚至一些小容量的芯片烧录器等），使用用户自定义的 HID 设备是很合适的。

5.1 MyUsbHid 工程的建立

本章程序将在第 4 章的 USB 键盘程序上进行修改。因为 USB 键盘本身是个 HID 设备，在 USB 键盘的基础上，只要进行少量的代码修改（甚至只修改应用集合的用途即可实现，这在过滤驱动程序的开发时会讲）即可实现。

将 USB 键盘程序 UsbKeyboard 复制一份，改名为 MyUsbHid。

5.2 描述符的修改

首先要修改的是设备描述符中的产品 ID 号（idProduct 字段），这里是第六个实例程序，取 0x0006。其次是要修改配置描述符集合中的接口描述符，因为自定义的 HID 设备不使用子类和协议，将 bInterfaceSubClass 字段和 bInterfaceProtocol 都改为 0。然后将产品字符串改为“《圈圈教你玩 USB》之用户自定义的 USB HID 设备”，设备序列号改为“2008－07－19”。这些描述符修改都比较容易，剩下就是大头——报告描述符的修改。

将报告描述符中开应用集合的用途改为 0x00，在普通桌面页（Generic Desktop Page）中，用途 ID 值 0x00 是未定义的。如果使用该用途来开集合，那么系统将不会把它当作标准系统设备，从而就成了一个用户自定义的 HID 设备。这里我们决定使用 8 字节的输入报告和输出

报告，它们的逻辑最小值为0，逻辑最大值为255。用途可以随便定义，就从1到8吧。至于这些数据具体怎么用，用户可以自己决定。例如像本实验中，输入报告的第一字节用来描述8个按键的状态，第二到第五字节返回报告的次数（增加一个长整型的变量Count，每发送一次报告就加1）。而输出报告的第一字节则用来控制板上8个LED的状态，第二字节（非0时）用来清除上面的报告计数器Count，其他字节未用。修改好的报告描述符代码如下：

```
//USB报告描述符的定义
code uint8 ReportDescriptor[] =
{
 //每行开始的第一字节为该条目的前缀，前缀的格式为：
 //D7～D4:bTag。D3～D2:bType;D1～D0:bSize
 //以下分别对每个条目注释
 //这是一个全局(bType为1)条目，将用途页选择为普通桌面Generic Desktop Page
 //后面跟1字节数据(bSize为1)，后面的字节数就不注释了，自己根据bSize来判断
 0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)

 //这是一个局部(bType为2)条目，用途选择为0x00。在普通桌面页中，该用途是未定义的，如果使用
 //该用途来开集合，那么系统将不会把它当作标准系统设备，从而就成了一个用户自定义的HID设备
 0x09, 0x00,                    // USAGE (0)

 //这是一个主条目(bType为0)，开集合，后面跟的数据0x01表示该集合是一个应用集合
 //它的性质在前面由用途页和用途定义为用户自定义
 0xa1, 0x01,                    // COLLECTION (Application)

 //这是一个全局条目，说明逻辑值最小值为0
 0x15, 0x00,                    // LOGICAL_MINIMUM (0)

 //这是一个全局条目，说明逻辑值最大值为255
 0x25, 0xff,                    // LOGICAL_MAXIMUM (255)

 //这是一个局部条目，说明用途的最小值为1
 0x19, 0x01,                    // USAGE_MINIMUM (1)

 //这是一个局部条目，说明用途的最大值为8
 0x29, 0x08,                    // USAGE_MAXIMUM (8)

 //这是一个全局条目，说明数据域的数量为8个
 0x95, 0x08,                    // REPORT_COUNT (8)

 //这是一个全局条目，说明每个数据域的长度为8位，即1字节
 0x75, 0x08,                    // REPORT_SIZE (8)

 //这是一个主条目，说明有8个长度为8位的数据域作为输入
 0x81, 0x02,                    // INPUT (Data,Var,Abs)

 //这是一个局部条目，说明用途的最小值为1
```

```
0x19, 0x01,                    // USAGE_MINIMUM (1)
//这是一个局部条目,说明用途的最大值为8
0x29, 0x08,                    // USAGE_MAXIMUM (8)
//这是一个主条目。定义输出数据(8字节,注意前面的全局条目)
0x91, 0x02,                    // OUTPUT (Data,Var,Abs)
//下面这个主条目用来关闭前面的集合。bSize为0,所以后面没数据
0xc0                           // END_COLLECTION
};
//////////////////////////报告描述符完毕//////////////////////////
```

通过上面的报告描述符的定义,我们知道返回的输入报告具有8字节。输出报告也有8字节。至于这8字节的数据是干什么用的,就由用户自己来决定了。

5.3 报告的实现

在发送报告的处理中,首先对计数器Count加1。然后将8个按键的情况写入到报告的第一字节,将Count(4字节)分别写入到报告的第二至第五字节,最后将8字节的报告写入到端点1中。发送报告的处理代码如下:

```
/*****************************************************************
函数功能:根据按键情况返回报告的函数
入口参数:无
返      回:无
备      注:无
*****************************************************************/
void SendReport(void)
{
//需要返回的8字节报告的缓冲。在本测试程序中,只使用前5字节
uint8 Buf[8] = {0,0,0,0,0,0,0,0};
//每发送一次数据,则将Count增加1
Count ++ ;
//根据不同的按键设置输入报告。这里将8个按键情况放在第一字节
Buf[0] = KeyPress;
//根据Count的值设置报告的第二到第五字节
Buf[1] = (Count&0xFF);                    //最低字节
Buf[2] = ((Count>>8)&0xFF);               //次低字节
Buf[3] = ((Count>>16)&0xFF);              //次高字节
```

```
 Buf[4] = ((Count>>24)&0xFF);                    //最高字节

 //报告准备好了,通过端点1返回,长度为8字节
 D12WriteEndpointBuffer(3,8,Buf);
 Ep1InIsBusy = 1;                                //设置端点忙标志

 //记得清除KeyUp和KeyDown
 KeyUp = 0;
 KeyDown = 0;
}
//////////////////////End of function////////////////////////////
```

再到端点1输出中断处理函数中,修改输出报告相关处理。原来缓冲区为1字节,这里要改为8字节,同时读取端点1输出缓冲时,要读8字节。第一字节的功能为控制LED,这里不用改动。第二字节为非0值时,将清除发送计数器Count的值。修改后的代码如下:

```
/*****************************************************************
函数功能:端点1输出中断处理函数
入口参数:无
返    回:无
备    注:无
*****************************************************************/
void UsbEp1Out(void)
{
 uint8 Buf[8];                   //用来保存8字节的输出报告
#ifdef DEBUG0
 Prints("USB端点1输出中断。\r\n");
#endif

 //读端点最后状态,这将清除端点1输出的中断标志位
 D12ReadEndpointLastStatus(2);

 //从端点1输出缓冲读回8字节数据
 D12ReadEndpointBuffer(2,8,Buf);

 //清除端点缓冲区
 D12ClearBuffer();

 //输出报告第一字节为LED状态,某位为1时,表示LED亮
 LEDs = ~Buf[0];

 //输出报告的第二字节非0时,清除发送计数器Count
 if(Buf[0]! = 0)
 {
```

```
    Count = 0;
  }
}
//////////////////////////End of function////////////////////////////
```

至此，用户自定义的USB HID设备的代码就修改完成了。然后编译工程，并下载到学习板中运行，就会发现新硬件。在设备管理器中，展开“人体学输入设备”，下面新增了一个“USB-compliant device”和一个“USB人体学输入设备”，如图5.3.1所示。

“USB-compliant device”是由“USB人体学输入设备”产生的，这个关系跟我们前面的USB鼠标、USB键盘与“USB人体学输入设备”之间的关系一样。如果在USB键盘中，将它下面的“USB人体学输入设备”所读到的报告描述符内容改了（过滤驱动程序可以干这个活），将它的应用集合用途改为0或0xFF，那么它就不再产生键盘设备了，而是像这里一样，变成了一个“USB-compliant device”。实现这个功能的过滤驱动程序可以安装在“USB人体输入学设备”之下（这样的过滤驱动叫做下层过滤驱动，相应地，还有上层过滤驱动），那么很自然的，它就位于“USB人体学输入设备”和USB总线驱动程序之间了，是它在中间做了手脚。

图5.3.1 设备管理器中新增的设备

5.4 对用户自定义的USB HID设备的访问

要访问HID设备，就必须跟HID设备的驱动程序打交道。在Windows操作系统环境下，设备通常被当作特殊文件处理。要打开这个设备，就需要知道该设备的路径（设备接口名）。要找到设备的路径，通常使用GUID来查找。

在设备安装时，Windows安装器和驱动程序负责将相应的设备与对应的GUID联系起来，并写入到注册表中。这样通过GUID就可找到对应设备。例如，HID设备的接口类GUID是{4D1E55B2-F16F-11CF-88CB-001111000030}。但是这个值在不同的系统下也许会不一样，所以在本程序中并不直接使用这个GUID，而是使用一个API函数来获取它。把系统中所有HID类的设备都列举出来，然后检查它的VID、PID以及设备版本号，看是不是要访问的设备。如果是，就可以对设备进行各种操作了。

5.5 访问HID设备时所用到的相关函数

本实例程序使用VC++ 6.0开发，使用其他开发环境（例如VB、Delphi等）开发也是类似的，都是调用一些API函数。下面分别介绍访问HID设备时需要使用的一些函数，这些函数的原型可以在MSDN中查找，或者在网上搜索，例如http://www.hszxcx.cn。

5.5.1 获取HID设备的接口类GUID的函数

```
void __stdcall HidD_GetHidGuid(OUT LPGUID HidGuid);
```

调用该函数可以获取HID设备的接口类GUID（Interface class GUID）。其中，OUT是个空的宏定义，仅是为了方便阅读，说明参数的作用是接收数据的；LPGUID是指向GUID结构体的指针类型，因此参数HidGuid就是一个GUID结构体指针。GUID是一个128位（16字节）的整数，分成不同的段用结构体来保存，详细结构可以参看代码中GUID结构体的定义。

5.5.2 获取指定类的所有设备信息集合的函数

```
HDEVINFO SetupDiGetClassDevs(const GUID * ClassGuid,
                             PCTSTR Enumerator,
                             HWND hwndParent,
                             DWORD Flags);
```

调用该函数将返回由ClassGuid指定的所有设备的一个信息集合的句柄。当信息集合使用完毕之后，需要调用函数SetupDiDestroyDeviceInfoList()去销毁。

入口参数ClassGuid就是要指定访问的设备类的GUID，可以是设备接口类GUID（Device Interface class GUID），也可以是安装类GUID（Setup class GUID）。这两种GUID有什么区别呢？设备接口类GUID就是由驱动程序负责添加的GUID，例如上面的那个HID设备的GUID为{4D1E55B2-F16F-11CF-88CB-001111000030}，该GUID将出现在系统注册表中的HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceClasses子键下。接口类GUID下有曾经安装过的设备，键值就是设备的路径（设备接口名）。而安装类GUID则是在设备安装时，由Windows安装器添加到注册表中。通常安装器从安装驱动的inf文件中获取这个安装类GUID，例如安装HID设备的inf文件是Windows/inf文件夹下的input.inf，打开它可以找到ClassGuid={745a17a0-74d3-11d0-b6fe-00a0c90f57da}，这就指定了HID设备的安装类GUID为{745A17A0-74D3-11D0-B6FE-00A0C90F57DA}。设备的安装类GUID出现在注册表的HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class子键下，与接口类GUID类似，安装类GUID下也有曾经安装过的设备，不过这

里是用数字表示的。另外，在注册表的HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum子键下，有不同设备的分类，在分类下可以找到对应的设备，里面记录了设备安装时的安装类GUID。搞清楚这两个GUID是很有必要的，如果类型搞错了，通常就会找不到设备，因为同一个设备的接口类GUID和安装类GUID通常是不同的。那么SetupDiGetClassDevs()函数调用时，如何决定传入的GUID是接口类的还是安装类，这由最后一个参数Flags里面的位4来决定，即DIGCF_DEVICEINTERFACE标志。当位4的值为0时，指定使用安装类GUID；当位4的值为1时，指定使用接口类GUID。另外Flags还有其他几个标志，读者可以自己去看该函数的帮助。本实例程序下要用到它的一个标志是DIGCF_PRESENT，它是Flags的位1。当位1为1时，表示只列举出已经连接的设备，否则将列出全部安装过的设备。

入口参数Enumerator是可选的，当其值为NULL时，将搜索全部设备；它也可以指定一个设备的PnP名字的字符串，从而限制搜索。

参数hwndParent为父窗口的句柄，可以指定为NULL。

当该函数调用失败时，将返回INVALID_HANDLE_VALUE，调用GetLastError()函数可以获取失败的原因。

5.5.3 从设备信息集合中获取一个设备接口信息的函数

```
BOOL SetupDiEnumDeviceInterfaces (HDEVINFO DeviceInfoSet,
                                  PSP_DEVINFO_DATA DeviceInfoData,
                                  const GUID * InterfaceClassGuid,
                                  DWORD MemberIndex,
                                  PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData);
```

调用该函数可以从设备信息集合中获取某个设备接口信息。当该函数调用成功时，返回非0值，当调用失败时(例如指定的设备不存在)，将返回0值。

入口参数DeviceInfoSet是一个设备信息集合的句柄，可以用5.5.2小节中介绍的SetupDiGetClassDevs()函数来获取。

入口参数DeviceInfoData是一个SP_DEVINFO_DATA型的结构体变量，可以用来强制获取某个设备的信息。该参数是可选的，值为NULL表示不使用该参数。

入口参数InterfaceClassGuid是一个指向设备的接口类GUID的指针。

入口参数MemberIndex是整型变量，它指定获取设备信息集合中某个特定的设备信息。0表示第一个设备，1表示第二个设备，以此类推。当该值超出实际的设备数量时，函数就会返回0，表示调用失败。此时使用GetLastError()函数将会得到一个没有更多条目的出错代码：ERROR_NO_MORE_ITEMS。

入口参数DeviceInterfaceData是一个SP_DEVICE_INTERFACE_DATA的结构体，用

来接收设备的信息。在函数调用之前，必须先设置好该结构体的 cbSize 成员为该结构体的大小。

5.5.4 获取指定设备接口详细信息的函数

```
BOOL SetupDiGetDeviceInterfaceDetail(
    HDEVINFO DeviceInfoSet,
    PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
    PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData,
    DWORD DeviceInterfaceDetailDataSize,
    PDWORD RequiredSize,
    PSP_DEVINFO_DATA DeviceInfoData);
```

调用该函数可以获取一个指定设备接口的详细信息，例如设备的路径(设备接口名)。函数调用成功将会返回非0值，调用失败时返回0。

入口参数 DeviceInfoSet 是设备信息集合的句柄。

入口参数 DeviceInterfaceData 是保存设备信息的结构体，可以使用前面介绍的函数 SetupDiEnumDeviceInterfaces()来获取。

入口参数 DeviceInterfaceDetailData 是一个 PSP_DEVICE_INTERFACE_DETAIL_DATA的结构体指针，用来接收设备接口的详细信息。该结构体的成员变量 DevicePath 中就保存着用来打开设备的路径(或者叫设备接口名)，打开设备的 API 函数 CreateFile 可以使用该参数来打指定的设备。在调用函数之前，必须要初始化该结构体中的成员变量 cbSize 为该结构体的大小(注意：不包括后面用来保存路径的缓冲区大小)。该参数可以为 NULL，此时 DeviceInterfaceDetailDataSize 参数必须为0。

入口参数 DeviceInterfaceDetailDataSize 是 DeviceInterfaceDetailData 缓冲区的大小，当该值比实际需要的字节数少时，函数调用失败，返回0。

入口参数 RequiredSize 是一个指向整型变量的指针，该变量用来保存设备接口详细信息实际需要的缓冲区大小。该参数是可选的。

入口参数 DeviceInfoData 是一个用来接收该接口所在设备的设备信息的结构体指针。该参数是可选的，值为 NULL 表示该参数无效。

通常，在获取设备详细信息时调用该函数两次。第一次是为了获取保存设备详细信息所需要的缓冲区长度，这时设置参数 DeviceInterfaceDetailData 为 NULL，参数 DeviceInterfaceDetailDataSize 为0，同时提供一个用来接收字节数变量的指针 RequiredSize。这样函数就会调用失败，调用 GetLastError()函数将会获取到一个缓冲区不足的错误代码。同时会将所需要的缓冲区大小写入到 RequiredSize 所指向的变量中。接着，根据所需要的字节数，去申请一个缓冲区作为接收设备接口详细信息的结构体，并设置该结构体的 cbSize 成员值为该结构体的大小。然后再次调用该函数，将参数 DeviceInterfaceDetailData 设置为缓冲区的首地址，而

DeviceInterfaceDetailDataSize 则为该缓冲区的大小。此时就不再需要参数 RequiredSize 了，将它置为 NULL。

5.5.5 打开设备的函数

```
HANDLE CreateFile(LPCTSTR lpFileName,
                  DWORD dwDesiredAccess,
                  DWORD dwShareMode,
                  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                  DWORD dwCreationDisposition,
                  DWORD dwFlagsAndAttributes,
                  HANDLE hTemplateFile);
```

调用该函数可以打开指定的设备，并返回一个指向该设备的句柄。

入口参数 lpFileName 是一个指向要打开的设备的名称(或者叫做路径)的字符串，通过获取设备详细信息可以找到设备的路径。

入口参数 dwDesiredAccess 是指访问的方式，可以是只读、只写或者读/写方式，也可以设置为无读/写操作(NULL)，这样并不访问设备，但是可以获取到设备的属性。例如 USB 鼠标、USB 键盘等是操作系统的独占设备，因而使用读/写方式是无法打开文件的，但是如果设置为无读/写操作，那么可以获取到这些设备的属性，例如 VID、PID 等。

入口参数 dwShareMode 是共享模式，可以是无共享、共享读、共享写或者共享读/写。当设置为无共享并成功打开设备后，该设备就不能再被其他程序打开来访问了。

入口参数 lpSecurityAttributes 是一个指向 SECURITY_ATTRIBUTES 结构体的指针，用来决定返回的句柄能否继承到子过程中去。当该值为 NULL 时，句柄不能继承。

入口参数 dwCreationDisposition 用来决定当被打开的设备或者文件不存在时，执行怎样的操作。可以选择为始终创建新文件，当文件不存在时创建新文件，或者只能打开已经存在的设备等。打开物理设备时，一般选择为只打开已经存在的设备(设备是一个特殊文件)。

入口参数 dwFlagsAndAttributes 为访问的标志和属性，可以选择普通访问，或者是异步调用等，具体的选项很多，可以参考 MSDN 帮助。一般同步调用时使用参数 FILE_ATTRIBUTE_NORMAL，异步调用时使用参数 FILE_FLAG_OVERLAPPED。

入口参数 hTemplateFile 可以指定一个临时文件的句柄，该参数可以为 NULL，即不使用临时文件。

使用该函数打开设备后，就可以使用该函数返回的句柄来获取设备的属性，以及对设备进行读/写操作。

5.5.6 获取HID设备属性的函数

```
BOOLEAN __stdcall HidD_GetAttributes(IN HANDLE HidDeviceObject,
                                     OUT PHIDD_ATTRIBUTES Attributes);
```

调用该函数可以获取指定设备的属性，例如VID、PID、设备版本号等。调用成功后返回非0值，调用失败时返回0。

入口参数HidDeviceObject是被访问设备的句柄，可调用CreateFile打开一个设备从而获得该句柄。

入口参数Attributes是一个指向HIDD_ATTRIBUTES结构体的指针，用来保存HID设备的属性。

5.5.7 从设备读取数据的函数

```
BOOL ReadFile(HANDLE hFile,
              LPVOID lpBuffer,
              DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesRead,
              LPOVERLAPPED lpOverlapped);
```


调用该函数将从指定的设备读取数据。当操作成功时，返回非0值；当操作失败时，返回0值。注意当CreateFile打开文件时，如果选择了异步打开，那么该函数调用后即使操作未完成也会立即返回。此时返回值为0，但这可能并不表示操作失败，通过调用GetLastError()函数可以获取出错的代码。如果出错代码为ERROR_IO_PENDING，则表示驱动程序将这个IRP挂起了，等数据成功返回时，这个IRP才会被完成。

入口参数hFile是需要访问的设备或文件的句柄。

入口参数lpBuffer为保存接收数据的缓冲区。

入口参数nNumberOfBytesToRead为需要读取数据的长度(字节数)。

入口参数lpNumberOfBytesRead为一个指向保存实际读取到多少字节的变量的指针。当lpOverlapped不为NULL时，该参数可以为NULL。

入口参数lpOverlapped为一个指向OVERLAPPED结构体的指针。当选择为异步调用时，必须提供一个lpOverlapped参数。该参数指向的结构体提供一个事件的句柄，当IRP完成时会触发该事件。调用该函数时，事件会自动设置为无效状态，调用者不用负责清除事件标志。另外，在该结构体中，还提供了读取数据的偏移量。使用了lpOverlapped参数后，将不再使用lpNumberOfBytesRead所指向的变量来保存实际接收到的字节数了，而是通过函数GetOverlappedResult()来获取。

对于USB HID设备，使用该函数只能从中断端点获取报告数据，如果要从控制端点获取

报告，则使用函数 HidD_GetInputReport()。

5.5.8 往设备写数据的函数

```
BOOL WriteFile(HANDLE hFile,
               LPCVOID lpBuffer,
               DWORD nNumberOfBytesToWrite,
               LPDWORD lpNumberOfBytesWritten,
               LPOVERLAPPED lpOverlapped);
```

调用该函数将把指定的数据发送到指定的设备。该函数的返回值的意义跟 ReadFile()函数一样。

入口参数 hFile 是需要访问的设备或文件的句柄。

入口参数 lpBuffer 是指向待写入数据的缓冲区。

入口参数 nNumberOfBytesToWrite 为需要写入的字节数。

入口参数 lpNumberOfBytesWritten 为指向一个用来保存实际写入字节数的变量的指针。

入口参数 lpOverlapped 为一个指向 OVERLAPPED 结构体的指针，用途跟 ReadFile()函数一样，请参看 ReadFile 的说明。

对于 USB HID 设备，使用该函数只能从中断端点发送报告，如果要从控制端点发送报告，则使用函数 HidD_SetOutputReport()。

5.5.9 通过控制端点0读取报告的函数

```
BOOLEAN __stdcall HidD_GetInputReport (IN HANDLE HidDeviceObject,
                                       OUT PVOID ReportBuffer,
                                       IN ULONG ReportBufferLength);
```

调用该函数可以通过控制端点0获取报告，调用成功时，返回非0值；调用失败时，返回0。调用该函数后，驱动程序将会发送获取报告的类输入请求，设备在数据阶段通过控制端点0返回其报告。如果设备在规定的时间内未返回报告，该函数将会超时返回。

入口参数 HidDeviceObjec 为已打开的设备的句柄。

入口参数 ReportBuffer 是用来接收报告的缓冲区。

入口参数 ReportBufferLength 是接收缓冲区的长度。

5.5.10 通过控制端点0发送报告的函数

```
BOOLEAN __stdcall HidD_SetOutputReport (IN HANDLE HidDeviceObject,
                                        IN PVOID ReportBuffer,
                                        IN ULONG ReportBufferLength);
```

调用该函数可以通过控制端点0发送报告，调用成功时，返回非0值；调用失败时，返回

0。调用该函数后，驱动程序将会发送设置报告的类输出请求，并在数据阶段将报告发送到设备的控制端点0。如果设备在规定的时间内未能接收报告，那么该函数将会超时返回。

入口参数 HidDeviceObjec 为已打开的设备的句柄。

入口参数 ReportBuffer 是保存待发送报告的缓冲区。

入口参数 ReportBufferLength 是需要发送报告的长度。

5.5.11 关闭句柄的函数

```
BOOL CloseHandle(HANDLE hObject);
```

调用该函数将关闭已打开的句柄 hObject。

入口参数 hObject 为需要关闭的设备或文件的句柄。

5.5.12 需要包含的库文件

函数名中包含 Hid 字样的函数属于 hid.lib，函数名中包含 Setup 字样的函数属于 setupapi.lib，需要在工程设置选项的 Link 下的 Library Modules 下加入 hid.lib 和 setupapi.lib，否则就会出现链接错误。不过，VC 自己并没有自带这个 hid.lib，在 DDK 中才有，有些读者可能没有安装 DDK，因此圈圈将这个 hid.lib 复制了一份放在工程包中。注意这些函数的声明，是标准的 C 函数格式，因此在 C++文件中引用头文件时加上 extern "C"，否则会链接不上。

5.6 访问 USB HID 设备的上位机软件的实现

5.6.1 上位机程序编写的思路

查找指定的 USB HID 设备的思路和步骤如下：

① 使用 HidD_GetHidGuid()函数获取 HID 设备的接口类 GUID。

② 使用 SetupDiGetClassDevs()函数获取 HID 类中所有设备的信息集合。

③ 在该设备信息集合中，使用 SetupDiEnumDeviceInterfaces()函数去获取一个设备的信息。如果调用该函数时返回失败，则说明已经到了设备集合的末尾，后面已经没有设备了，此时应该退出查找。

④ 使用 SetupDiGetDeviceInterfaceDetail()函数获取某个设备的详细信息。要获取某个设备的详细信息，SetupDiGetDeviceInterfaceDetail()函数必须调用两次。第一次调用是为了得到保存设备详细信息需要多大的缓冲区，第二次调用才是真正的获取设备详细信息。

⑤ 获得设备的详细信息后，就可以使用 CreateFile()函数打开指定的设备了。

⑥ 打开设备后，再使用 HidD_GetAttributes()函数获取设备的属性，在属性中包含了 VID、PID 以及产品版本号等信息。然后再比较 VID、PID 以及产品版本号是否跟所指定的一

致。如果一致，则退出查找，说明设备已经找到；如果不一致，说明这个设备不是我们需要的设备，切换到下一个设备，然后重复前面的步骤③～⑥。

找到指定的设备后，就可以对设备进行读/写数据了。由于我们设计的HID设备是使用中断端点来传输数据，因此应该调用ReadFile()和WriteFile()函数来读、写报告。如果要操作的设备是使用默认的控制端点来读/写报告的，应该调用HidD_GetInputReport()函数和HidD_SetOutputReport()函数。

为了演示应用程序跟实验板之间的通信，在界面上放置两排8个的LED按键。一排用来控制板上的LED状态，可以用鼠标单击在开和关之间切换，另一排用来显示学习板上按键的情况。当某个按键按下时，界面上对应的LED就会亮。

实现开关状态显示以及计数值显示的编程思路是：在读报告线程中调用ReadFile()函数发送读取报告的请求，然后等待事件的发生。当学习板上有按键变动时，就会返回数据。此时ReadFile()发送的请求就会被完成，并设置事件为有效状态。这将唤醒读报告线程，从而根据接收到的数据设置LED状态以及计数器值。处理完毕后再发送读取报告的请求，等待下一次报告的返回。

实现开关控制以及清除计数值的编程思路是：当对应的按键按下后，就设置好要发送的报告数据，然后使用WriteFile()函数将报告发送到设备。同时设置一个标志，说明数据正在发送中。当数据发送完毕，就会触发事件，然后在事件响应代码中将该标志清除。在每次发送数据前，应该先检查该标志。如果数据正在发送中，那么就不能发送数据，操作无效。

需要注意的是，虽然我们的设备并没有设置报告ID，但是数据在从“USB人体学输入设备”传到“USB-compliant device”时，会自动增加一个值为0的报告ID。这样获取报告时实际上会读到9字节的数据，后面8字节才是设备真正返回的数据。同样，在发送数据时，也要事先增加一个值为0的报告ID，下层的驱动会自动将前面的报告ID去掉，然后将剩余的数据发送到设备。当然，如果设备指定了报告ID，就不存在上面这个问题。

当设备在操作过程中被拔下时，应用程序应该要能够感知到这一事件，并停止对设备的继续操作。另外，当不再需要使用设备或者关闭程序时，应该要关闭已经打开的设备句柄。

下面给出部分该测试程序的关键代码，只要把这些代码搞清楚了，应该就会知道如何去查找设备、打开设备以及操作设备了。

5.6.2 查找及打开HID设备的代码

在该程序的操作界面上，有一个打开设备的按钮，当点击该按钮时，就会在所有已经连接的HID设备中搜索目的HID设备。当指定的设备找到后，就会退出查找，并分别以读方式和写方式打开设备。然后禁止打开设备的按钮，使能其他操作按钮。该函数中使用了好几个上面介绍过的API函数，记得需要包括相应的头文件以及在Link选项中增加相应的库文件。函数中使用了一些全局变量，用来保存设备路径名及打开设备的句柄等。代码如下：

```
//点击打开设备按钮的处理函数
void CMyUsbHidTestAppDlg::OnOpenDevice()
{
  //定义一个 GUID 的结构体 HidGuid 来保存 HID 设备的接口类 GUID
  GUID HidGuid;
  //定义一个 DEVINFO 的句柄 hDevInfoSet 来保存获取到的设备信息集合句柄
  HDEVINFO hDevInfoSet;
  //定义 MemberIndex,表示当前搜索到第几个设备,0 表示第一个设备
  DWORD MemberIndex;
  //DevInterfaceData,用来保存设备的驱动接口信息
  SP_DEVICE_INTERFACE_DATA DevInterfaceData;
  //定义一个 BOOL 变量,保存函数调用是否返回成功
  BOOL Result;
  //定义一个 RequiredSize 的变量,用来接收需要保存详细信息的缓冲长度
  DWORD RequiredSize;
  //定义一个指向设备详细信息的结构体指针
  PSP_DEVICE_INTERFACE_DETAIL_DATA pDevDetailData;
  //定义一个用来保存打开设备的句柄
  HANDLE hDevHandle;
  //定义一个 HIDD_ATTRIBUTES 的结构体变量,保存设备的属性
  HIDD_ATTRIBUTES DevAttributes;
  //初始化设备未找到
  MyDevFound = FALSE;
  //获取在文本框中设置的 VID、PID、PVN
  GetMyIDs();
  //初始化读、写句柄为无效句柄
  hReadHandle = INVALID_HANDLE_VALUE;
  hWriteHandle = INVALID_HANDLE_VALUE;
  //对 DevInterfaceData 结构体的 cbSize 初始化为结构体大小
  DevInterfaceData.cbSize = sizeof(DevInterfaceData);
  //对 DevAttributes 结构体的 Size 初始化为结构体大小
  DevAttributes.Size = sizeof(DevAttributes);
  //调用 HidD_GetHidGuid()函数获取 HID 设备的 GUID,并保存在 HidGuid 中
  HidD_GetHidGuid(&HidGuid);
  //根据 HidGuid 来获取设备信息集合。其中 Flags 参数设置为 DIGCF_DEVICEINTERFACE|DIGCF_PRESENT
  //前者表示使用的 GUID 为接口类 GUID,后者表示只列举正在使用的设备
  //因为我们这里只查找已经连接上的设备,返回的句柄保存在 hDevinfo 中
```

```
//注意设备信息集合在使用完毕后要使用函数 SetupDiDestroyDeviceInfoList()销毁
//不然会造成内存泄漏
hDevInfoSet = SetupDiGetClassDevs(&HidGuid,
                                  NULL,
                                  NULL,
                                  DIGCF_DEVICEINTERFACE|DIGCF_PRESENT);

AddToInfOut("开始查找设备");
//对设备集合中每个设备进行列举,检查是否是我们要找的设备
//当找到指定的设备,或者设备已经查找完毕时,就退出查找
//首先指向第一个设备,即将 MemberIndex 置为 0
MemberIndex = 0;
while(1)
{
 //调用 SetupDiEnumDeviceInterfaces,在设备信息集合中获取编号为 MemberIndex 的设备信息
 Result = SetupDiEnumDeviceInterfaces(hDevInfoSet,
                                      NULL,
                                      &HidGuid,
                                      MemberIndex,
                                      &DevInterfaceData);
 //如果获取信息失败,则说明设备已经查找完毕,退出循环
 if(Result == FALSE) break;

 //将 MemberIndex 指向下一个设备
 MemberIndex ++ ;

 //如果获取信息成功,则继续获取该设备的详细信息。在获取设备详细信息时,需要先知道保存
 //详细信息需要多大的缓冲区,这通过第一次调用函数 SetupDiGetDeviceInterfaceDetail 来获取
 //这时提供缓冲区和长度都为 NULL 的参数,并提供一个用来保存需要多大缓冲区的变量 RequiredSize
 Result = SetupDiGetDeviceInterfaceDetail(hDevInfoSet,
                                          &DevInterfaceData,
                                          NULL,
                                          NULL,
                                          &RequiredSize,
                                          NULL);
 //然后,分配一个大小为 RequiredSize 的缓冲区,用来保存设备详细信息
 pDevDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(RequiredSize);
 if(pDevDetailData == NULL) //如果内存不足,则直接返回
 {
  MessageBox("内存不足!");
  SetupDiDestroyDeviceInfoList(hDevInfoSet);
```

```
 return;
}

//设置 pDevDetailData 的 cbSize 为结构体的大小(注意只是结构体大小,不包括后面缓冲区)
pDevDetailData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

//再次调用 SetupDiGetDeviceInterfaceDetail()函数来获取设备的详细信息
//这次调用设置使用的缓冲区以及缓冲区大小
Result = SetupDiGetDeviceInterfaceDetail(hDevInfoSet,
                                         &DevInterfaceData,
                                         pDevDetailData,
                                         RequiredSize,
                                         NULL,
                                         NULL);

//将设备路径复制出来,然后销毁刚刚申请的内存
MyDevPathName = pDevDetailData->DevicePath;
free(pDevDetailData);

//如果调用失败,则查找下一个设备
if(Result == FALSE) continue;

//如果调用成功,则使用不带读/写访问的 CreateFile()函数来获取设备的属性,包括 VID、PID、版本号等
//对于一些独占设备(例如 USB 键盘),使用读访问方式是无法打开的,
//而使用不带读/写访问的格式才可以打开这些设备,从而获取设备的属性
hDevHandle = CreateFile(MyDevPathName,
                        NULL,
                        FILE_SHARE_READ|FILE_SHARE_WRITE,
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);

//如果打开成功,则获取设备属性
if(hDevHandle! = INVALID_HANDLE_VALUE)
{
 //获取设备的属性并保存在 DevAttributes 结构体中
 Result = HidD_GetAttributes(hDevHandle,
                             &DevAttributes);

 //关闭刚刚打开的设备
 CloseHandle(hDevHandle);

 //获取失败,查找下一个
 if(Result == FALSE) continue;
```

```
//如果获取成功,则将属性中的VID、PID以及设备版本号与我们需要的进行比较
//如果都一致,则说明它就是要找的设备
if(DevAttributes.VendorID == MyVid)                    //如果VID相等
 if(DevAttributes.ProductID == MyPid)                  //并且PID相等
  if(DevAttributes.VersionNumber == MyPvn)             //并且设备版本号相等
  {
   MyDevFound = TRUE;                                  //设置设备已经找到
   AddToInfOut("设备已经找到");

   //分别使用读、写方式打开之,保存其句柄并且选择为异步访问方式
   //读方式打开设备
   hReadHandle = CreateFile(MyDevPathName,
                            GENERIC_READ,
                            FILE_SHARE_READ|FILE_SHARE_WRITE,
                            NULL,
                            OPEN_EXISTING,
                            FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED,
                            NULL);
   if(hReadHandle! = INVALID_HANDLE_VALUE)
     AddToInfOut("读访问打开设备成功");
   else AddToInfOut("读访问打开设备失败");

   //写方式打开设备
   hWriteHandle = CreateFile(MyDevPathName,
                             GENERIC_WRITE,
                             FILE_SHARE_READ|FILE_SHARE_WRITE,
                             NULL,
                             OPEN_EXISTING,
                             FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED,
                             NULL);
   if(hWriteHandle! = INVALID_HANDLE_VALUE)
     AddToInfOut("写访问打开设备成功");
   else AddToInfOut("写访问打开设备失败");

   DataInSending = FALSE;          //可以发送数据

   //手动触发事件,让读报告线程恢复运行。因为在这之前并没有调用读数据的函数
   //也就不会引起事件的产生,所以需要先手动触发一次事件,让读报告线程恢复运行
   SetEvent(ReadOverlapped.hEvent);

   //显示设备的状态
   SetDlgItemText(IDC_DS,"设备已打开");
```

```
            //找到设备,退出循环。本程序只检测一个目标设备,查找到后就退出查找了
            //如果你需要将所有的目标设备都列出来,可以设置一个数组,
            //找到后就保存在数组中,直到所有设备都查找完毕才退出查找
            break;
        }
    }
    //如果打开失败,则查找下一个设备
    else continue;
}

//调用 SetupDiDestroyDeviceInfoList()函数销毁设备信息集合
SetupDiDestroyDeviceInfoList(hDevInfoSet);

//如果设备已经找到,那么应该使能各操作按钮,并同时禁止打开设备按钮
if(MyDevFound)
{
    //禁止打开设备按键,使能关闭设备,清计数器按键
    GetDlgItem(IDC_OPEN_DEVICE)->EnableWindow(FALSE);
    GetDlgItem(IDC_CLOSE_DEVICE)->EnableWindow(TRUE);
    GetDlgItem(IDC_CLEAR_COUNTER)->EnableWindow(TRUE);

    //使能 LED 控制按键
    GetDlgItem(IDC_LED1)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED2)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED3)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED4)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED5)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED6)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED7)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED8)->EnableWindow(TRUE);
}
else
{
    AddToInfOut("设备未找到");
}
}
//////////////////////////////End of function//////////////////////
```

5.6.3 读输入报告线程的代码

为了避免读操作时应用程序界面失去响应,单独创建一个读输入报告的线程。该线程的

作用就是使用ReadFile()函数不断地从设备读取数据，然后根据读取到的数据设置按键情况及计数值。由于使用的是异步调用，因而在调用ReadFile()函数时提供一个Overlapped的结构，该结构中含有一个事件的句柄。事件的创建以及Overlapped结构的初始化在主窗口初始化时完成。

当读报告线程被创建时，设备还没有打开，因而等待事件触发。当设备打开后，手动设置事件为有效状态，从而唤醒读报告线程。此时读报告线程检测到设备已经打开，使用ReadFile()函数请求设备输入数据。然后读报告线程等待事件被触发，进入挂起状态。当ReadFile()函数完成后，会设置事件为有效状态，从而唤醒读报告线程。接着读报告线程就可以对返回的数据做相关处理了。数据处理完毕后，又可以再次调用ReadFile()函数来读取数据，从而实现一直请求数据输入的功能。

需要注意的是，要让该线程能够开始读取报告，必须要在打开设备后手动触发一次，或者调用一次读数据的ReadFile()函数，不然它就会一直等待事件的触发。

对返回的数据处理，主要是检查返回的数据量以及报告ID是否正确，如果正确，那么就设置界面上各开关的状态以及计数器的值。

由于该函数并不是CMyUsbHidTestAppDlg类(就是该工程的主窗口类)中的成员函数，所以无法直接调用CMyUsbHidTestAppDlg类中的成员函数。在创建该线程时，通过pParam参数传递了一个this指针，将参数pParam强制转化为CMyUsbHidTestAppDlg类的指针即可访问CMyUsbHidTestAppDlg类中的成员函数。

读报告的线程代码如下：

```
//读报告的线程
UINT ReadReportThread(LPVOID pParam)
{
 CMyUsbHidTestAppDlg * pAppDlg;
 DWORD Length, Counter;
 UINT i;
 CString Str;
 //将参数pParam取出，并转换为CMyUsbHidTestAppDlg型指针，以供下面调用其成员函数
 pAppDlg = (CMyUsbHidTestAppDlg * )pParam;
 //该线程是个死循环，直到程序退出时，它才退出
 while(1)
 {
  //设置事件为无效状态
  ResetEvent(ReadOverlapped.hEvent);
  //如果设备已经找到
  if(MyDevFound == TRUE)
```

```
{
 if(hReadHandle == INVALID_HANDLE_VALUE) //如果读句柄无效
 {
  pAppDlg->AddToInfOut("无效的读报告句柄,可能打开设备时失败");
 }
 else  //否则,句柄有效
 {
  //则调用 ReadFile()函数请求 9 字节的报告数据
  ReadFile(hReadHandle,
           ReadReportBuffer,
           9,
           NULL,
           &ReadOverlapped);
 }
 //等待事件触发
 WaitForSingleObject(ReadOverlapped.hEvent,INFINITE);
 //如果等待过程中设备被拔出,也会导致事件触发,但此时 MyDevFound 被设置为假
 //因此如果在这里判断 MyDevFound 为假,则就进入下一轮循环
 if(MyDevFound == FALSE) continue;
 //如果设备没有被拔下,则是 ReadFile()函数正常操作完成
 //通过 GetOverlappedResult()函数来获取实际读取到的字节数
 GetOverlappedResult(hReadHandle,&ReadOverlapped,&Length,TRUE);
 //如果字节数不为 0,则将读到的数据显示到信息框中
 if(Length! = 0)
 {
  pAppDlg->AddToInfOut("读取报告"+pAppDlg->itos(Length)+"字节");
  Str = "";
  for(i = 0;i<Length;i++)
  {
   Str+ = pAppDlg->itos(ReadReportBuffer[i],16).Right(2)+" ";
  }
  pAppDlg->AddToInfOut(Str, FALSE);
 }

 //如果字节数为 9,则说明获取到了正确的 9 字节报告
 if(Length == 9)
 {
  //第一字节为报告 ID,应该为 0
  if(ReadReportBuffer[0] == 0)
```

```
    {
     //第二字节为按键状态,将其保存到 KeyStatus 中
     KeyStatus = ReadReportBuffer[1];
     //刷新按键的情况
     pAppDlg->SetKeyStatus();
     //第三、四、五、六字节为设备返回的发送次数值。计算出值后并显示
     Counter = ReadReportBuffer[5];
     Counter = (Counter<<8) + ReadReportBuffer[4];
     Counter = (Counter<<8) + ReadReportBuffer[3];
     Counter = (Counter<<8) + ReadReportBuffer[2];
     pAppDlg->SetCounterNumber(Counter);
    }
   }
   else
   {
    //阻塞线程,直到下次事件被触发
    WaitForSingleObject(ReadOverlapped.hEvent,INFINITE);
   }
  }
  return 0;
}
////////////////////////////End of function////////////////////
```

5.6.4 写输出报告的代码(发送LED的状态)

当用鼠标点击界面上的LED灯时,将触发对应按钮的单击事件。在事件响应代码中,将当前LED设置的状态组织成报告的格式,再通过WriteFile()函数发送给设备。设备接收到报告之后就会根据报告中的数据来设置板上LED的状态。

发送输出报告的函数代码如下:

```
//发送 LED 的状态。
BOOL CMyUsbHidTestAppDlg::SendLedStatus()
{
 BOOL Result;
 UINT LastError;
 UINT i;
 CString Str;
 //如果设备没有找到,则返回失败
```

```
if(MyDevFound == FALSE)
{
 AddToInfOut("设备未找到");
 return FALSE;
}

//如果句柄无效,则说明打开设备失败
if(hWriteHandle == INVALID_HANDLE_VALUE)
{
 AddToInfOut("无效的写报告句柄,可能是打开设备时失败");
 return FALSE;
}

//如果数据仍在发送中,则返回失败
if(DataInSending == TRUE)
{
 AddToInfOut("数据正在发送中,暂时不能发送");
 return FALSE;
}

//设置要发送报告的数据
WriteReportBuffer[0] = 0x00;                    //报告 ID 为 0
WriteReportBuffer[1] = LedStatus;               //将 LED 状态放到缓冲区中

//显示发送数据的信息
AddToInfOut("发送输出报告 9 字节");
Str = "";
for(i = 0;i<9;i ++ )
{
 Str + = itos(WriteReportBuffer[i],16).Right(2) + " ";
}
AddToInfOut(Str,FALSE);

//设置正在发送标志
DataInSending = TRUE;

//调用 WriteFile()函数发送数据
Result = WriteFile(hWriteHandle,
                   WriteReportBuffer,
                   9,
                   NULL,
                   &WriteOverlapped);

//如果函数返回失败,则可能是真的失败,也可能是 I/O 挂起了
```

```
if(Result == FALSE)
{
 //获取最后错误代码
 LastError = GetLastError();
 //看是否是真的 I/O 挂起
 if((LastError == ERROR_IO_PENDING)||(LastError == ERROR_SUCCESS))
 {
  return TRUE;
 }
 //否则,是函数调用时发生错误,显示错误代码
 else
 {
  DataInSending = FALSE;
  AddToInfOut("发送失败,错误代码:" + itos(LastError));
  //如果最后错误为 1,说明该设备不支持该函数
  if(LastError == 1)
  {
   AddToInfOut("该设备不支持 WriteFile 函数。", FALSE);
  }
  return FALSE;
 }
}
//否则,函数返回成功
else
{
 DataInSending = FALSE;
 return TRUE;
}
}
////////////////////////////End of function////////////////////
```

5.6.5 写输出报告线程的代码

写输出报告线程的功能比较简单,它只是简单地等待事件被触发,然后清除一个正在发送数据的标志 DataInSending。在主线程中,需要发送数据时先检查 DataInSending 标志,如果可以发送数据,则调用 WriteFile()函数将数据发送到设备,并设置 DataInSending 标志为忙状态。数据发送完成后,就会触发事先设置的事件,从而唤醒写报告线程。写报告线程唤醒后就将数据正在发送的标志 DataInSending 清除,又进入到等待事件的状态。具体代码如下:

```
//写报告的线程
UINT WriteReportThread(LPVOID pParam)
{
 while(1)
 {
  //设置事件为无效状态
  ResetEvent(WriteOverlapped.hEvent);

  //等待事件触发
  WaitForSingleObject(WriteOverlapped.hEvent,INFINITE);

  //清除数据正在发送标志
  DataInSending = FALSE;

  //WriteReportBuffer[2]为非0值时将让设备清除它的计数值,
  //当点击清除计数器按键时,将会设置该值为非0,等数据发送完毕后,将它改回0
  //这样在发送LED状态时,就可以不用去设置WriteReportBuffer[2]的值了
  WriteReportBuffer[2] = 0;
 }
 return 0;
}
//////////////////////////////End of function//////////////////////
```


5.6.6 线程的创建以及设备插拔事件的注册

使用多线程时,需要在合适的时候创建好需要的线程。本程序在窗口显示时分别创建读报告线程和写报告线程,并同时将窗口的 this 指针作为参数传递给它们,让它们能够对窗口进行相关操作。这两个线程一旦被创建,就一直处于运行状态。因为里面是一个死循环,只有当程序退出时才被销毁。

另外,异步访问时需要给 ReadFile()和 WriteFile()函数提供一个 Overlapped 的结构体。该结构体在使用之前要先初始化,包括设置读数据的偏移量、事件的句柄等。通过调用创建事件的函数来获取一个事件的句柄,然后将其赋给 Overlapped 结构体中的事件句柄成员。本实例中分别给 ReadFile()和 WriteFile()函数创建了 Overlapped 结构体和对应的事件。

另外,当设备被插入或者拔下时,应用程序应该要能够感知到,并做出相关动作。要实现这个功能,就需要事先申请注册接收该设备类状态改变时产生的通知。注册设备状态改变通知使用的函数为 RegisterDeviceNotification()。注册成功后,当设备的状态发生改变(例如插、拔)时,事件的接收者(在注册时指定)就会收到一个 WM_DEVICECHANGE 的消息。

以上这些初始化处理工作都在窗口显示时完成,具体的代码如下:

```
//初始化写报告时用的Overlapped结构体
```

```
//偏移量设置为0
WriteOverlapped.Offset = 0;
WriteOverlapped.OffsetHigh = 0;
//创建一个事件,提供给WriteFile使用,当WriteFile完成时,会设置该事件为触发状态
WriteOverlapped.hEvent = CreateEvent(NULL,TRUE,FALSE,NULL);
//初始化读报告时用的Overlapped结构体偏移量设置为0
ReadOverlapped.Offset = 0;
ReadOverlapped.OffsetHigh = 0;
//创建一个事件,提供给ReadFile使用,当ReadFile完成时,会设置该事件为触发状态
ReadOverlapped.hEvent = CreateEvent(NULL,TRUE,FALSE,NULL);
//创建写报告的线程(处于挂起状态)
pWriteReportThread = AfxBeginThread(WriteReportThread,
                                    this,
                                    THREAD_PRIORITY_NORMAL,
                                    0,
                                    CREATE_SUSPENDED,
                                    NULL);
//如果创建成功,则恢复该线程的运行
if(pWriteReportThread! = NULL)
{
 pWriteReportThread->ResumeThread();
}
//创建一个读报告的线程(处于挂起状态)
pReadReportThread = AfxBeginThread(ReadReportThread,
                                   this,
                                   THREAD_PRIORITY_NORMAL,
                                   0,
                                   CREATE_SUSPENDED,
                                   NULL);
//如果创建成功,则恢复该线程的运行
if(pReadReportThread! = NULL)
{
 pReadReportThread->ResumeThread();
}
//获取HID设备的接口类GUDI
HidD_GetHidGuid(&HidGuid);
//设置DevBroadcastDeviceInterface结构体,用来注册设备改变时的通知
DevBroadcastDeviceInterface.dbcc_size = sizeof(DevBroadcastDeviceInterface);
```

```
DevBroadcastDeviceInterface.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
DevBroadcastDeviceInterface.dbcc_classguid = HidGuid;
//注册设备改变时收到通知
RegisterDeviceNotification(m_hWnd,
                           &DevBroadcastDeviceInterface,
                           DEVICE_NOTIFY_WINDOW_HANDLE);
```

5.6.7 对设备状态改变事件的处理

前面说过，成功注册了接收某类设备状态改变时的通知后，当设备状态改变时就会收到一个WM_DEVICECHANGE消息。在该消息处理中，可以获知设备的状态，例如设备插入并已经可用，设备已经拔下等。另外，还可以获知发生状态改变的设备的路径名。通过对比路径名，可以判断发生状态改变的设备是否是我们指定的设备。如果是，则做相关处理。例如，如果检测到指定的设备已经被拔出，则设置设备未找到，并关闭原来打开过的设备。具体的实现代码如下：


```
//设备状态改变时的处理函数
BOOL CMyUsbHidTestAppDlg::OnDeviceChange(UINT nEventType, DWORD dwData)
{
 PDEV_BROADCAST_DEVICEINTERFACE pdbi;
 CString DevPathName;

 //dwData是一个指向DEV_BROADCAST_DEVICEINTERFACE结构体的指针，
 //在该结构体中保存了设备的类型、路径名等参数。通过跟我们指定设备的路径名比较，
 //即可以判断是否是我们指定的设备拔下或者插入了
 pdbi = (PDEV_BROADCAST_DEVICEINTERFACE)dwData;

 switch(nEventType)                    //参数nEventType中保存着事件的类型
 {
  //设备连接事件
  case DBT_DEVICEARRIVAL:
   if(pdbi->dbcc_devicetype == DBT_DEVTYP_DEVICEINTERFACE)
   {
    DevPathName = pdbi->dbcc_name;      //保存发生状态改变的设备的路径名
    //比较是否是我们指定的设备
    if(MyDevPathName.CompareNoCase(DevPathName) == 0)
    {
     AddToInfOut("设备已连接");
    }
   }
   return TRUE;
```

```
//设备拔出事件
case DBT_DEVICEREMOVECOMPLETE:
 if(pdbi->dbcc_devicetype == DBT_DEVTYP_DEVICEINTERFACE)
 {
  DevPathName = pdbi->dbcc_name;        //保存发生状态改变的设备的路径名
  //比较是否是我们指定的设备
  if(MyDevPathName.CompareNoCase(DevPathName) == 0)
  {
   AddToInfOut("设备被拔出");
   //设备被拔出,应该关闭设备(如果处于打开状态的话),停止操作
   if(MyDevFound == TRUE)
   {
    MyDevFound = FALSE;
    OnCloseDevice();
   }
  }
 }
 return TRUE;
default:
 return TRUE;
}
}
//////////////////////////////End of function//////////////////////
```

5.7 软件界面以及使用方法

最终设计的软件界面如图5.7.1所示。当打开设备后,就可以使用鼠标点击LED按钮来控制学习板上LED的情况了(图中LED7被点亮)。下面一行则显示学习板上按键的情况,当某个键被按住时,对应的键就会变成红色。下方的信息框显示更详细的操作、数据、时间等信息。旁边的VID、PID、PVN分别对应着厂商ID、产品ID、产品版本号。也可以在这里填入另外一个HID设备的ID值,例如USB鼠标、USB键盘等,看能否找到。至于自己的HID设备的各种ID值如何去找,通过前面这么多内容的学习,应该知道了吧!圈圈知道至少有四种方法可以找到:设备管理器中、注册表中、本程序查找设备时、使用Bus Hound抓包分析。不过,由于USB鼠标、USB键盘是系统独占设备,所以该应用程序将无法打开它们。对于USB鼠标和键盘,虽然以写访问方式能够打开,但是在调用WriteFile()函数时也会出错,错误原因是不支持WriteFile()函数。

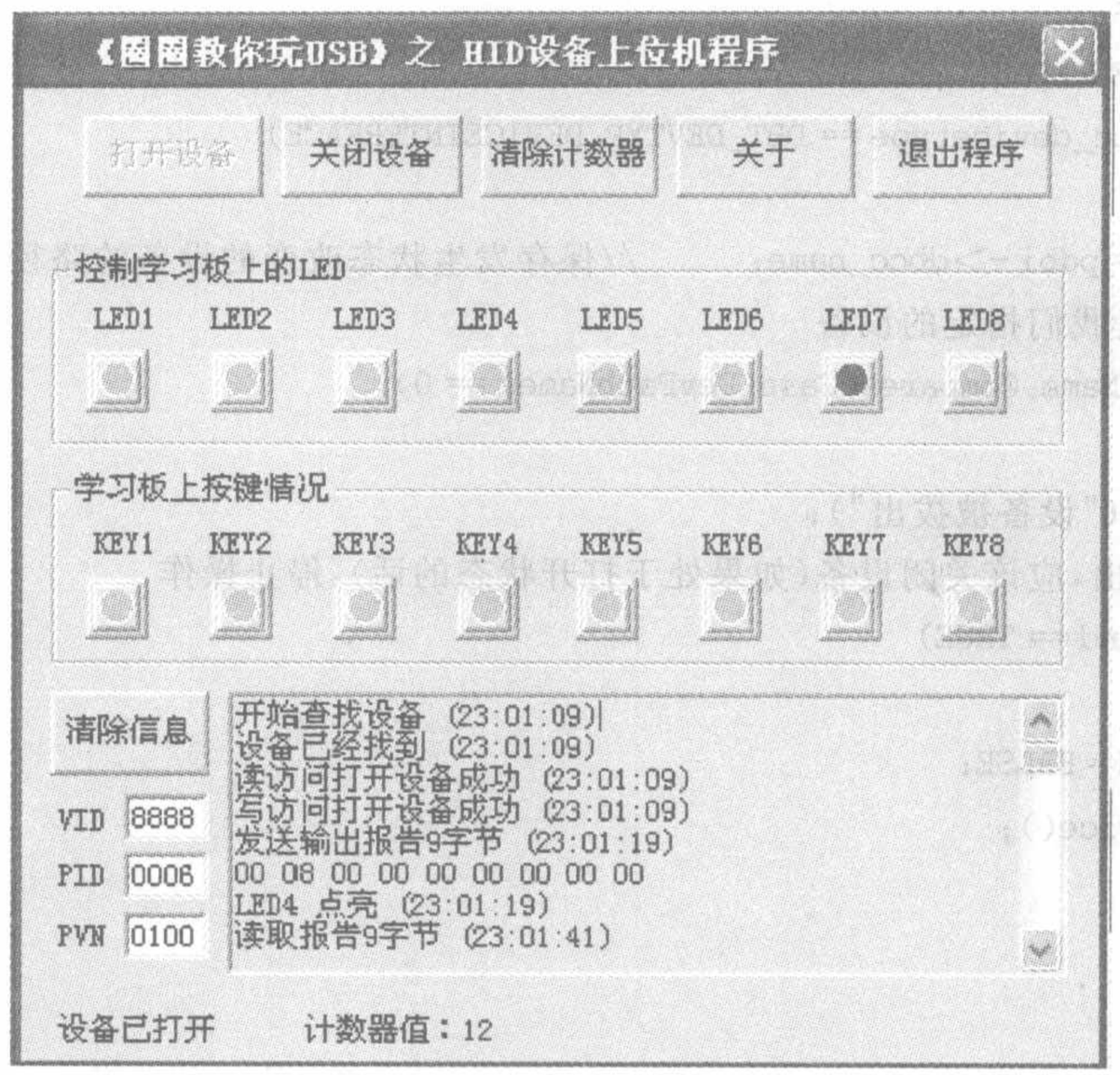


图 5.7.1 软件的界面图

5.8 本章小结

本章通过用户自定义的 USB HID 设备实例，介绍了用户自定义的 USB HID 设备的实现方法，并详细介绍了访问 HID 设备的上位机软件的设计方法及思路。本章内容以代码为主，因为这个操作思路是比较简单的，主要是需要知道一些函数的具体用法。如果要学会编写访问 USB 设备的上位机软件，这些 API 函数是必须要掌握的，至少要知道有这些函数。至于函数的具体参数、格式，问题不大，因为可以使用 MSDN 帮助文档或者上网搜索来找到。

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 descriptor,
//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“浏览”按钮，指定光盘中 USB 转

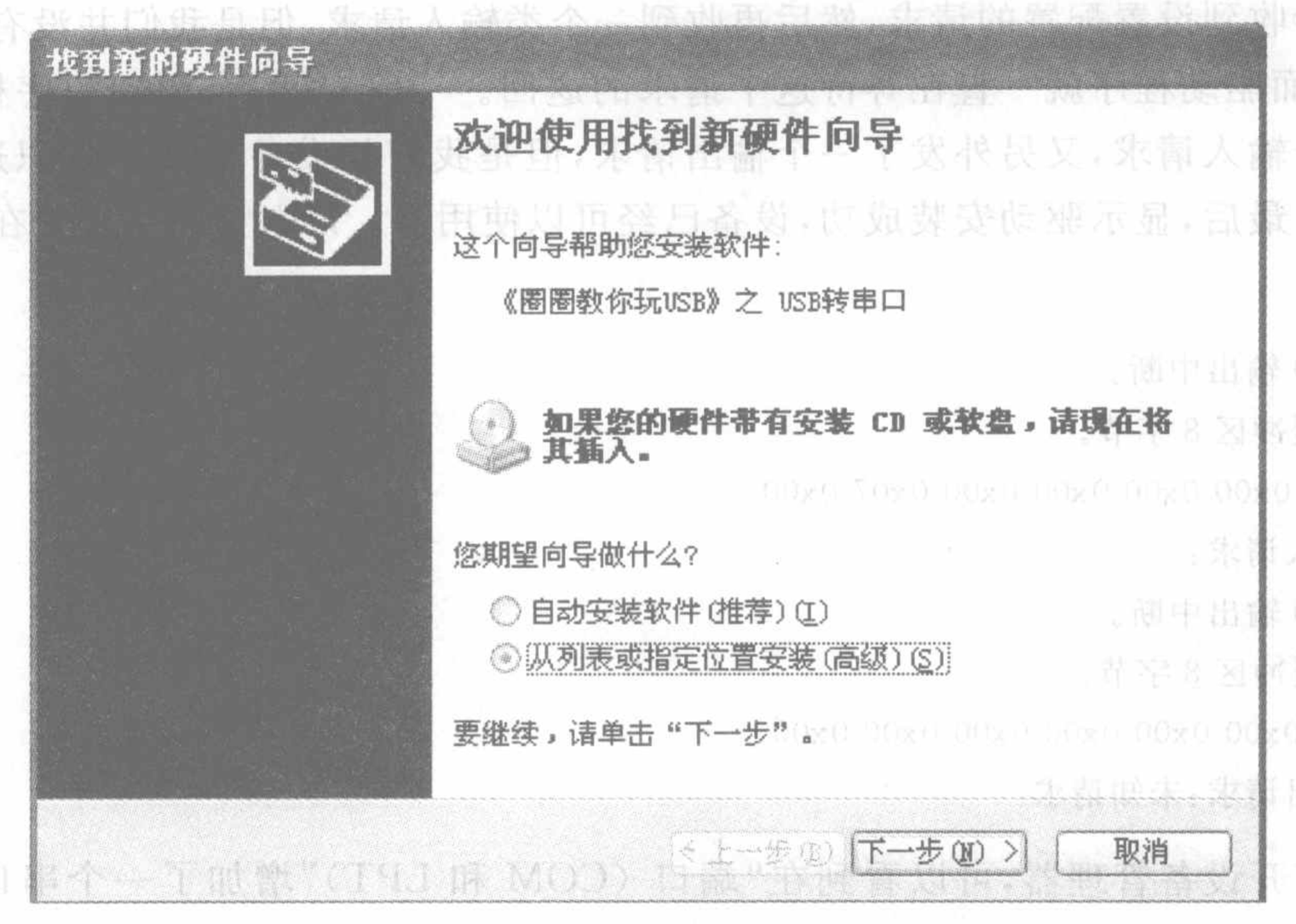


图 6.6.2　驱动安装步骤 1

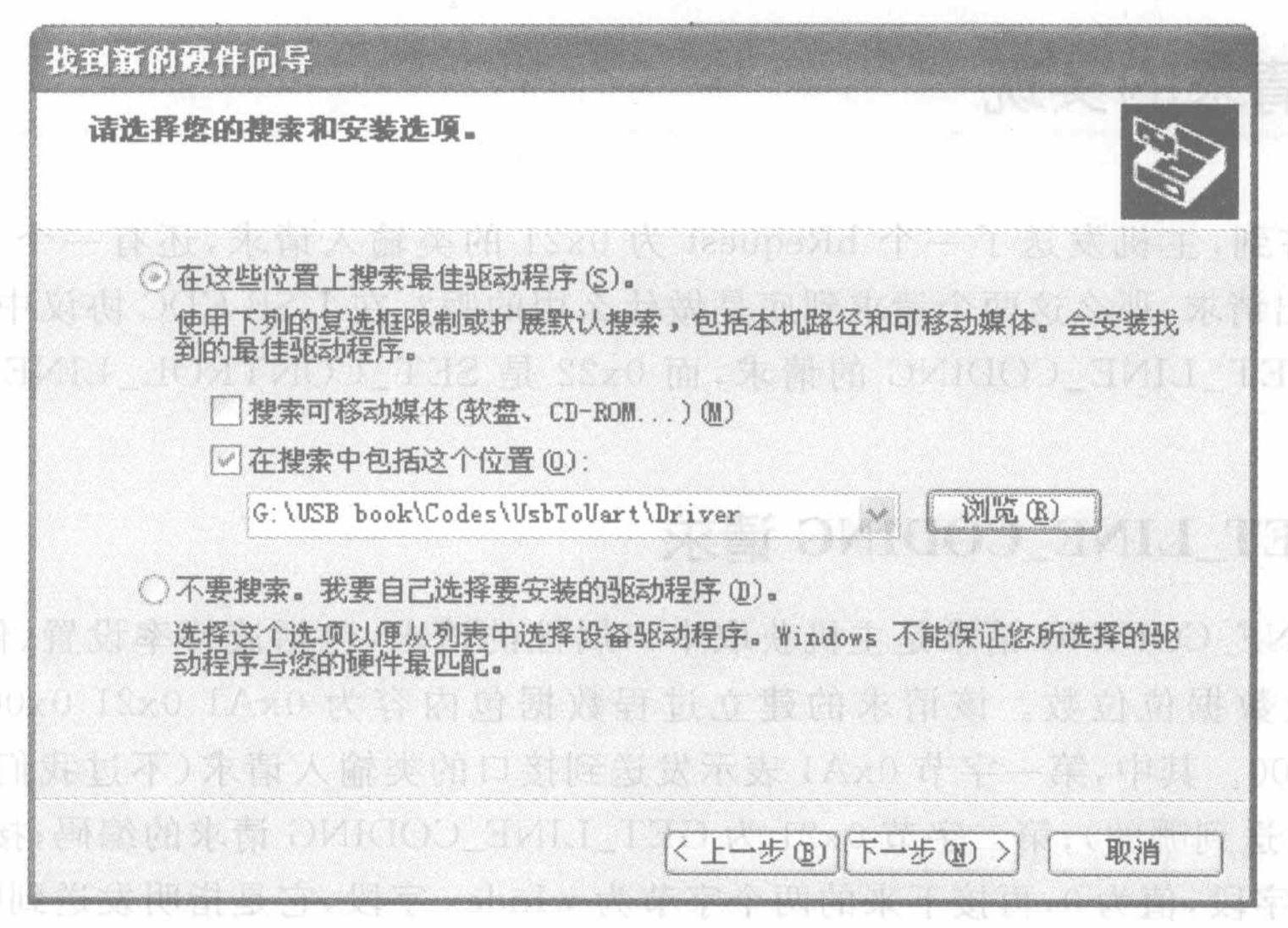


图 6.6.3　驱动安装步骤 2

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。

```
SSCOM3.2 (作者:聂小猛(丁丁), 主页http://www.mcu51.com,
USB端点0输出中断。
读端点0缓冲区8字节。
0x00 0x09 0x01 0x00 0x00 0x00 0x00 0x00
USB标准输出请求：设置配置。
写端点0缓冲区0字节。
USB端点0输入中断。
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：GET_LINE_CODING。
写端点0缓冲区7字节。
0x80 0x25 0x00 0x00 0x00 0x00 0x08
USB端点0输入中断。
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：SET_CONTROL_LINE_STATE。
写端点0缓冲区0字节。
USB端点0输入中断。
```

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。

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

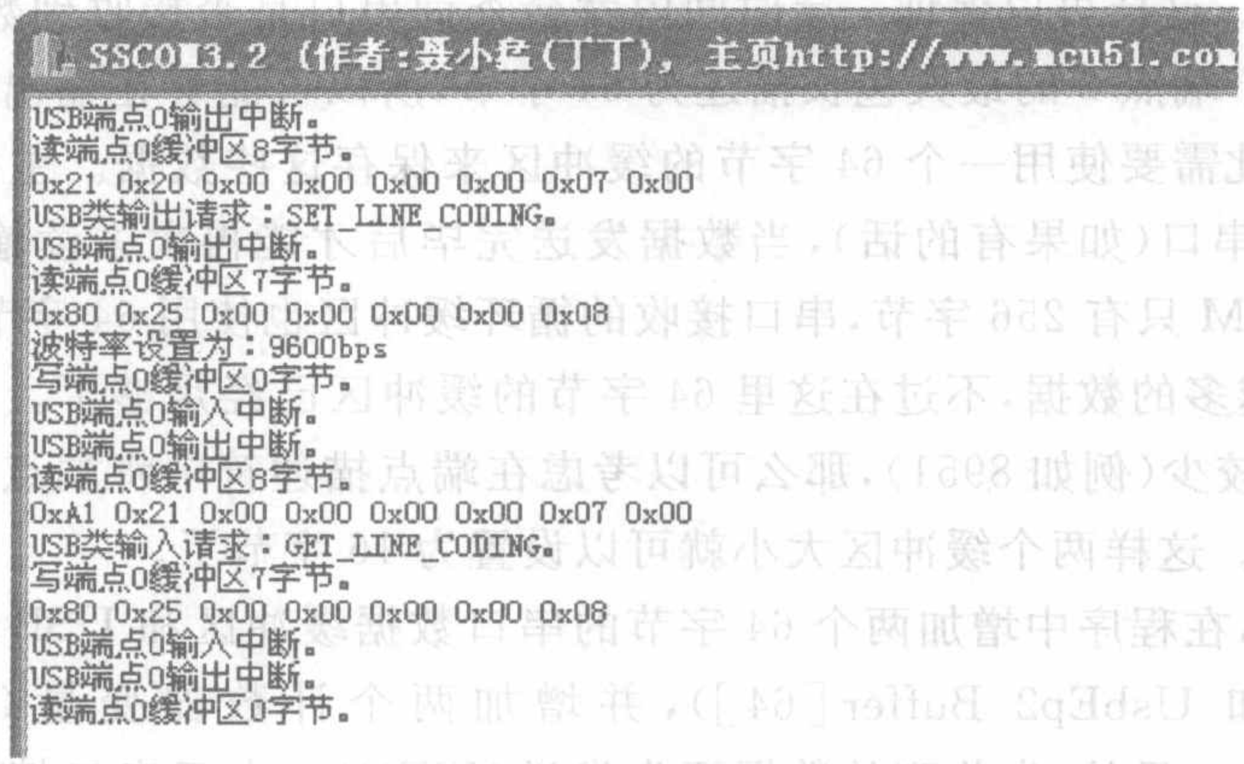


图 6.7.2 使用串口调试助手打开时的部分调试信息

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。

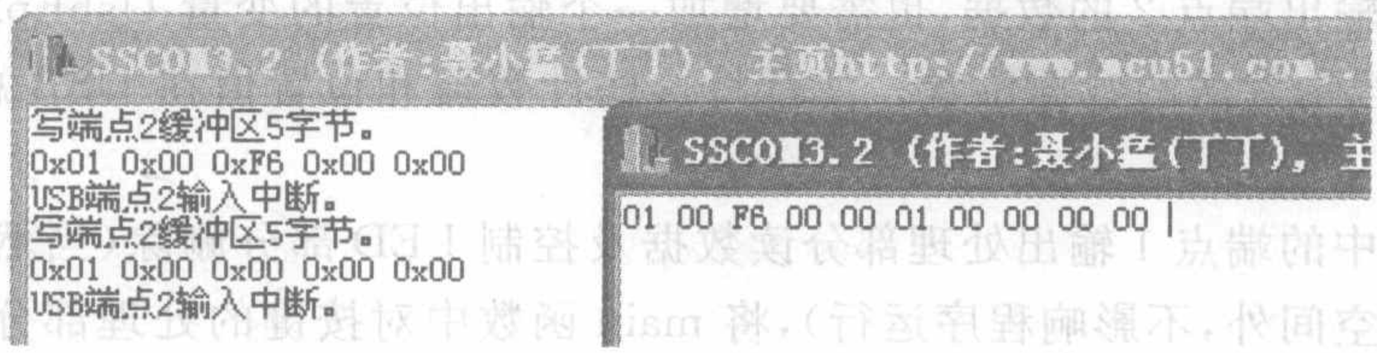


图 6.7.3 按下 KEY2 时的调试信息(左)和接收的数据(右)

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，判断两个缓冲区是否都满了。如果两个缓冲区都满了，那么就先不清除中断标志；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输出中断处理中修改原来的清除中断语句如下：

```
//读最后接收状态,这将清除端点2输出的中断标志位。
//注意端点2有个双缓冲机制,在清除中断之前,先检查两个缓冲区是否全满了,如果两个缓冲区
//全满了,就不用清除中断标志。只有当两个缓冲区不全满的时候才需要清除中断标志
if((D12ReadEndpointStatus(4)&0x60)!=0x60)
{
 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。

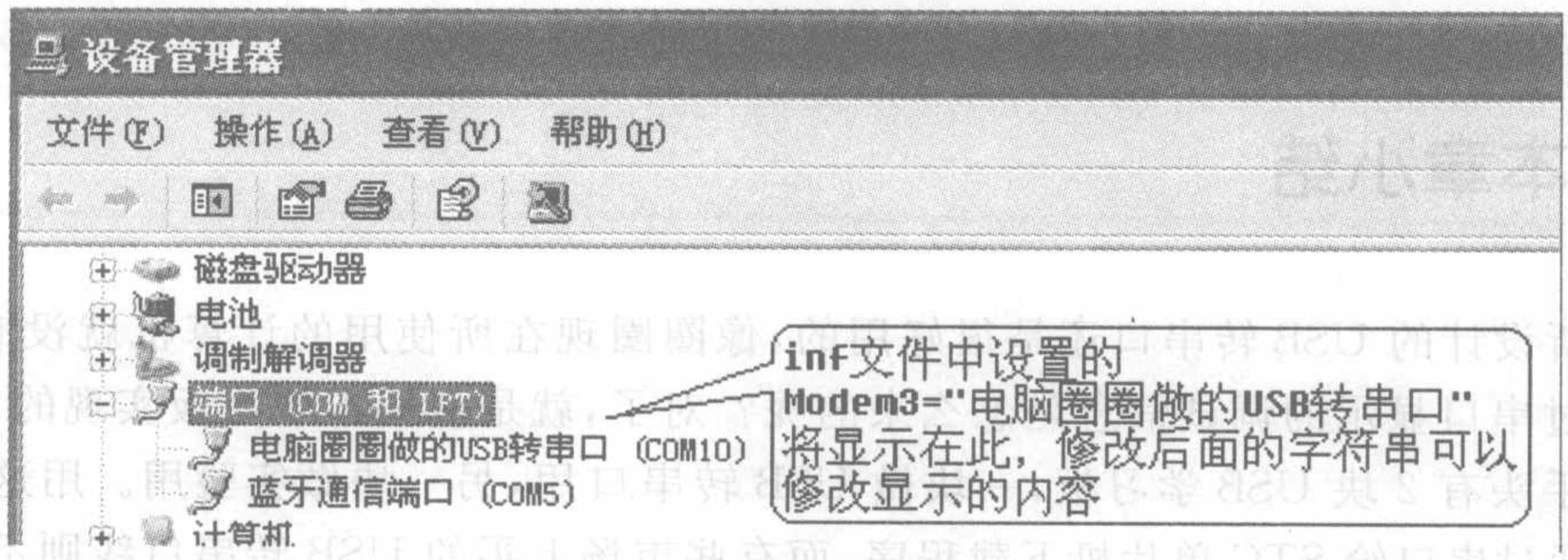


图 6.9.1　设备管理器中显示的设备名称

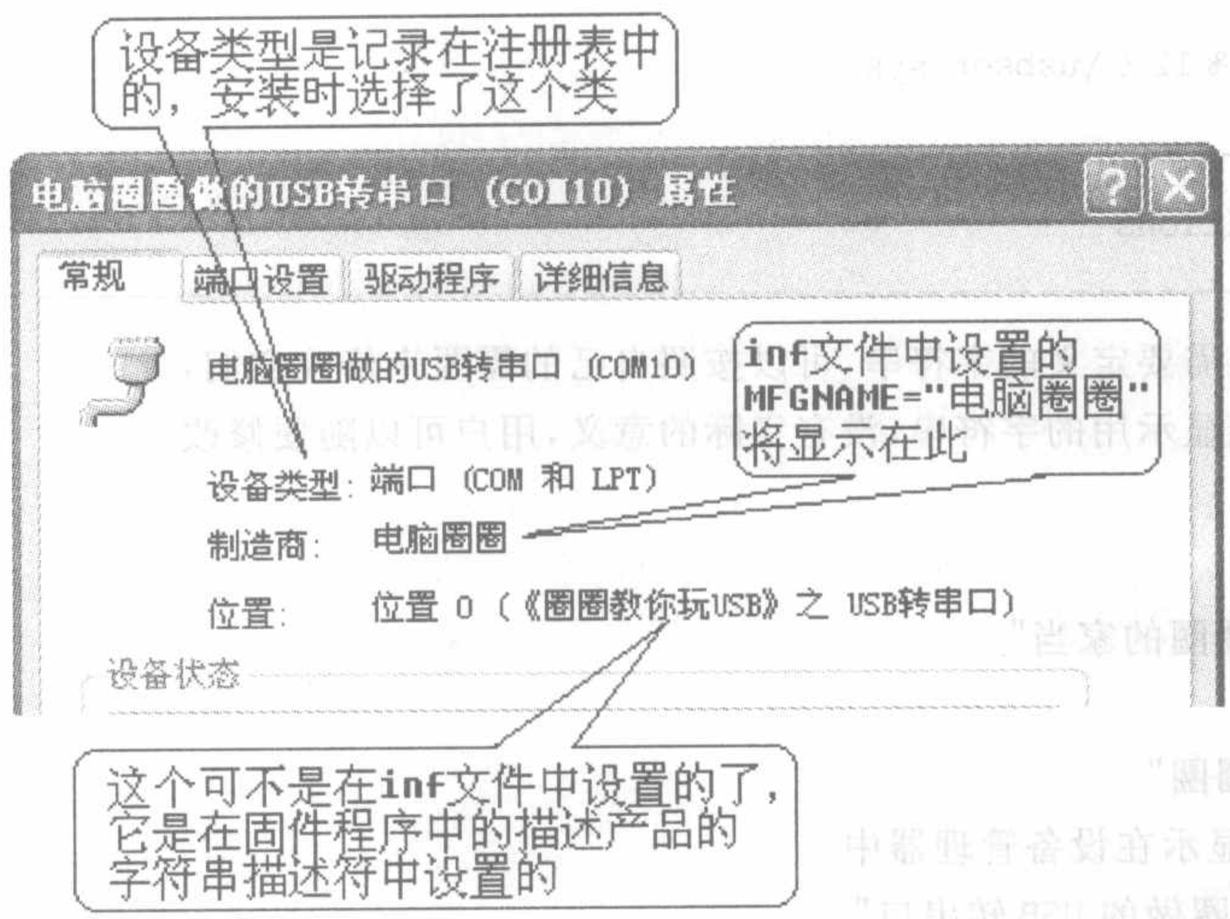


图 6.9.2　设备属性中的常规选项卡

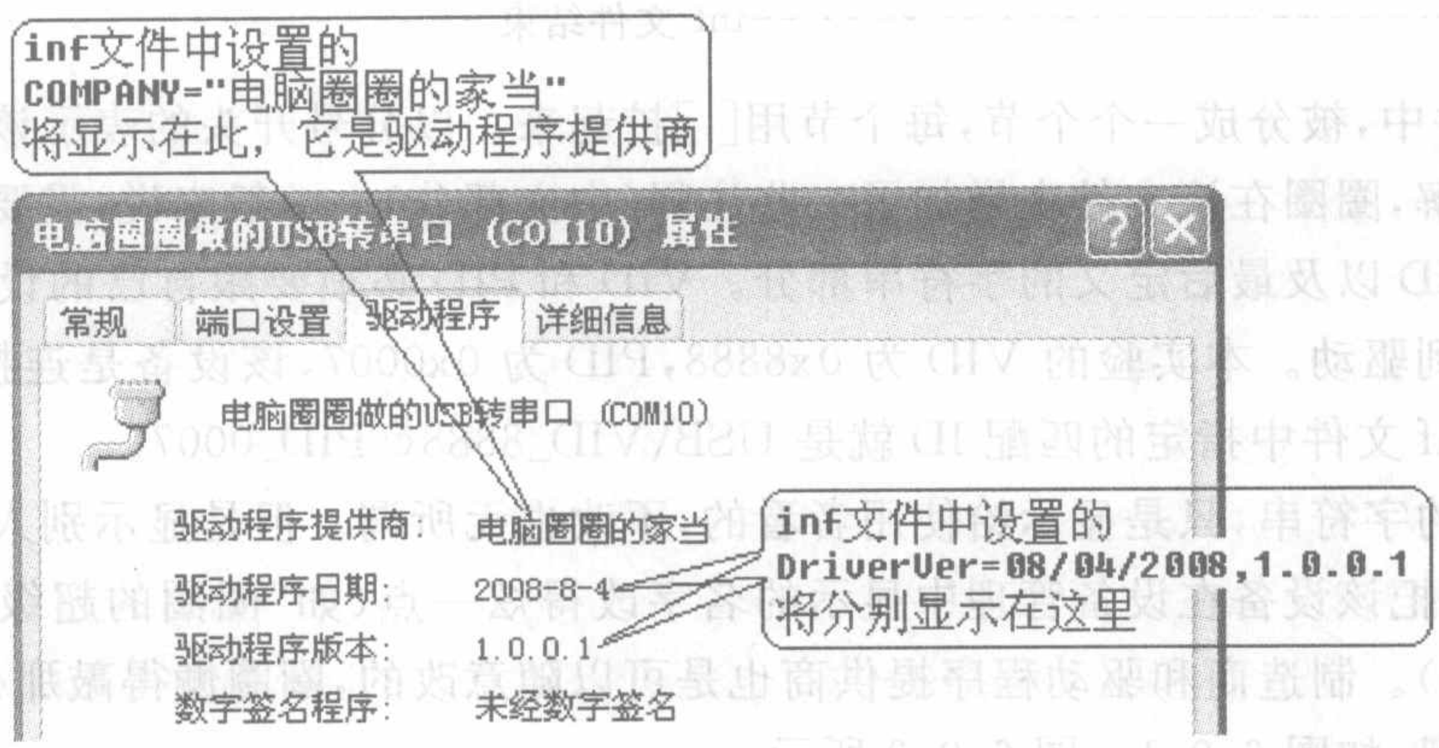


图 6.9.3　设备属性中的驱动程序选项卡

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。

第7章 USB MIDI 键盘

音乐在人们的生活中起着至关重要的作用，难以想象，如果没有了音乐，世界将会变成什么样子。本章将在 USB 学习板上实现一个简易的 USB MIDI 键盘，通过它可以弹奏曲子，它也可以自动弹奏。通过对程序的简单修改，还可以实现 MIDI 接口转 USB 接口的功能。

7.1 MIDI 简介

MIDI(Musical Instrument Digital Interface)是乐器数字接口的缩写，是国际 MIDI 协会开发的一种通用的标准。MIDI 传输和存储的并不是音频数据本身，而是演奏信息。这样可以大大节约存储空间，例如：一首 3 分钟的歌曲，如果保存为 CD 音质的 WAV 文件大概需要 30 MB 的存储空间，而同样的 MIDI 文件则只需要几十 KB。播放 MIDI 时，将 MIDI 控制信息交给 MIDI 合成器，由 MIDI 合成器将声音合成出来。

1983 年国际乐器制造商协会 NAMM(National Association of Music Merchants)正式通过了 MIDI 标准 1.0 版本。MIDI 的出现是现代乐器制造史上的一个里程碑，它标志着以电子计算机技术为基础的高性能电子乐器进入市场的开始。很快，许多厂商相继推出了具有MIDI 功能的电子乐器和相关外围设备，例如 Roland、YAMAHA 等乐器公司所生产的电子琴、鼓机、音序器等。

从 MIDI 出现到发展至今，具有 MIDI 接口的乐器得到了广泛的使用。在音乐的创作、制作，特别是在流行音乐领域，MIDI 乐器几乎无所不在，如 MIDI 键盘、MIDI 萨克斯管、MIDI 黑管等。

MIDI 音频与传统的音频概念上有着很大的区别。传统的音频录音，不管是模拟的还是数字的，都是直接将声音信息记录在介质上。而 MIDI 记录的并不是音频信息，而是演奏信息，如演奏的力度、音高等。在 MIDI 键盘上按下中央 C 音时，MIDI 键盘就可能会产生 3 字节的 MIDI 消息：90H、3CH、64H。其中，90H 表示在通道 0 上播放；3CH 表示音高，为中央 C；64H 表示力度。MIDI 合成器在收到该 MIDI 消息时，就会按照这些参数，以及选择好的音色来合成一个中央 C 的音。只要 MIDI 合成库的音色库里具有某种乐器，就可以使用 MIDI 键盘来演奏这种乐器，这足见 MIDI 的优点。由于 MIDI 存储的并不是音频数据本身，而是演奏

信息，这样可以大大节约存储空间，同样长度的音乐，MIDI文件大概只有WAV文件大小的千分之一左右。

电子琴、鼓机等通常都具有MIDI接口，可以与其他MIDI设备相连。标准的PC声卡上也具有MIDI接口（通常和游戏控制口在一起，即声卡背后的那个D型连接头），可以将电子琴、鼓机等MIDI设备与PC相连。利用计算机强大的处理能力，可以对MIDI进一步编辑和处理。对于电影、电视声音的后期制作，MIDI技术更显现出其得天独厚的优势。计算机上的MIDI音乐容易制作和编辑，演奏速度可以随意改变，即使不会乐器，也可以使用计算机制作出音乐来。

7.2 MIDI的工作原理

通常我们所听到的音乐都是乐队现场演奏，然后用录音技术直接录下来的。但是，直接录音需要巨大的数据量。以量化位数为16位，采样率为44.1 kHz的双声道CD来说，每秒产生的数据量为16×2×44.1/8=176.4 KB。那么能不能有一种办法，让计算机来“现场演奏”呢？这就是MIDI的基本思想，只保存演奏信息，然后由计算机“现场演奏”。对于USB MIDI键盘，就是通过USB接口发送弹奏信息给计算机，然后由计算机将声音合成出来。

音有几个基本的特征：音高、响度、音色和时值。只要这几个参数决定了，那么音也就确定了，MIDI信息正是由这些参数组成的。通常，MIDI合成器有16个不同的通道，可以给不同的通道（通道9（第十通道）是打击乐专用）选择不同的乐器（即音色）。演奏时，不同的设备发送各自的控制信息给不同的通道，控制信息包括通道编号、音高、力度。当需要让某个音停止发声时，就关闭它（通常是发一个力度为0的命令），从而控制了时值。

MIDI控制命令有很多，这里不详细讲解了，只讲一个程序中用到的，就是让指定通道发声的消息——Note On Message。该命令的格式（十六进制）为9n，kk，vv。其中，9表示该消息的ID；n表示发送到的通道，可以选择为0～15，共有16个通道；kk为音符的音高，60（十进制）为中央C音；vv表示演奏的力度，0为关闭声音，127为最大音量。

将MIDI信息转换为声音由MIDI合成器来完成。合成器通常使用波表（对实际乐器声音的波形采样然后建立的表格）来合成，它极大地决定了音色的好坏。相同的MIDI音源，在不同的合成器中所表现的效果可能会相差很远。像普通声卡所使用的软件波表合成器，延迟就会很明显，而使用高档一些的硬件波表合成声卡，则几乎感觉不到延迟。

7.3 USB MIDI设备的数据流模型

一个USB MIDI设备可以有很多模块，例如MIDI合成器、MIDI转换器（必须的）、插孔等。这里要设计的MIDI键盘没有合成器等模块，仅有转换器和MIDI插孔。这里的插孔是

一个抽象的概念，供MIDI信息输入输出使用；转换器的作用就是将各个插孔、模块连接起来。

USB MIDI设备使用批量端点来传输MIDI消息，一个批量端点可以传输多路MIDI消息。在USB批量端点传输的数据并不是原始的MIDI消息，而是由打包为32位的USB-MIDI事件包发送。具体可以参看USB MIDI设备类的文档，对于Note On消息，就是在前面加上0xP9，其中P为某一路MIDI消息的编号，本实例只有一路，设置为0即可。

本实例将在USB键盘的实例上进行修改，将UsbKeyboard复制一份，改名为UsbMidiKeyboard。USB MIDI设备在Windows XP下是自带驱动的，用户无须自行安装驱动，插上即可使用。

7.4 设备描述符

设备描述符只需要修改PID即可，其他内容保持不变。这已经是第八个程序了，将PID改成0x0008。

7.5 配置描述符集合

USB MIDI设备不需要HID描述符和报告描述符，将配置描述符集合当中的HID描述符删除，同时报告描述符、获取报告描述符的代码也一并删除。在USB MIDI设备中使用了MIDI流接口以及类特殊接口描述符和类特殊端点描述符，详细情况请看下面的介绍以及USB MIDI设备类的协议文档。

7.5.1 配置描述符

该USB MIDI键盘设备的配置需要两个接口，修改配置描述符中的bNumInterfaces字段为0x02。虽然在这里并没有用到音频控制接口，但是作为一个音频设备，音频控制接口是强制需要的。

7.5.2 音频控制接口描述符

每个音频设备都有一个音频控制接口AC(Audio Control)，可以有多个音频流接口或者MIDI流接口。在这里，音频控制接口没有实际的用途，仅是协议强制要求而已。所使用的端点数目为0，接口类为音频类(0x01)，子类为音频控制子类(0x01)，没有使用协议，为0值。修改好的音频控制接口描述符如下：

```
/*****************音频控制接口描述符*******************/
 //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。该接口没有端点
0x00,
//bInterfaceClass 字段。该接口所使用的类。音频接口类的代码为 0x01
0x01,
//bInterfaceSubClass 字段。该接口所使用的子类。音频控制接口的子类代码为 0x01
0x01,
//bInterfaceProtocol 字段。没有使用协议
0x00,
//iConfiguration 字段。该接口的字符串索引值。这里没有,为 0
0x00,
```

7.5.3 类特殊音频控制接口描述符

类特殊音频控制接口描述符用来描述该音频控制接口的属性,这里仅有一个头描述,它用来说明从属于该音频控制接口的其他接口有哪些。该描述符的结构如表 7.5.1 所列,其中取值一栏为本实例所使用的值,具体的代码请参看光盘中的源代码。这里只有一个流接口,并且它的编号为 1,所以在这里最后 2 字节的值都为 0x01。

表 7.5.1 类特殊音频控制接口描述符的结构

偏移量	域	大 小	取 值	描 述
0	bLength	1	0x09	该描述符的大小
1	bDescriptorType	1	0x24	描述符类型(CS_INTERFACE)
2	bDescriptorSubtype	1	0x01	描述符子类(MS_HEADER)
3	bcdADC	2	0x0100	该协议的版本号(1.0)
5	wTotalLength	2	0x0009	该描述的总长度
7	bInCollection	1	0x01	流接口的数目(只有一个)
8	baInterfaceNr(1)	1	0x01	哪个 MIDI 流接口属于该音频控制接口(这里为接口 1,即 MIDI 流接口)

7.5.4 MIDI流接口描述符

MIDI流接口描述符使用标准的接口描述符类型，它使用两个批量端点来分别输入和输出MIDI流数据。将前面的音频控制接口描述符复制一份修改为MIDI流接口描述符。这里它属于第二个接口，所以将bInterfaceNumber字段改为0x01。端点数目bNumEndpoints字段修改为0x02，类代码依旧为音频类，子类代码bInterfaceSubClass字段改为0x03，表示MIDI流接口子类。修改好的MIDI流接口描述符代码如下：

```
/*****************MIDI流接口描述符*******************/
//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。MIDI流接口使用一对批量输出/输出端点
0x02,

//bInterfaceClass字段。该接口所使用的类。音频接口类的代码为0x01
0x01,

//bInterfaceSubClass字段。该接口所使用的子类。MIDI流接口的子类代码为0x03
0x03,

//bInterfaceProtocol字段。没有使用协议
0x00,

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

7.5.5 类特殊MIDI流接口描述符

类特殊MIDI流接口描述符用来描述USB MIDI设备内各种模块(包括元件与插孔)间的连接关系。在USB MIDI设备中有4种插孔：内嵌输入插孔、内嵌输出插孔、外部输入插孔和外部输出插孔。注意：这里的输入和输出是针对设备来说的，与前面所说的端点方向刚好相反。

内嵌插孔是一种逻辑插孔，它是由USB端点产生的。一个USB批量输出端点可以设置一到多个内嵌输入插孔，USB发出的数据从该插孔流入USB MIDI设备中。一个USB批量

输入端点可以设置一到多个内嵌输出插孔，USB MIDI设备的输出数据流（例如从外部输入插孔上获取或者本身键盘产生）从该插孔流出到主机中。

外部插孔为实际的（物理上的）MIDI端子，可以用来连接外部的MIDI设备。事实上也可以为逻辑上的虚拟端子，例如，本例中虚拟的MIDI键盘输入插孔，但是主机并不知道这些细节，都把它们当作实际的物理插孔来看待。外部输入插孔用来接收其他MIDI设备发来的数据，外部输出插孔则将数据发送给其他MIDI设备。其实标准的MIDI接口所使用的传输协议很简单，它就是标准的UART串口，波特率为31.25 kb/s。

在USB MIDI设备中，每一个插孔都有一个唯一的ID号，用来标识它的身份。对于输出插孔，它可以具有多个输入引脚，用来选择哪些输入插孔作为其输入源。例如，要做的USB MIDI键盘，可以在内嵌输出插口（数据将输出给计算机）描述符中指定一个外部输入插孔作为其数据源；而如果要将计算机端的MIDI数据发送到实际的MIDI接口，则可以在一个外部输出插孔描述符中指定内嵌输入插孔作为其数据源。

输入插孔和输出插孔分别由MIDI输入插孔描述符（MIDI IN jack descriptor）和MIID输出插孔描述符（MIDI OUT jack descriptor）描述。此外，还有一个元件描述符（element descriptor），它用来描述USB MIDI设备的元件，例如MIDI合成器等（本实例没有用到）。所有的这些类特殊MIDI流接口描述符由一个头描述符来引导：类特殊MIDI流接口头描述符（class-specific MS interface header descriptor）。

下面将依次讲述这些描述符的结构以及具体的实现类特殊MIDI流接口头描述符的结构如表7.5.2所列。其中，wTotalLength为整个类特殊MIDI流接口描述符的总长度，视具体的描述符而定，该实例中为37字节。

表7.5.2 类特殊MIDI流接口头描述符的结构

偏移量	域	大 小	取 值	描 述
0	bLength	1	0x07	该描述符的字节数
1	bDescriptorType	1	0x24	该描述符的类型（CS_INTERFACE）
2	bDescriptorSubtype	1	0x01	该描述符子类（MS_HEADER）
3	bcdMSC	2	0x0100	MIDI流子类的协议版本号（1.0）
5	wTotalLength	2	0x0025	整个类特殊MIDI流接口描述符的总长度，包括头描述符、各种插孔描述符、元件描述符的长度总和

说明：取值一栏的数据为本实例所使用的数据，实际设计时根据需要设置。

MIDI输入插孔描述符的结构如表7.5.3所列。其中，bJackType字段为插孔的类型，可以选择内嵌（0x01）或者外部（0x02）。bJackID字段为插孔的唯一ID，可以用在输出插孔的输入源选择中。

表 7.5.3 MIDI 输入插孔描述符的结构

偏移量	域	大 小	取 值	描 述
0	bLength	1	0x06	该描述符的字节数
1	bDescriptorType	1	0x24	该描述符的类型(CS_INTERFACE)
2	bDescriptorSubtype	1	0x02	该描述符的子类(MIDI_IN_JACK)
3	bJackType	1	0x01 或 0x02	插孔的类型(EMBEDDED 或 EXTERNAL)
4	bJackID	1	ID	该插孔的唯一 ID
5	iJack	1	0x00	描述该插孔的字符串描述符的索引值

说明:Value 一栏的数据为本实例所使用的数据,实际设计时根据需要设置。

MIDI 输出插孔描述符的结构如表 7.5.4 所列。输出插孔描述符和输入插孔描述符的前 5 字节内容意义是一样的。bNrInputPins 字段为该输出插孔的输入引脚数量,一个输出插孔可以有多个输入引脚,从而可以把多个输入插孔连接到输出插孔,以提供输出数据。baSourceID 为连接到该输出插孔的输入插孔的 ID 号;BaSourcePin 为输入源(上述输入插孔)连接在该输出插孔的输入引脚号。中间省略号部分表示可以有多组 baSourceID 和 BaSourcePin。iJack 为描述该插孔的字符串描述符的索引,本实例中没有该字符串。

表 7.5.4 MIDI 输出插孔描述符的结构

偏移量	域	大 小	取 值	描 述
0	bLength	1	长度	该描述符的长度
1	bDescriptorType	1	0x24	该描述符的类型(CS_INTERFACE)
2	bDescriptorSubtype	1	0x03	该描述符的子类(MIDI_OUT_JACK)
3	bJackType	1	0x01 或 0x02	插孔的类型(EMBEDDED 或 EXTERNAL)
4	bJackID	1	ID	该插孔的唯一 ID
5	bNrInputPins	1	n	该输出插孔的输入引脚数
6	baSourceID(1)	1	ID	连接到该输出插孔的输入引脚的输入插孔的 ID 号
7	BaSourcePin(1)	1	引脚	连接到该输出插孔的那个输入引脚
⋮	⋮	⋮	⋮	⋮
	iJack	1	0x00	描述该插孔的字符串描述符的索引值

Element Descriptor 描述符在本实例中没有用到,这里就不详述了,感兴趣的读者可以查看 USB MIDI 设备协议。

本实例设置了 4 个插孔,分别为内嵌输入插孔、内嵌输出插孔、外部输入插孔和外部输出插孔。键盘的数据通过内嵌输出插孔输出给计算机,但是键盘的数据不能直接发送到内嵌输

出插孔，而是假设有一个外部输入插孔（虚拟的），将该输入插孔连接到内嵌输出插孔的输入引脚上。对于计算机端输出的数据也相类似，数据将到达内嵌输入插孔，将内嵌输入插孔连接到外部输出插孔的输入引脚上，数据就可以发送出去了。这里指定内嵌输入插孔的ID为1，外部输入插孔的ID为2，内嵌输出插孔的ID为3，外部输出插孔的ID为4。在内嵌输出插孔的baSourceID中指定ID为2，即外部输入插孔。在外部输出插孔的baSourceID中指定ID为1，即内嵌输入插孔。最终设置好的类特殊MIDI流接口描述符如下：

```
/**********内嵌输入插孔描述符********/
//bLength字段。该描述符的长度，为6字节
0x06,

//bDescriptorType字段。该描述符的类型为CS_INTERFACE
0x24,

//bDescriptorSubtype字段。描述符子类，为MIDI_IN_JACK
0x02,

//bJackType字段。该插孔的类型为内嵌
0x01,

//bJackID字段。该插孔的唯一ID，这里取值1
0x01,

//iJack字段。该插孔的字符串描述符索引，这里没有，为0
0x00,

/**********外部输入插孔描述符********/
//bLength字段。该描述符的长度，为6字节
0x06,

//bDescriptorType字段。该描述符的类型为CS_INTERFACE
0x24,

//bDescriptorSubtype字段。描述符子类，为MIDI_IN_JACK
0x02,

//bJackType字段。该插孔的类型为外部
0x02,

//bJackID字段。该插孔的唯一ID，这里取值2
0x02,

//iJack字段。该插孔的字符串描述符索引，这里没有，为0
0x00,

/**********内嵌输出插孔描述符********/
//bLength字段。该描述符的长度为9字节
```

```
0x09,
//bDescriptorType 字段。该描述符的类型为 CS_INTERFACE
0x24,
//bDescriptorSubtype 字段。描述符子类为 MIDI_OUT_JACK
0x03,
//bJackType 字段。该插孔的类型为内嵌
0x01,
//bJackID 字段。该插孔的唯一 ID,这里取值 3
0x03,
//bNrInputPins 字段。该输出插孔的输入引脚数。这里仅有一个
0x01,
//baSourceID 字段。连接到该插孔输入引脚的输入插孔的 ID,选择为外部输入插孔
0x02,
//BaSourcePin 字段。外部输入插孔连接到该插孔的输入引脚 1 上
0x01,
//iJack 字段。该插孔的字符串描述符索引,这里没有,为 0
0x00,
/**********外部输出插孔描述符********/
//bLength 字段。该描述符的长度,为 9 字节
0x09,
//bDescriptorType 字段。该描述符的类型为 CS_INTERFACE
0x24,
//bDescriptorSubtype 字段。描述符子类,为 MIDI_OUT_JACK
0x03,
//bJackType 字段。该插孔的类型为外部
0x02,
//bJackID 字段。该插孔的唯一 ID,这里取值 4
0x04,
//bNrInputPins 字段。该输出插孔的输入引脚数。这里仅有一个
0x01,
//baSourceID 字段。连接到该插孔输入引脚的输入插孔的 ID,选择为内嵌输入插孔
0x01,
//BaSourcePin 字段。内嵌输入插孔连接到该插孔的输入引脚 1 上
0x01,
```

```
//iJack 字段。该插孔的字符串描述符索引,这里没有,为 0
0x00,
```

7.5.6 端点描述符和类特殊端点描述符

在 USB MIDI 设备中,除了有标准的批量数据端点描述符之外,还有类特殊 MIDI 流批量数据端点描述符(class-specific MS bulk data endpoint descriptor),它们用来描述内嵌插孔是如何在端点上组织的。每个标准的批量数据端点描述符后面跟一个类特殊批量数据端点描述符。

类特殊 MIDI 流批量数据端点描述符的结构如表 7.5.5 所列。其中 bNumEmbMIDIJack 字段为分配在该批量端点的内嵌插孔的数量,在本实例中,仅有一个内嵌输入插孔或一个内嵌输出插孔,所以值为 1。baAssocJackID 字段为分配在此批量端点的内嵌插孔的 ID 号,可以有多个,视该端点内嵌插孔的数量而定;对于输入端点,指定为内嵌输出插孔的 ID 号,对于输出端点,指定为内嵌输入插孔的 ID 号。

表 7.5.5 类特殊 MIDI 流批量数据端点描述符的结构

偏移量	域	大 小	取 值	描 述
0	bLength	1	4+n	该描述符的长度
1	bDescriptorType	1	0x25	该描述符的类型(CS_ENDPOINT)
2	bDescriptorSubType	1	0x01	该描述符的子类(MS_GENERAL)
3	bNumEmbMIDIJack	1	n	分配在此端点的内嵌插孔的数量
4	baAssocJackID(1)	1	ID1	分配在此端点的第一个内嵌插孔的 ID 号
⋮	⋮	⋮	⋮	⋮
4+(n-1)	baAssocJackID(n)	1	IDn	分配在此端点的最后一个内嵌插孔的 ID 号

在本实例中,需要一个批量输入端点和一个批量输出端点,这里使用 D12 的端点 2 来实现。在每个标准端点描述符下增加一个类特殊端点描述符,最终实现的代码如下:

```
/**************标准批量数据输入端点描述符****************/
//bLength 字段。端点描述符长度为 7 字节
0x07,

//bDescriptorType 字段。端点描述符的编号为 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。该端点为批端点
//批量端点的编号为 2,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,此处无意义
0x00,

/**********类特殊 MIDI 流批量数据端点描述符**************/
//bLength 字段,该描述符的长度为 5 字节
0x05,

//bDescriptorType 字段,该描述符的类型为类特殊端点描述符(CS_ENDPOINT)
0x25,

//bDescriptorSubType 字段,该描述符的子类型为 MS_GENERAL
0x01,

//bNumEmbMIDIJack 字段,该端点的内嵌输出插孔的数量,这里只有 1 个
0x01,

//baAssocJackID 字段,该端点的内嵌输出插孔的 ID 号
//在前面定义了一个内嵌输出插孔,ID 号为 3
0x03,

/************标准批量数据输出端点描述符**************/
//bLength 字段。端点描述符长度为 7 字节
0x07,

//bDescriptorType 字段。端点描述符的编号为 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。该端点为批端点
//批量端点的编号为 2,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,此处无意义
```

```
0x00,

/**********类特殊MIDI流批量数据端点描述符********/
//bLength字段,该描述符的长度为5字节
0x05,

//bDescriptorType字段,该描述符的类型为类特殊端点描述符(CS_ENDPOINT)
0x25,

//bDescriptorSubType字段,该描述符的子类型为MS_GENERAL
0x01,

//bNumEmbMIDIJack字段,该端点的内嵌输入插孔的数量,这里只有1个
0x01,

//baAssocJackID字段,该端点的内嵌输入插孔的ID号。在前面定义了一个内嵌输入插孔,ID号为1
0x01
```

7.5.7 字符串描述符

字符串描述符可以按照自己的需要和喜好来修改,这里将产品字符串改为"《圈圈教你玩USB》之USB MIDI键盘",设备序列号改为"2008-08-08"(2008北京奥运会开幕式时间)。

7.6 修改好描述符后的测试

实际上,在USB MIDI协议中还规定了一些类特殊请求,但是在该设备中并不会使用到,因而也就没去实现它们。目前我们的程序仅完成了描述符的修改,对具体的MIDI数据处理还没有实现,但是可以先测试一下,描述符是否修改正确了。如果正确的话,就会显示出一个USB音频设备。

将修改好的程序编译,并下载到学习板中运行。可以看到弹出发现新硬件的对话框,如图7.6.1所示。等设备的驱动程序安装完成后,在设备管理器中可以看到一个USB Composite Device(USB复合设备)和一个USB Audio Device(USB音频设备),如图7.6.2所示。剩下的工作就是对两个批量端点的处理了。

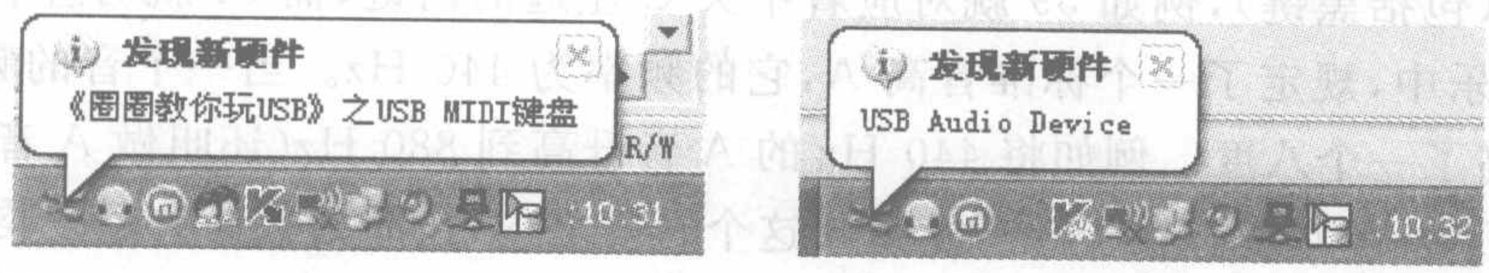


图7.6.1 发现新硬件对话框

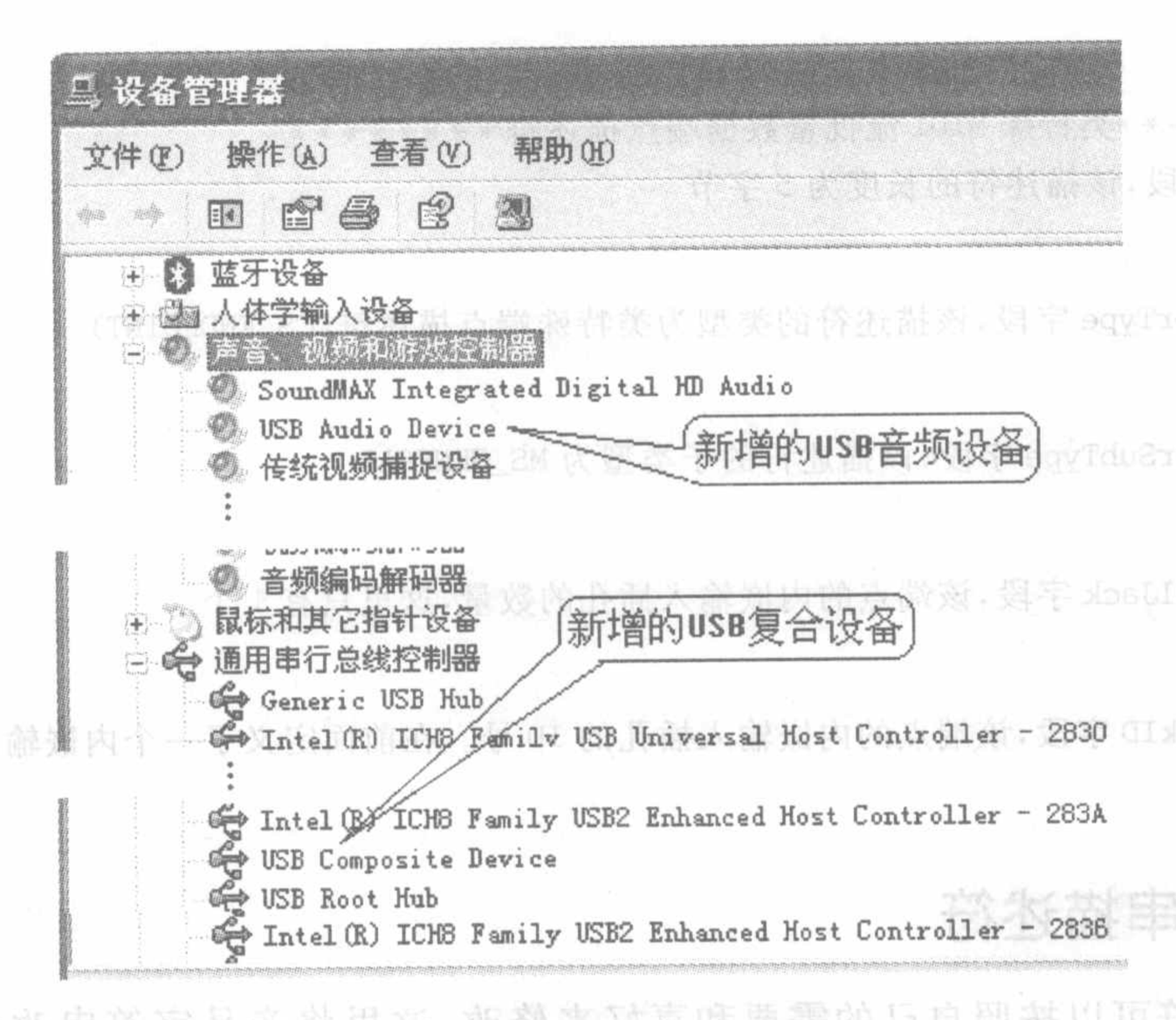


图 7.6.2 设备管理器

7.7 USB MIDI 键盘的数据返回

对于学习板上按键的按下，如何转换为 MIDI 消息发送到 USB 输入端点中去呢？在前面有提到过，要打包为 32 位的 MIDI 事件包发送。本 USB MIDI 键盘功能比较简单，仅能产生一条 Note On(音符开)消息，它的 32 位 MIDI 事件包格式(十六进制)为：P9，9n，kk，vv。其中，第一字节 P9 为 USB MIDI 协议中增加的包头，P 为某一路 MIDI 消息的编号(这里仅有一个内嵌输出插孔，为 0)，9 为该包的 ID 标识。后面的 3 字节为实际的 MIDI 消息，9n 表示在通道 n 上发送 Note On 消息，kk 表示音符的音高，vv 表示音符的力度(响度，127 为最大声)。

通道和力度容易理解，就这个音符的音高的值稍微麻烦点。在钢琴上，有个中央 C 键，而这里的音高值为 60 时，对应的就是钢琴上的中央 C 键。MIDI 消息中音高的每个值对应着钢琴上的一个键(包括黑键)，例如 59 就对应着中央 C 左边的白键，而 61 就对应着中央 C 右边的黑键。在音乐中，规定了一个标准音高 A，它的频率为 440 Hz。当一个音的频率升高一倍时，就叫做升高了一个八度。例如将 440 Hz 的 A 音升高到 880 Hz(还叫做 A 音，但用后缀来区分)，就升高了一个八度。在 440～880 Hz 这个八度之内，按照指数的关系平均分成 12 等份(每个音的频率比它前一个音的频率高 2 的 1/12 次方，即 12 平均律)，就得到了 12 个频率的音，分别记做 A、#A、B、C、#C、D、#D、E、F、#F、G、#G。其中带有#号的就是钢琴中的黑键，没有#号的就是钢琴上的白键。相差一个八度的音，所使用的字母是一样的，但使用大小

写以及带后缀来区分。钢琴上的键就是按照上述的关系排布的，MIDI中的音高值也是。像这样的两个相邻的音之间的频率差距，叫做一个半音，也叫做一个小二度。例如B、C之间相差一个半音，C、#C之间相差一个半音。距离为两个半音叫做一个全音（或者叫大二度），例如C、#A之间相差一个全音，C、D之间相差一个全音。

但是在音乐中常使用简谱来记谱，那么简谱跟刚刚所说的这些音之间有什么关系呢？简谱使用1～7这7个阿拉伯数字来表示，其中，3和4、7和1之间相差一个半音，其他相邻音之间相差为一个全音。为了表示全音中间的半音，可以增加#号来表示。这样在一个八度中，简谱的12个音就是：1、#1、2、#2、3、4、#4、5、#5、6、#6、7。那么1的频率是多少呢？1的频率是不固定的，可以在前面用字母表示的音符中任选一个作为1。这样的表示叫做相对音高，而用字母表示的则是绝对音高。在简谱中，不带#号的7个音叫做自然音阶，大部分歌曲中只使用自然音阶（另外像五声调式中，只有五个音，没有4和7）。如果一个曲以1为主音，那么它就叫做大调，如果以6为主音，那么就叫做小调。大调给人明亮、欢快的感觉，而小调则给人压抑、忧伤的感觉。如果一个大调的歌曲选择C作为1，就叫做C大调，选择G作为1，就叫做G大调。通过对照可以发现，如果选择C作为1，那么所有的自然音阶都刚好落在钢琴的白键上，这就是为什么很多曲子是C大调（或者A小调，选择A为6）的原因。

本USB学习板仅有8个键，为了能够演奏一些简单的乐曲，将其按照五声调式来排布，选择为C调。KEY1～KEY8分别为简谱的5、6、1、2、3、5、6、1，第一个1为中央C，按照前面介绍的关系，可以计算出它们在MIDI消息中的音高值分别为55、57、60、62、64、67、69、72。

将原来返回报告的函数SendReport（改为SendNoteOnMsg），在该函数中根据不同按键的按下和弹起来发送Note On消息。当某个键按下时，就发送力度值为最大的Note On消息（由于本学习板的按键没有力度检测，只好发送最大力度了，在高档的电子琴中按键有力度检测，可以根据具体的弹奏力度来设置该消息），这将让某个音符发声；当某个键弹起时，就发送力度为0的消息，这将停止该音符的发声。由于8个按键的处理非常类似，这里只贴出KEY1的按下和弹起处理，其他部分省略。

```
/*****************************************************************
函数功能：根据按键情况返回Note On消息的函数
入口参数：无
返　　回：无
备　　注：无
*****************************************************************/
void SendNoteOnMsg(void)
{
 //4字节的缓冲区
 uint8 Buf[4];

 //Note On消息第一字节固定为0x09，第二字节为0x9n（n为通道号），第三字节为0xKK（K为音高），第
```

四字节为0xVV(V为力度)

```
Buf[0] = 0x09;                  //Note On 消息的包头
Buf[1] = 0x90;                  //在通道 0 上发送 Note On 消息
Buf[3] = 0x7F;                  //音量设置为最大

if(KeyDown&KEY1)
{
 Buf[2] = 55;                   //C 调的 5(绝对音高为 G 音)
 //通过端点 2 返回 4 字节 MIDI 事件包
 D12WriteEndpointBuffer(5,4,Buf);
 Ep2InIsBusy = 1;               //设置端点忙标志
 KeyDown& = ~KEY1;              //清除对应的按键
 return;                        //发送一个音符后就返回
}
⋮                               //此处省略部分代码
//如果有按键弹起,则关闭对应的音
Buf[3] = 0x00;                  //音量设置为 0

if(KeyUp&KEY1)
{
 Buf[2] = 55;                   //C 调的 5(绝对音高为 G 音)
 //通过端点 2 返回 4 字节 MIDI 事件包
 D12WriteEndpointBuffer(5,4,Buf);
 Ep2InIsBusy = 1;               //设置端点忙标志
 KeyUp& = ~KEY1;                //清除对应的按键
 return;                        //发送一个音符后就返回
}
⋮                               //此处省略部分代码
```


对于输出数据,由于本实验板无法设置为31.25 kb/s的波特率,所以对于端点2输出的数据就直接丢弃了,仅在端点2输出中断处理中清除中断标志和清空缓冲区。需要注意的是,对于输出数据的处理速度一定要够,否则可能会导致应用软件停止响应甚至整个操作系统崩溃。如果设备用不到MIDI输出,则干脆在外部输出插孔描述符中修改baSourceID字段为0x02,选择输入源为外部输入插孔。这样内嵌输入插孔就没有被使用,从而Windows就不会增加MIDI输出设备。正常使用时,将config.h中定义的调试宏删除,避免多余的消耗。

7.8 USB MIDI键盘的使用

这个USB MIDI键盘怎么使用呢?有很多软件是支持MIDI输入/输出设备的,比较有名

的如 Cakewalk、Guitar Pro 等。在这里，介绍一款比较小巧的软件：HappyEO 电子琴软件，该软件可以在网上或者圈圈的 USB 小组中下载。

该 USB MIDI 键盘连接到计算机后，会产生一个 MIDI 输出设备，Windows 操作系统有时会自动将它选择为“MIDI 音乐播放”的默认设备。如果此时播放一个 MIDI 文件，数据将发送到 USB MIDI 设备去，从而计算机声卡无声音输出。可以进入控制面板的“声音和音频设备”中选择需要的 MIDI 设备，如图 7.8.1 所示。对于一般的声卡，都是使用微软的 GS 波表软件合成器，选择它就可以了。如果声卡比较高档，有硬件合成器，当然也可以选它们。如果想观察播放一个 MIDI 文件时到底发了些什么数据到 USB MIDI 设备中，可以选择下面的 USB 音频设备，然后打开 bus hound，选择捕捉该设备的数据。这些数据都是一些 MIDI 消息，需要了解 MIDI 协议才能看懂它的含义，其中最多的就是 Note On 消息和 Note Off 消息了。如果不想改这个 MIDI 音乐播放的默认设备，可以参看 7.7 节的最后一段。

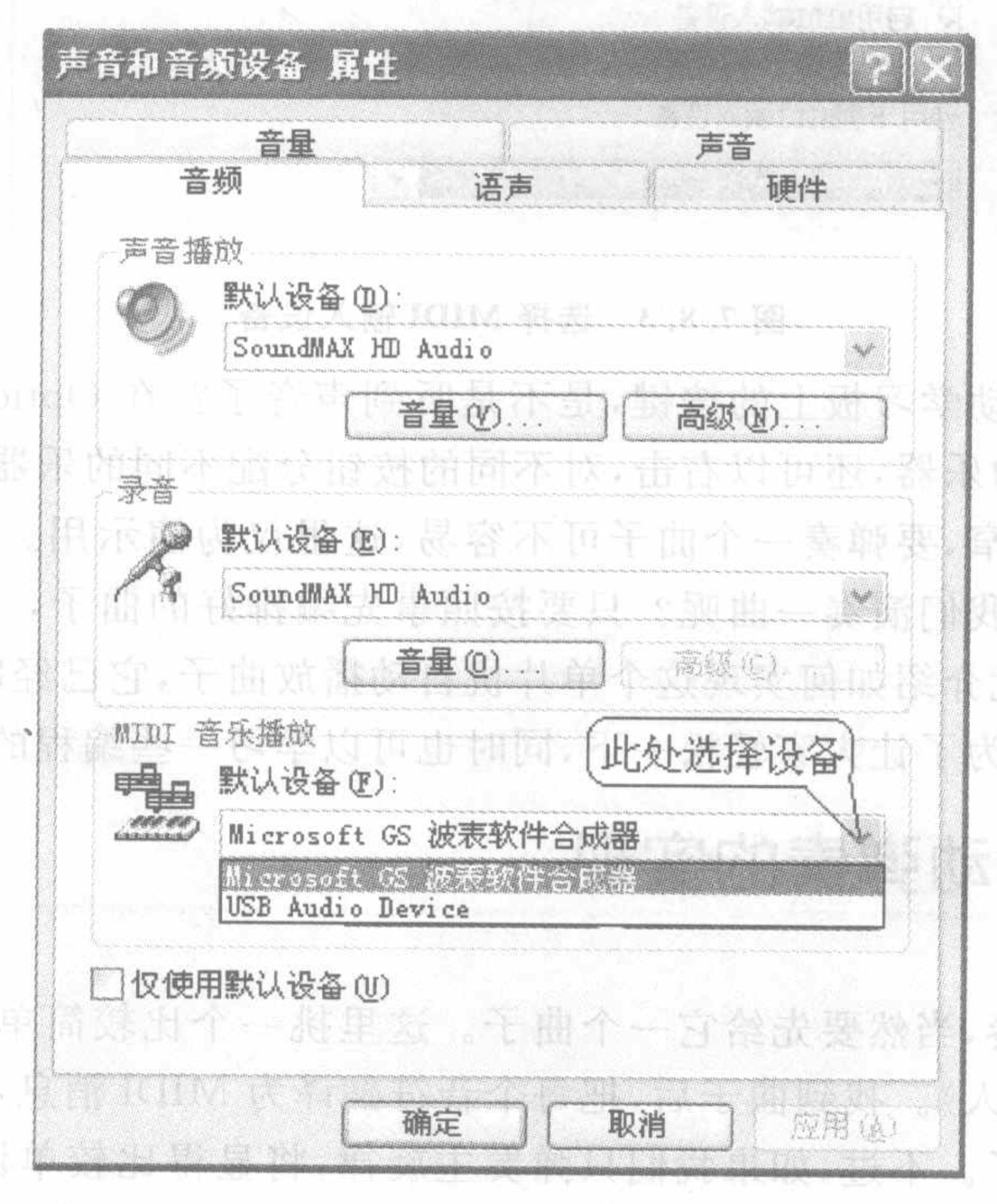


图 7.8.1 选择 MIDI 音乐播放的默认设备

要在 HappyEO 电子琴软件中使用该设备，先需要对软件做一些设置。运行 HappyEO，单击 Option 按钮，在“常用”选项卡中，选择一个 MIDI 输出设备，这里要选择一个能够发声的 MIDI 设备，所以不能选择 USB Audio Device，如图 7.8.2 所示。

然后，再切换到 MIDI 输入标签，将“启用 MIDI 输入设备”勾选上，并在下面选择 USB

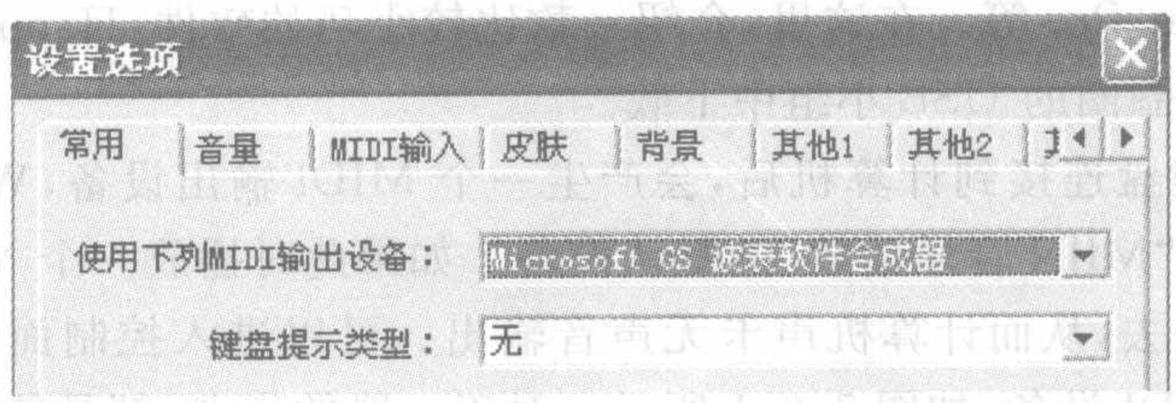


图 7.8.2 选择 MIDI 输出设备

Audio Device，如图 7.8.3 所示。

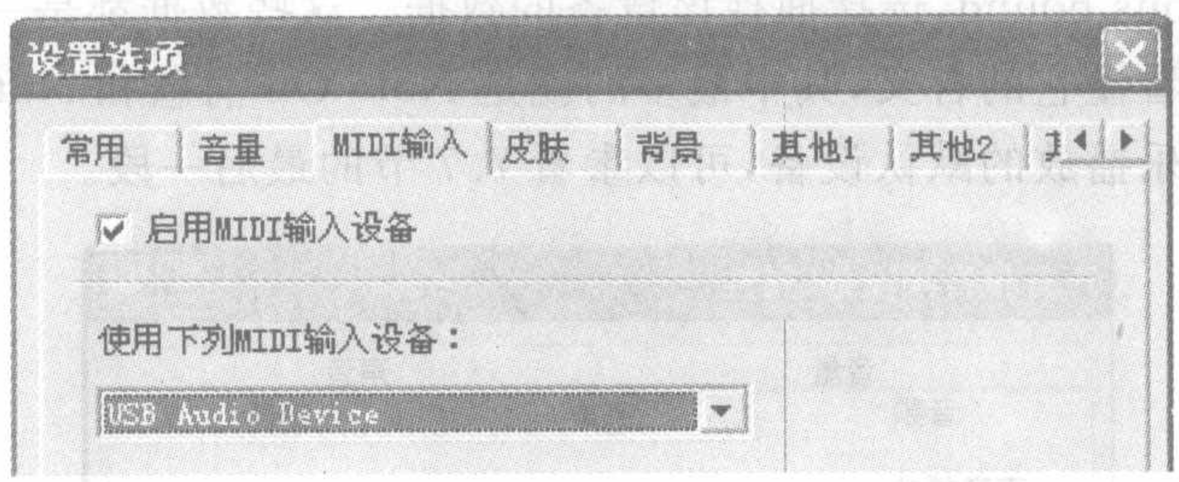


图 7.8.3 选择 MIDI 输入设备

设置好之后，再按动学习板上的按键，是不是听到声音了？在 Option 按钮下面的一排数字按钮可以选择不同的乐器，还可以右击，对不同的按钮分配不同的乐器。不过这个键盘使用起来有难度，音域也很窄，要弹奏一个曲子可不容易，这里仅为演示用。既然人工演示起来麻烦，能不能叫单片机帮我们演奏一曲呢？只要按照事先编排好的曲子，一次次发送这些 MIDI 消息即可。下面一节就介绍如何实现这个单片机自动播放曲子，它已经跟 USB 没有多少关系了，放在这里的目的是为了让大家轻松一下，同时也可以学习一些编程的方法。

7.9 单片机自动弹奏的实现

要单片机自动弹奏，当然要先给它一个曲子。这里挑一个比较简单、又比较好听的曲子：王菲的《容易受伤的女人》。找到曲子后，把每个音符翻译为 MIDI 消息，然后按照谱子中给定的时间间隔发送就行了。不过，如果我们只弹奏主旋律，将显得比较单调，应该再加上和弦和打击乐伴奏才好听。这样一个时刻就需要发送多个音符，如果把全部音符的 MIDI 消息都记录下来，数据量将会比较大。为了减少数据存储量，这里定义了一种结构：曲子以行的格式保存，每行在同一个时刻（实际上是比较短的时间内分多个包发送的）发声。每行的第一字节为该行中需要发声的个数，接下来的两字节分别是音符的音高和力度，一行有多个音时，音高和力度如此重复下去。最后两字节为该行音符发送后停留的时间。但是旋律音和打击乐是在不同的通道上的，那怎么区分通道呢？注意到数据中不会用到 0xFF，为此利用 0xFF 作为通道

切换指示。这里仅使用了两个通道，当遇到 0xFF 时，就切换到另一个通道。整个曲子保存在一个数组中，数组的前两字节为整个曲子的行数，接下来就是一行行的 MIDI 消息数据。具体的数据格式可以参看源代码中的 song.c 文件，里面包括了曲子的数据和播放曲子的函数。感兴趣的读者可以自己找个谱子，然后按照这个格式编一下。不过这个过程有点烦琐，还比较费时间。同时要注意别选太长的曲子，否则 ROM 会不够的。

怎么启动播放呢？这里使用 KEY1 和 KEY8 组合来播放，同时按下 KEY1 和 KEY8 后，将开始自动播放。你也可以自行决定使用哪些组合键，或者使用串口接收命令来播放等。在光盘中有该自动弹奏时的录音文件，读者可以播放来听听，由于没有使用一些音效，所以显得有点呆板。该录音是圈圈计算机上的 MIDI 合成器合成出来的声音，也许跟你在计算机上实验时的声音差别比较大，这是正常现象，也是 MIDI 的特性之一。

7.10　本章小结

本章简单介绍了 USB MIDI 键盘的实现，由于涉及到一些 MIDI 以及音乐方面的知识，这里不方便详述，仅作了一点简单介绍。如果要制作一个完整、符合规范的 USB MIDI 键盘，还有很多工作要做。本章旨在介绍 USB MIDI 设备的基本功能实现，给读者有一个概念上的认识。

第8章

U 盘

U盘这个东西大家都很熟悉了，是一种移动存储设备。早在几年前，软盘还比较流行，几乎每台PC上都有一个软盘驱动器。U盘的出现和发展，打破了这种格局，现在的PC上几乎没人再装软驱了。U盘刚开始出现时，容量只有几十MB，经过短短几年的发展，容量已达到几G甚至更大。

8.1 USB大容量存储设备

在USB协议中，规定了一类大容量存储设备(mass storage device)，U盘就属于大容量存储设备。USB大容量存储设备还包括USB移动硬盘、USB移动光驱等。大容量存储设备的接口类代码(bInterfaceClass字段)为0x08，接口子类代码(bInterfaceSubClass字段)有好几种，但大部分U盘都使用代码0x06，即SCSI通明命令集。协议代码(bInterfaceProtocol字段)有3种：0x00、0x01、0x50，前两种需要使用中断传输，最后一种仅使用批量传输。本实例将使用最后一种协议。

既然是大容量存储设备，那么必须要有一个大容量存储器，它可以是FLASH、硬盘、光盘等。在我们的实验板上，有一个IDE接口，可以在上面挂一块IDE接口的硬盘来作为存储器。不过，考虑到很多读者并没有闲置的硬盘，所以本章第一个实例将在8952单片机的内部模拟一个FAT16的文件系统，做一个假U盘。第二个实例将使用IDE接口的硬盘来作为存储器，即IDE转USB。

将第7章的USB MIDI键盘实例复制一份，改名为UsbDisk。之所以选择它作为修改的模板，是因为它使用了端点2。将key.c、song.c文件从工程中移除，并将代码中无关的部分也删除，这里不再用到它们了。

8.2 设备描述符

设备描述符只需要修改产品ID号(PID)即可，这里是第九个实验，我们改为0x0009。其他字段保持不变。

8.3 字符串描述符

产品字符串改为“《圈圈教你玩 USB》之假 U 盘”，产品序列号改为“2008－08－14”。注意，如果是制作真正的 U 盘，每块 U 盘的序列号都要不一样，否则可能会几块相同的 U 盘无法同时使用。其实圈圈这里设置的这个产品序列号是不符合 USB 仅批量传输协议的，协议里规定该序列号最后至少有 12 位十六进制的数据位。不过 Windows 对此检查并不严格，所以还是可以正常使用。如果是做产品，最好还是按照协议来做。

8.4 配置描述符集合

先将配置描述符集合中与 MIDI 键盘相关的类特殊描述符删除，包括类特殊接口描述符、类特殊端点描述符等。只保留配置描述符、第一个接口描述符和两个标准端点描述符。同时要修改描述符集合的总长度。

8.4.1 配置描述符

配置描述符仅需修改接口数量，原来的 MIDI 键盘为 2 个接口，这里只有一个，将配置描述符的接口数量（bNumInterfaces 字段）改为 0x01。

8.4.2 接口描述符

该接口使用两个批量端点，修改 bNumEndpoints 字段为 0x02。该接口所使用的类为大容量存储设备（代码为 0x08），修改 bInterfaceClass 字段为 0x08。子类为 SCSI 透明命令集（代码为 0x06），修改 bInterfaceSubClass 字段为 0x06。协议使用仅批量传输（bulk only transport）协议（代码为 0x50），修改 bInterfaceProtocol 字段为 0x50。修改好的接口描述符代码如下：

```
/*****************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。该接口有 2 个批量端点
0x02,

//bInterfaceClass 字段。该接口所使用的类。大容量存储设备接口类的代码为 0x08
0x08,

//bInterfaceSubClass 字段。该接口所使用的子类。SCSI 透明命令集的子类代码为 0x06
0x06,

//bInterfaceProtocol 字段。协议为仅批量传输,代码为 0x50
0x50,

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

8.4.3 端点描述符

端点描述符使用原来的端点 2 描述符即可,不用修改。

8.5 测 试

在描述符改完之后,接下来似乎就不知道该改什么了。事实上,在大容量存储设备的仅批量传输协议中规定了两个类特殊请求:Bulk-Only Mass Storage Reset 和 Get Max LUN。这里为了增加感性认识,先不实现这两个类请求,而是打开调试信息,根据调试信息显示的内容进入下一步操作。如图 8.5.1 所示,就是修改好描述符,运行后返回的调试信息。从图中可以看到,主机发送了一个类输入请求,通过查看 USB 大容量存储设备的协议,知道它正是前面提到过的 Get Max LUN 请求。另外,使用 BUS Hound 捕捉数据时,发现发送了一个 Set Interface 的请求,但是设备并没有收到(调试信息中未显示出来),估计这个请求是上层驱动发给下层驱动的,而不是发送给设备。

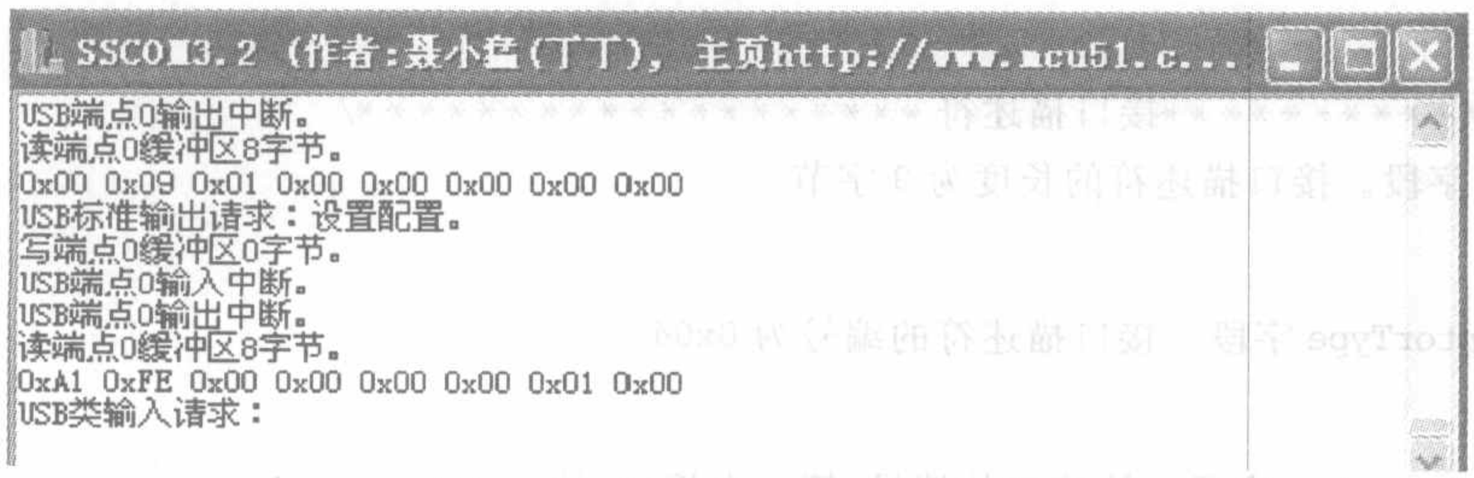


图 8.5.1 修改好描述符后的调试信息

8.6 类特殊请求

在 USB 大容量存储设备的 Bulk Only Transport 协议中，规定了两个类特殊请求：Bulk-Only Mass Storage Reset 和 Get Max LUN。前者是复位到命令状态的请求，后者是获取最大逻辑单元请求。

8.6.1 Get Max LUN 请求

Get Max LUN 请求的格式如表 8.6.1 所列。由 bmRequestType 可知，它是发送到接口的类输入请求，bRequest 值为 0xFE，wIndex 的值为请求的接口号，本实例中为接口 0。传输的数据长度为 1 字节，设备将在数据过程返回 1 字节的数据。该字节表示设备有多少个逻辑单元，值为 0 时表示有一个逻辑单元，为 1 时表示有两个逻辑单元，以此类推，最大可以取 15。通过与前面的调试信息相对照，可以知道刚刚的那个类输入请求正是这个 Get Max LUN 请求。

表 8.6.1 Get Max LUN 请求的格式

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001b	11111110b	0000h	Interface	0001h	1 byte

在端点 0 输出中断处理函数中的类输入请求分支下，增加对该请求的响应代码。首先要定义一个最大逻辑单元的变量。跟描述符类型一样，我们使用数组的方式来定义，尽管它只有一个元素。本实例仅实现一个逻辑单元，所以返回的值为 0。因为只有一个逻辑单元，所以在后面的命令处理中，就忽略了逻辑单元这个字段。如果将最大逻辑单元改成 1，那么系统将会认为有两个逻辑单元，从而分别读两次磁盘数据，结果读到了一样的数据(因为对命令处理时，忽略了逻辑单元字段，返回了一样的数据)，那么就会显示两块参数和内容一样的磁盘。改成 n(最大 15)，就可以得到 $n+1$ 块磁盘，感兴趣的读者可以自己修改来试试(圈圈曾尝试过设置 n 为最大值 15，经过一个漫长的枚举过程后，“我的电脑”中就多了 16 块新硬盘，场面很是壮观，圈圈长这么大还第一次见那么多的硬盘，盘符都到“Y”了)。新增的部分代码如下：

```
switch(bRequest)
{
 case GET_MAX_LUN:                          //请求为 GET_MAX_LUN(0xFE)
  #ifdef DEBUG0
   Prints("获取最大逻辑单元。\r\n");
  #endif

  pSendData = MaxLun;                       //要返回的数据位置
```

```
    SendLength = 1;                          //长度为 1 字节
    //如果请求的长度比实际长度短,则仅返回请求长度
    if(wLength<SendLength)
    {
     SendLength = wLength;
    }
    //将数据通过 EP0 返回
    UsbEp0SendData();
   break;
```

8.6.2 Bulk-Only Mass Storage Reset 请求

Bulk-Only Mass Storage Reset 请求是通知设备接下来的批量端点输出数据为命令块封包 CBW(Command Block Wrapper),其结构如表 8.6.2 所列。在这个请求处理中,仅需要设置一下状态,说明接下来的数据为 CBW,然后返回一个 0 长度的状态数据包即可。

表 8.6.2 Bulk-Only Mass Storage Reset 请求的结构

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001b	11111111b	0000h	Interface	0000h	none

实现该请求的代码如下:

```
switch(bRequest)
{
 case MASS_STORAGE_RESET:
  #ifdef DEBUG0
   Prints("大容量存储设备复位。\r\n");
  #endif

  //接下来的数据为命令阶段(CBW)
  TransportStage = COMMAND_STAGE;
  //返回一个 0 长度的数据包
  SendLength = 0;
  NeedZeroPacket = 1;
  //将数据通过 EP0 返回
  UsbEp0SendData();
 break;
```

8.7 仅批量传输协议的数据流模型

前面的工作完成之后，接下来就是通过批量端点传输数据了。在仅批量传输协议中，规定了数据传输的结构和过程，共分成三个阶段：命令阶段、数据阶段和状态阶段。这有点类似控制传输，但又不完全相同。命令阶段由主机通过批量端点发送一个 CBW（命令块封包）的结构，在 CBW 中定义了要操作的命令以及传输数据的方向和数量。数据阶段的传输方向由命令阶段决定，而状态阶段则总是由设备返回该命令完成的状态。

8.7.1 命令块封包 CBW 的结构

CBW（Command Block Wrapper）的结构如表 8.7.1 所列，总共有 31 字节。

表 8.7.1 CBW 的结构

字节＼位	7	6	5	4	3	2	1	0
0～3	dCBWSignature							
4～7	dCBWTag							
8～11(08h～0Bh)	dCBWDataTransferLength							
12(0Ch)	bmCBWFlags							
13(0Dh)	保留（值为 0）				bCBWLUN			
14(0Eh)	保留（值为 0）			bCBWCBLength				
15～30(0Fh～1Eh)	CBWCB							

dCBWSignature 该字段为 CBW 的标志，为字符串 USBC（即 USB 命令的缩写）。用 ASCII 码来表示就是 0x55、0x53、0x42、0x43。如果用小端模式的 4 字节整数来表示，其值就是 0x43425355。

dCBWTag CBW 的标签，由主机分配，设备在完成该命令返回状态时，需要在 CSW（命令状态封包）中的 dCSWTag 字段中填入命令的 dCBWTag。

dCBWDataTransferLength 需要在数据阶段传输数据的字节数。小端结构，即低字节在先。

bmCBWFlags CBW 的标志。最高位（D7）表示数据传输的方向，0 表示输出数据（从主机到设备），1 表示输入数据。其他位为 0。

bCBWLUN 该字段为目标逻辑单元的编号。当有多个逻辑单元时，使用该字段来区分不同的目标单元。该字段仅使用低 4 位，高 4 位为保留值 0。

bCBWCBLength CBWCB的长度。该字段仅使用5位，有效的取值范围为1～16。不同的命令(由CBWCB决定)，其长度可能是不一样的。如果命令的长度不足16字节，则后面的部分值为0。

CBWCB 需要执行的命令，由选择的子类决定使用哪些命令。

8.7.2 命令状态封包CSW的结构

CSW(Command Status Wrapper)的结构如表8.7.2所列，共有13字节。

表8.7.2 CSW的结构

字节＼位	7	6	5	4	3	2	1	0
0～3	dCSWSignature							
4～7	dCSWTag							
8～11(8－Bh)	dCSWDataResidue							
12(Ch)	bCSWStatus							

dCSWSignature 该字段为CSW的标志，为字符串USBS(即USB状态的缩写)。用ASCII码来表示就是0x55、0x53、0x42、0x53。如果用小端模式的4字节整数来表示，其值就是0x53425355。

dCSWTag 该命令状态封包的标签，其值为CBW中的dCBWTag，响应哪个CBW，就设置为哪个CBW的dCBWTag。

dCSWDataResidue 命令完成时的剩余字节数。它表示实际完成传输的字节数与主机在CBW中设置的长度dCBWDataTransferLength之间的差额。

bCSWStatus 命令执行的状态。0x00表示命令成功执行，0x01表示命令执行失败，0x02表示阶段错误。其他值为保留值。

通常，命令都能够成功完成，这时只需要设置前面两个字段为相应的值，后面两个字段都设置为0即可。

8.7.3 对批量数据的处理

第一次批量数据，肯定是CBW。定义一个缓冲区，用来接收命令块封包CBW。然后进入到数据阶段，在数据阶段中，对CBW进行解码，返回或者接收相应的数据。数据发送或接收完毕后，进入到状态阶段，返回命令执行的情况。然后再次进入命令阶段，等待主机发送CBW命令块封包。

8.8 SCSI 命令集和 UFI 命令集

SCSI(Small Computer System Interface)是小型计算机系统接口的缩写,有一套完整的协议规定其命令和命令数据的响应。SCSI 命令有很多,但是在 U 盘中常用的就几个:INQUIRY 命令、READ CAPACITY 命令、READ(10)命令、WRITE(10)命令等。

然而,虽然在接口子类代码(bInterfaceSubClass 字段)中指定使用 SCSI 透明命令集,但是圈圈却发现 Windows 并不按照这一套来搞,而使用的是 USB 定义的 UFI 协议。但是如果将接口子类代码指定为 UFI(即 0x04),Windows 的标准驱动又认不出来,弄不明白系统是咋回事。没办法,这里只好指定接口子类代码为 SCSI 透明命令集,而实际使用 UFI 协议的命令。虽然程序中的文件名、函数及变量名用的是 SCSI,但实际上用的是 UFI 命令集,读者要注意这一点。另外,它又并不是完全遵循 UFI 协议,例如在命令中的逻辑单元一项始终为 0,不过好在逻辑单元可以从 CBW 中获取;还有主机会发送一个操作代码为 0x1A 的命令,这在 UFI 协议中找不到(在 SCSI 命令中倒是有),只好返回命令执行失败。

USB 软盘接口 UFI(USB Floppy Interface)命令是结合 SCSI-2 和 SFF-8070i 命令集而抽取出来的一些命令,所以在 SCSI 协议以及 SCSI 块设备命令中能够找到很多命令的定义,主要是一些强制命令。

UFI 命令出现在 CBW 的 CBWCB 字段中,最多可以有 16 字节,如果不足 16 字节,多余的将被忽略,通常设置为 0。所有 UFI 命令的第一字节(在 CBW 中偏移量为 15)为操作代码,在程序中根据不同的操作代码进行处理。

以下分别对要用到的几个 UFI 命令以及返回数据的格式作详细介绍,另外还有一些 UFI 命令没有在此详述,可以参看 USB UFI 协议文档。

8.8.1 查询命令 INQUIRY

INQUIRY 命令请求目标设备的一些基本信息,其格式如表 8.8.1 所列,操作代码为 0x12。其中 EVPD 字段和页码字段只支持 0。分配的缓冲区长度为主机接收该命令返回数据所分配的缓冲区,缓冲区长度通常为 0x24(即 36 字节)。

表 8.8.1 INQUIRY 命令的格式

字节 \ 位	7	6	5	4	3	2	1	0
0	操作码(0x12)							
1	逻辑单元号			保留(0)				EVPD(0)
2	页码(只支持 0)							
3	保留(0)							

续表 8.8.1

字节 \ 位	7	6	5	4	3	2	1	0
4	为返回数据分配的存储空间长度，通常为 36 字节							
5～11	保留							

INQUIRY 命令返回的数据为 36 字节，如表 8.8.2 所列。外设类型为 0 表示直接寻址设备（例如磁盘）。

表 8.8.2 INQUIRY 命令响应数据格式

字节 \ 位	7	6	5	4	3	2	1	0
0	保留 0			外设类型				
1	RAM	保留						
2	ISO 版本号(0)		ECMA 版本号(0)			ANSI 版本号(0)		
3	保留				响应数据格式(0x01)			
4	附加数据长度(31 字节)							
5～7	保留							
8～15	厂商信息							
16～31	产品信息							
32～35	产品版本信息(n. nn)							

RMB 位表示存储媒介是否可移除，0 为不可移除，1 为可移除。

如果设置为不为可移除的，那么将显示在“我的电脑”中的“硬盘”区；如果设置为可移除的，那么将显示在“有可移动存储的设备”区，两者的图标也不一样，如图 8.8.1 所示。

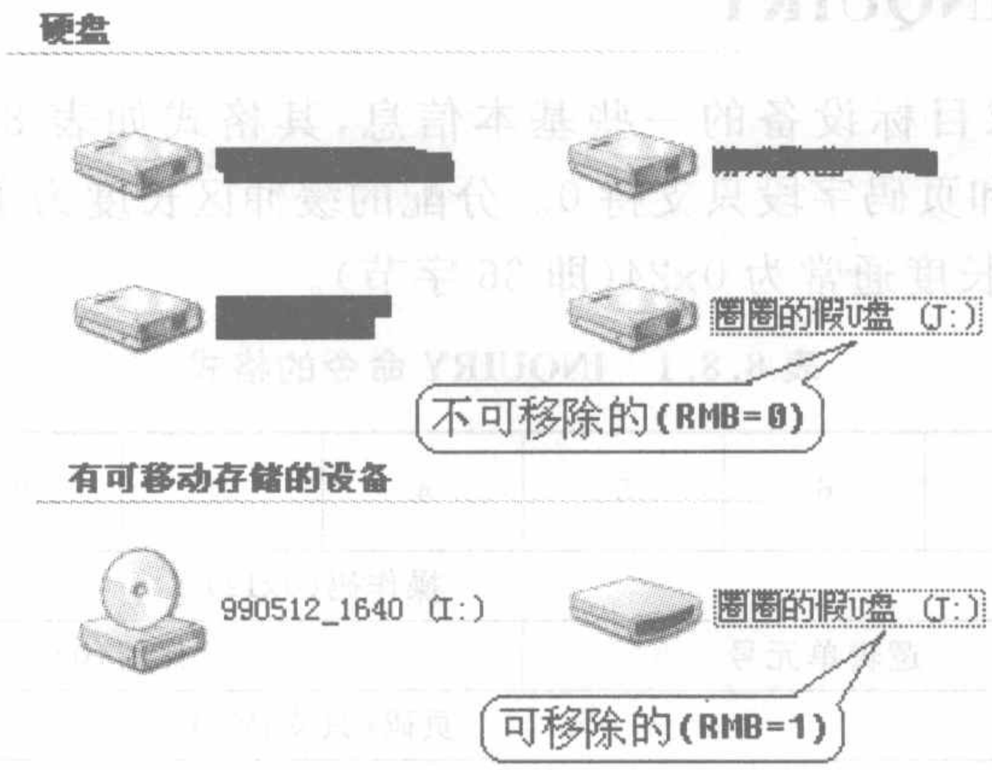


图 8.8.1 不同 RMB 属性时的磁盘

ISO、ECMA、ANSI 等各种版本号规定为 0，“响应数据格式”为 0x01。

附加数据长度是后面附加数据的长度，为 31 字节。

厂商信息、产品信息、产品版本号可以根据自己的需要设置。

具体的返回信息内容可参看源代码中的 DiskInf 数组，它在 SCSI.c 文件中。

8.8.2 读格式化容量命令 READ FORMAT CAPACITIES

READ FORMAT CAPACITIES 命令可让主机读取设备各种可能的格式化容量的列表，如果设备中没有存储媒介，则设备返回最大能够支持的格式化容量。该命令的格式表 8.8.3 所列。

表 8.8.3 READ FORMAT CAPACITIES 命令格式

字节＼位	7	6	5	4	3	2	1	0
0	操作代码(0x23)							
1	逻辑单元号			保留				
2～6	保留							
7	分配的缓冲区长度(高字节)							
8	分配的缓冲区长度(低字节)							
9	保留							
10	保留							
11	保留							

表 8.8.4 是没有存储媒介时返回最大格式化容量的数据格式。其中，容量列表长度为 8 字节(即后面的 8 字节)，描述符代码为 3。容量的计算方法为

$$容量=块数\times每块字节数$$

另外，还有几种格式化容量的列表格式，本实例用不到，读者可以参看 UFI 协议文档。

通常每块字节数为 512 字节(即 0x200)，块数可以设置得大一些，它表示该设备最大支持的格式化容量，实际的容量要由存储媒介来决定。

表 8.8.4 返回的最大格式化容量格式

字节 \ 位	7	6	5	4	3	2	1	0
0	保留							
1	保留							
2	保留							
3	容量列表的长度							
4	(MSB)							
5				块数(高字节在先)				
6								
7								(LSB)
8	保留							描述符代码
9	(MSB)							
10				每块字节数(高字节在先)				
11								(LSB)

8.8.3 读容量命令 READ CAPACITY

READ CAPACITY 命令可以让主机读取到当前存储媒介的容量，其格式如表 8.8.5 所列，操作代码为 0x25。该命令除了操作代码为 0x25 之外，其他各字段的值都为 0。

READ CAPACITY 命令的数据响应格式如表 8.8.6 所列。最后逻辑块地址为存储媒介能够被访问的最大逻辑块地址，块(逻辑块)大小为每个逻辑块的字节数，通常为每块 512 字节。磁盘的总容量计算方法为

磁盘容量(字节数)=(最大逻辑块地址+1)×块大小

由于逻辑块地址是从 0 开始的，所以需要加 1。

READ FORMAT CAPACITIES 命令读到的是设备所支持最大的容量，而 READ CAPACITY 命令读到的才是实际的磁盘容量，注意二者的不同。

表 8.8.5 READ CAPACITY 命令格式

字节 \ 位	7	6	5	4	3	2	1	0
0	操作代码(0x25)							
1	逻辑单元号			保留				RelAdr

续表 8.8.5

字节 \ 位	7	6	5	4	3	2	1	0
2	(MSB)							
3				逻辑块地址(LBA)				
4								
5								(LSB)
6	保留							
7	保留							
8	保留							PMI
9	保留							
10	保留							
11	保留							

表 8.8.6 READ CAPACITY 命令返回数据格式

字节 \ 位	7	6	5	4	3	2	1	0
0	(MSB)							
1				最后逻辑块地址				
2								
3								(LSB)
4	(MSB)							
5				块大小(字节数)				
6								
7								(LSB)

8.8.4 READ(10)命令

主机通常使用 READ(10)命令来读取实际的磁盘数据，其命令格式如表 8.8.7 所列，操作代码为 0x28。另外还有一个 READ(12)命令(操作代码为 0xA8)，它的格式跟 READ(10)命令差不多，仅传输长度字段不一样，它的字节 6～9 都为传输长度，而 READ(10)命令只有字节 7～8 为传输长度。

其中 DPO、FUA、RelAdr 等字段都为 0 值。

逻辑块地址字段的值为需要读取数据的起始块的地址。在磁盘设备中，读、写通常都是按照块来操作的(所以叫做块设备)。一般来说，一个逻辑块就是一个扇区，大小为512字节。当然也有其他大小的逻辑块，例如在光盘文件系统中一个块就是2 048字节。

传输长度字段的值为需要传输的逻辑块的数量，实际传输的字节数为传输长度值乘以每块大小。

设备根据命令中指定的逻辑块地址，从存储媒介中读取数据并通过批量端点返回。当全部数据都返回后，再返回命令执行状态CSW。

表8.8.7 READ(10)命令格式

字节＼位	7	6	5	4	3	2	1	0
0	操作代码(0x28)							
1	逻辑单元号			DP0	FUA	保留		RelAdr
2	(MSB)							
3	逻辑块地址							
4								
5								(LSB)
6	保留							
7	传输长度(MSB)							
8	传输长度(LSB)							
9	保留							
10	保留							
11	保留							

8.8.5 WRITE(10)命令

主机通常使用WRITE(10)命令往设备写入实际的磁盘数据，其命令格式如表8.8.8所列，操作代码为0x2A。另外还有一个WRITE(12)命令(操作代码为0xAA)，它的格式跟命令WRITE(10)差不多，仅传输长度字段不一样，它的字节6～9都为传输长度，而WRITE(10)命令只有字节7～8为传输长度。

该命令的各参数跟READ(10)命令类似，请参看READ(10)命令。

主机在发送此命令之后，接着就会发出实际要传送的数据，设备在收到全部数据后，返回命令执行的情况CSW。

表 8.8.8 WRITE(10)命令格式

字节 \ 位	7	6	5	4	3	2	1	0
0	操作代码(0x2A)							
1	逻辑单元号			DP0	FUA	保留		RelAdr
2	(MSB)							
3	逻辑块地址							
4								
5								(LSB)
6	保留							
7	传输长度(MSB)							
8	传输长度(LSB)							
9	保留							
10	保留							
11	保留							

8.8.6 REQUEST SENSE 命令

REQUEST SENSE 命令用来探测上一个命令执行失败的原因，主机可在每个命令之后使用该命令来读取命令执行的情况。其命令代码为 0x03，格式如表 8.8.9 所列。

表 8.8.9 REQUEST SENSE 命令的格式

字节 \ 位	7	6	5	4	3	2	1	0
0	操作代码(0x03)							
1	逻辑单元号			保留				
2	保留							
3	保留							
4	为返回数据分配的缓冲区长度							
5～11	保留							

该请求返回的数据格式如表 8.8.10 所列。其中 Valid 字段指示信息字段是否符合 UFI 规范，Valid 为 1 时，说明信息字段符合 UFI 规范。在 UFI 协议中，信息字段通常用来返回哪个逻辑块地址出现了错误。Sense Key、Additional Sense Code(ASC)、Additional Sense Code Qualifier(ASCQ)表示出错的代码，可以在 UFI 协议的最后找到。在本实例中，仅返回一种错误原因：无效的命令操作码，其 Sense Key 为 0x05，ASC 为 0x20，ASCQ 为 0。当然，对于一个

实际的 U 盘，可能会有更多的出错原因，这就需要根据具体的情况来决定了。

表 8.8.10 REQUEST SENSE 命令返回的数据格式

字节＼位	7	6	5	4	3	2	1	0
0	Valid	错误代码(固定为 0x70)						
1	保留							
2	保留				Sense Key			
3	(MSB)							
4	信息							
5								
6								(LSB)
7	附加长度(固定为 10 字节)							
8～11	保留							
12	Additional Sense Code(ASC)							
13	Additional Sense Code Qualifier(ASCQ)							
14～17	保留							

8.8.7 TEST UNIT READY 命令

TEST UNIT READY 命令用来测试设备的某个逻辑单元是否准备好，操作代码为 0x00，其格式如表 8.8.11 所列。该命令的响应比较简单，如果设备已经准备好，则在状态阶段返回命令执行成功；否则返回命令执行失败。当主机使用 REQUEST SENSE 命令来探测错误原因时，设置 Sense Key 为 NOT READY。本实例中总是返回成功，即逻辑单元已准备好。

表 8.8.11 TEST UNIT READY 命令格式

字节＼位	7	6	5	4	3	2	1	0
0	操作代码(0x00)							
1	逻辑单元号			保留				
2～11	保留							

8.9 FAT 文件系统

由于在实验板上没有额外的 FLASH 等存储设备，因而本实验要在 ROM 中模拟一个 FAT 文件系统。如果有存储设备，则可以让 Windows 帮忙格式化，而不用自己建立文件系

统。FAT 文件系统稍微有点复杂，读者可以在网上去搜索一篇叫做《4.5 万字透视 FAT32 系统》的文章，里面说得比较详细，当然也可以参看官方的 FAT 文件系统文档。在这里仅简单地说说在模拟 FAT 文件系统时需要用到的一些内容。

FAT(File Allocation Table)是文件分配表的缩写，是为了方便文件的存储、检索、添加和删除等操作而提出的一种链表式的文件组织结构。如果给每个文件分配同样大小的存储空间这显然是不现实的，因此在 FAT 文件系统中，文件大小是不固定的。这些文件是以簇为最小单位，将每个簇号以链表的格式保存在一张专门的表格中，这张表格就叫做 FAT。目录也可以看作一个特殊的文件，这个文件被分成一个个的目录项，每个目录项保存了文件的文件名、文件长度、创建日期、起始簇号等重要信息。每个逻辑磁盘都有一个根目录，所有的子目录和文件都放在根目录下。在 FAT12/16 文件系统中，根目录位于整个磁盘的文件数据开始处，这样系统就能够找到根目录。通过读取根目录的数据，就能找到各个文件(包括目录)的目录项并打开文件。

一块磁盘应该包括主引导记录 MBR(Master Boot Record)、扩展引导记录 EBR(Extended Boot Record)、磁盘操作系统引导记录 DBR(DOS Boot Record)、文件分配表(File Allocation Table)、根目录区和文件数据区等几个重要部分。

其中，主引导记录 MBR 中记录了 MBR 引导代码、磁盘分区表等重要信息。当主引导记录中的磁盘分区表不够用时，就要用到扩展引导记录 EBR。DBR 则是每个逻辑分区的一个引导记录，里面记录了该分区的众多重要信息以及引导代码，所以十分重要。每个 MBR、EBR、DBR 通常分别只占用一个扇区(512 字节)。

在 U 盘系统中，可以没有 MBR 和 EBR，而仅有 DBR。在 DBR 之后，就是 FAT 区(通常有两个，一个为副本)，接着就是文件(包括目录)数据区。FAT12/16 文件系统跟 FAT32 文件系统的根目录区有点不一样，FAT32 的根目录是不固定大小的，跟普通目录一样，而 FAT12/16 的根目录是固定大小的，所以通常在 FAT12/16 文件系统中把根目录区单独列出来。

在 FAT 文件系统中，使用的都是小端结构，即低字节在先。

本实例将使用 FAT16 文件系统，以下内容仅针对 FAT16 文件系统(除非特别说明)，FAT32 所使用的结构跟 FAT16 的有所差别。

8.9.1 关于 DBR

DBR 占据逻辑分区的 0 扇区(逻辑块地址)，大小通常为 512 字节，DBR 各个部分的意义如表 8.9.1 所列。

表 8.9.1 中跳转指令为引导操作系统时用，如果该分区不作为启动盘，可以不用理会该字段。但是操作系统在识别磁盘时，会检查这些值是否有效，所以不能任意设置，可以找个现成的、已经格式化好的 U 盘来查看这些数据是什么(用 BUS Hound 很容易捕捉)，然后添加进来。OEM 字段为厂商定义的字符串，通常为格式化该分区的操作系统的类型以及版本号，为

了向旧版本兼容，通常 Windows 格式化时会设置该字段为字符串 MSDOS5.0。每扇区字节数就是一个扇区的字节数(也就是前面所说的块大小)，可以选择为 512、1 024、2 048、4 096，通常为 512 字节。每簇扇区数就是一簇所含有的扇区数，必须为 2 的整数次方，并且要保证每簇字节数不大于 32 KB(在本实例中，每簇扇区数设置为 32，每扇区字节数设置为 512，因此每簇大小为 16 KB)。保留扇区数在 FAT16 文件系统中通常规定为 1，即扇区 0 的 DBR。FAT 数为该分区中 FAT 的个数，通常为两个，另一个是备份。根目录项数就是根目录有多少个目录项，每个目录项为 32 字节，该值乘以 32 必须为每扇区字节的整数倍，即要求所有的根目录项刚好放入整数个扇区内，为了保持兼容性，该值最好设置为 512。小扇区数和大扇区数用来表示该分区有多少个扇区，当值小于 65 536 时，使用小扇区数；否则使用大扇区数。“每 FAT 扇区数”表示每个 FAT 占用多少个扇区，在 FAT16 中，每个 FAT 表项为 2 字节，因此每 FAT 扇区数就决定了总共有多少个 FAT 表项，也就决定了该分区最多可能的簇数。隐藏扇区数为该分区之前隐藏的扇区数，具体的值由操作系统格式化时决定，可以设置为 0。DBR 最后两字节必须为 0x55、0xAA，否则为无效。

表 8.9.1 FAT16 文件系统的 DBR 的结构

偏移量	字段长度	字段名	说 明
0x00	3	跳转指令	跳转到引导代码处的跳转指令
0x03	8	OEM	厂商定义的字符串
0x0B	2	每扇区字节数	每扇区的字节数，通常为 512 字节
0x0D	1	每簇扇区数	每簇的扇区数
0x0E	2	保留扇区数	保留的扇区数，FAT16 通常为 1
0x10	1	FAT 数	该分区的 FAT 数量，通常为 2(一个备份)
0x11	2	根目录项数	该字段仅被 FAT12/16 使用，FAT32 为 0
0x13	2	小扇区数	该分区的扇区数量(小于 65535 个扇区时使用)
0x15	1	媒体描述符	0xF8 表示硬盘，0xF0 表示高密度 3.5 寸软盘
0x16	2	每 FAT 扇区数	每个 FAT 所占用的扇区数量，只被 FAT12/16 使用
0x18	2	每道扇区数	每个磁道上的扇区数量
0x1A	2	磁头数	磁头的数量
0x1C	4	隐藏扇区数	引导扇区之前的扇区数
0x20	4	大扇区数	该分区的扇区数量(大于 65 535 个扇区时使用)
0x24	1	物理驱动器号	0x80 表示硬盘，0x00 表示软盘
0x25	1	保留	值为 0
0x26	1	扩展引导标签	为 0x29，表明接下来 3 个域可用
0x27	4	标卷序列号	由时间和日期构成的 32 位数

续表 8.9.1

偏移量	字段长度	字段名	说 明
0x2B	11	磁盘标卷	与根目录的磁盘标卷一致，无标卷时为 NO NAME
0x36	8	文件系统类型	为字符串“FAT16”
0x3E	448	引导代码	启动操作系统的引导代码
0x1FE	2	有效结束标志	有效的结束标志为 0x55、0xAA

8.9.2 关于 FAT 表

在 FAT16 文件系统中，FAT 表紧跟在 DBR 之后（因为只有一个保留扇区，即 0 扇区的 DBR），也就是说，FAT 表从 1 扇区开始。通常 FAT 有一个副本，该副本紧跟在第一个 FAT 之后。

FAT 是一张保存了根据当前簇号就能找到下一簇号的表格。FAT 是一块连续的数据区，在这块连续的数据区中分成很多大小一样的项，并从 0 开始编号。根据 FAT 文件系统类型的不同，分项时所使用的长度也不一样。FAT 文件系统后面所跟的数字就表示分项时所使用的位(Bit)数，例如 FAT16 就表示每个项为 16 位，即 2 字节，而 FAT32 则为 32 位，即 4 字节。在每个项中，保存了文件所在的下一簇的簇号（或者文件结束标志）。由于要保存簇号，因而不同长度的项就对最大簇号有了不同的限制，例如 FAT32 就比 FAT16 能够表达更多的簇号，从而能够管理的分区容量也更大。理论上来说，FAT16 能够分配的最大分区容量为 65 536×32 KB=2 GB（实际上要比这个略微小一点，因为有些表项值有特殊意义）。

那么 FAT 的表项编号跟数据区之间有什么关系呢？前面提到过，文件空间分配是以簇为最小单位的，也就是说，整个数据区是被分成一个个簇的。而 FAT 呢？也刚好是把整个 FAT 区分成一个个项。如果我们把 FAT 区、根目录区分别当作一个簇（虽然其大小不是刚好等于一个簇，但是可以在逻辑上这样去划分），那么整个分区都被分成一个个簇了。将这些簇也从 0 开始编号（叫做簇号），那么每个簇刚好在 FAT 表中有个对应，即簇 0（也就是 FAT 表区）对应着 FAT 表中的项 0，而簇 1（即根目录区）对应着 FAT 表中的项 1，真正的数据区从簇 2（项 2）开始。

在文件目录项中，保存了一个文件（包括目录）的起始簇号，通过它可以找到文件的第一簇的数据，并且在该簇号对应的 FAT 表项中记录了下一簇的簇号。根据下一簇的簇号，又可以找到下一簇的数据，并且在该簇号对应的 FAT 表项中又记录了下下簇的簇号……按照这个关系，可以一直将整个文件的数据找出来。那么怎么知道文件已经结束了呢？有一些特殊的簇号（例如 FAT16 的 0xFFF8～0xFFFF）来表示文件结束，当某个 FAT 表项中保存的簇号为文件结束簇号时，就知道该簇是该文件的最后一簇了。当然，簇的单位比较粗糙，不能精确到多少字节，在目录项中还有一个精确的文件长度，它保存了整个文件的字节数。如果最后一簇

数据都已经读入后还不足目录项中所记录的文件长度时，则说明该文件已经损坏。

在 FAT 表项中，0 表示该簇未被使用，0xFFF0～0xFFF6 表示保留簇，0xFFF7 表示该簇已经损坏，0xFFF8～0xFFFF 表示文件最后一簇，其他簇号就用来表示该文件的下一簇的簇号。前面提到过，簇 0 和簇 1 分别可以看作 FAT 表区和根目录区，它们都已经被使用了，所以其对应的 FAT 表项没有实际的意义，因而规定 FAT 项 0 的值为 0xFFF8，而项 1 的值为 0xFFFF。当一个文件被删除时，并没有真正地删除文件的数据，而只是将文件的目录项设置为删除状态，并将其占用的 FAT 表项设置为 0。

8.9.3 关于目录项

目录也是一个特殊的文件(FAT12/16 的根目录除外，它是固定的)，里面的数据被划分为一个个目录项。每个目录项大小都为 32 字节，其结构如表 8.9.2 所列。该结构为短文件名的格式，另外还有长文件名的格式。在这里仅介绍短文件名，并且在实例中也仅使用短文件名。

其中文件名为 8 个字节，不包括点和扩展名。扩展名由后面的 3 字节指定，而文件名和扩展名之间的点是由操作系统负责增加的。短文件名目录项的文件名和扩展名必须使用大写字母，如果文件名或扩展名长度不够，则用空格(0x20)来填充。注意文件名的第一个字节具有特殊意义。当第一字节为 0 时，表示该目录项为空，可以使用，并且其后面所有该目录下的目录项都为空。当第一字节为 0xE5 时，表示该目录项曾经被使用过，但是现在该文件已经被删除了，该目录项目前为空，可用。当第一字节为 0x05 时，表示 ASCII 为 0xE5(在日文中为有效字符)的字符，避免系统误认为该文件已删除。如果文件名为“.”，则表示当前目录；文件名为“..”，则表示其父目录。文件名的第一字节不可为空格(0x20)。

偏移为 0x0B 的字段为属性字节，它表示一个文件的属性，每个属性由一个二进制位指示。当某个二进制位为 1 时，就表示该文件具有这样的属性，例如一个只读的隐藏文件属性为 00000011B，即 0x03。

时间格式(16 位)为：位 15～11 表示小时，可以取值为 0～23；位 10～5 表示分，可以取值为 0～59；位 4～0 表示秒，可以取值为 0～29，每单位为 2 s，即实际的秒值为该值的 2 倍。

日期格式(16 位)为：位 15～9 表示年份，可以取值为 0～127，它表示距离 1980 年差值，即实际的年份为该值加上 1980，最大可表示到 2107 年；位 8～5 表示月份，可以取值为 1～12；位 4～0 表示几号，可以取值为 1～31。

在 FAT16 中，簇号只有 2 字节，因此只使用起始簇号的低字，高字部分为 0，在 FAT32 中会使用高字。文件长度为 4 字节(32 位)，因此单个文件最大长度为 4 GB。注意目录项中的所有整数为小端结构，即低字节在先。

在每个分区的根目录下，有个特殊的目录项，就是磁盘标卷。它的属性字节值为 0x08。其文件名就是所显示的磁盘名。

表 8.9.2 目录项的结构

偏移量	字节数	字段名	说 明
0x00	8	文件名	保存文件名的 8 字节(不包括扩展名和点)
0x08	3	扩展名	文件的扩展名
0x0B	1	属性字节(为右边所列出的属性的组合)	00000000B 可读/写文件(B 表示二进制,下同)
			00000001B 只读文件
			00000010B 隐藏文件
			00000100B 系统文件
			00001000B 标卷
			00010000B 子目录
			00100000B 归档
0x0C	1	保留	系统保留,值必须设置为 0
0x0D	1	创建时间(ms)	文件创建时的毫秒时间,协议中说单位为 1/10 s。但是又说取值为 0～199,不知是否搞错了
0x0E	2	创建时间	文件创建的时间
0x10	2	创建日期	文件创建时的日期
0x12	2	最后访问日期	文件最后被访问的日期。如果是写访问,必须要等于最后修改日期
0x14	2	起始簇号高字	该文件第一簇所在簇号的高字,FAT16 必须为 0
0x16	2	最后修改时间	该文件最后被修改的时间
0x18	2	最后修改日期	该文件最后被修改的日期
0x1A	2	起始簇号低字	该文件第一簇所在簇号的低字
0x1C	4	文件长度	该文件的长度(字节数)

8.10 模拟一个 FAT16 文件系统

如何来模拟一个 FAT16 文件系统呢?一个完整的 FAT16 系统需要 4 个部分:DBR、FAT 表、根目录和数据区。由于实验板上的 ROM 空间有限,仅设置关键部分的数据,其余部分数据设置为 0 即可。那么哪些部分的数据是关键的呢?分别是:整个 DBR、两个 FAT(包括一个备份)的前 6 字节(这里只设置一个文件,因而只用到表项 2)、根目录区的前两个目录项(系统标卷和一个文件)、数据区的文件内容部分。将这些关键数据以及 0 以数组的格式保存起来,在适当的时候返回即可。那么又怎样去判断何时返回这些必要数据和数据 0 呢?根据

主机发送的 READ(10)命令的逻辑块地址以及当前返回的字节数即可判断出来。

本实例打算模拟一个 128 MB 的 U 盘，扇区大小为 512 字节，每簇大小为 16 KB(即每簇扇区数为 32)，总共有 128 MB/16 KB=8 K 簇，相应地，也需要这么多个 FAT 表项。FAT16 的每个 FAT 表项为 2 字节，因此一个扇区可以保存 256 个 FAT 表项，那么保存 8 K 个表项需要的扇区数为 8 192/256=32 个，即每 FAT 扇区数为 32。根目录项数设置为 512 个，每个目录项需要 32 字节，因此总共需要 32 个扇区保存根目录。

根据以上分析，可以得出整个模拟磁盘的结构如下：0 扇区为 DBR(前 62 字节需要自己设计，后面的引导代码可以直接复制一个现成 U 盘的，注意跳转指令和后面的引导代码部分等不要设置为 0，否则会让操作系统认不出来或者提示未格式化)，1～32 扇区为 FAT 表，33～64 扇区为 FAT 表备份，65～96 扇区为根目录区，97 扇区至结束为数据区。其中后面的 FAT、根目录、数据区等只有前面小部分数据为非 0，其他部分全部填充为 0，如表 8.10.1 所列。当主机使用 READ(10)读数据时，将逻辑块地址 LBA 转换为字节地址，然后查看它在哪个区间，就可以知道该返回什么数据了。当然，主机一次读操作至少为一个扇区，所以每发送完一个数据包后都要重新计算字节地址，并再次获取需要返回数据的地址。由于 D12 的端点 2 大小为 64 字节，所以这些数组至少为 64 字节，一次数据包返回才正确。对于连续的数据(例如 0 扇区的 DBR)，在发送数据时会自动调整指针位置，只需要设置数据的起点位置即可。

表 8.10.1 模拟磁盘的结构

扇区地址	内 容	数 据	字节地址范围
0	DBR	前 62 字节关键数据以及后面的引导代码，整个 DBR 扇区为一个数组	0～511
1～32	FAT(1)	前 64 字节关键数据(仅前 6 字节的表项有用)	512～575
		填充 0	576～16 895
33～64	FAT(2)	前 64 字节关键数据(仅前 6 字节的表项有用)	16 896～16 959
		填充 0	16 960～33 279
65～96	根目录	前 64 字节关键数据(磁盘标卷以及测试文件)	33 280～33 343
		填充 0	33 344～49 663
97～结束	数据区	前半段测试文件数据(不足一扇区)	49 664～50 175
		填充 0	大于 501 765

8.11 实验结果

前面对这些命令的结构、返回数据格式都说得比较详细，这里就不再贴出处理的源代码了，读者可以对照书中的内容来阅读光盘中的源代码，主要在 SCSI.c 和 FAT.c 两个文件中。

由于实验板上没有存储设备，所以对于输出的数据就直接丢弃了。但是还是可以往这个 U 盘中复制数据的，并且少量数据还可以正常读回来，这主要是因为 Windows 的缓冲机制。为了加快访问速度，Windows 会将少量的文件数据缓冲在内存中，当下次读取时，就直接从缓冲区中读取，而不是直接访问磁盘。

当枚举成功后，会在任务栏的右下角出现拔下 U 盘的图标，双击它会弹出安全删除硬件的对话框。在这里可以看到在 INQUIRY 命令中返回的厂商信息和产品信息，如图 8.11.1 所示。

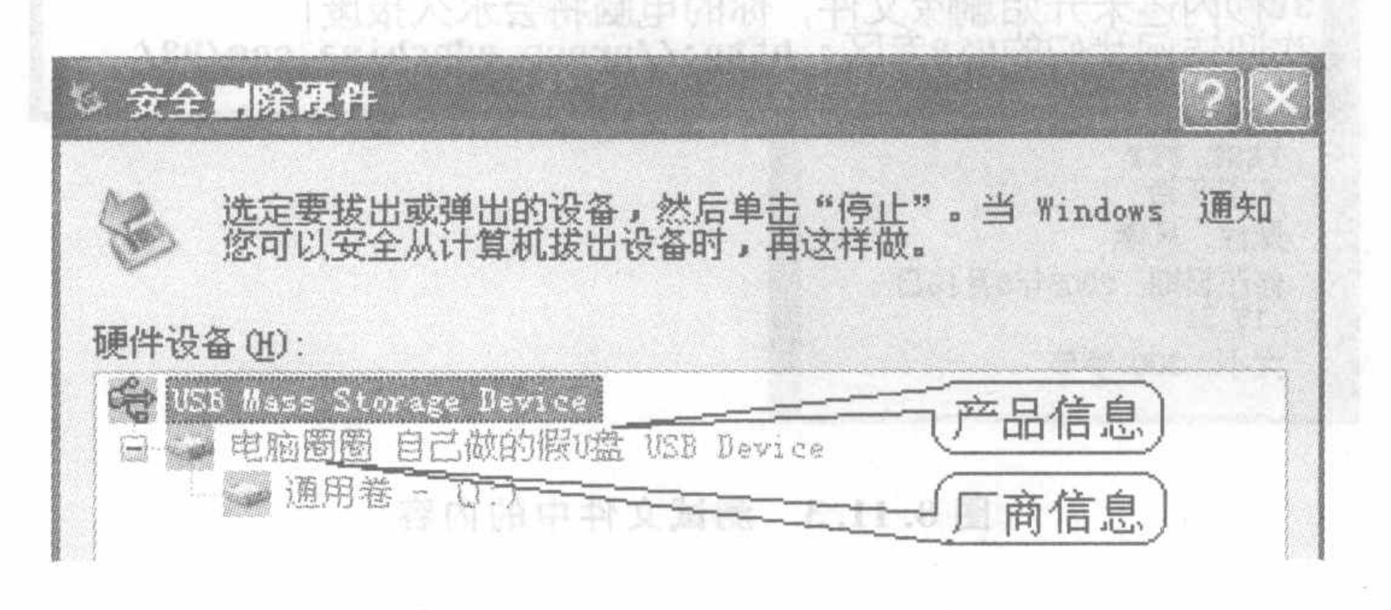


图 8.11.1 安全删除硬件的对话框

进入“我的电脑”，可以看到多出一块磁盘(如果你将 MXA LUN 设置为更大，可以获得多块磁盘)，右击，选择属性，可以看到它的大小为 127 MB，已用空间为 16 KB，如图 8.11.2 所示。因为里面有一个测试文件 TEST.TXT，所以占用了 16 KB 的空间。虽然这个文件只有 300 字节，但是前面说过，文件是以簇为最小单位分配空间的，而这里每簇大小为 16 KB，所以要占用 16 KB 的空间。在磁盘上，很多小文件通常会占用很大的空间，就是因为按簇分配空间导致的，将它们压缩成单个文件，就会节省不少空间。

进入这个假 U 盘，可以看到里面有一个名为“TEST.TXT”的文本文件，这是在程序中模拟的一个文件。文件名、文件的创建时间、修改时间(在文件属性中查看)等信息保存在根目录的第二个目录项中，而文件中的内容保存在 TestFileData 数组中。打开这个文本文件，可以看到我们所设置的文本内容，如图 8.11.3 所示。

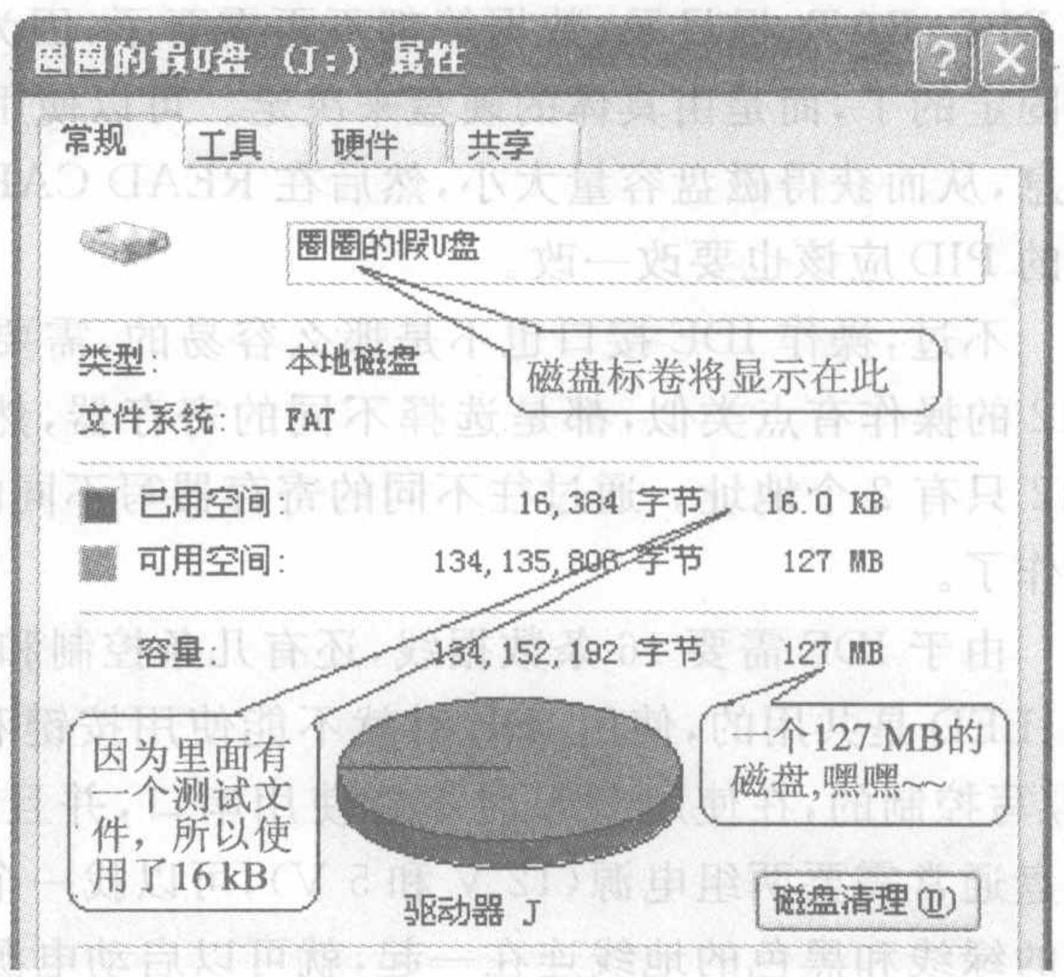


图 8.11.2 假 U 盘的属性

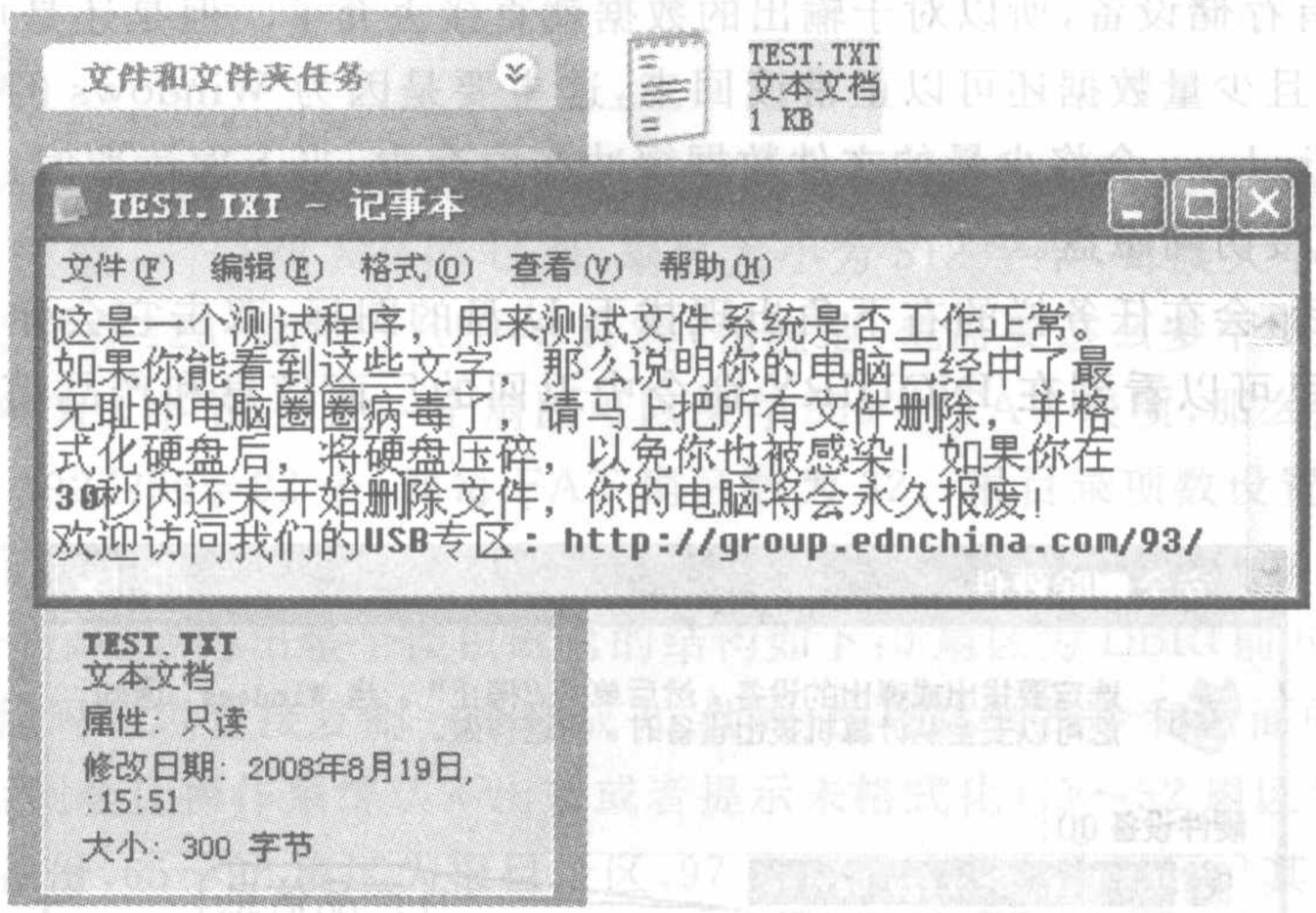


图 8.11.3 测试文件中的内容

8.12 IDE 转 USB 的实现

在上面假 U 盘的基础上，实现 IDE 转 USB 就很容易了。只要在 READ(10)命令处理中从硬盘读取数据，在 WRITE(10)命令处理中将数据写入到硬盘即可。原来程序中模拟磁盘的 DBR、FAT、根目录、数据等都不再需要了，因为它们都保存在硬盘中。当然磁盘容量不再是固定的了，而是由具体的硬盘来决定。可以使用 Identify Drive 命令从硬盘获取硬盘的参数信息，从而获得磁盘容量大小，然后在 READ CAPACITY 命令中返回即可。另外设备描述符中的 PID 应该也要改一改。

不过，操作 IDE 接口也不是那么容易的，需要查看 IDE 接口的文档。IDE 接口的操作跟 D12 的操作有点类似，都是选择不同的寄存器，然后操作。不同的是 IDE 具有多个地址，而 D12 只有 2 个地址。通过往不同的寄存器写不同的值，然后再写一个命令，就可以执行需要的操作了。

由于 IDE 需要 16 条数据线，还有几条控制和地址线，所以在学习板上 IDE 的数据线与按键、LED 是共用的，使用 IDE 时就不能使用按键和 LED 了。还有串口是用来做 IDE 接口的读/写控制的，在使用 IDE 时不能使用串口，并且需要将串口的两个跳线冒拔下。另外，IDE 硬盘通常需要两组电源(12 V 和 5 V)，可以找一个闲置的 PC 电源来给硬盘供电。将 PC 电源中的绿线和黑色的地线连在一起，就可以启动电源。

IDE 硬盘可以设置为逻辑块地址(LBA)访问方式，这样在读/写命令时，可以直接将收到的 LBA 地址写入到硬盘的 LBA 寄存器中。通过配置 Drive/Head 寄存器来选择为 LBA 模

式；另外，还有一个扇区计数(sector count)寄存器，用来保存读/写操作时需要传输的扇区个数。配合 LBA 寄存器、扇区计数寄存器就可以使用 Read Sector 或 Write Sector 命令来读或写扇区了。

需要注意的是，IDE 口的数据线是 16 位的，每次读/写操作都是 2 字节的数据，操作时序可以参看 IDE(ATA)协议。

具体如何实现在这里就不再详述了，仅在光盘中给出实现的程序，感兴趣的读者可以参考阅读。

8.13 本章小结

本章通过对 U 盘的描述符以及命令的分析，实现了一个简单的假 U 盘，并在最后简单地介绍了 IDE 转 USB 的实现。当然，这里仅是一个实验性地实现，要真正实现一个产品级的 U 盘，还有很多路要走。例如，U 盘通常选择 FLASH 作为存储媒介，而 FLASH 有一个特点就是擦除必须按块操作。在没有大容量 RAM 缓冲的条件下实现 FLASH 块的擦写是不容易的。另外，如何做一个有效的算法来尽量平均地使用 FLASH 中的各块，以增加存储器的寿命也是一个难题。此外还要尽可能快地提高传输速度。

如今，自己再来开发 U 盘已经没有多大的商机，因为这东西现在几乎是白菜价了。使用通用 USB 芯片来开发 U 盘那更是不值得了，可以考虑使用专门的 U 盘芯片。但是，学习 U 盘的开发也并不是完全无用了，在很多场合需要在现有的系统中整合一个 U 盘的功能，例如：具有 USB 接口的手机、便携式多媒体播放器、仪表仪器(可以通过 U 盘的方式将数据导入到计算机)、固件更新(可以利用 U 盘功能，将固件直接复制进去来实现更新)等等。在这些场合可以由程序实现或者模拟一个 U 盘，而不宜直接使用专用的 U 盘芯片。因此，学习 U 盘开发的主要价值就在于在已有 USB 接口的基础上，增加一个 U 盘的功能。另外，对开发具有读/写 U 盘功能的嵌入式 USB 主机也有一定的帮助，因为当你知道设备的工作原理之后，设计主机的思路也就有了。

第9章

自定义 USB 设备及驱动开发

在前面所给出的实例中，使用的都是 Windows 自带的驱动，无需用户自行开发。但有时无可避免地要自己开发一个驱动，例如，使用用户自定义的 USB 设备。本章将介绍用户自定义 USB 设备以及其驱动程序的开发。

9.1 用户自定义 USB 设备

在 USB 设备类中，规定类代码 0xFF 为用户自定义 USB 设备。因此，只要在设备描述符中将设备类(bDeviceClass 字段)改成 0xFF，即可实现用户自定义的 USB 设备。

本章的自定义 USB 设备将使用 D12 的端点 1 和端点 2：端点 1 为中断端点，端点 2 为批量端点。为了演示数据通信，端点 1 的数据将实现跟自定义 HID 设备一样的功能，而端点 2 则实现类似 USB 转串口的功能，即端点 2 接收到的数据发送到串口，串口接收到的数据则通过端点 2 返回。

由于与自定义 HID 设备的功能有些相似，所以本实例将在自定义 HID 设备的基础上修改。将 MyUsbHid 复制一份，改名为 MyUsbDevcie。

9.1.1 设备描述符

设备描述符将 bDeviceClass 字段改为 0xFF，即指定为用户自定义设备。子类和协议没有规定，值为 0。由于这是第 11 个实验，将产品 ID 改成 0x000B。

9.1.2 配置描述符集合

配置描述符不用修改。将接口描述符中的端点数量改为 0x04，接口类、子类、协议都改为 0。将 HID 描述符删除，同时上面的报告描述符也删除，这里不再用到它们了。当然，返回报告描述符部分的代码也要删除。另外，还需要在最后增加两个批量端点 2 的端点描述符，可以直接从 USB 转串口或者 U 盘程序中复制过来。

9.1.3 字符串描述符

字符串描述符可以根据自己的需要修改。

9.1.4 数据的处理

对于端点1的数据处理，可以不用修改，保留原来的处理即可（具体的功能参看第5章）。端点2的数据处理，可以参考第6章的USB转串口程序，将那里的处理部分移植过来即可，这里不再详述了，读者可以参看光盘中附带的源代码。当然，你可以根据自己的需要来决定对这些数据的处理，这里仅是为了演示。例如你可以做一个LCD或者数码管显示，在应用程序端将计算机的系统时钟发送出去让它显示在上面，或者安装一个温度传感器，在应用程序上显示温度等。

9.2 驱动程序开发简介

关于WDM驱动开发的书籍在市面上也有不少，推荐大家参看《Windows 2000/XP WDM设备驱动程序开发（第二版）》，武安和编著。本章的重点不在于介绍WDM驱动的结构、各种类等，而是从实例的角度出发，一步步带领读者创建一个自己的简易USB驱动。圈圈也是WDM驱动开发的初学者，所以在WDM驱动内容方面就不在这里班门弄斧了，需要详细了解驱动程序开发的读者可以去购买这方面的专业书籍。

9.3 WDM驱动开发编程环境的建立

工欲善其事，必先利其器。要开发一个USB驱动，必须要选择好合适的开发环境。开发WDM驱动必不可少的软件是VC6和DDK（驱动程序开发工具包，Driver Development Kit）。但是光使用DDK来开发驱动太难了，需要学习太多的驱动方面的知识。我们可以选择一个更容易使用的第三方工具：Driver Studio。Driver Studio（以下简称DS）以类的方式对DDK进行了包装，可以使用简单的向导方式来生成一个驱动框架，因为大部分驱动程序的框架都是差不多的。另外，还可能会调用操作系统的一些相关API函数，这需要安装一个平台SDK。

VC++、DDK、SDK、DS等都有不同的版本，本书中所使用的开发环境为：VC++ 6.0、Windows XP DDK、Windows XP SP2 SDK以及DS 3.2。这些软件可以从网上下载，其中SDK、DDK可到微软官方网站上去搜索并下载。

首先安装VC++ 6.0，然后安装SDK、DDK，最后才安装DS 3.2。

由于DS所使用的类库是在DDK库函数的基础上生成的，所以在安装完DS之后首先要编译出一个库文件；否则，在编译WDM驱动时，就会提示找不到库文件vdw_wdm.lib等。编

译库文件的方法如下：启动 VC 6.0，选择菜单项 File→Open Workspace，在弹出的对话框中找到并打开 DS 安装目录下的 VdwLibs.dsw 工程文件（例如，圈圈的目录为"D:\Program Files\Compuware\DriverStudio\DriverWorks\source\"），如图 9.3.1 所示。然后，单击 VC 菜单栏上的 DriverStudio，选择 DDK Build Settings，设置 DDK 所在的安装目录，如图 9.3.2 所示（图中是圈圈的 DDK 路径，读者请根据自己的安装选择）。如果不设置，将不能正常编译。然后选择菜单项 Build→Batch Build，选择需要编译的工程，不同的平台需要选择不同的工程，如图 9.3.3所示。选择好之后，单击 Rebuild All 按钮，编译工程。经过一个漫长的等待之后，库文件就编译好了，可以查看输出窗口看是否有错误发生。如果编译通不过，检查是不是在 Batch Build 窗口中选错了工程。类库编译的操作只需要一次就行了，以后创建其他驱动无需再编译类库。

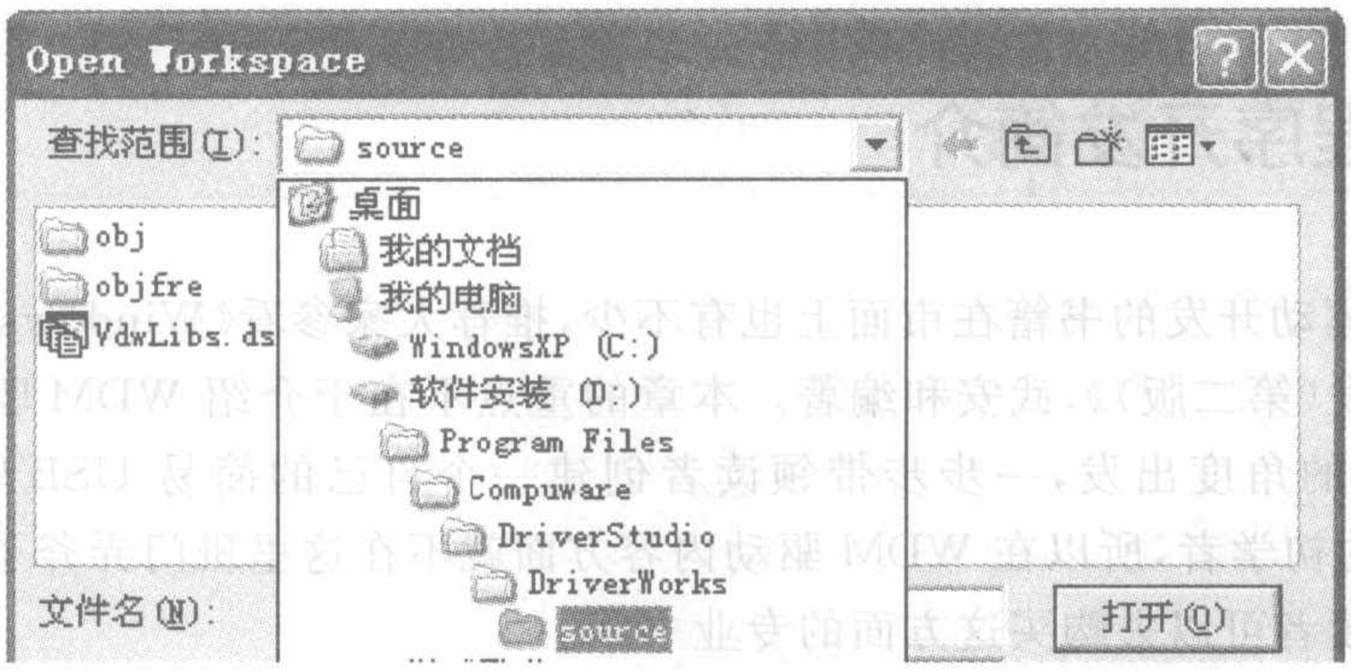


图 9.3.1 打开需要编译的库

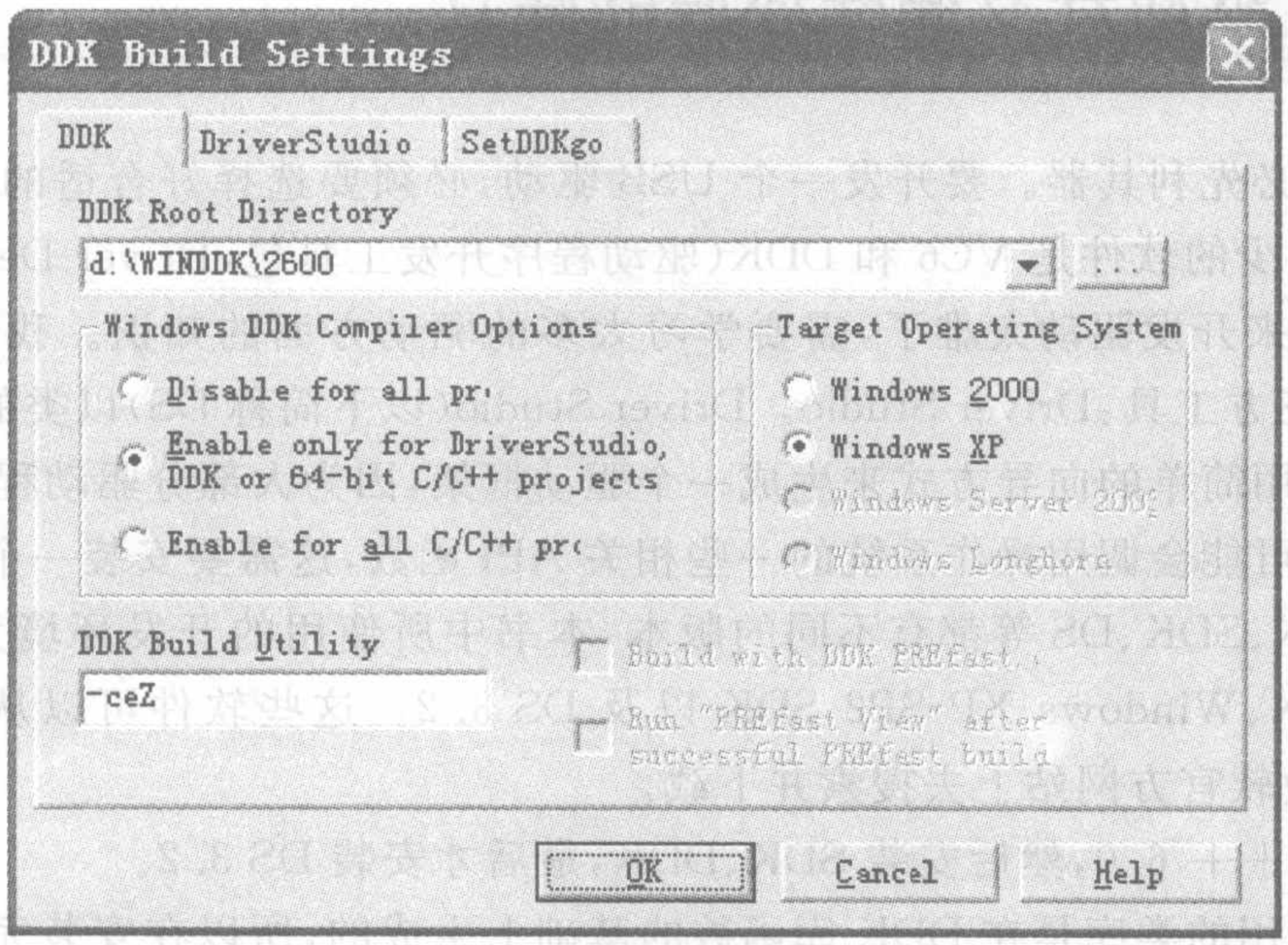


图 9.3.2 选择 DDK 的目录

前面提到要在 DDK Build Settings 里面设置 DDK 的路径，以后每次重新打开驱动程序时，都要先设置一下 DDK 路径。这很麻烦，不知道为什么安装时没有自动设置好。在环境变量的系统变量中增加一个值为 DDK 所在目录的变量 BASEDIR 即可解决此问题，如图 9.3.4 所示。进入环境变量的方法是：右击“我的电脑”，选择“属性”，然后选择“高级”标签，在最下方就可以看到“环境变量”按钮了。

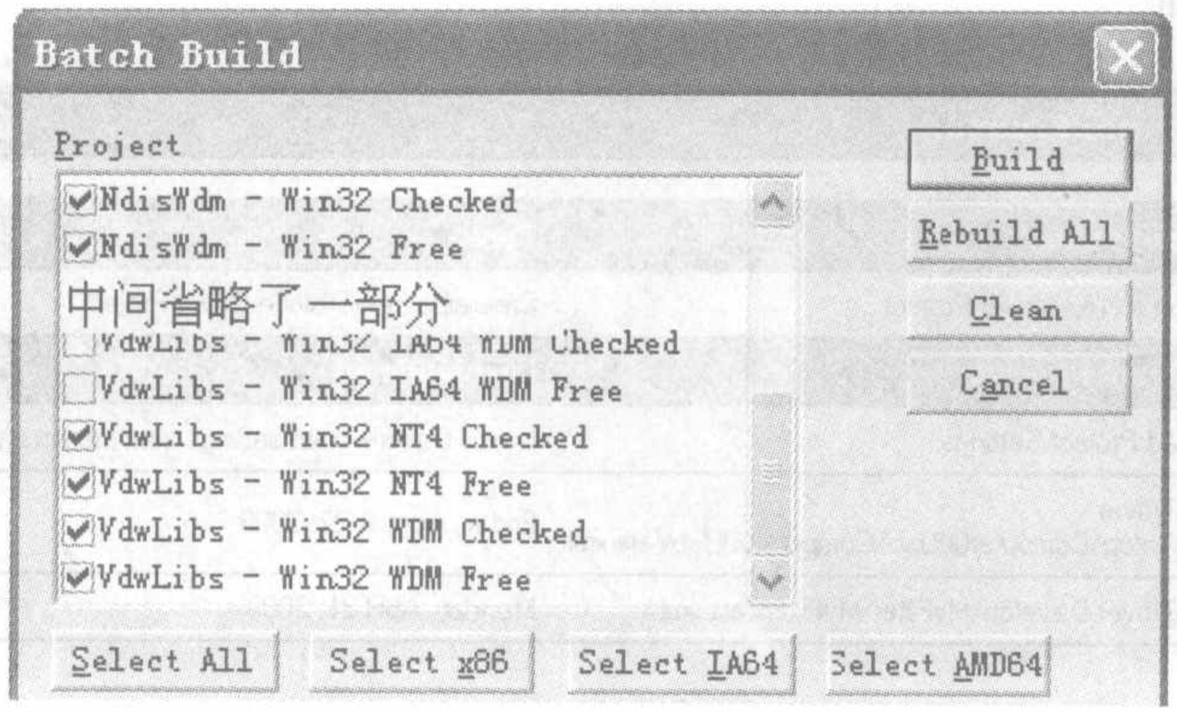


图 9.3.3 选择需要编译的工程

图 9.3.4 设置环境变量

9.4 创建一个USB WDM驱动程序

当安装完DS之后，会在VC的菜单栏下增加一个Driver Studio的菜单项，其中有一个驱动向导的菜单DriverWizard，单击它，将会弹出驱动向导对话框，如图9.4.1所示，不同版本的DS可能稍微有点区别。

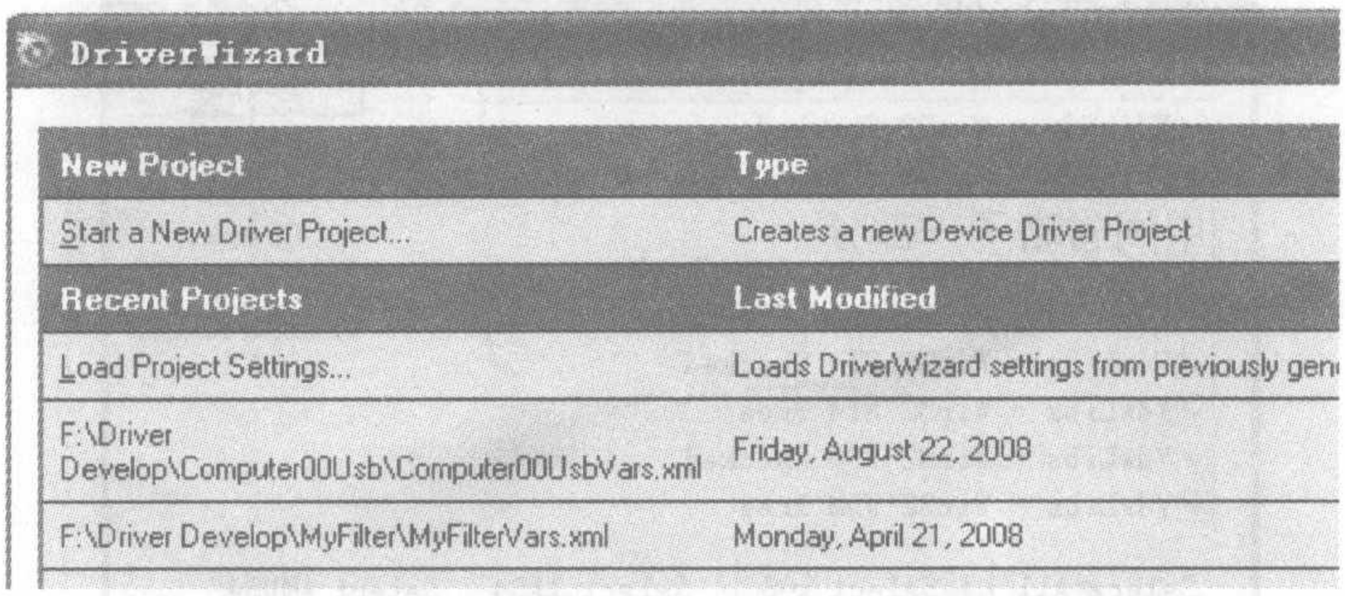


图9.4.1 驱动向导对话框

单击Start a new Driver Project将创建一个新的驱动程序工程，Recent Projects中显示的是最近使用过的工程。下面将按照向导的步骤一步步来介绍如何创建一个名为Computer00Usb的工程。

(1) 设置工程路径以及工程名。在弹出的输入工程名对话框中，将工程名设置为Computer00Usb，并在下面选择一个合适的目录，然后单击Next按钮，如图9.4.2所示。

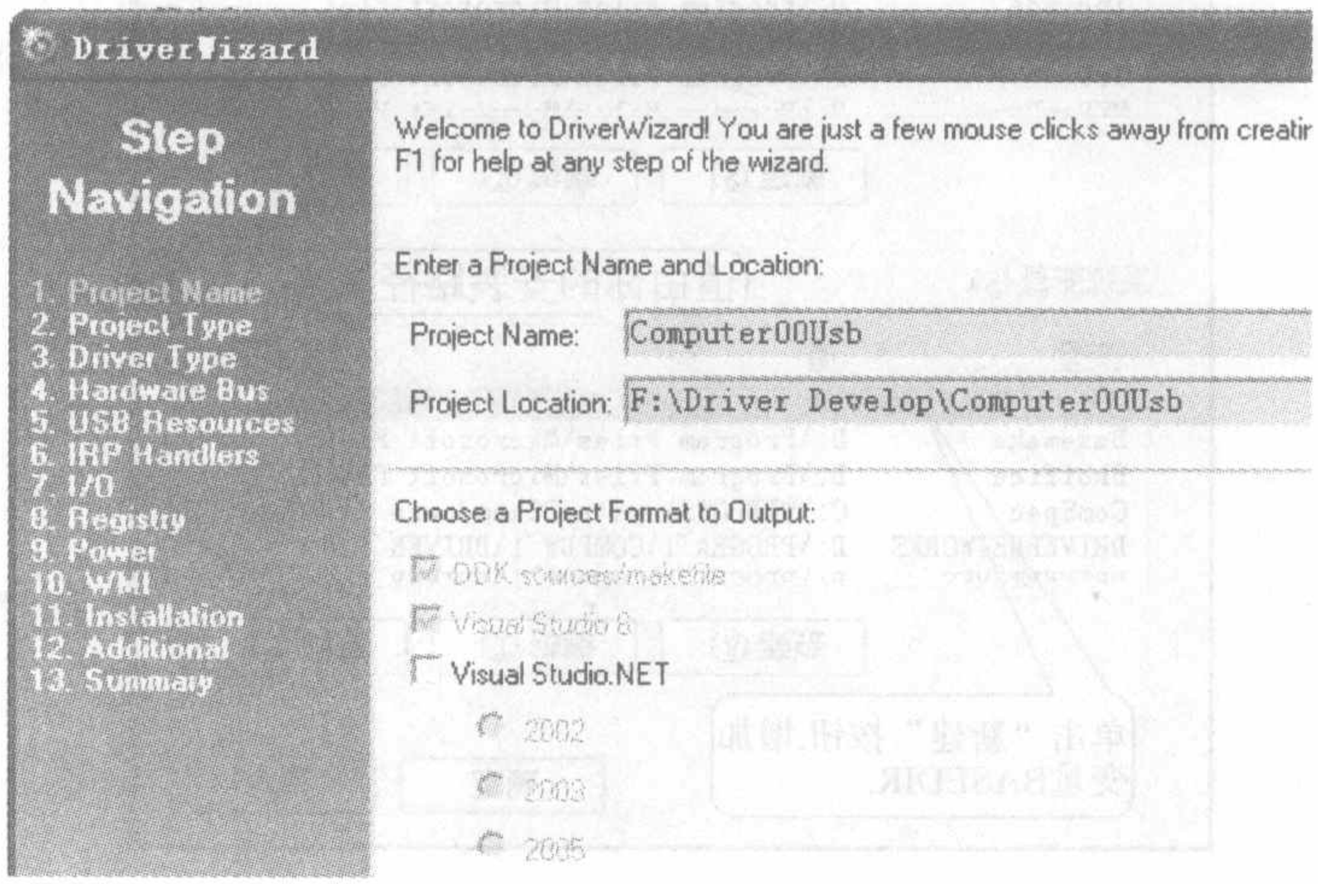


图9.4.2 设置工程目录和工程名

(2) 选择工程的类型。在弹出的设置工程类型对话框中选择 WDM 驱动(因为 USB 驱动属于 WDM 驱动),如图 9.4.3 所示。驱动框架选择为 C++框架,单击 Next 按钮。

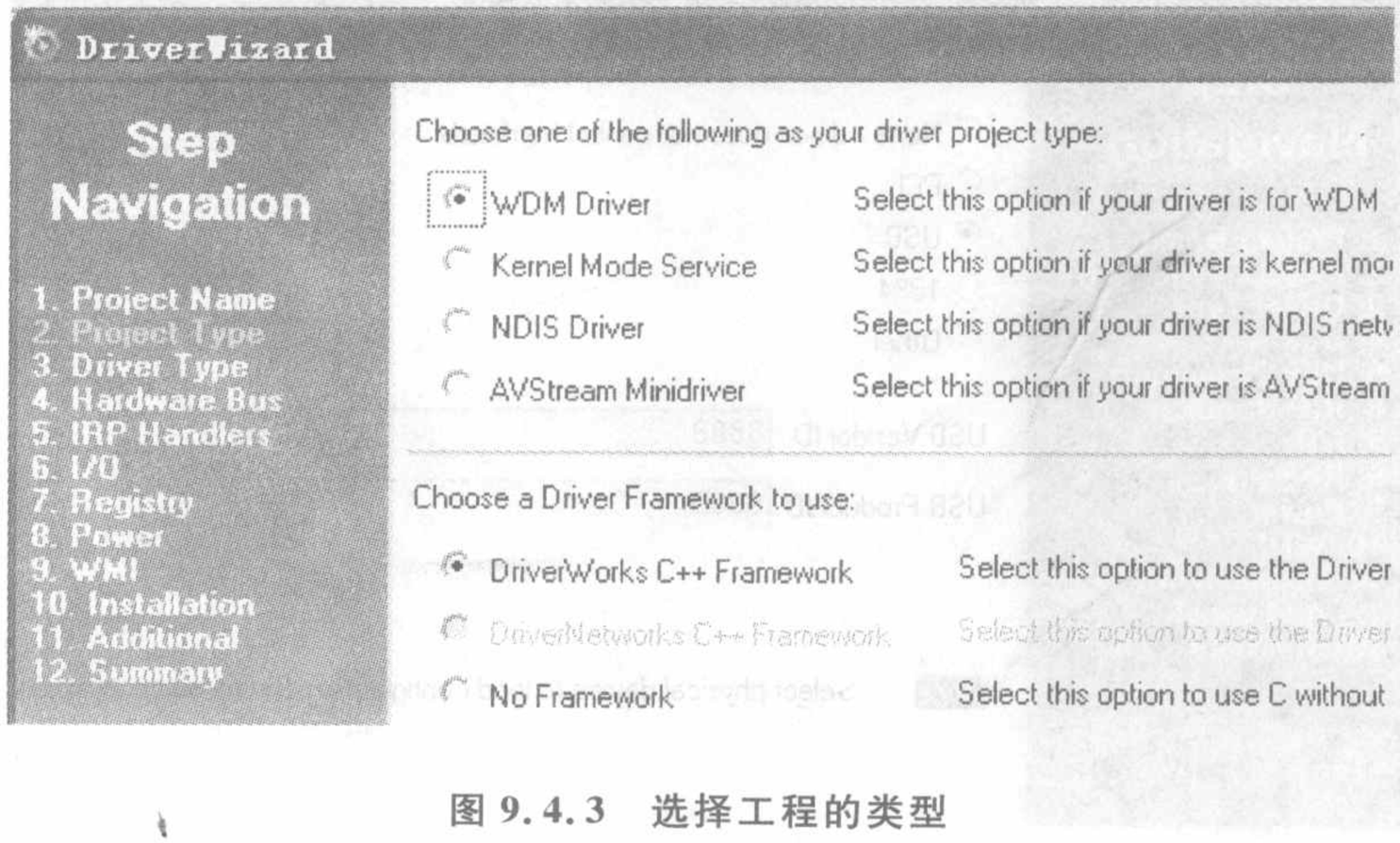


图 9.4.3 选择工程的类型

(3) 选择驱动的类型。在弹出的选择驱动类型的对话框中选择 WDM 功能驱动(WDM Function Driver),单击 Next 按钮,如图 9.4.4 所示。本驱动是为了产生一个新的设备,所以要选择功能驱动程序。

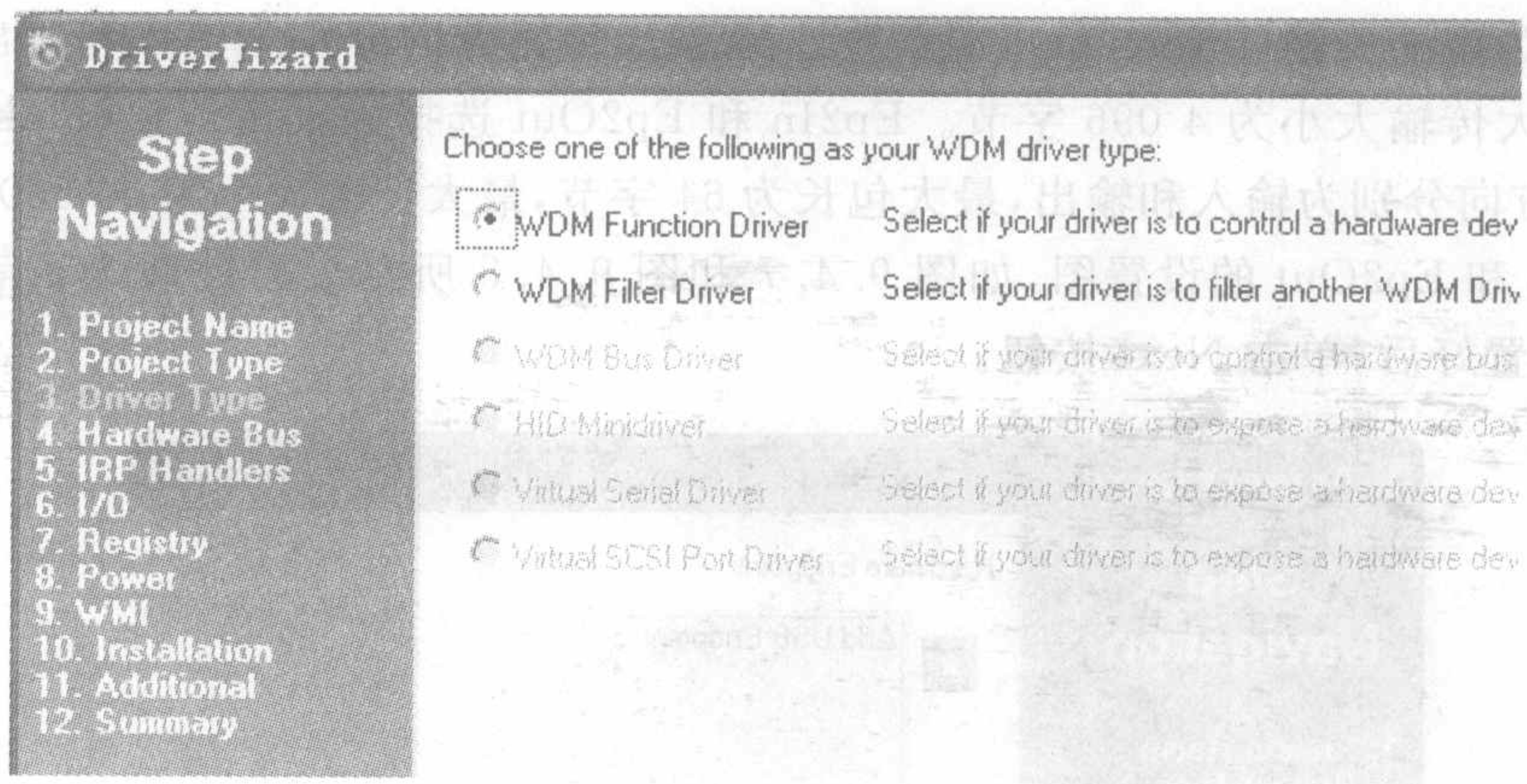


图 9.4.4 选择驱动的类型

(4) 选择硬件总线的类型。在选择硬件总线类型的对话框中,选择为 USB 总线,如图 9.4.5所示。这时将在下面出现设置 USB Vendor ID 和 USB Product ID 的文本框,在这里分别填入在固件中设定的 VID(8888)和 PID(000B),然后单击 Next 按钮。

另外,还可以直接通过单击下面的 Select physical device to load configuration and resource information from 左边的按钮选择一个需要的设备,当然这需要你的设备已经接入到

了计算机才可以找到。

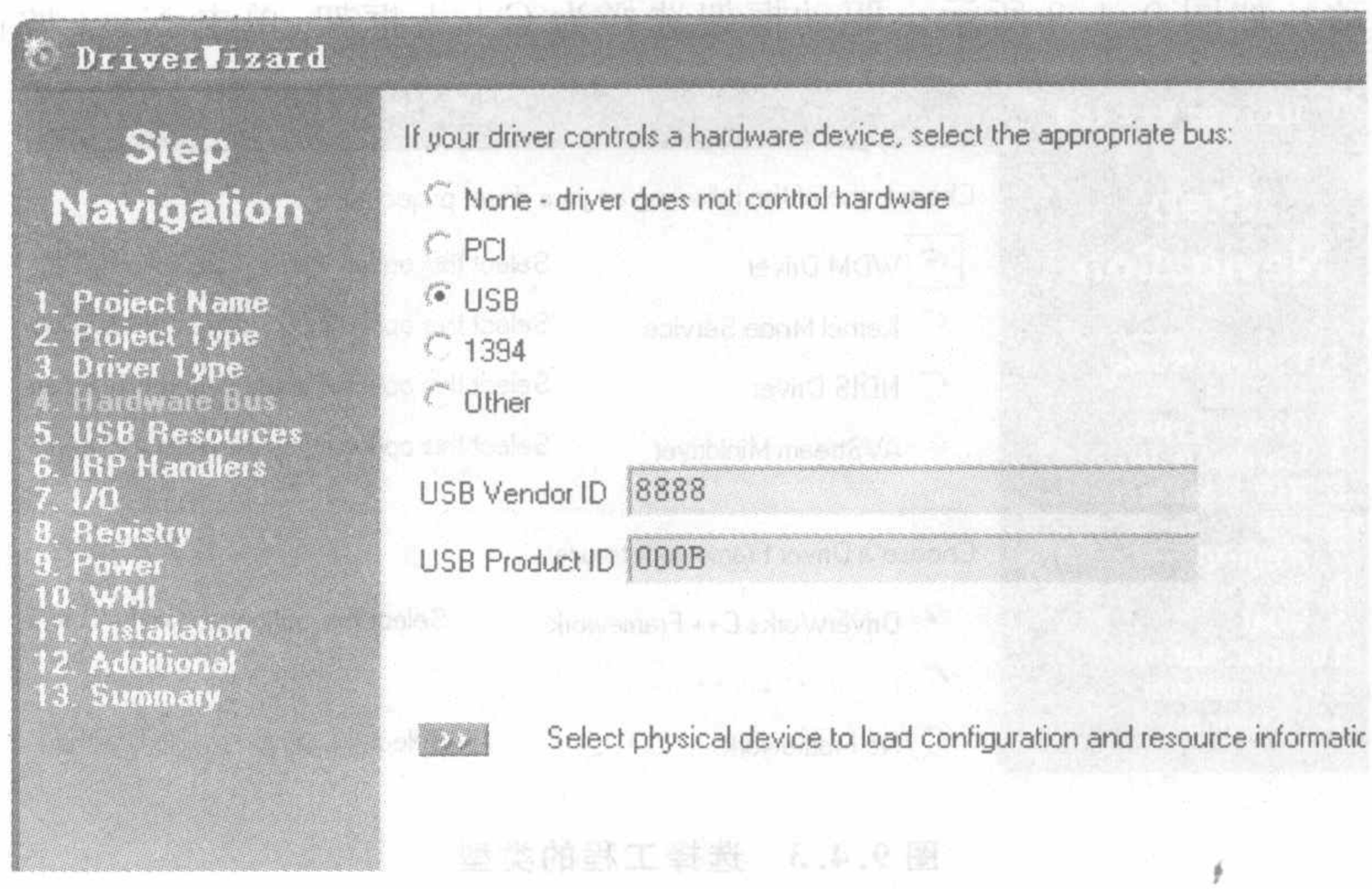


图 9.4.5　选择硬件总线类型

(5) 设置 USB 资源。如图 9.4.6 所示，单击 Add 按钮增加需要的端点。在这里需要增加 4 个端点，管道名(pipe name)分别为 Ep1In、Ep1Out、Ep2In 和 Ep2Out。其中 Ep1In 和 Ep1Out 选择为中断传输(interrupt)，端点地址都为 1，传输方向分别为输入和输出，最大包长为 8 字节，最大传输大小为 4 096 字节。Ep2In 和 Ep2Out 选择为批量传输(bulk)，端点地址都为 2，传输方向分别为输入和输出，最大包长为 64 字节，最大传输大小为 40 960 字节。这里仅给出 Ep1In 和 Ep2Out 的设置图，如图 9.4.7 和图 9.4.8 所示。另外两个读者自己填写。4 个端点都设置好后，单击 Next 按钮。

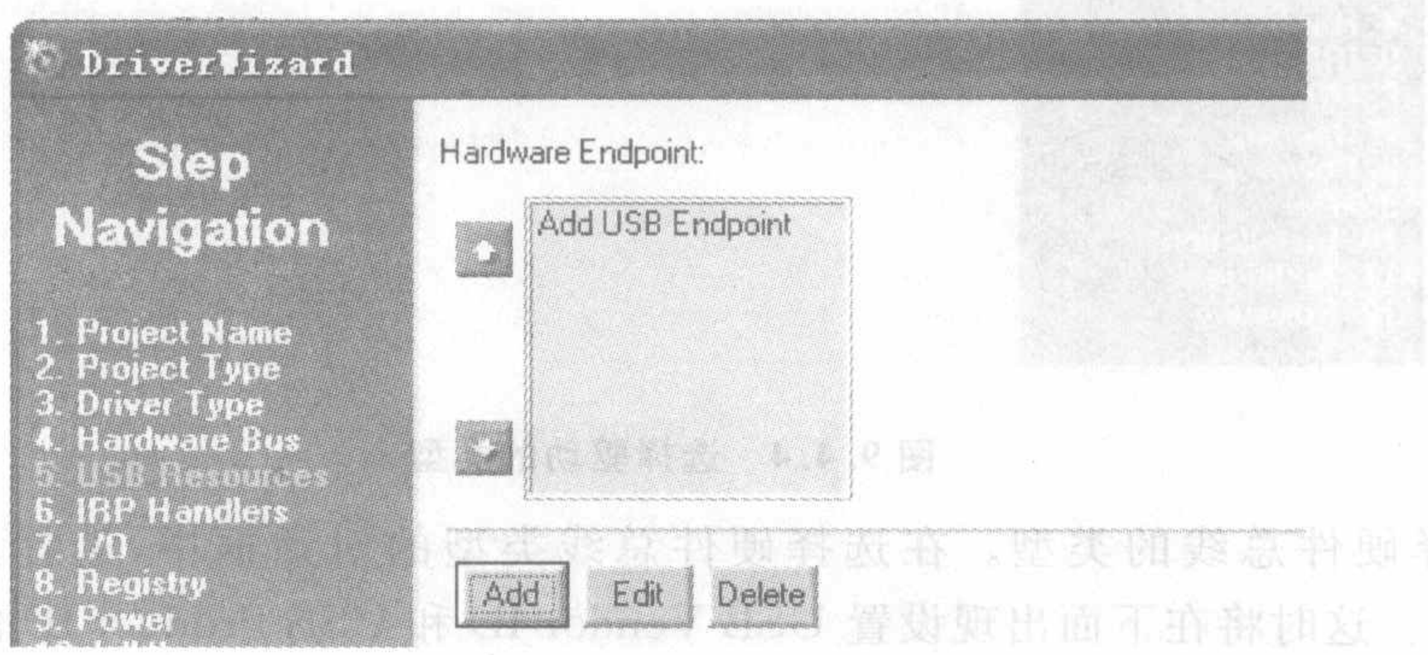


图 9.4.6　设置 USB 资源对话框

(6) 选择需要处理的 IRP。在这里主要处理的就是 IRP_MJ_DEVICE_CONTROL、IRP_

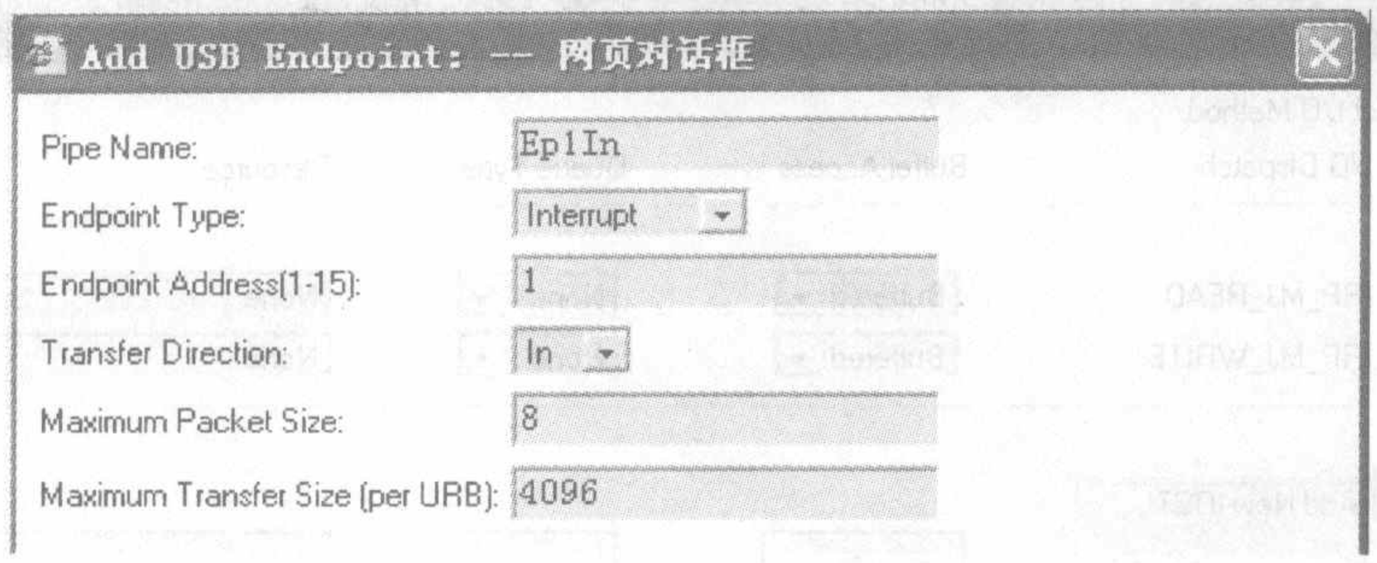


图 9.4.7　端点 Ep1In 的设置

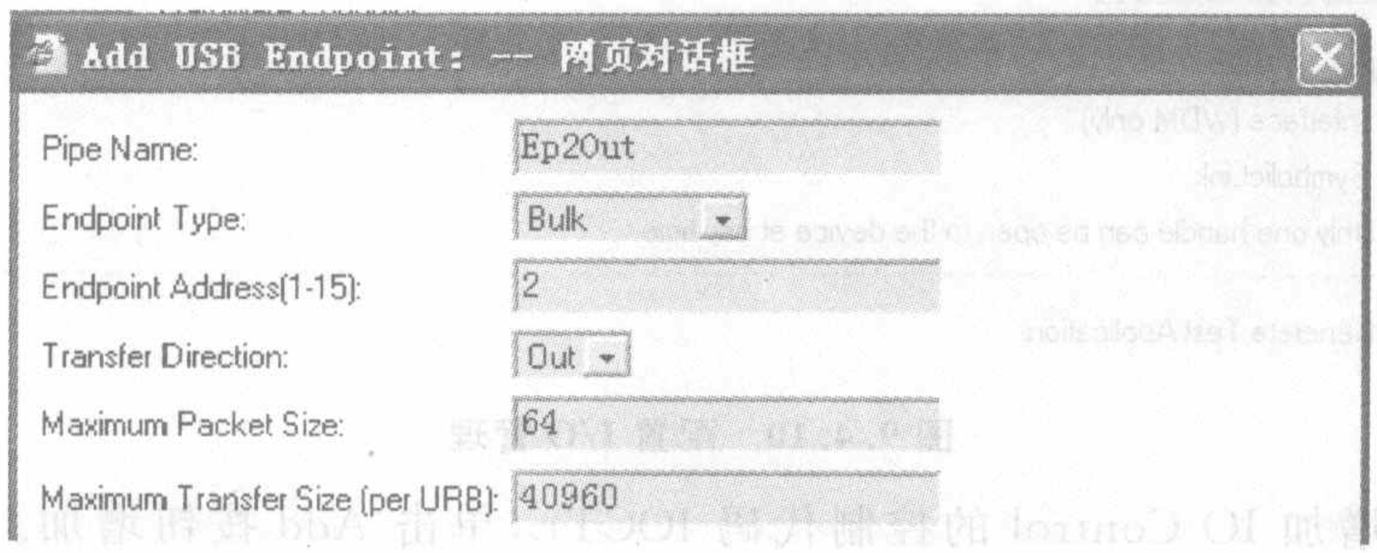


图 9.4.8　端点 Ep2Out 的设置

MJ_READ 以及 IRP_MJ_WRITE。它们分别对应着 API 函数 DeviceIoControl、ReadFile 以及 WriteFile 的响应。单击下面的 Show Minor Pnp 或 Show Other IRPs 可以看到更多的 IRP，可以根据自己的需要进行选择。如图 9.4.9 所示，选择好之后单击 Next 按钮。

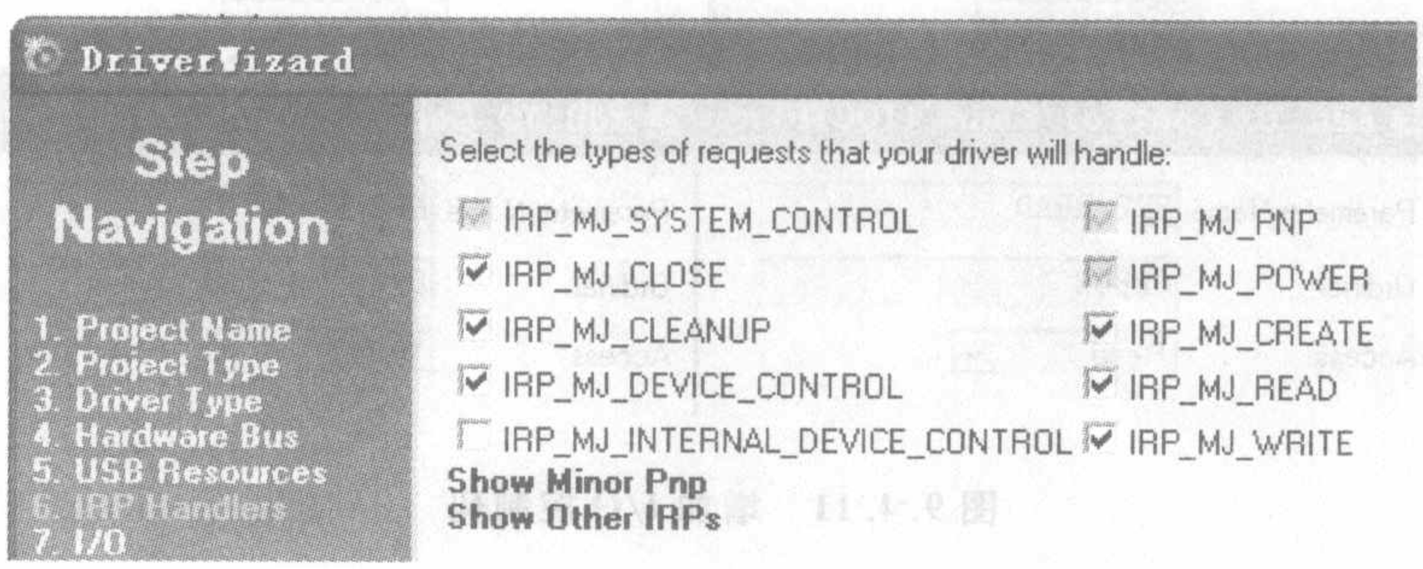


图 9.4.9　选择需要处理的 IRP

(7) 配置 I/O 管理，如图 9.4.10 所示。其中 IRP_MJ_READ 和 IRP_MJ_WRITE 的缓冲方式分别选择为 Buffered，这是一种比较安全的访问方式。选择不同的缓冲方式，在处理 IRP 时获取缓冲区地址的方式也不一样，具体请参看源代码以及驱动开发相关书籍。

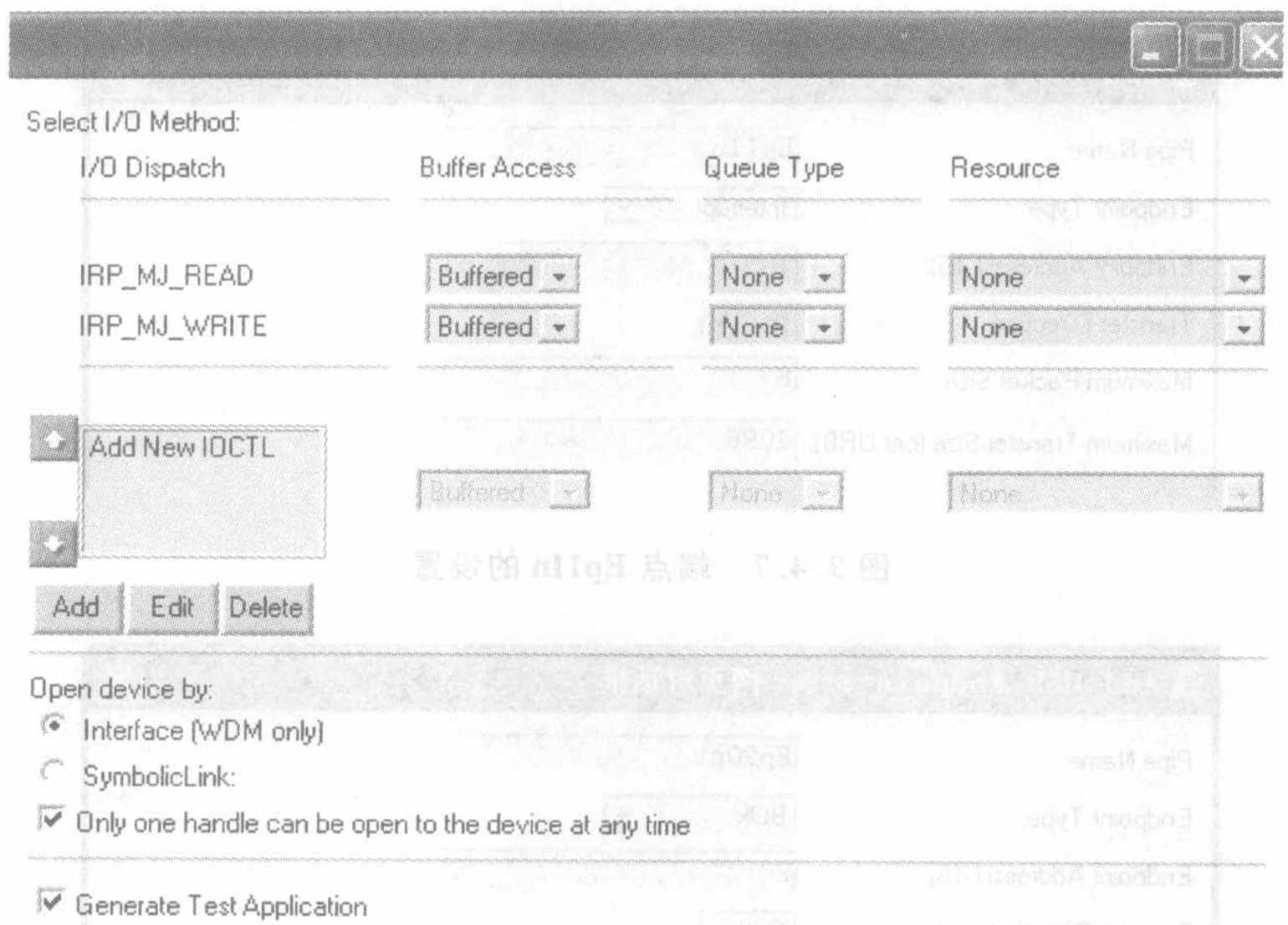


图 9.4.10　配置 I/O 管理

下面还需要增加 IO Control 的控制代码 IOCTL，单击 Add 按钮增加。分别增加 4 个 IOCTL 代码：EP1_READ、EP1_WRITE、EP2_READ 和 EP2_WRITE，如图 9.4.11 所示。

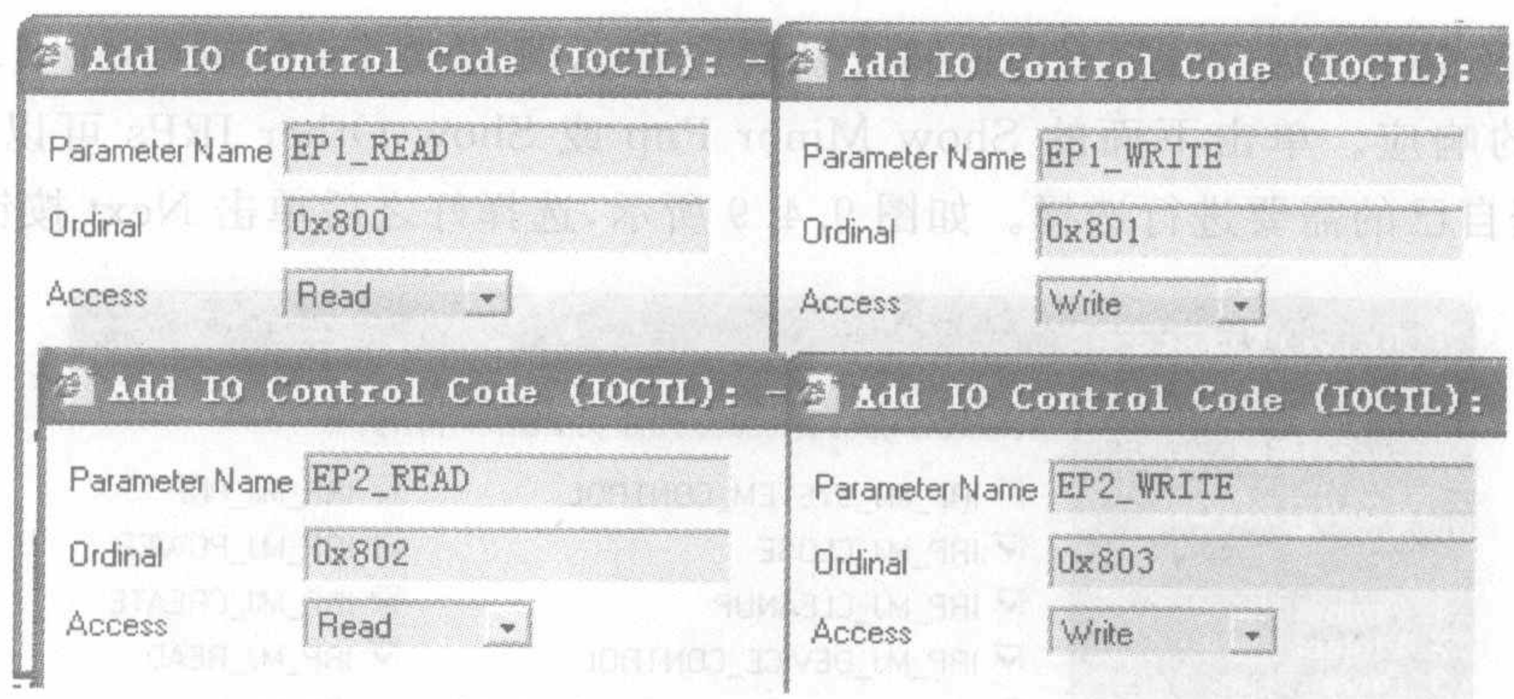


图 9.4.11　增加 I/O 控制码

打开方式选择 Interface（即使用 GUID 的方式打开），将 Only one handle can be open to the device at any time 勾选上，这将限制该驱动程序仅能够被打开一次，以避免多个应用程序同时打开时造成操作混乱。将 Generate Test Application 也勾选上，这将会增加一个测试工程，以方便调试，也可以学习如何打开和操作设备。

（8）增加注册表项，如图 9.4.12 所示。这里为用户自己需要增加的注册表项，可以根据

需要增加，例如可以增加一些设备名称、公司名称、版本号之类的东西，也可以不增加。这里为了演示，仅增加一个设备名(device name)的注册表项。Value Name为出现在注册表中的表项名，Type为所增加的类型，这里是字符串，所以选择REG_SZ即可，Root为该表项需要增加到哪个根键下，SubKey为该表项出现在哪个子键下，Default Value为驱动安装时设置的默认值，Driver Variable Name是该项在驱动程序中出现的变量名。

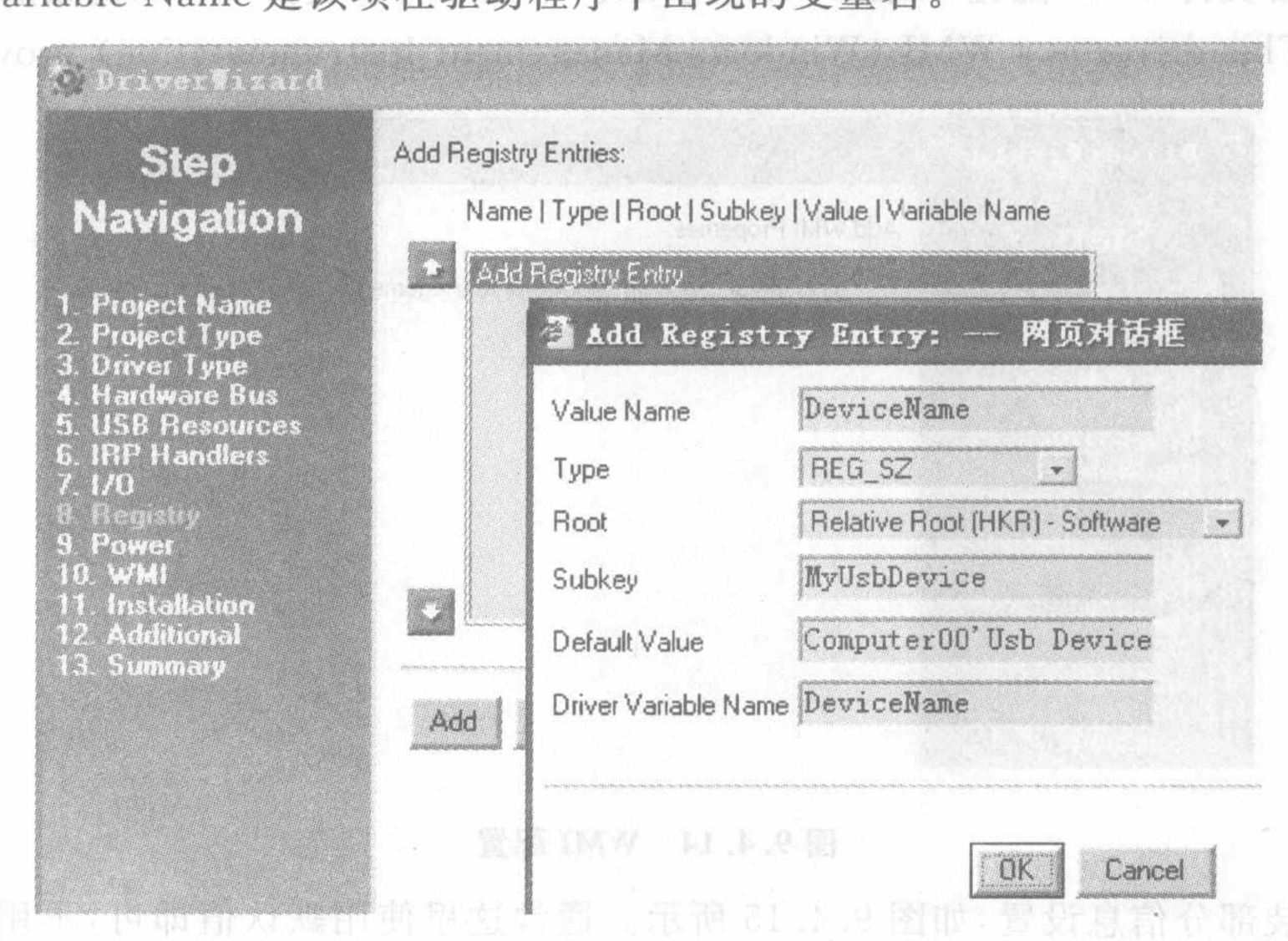


图9.4.12 增加注册表项

(9) 配置电源管理，如图9.4.13所示。将Device requires an inrush of power at startup

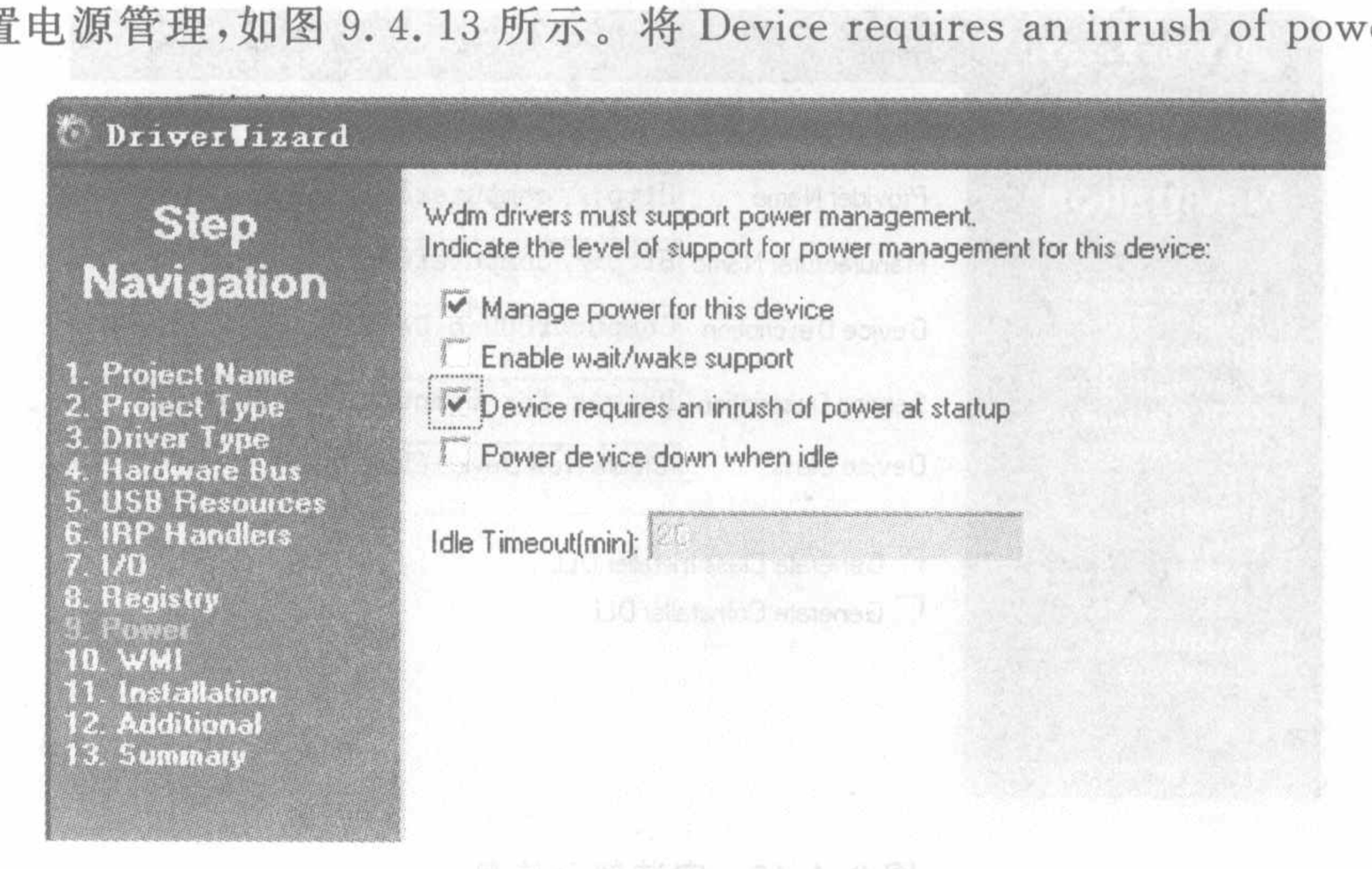


图9.4.13 配置电源管理

勾选上，这说明该设备启动时需要大电流，设置该属性可以保证不会多个这样属性的设备同时上电，从而减少对电源的冲击。Power device down when idle 不要勾选，它表示当设备空闲（没有 IRP 操作）一段时间（由下面的 Idle Timeout 指定，单位为分钟）后，就进入关闭状态（这时 USB 设备将进入挂起状态）。

（10）是否支持 WMI 配置，如图 9.4.14 所示。本程序将不支持 WMI（比较麻烦和复杂），所以不勾选 This driver is a WMI（Windows Management Instrumentation）provider。

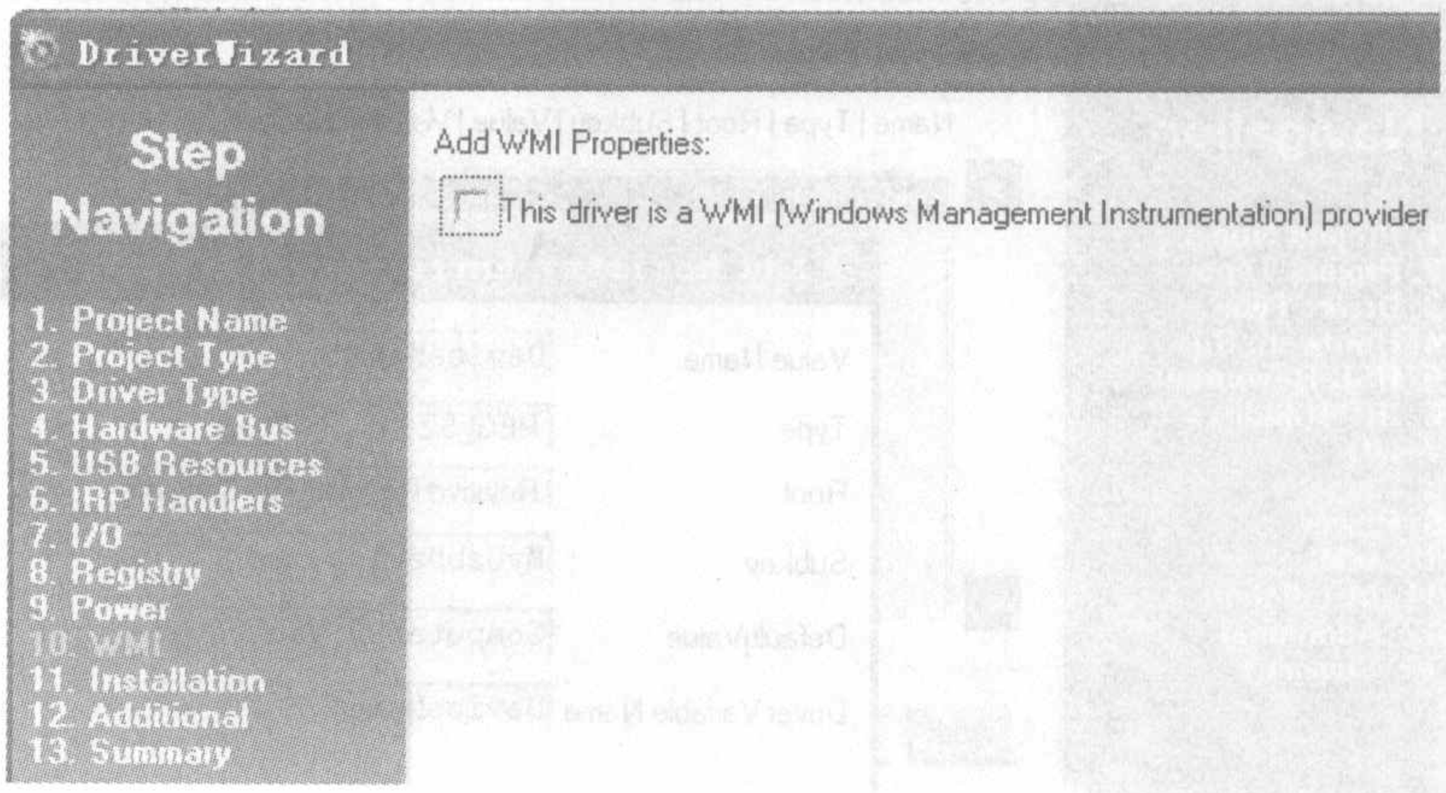


图 9.4.14 WMI 配置

（11）安装部分信息设置，如图 9.4.15 所示。通常这里使用默认值即可，不用修改。Provider Name 为供应商名；Manufacture Name 为厂商名；Device Description 为设备描述，也就

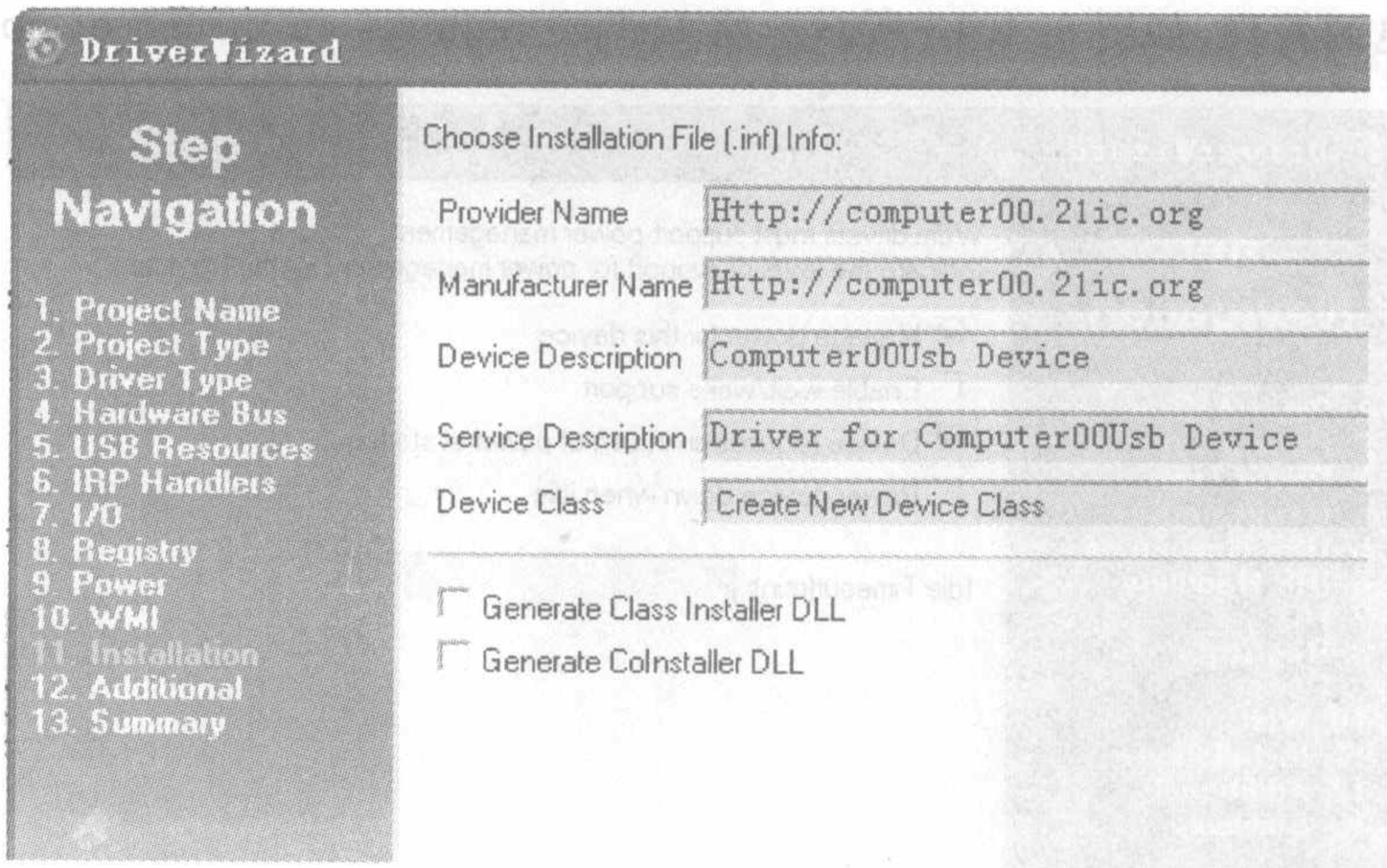


图 9.4.15 安装部分信息

是出现在设备管理中的设备名；Service Description 是出现在注册表中驱动的服务名。Device Class 可以选择创建一个新类（这里就是），也可以选择已经存在的类，例如 USB。

（12）附加信息。这里不用设置，使用默认值即可，如图 9.4.16 所示。

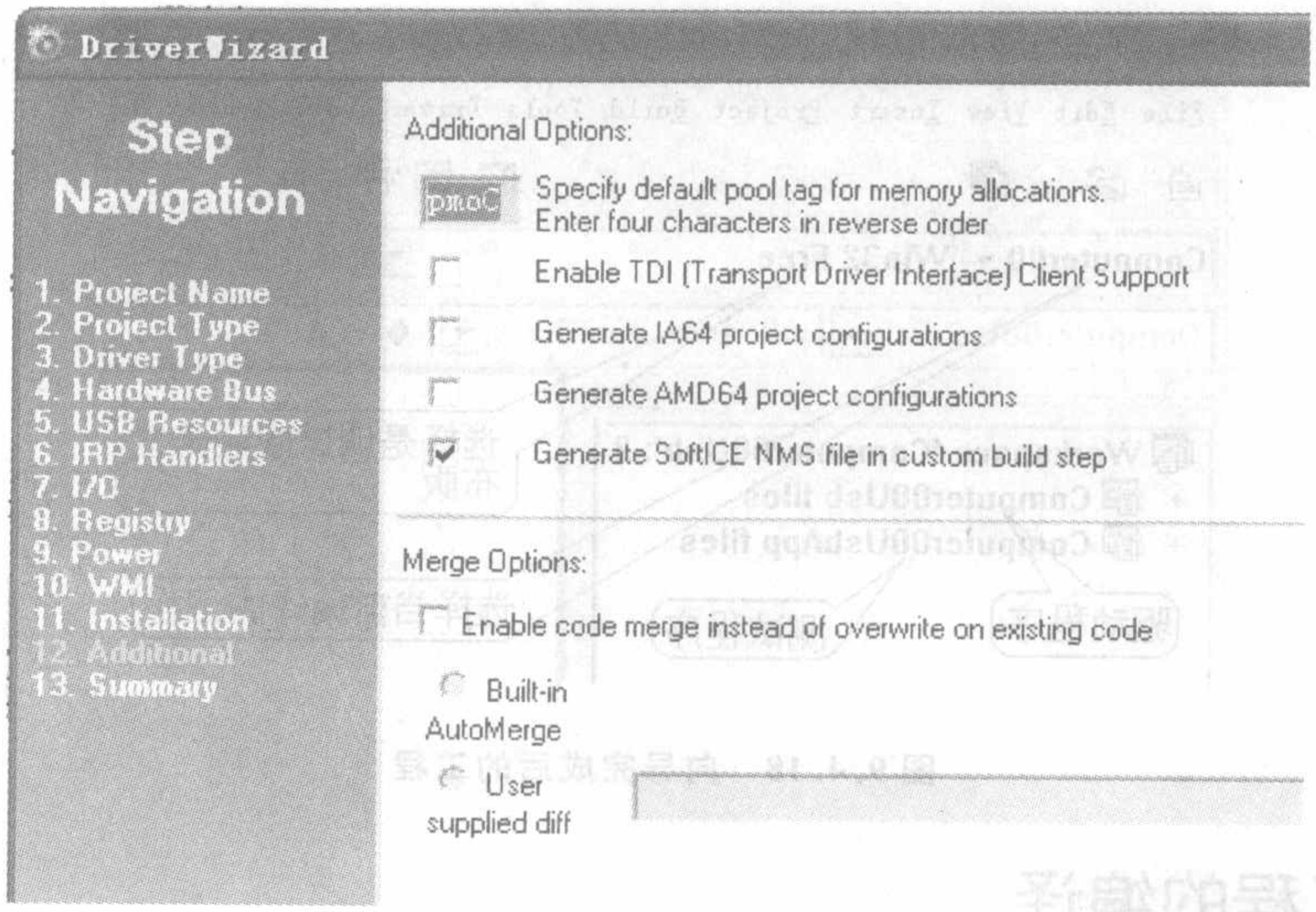


图 9.4.16 附加信息

（13）查看总结信息，在这里可以大致检查一下设置是否正确，如果发现有错误，还可以后退修改。确认无误后，就可以单击 Finish 完成向导了，如图 9.4.17 所示。

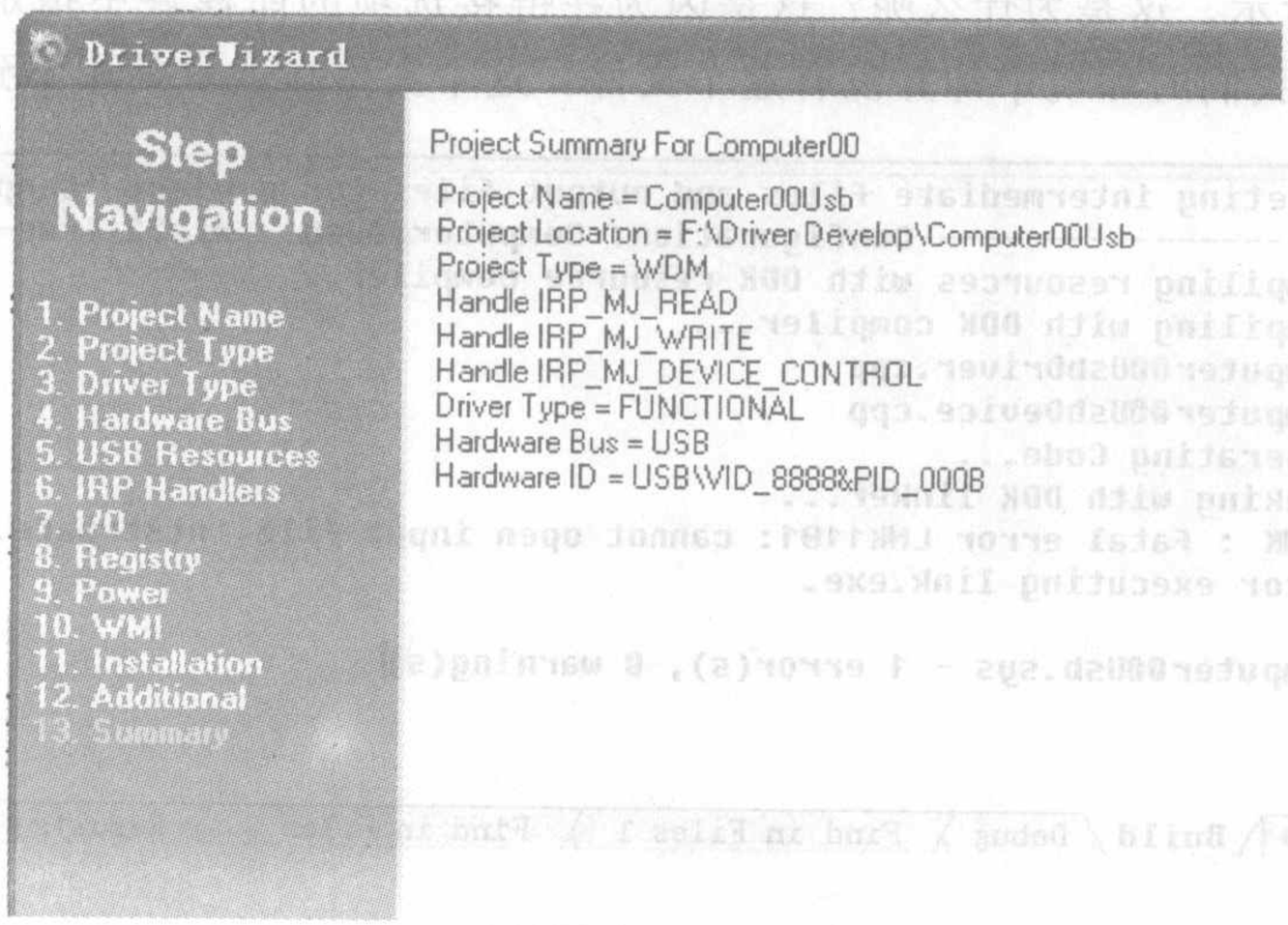


图 9.4.17 查看总结信息

向导完成后，就可以看到生成了两个工程。一个是驱动程序，一个测试驱动程序的应用程序，在上面的下拉列表中可以选择当前活动的是哪个工程，如图9.4.18所示。打开这里面的源文件，就已经有很多代码了。

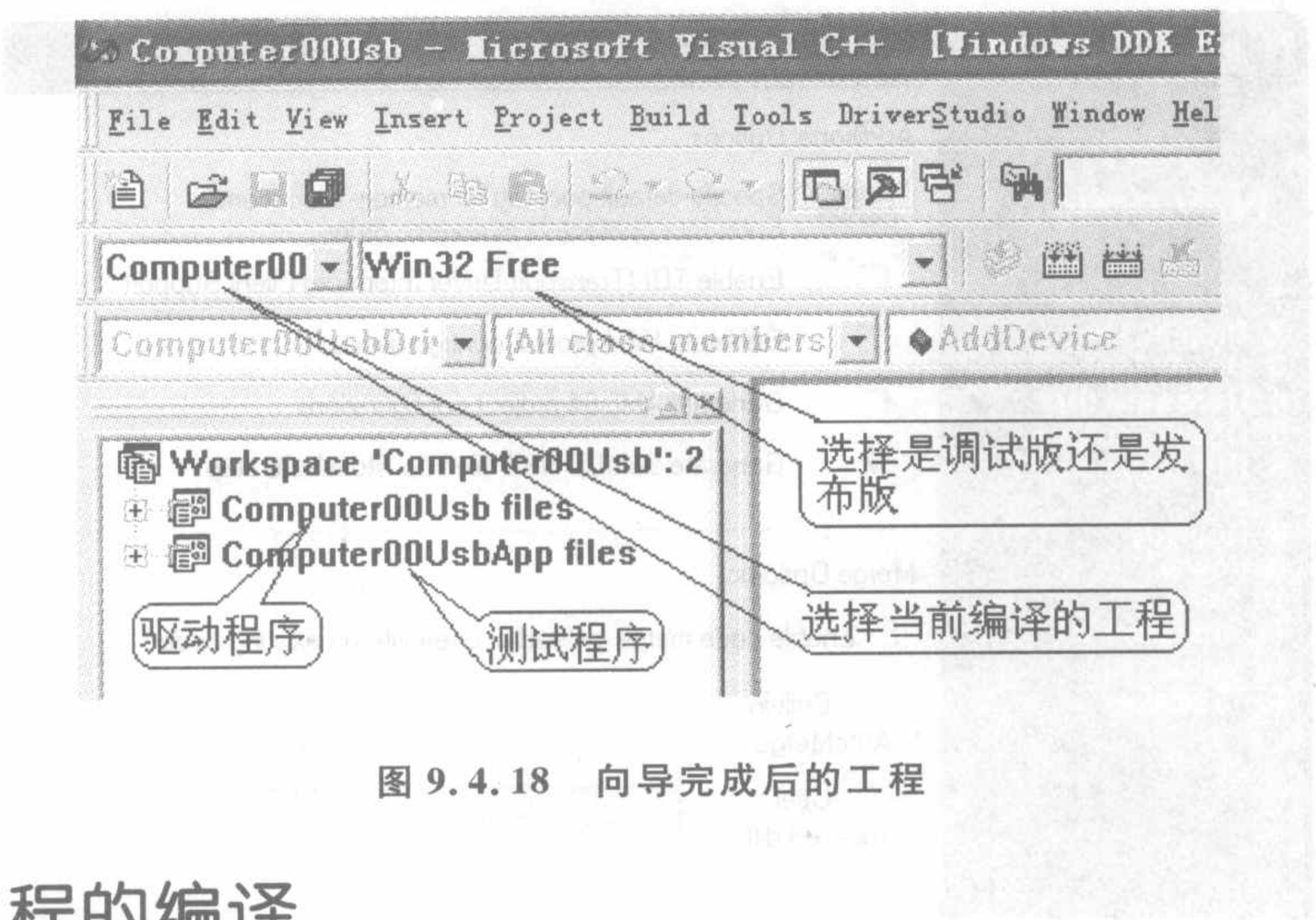


图9.4.18 向导完成后的工程

9.5 对工程的编译

这样一个完整的工程就建立好了，选择驱动工程，很高兴地点击编译。结果却不尽人愿，编译通不过，给出一个错误：LINK : fatal error LNK1181: cannot open input file'ntstrsafe.lib'，如图9.5.1所示。这是为什么呢？这是因为在链接选项的库模块中指定了ntstrsafe.lib这个库文件，但是我们的环境中却并没有这个文件。这个文件在这里是用不到的，只要去工程

```
Deleting intermediate files and output files for project 'Computer
--------------------Configuration: Computer00Usb - Win32 Free-----
Compiling resources with DDK resource compiler...
Compiling with DDK compiler...
Computer00UsbDriver.cpp
Computer00UsbDevice.cpp
Generating Code...
Linking with DDK linker...
LINK : fatal error LNK1181: cannot open input file 'ntstrsafe.lib'
Error executing link.exe.

Computer00Usb.sys - 1 error(s), 0 warning(s)
```

图9.5.1 编译结果

设置中删除掉它即可。操作方法如下：右击 Computer00Usb files，选择 Settings，这时会弹出工程设置对话框。选择 Link 选项卡，Category 列表框选择 General，在 Object/Library modules 中将 ntstrsafe.lib 删除即可，如图 9.5.2 所示。注意不要同时选中两个工程，还有上面的 Win32 Free 和 Win32 Checked 要分别设置，因为这两个的设置是独立的。

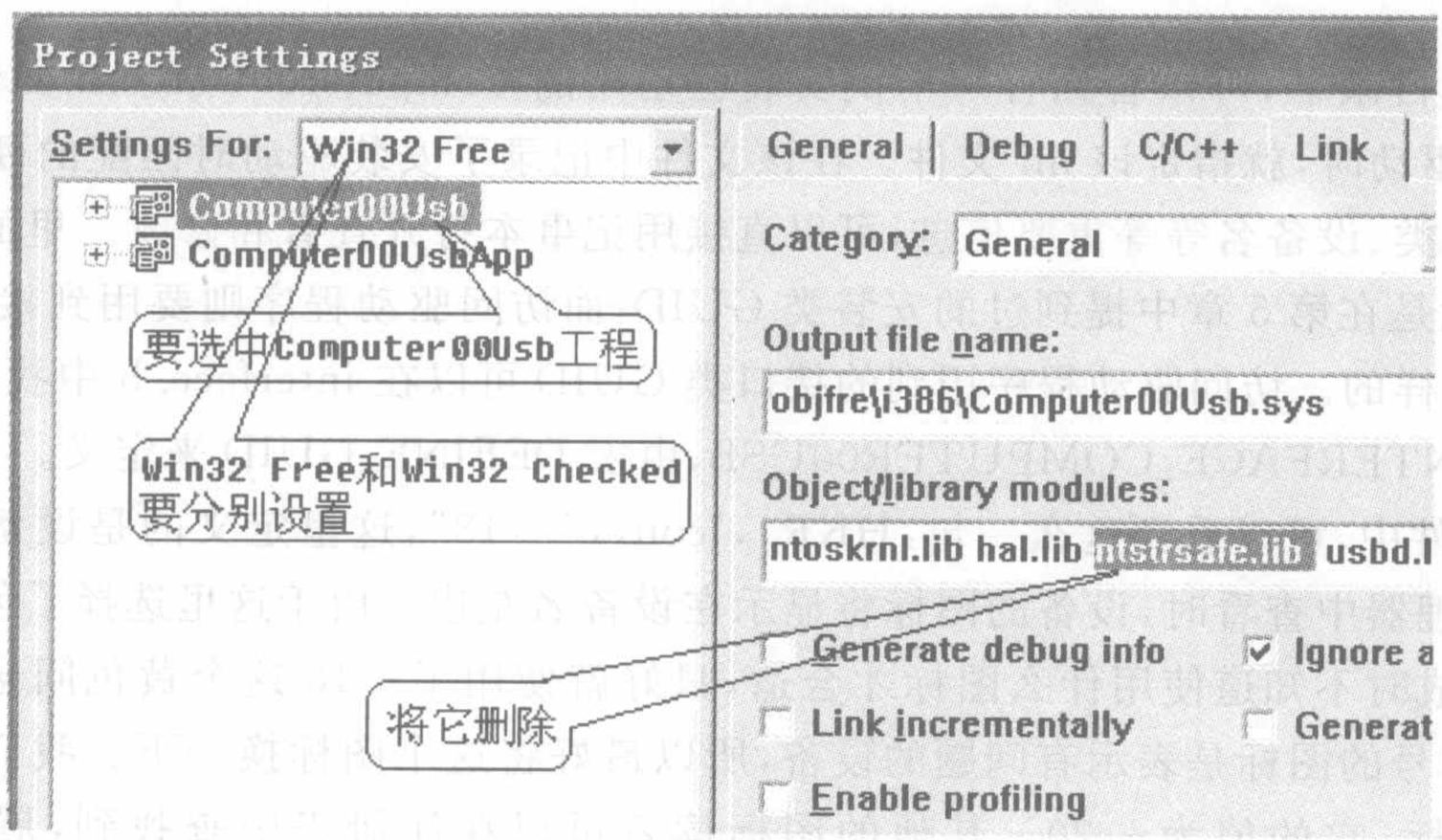


图 9.5.2　工程设置窗口

然后再次编译，就可以看到编译顺利通过了，显示“0 error(s)，0 warning(s)”，如图 9.5.3 所示。

可以看到编译报告中有一行“MODULE=.\objfre\i386\Computer00Usb.sys”，这个就是我们最后所需要的驱动文件。这里选择的是发行版，所以生成的 sys 文件在 objfre 目录下，它不包含调试信息。如果选择的是调试版，则 sys 文件在目录 objchk 目录下，并包含调试信息。调试、发行驱动程序时要注意，要选择对应的 sys 文件安装。所有跟驱动相关的文件放在目录 Computer00Usb\driver 下，而测试用的应用程序则在 Computer00Usb\app 目录下。

```
Linking with DDK linker...
---------------------------Build SoftICE Symbols---------
Compuware NM32 Symbol Translator/Loader version 4.3.2
(C) Compuware Corporation, 1996-2004
MODULE=.\objfre\i386\Computer00Usb.sys
PROMPT=OFF
SoftICE is not active
Translation of F:\Driver Develop\Computer00Usb\Driver\obj

Computer00Usb.sys - 0 error(s), 0 warning(s)
```

图 9.5.3　成功编译后的输出窗口

顺便提一下的是在编译测试程序时，需要使用DDK编译(编译图标上写着棕色DDK字样的)，直接使用VC6的编译，生成的测试程序文件将无法执行。

9.6 关于inf文件

在driver目录下，可以看到有一个供安装驱动用的Computer00Usb.inf文件，当插上设备，提示安装驱动时，就指定该inf文件。在该文件中记录了安装驱动时匹配的硬件ID号、需要增加的设备类、设备名等等重要信息，可以直接用记事本打开查看和修改。里面有个ClassGUID，这个就是在第5章中提到过的安装类GUID，而访问驱动程序则要用到接口类GUID，这两个是不一样的。访问驱动程序用到的接口类GUID可以在interface.h中找到，其名称为GUID_DEVINTERFACE_COMPUTER00USB，由宏DEFINE_GUID来定义。

在inf文件中，可以看到这么一句：HKR,,Icon,,"-18"，这里定义的是设备所使用的图标，在设备管理器中查看时，设备的图标将显示在设备名左边。由于这里选择了创建新的设备类，向导在生成时不知道使用什么图标才合适，只好就使用了-18这个黄色问号的图标。通常看到黄色问号的图标是表示有问题的设备，所以最好将这个图标换一下。我们可以指定使用USB的图标，它的值为-20。其他的图标读者可以在注册表中查找到，展开注册表的HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class，下面有很多GUID，这些GUID就是安装时指定的ClassGUID，点击某个GUID，就可以看到该类的名称和所使用的图标(Icon)了。在inf文件最后的[Strings]中，可以看到在向导的第11步所指定的字符串，可以直接在inf文件中修改这些字符串，具体的可以参看第5章中的inf文件描述以及inf文件中的注释，这里就不详述了。关于inf文件的细节，可以参看MSDN。

9.7 驱动程序的修改

虽然现在这个sys文件已经可以正常安装，但是它仅是一个简单的驱动框架，并没有实现具体的数据处理。要实现数据的处理，还需要自己修改和增加代码。

首先要修改的是DS向导生成代码所产生的bug，如果不修改这个bug，访问输入端点时将会蓝屏。打开Computer00UsbDevice.cpp文件，找到Computer00UsbDevice类的构造函数Computer00UsbDevice::Computer00UsbDevice，可以看到如下初始化管道的代码：

```
// Initialize each Pipe object
Ep1In.Initialize(m_Lower, 81, 8);
Ep1Out.Initialize(m_Lower, 1, 8);
Ep2In.Initialize(m_Lower, 82, 64);
Ep2Out.Initialize(m_Lower, 2, 64);
```

对于输入端点1和2地址，应该分别为0x81和0x82才对，这里漏掉了0x，将其添上。

在下面还可以看到一个LoadRegistryParameters()函数，它就是负责将在向导中添加的注册表信息读进来的，可以在这里对读入的注册表信息作相关处理。

接下来就是修改数据处理的函数了，它们都在Computer00UsbDevice.cpp文件中，主要修改Computer00UsbDevice类的成员函数Read()、Write()及DeviceControl()中调用的4个IoControl()函数。

9.7.1 Read(KIrp I)函数

Read()函数对应着Win32 API函数的ReadFile，其中参数通过KIrp I传递进来。通过调用KIrp类的成员函数，可以获取到缓冲区地址、需要读的字节数等参数。在向导生成的代码中，就有很多可以使用的代码了，例如获取缓冲区地址、读数据的长度等。我们只要在原来的基础上修改就可以了，为了让读者了解修改的过程，将在源程序中保留修改的痕迹，使用"/**/"注释向导生成的代码。

由于指定的缓冲方式为Buffered方式，所以应该使用I.BufferedReadDest()函数来获取保存数据的缓冲区地址，使用I.ReadSize()函数来获取读数据的长度。如果获取到的缓冲区地址为0(NULL)，则说明参数无效，那么以STATUS_INVALID_PARAMETER来完成该IRP，并返回STATUS_INVALID_PARAMETER。如果需要读取的字节数为0，则直接成功完成该IRP，不需要数据处理。相关代码如下：

```
NTSTATUS status = STATUS_SUCCESS;

 //原来向导生成在后面，从后面移动上来
 // Get a pointer to the caller's buffer.
 PUCHAR pBuffer = (PUCHAR) I.BufferedReadDest();
 ULONG readSize = I.ReadSize();
 ULONG bytesRead = 0;
 // TODO: Validate the parameters of the IRP.  Replace "FALSE"
 //    in the following line with error checking code that
 //    evaulates to TRUE if the request is not valid.
 /* if (FALSE) */
 if(pBuffer == NULL)                    //如果准备的读缓冲区为无效地址，则返回参数无效
 {
  status = STATUS_INVALID_PARAMETER;    //状态设置为无效参数
  I.Information() = 0;                  //读取字节数为0
  I.PnpComplete(status);                //完成该I/O操作
//调试信息
  T.Trace(TraceWarning, __FUNCTION__" -- . IRP %p, STATUS %x\n", I, status);
```

```
    return status;                          //返回状态
  }

  // Always ok to read 0 elements
  if (I.ReadSize() == 0)                    //如果读长度为0,那么总是成功的
  {
    I.Information() = 0;                    //读字节数为0
    I.PnpComplete(this, status);            //完成该I/O
//调试信息
    T.Trace(TraceInfo, __FUNCTION__" -- . IRP %p, STATUS %x\n", I, status);

    return status;
  }
```

接下来就要创建一个URB来处理数据。为了能够直接使用前面的自定义HID设备的应用程序代码,这里的Read操作将返回输入端点1的数据。

由于端点1是中断端点,为了从端点1读取数据,必须要创建一个中断传输的URB并提交它,这时下层的USB总线驱动程序就会负责从端点1读取数据,当数据读取完毕后提交URB的函数就会返回。端点管道KUsbPipe类提供了一个创建中断传输URB的函数,创建中断传输URB的函数原型如下:


```
PURB BuildInterruptTransfer(PVOID Buffer,
                            ULONG Length,
                            BOOLEAN bShortOk = TRUE,
                            PURB Link = NULL,
                            PURB pUrb = NULL,
                            BOOLEAN bIn = TRUE);
```

其中,Buffer为指向保存数据缓冲区的地址。Length为需要传输数据的字节数。bShortOk表示实际传输的数据是否可以比请求的短,值为TRUE时表示实际传输的数据可以短于请求的数据量(即数据传输量未达到指定值时提交URB的函数也会返回),值为FALSE时表示实际传输的数据必须要达到指定的长度时,提交URB的函数才能返回。Link表示链接下一个URB的指针,值为NULL表示没有。pUrb表示指向一个已经存在的URB,如果没有,指定为NULL,这时该函数将会为新创建的URB分配空间,记得在使用完毕后释放该URB。bIn表示数据传输的方向,值为TURE时表示输入数据,为FALSE时为输出数据。

在这个驱动类中,共定义了4个端点管道KUsbPipe类的实例,这里是对输入端点1操作,所以使用端点1的管道实例Ep1In(其名称是在向导的第五步增加USB资源时所设置的)来创建这个URB。实际使用的代码如下:

```
//创建一个中断传输的URB,用来从端点1读取数据
```

```
PURB pUrb = Ep1In.BuildInterruptTransfer(
            pBuffer,        //接收数据的缓冲区
            readSize,       //读数据的数据字节数
            TRUE,           //TRUE 表示设备传输的字节数可以少于指定的字节数
            NULL,           //连接下一个传输的 URB,这里没有,置为 NULL
            NULL,           //指向一个已经存在的 URB,置为 NULL,分配一个新的 URB
            TRUE);          //TURE 表示读数据
```

创建好 URB 之后,就需要提交它。使用端点管道 KUsbPipe 类的成员函数 SubmitUrb 来提交这个 URB,函数的原型如下:

```
SubmitUrb(PURB pUrb,
          PIO_COMPLETION_ROUTINE pfnCompletionRoutine,
          PVOID pContext,
          ULONG mSecTimeOut)
```

其中,pUrb 即为指向一个 URB 的指针。pfnCompletionRoutine 是该 URB 完成时调用的一个完成函数,可以设置为 NULL,表示没有函数。pContext 为传递给 pfnCompletionRoutine()函数的环境变量,可以设置为 NULL。mSecTimeOut 为超时时间,单位为毫秒,设置为 0 时表示无限期等待,只有当数据正确读取后该函数才能返回。

在 URB 创建完之后,应该先检查是否分配成功,如果已经成功分配,则可以调用 SubmitUrb()函数提交这个 URB,否则就返回资源不足的错误代码。当 SubmitUrb()函数返回后,就可以使用该 URB 的成员变量 UrbBulkOrInterruptTransfer. TransferBufferLength 来获取实际读取到的字节数,最后还要记得删除前面创建的 URB。这里为了防止读操作时设备长时间无数据返回而导致驱动无法关闭,设置了 3 s 的时间限制。如果设备超过 3 s 还未返回数据,则驱动程序将返回读取数据长度为 0 字节。具体的实现代码如下:

```
if(pUrb == NULL)  //如果分配失败
 {
  status = STATUS_INSUFFICIENT_RESOURCES;     //设置状态为资源不足
 }
 else
 {
  //提交 URB,并设置超时时间为 3 s
  status = Ep1In.SubmitUrb(pUrb,NULL,NULL,3000);
  //获取实际读到的数据字节数
  bytesRead = pUrb->UrbBulkOrInterruptTransfer.TransferBufferLength;
  delete pUrb;                                //删除刚刚分配的 URB
 }
```

最后，还需完成该IRP，在完成IRP之前要先设置传输数据的字节数。一个默认的约定就是使用IRP的Information成员来保存传输的字节数。具体的实现代码如下：

```
//保存传输数据的字节数
I.Information() = bytesRead;
 //完成该 IRP
I.PnpComplete(this, status);
```

至此，Read()函数就算改完了，怎样？简单吧。这主要是依赖了DS强大的驱动程序生成向导，否则构造整个框架都不知道需要花费多少时间。

9.7.2 Write(KIrp I)函数

Write()函数的处理方法与Read()函数的方法十分类似，所不同的是获取缓冲区地址和获取写数据长度的函数不一样，另外创建URB时要指定为输出。

使用入口参数KIrp类I的成员函数I.BufferedWriteSource()来获取缓冲区地址，使用函数I.WriteSize()来获取传输的字节数。

这里就不再对程序的细节分析了，仅给出该函数的实现代码(删除了一些调试信息和向导的注释)，读者可以对照注释理解。

```
NTSTATUS Computer00UsbDevice::Write(KIrp I)
{
 NTSTATUS status = STATUS_SUCCESS;
 //原来向导生成在后面，从后面移动上来
 // Get a pointer to the caller's buffer.
 PUCHAR pBuffer = (PUCHAR) I.BufferedWriteSource();
 ULONG writeSize = I.WriteSize();
 ULONG bytesSent = 0;
/* if (FALSE) */
 if(pBuffer == NULL)                    //如果要发送的数据缓冲区为无效地址，则返回参数无效
 {
  status = STATUS_INVALID_PARAMETER;    //状态为无效参数
  I.Information() = 0;                  //传输字节为0
  I.PnpComplete(status);                //完成该 IRP
  return status;
 }
 if (I.WriteSize() == 0)                //如果读长度为0，那么总是成功的
 {
```

```
 I.Information() = 0;
 I.PnpComplete(this, status);
 return status;
}
/**************以下为圈圈新增代码************/
//创建一个中断传输的URB,用来往端点1发送数据
PURB pUrb = Ep1Out.BuildInterruptTransfer(
            pBuffer,                    //发送数据的缓冲区
            writeSize,                  //发送数据的数据字节数
            FALSE,                      //FALSE表示设备传输的字节少数不可以指定的字节数
            NULL,                       //连接下一个传输的URB,这里没有,置为NULL
            NULL,                       //指向一个已经存在的URB,置为NULL,分配一个新的URB
            FALSE);                     //FALSE表示发送数据

if(pUrb == NULL)                        //如果分配失败
{
 status = STATUS_INSUFFICIENT_RESOURCES;  //设置状态为资源不足
}
else
{
 //提交URB,并无限等待
 status = Ep1Out.SubmitUrb(pUrb,NULL,NULL,0);
 //获取实际发送的数据字节数
 bytesSent = pUrb->UrbBulkOrInterruptTransfer.TransferBufferLength;
 delete pUrb;  //删除刚刚分配的URB
}
/**************新增代码完毕*****************/
 I.Information() = bytesSent;
 I.PnpComplete(this, status);
 return status;
}
```

9.7.3 EP1_READ_Handler(KIrp I)函数

当应用程序使用指定的读端点1的I/O控制代码通过DeviceIoControl的方式访问本驱动时,该函数将会被调用。该函数实现的功能跟Read()函数是一样的,所不同的是DeviceIoControl访问方式与获取缓冲区和数据长度的方式不一样。

需要注意的是,在DeviceIoControl方式时,数据的方向是在驱动程序和应用程序之间的,

并且以驱动程序为主导。当应用程序传递数据给驱动程序时，在驱动程序看来这叫做数据输入，相应的缓冲就叫做输入缓冲区。这时驱动程序应该将输入数据发送到物理设备，也就是说，要发送给设备的数据其实是放在输入缓冲(InBuffer)中的。而驱动程序将数据返回给应用程序的，叫做数据输出，相应的缓冲就叫做输出缓冲区。也就是说，驱动程序从物理设备读回的数据，应该放到输出缓冲区(OutBuffer)中。

不过，由于本实例使用的是 Buffered 缓冲方式，实际的输出缓冲和输出缓冲是同一个，可以使用函数 I. IoctlBuffer()获取。该缓冲区是在调用 DeviceIoControl()函数时由操作系统分配的，大小为调用 DeviceIoControl()函数时所指定的输入和输出字节数的最大值。在 DeviceIoControl()函数调用时，操作系统会负责将输入缓冲区中的数据复制到该缓冲区中，函数返回时，操作系统会负责将该缓冲区中的数据复制到输出缓冲区中。驱动程序应该先从该缓冲区中读出输入数据(如果需要的话)，然后再将需要输出数据(通常是从物理设备读回)添加到该缓冲区中。

但是输入和输出字节数是不一样的，分别用函数 I. IoctlInputBufferSize()和函数 I. IoctlOutputBufferSize()来获取。

获取到缓冲区地址和数据传输长度后，这里的处理跟 Read()函数就基本上没有什么区别了。注意在该函数以及下面的几个 Handler()函数中不用完成 IRP，因为这些函数是由 DeviceControl()函数调用的，当返回到函数 DeviceControl(KIrp I)中时会统一完成 IRP。实际的 EP1_READ_Handler()函数代码如下：

```
NTSTATUS Computer00UsbDevice::EP1_READ_Handler(KIrp I)
{
 NTSTATUS status = STATUS_SUCCESS;
 ULONG inputSize = I.IoctlInputBufferSize();
 ULONG outputSize = I.IoctlOutputBufferSize();
 // Buffered ioctl - using the same buffer so read the buffer before writing the buffer
 PVOID inputBuffer = I.IoctlBuffer();
 PVOID outputBuffer = I.IoctlBuffer();
//注意:当读数据时,应该将数据保存在应用程序提供的输出缓冲中,因为驱动程序将会将读到的数据
//输出给应用程序
 if(outputBuffer == NULL)                    //如果缓冲区为无效地址
 {
  status = STATUS_INVALID_PARAMETER;
  I.Information() = 0;
 }
 else                                        //如果参数有效
 {
```

```
/**************圈圈新增代码*************/
if(outputSize == 0)                          //如果读数据长度为0,则不用传输数据
{
 I.Information() = 0;
}
else                                         //数据长度不为0
{
 //创建一个中断传输的URB,用来从端点1读取数据
 PURB pUrb = Ep1In.BuildInterruptTransfer(
            outputBuffer,            //接收数据的缓冲区
            outputSize,              //读数据的数据字节数
            TRUE,                    //TRUE表示设备传输的字节数可以少于指定的字节数
            NULL,                    //连接下一个传输的URB,这里没有,置为NULL
            NULL,                    //指向一个已经存在的URB。置为NULL,分配一个新的URB
            TRUE);                   //TURE表示读数据

 if(pUrb == NULL)                    //如果分配失败
 {
  status = STATUS_INSUFFICIENT_RESOURCES;  //设置状态为资源不足
 }
 else
 {
  //提交URB,并设置超时时间为3 s
  status = Ep1In.SubmitUrb(pUrb,NULL,NULL,3000);
  //获取实际读到的数据字节数
  I.Information() = pUrb->UrbBulkOrInterruptTransfer.TransferBufferLength;
  delete pUrb;  //删除刚刚分配的URB
 }
}
/**************新增代码完毕*****************/
}
return status;
}
```

9.7.4 EP1_WRITE_Handler(KIrp I)函数

当应用程序使用指定的写端点1的I/O控制代码通过DeviceIoControl的方式访问本驱动时,该函数将会被调用。该函数与Write()函数以及EP1_READ_Handler(KIrp I)处理方法类似,不同的地方在于这里要使用输入数据缓冲区和输入数据长度。另外,创建URB时数

据方向是输出的。具体的实现方法请读者参考前面的EP1_READ_Handler()函数和Write()函数。

9.7.5 EP2_READ_Handler(KIrp I)函数

EP2_READ_Handler(KIrp I)函数的处理跟EP1_READ_Handler(KIrp I)几乎是一样的,不同的地方在于创建和提交URB时所使用的管道换成了Ep2In;另外,由于端点2是批量端点,创建URB时要使用创建批量传输URB的函数,该函数的格式如下(注意跟中断的区别):

```
PURB BuildBulkTransfer(PVOID Buffer,
                       ULONG Length,
                       BOOLEAN bIn,
                       PURB Link = NULL,
                       BOOLEAN bShortOk = FALSE,
                       PURB pUrb = NULL);
```


其中,Buffer为指向保存数据缓冲区的地址。Length为需要传输数据的字节数。bIn表示数据传输的方向,值为TURE时表示输入数据,为FALSE时为输出数据。Link表示链接下一个URB的指针,值为NULL表示没有。bShortOk表示实际传输的数据是否可以比请求的短,值为TRUE时表示实际传输的数据可以短于请求的数据量(即数据传输量未达到指定值时提交URB的函数也会返回),值为FALSE时表示实际传输的数据必须要达到指定的长度时,提交URB的函数才能返回。pUrb表示指向一个已经存在的URB,如果没有,指定为NULL,这时该函数将会为新创建的URB分配空间,记得在使用完毕后释放该URB。

9.7.6 EP2_WRITE_Handler(KIrp I)函数

EP2_WRITE_Handler(KIrp I)函数的修改参看EP1_WRITE_Handler(KIrp I)函数和EP2_READ_Handler(KIrp I)函数的修改。

9.8 驱动的安装及安装后的信息

将上面修改的用户自定义USB设备的固件程序烧入到学习板中,然后运行,这时弹出发现新硬件的窗口,并弹出新硬件向导对话框,选择"从列表或指定位置安装(高级)(S)",如图9.8.1所示,单击"下一步"按钮。

然后指定驱动安装用的inf文件所在的位置,单击"下一步"按钮,如图9.8.2所示。该inf文件可以在驱动工程目录的driver目录下找到,是由向导自动生成的。

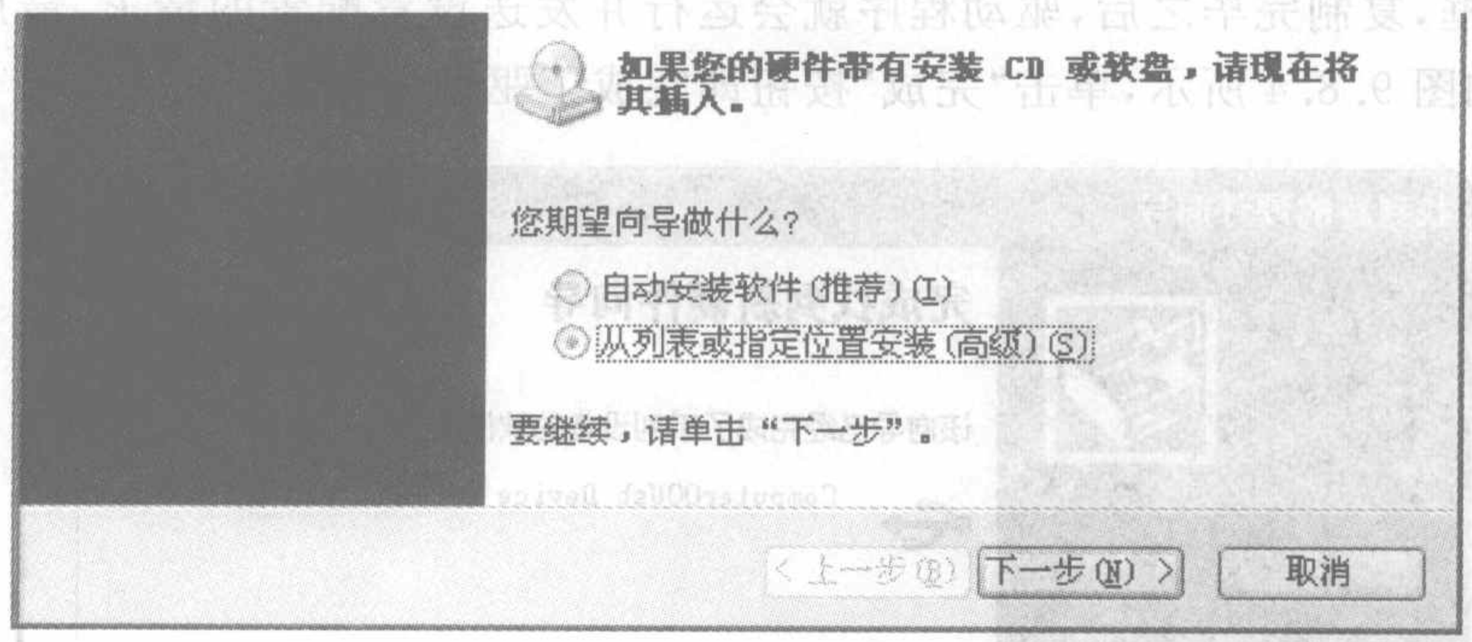


图 9.8.1 新硬件向导对话框

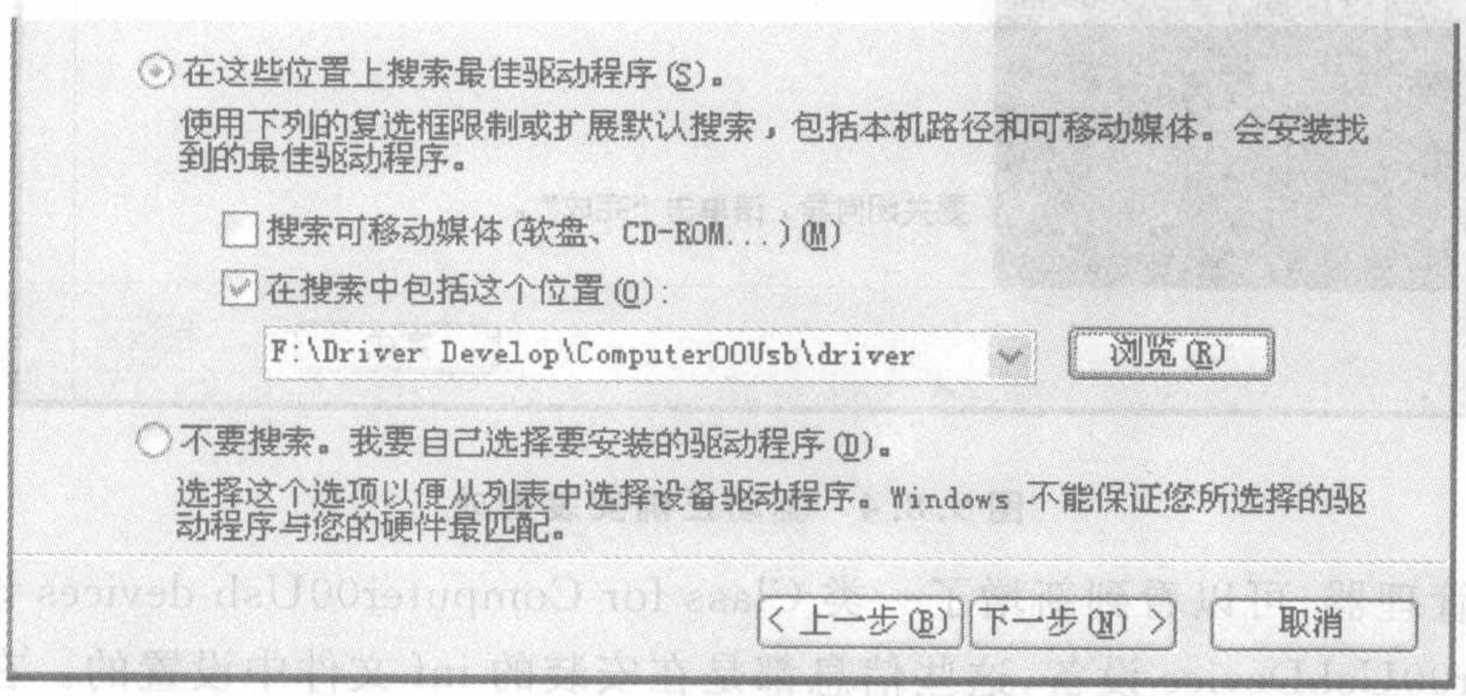


图 9.8.2 指定 inf 文件的位置

经过短暂的等待之后，提示需要 Computer00Usb.sys 文件，这是因为 inf 文件跟 sys 文件不在同一个文件夹下所至。如果不想看到这一步，把两个文件放在同一个目录下即可。这里指定 Computer00Usb.sys 的路径，然后单击“确定”按钮，如图 9.8.3 所示。接着就会出现复

图 9.8.3 指定 sys 文件所在位置

制文件的对话框，复制完毕之后，驱动程序就会运行并发送设置配置的请求，最后出现安装完成的对话框，如图9.8.4所示，单击“完成”按钮即完成了驱动的安装。

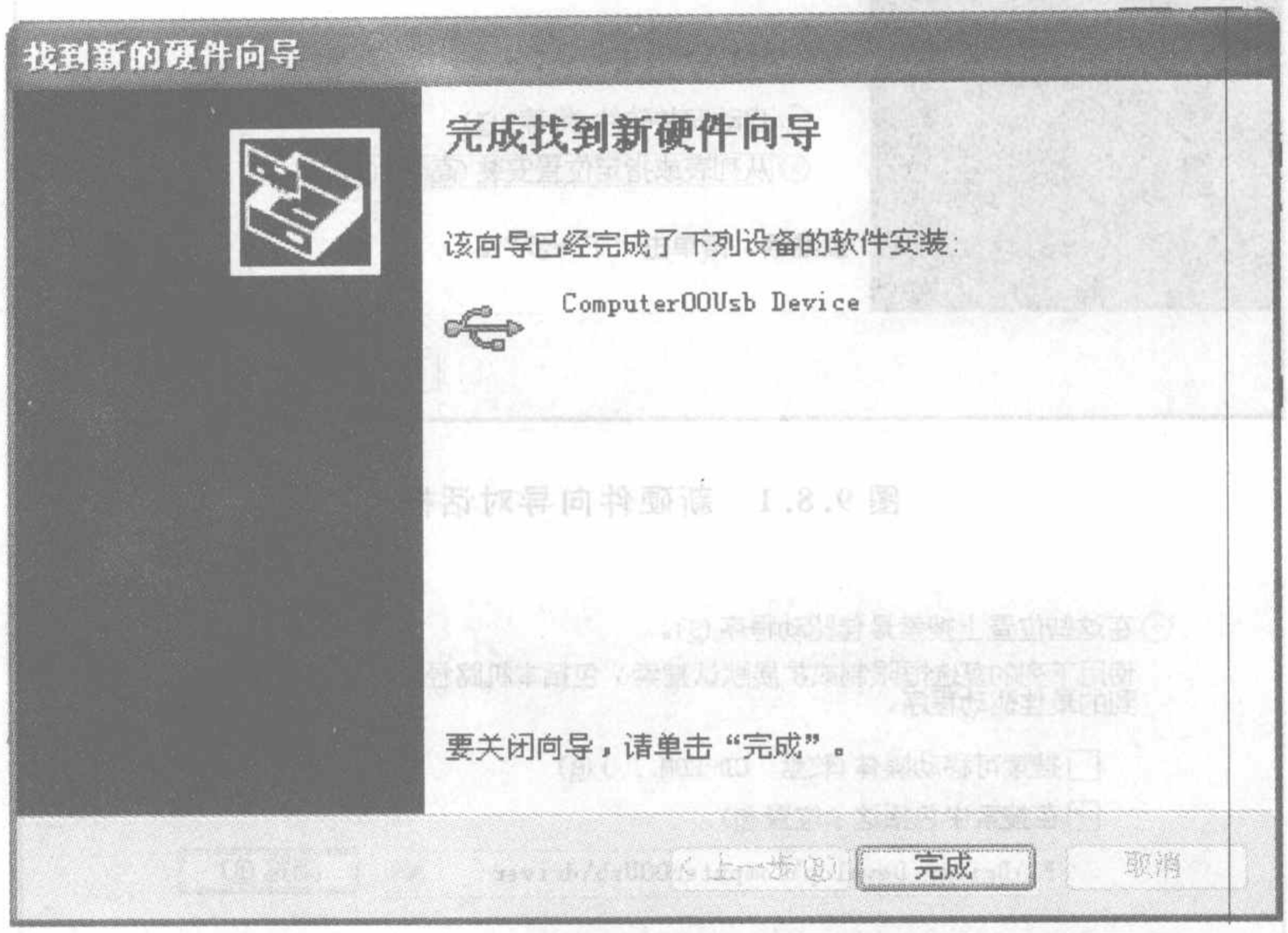


图9.8.4 驱动正确安装完成

打开设备管理器，可以看到新增了一类Class for Computer00Usb devices设备，在下面有一个Computer00UsbDevice设备，这些信息都是在安装的inf文件中设置的。在调试驱动时，需要重新安装驱动，这时右击这个设备，选择卸载，卸载完成后再单击图9.8.5中工具栏上那个计算机前有放大镜的图标(或者通过菜单“操作”，“扫描检测硬件改动”)即可发现新硬件并弹出新硬件向导的对话框。双击该设备可以查看更详细的信息，读者可以对照inf文件中设置的字段来看看不同的字段会出现在哪里。

再来看看注册表中增加了哪些有用的信息。按下Window键和R键打开运行对话框，输入regedit，确定，这将打开注册表编辑器。在注册表中，展开HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class子键，在下面可以找到一个{4731A720-D55A-4583-84F9-29144E14D709}的子键，它就是在安装驱动时指定的inf文件中的Class GUID。单击这个子键，可以在右边找到在inf文件中设置的类名、图标等。再展开这个子键，可以看到下面有个0000子键和Properties子键，如图9.8.6所示。在0000子键下，可以看到在向导中设置的子键MyUsbDevice，单击它，在右边可以看到一个值为Computer00′Usb Device的DeviceName项，它就是我们在驱动向导中增加的注册表项(当然在inf文件中也可以找到，因为它是记录在inf文件中的)。

图 9.8.5 设备管理器中的新设备

图 9.8.6 注册表下的 Class 子键

单击 0000 子键，可以看到右边有很多表项，如图 9.8.7 所示。其中 DriverDesc 就是设备在设备管理器中显示的名称，InfPath 是安装后备份在 Windows/inf 目录(属性为隐藏)下的 inf 文件。根据这个特性，通过查看注册表可以找到安装某设备的 inf 文件。MatchingDeviceId 就是要匹配的硬件 ID，这里指定的是 VID 为 0x8888，PID 为 0x000B 的 USB 设备。

名称	类型	数据
(默认)	REG_SZ	(数值未设置)
DriverDate	REG_SZ	8-22-2008
DriverDateData	REG_BINARY	00 40 20 05 ea 03 c9 01
DriverDesc	REG_SZ	ComputerOOUsb Device
DriverVersion	REG_SZ	1.0.0.0
InfPath	REG_SZ	oem7.inf
InfSection	REG_SZ	ComputerOOUsb_DDI
InfSectionExt	REG_SZ	.NT
MatchingDeviceId	REG_SZ	usb\vid_8888&pid_000b
ProviderName	REG_SZ	Http://computerOO.21ic.org

图 9.8.7 0000 子键下的表项

再来看看 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceClasses 子键，将其展开，可以找到一个{1cd961b7-470e-4586-954d-e6b8411b55c7}的子键，它就是我们在驱动程序 interface.h 中定义的接口类 GUID，如图 9.8.8 所示。点击图中那个单独的“#”号，在右边可以看到一个名为 SymbolicLink 的项，它的值其实就是在打开驱动时用的路径，值为“\\?\USB#Vid_8888&Pid_000b#2008-08-22#{1cd961b7-470e-4586-954d-e6b8411b55c7}”。从这里可以推测出该设备的 VID 为 0x8888，PID 为 0x000B，设备序列号为 2008-08-22。如果你知道一个设备的驱动安装了，但是不知道它的接口类 GUID，可以通过 VID、PID 等到注册表的 DeviceClasses 子键下中反查其 GUID。另外还有一个比较重要的子键，就是 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB\Vid_8888&Pid_000b，里面保存了设备的驱动程序服务名、硬件 ID、安装类 GUID 等重要信息。

图 9.8.8 注册表下的 DeviceClasses 子键

9.9 应用程序对驱动的访问

驱动是做好了，并且也装上了，那么应用程序如何去访问它呢？可以参考第 5 章中所介绍的那几个 API 函数来查找和打开设备。在打开设备之前首先要获取该设备驱动的接口类 GUID，这里可以直接使用驱动程序向导生成的那个 interface. h，将它复制到我们的应用程序目录下。在该头文件中是用 DEFINE_GUID 来定义 GUID 的，不过圈圈发现 DEFINE_GUID 这个宏不大好处理，干脆直接改成了自己定义（参看源代码），其实那个宏展开也是这个样子的。另外，在 interface. h 中定义那些 IoControl 的代码时，缺少对头文件 WINIOCTL. H 的引用，自己添加之。

该应用程序将直接在第 5 章的 HID 测试应用程序上修改，把原来的 HID GUID 换成我们设备的 GUID。VID、PID 以及版本号的比较就不再需要了，因为只有 ID 匹配时才能装上该驱动。另外，由于该驱动程序仅能够被打开一次，所以不再像原来的 HID 测试程序那样，读和写分别打开一个句柄，而是以读/写方式只打开一次。对于端点 1 的数据处理，几乎是一样的，不同的在于这里的数据没有第一字节的报告 ID。只要将多余的报告 ID 去掉，就可以实现原来的控制 LED 和显示开关状态的功能了。

为了演示端点 2 的读/写操作，分别新增了端点 2 读、写线程。端点 2 的处理跟端点 1 的基本上是一样的，所不同的是操作端点 2 必须使用 DeviceIoControl()函数，而端点 1 使用的是 ReadFile()和 WriteFile()函数（当然也可以通过 DeviceIoControl()函数，这里为了不重复，端点 1 操作选择了 ReadFile()和 WriteFile()函数。DeviceIoControl()函数原型如下：

```
BOOL DeviceIoControl(HANDLE hDevice,
                     DWORD dwIoControlCode,
                     LPVOID lpInBuffer,
                     DWORD nInBufferSize,
                     LPVOID lpOutBuffer,
```

```
                DWORD nOutBufferSize,
                LPDWORD lpBytesReturned,
                LPOVERLAPPED lpOverlapped);
```

其中，hDevice就是所打开设备的句柄。dwIoControlCode就是在驱动向导中指定的IOCTL代码(定义在interface.h中)。lpInBuffer为应用程序提交给驱动程序的输入数据的缓冲区，nInBufferSize为lpInBuffer的大小(也就是数据的个数)。lpOutBuffer是应用程序用来接收驱动程序返回数据的缓冲区，nOutBufferSize为lpOutBuffer的大小。lpBytesReturend为接收实际传输字节数的变量的地址，lpOverlapped为一个指向OVERLAPPED结构体的指针，更详细的描述请参看第5章中对ReadFile()和WriteFile()的说明。当调用成功时，函数将返回非0值。

为了测试端点2的操作，增加了8个文本输入框和一个发送按钮，在文本框中输入需要发送的数据(十六进制)，然后点击发送即可将数据发送到端点2。当端点2接收到数据时，将显示在下面的信息框中。在设备中，发送到端点2的数据将通过串口输出，而串口接收到的数据将通过端点2返回。将学习板的串口连接到电脑的串口上并打开串口调试助手(9 600波特率)可以看到发送到端点2的数据，往串口发送的数据将通过端点2返回并显示在测试程序的信息框中。

调用DeviceIoControl()函数往端点2发送8字节数据的格式如下：

```
Result = DeviceIoControl(
            hMyDevHandle,               //我们的设备
            EP2_WRITE,                  //驱动程序定义的IoControl的功能代码
            Ep2WriteBuffer,             //输入缓冲
            8,                          //输入字节数
            NULL,                       //输出缓冲,无
            0,                          //输出字节数
            NULL,                       //使用Overlapped时不在这里保存传输字节数
            &Ep2WriteOverlapped);       //指定的Overlapped结构
```

由于端点描述符描述的端点2最大包长度为64字节，所以在读取端点2数据时，提供的缓冲区至少要为64字节，否则设备返回的数据超过64字节，会导致驱动出错。当然缓冲区更大是没有问题的(发送数据时也是，一次发送的数据可以多于64字节，下层的USB总线驱动程序会负责拆包)。由于在驱动程序中指定了读数据时可以少于指定的字节数，所以实际读到的数据可能会比指定的字节数要少，通过GetOverlappedResult()函数可以获取实际传输的数据量。调用DeviceIoControl()函数从端点2读取64字节数据的格式如下：

```
DeviceIoControl(hMyDevHandle,                //我们的设备
                EP2_READ,                    //驱动程序定义的 IoControl 的功能代码
                NULL,                        //输入缓冲,无
                0,                           //输入字节数
                Ep2ReadBuffer,               //输出缓冲
                64,                          //输出字节数
                NULL,                        //使用 Overlapped 时不在这里保存传输字节数
                &Ep2ReadOverlapped);         //指定的 Overlapped 结构
```

整个测试应用程序的源代码可以参看附带光盘中的 MyUsbDeviceTestApp 工程,限于篇幅,这里就不再详细介绍了。

9.10 测试软件的使用

将 USB 学习板连接上,然后运行测试程序。测试程序的界面如图 9.10.1 所示。单击"打开设备"按钮,下面的信息框将显示打开设备的状态。如果打开设备成功,那么"关闭设备"、"清除计数器"、"发送数据"、控制 LED 等按钮将变成可用状态。单击"关闭按钮"可以关闭已打开的设备句柄,如果单击"关闭按钮"之后很快又单击"打开设备"按钮,可能会提示"打开设

图 9.10.1 测试程序界面

备失败:设备已经被打开”的错误。这是因为发送了读数据的请求,但是板上没有数据返回,导致驱动还未完成前面的操作而依然处于打开状态。还记得生成驱动向导时,我们指定了该设备只允许打开一个句柄的属性吗?所以接下来的打开驱动操作就返回失败了。等延迟3 s之后,读数据的请求超时返回,驱动才被关闭,这时才可以重新打开驱动。

- 单击“关于”按钮将显示关于对话框,如图9.10.2所示;
- 单击LED1~LED8按钮可以控制学习板上的LED状态;
- 学习板上按键的状态可以显示在KEY1~KEY8上;
- 发送数据区的文本输入框可以设置需要发送的数据,单击“发送数据”按钮将把这些数据发送到端点2;
- “总发送”显示的是单击“发送数据”按钮所发送到端点2数据的总数,“总接收”显示的是从端点2接收到数据的总数;
- 信息框显示的是操作的基本信息、收发的数据以及操作的时间,单击“清除信息”按钮可以清除信息;
- “计数器值”显示的是学习板中的一个计数器值,可以改成电压或者温度显示等(当然还得在板上增加相应的传感器),单击“清除计数器”按钮可以将计数器清0。

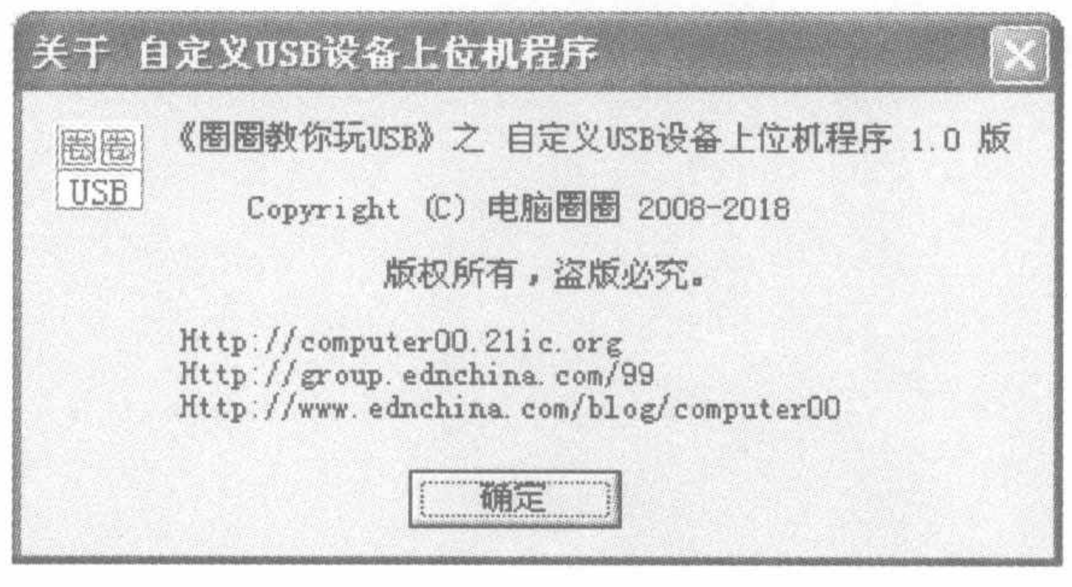


图9.10.2 关于对话框

在程序中有对设备插入和拔下的检测(OnDeviceChange()函数),当设备连接或拔下时,会在信息窗口显示相应的信息。如果设备是在打开的状态下被拔出,那么会自动关闭设备。

9.11 本章小结

本章详细地介绍了一个用户自定义USB设备的驱动程序开发过程,让刚接触USB WDM驱动开发的读者有个大概地了解。对访问驱动程序的方式也作了简单介绍,并给出了测试该驱动的实例工程。读者按照书中的步骤一步步操作,应该能够创建出一个属于自己的驱动程序。当然这其中还有很多细节问题,另外还可能会遇到一些像编译通不过之类的问题,这都需

要读者自己去学习、去解决。如果在编译时通不过，将编译的错误信息放到网上去搜索，往往能够找到答案；或者对某个函数不了解时，去网上搜索也往往能找到答案；或者去查找 MSDN 的帮助文档。总之有很多问题是需要自己解决的，不能总依赖书本和他人，学会解决问题的方法才是最主要的。读者可以在本实验的基础上修改或者增加一些更有趣的功能，例如采集温度、采集电压，或者拿它来做一个 USB 接口的编程器等等。有了用户自定义的 USB 设备和驱动，从某种程度上来说，用户就可以不再受到各种设备类的限制而为所欲为了。

第10章 USB过滤驱动开发

有时我们不想修改设备或者设备程序无法修改，但是又想改变设备的功能或者数据，这时就可以考虑使用过滤驱动了。另外，过滤驱动还可以用来截获设备传输的数据，著名的 Bus Hound 之所以能够捕捉设备的数据，使用的就是过滤驱动的方法。本章将介绍一个简单的 USB 过滤驱动的开发，它实现的功能是将第 4 章中的 USB 键盘“变”成第 5 章中的用户自定义的 USB HID 设备。

10.1 过滤驱动简介

过滤驱动(Filter Driver)有上层过滤和下层过滤两种，处于功能驱动之上的叫做上层过滤驱动，处于功能驱动程序之下(但仍处于总线驱动之上)的叫做下层过滤驱动。上层过滤驱动夹在应用程序(或者更高层的驱动程序)与驱动程序之间，而下层过滤驱动则夹在驱动程序与其下层驱动程序之间。数据在传递时必然会经过过滤驱动，因而过滤驱动可以将数据捕捉、修改或者拦截。

在本章所介绍的 USB 下层过滤驱动的开发过程中，过滤驱动的作用是将 USB 键盘改成用户自定义的 USB HID 设备。我们知道，在 Windows 下 USB 键盘是系统独占设备，所以使用第 5 章设计的访问 HID 设备的应用程序将无法以读方式打开这个 HID 设备。如果在这个 HID 设备上安装一个过滤驱动，将它返回的报告描述符中的应用集合用途改成用户自定义(即 0xFF)，那么系统将认不出这个集合，从而就不会加载键盘的驱动，而是加载一个 HID 兼容设备的驱动，这样使用我们的软件就可以访问它了。当然，还需要修改报告描述符中输出报告的定义，因为我们软件的输出报告是 8 字节的。另外，要能够正确显示开关状态，返回数据也得修改。

Driver Studio 提供了创建过滤驱动的向导，只要按照向导一步步生成，就可以构造出一个过滤驱动的基本框架。

10.2　使用 DS 创建一个下层过滤驱动

创建过滤驱动与创建功能驱动的前几个步骤是一样的，以下为具体步骤：

(1)设置工程路径以及工程名。如图 10.2.1 所示，将工程名设置为 MyUsbLowerFilter，然后单击 Next 按钮。

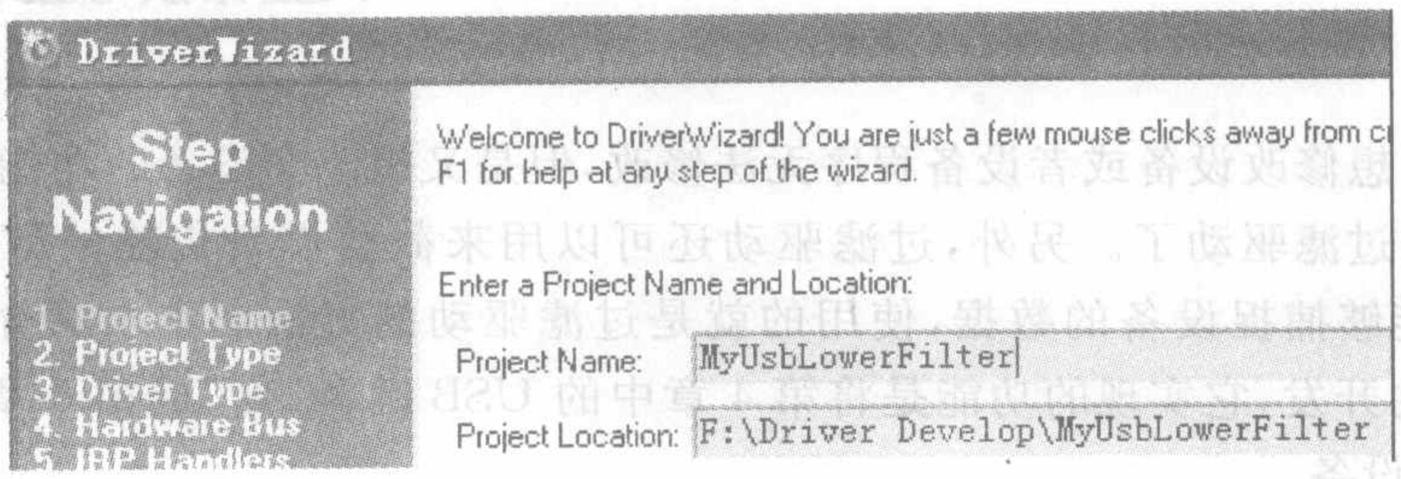


图 10.2.1　设置工程路径及工程名

(2)选择工程的类型。在弹出的设置工程类型对话框中选择 WDM 驱动(因为 USB 过滤驱动是属于 WDM 驱动)，如图 10.2.2 所示。驱动框架选择为 C++框架，之后单击 Next 按钮。

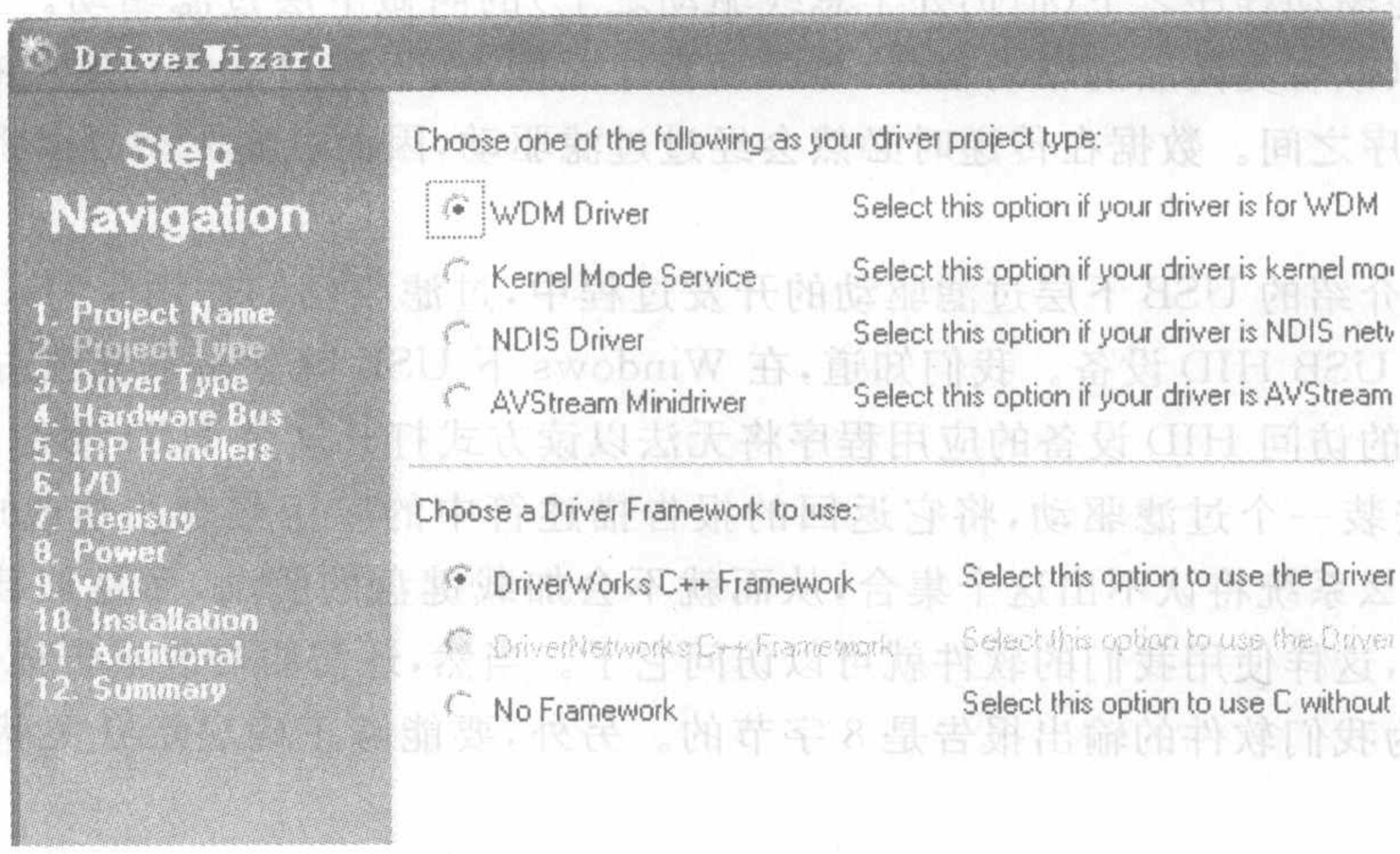


图 10.2.2　选择工程的类型

(3)选择驱动的类型。在选择驱动类型对话框中选择驱动类型为 WDM 过滤驱动(WDM Filter Driver)，然后单击 Next 按钮，如图 10.2.3 所示。

(4)选择过滤器的类型。如图 10.2.4 所示，在选择过滤器的对话框中，选择过滤的对象为

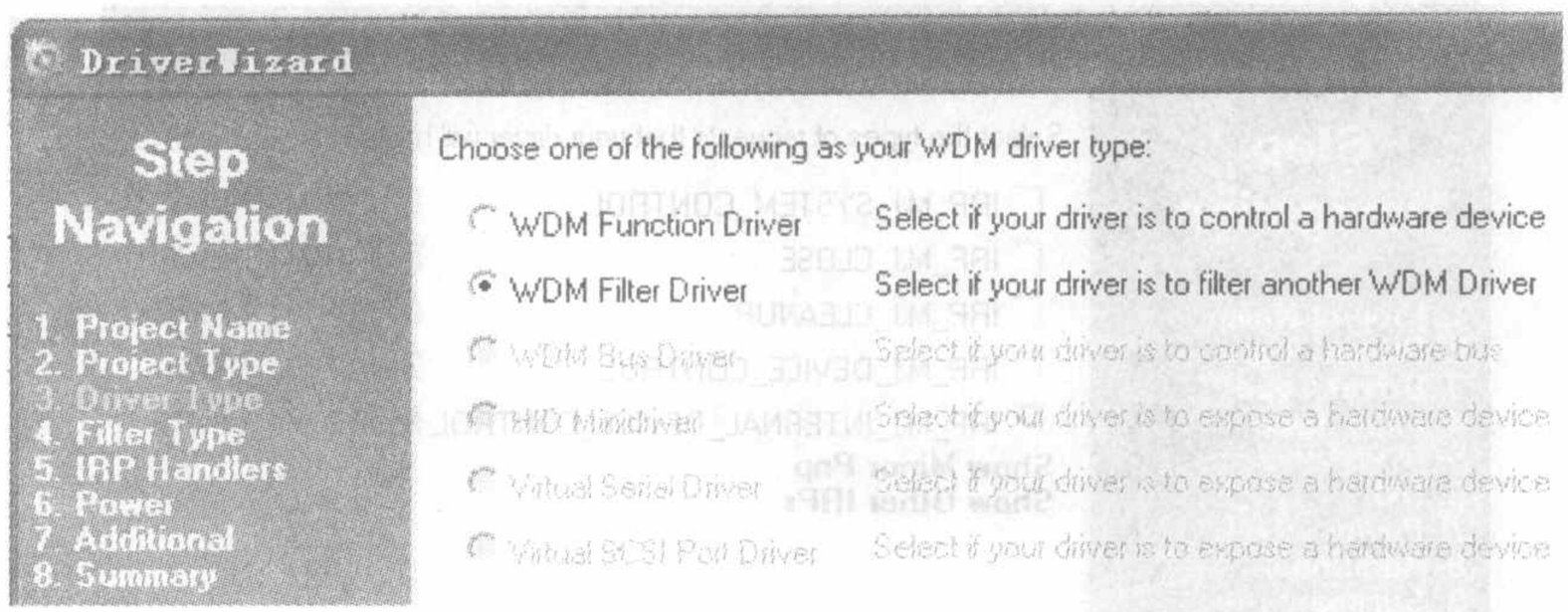


图 10.2.3 选择驱动的类型

设备(device),设备描述填入"USB人体学输入设备"。过滤器类型选择为下层过滤驱动(lower-level Filter),并将下面的Create Installation DLL勾选上,这将帮我们产生一个安装驱动用的DLL工程,可以在安装过滤驱动时由应用程序调用,然后单击Next按钮。

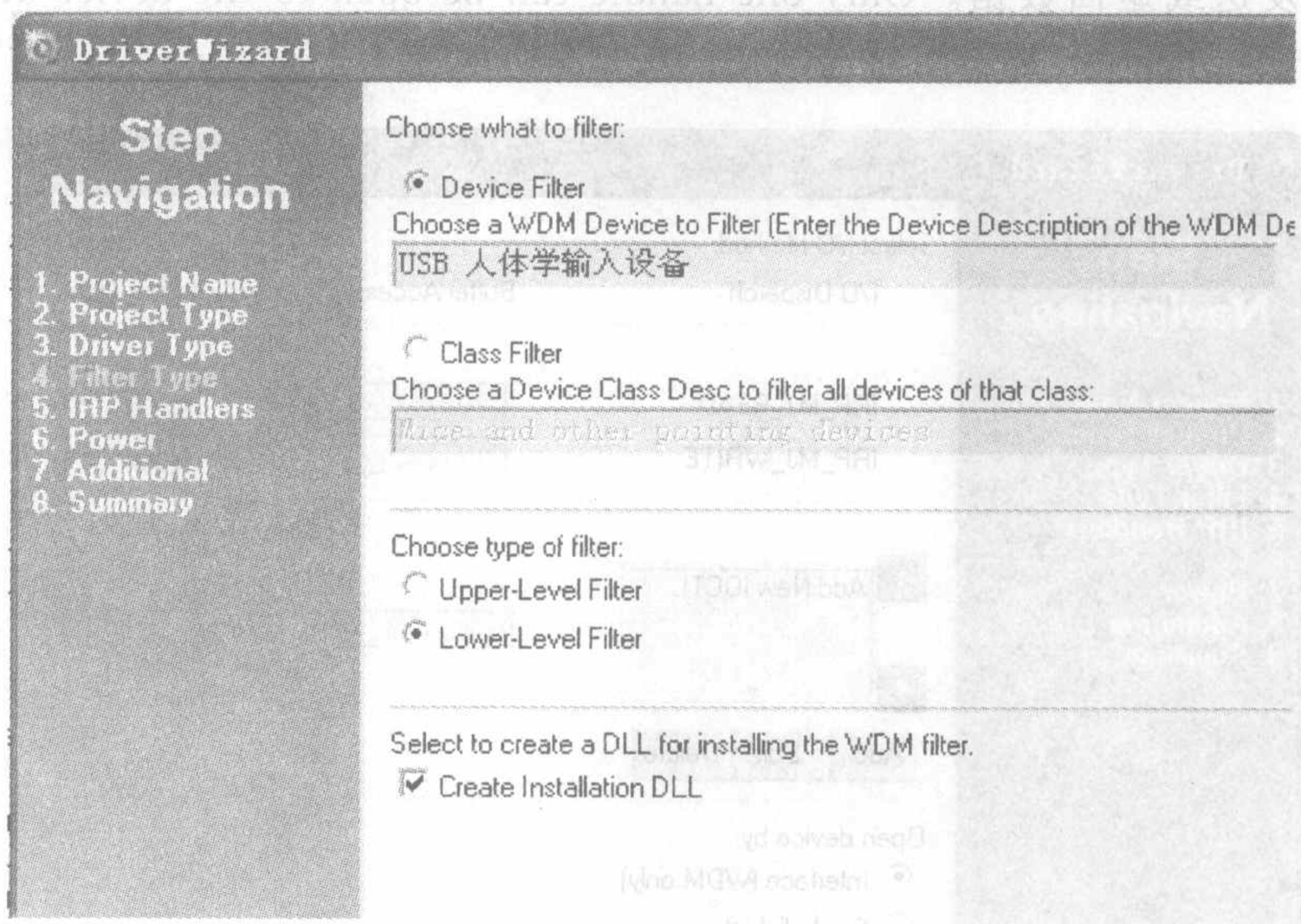


图 10.2.4 选择过滤器的类型

(5)选择要过滤的IRP。这里将IRP_MJ_INTERNAL_DEVICE_CONTROL、IRP_MJ_READ、IRP_MJ_WRITE三个勾选上,如图10.2.5所示。获取报告描述符以及读、写数据就是通过IRP_MJ_INTERNAL_DEVICE_CONTROL请求实现的,在这个请求完成时,修改返回的数据即可修改报告描述符和读取到的报告数据。

(6)配置I/O管理。如果你有应用程序需要访问该过滤驱动(例如捕捉经过该过滤驱动

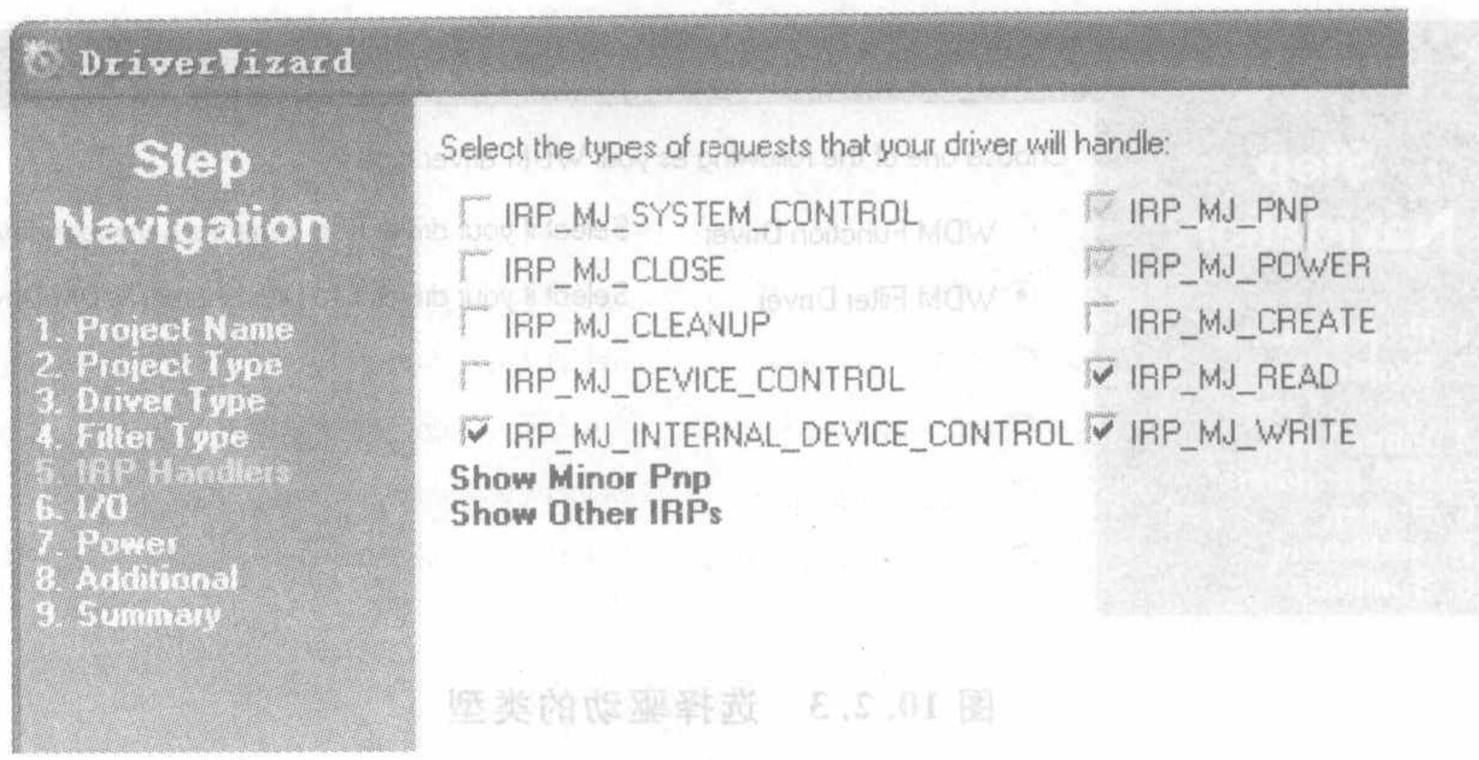


图 10.2.5 选择要过滤的 IRP

的数据，或者直接控制设备等)，在这里可以添加新的 IoControl 代码。我们这里不需要添加，仅是改变一下发送或返回数据。Only one handle can be open to the device at any time 和 Generate Test Application 都不用勾选，如图 10.2.6 所示。

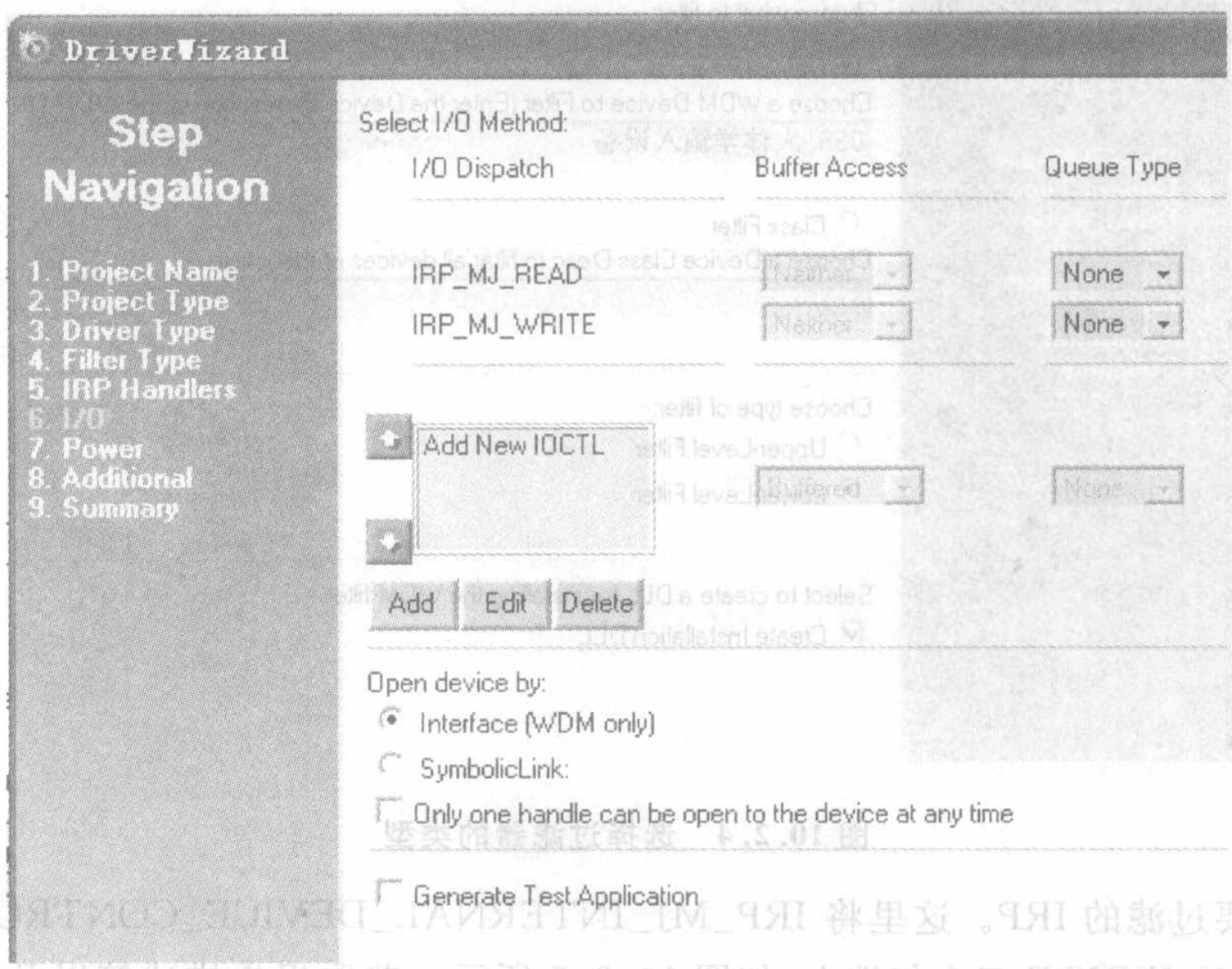


图 10.2.6 配置 I/O 管理

(7)配置电源管理。电源管理使用向导的默认值即可，如图 10.2.7 所示。

(8)配置附加信息。附加信息使用向导的默认值即可，如图 10.2.8 所示。

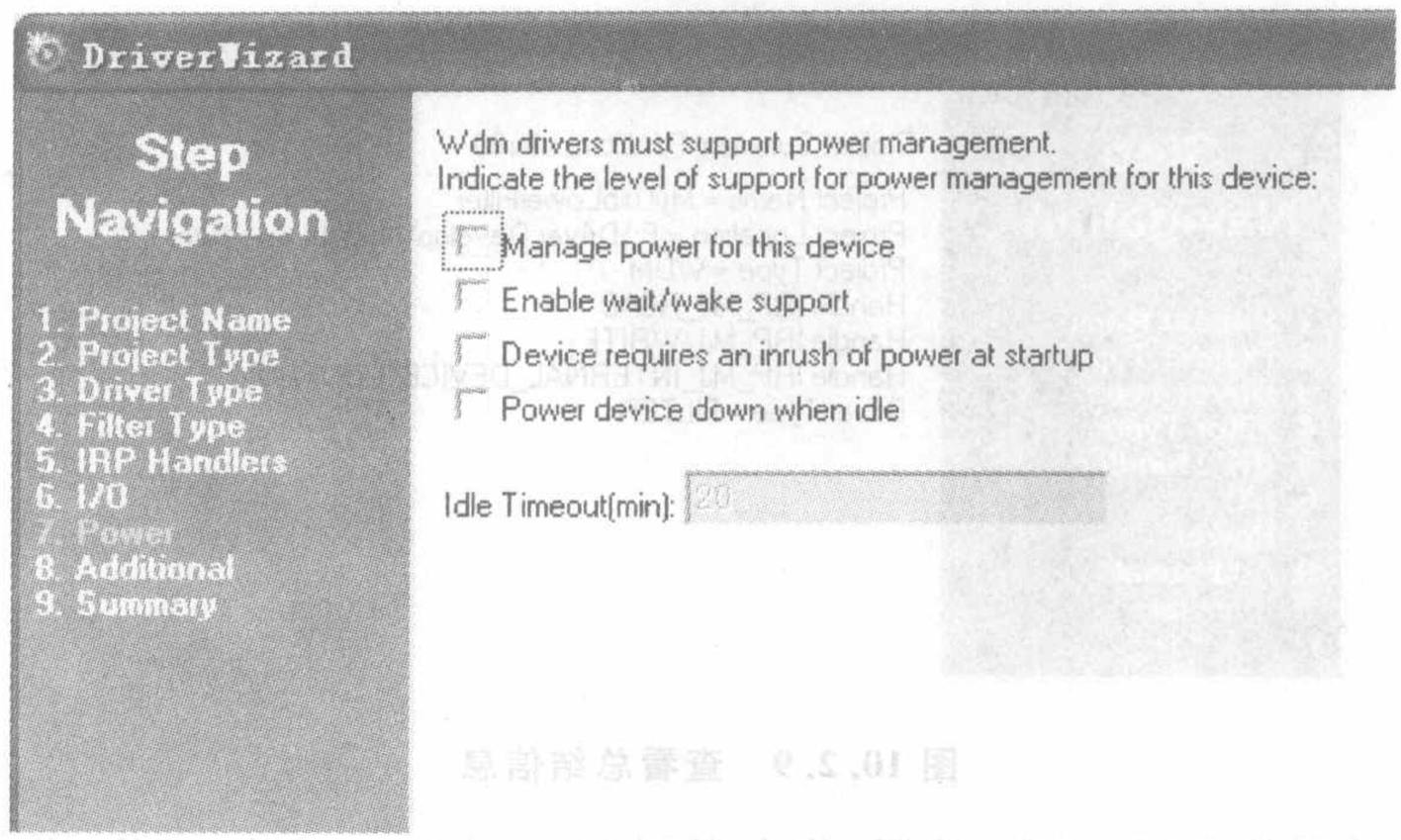


图 10.2.7 配置电源管理

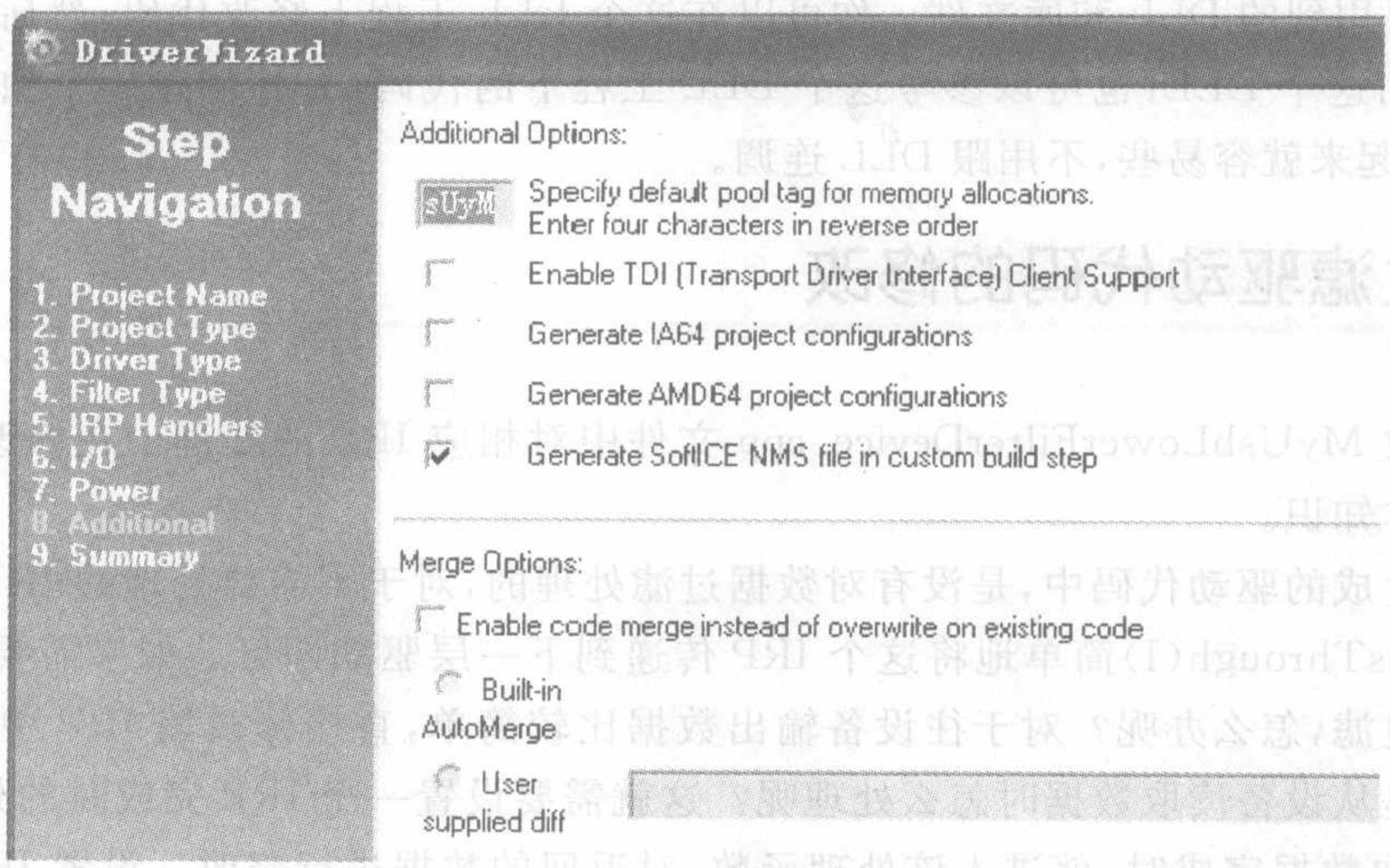


图 10.2.8 配置附加信息

(9)查看总结信息,在这里可以大致检查一下设置是否正确,如果发现有错误,还可以后退修改。确认无误后,就可以单击 Finish 按钮完成向导了,如图 10.2.9 所示。

(10)编译工程。选择驱动工程,编译,同样会出现找不到 ntstrsafe.lib 的错误,在工程的 Settings 里将 ntstrsafe.lib 删除即可(参看第 9 章)。编译成功之后,就会在相应的目录(调试和发布版)下生成一个 MyUsbLowerFilter.sys 文件,它就是我们最终需要的驱动文件(当然接下来还需要修改相关的代码)。

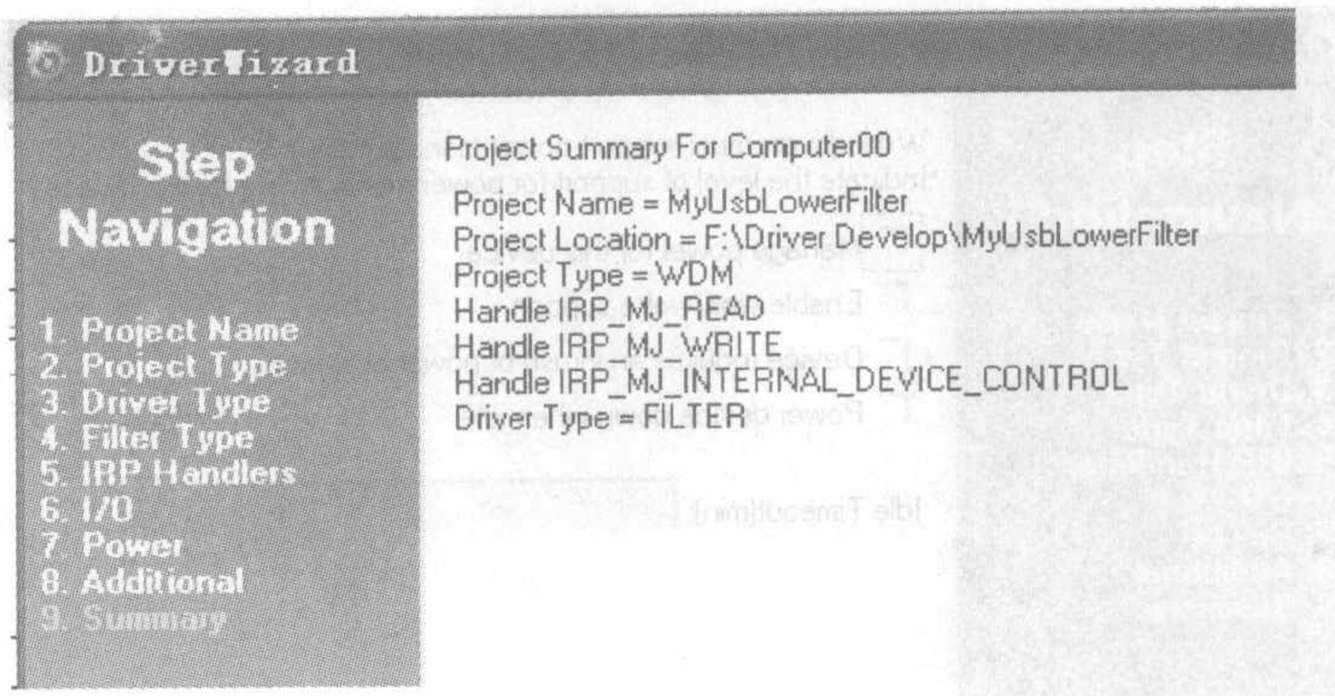


图 10.2.9 查看总结信息

选择安装驱动用的 DLL 工程，编译，将会得到 MyUsbLowerFilterDll. dll 和 MyUsbLowerFilterDll. lib 两个文件(位于 install 目录下的 objchk 或 objfre 目录下)。这两个文件就是开发安装程序要用到的 DLL 和库文件。你可以在这个 DLL 工程下修改代码，然后另外写个应用程序来调用这个 DLL；也可以参考这个 DLL 工程下的代码，另外创建一个独立的安装程序，这样调试起来就容易些，不用跟 DLL 连调。

10.3 过滤驱动代码的修改

主要修改 MyUsbLowerFilterDevice. cpp 文件中对相应 IRP 的处理。在修改代码之前，先说一些基本知识。

在向导生成的驱动代码中，是没有对数据过滤处理的，对于不需要处理的 IRP 请求，只要调用函数 PassThrough(I)简单地将这个 IRP 传递到下一层驱动即可。如果需要对某个 IRP 的数据进行过滤，怎么办呢？对于往设备输出数据比较简单，直接修改该 IRP 中缓冲区的数据即可。但是从设备读取数据时怎么处理呢？这就需要设置一个 IRP 完成时的处理函数了，当该 IRP 读取数据完成时，就进入该处理函数，对返回的数据进行修改。设置 IRP 完成处理函数的函数格式为

```
PassThrough(I, LinkTo(IrpCompletionRoutine), this)
```

其中，LinkTo 宏中的 IrpCompletionRoutine 就是要设置的完成函数，在这个函数中处理返回的数据即可。当然并不是对所有的 IRP 操作都需要设置完成函数，只有那些需要被过滤的请求才需要设置，可以从 IRP 中获取参数并判断是否为需要过滤的请求。

驱动程序调试比较麻烦，它不像普通程序那样可以设置断点、单步运行等。只能借助其他的一些工具来调试驱动，例如 DS 提供的 DriverMonitor 软件(所在位置：Compuware\ Driver-

Studio\Tools\Monitor\monitor.exe)。在驱动程序中使用KTrace类来发送调试信息,这些发送的调试信息将显示在DriverMonitor中。在向导生成的代码中,有一个全局变量T,它就是KTrace类的实例,我们可以使用T来显示调试信息。KTrace类有很多重载函数,可以直接显示不同的数据类型。注意只有调试版(选择为Win32 Checked)的驱动才会显示调试信息,发行版(选择为Win32 Fre)的驱动不会显示调试信息。

下面就来修改驱动的代码,打开MyUsbLowerFilterDevice.cpp文件,可以找到一个InternalDeviceControl(KIrp I)的函数。这个函数就是上层驱动与下层驱动之间打交道的函数。在这里先判断I/O控制代码是否为IOCTL_INTERNAL_USB_SUBMIT_URB(需要引用头文件kusb.h),如果是,则说明是提交一个URB。然后再从入口参数的KIrp类的I中获取到当前的URB,就可以知道具体是什么请求了。获取URB的方法是调用KIrp类的URB函数,入口参数为CURRENT表示当前的IRP。然后从URB结构体中就可以获取到该URB的功能(Function)代码。我们知道请求报告描述符是发送到设备的接口的,因而该URB的功能代码就是URB_FUNCTION_GET_DESCRIPTOR_FROM_INTERFACE(0x28),而批量或中断传输的URB功能代码为URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER(0x09)。如果功能代码为从接口获取描述符,那么就判断描述符类型是否为报告描述符(0x22),如果是,就设置完成函数。如果功能代码为批量或中断传输,那么先判断是输入还是输出,如果是输入请求,就设置完成函数;如果是输出数据,就将数据长度改成1字节,并判断是否需要清除计数器(还记得自定义HID设备中那个计数器吗?这里将在过滤驱动中增加一个MyCount的变量实现)。

下面是实际修改好的InternalDeviceControl的代码:

```
NTSTATUS MyUsbLowerFilterDevice::InternalDeviceControl(KIrp I)
{
 T<<"进入 InternalDeviceControl 函数\n";              //显示调试信息

//如果 Ioctl 代码为 IOCTL_INTERNAL_USB_SUBMIT_URB,那么就是提交 URB 的请求,
//在这里判断我们需要过滤的 IRP
if(I.IoctlCode() == IOCTL_INTERNAL_USB_SUBMIT_URB)
{
 PURB pUrb = I.Urb(CURRENT);                          //获取当前 IRP 的 URB
 if(pUrb! = NULL)                                     //如果 pUrb 有效
 {
  //T << *pUrb;          可以显示 URB 的详细信息(需要增加 usbd.lib 才能使用)
  T<<"URB 功能代码为:";                               //显示 URB 的功能代码
  T<<pUrb->UrbHeader.Function;                        //获取功能代码
  T<<"\n";                                            //换行
  switch(pUrb->UrbHeader.Function)                    //对功能代码散转
```

```
{
 //如果功能代码为从接口请求描述符(0x28)
 case URB_FUNCTION_GET_DESCRIPTOR_FROM_INTERFACE:
  T<<"从接口获取描述符的 URB\n";                    //显示调试信息
  //判断请求的描述符是否为报告描述符(0x22)
  if(pUrb->UrbControlDescriptorRequest.DescriptorType == 0x22)
  {
   T<<"描述符的类型为报告描述符(0x22)\n";   //显示调试信息
   T<<"设置完成函数\n";
   //设置完成函数为 IrpCompletionRoutine,并返回
   return PassThrough(I, LinkTo(IrpCompletionRoutine), this);
  }
 break;

 //如果功能代码为批量或中断传输(0x09)
 case URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER:
  T<<"批量或中断传输的 URB ";                       //显示调试信息
  //如果为输入请求(TransferFlags 的最低位为传输方向)
  if((pUrb->UrbBulkOrInterruptTransfer.TransferFlags)&0x01)
  {
   T<<"(输入传输)\n";                               //显示调试信息
   T<<"设置完成函数\n";
   //设置完成函数为 IrpCompletionRoutine,并返回
   return PassThrough(I, LinkTo(IrpCompletionRoutine), this);
  }
  else                                              //为输出请求
  {
   UCHAR * pBuf;                                    //保存缓冲区地址
   ULONG Len;                                       //保存长度

   T<<"(输出传输)\n";                               //显示调试信息
   //获取缓冲区地址
   pBuf = (UCHAR *)pUrb->UrbBulkOrInterruptTransfer.TransferBuffer;
   //获取传输的长度
   Len = pUrb->UrbBulkOrInterruptTransfer.TransferBufferLength;
   if(Len == 8)                                     //8 字节的报告
   {
    if(pBuf[1]! = 0)                                //当第二字节不为 0 时,要清除计数器
    {
     T<<"清除计数器\n";
```

```
        MyCount = 0;                                    //计数器清0
      }
    }
    //仅发送1字节数据
    pUrb->UrbBulkOrInterruptTransfer.TransferBufferLength = 1;
    T<<"修改输出报告完毕\n";
  }
  break;
  default:
  break;
}
}
}
//对于不需要过滤的IRP,将它直接传递给下层即可
NTSTATUS status = PassThrough(I);
return status;
}
```

那么完成函数 IrpCompletionRoutine()中该怎么处理呢？这里要先判断传输的类型(即URB的功能代码)。

如果是控制传输(当获取描述符完成时，URB的功能代码会变为控制传输，其代码值为0x08，宏定义为URB_FUNCTION_CONTROL_TRANSFER)，则判断描述符的类型；如果是报告描述符，则要对报告描述符修改。报告描述符要修改应用集合的用途以及输出报告的长度。原来的报告描述符中设置的应用集合为0x06(即用途为键盘)，现在要改为用户自定义设备，所以改成0xFF(改成0x00也行，只要系统不认识就好)。而输出报告的长度则通过修改填充的3位常量字段，因为原来的键盘程序使用了5位做LED，这里将3位的填充改成59位，那么总共就有64位了(即8字节)。集合用途位于报告描述符偏移量3处(即第4字节)，填充字段的长度位于偏移量61处(即第62字节)。

如果是批量或者中断传输，则判断是否为输入传输，如果是，就说明接收到了数据。然后读取数据的长度，看是否为8字节，如果是，就要修改这8字节的报告值。在原来的键盘程序中，按键情况是分布在这8字节输入报告中的，这里要根据这8字节数据将按键状态放入到第一字节中。具体请参看键盘程序的代码以及下面函数的代码。而第2至第4字节保存的为计数器值，这里将MyCount的值装入到这4字节中。剩余的3字节没有用到，将它们清零。

修改好的 IrpCompletionRoutine()函数如下：

```
NTSTATUS MyUsbLowerFilterDevice::IrpCompletionRoutine(KIrp I)
{
```

```
NTSTATUS status = I.Status();
//T.Trace(TraceInfo, __FUNCTION__" IRP %p, STATUS %x\n", I, status);
// TODO: Add driver specific code to process the IRP that the lower
// device has completed.
UCHAR *pBuf;
ULONG Len;
T<<"进入完成函数\n";
PURB pUrb = I.Urb(CURRENT);                    //获取当前 IRP 中的 URB
if(pUrb! = NULL)                               //如果 pUrb 有效
{
 switch(pUrb->UrbHeader.Function)              //对功能代码散转
 {
  //如果是控制传输(注意:当获取描述符的 IRP 完成时,功能代码会变成控制传输)
  case URB_FUNCTION_CONTROL_TRANSFER:          //值为 0x08
   //判断描述符类型是否为报告描述符(0x22)
   if(pUrb->UrbControlDescriptorRequest.DescriptorType == 0x22)
   {
    T<<"获取报告描述符完成\n";
    //获取缓冲区地址
    pBuf = (UCHAR *)pUrb->UrbControlDescriptorRequest.TransferBuffer;
    //获取传输的长度
    Len = pUrb->UrbControlDescriptorRequest.TransferBufferLength;
    if(Len == 65)  //说明报告描述符长度正确,因为我们的键盘报告描述符为 65 字节
    {
     //原来的报告描述符第四字节为 0x06,表示集合用于键盘,这里改成 0xFF,表示自定义
     T<<"报告描述符第 4 字节被修改为 0xFF\n";
     pBuf[3] = 0xFF;
     //原来的报告描述符第 62 字节为 0x03,表示附加字段的大小为 3 位,
     //上面已经定义了 5 位,共 8 位,即 1 字节。现在需要再增加 7 字节,
     //即增加 7×8 = 56 位,加上原来的 3 位后为 59 位,因此修改 pBuf[61]为 59
     pBuf[61] = 59;                            //这样整个输出报告就是 8 字节的了
     T<<"报告描述符第 62 字节被修改为 59\n";
    }
   }
  }
 break;
```

```
//如果是批量或中断传输
case URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER:  //值为 0x09
 //如果为输入请求(TransferFlags 的最低位为传输方向)
 if((pUrb->UrbBulkOrInterruptTransfer.TransferFlags)&0x01)
 {
  T<<"批量或中断读数据完成\n";
  //获取缓冲区地址
  pBuf = (UCHAR * )pUrb->UrbBulkOrInterruptTransfer.TransferBuffer;
  //获取传输的长度
  Len = pUrb->UrbBulkOrInterruptTransfer.TransferBufferLength;
  if(Len == 8)  //如果返回的是 8 字节数据
  {
   unsigned int i;
   //用来保存按键的状态
   UCHAR KeyStatus;
   //在 USB 键盘程序中,KEY1、KEY2、KEY3 刚好对应着报告第一字节的低 3 位,
   //所以这里可以直接将 pBuf[0]的低 3 位赋值给 KeyStatus
   KeyStatus = (pBuf[0])&(0x07);
   //第二字节(即 pBuf[1])为保留字节,剩下的 6 个字节(pBuf[2]~pBuf[7])为 6 个键码,当某个
   //键按下时会出现对应的键码,KEY4 对应的键码为 0x59,KEY5 对应的键码为 0x5A,
   //KEY6 对应的键码为 0x5B,KEY7 对应的键码为 0x39,KEY8 对应的键码为 0x53
   //在这 6 个字节中查找是否有对应的键码出现就知道该键是否被按下了
   for(i = 2;i<8;i ++ )
   {
    if(pBuf[i] == 0x59)KeyStatus| = 0x08;          //KEY4
    if(pBuf[i] == 0x5A)KeyStatus| = 0x10;          //KEY5
    if(pBuf[i] == 0x5B)KeyStatus| = 0x20;          //KEY6
    if(pBuf[i] == 0x39)KeyStatus| = 0x40;          //KEY7
    if(pBuf[i] == 0x53)KeyStatus| = 0x80;          //KEY8
   }
   pBuf[0] = KeyStatus;                            //设置按键情况
   MyCount ++ ;                                    //计数器加 1
   //pBuf[1]~pBuf[4]为计数器值
   pBuf[1] = (UCHAR)(MyCount&0xFF);                //最低字节
   pBuf[2] = (UCHAR)((MyCount>>8)&0xFF);           //次低字节
   pBuf[3] = (UCHAR)((MyCount>>16)&0xFF);          //次高字节
   pBuf[4] = (UCHAR)((MyCount>>24)&0xFF);          //最高字节
   //pBuf[5]~[7]设置为 0
   pBuf[5] = 0;
```

```
            pBuf[6] = 0;
            pBuf[7] = 0;
            T<<"修改输入报告完毕\n";
          }
        }
      break;
      default:
      break;
    }
  }
  return status;
}
```

至此，该过滤驱动就算改好了，不算太难吧？然后编译程序，就可以得到一个驱动程序文件 MyUsbLowerFilter.sys。下面介绍如何安装这个过滤驱动。

10.4 过滤驱动的安装

在使用向导创建驱动时，选择了创建安装用的 DLL 工程，但是 DLL 的调试不像普通应用程序调试那么方便，所以可以参考该安装用的 DLL 工程里的代码，自己写一个应用程序来安装过滤驱动。不过，向导产生的安装程序比较简单，只是简单地根据设备描述来判断需要安装过滤驱动的设备，并增加服务。实际上，为了准确地安装到某个设备上，应该通过 GUID 查找指定设备类下的硬件 ID 来决定是否需要安装过滤驱动。另外，由于该过滤驱动修改了报告描述符应用集合的用途，导致最终产生的设备不一样（原来为键盘设备，安装过滤驱动后变成了 HID 兼容设备），需要先将原来的驱动卸载掉，然后才能安装过滤驱动；否则系统会不知道这个情况，还是使用以前的驱动，也就是说，并不会出现新的硬件。如果这时强行将设备拔下，还会导致系统死机。安装过滤驱动后还需要重新启动设备，主机才会识别到新硬件。

就在当前 Workspace 下新增一个 FilterInstallerOfComputer00 的工程（选择为基于对话框的模板），放置两个按钮，一个为安装过滤驱动，另外一个为卸载过滤驱动。

安装过滤驱动的步骤如下：

（1）复制过滤驱动文件 MyUsbLowerFilter.sys 到当前系统的 Windows\System32\Drivers 目录下。可以调用 GetWindowsDirectory()函数获取操作系统下的 Windows 目录，然后生成目标路径，接着使用 CopyFile()函数复制文件。注意在 VC 环境下调试时，复制文件会提示不成功，这时可以手动复制一次文件或者直接运行一次程序以复制文件，文件只需要复制一次就可以了，除非驱动文件有改动。注意过滤驱动文件 MyUsbLowerFilter.sys 的版本，有发行版和调试版，我们分别将它们复制到 FilterInstallerOfComputer00 工程下的调试目录和发

行目录。当然，如果是在调试驱动，总这样复制也不方便，可以直接在驱动程序工程设置中指定输出目标文件的路径到 FilterInstallerOfComputer00 工程下的对应目录中。

(2) 卸载旧的键盘驱动。通过查找所有键盘设备，查看是否有指定硬件 ID 的设备，如果有，则卸载它。这里使用键盘的安装类 GUID 来查找设备，那么首先应该要找到键盘的安装类 GUID。去哪儿找呢？在注册表中可以找到。将带有 USB 键盘程序的 USB 学习板连上，然后展开注册表中的 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\HID(可按 F5 键刷新注册表)，找到下面的 Vid_8888&Pid_0002 并展开它，可以看到下面有很多子键，分别点击每个子键，然后查看右边的表项。检查名称为 Class 的数据是否为 Keyboard，如果是，就可以看到它的 ClassGUID 为{4D36E96B - E325 - 11CE - BFC1 - 08002BE10318}；然后再看 HardwareID，肯定有一个是与我们设备相匹配的，即 VID 为 8888，PID 为 0002，版本为 0100 的 HardwareID“HID\Vid_8888&Pid_0002&Rev_0100”。通过列举该 GUID 的所有已连接的设备，判断 HardwareID 是否为“HID\Vid_8888&Pid_0002&Rev_0100”，即可确认是否为指定的设备。使用 SetupDiGetClassDevs()函数可以获取指定类的设备集合信息，其中参数 GUID 可以通过最后的 Flags 参数指定为安装类 GUID 还是接口类 GUID，这里使用的是安装类 GUID，别搞错了。然后调用 SetupDiEnumDeviceInfo()函数列举所有该集合中的设备，并使用 SetupDiGetDeviceRegistryProperty()函数获取 HardwareID。获取到 HardwareID 后，判断是否为我们设备的 HardwareID“HID\Vid_8888&Pid_0002&Rev_0100”。如果是，就调用 SetupDiCallClassInstaller()函数删除这个设备。

(3) 添加服务。调用 OpenSCManager()函数打开 SCManager 数据库，然后调用 CreateService()函数增加 MyUsbLowerFilter 服务。

(4) 更新设备过滤驱动服务名称列表。我们的过滤驱动是安装在键盘设备下面那个 HID 设备上的，过滤驱动应该增加在这个设备上。在注册表中可以找到这个设备的信息，展开注册表中的“HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB”子键，在下面找到 Vid_8888&Pid_0002 的子键并展开它，可以找到一个“2008 - 07 - 12”的子键(它就是我们在固件程序中指定的设备序列号)，选中它。然后在右边就可以看到它的设备类为 HIDClass，安装 ClassGUID 为{745A17A0 - 74D3 - 11D0 - B6FE - 00A0C90F57DA}，HardwareID 为“USB\Vid_8888&Pid_0002&Rev_0100”。与步骤(2)中的查找方式类似，使用 GUID 和硬件 ID 来找到这个设备。找到后就可以将 MyUsbLowerFilter 增加到下层过滤驱动服务名称列表中(当然如果已经存在就不用再增加了，这个需要自己程序判断)。成功更新后可在右边看到增加了一个名称为 LowerFilters 的表项，数据为 MyUsbLowerFilter。这就是告诉系统该设备将使用服务名为 MyUsbLowerFilter 的下层过滤驱动器。如果没有看到，可以使用 F5 键刷新一下。设置服务名称列表的函数为 SetupDiSetDeviceRegistryProperty()。

(5) 重新启动设备。使用 SetupDiSetClassInstallParams 及 SetupDiCallClassInstaller()函数来停止和启动设备。安装过滤驱动后要能够自动发现新设备，必须要重新启动设备。

具体实现的代码较长，这里就不再给出了，读者可以参看光盘中 MyUsbLowerFilter 目录下的 FilterInstallerOfComputer00 工程，主要在 FilterInstallerOfComputer00Dlg.cpp 文件中。

10.5 过滤驱动的卸载

同过滤驱动的安装一样，卸载过滤驱动后导致了新设备的产生，所以需要先卸载掉原来的旧驱动（这次需要卸载的旧驱动就是 HID 兼容设备了）；接着再更新过滤驱动服务名称列表，最后删除服务，并重新启动设备。其详细步骤如下：

(1) 卸载旧的 HID 兼容设备的驱动。由于该过滤驱动安装之后，会变成一个 HID 兼容设备，卸载过滤驱动之前要先卸载这个 HID 兼容设备。跟安装过滤驱动的步骤(2)类似，要先找到该设备的 GUID 和硬件 ID。同样在 Enum\HID\Vid_8888&Pid_0002 下可以找到该 HID 兼容设备，设备类为 HIDClass，设备描述 DeviceDesc 为 HID-compliant device，ClassGUID 为 {745A17A0 - 74D3 - 11D0 - B6FE - 00A0C90F57DA}，硬件 ID 为“HID\Vid_8888&Pid_0002&Rev_0100”。用该 GUID 和硬件 ID 去搜索设备，如果找到，则卸载该设备。

(2) 更新设备过滤驱动服务名称列表。与安装过滤驱动时的步骤(4)刚好相反，这里是将 MyUsbLowerFilter 从过滤驱动服务名称列表中移除，告诉系统该设备不再使用该服务作为下层过滤驱动了。成功更新后，可以看到名称为 LowerFilters 的表项中不再有前面指定的服务了。

(3) 删除服务。调用 DeleteService()函数将原来新增的 MyUsbLowerFilter 服务删除。

(4) 重新启动设备。同安装过滤驱动时的步骤(5)。

图 10.5.1～10.5.3 分别为 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB\Vid_8888&Pid_0002\2008-07-12 子键右边表项安装过滤驱动之前、安装过滤驱动之后和卸载过滤驱动之后的变化。

名称	类型	数据
(默认)	REG_SZ	(数值未设置)
Capabilities	REG_DWORD	0x00000094 (148)
Class	REG_SZ	HIDClass
ClassGUID	REG_SZ	{745A17A0-74D3-11D0-B6FE-
CompatibleIDs	REG_MULTI_SZ	USB\Class_03&SubClass_01
ConfigFlags	REG_DWORD	0x00000000 (0)
DeviceDesc	REG_SZ	USB 人体学输入设备
Driver	REG_SZ	{745A17A0-74D3-11D0-B6FE-
HardwareID	REG_MULTI_SZ	USB\Vid_8888&Pid_0002&Re
LocationInformation	REG_SZ	USB Device
Mfg	REG_SZ	(标准系统设备)
ParentIdPrefix	REG_SZ	7&2824ea3b&3
Service	REG_SZ	HidUsb
UINumber	REG_DWORD	0x00000000 (0)

图 10.5.1 安装过滤驱动之前

从图 10.5.2 中可以看出，安装过滤驱动之后，新增了一个名称为 LowerFilters(下层过滤器)，数据为 MyUsbLowerFilter 的表项。而卸载过滤驱动之后，LowerFilters 表项中的数据被设置成了空，即将 MyUsbLowerFilter 删除了。如果原来就安装过其他下层过滤器，在这里也可以看到，卸载 MyUsbLowerFilter 并不会影响它们，这些都是在安装程序中实现的。

名称	类型	数据
(默认)	REG_SZ	(数值未设置)
Capabilities	REG_DWORD	0x00000094 (148)
Class	REG_SZ	HIDClass
ClassGUID	REG_SZ	{745A17A0-74D3-11D0-B6FE-
CompatibleIDs	REG_MULTI_SZ	USB\Class_03&SubClass_01&
ConfigFlags	REG_DWORD	0x00000000 (0)
DeviceDesc	REG_SZ	USB 人体学输入设备
Driver	REG_SZ	{745A17A0-74D3-11D0-B6FE-
HardwareID	REG_MULTI_SZ	USB\Vid_8888&Pid_0002&Rev
LocationInformation	REG_SZ	USB Device
LowerFilters	REG_MULTI_SZ	MyUsbLowerFilter
Mfg	REG_SZ	(标准系统设备)
ParentIdPrefix	REG_SZ	7&2824ea3b&3
Service	REG_SZ	HidUsb
UINumber	REG_DWORD	0x00000000 (0)

注意这里的改变

图 10.5.2　安装过滤驱动之后

名称	类型	数据
(默认)	REG_SZ	(数值未设置)
Capabilities	REG_DWORD	0x00000094 (148)
Class	REG_SZ	HIDClass
ClassGUID	REG_SZ	{745A17A0-74D3-11D0-B6FE-
CompatibleIDs	REG_MULTI_SZ	USB\Class_03&SubClass_01&
ConfigFlags	REG_DWORD	0x00000000 (0)
DeviceDesc	REG_SZ	USB 人体学输入设备
Driver	REG_SZ	{745A17A0-74D3-11D0-B6FE-
HardwareID	REG_MULTI_SZ	USB\Vid_8888&Pid_0002&Rev
LocationInformation	REG_SZ	USB Device
LowerFilters	REG_MULTI_SZ	
Mfg	REG_SZ	(标准系统设备)
ParentIdPrefix	REG_SZ	7&2824ea3b&3
Service	REG_SZ	HidUsb
UINumber	REG_DWORD	0x00000000 (0)

注意这里的改变

图 10.5.3　卸载过滤驱动之后

10.6　驱动程序测试

图 10.6.1 为安装过滤驱动时显示的信息，图 10.6.2 为卸载过滤驱动时显示的信息。注意被安装过滤驱动的设备是一个“USB 人体学输入设备”，它的硬件 ID 开头为 USB。而原来

的键盘是由该“USB 人体学输入设备”产生的，它的硬件 ID 以 HID 开头。安装过滤驱动之后的“HID 兼容设备”也是由这个“USB 人体学输入设备”产生的，硬件 ID 也是以 HID 开头。“USB 人体学输入设备”和“HID 兼容设备”的安装类 GUID 都为{745A17A0-74D3-11D0-B6FE-00A0C90F57DA}，通过硬件 ID 可以区分它们。而键盘设备的安装类 GUID 为{4D36E96B-E325-11CE-BFC1-08002BE10318}，可以与“HID 兼容设备”区分开来。

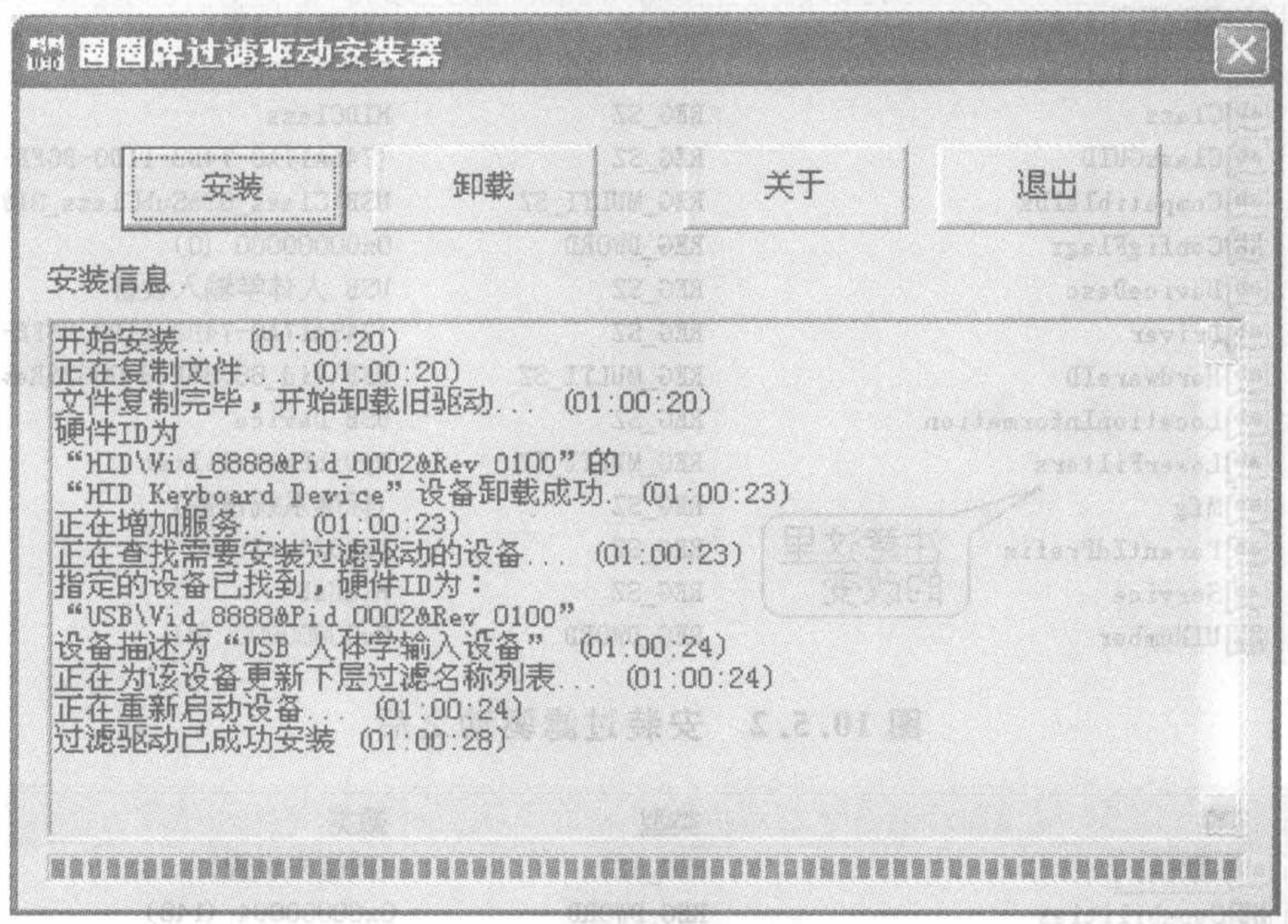


图 10.6.1 安装过滤驱动

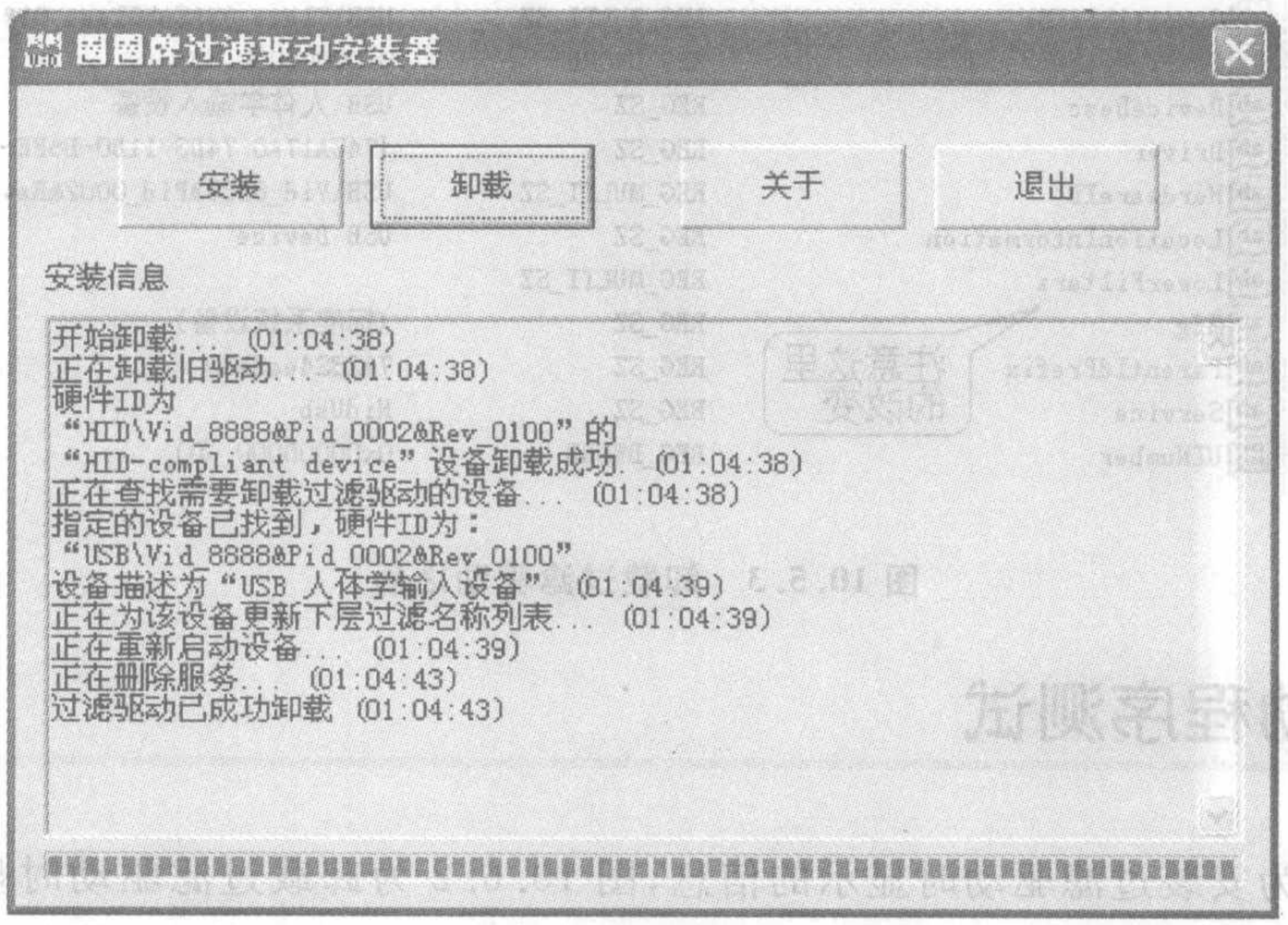


图 10.6.2 卸载过滤驱动

连接USB学习板，启动第5章中的自定义HID设备测试软件(MyUsbHidTestApp)，将PID修改为0002(记得一定要修改，否则会找不到设备)，单击“打开设备”按钮，提示设备已经找到，但是“读访问打开设备失败”。这就是因为该HID设备已经被系统独占了，所以无法访问。然后单击“关闭设备”按钮，再安装过滤驱动。等过滤驱动安装完毕之后，再单击“打开设备”按钮，这时设备就被成功打开了，并且应用程序能够控制板上的LED，也能够正常显示板上按键的情况了。

打开设备管理器，可以看到在安装过滤驱动之后，原来在键盘类下的HID Keyboard Device不见了，而在“USB人体学输入设备”下增加了一个新的HID-compliant device设备。

使用BUS Hound捕捉“USB人体学输入设备”以及“HID-compliant device”的数据，结果如图10.6.3所示。其中Device为27.1的为“USB人体学输入设备”，Device为29的是我们的测试软件操作的“HID-compliant device”。前两行数据是单击应用程序上LED1时产生的数据，这两行数据的顺序颠倒一下看起来更清楚些。第二行的数据就是应用程序发出给HID-compliant device的数据，第1字节00为报告ID，第二字节01表示LED1亮。这些数据由“HID-compliant device”提交给“USB人体学输入设备”之后，输出就变成一字节的01了，这正是我们键盘程序所需要接收的1字节数据。如果没有过滤驱动修改，“USB人体学输入设备”输出的数据应该是去掉报告ID之后剩下的8字节数据。而三四行是按下KEY8时捕捉到的数据，五、六行是松开KEY8时捕捉到的数据。第三行数据中第三字节为53，它正是键盘程序中KEY8返回的键码值，经过过滤驱动之后，将第四行数据中第二字节的最高位设置成了1，表示KEY8被按下了。同时，后面增加了计数器值01。第五行数据全为00，表示所有按键都已经释放，经过过滤驱动之后，第六行第一字节就为00，同时后面增加了计数器值02。总之，过滤驱动将键盘输入的数据格式翻译成了应用程序所需要的格式，将应用程序输出的数据格式翻译成了键盘所需要的输出数据格式。

Device	Phase	Data
27.1	DO	01
29	DO	00 01 00 00 00 00 00 00 00
27.1	DI	00 00 53 00 00 00 00 00
29	DI	00 80 01 00 00 00 00 00 00
27.1	DI	00 00 00 00 00 00 00 00
29	DI	00 00 02 00 00 00 00 00 00

图10.6.3 BUS Hound捕捉到的数据

如果安装的是调试版的驱动程序，那么就可以使用DriverMonitor来显示在驱动中输出的调试信息了，如图10.6.4所示。启动DriverMonitor，打开需要显示调试信息的驱动文件，这时就能在DriverMonitor的信息窗口中显示出调试信息。图10.6.4所显示的是安装过滤驱动、控制LED和按动开关时的部分信息。为了让读者方便阅读，圈圈将它们各个阶段划分开来，并在右边增加了注释。显示的第一阶段为捕捉到请求报告描述符的请求，这时需要设置完成函数。第二阶段就是报告描述符返回后，进入完成函数之后的处理，对返回的报告描述符进行了修改。第三、四阶段是检测到读取数据的请求，需要设置完成函数。第五阶段显示检测到一个输出数据的请求(由操作LED引起的)，这时只需要修改输出的数据长度，不用设置完

成函数。第六阶段显示有数据返回(按动了学习板上的按键),从而进入了设置的完成函数,在这里对返回数据进行了修改。当数据成功返回后,上层程序又一次请求数据输入(第七阶段)。

图 10.6.4 使用 DriverMonitor 显示驱动的调试信息

调试版的驱动通常只在调试阶段使用,实际使用的应该是发行版的驱动。调试版和发行版的驱动文件分别位于 driver\objchk\i386 目录和 driver\objfre\i386 目录下,要分别编译。安装时要将驱动文件和安装程序放在同一个目录下。

10.7 本章小结

本章通过一个 USB 下层过滤驱动的实例介绍了如何构造和安装一个过滤驱动程序,并在光盘中给出了完整的代码。该过滤驱动程序通过修改设备返回的报告描述符以及设备通信的数据,将原来的 USB 键盘设备“改装”成了 HID 兼容设备。读者可以发挥自己的想象,改出更好玩的东西来,例如,在键盘上增加鼠标功能;把键盘改成鼠标;把普通键盘改成多媒体键盘(可以控制音量、播放等);屏蔽某些键等,爱怎么玩就怎么玩,想怎么玩就怎么玩,总之过滤驱动是个好东西。

附 录

第 3 章实例的完整调试信息

```
*************************************************************
******          《圈圈教你玩 USB》之 USB 鼠标          ******
******                AT89S52 CPU                       ******
******            建立日期:Jul 10 2008                  ******
******            建立时间:00:31:10                     ******
******              作者:电脑圈圈                       ******
******              欢迎访问作者的                      ******
******      USB 专区:http://group.ednchina.com/93/      ******
******   BLOG1:http://www.ednchina.com/blog/computer00  ******
******      BLOG2:http://computer00.21ic.org            ******
******            请按 K1～K8 分别进行测试              ******
****** K1:光标左移  K2:光标右移  K3:光标上移  K4:光标下移 ******
****** K5:滚轮下滚  K6:滚轮上滚  K7:鼠标左键  K8:鼠标右键 ******
*************************************************************
Your D12 chip's ID is: 0x1012. ID is correct! Congratulations!
断开 USB 连接。
连接 USB。
USB 总线复位。
USB 总线挂起。
USB 总线挂起。
USB 总线复位。
USB 端点 0 输出中断。
读端点 0 缓冲区 8 字节。
0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
USB 标准输入请求:获取描述符——设备描述符。
写端点 0 缓冲区 16 字节。
```

0x12 0x01 0x10 0x01 0x00 0x00 0x00 0x10 0x88 0x88 0x01 0x00 0x00 0x01 0x01 0x02

USB 端点 0 输入中断。

写端点 0 缓冲区 2 字节。

0x03 0x01

USB 总线复位。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x00 0x05 0x1B 0x00 0x00 0x00 0x00 0x00

USB 标准输出请求:设置地址。地址为:0x1B

写端点 0 缓冲区 0 字节。

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x00 0x01 0x00 0x00 0x12 0x00

USB 标准输入请求:获取描述符——设备描述符。

写端点 0 缓冲区 16 字节。

0x12 0x01 0x10 0x01 0x00 0x00 0x00 0x10 0x88 0x88 0x01 0x00 0x00 0x01 0x01 0x02

USB 端点 0 输入中断。

写端点 0 缓冲区 2 字节。

0x03 0x01

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x00 0x02 0x00 0x00 0x09 0x00

USB 标准输入请求:获取描述符——配置描述符。

写端点 0 缓冲区 9 字节。

0x09 0x02 0x22 0x00 0x01 0x01 0x00 0x80 0x32

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x00 0x03 0x00 0x00 0xFF 0x00

USB 标准输入请求:获取描述符——字符串描述符(语言 ID)。

写端点 0 缓冲区 4 字节。

0x04 0x03 0x09 0x04

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x03 0x03 0x09 0x04 0xFF 0x00

USB 标准输入请求:获取描述符——字符串描述符(产品序列号)。

写端点 0 缓冲区 16 字节。

0x16 0x03 0x32 0x00 0x30 0x00 0x30 0x00 0x38 0x00 0x2D 0x00 0x30 0x00 0x37 0x00

USB 端点 0 输入中断。

写端点 0 缓冲区 6 字节。

0x2D 0x00 0x30 0x00 0x37 0x00

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x00 0x02 0x00 0x00 0xFF 0x00

USB 标准输入请求:获取描述符——配置描述符。

写端点 0 缓冲区 16 字节。

0x09 0x02 0x22 0x00 0x01 0x01 0x00 0x80 0x32 0x09 0x04 0x00 0x00 0x01 0x03 0x01

USB 端点 0 输入中断。

写端点 0 缓冲区 16 字节。

0x02 0x00 0x09 0x21 0x10 0x01 0x21 0x01 0x22 0x34 0x00 0x07 0x05 0x81 0x03 0x10

USB 端点 0 输入中断。

写端点 0 缓冲区 2 字节。

0x00 0x0A

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x00 0x03 0x00 0x00 0xFF 0x00

USB 标准输入请求:获取描述符——字符串描述符(语言 ID)。

写端点 0 缓冲区 4 字节。

0x04 0x03 0x09 0x04

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x02 0x03 0x09 0x04 0xFF 0x00

USB 标准输入请求:获取描述符——字符串描述符(产品描述)。

写端点 0 缓冲区 16 字节。

0x22 0x03 0x0A 0x30 0x08 0x57 0x08 0x57 0x59 0x65 0x60 0x4F 0xA9 0x73 0x55 0x00

USB 端点 0 输入中断。

写端点 0 缓冲区 16 字节。

0x53 0x00 0x42 0x00 0x0B 0x30 0x4B 0x4E 0x55 0x00 0x53 0x00 0x42 0x00 0x20 0x9F

USB 端点 0 输入中断。

写端点 0 缓冲区 2 字节。

0x07 0x68

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x00 0x03 0x00 0x00 0xFF 0x00

USB 标准输入请求:获取描述符——字符串描述符(语言 ID)。

写端点 0 缓冲区 4 字节。

0x04 0x03 0x09 0x04

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x02 0x03 0x09 0x04 0xFF 0x00

USB 标准输入请求:获取描述符——字符串描述符(产品描述)。

写端点 0 缓冲区 16 字节。

0x22 0x03 0x0A 0x30 0x08 0x57 0x08 0x57 0x59 0x65 0x60 0x4F 0xA9 0x73 0x55 0x00

USB 端点 0 输入中断。

写端点 0 缓冲区 16 字节。

0x53 0x00 0x42 0x00 0x0B 0x30 0x4B 0x4E 0x55 0x00 0x53 0x00 0x42 0x00 0x20 0x9F

USB 端点 0 输入中断。

写端点 0 缓冲区 2 字节。

0x07 0x68

USB 端点 0 输入中断。

USB 端点 0 输出中断。

读端点 0 缓冲区 8 字节。

0x80 0x06 0x00 0x01 0x00 0x00 0x12 0x00

USB 标准输入请求:获取描述符——设备描述符。

写端点0缓冲区16字节。

0x12 0x01 0x10 0x01 0x00 0x00 0x00 0x10 0x88 0x88 0x01 0x00 0x00 0x01 0x01 0x02

USB端点0输入中断。

写端点0缓冲区2字节。

0x03 0x01

USB端点0输入中断。

USB端点0输出中断。

读端点0缓冲区8字节。

0x80 0x06 0x00 0x02 0x00 0x00 0x09 0x00

USB标准输入请求:获取描述符——配置描述符。

写端点0缓冲区9字节。

0x09 0x02 0x22 0x00 0x01 0x01 0x00 0x80 0x32

USB端点0输入中断。

USB端点0输出中断。

读端点0缓冲区8字节。

0x80 0x06 0x00 0x02 0x00 0x00 0x22 0x00

USB标准输入请求:获取描述符——配置描述符。

写端点0缓冲区16字节。

0x09 0x02 0x22 0x00 0x01 0x01 0x00 0x80 0x32 0x09 0x04 0x00 0x00 0x01 0x03 0x01

USB端点0输入中断。

写端点0缓冲区16字节。

0x02 0x00 0x09 0x21 0x10 0x01 0x21 0x01 0x22 0x34 0x00 0x07 0x05 0x81 0x03 0x10

USB端点0输入中断。

写端点0缓冲区2字节。

0x00 0x0A

USB端点0输入中断。

USB端点0输出中断。

读端点0缓冲区8字节。

0x00 0x09 0x01 0x00 0x00 0x00 0x00 0x00

USB标准输出请求:设置配置。

写端点0缓冲区0字节。

USB端点0输入中断。

USB端点0输出中断。

读端点0缓冲区8字节。

0x21 0x0A 0x00 0x00 0x00 0x00 0x00 0x00

USB类输出请求:设置空闲。

写端点0缓冲区0字节。

USB端点0输入中断。

USB端点0输出中断。

读端点0缓冲区8字节。

0x81 0x06 0x00 0x22 0x00 0x00 0x74 0x00

USB标准输入请求:获取描述符——报告描述符。

写端点0缓冲区16字节。

0x05 0x01 0x09 0x02 0xA1 0x01 0x09 0x01 0xA1 0x00 0x05 0x09 0x19 0x01 0x29 0x03

USB端点0输入中断。

写端点0缓冲区16字节。

0x15 0x00 0x25 0x01 0x95 0x03 0x75 0x01 0x81 0x02 0x95 0x01 0x75 0x05 0x81 0x03

USB端点0输入中断。

写端点0缓冲区16字节。

0x05 0x01 0x09 0x30 0x09 0x31 0x09 0x38 0x15 0x81 0x25 0x7F 0x75 0x08 0x95 0x03

USB端点0输入中断。

写端点0缓冲区4字节。

0x81 0x06 0xC0 0xC0

USB端点0输入中断。

USB端点0输出中断。

读端点0缓冲区0字节。

/*******以下信息为分别点击KEY1～KEY8的信息*******/

写端点1缓冲区4字节。

0x00 0xFF 0x00 0x00

USB端点1输入中断。

写端点1缓冲区4字节。

0x00 0x01 0x00 0x00

USB端点1输入中断。

写端点1缓冲区4字节。

0x00 0x00 0xFF 0x00

USB端点1输入中断。

写端点1缓冲区4字节。

0x00 0x00 0x01 0x00

USB 端点 1 输入中断。
写端点 1 缓冲区 4 字节。
0x00 0x00 0x00 0xFF
USB 端点 1 输入中断。
写端点 1 缓冲区 4 字节。
0x00 0x00 0x00 0x01
USB 端点 1 输入中断。
写端点 1 缓冲区 4 字节。
0x01 0x00 0x00 0x00
USB 端点 1 输入中断。
写端点 1 缓冲区 4 字节。
0x00 0x00 0x00 0x00
USB 端点 1 输入中断。
写端点 1 缓冲区 4 字节。
0x02 0x00 0x00 0x00
USB 端点 1 输入中断。
写端点 1 缓冲区 4 字节。
0x00 0x00 0x00 0x00
USB 端点 1 输入中断。

参考文献

[1] USB Implementer's Forum. USB Specification Rev2. 0. http://www. usb. org,2000.

[2] USB Implementer's Forum. USB Specification Rev1. 1. http://www. usb. org,1998.

[3] USB Implementer's Forum. USB Device Class Definition for HID Rev1. 11. http://www. usb. org,2001.

[4] USB Implementer's Forum. USB HID Usage Tables Rev1. 12. http://www. usb. org,2005.

[5] USB Implementer's Forum. USB Device Class Definition for MIDI Devices Rev1. 0. http://www. usb. org, 1999.

[6] USB Implementer's Forum. USB Class Definitions for Communication Devices Rev1. 1. http://www. usb. org,1999.

[7] USB Implementer's Forum. USB Mass Storage Class Bulk-Only Transport Rev1. 0. http://www. usb. org,1999.

[8] USB Implementer's Forum. USB Mass Storage Class Specification Overview Rev1. 2. http://www. usb. org,2003.

[9] USB Implementer's Forum. USB Mass Storage Class UFI Command Specification Rev1. 0. http://www. usb. org,1993.

[10] USB Implementer's Forum. USB Language Identifiers Rev1. 0. http://www. usb. org,2000.

[11] Microsoft Corporation. FAT32 File System Specification Rev1. 03. http://www. microsoft. com,2000.

[12] Philips. PDIUSBD12 Datasheet Rev. 08. http://www. semiconductors. philips. com,2001.

[13] 马伟. 计算机 USB 系统原理及其主/从机设计[M]. 北京:北京航空航天大学出版社,2004.

[14] 武安和. Windows 2000/XP WDM 设备驱动程序开发[M]. 第 2 版. 北京:电子工业出版社,2005.

[15] 电脑圈圈的家当. USB 入门系列文章. http://computer00. 21ic. org,2007.

后　记

经过两个多月的苦战，终于完成了本书的初稿。圈圈想说，写书真的是件挺辛苦的事。就算拿本几百页的书，全部敲到计算机中也不容易，何况自己写几百页出来呢。不过看到网上很多网友对本书的期待，以及周围朋友的支持，给了圈圈很大动力。在这里圈圈要对他们说声谢谢。

这是圈圈第一次写书（应该叫处女作比较专业），所以没啥经验，不知道最后出来的效果咋样。圈圈从小就害怕写作文（读书时为了应付作文，经常找一些书来抄，或者找家长代劳），语文成绩也一直是刚刚及格的水平，所以写出来的文章干巴巴的，有点像记流水帐（很多语文老师对圈圈的作文如此评价）。为了让读者能够更容易理解和接受，圈圈尽量使用一些简单、通俗的句子，有很多地方还有重复的说明，以让读者能够得到确切的答案。如果你觉得本书说得太简单，或者有些重复，那很正常，因为这本书主要面向的是初学者。

限于圈圈的水平，同时为了减少读者的负担，后记也就只能凑出这么几百个字了。如果你对本书有什么好的建议，请不要吝啬，可以给圈圈发邮件或者留言；如果你觉得本书写得不错，请多多宣传一下；如果你觉得本书写得不好，也可以给圈圈扔砖头，因为鲜花与砖头永远是共同存在的；如果你在找书垫桌子脚时，请不要用本书，如果真的要用，也请麻烦顺手把封面朝上……

最后，感谢广大网友、读者以及家人和朋友支持，感谢北航出版社，感谢胡编，感谢 21IC，感谢 CEPARK，感谢 EDN，感谢 OURAVR，感谢圈圈，感谢 CCTV……

电脑圈圈

2009 年 1 月

于广州